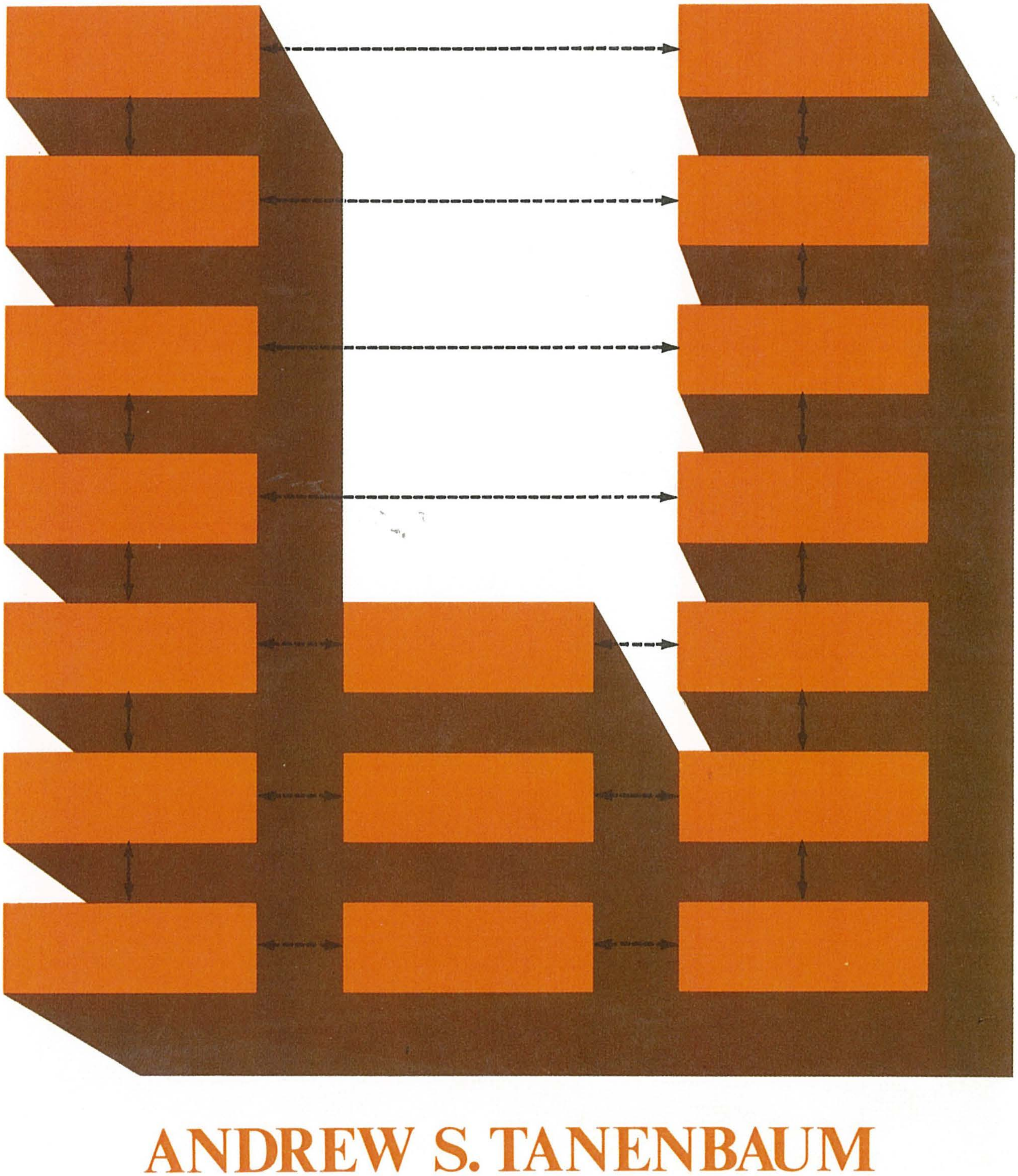
SECOND EDITION

COMPUTER NETWORKS



ANDREW S. TANENBAUM

INTEL Ex.1067.001

COMPUTER NETWORKS

SECOND EDITION

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE HALL
ENGLEWOOD CLIFFS, NEW JERSEY 07632

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.
Computer networks / Andrew S. Tanenbaum.—2nd ed.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-13-162959-X
1. Computer networks. I. Title.
TK5105.5.T36 1988
004.6—dc19 88-16937
CIP

Editorial/production supervision: **Lisa Schulz Garboski**
Manufacturing buyer: **Mary Ann Gloriande**

Prentice Hall Software Series
Brian W. Kernighan, Advisor

Printed in the United States of America

10 9

ISBN 0-13-162959-X

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*

To Suzanne, Barbara, Marvin, Muis, and the memory of Sweetie π

CONTENTS

8 THE PRESENTATION LAYER 471

9 THE APPLICATION LAYER 528

PREFACE

Computer networking has changed enormously over the past decade, and this book has changed along with it. Ten years ago, computer networks were exotic research tools used only by few specialists. Today, computers ranging from personal computers to supercomputers are more likely to be part of a network than not. Most organizations that use computers either already have, or are soon planning to install, one or more local area networks. Worldwide electronic mail is a daily reality for millions of people. In short, networks have evolved from an academic curiosity to an essential tool for users in business, government, and universities.

Only a few short years ago, the design of a computer network was something of a black art. Every computer manufacturer had its own network architecture, no two of which were compatible. All that has changed now. Virtually the entire computer industry has now agreed to a series of International Standards for describing network architectures. These standards are known as the OSI Reference Model. In the near future, almost all other network architectures will disappear, and computers from one vendor will be able to communicate effortlessly with computers from another vendor, thus stimulating network usage even more.

This book uses the OSI Reference Model as a framework. The model is based on a principle first enunciated by Julius Caesar: Divide and Conquer. The idea is to design networks as a sequence of layers, one based upon the previous one. By reducing the study of the whole to the study of its parts, the entire process becomes more manageable.

Because networking has changed so much since the first edition, this second edition is almost a completely new book. Over half of the material is completely new, and much of the remainder has also been heavily updated and improved. Whole new areas are now discussed in detail, including ISDN, LANs, fiber optics, and bridges. The treatment of the upper OSI layers has been completely rewritten and expanded by over a hundred pages to include important topics such as the OSI transport protocols, ASN.1, FTAM, and VTP. New networks, including MAP, TOP, and USENET are now examined in some detail and network software is also given more coverage.

The idea of organizing the book around the seven OSI layers has proved very successful and has been retained in this second edition. Chapter 1 provides an introduction to the subject of computer networks in general and layered protocols in particular. The next five chapters deal with the lower layers, physical through transport, which collectively are the providers of the transport service. Chapter 2 covers physical media, analog and digital transmission, the telephone system and ISDN. Chapter 3 looks at the MAC sublayer and local area networks, including IEEE standard 802. Chapter 4 is about the data link layer and its protocols: algorithms for reliably transmitting data over unreliable lines. Chapter 5 treats the network layer, especially routing, congestion control, and internetworking. Finally, chapter 6 studies the transport layer, in particular, connection management and end-to-end protocols.

The last three chapters deal with the upper layers, the users of the transport service. Chapter 7 is about the session layer, which is concerned with providing reliable service, even in the face of unreliable hardware. Chapter 8 covers the presentation layer, including the OSI abstract syntax notation, data compression, and cryptography. Chapter 9 provides an introduction to some application layer issues, including file transfer, electronic mail, virtual terminals, remote job entry, and directory services. Chapter 10 contains a reading list and bibliography.

Queueing theory is a basic mathematical tool that is used to analyze computer networks, so an appendix is provided on it for the benefit of readers not familiar with this subject.

The book can be used as a text for undergraduates or beginning graduate students in computer science, electrical engineering, and related disciplines. The only prerequisites are a general familiarity with computer systems and programming, although a little knowledge of elementary calculus and probability theory is useful in a few places, but is not essential. Some of the examples are given in Pascal, so some knowledge of this, or a similar programming language is a plus. Since the amount of material in the book may be too much for a one semester course, depending on the level of the students, I have made a serious attempt to make each chapter relatively independent of the other ones. In this way an instructor could choose to emphasize, for example, data communication and the lower layers, or alternatively, software and the upper layers.

The book can also be used by computer professionals who are interested in networking. For this reason, I have attempted to limit the amount of mathematics used, and have included numerous practical examples instead of providing many pages of abstract derivations. Even programmers or technical managers who are not network specialists should be able to follow a considerable amount of the book.

Many people have helped me during the preparation of the second edition. I would especially like to thank Imrich Chlamtac Bdale Garbee, John Henshall, Brian Kernighan, John Limb, Chris Makemson, Daniel Pitt, Sandy Shaw, Jennifer Steiner and my editor, John Wait. I would also like to thank my students for helping to debug the text. Special thanks go to Jeroen Belien, Berend Jan Beugel, Remco Feenstra, Anneth de Gee, Cornelis Kroon, Roemer Lievaart, Maarten Litmaath, Paul Polderman, Rob van Swinderen, Luuk Uljee, and Felix Yap.

Last but not least, I would like to thank Suzanne, Barbara, Marvin, and Muis. Suzanne for still being so understanding after all these years; Barbara and Marvin for using their computer instead of mine, thus making this book possible; and and Muis for being as quiet as a mouse while I was writing.

ANDREW S. TANENBAUM

1

INTRODUCTION

Each of the past three centuries has been dominated by a single technology. The eighteenth century was the time of the great mechanical systems accompanying the Industrial Revolution. The nineteenth century was the age of the steam engine. During the twentieth 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.

As we move toward the final years of this century, 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 rapidly becoming obsolete. This model has not one, but at least two flaws: the concept of a single large computer doing all the work, and the idea of users bringing work to the computer instead of bringing the computer to the users.

The old model of a single computer serving all of the organization's computational needs, is rapidly being 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 analysis 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; lasers, 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 card readers, 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, a user 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 user's knowledge.

In effect, a distributed system is a special case of a network, one whose 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.

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

Nevertheless, there is a lot of 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. Although this book primarily focuses on networks, many of the topics are also important in distributed systems. For more information about distributed systems, see Crichlow (1988), Sloman and Kramer (1987), and Tanenbaum and van Renesse (1985).

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. Network Goals

Many organizations already 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, data, and equipment 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. Load sharing is another aspect of resource sharing. 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, and many other applications, the ability to continue operating in the face of hardware problems is of great importance.

Another goal is **saving money**. Small computers have a much better price/performance ratio than large ones. Mainframes are roughly a factor of ten faster than the fastest single chip microprocessors, but they cost a thousand times more. This imbalance has caused many systems designers to build systems consisting of powerful personal computers, one per user, with data kept on one or more shared **file server** machines.

This goal leads to networks with many computers located in the same building.

Such a network is called a **LAN** (**local area network**) to contrast it with the far-flung **WAN** (**wide area network**), which is also called a **long haul network**.

A closely related point is the ability to increase system performance gradually as the workload grows just by adding more processors. With central mainframes, when the system is full, it must be replaced by a larger one, usually at great expense and with even greater disruption to the users.

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 people. Using a network, it is easy for two or more people who live far apart to write a report together. When one author makes a change to the document, which is kept online, 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 may prove more important than technical goals such as improved reliability.

In Fig. 1-1 we give a classification of multiple processor systems arranged by 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 **multiprocessors**, systems that communicate via shared memory. Beyond the multiprocessors are the true networks, computers that communicate by exchanging messages. Finally, the connection of two or more networks is called **internetworking**.

Interprocessor distance	Processors located in same	Example
0.1 m	Circuit board	Data flow machine
1 m	System	Multiprocessor
10 m	Room	Local network
100 m	Building	
1 km	Campus	
10 km	City	Long haul network
100 km	Country	
1000 km	Continent	Interconnection of long haul networks
10,000 km	Planet	

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

1.1.2. Applications of Networks

Replacing a single mainframe by workstations on a LAN does not make many new applications possible, although it may improve the reliability and performance. In contrast, the availability of a (public) WAN makes many new applications feasible. Some of these new applications may have important effects on society as a whole. To give an idea about some important uses of computer networks, we will now briefly look at just three examples: access to remote programs, access to remote databases, and value-added communication facilities.

A company that has produced a model simulating the world economy may allow its clients to log in over the network and run the program to see how various projected inflation rates, interest rates, and currency fluctuations might affect their businesses. This approach is often preferable to selling the program outright, especially if the model is constantly being adjusted or requires an extremely large mainframe computer to run.

Another major area of network use is access to remote databases. It may soon be easy for the average person sitting at home to make reservations for airplanes, trains, buses, boats, hotels, restaurants, theaters, and so on, anywhere in the world with instant confirmation. Home banking and the automated newspaper also fall in this category. Present newspapers offer a little bit of everything, but electronic ones can be easily tailored to each reader's personal taste, for example, everything about computers, the major stories about politics and epidemics, but no football, thank you.

The next step beyond automated newspapers (plus magazines and scientific journals) is the fully automated library. Depending on the cost, size, and weight of the terminal, the printed word may become obsolete. Skeptics should take note of the effect the printing press had on the medieval illuminated manuscript.

All these applications use networking for economic reasons: calling up a distant computer via a network is cheaper than calling it directly. The lower rate is possible because a normal telephone call ties up an expensive, dedicated circuit for the duration of the call, whereas access via a network ties up long-distance lines only while data are actually being transmitted.

A third category of potential widespread network use is as a communication medium. Computer scientists already take it for granted that they can send electronic mail from their terminals to their colleagues anywhere in the world. In the future, it will be possible for everyone, not just people in the computer business, to send and receive electronic mail. Furthermore, this mail will also be able to contain digitized voice, still pictures and possibly even moving television and video images. One can easily imagine children in different countries trying to learn each other's languages by drawing a picture of a child on a shared screen and labeling it girl, jeune fille, or meisje.

Electronic bulletin board systems already exist, but these tend to be used by computer experts, are oriented towards technical topics, and are often limited in geographic scope. Future systems will be national or international, be used by millions of nontechnical people, and cover a much broader range of subjects. Using a bulletin board may be as common as reading a magazine.

It is sometimes said that there is a race going on between transportation and communication, and whichever one wins will make the other unnecessary. Using a computer network as a sophisticated communication system may reduce the amount of traveling done, thus saving energy. Home work may become popular, especially for part-time workers with young children. The office and school as we now know them may disappear. Stores may be replaced by electronic mail order catalogs. Cities may disperse, since high-quality communication facilities tend to reduce the need for physical proximity. The information revolution may change society as much as the Industrial Revolution did.

1.2. NETWORK STRUCTURE

It is now time to turn our attention from the social implications of networking to the technical issues involved in network design. In any network there exists a collection of machines intended for running user (i.e., application) programs. We will follow the terminology of one of the first major networks, the ARPANET, and call these machines **hosts**. The term **end system** is sometimes also used in the literature. The hosts are connected by the **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. Again following the original ARPANET terminology, we will call the switching elements **IMPs** (**Interface Message Processors**) throughout the book, although the terms **packet switch node**, **intermediate system**, and **data switching exchange** are also commonly used. Unfortunately, there is no consensus on terminology here; every writer on the subject seems to use a different name. The term "IMP" is probably as good as any. In this model, shown in Fig. 1-2, each host is connected to one (or occasionally several) IMPs. All traffic to or from the host goes via its IMP.

Broadly speaking, there are two types of designs for the communication subnet:

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

In the first one, the network contains numerous cables or leased telephone lines, each one connecting a pair of IMPs. If two IMPs that do not share a cable nevertheless wish to communicate, they must do this indirectly, via other IMPs.

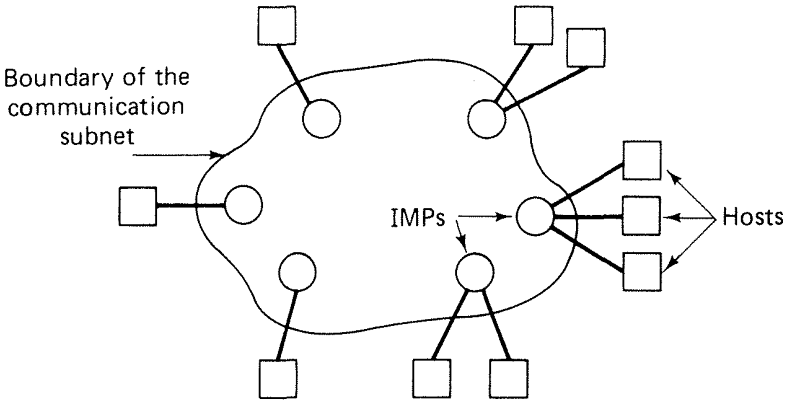


Fig. 1-2. Relation between hosts and the subnet.

When a message (in the context of the subnet often called a **packet**), is sent from one IMP to another via one or more intermediate IMPs, the packet is received at each intermediate IMP 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 have store-and-forward subnets.

When a point-to-point subnet is used, an important design issue is what the IMP interconnection topology should look like. Figure 1-3 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.

The second kind of communication architecture uses broadcasting. Most local area networks and a small number of wide area networks are of this type. In a local area network, the IMP is reduced to a single chip embedded inside the host, so there is always one host per IMP, whereas in a wide area network there may be many hosts per IMP.

Broadcast systems have a single communication channel that is shared by all the machines on the network. Packets 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 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.

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. Some broadcast systems also support transmission to a subset of the machines, something known as **multicasting**. A common scheme is to have all

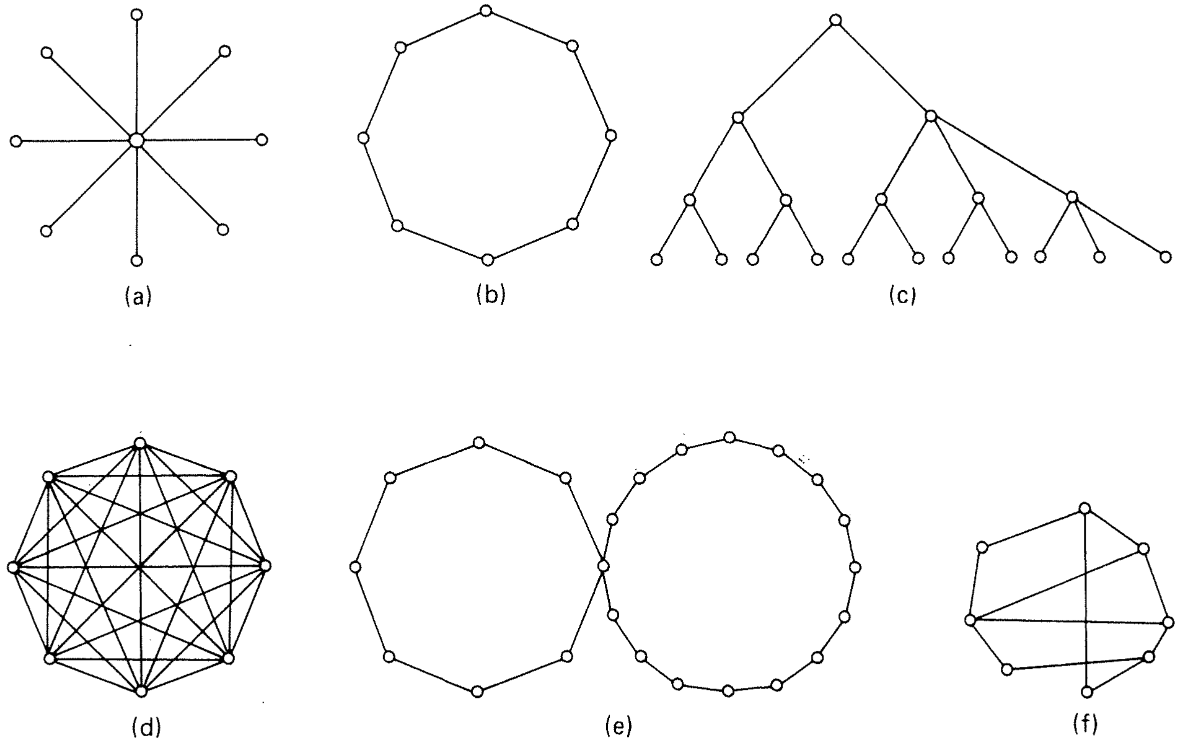


Fig. 1-3. Some possible topologies for a point-to-point subnet. (a) Star. (b) Ring. (c) Tree. (d) Complete. (e) Intersecting rings. (f) Irregular.

addresses with the high-order bit set to 1 be reserved for multicasting. The remaining $n - 1$ addresses bits form a bit map corresponding to $n - 1$ groups. Each machine can "subscribe" to any or all of the $n - 1$ groups. If a packet with, say, bits x, y, and z set to 1 is transmitted, it is accepted by all machines subscribing to one or more of those three groups.

Figure 1-4 shows some of the possibilities for broadcast subnets. In a bus or 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.

A second possibility is a satellite or ground radio system. Each IMP has an antenna through which it can send and receive. All IMPs can hear the output *from* the satellite, and in some cases they can also hear the upwards transmissions of their fellow IMPs *to* the satellite as well.

A third 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.

Broadcast subnets can be further divided into static and dynamic, depending on

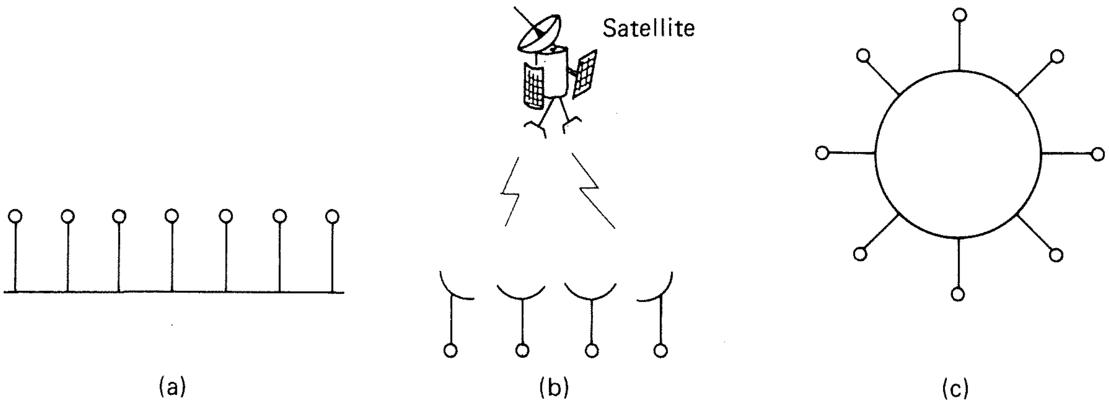


Fig. 1-4. Communication subnets using broadcasting. (a) Bus. (b) Satellite or radio. (c) Ring.

how the channel is allocated. A typical static allocation would be to divide up time into discrete intervals, and run a round robin, 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 some 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.

1.3. NETWORK ARCHITECTURES

Modern computer networks are designed in a highly structured way. In the following sections we examine the structuring technique in some detail.

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 its predecessor. 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**, as illustrated in Fig. 1-5 for a seven-layer network. The entities comprising the corresponding layers on different machines are called **peer processes**. In other words, it is the peer processes that communicate using the protocol.

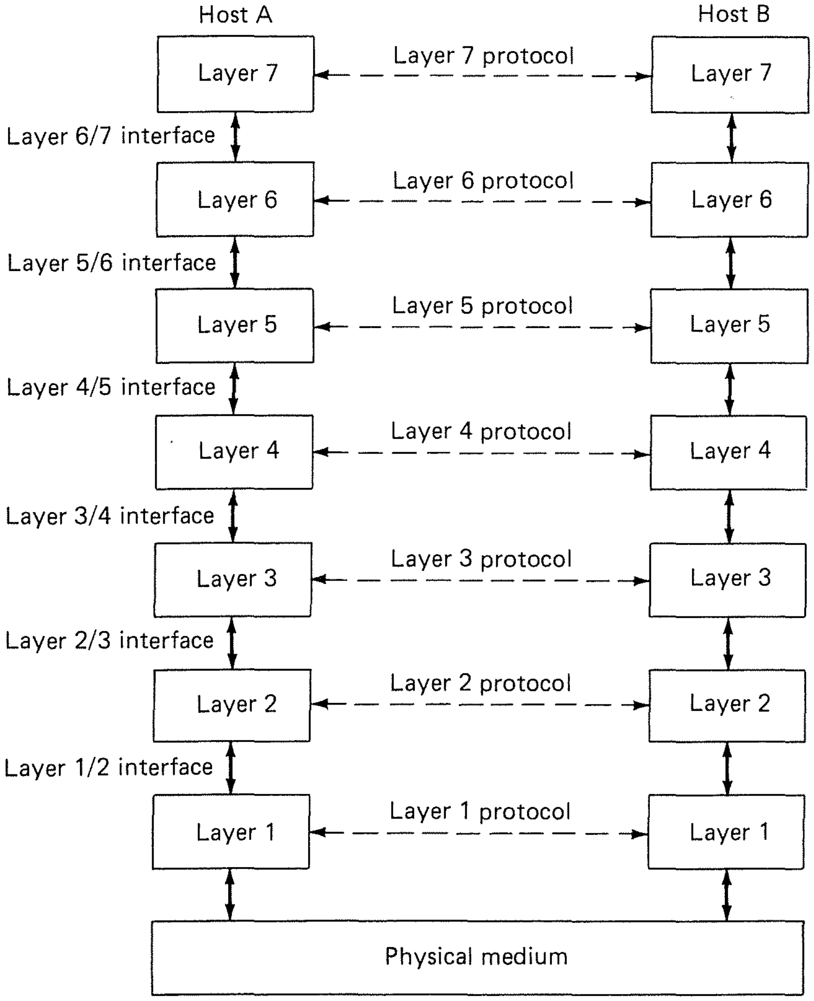


Fig. 1-5. Layers, protocols, and interfaces.

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. 1-5, 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 offer exactly the same set of services to its upstairs neighbor as the old implementation did.

The set of layers and protocols is called the **network architecture**. The specification of the 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. The subjects of network architectures and protocols 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 in Kenya and one in Indonesia, who want to communicate. Since they have no common language, they each engage a translator (peer processes at layer 2), each of whom in turn contacts an engineer (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 Swahili) across the 2/3 interface, to his translator, who might render it as "I like rabbits" or "J'aime des lapins" or "Ik hou van konijnen," depending on the layer 2 protocol.

The translator then gives the message to his engineer for transmission, by telegram, telephone, computer network, or some other means, depending on what the two engineers have agreed on in advance (the layer 1 protocol). When the message arrives, it is translated into Indonesian 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 French to Dutch at will, provided that they both agree, and neither changes his interface with either layer 1 or layer 3.

Now consider a more technical example: how to provide communication to the top layer of the seven-layer network in Fig. 1-6. A message, *m*, is produced by a process running in layer 7. The message is passed from layer 7 to layer 6 according to the definition of the layer 6/7 interface. In this example, layer 6 transforms the message in certain ways (e.g., text compression), and then passes the new message, *M*, to layer 5 across the layer 5/6 interface. Layer 5, in the example, does not modify the message but simply regulates the direction of flow (i.e., prevents an incoming message from being handed to layer 6 while layer 6 is busy handing a series of outgoing messages to layer 5).

In many networks, there is no limit to the size of messages accepted by layer 4,

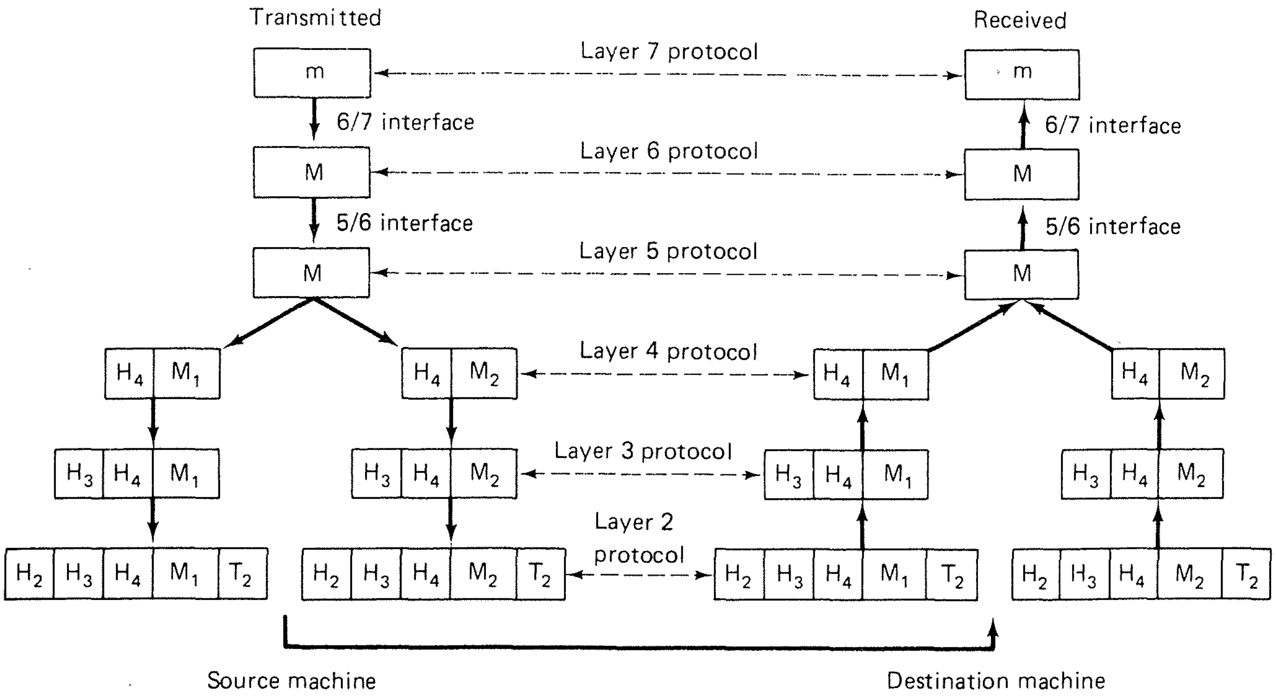


Fig. 1-6. Example information flow supporting virtual communication in layer 7.

but there is a limit imposed by layer 3. Consequently, layer 4 must break up the incoming messages into smaller units, prepending a **header** to each unit. The header includes control information, such as sequence numbers, to allow layer 4 on the destination machine to get the pieces back together in the right order if the lower layers do not maintain sequence. In many layers, headers also contain contain sizes, times and other control fields.

Layer 3 decides which of the outgoing lines to use, attaches its own headers, and passes the data 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. 1-6 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 *SendToOtherSide* and a procedure *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. Without this abstraction technique, it would be difficult, if not impossible, to partition the design of the complete network, an unmanageable problem, into several smaller, manageable, design problems, namely the design of the individual layers.

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 must have a mechanism for connection establishment. 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 establish a connection. As a consequence of having multiple destinations, some form of addressing is needed in order to specify a specific destination.

Closely related to the mechanism for establishing connections across the network is the mechanism for terminating them once they are no longer needed. As we will see in Chapter 6, this seemingly trivial point can actually be quite subtle.

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. All of them involve some kind of feedback from the receiver to the sender, either directly or indirectly, about the receiver's current situation.

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 current traffic.

1.4. THE OSI REFERENCE MODEL

Now that we have discussed layered networks in the abstract, it is time to look at the set of layers that will be used throughout this book. The model is shown in Fig. 1-7. This model is based on a proposal developed by the International Standards Organization (ISO) as a first step toward international standardization of the various protocols (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.

In Sections 1.5.1 through 1.5.7 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 strictly speaking part of the model. Each one has been published as a separate international standard.

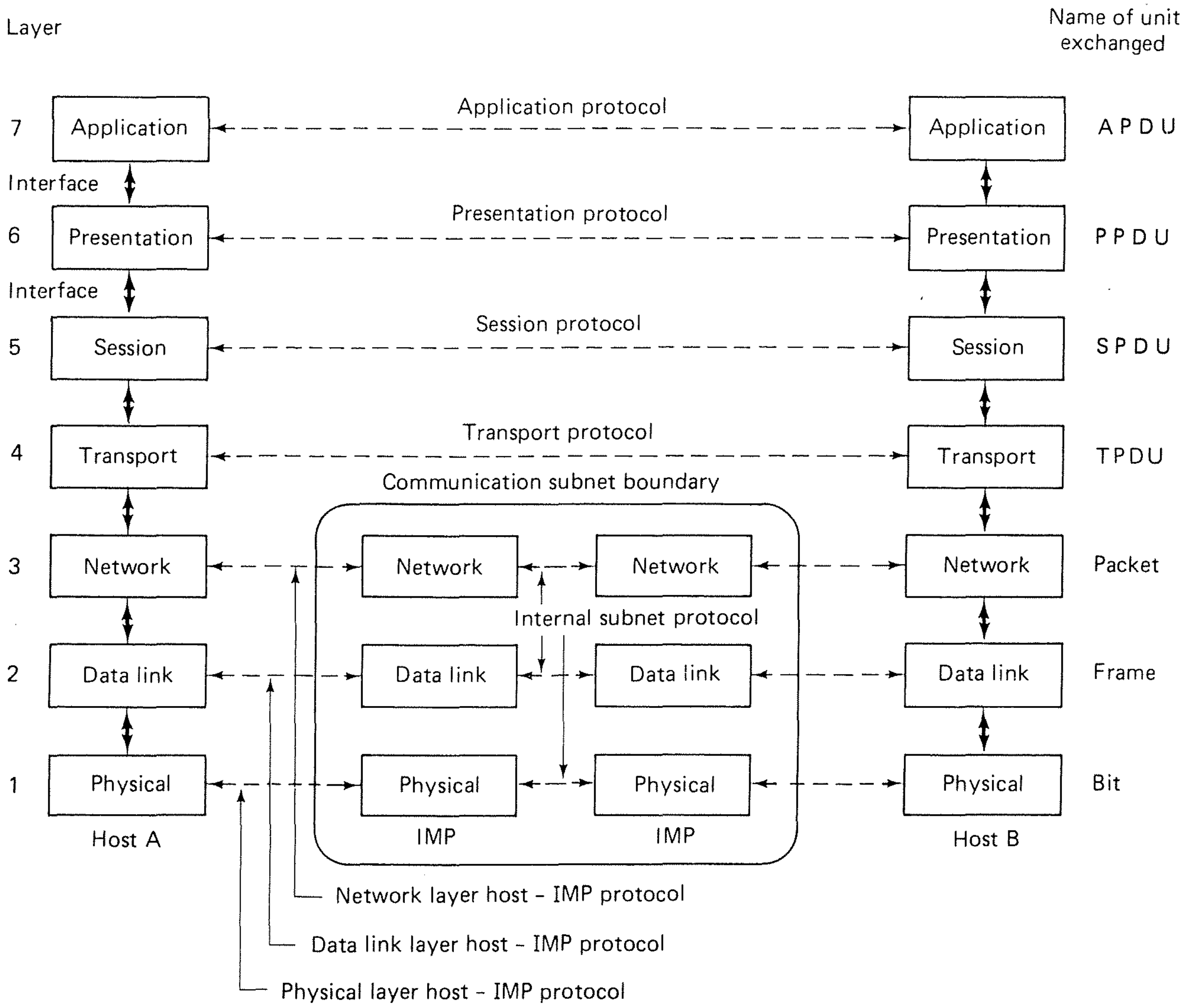


Fig. 1-7. The network architecture used in this book. It is based on the OSI model.

1.4.1. 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. Physical layer design can be properly considered to be within the domain of the electrical engineer.

1.4.2. 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 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 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 avoid confusion.

A noise burst on the line can destroy a frame completely. In this case, the data link layer software on the source machine must retransmit the frame. However, multiple transmissions of the same frame introduce the possibility of duplicate frames. A duplicate frame could be sent, for example, if the acknowledgement frame from the receiver back to the sender was destroyed. 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, for convenience.

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.

1.4.3. 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 could be based on static tables that are "wired into" the network and rarely changed. They could also be determined at the start of each conversation, for example a terminal session. Finally, they could 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 heterogenous networks to be interconnected.

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

1.4.4. 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 session layer 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 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 source-to-destination or **end-to-end** layer. 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 IMPs. The difference between layers 1 through 3, which are chained, and layers 4 through 7, which are end-to-end, is illustrated in Fig. 1-7.

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. 1-6) is one place this information could 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. Flow control between hosts is distinct from flow control between IMPs, although we will later see that similar principles apply to both.

1.4.5. 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 some enhanced services useful in a some applications. A session might be used to allow a user to log into a remote time-sharing 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 two-hour file transfer between two machines on a network 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 when the network next crashed. 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 after the last checkpoint have to be repeated.

1.4.6. 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 EBCDIC), 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 job of managing these abstract data structures and converting from the representation used inside the computer to the network standard representation is handled by the presentation layer.

The presentation layer is also concerned with other aspects of information representation. For example, data compression can be used here to reduce the number of bits that have to be transmitted and cryptography is frequently required for privacy and authentication.

1.4.7. 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.

1.4.8. Data Transmission in the OSI Model

Figure 1-8 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.

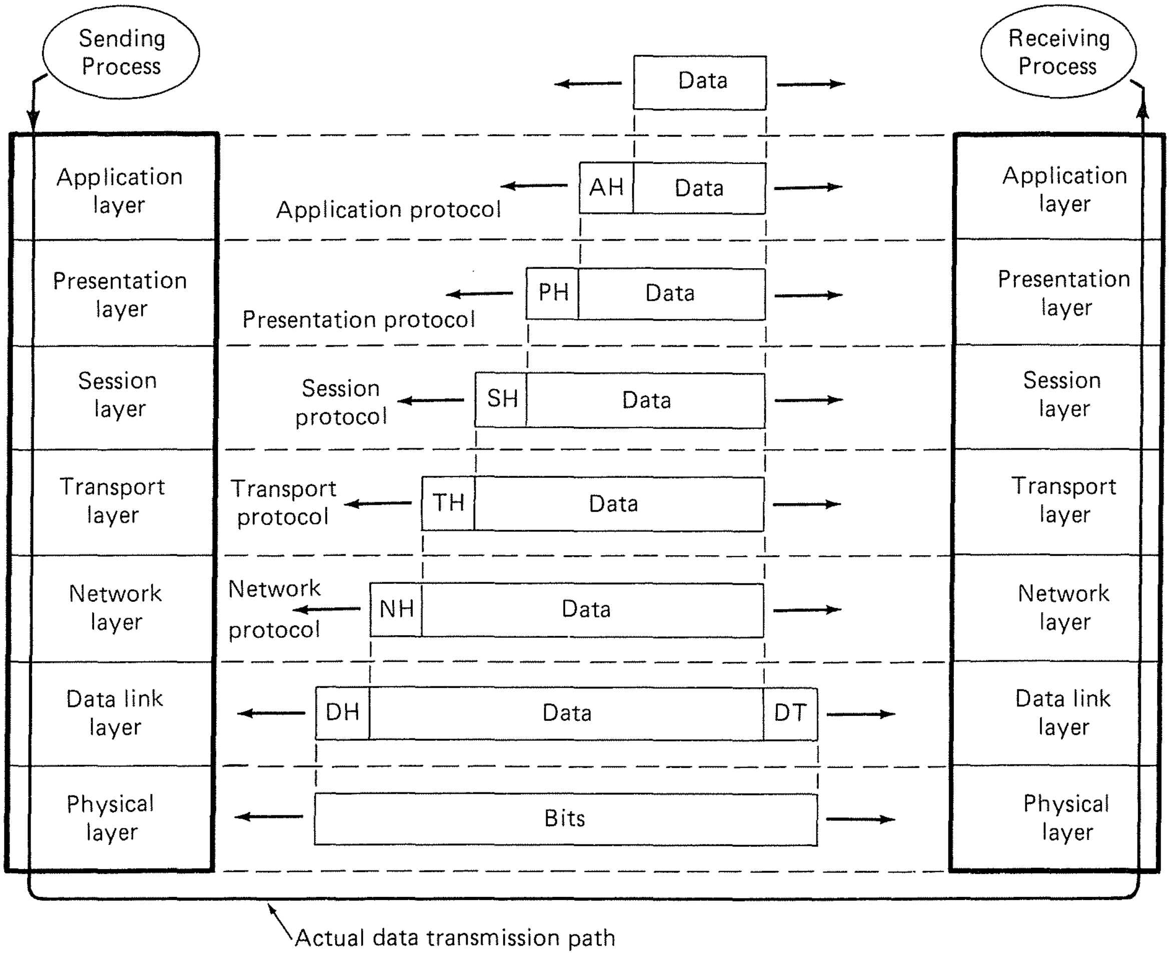


Fig. 1-8. An example of how the OSI model is used. Some of the headers may be null. (Source: H.C. Folts. Used with permission.)

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. Nor should it be aware.

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. 1-8, each layer is programmed as though it were really 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 an Uighur-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.

1.5. SERVICES

The real function of each layer in the OSI model 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 of the OSI terminology.

1.5.1. OSI Terminology

The active elements in each layer are 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 layer 7 entities are called **application entities**; the layer 6 entities are called **presentation entities**, and so on.

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 **SAPs**. (**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 his SAP address. Similarly, in the postal system, the SAP addresses are street addresses and post office boxes. To send a letter, you must know the addressee's SAP address. In Berkeley UNIX†, the SAPs are the sockets and the SAP addresses are the socket numbers. The SAP concept is discussed in detail by Tomas et al. (1987).

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. 1-9. 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

† UNIX is a registered trademark of AT&T Bell Laboratories.

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.

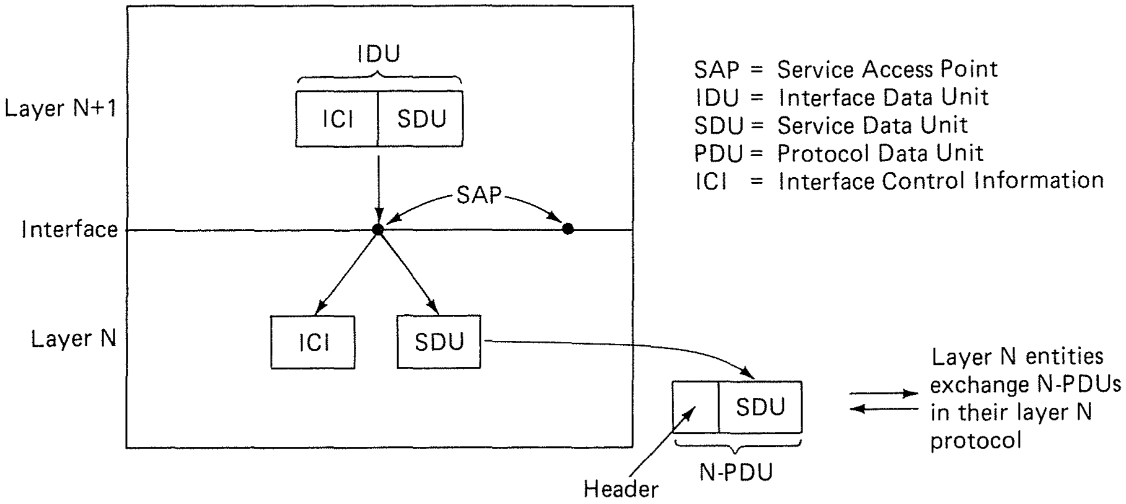

Layer N+1
Interface
Layer N
IDU
ICI
SDU
SAP
ICI
SDU
SDU
Header
N-PDU
SAP = Service Access Point
IDU = Interface Data Unit
SDU = Service Data Unit
PDU = Protocol Data Unit
ICI = Interface Control Information
Layer N entities exchange N-PDUs in their layer N protocol

Fig. 1-9. Relation between layers at an interface.

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 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. The transport, session and application PDUs are often referred to as TPDUs, SPDUs, and APDUs, respectively. No one talks much about the other PDUs.

This language is often known as **internationalbureaucratspeak**. We will avoid it where possible in favor of the more familiar nomenclature actually used by working computer scientists and engineers.

1.5.2. 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. Additional material can be found in (Bucciarelli and Caneschi, 1985; and Chapin, 1983).

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 terminates the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects 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 1K messages are sent, they arrive as two distinct 1K messages, never as one 2K message. 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 2K message, two 1K messages, or 2048 one-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 time-sharing 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.

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.

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. Figure 1-10 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. 1-10. Six different types of service.

1.5.3. 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. In the OSI model, the service primitives can be divided into four classes as shown in Fig. 1-11.

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	An entity is to be informed about its request

Fig. 1-11. Four classes of service primitives

The first class of primitive is the *request* primitive. It is used to get work done, for example, to establish a connection or to send data. When the work has been performed, the peer entity is signaled by an *indication* primitive. For example, after a *CONNECT.request* (in OSI notation), the entity being addressed gets a *CONNECT.indication* announcing that someone 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, the OSI model carefully avoids 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, the official terms are "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. *CONNECTION.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 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 1-12 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.

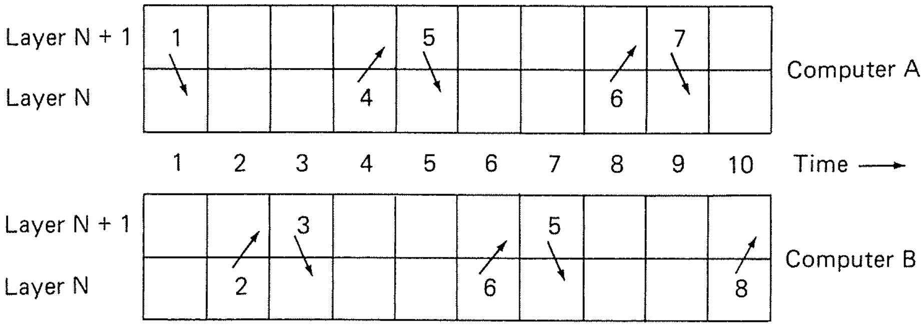


Fig. 1-12. How a computer would invite its Aunt Millie to tea. The numbers near the tail end of each arrow refer to the eight service primitives discussed in this section.

1.5.4. 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 protocol 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. 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 of the pre-OSI 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. It is now universally accepted that such a design is a blunder of major proportions.

1.6. NETWORK STANDARDIZATION

In the early days of networking, each computer manufacturer had its own network protocols. IBM had more than a dozen. The result was that users who had computers from several vendors could not connect them together into a single network. This chaos led many users to demand standardization.

Not only do standards allow different computers to communicate, but they also increase the market for products adhering to the standard, which leads to mass production, economies of scale in manufacturing, VLSI implementations, and other benefits that decrease price and further increase acceptance. In this section we will take a quick look at the important, but little-known, world of international standardization.

Standards fall into two categories: de facto and de jure. **De facto** (Latin for "from the fact") standards are those that have just happened, without any formal plan. The IBM PC and its successors are de facto standards for small office computers because dozens of manufacturers have chosen to copy IBM's machines very closely. UNIX is the de facto standard for operating systems in university computer science departments.

De jure (Latin for "by law") standards, in contrast, are formal, legal standards adopted by some authorized standardization body. International standardization authorities are generally divided into two classes: those established by treaty among national governments, and voluntary, nontreaty organizations. In the area of computer network standards, there are two principal organizations, one of each type. Both standards organizations are important and will be discussed below.

1.6.1. Who's Who in the Telecommunication World

The legal status of the world's telephone companies varies considerably from country to country. At one extreme is the United States, which has 1600 separate, privately owned telephone companies. Before it was broken up in 1984, AT&T, at that time the world's largest corporation, completely dominated the scene. It provided telephone service to about 80 percent of America's telephones, spread throughout half of its geographical area, with all the other companies combined servicing the remaining (mostly rural) customers. Since the breakup, AT&T continues to provide long distance service, with the seven regional Bell Operating Companies providing local service.

Companies in the United States that provide communication services to the public are called **common carriers**. Their offerings and prices are described by a document called a **tariff**, which must be approved by the Federal Communications Commission for the interstate and international traffic, and by the state Public Utilities Commissions for intrastate traffic.

In recent years a new breed of telecommunication company has emerged to provide specialized data communication services, often in direct competition with the telephone companies. Some of these companies offer high-performance long-distance transmission facilities (e.g., using satellites), whereas others provide time-sharing, networking, or other services using transmission facilities that they themselves rent from other common carriers.

At the other extreme are countries in which the national government has a complete monopoly on all communication, including the mail, telegraph, telephone, and often radio and television as well. Most of the world falls in this category. In some cases the telecommunication authority is a nationalized company, and in others it is simply a branch of the government, usually known as the **PTT** (**Post, Telegraph & Telephone** administration).

With all these different suppliers of services, there is clearly a need to provide compatibility on a worldwide scale to ensure that people (and computers) in one country can call their counterparts in another one. This coordination is provided by an agency of the United Nations called **ITU** (**International Telecommunication Union**). ITU has three main organs, two of which deal primarily with international radio broadcasting and one of which is primarily concerned with telephone and data communication systems.

The latter group is called **CCITT**, an acronym for its French name: Comité Consultatif International de Télégraphique et Téléphonique. CCITT has five classes of members: *A* members, which are the national PTTs; *B* members, which are recognized private administrations (e.g., AT&T); *C* members, which are scientific and industrial organizations; *D* members, which are other international organizations; and *E* members, which are organizations whose primary mission is in another field but which have an interest in CCITT's work. Only *A* members may vote. Since the U.S. does not have a PTT, somebody else had to represent it in

CCITT. This task fell to the State Department, probably on the grounds that CCITT had to do with foreign countries, the State Department's specialty.

CCITT's task is to make technical recommendations about telephone, telegraph, and data communication interfaces. These often become internationally recognized standards. Two examples are V.24 (also known as EIA RS-232 in the United States), which specifies the placement and meaning of the various pins on the connector used by most asynchronous terminals, and X.25, which specifies an interface between a computer and a (packet-switched) computer network.

1.6.2. Who's Who in the Standards World

International standards are produced by **ISO (International Standards Organization**†), a voluntary, nontreaty organization founded in 1946. Its members are the national standards organizations of the 89 member countries. These members include ANSI (U.S.), BSI (Great Britain), AFNOR (France), DIN (W. Germany), and 85 others.

ISO issues standards on a vast number of subjects, ranging from nuts and bolts (literally) to telephone pole coatings. ISO has almost 200 Technical Committees, numbered in the order of their creation, each dealing with a specific subject. TC1 deals with the nuts and bolts (standardizing screw thread pitches). TC97 deals with computers and information processing. Each TC has subcommittees (SCs) divided into working groups (WGs).

The real work is done largely in the WGs by over 100,000 volunteers worldwide. Many of these "volunteers" are assigned to work on ISO matters by their employers, whose products are being standardized. Others are government officials keen on having their country's way of doing things become the international standard. Academic experts also are active in many of the WGs.

On issues of telecommunication standards, ISO and CCITT *sometimes* cooperate (ISO is a *D* class member of CCITT) to avoid the irony of two official and mutually incompatible international standards.

The U.S. representative in ISO is **ANSI (American National Standards Institute)**, which despite its name, is a private, nongovernmental, nonprofit organization. Its members are manufacturers, common carriers, and other interested parties. ANSI standards are frequently adopted by ISO as international standards.

The procedure used by ISO for adopting standards is designed to achieve as broad a consensus as possible. The process begins when one of the national standards organizations feels the need for an international standard in some area. A working group is then formed to come up with a **DP (Draft Proposal)**. The DP is then circulated to all the member bodies, which get six months to criticize it. If a

† For the purist, despite the acronym, ISO's true name is the International Organization for Standardization.

substantial majority approves, a revised document, called a **DIS (Draft International Standard)** is produced and circulated for comments and voting. Based on the results of this round, the final text of the **IS (International Standard)** is prepared, approved, and published. In areas of great controversy, a DP or DIS may have to go through several versions before acquiring enough votes, and the whole process can take years.

NBS (National Bureau of Standards), is an agency of the U.S. Dept. of Commerce. It issues standards that are mandatory for purchases made by the U.S. Government, except for those of the Department of Defense, which has its own.

Another major player in the standards world is **IEEE (Institute of Electrical and Electronics Engineers)**, the largest professional organization in the world. In addition to publishing scores of journals and running numerous conferences each year, IEEE has a standardization group that develops standards in the area of electrical engineering and computing. IEEE's 802 standard for local area networks is the key standard for LANs. It has subsequently been taken over by ISO as the basis for ISO 8802.

1.6.3. Discussion of the Standardization of the OSI model

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. 1-13.

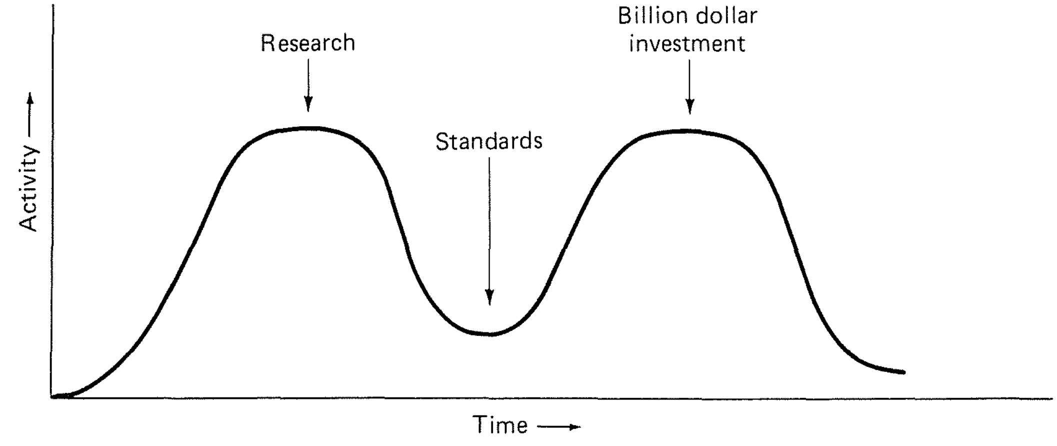


Fig. 1-13. The apocalypse of the two elephants.

This figure shows the amount of activity surrounding a new subject. When the subject is first discovered, there is a lot 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 is the belief of some workers in the field that this is what happened with the OSI model. 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. In the paragraphs that follow, we present some dissenting arguments.

It would have been nice if all seven layers had roughly the same size and importance. 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 5 layers, not 7. 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.

The 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. One problem 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 unnecessary and inefficient.

Another issue is that the placement of certain features in particular layers is not always obvious. The virtual terminal handling (now in the application layer) was in the presentation layer during part 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 that is the way most local area networks work. Subsequent addenda (known in the software world as bug fixes) have addressed this issue.

Perhaps the most serious criticism is that the model is completely 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 set of primitives discussed in Sec. 1.5.3 and listed in Fig. 1-11. 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.

If the program was expecting the attempt to connect to it, 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. Computers do not ring. In short, the semantic model of an interrupt-driven system is conceptually a very poor idea and totally at odds with all modern ideas of structured programming. This problem and similar ones are discussed by Langsford (1984).

1.7. 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 representative networks to get an idea of what they are like and how they differ from one another. Other networks are described by Quarterman and Hoskins (1986).

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.

1.7.1. Public Networks

In many countries the government or private companies have begun to offer networking services to any organization that wishes to subscribe. The subnet is owned by the network operator, providing communication service for the customers' hosts and terminals. Such a system is called a **public network**. It is analogous to, and often a part of, the public telephone system.

Although public networks in different countries are frequently quite different internally, virtually all of them use the OSI model and the standard CCITT or OSI protocols for all the layers. In addition, many private networks also use the OSI protocols, or are planning to use them in the near future.

For the lowest three layers, CCITT has issued recommendations that have been universally adopted by public networks worldwide. These layers are always known collectively as **X.25** (the CCITT recommendation number), although ISO has also adopted and numbered them as standards.

The physical layer protocol, called **X.21**, specifies the physical, electrical, and procedural interface between the host and the network. Very few public networks actually support this standard, because it requires digital, rather than analog signaling on the telephone lines, but it may conceivably become more important in the future. As an interim measure, an analog interface similar to the familiar RS-232 standard has been defined.

The data link layer standard has a number of (slightly incompatible) variations. They all are designed to deal with transmission errors on the telephone line between the user's equipment (host or terminal) and the public network (IMP).

The network layer protocol deals with addressing, flow control, delivery confirmation, interrupts and related issues.

Because the world is still full of terminals that do not speak X.25, another set of standards has been defined that describes how an ordinary (nonintelligent) terminal communicates with an X.25 public network. In effect, the user or network operator installs a "black box" to which these terminals can connect. The black box is called a **PAD** (**Packet Assembler Disassembler**), and its function is described in a CCITT recommendation known as **X.3**. A standard protocol has been defined between the terminal and the PAD, called **X.28**; another standard protocol exists between the PAD and the network, called **X.29**. Together, these three recommendations are often called **triple X**. We will discuss these standards in Chap. 9.

Above the network layer, the situation is less uniform. ISO has developed standards for a connection-oriented transport layer service definition (ISO 8072) and a connection-oriented transport layer protocol (ISO 8073). Most public networks will no doubt eventually adopt these. The transport service has five different variants. These will be discussed later in the book in the chapter on the transport layer.

ISO has also adopted standards for the connection-oriented session service and protocol (ISO 8326 and ISO 8327) and presentation service and protocol (ISO 8822 and ISO 8823). These will also be adopted eventually by most public networks, although the need is less critical than for a uniform transport service since many applications have no real need for session or presentation services at all.

The application layer contains not just one, but a whole collection of protocols for various applications. The **FTAM** (**File Transfer, Access, and Management**) protocol provides a way to transfer, access, and generally manipulate remote files in a uniform way. The **MOTIS** (**Message-Oriented Text Interchange Systems**) protocol is used for electronic mail. It is similar to the CCITT **X.400** series of recommendations. The **VTP** (**Virtual Terminal Protocol**) protocol provides a terminal-independent way for programs to access remote terminals (e.g., for full-screen editors). The **JTM (Job Transfer and Manipulation)** protocol is used for submitting jobs to remote mainframe computers for batch processing. It can be used to move both programs and data files. Numerous other industry-specific and application-specific protocols have also been defined, and new ones are being thought up and standardized all the time.

Figure 1-14 shows the numbers of a few of the key international standards. The style of the standards changes radically between layers 3 and 4, since the lower ones were done by CCITT and the upper ones were done by ISO. In particular, CCITT never bothered to distinguish between the service and the protocol, something ISO has meticulously done. Many other standards, including most of the connectionless ones, are not listed. Copies of all the standards are available from national standards organizations (e.g., ANSI).

Layer	Standard	Description
1-7	ISO 7498	ISO OSI Basic reference model
7	ISO 8571	File transfer, access and manipulation service
	ISO 8572	File transfer, access and manipulation protocol
	ISO 8831	Job transfer and manipulation service
	ISO 8832	Job transfer and manipulation protocol
	ISO 9040	Virtual terminal service
	ISO 9041	Virtual terminal protocol
	CCITT X.400	Message handling (electronic mail)
6	ISO 8822	Connection-oriented presentation service
	ISO 8823	Connection-oriented presentation protocol
5	ISO 8326	Connection-oriented session service
	ISO 8327	Connection-oriented session protocol
4	ISO 8072	Connection-oriented transport service
	ISO 8073	Connection-oriented transport protocol
3	CCITT X.25	X.25 layer 3 protocol
2	ISO 8802	Local area networks
	CCITT X.25	HDLC/LAPB data link layer
1	CCITT X.21	Physical layer digital interface

Fig. 1-14. A few of the key ISO and CCITT network standards. ISO refers to an ISO International Standard. ISO 8802 is derived from IEEE 802.

1.7.2. The ARPANET

The ARPANET is the creation of ARPA (now DARPA), the (Defense) Advanced Research Projects Agency of the U.S. Department of Defense. Starting in the late 1960s it began stimulating research on the subject of computer networks by providing grants to computer science departments at many U.S. universities, as well as to a few private corporations. This research led to an experimental four-node network that went on the air in December 1969. It has been operating ever since, and has subsequently grown to several hundred computers spanning half the globe, from Hawaii to Sweden. Much of our present knowledge about networking is a direct result of the ARPANET project. To honor this pioneering work, we have adopted some of the original ARPANET terminology (e.g., host, IMP, subnet).

After the ARPANET technology had proven itself by years of highly reliable service, a military network, MILNET, was set up using the same technology. An extension of MILNET in Europe, called MINET, was also created. MILNET and MINET are connected to the ARPANET, but the traffic between the parts is tightly controlled. Two satellite networks, SATNET and WIDEBAND, were also hooked up later. Since many of the universities and government contractors on the ARPANET had their own LANs, eventually these were also connected to the IMPs, leading to the **ARPA internet** with thousands of hosts and well over 100,000 users. The ARPANET will be frequently cited throughout this book, but most of what is said about the ARPANET (e.g., the protocols used) also holds for the entire internet.

The original ARPANET IMPs were Honeywell DDP-516 minicomputers with 12K 16-bit words of memory. As time has gone on, the IMPs have been replaced several times by more powerful machines. They are now called **PSN**s (**Packet Switch Nodes**), but their function is the same as it was.

Some of the IMPs have been configured to allow user terminals to call them directly, instead of logging into a host. These were called **TIPs** (**Terminal Interface Processors**) and are now called **TACs** (**Terminal Access Controllers**). The IMPs were originally connected by 56 kbps leased lines, although higher speed (e.g., 230.4 kbps), lines are now also used. The original IMPs could handle one to four hosts apiece. The current ones can handle tens of hosts and hundreds of terminals simultaneously.

The ARPANET does not follow the OSI model at all (it predates OSI by more than a decade). The IMP-IMP protocol really corresponds to a mixture of the layer 2 and layer 3 protocols. Layer 3 also contains an elaborate routing mechanism. In addition, there is a mechanism that explicitly verifies the correct reception at the destination IMP of each and every packet sent by the source IMP. Strictly speaking, this mechanism is another layer of protocol, the source IMP to destination IMP protocol. However, this protocol does not exist in the OSI model. We will treat it as part of layer 3, since it is closer to that layer than to any other.

The ARPANET does have protocols that roughly cover the same territory as the

OSI network and transport protocols. The network protocol, called **IP (Internet Protocol)**, is connectionless and was designed to handle the interconnection of the vast number of WAN and LAN networks comprising the ARPA internet. The OSI model only grudgingly dealt with internetworking at all, as an afterthought, whereas the concept was central to the design of IP.

The ARPANET transport protocol is a connection-oriented protocol called **TCP (Transmission Control Protocol)**. It resembles the OSI transport protocol in its general style, but it differs in all the formats and details. TCP is used in Berkeley UNIX, which makes it very widespread even though it is not part of the OSI suite. Just as an aside, it should be pointed out that TCP is the second generation transport protocol. The first one is no longer used.

There are no session or presentation layer protocols in the ARPANET as no one has had any use for them in the first 20 years of operation. Various application protocols exist, but they are not structured the same way as their OSI counterparts. The ARPANET services include file transfer, electronic mail, and remote login. These services are supported by the well-known protocols **FTP (File Transfer Protocol)**, **SMTP (Simple Mail Transfer Protocol)**, and **TELNET** (remote login). Various specialized protocols are available for other applications.

1.7.3. MAP and TOP

If two sites each use the OSI model, there is no guarantee that they will be able to communicate with each other. The model consists of a framework telling what the layers are, but does not itself specify the protocols to be used in each layer. While ISO has standardized certain protocols, other nonstandard protocols also exist. Furthermore, in some layers, multiple, incompatible standard protocols also exist. If two sites use different protocols in any layer, they will not be able to communicate.

Many readers may just throw up their hands in disgust and say: "Why can't the experts just agree on one standard protocol in each layer?" It is instructive to look at a concrete example to see how the problem arises. In 1973, Robert Metcalfe wrote a Ph.D. thesis at M.I.T. in which he described his research on LANs. Metcalfe then moved to the Xerox Corporation where he teamed up with David Boggs and others to implement the **Ethernet**† LAN based on the ideas in Metcalfe's thesis. Ethernet was quickly adopted by many companies and Intel later built a single-chip controller for it. It did not take long before Ethernet became the de facto standard for LANs.

Shortly thereafter, some people felt that there should be an *official* standard for LANs. A group of people working under the auspices of IEEE was set up to write

† Ethernet is a trademark of the Xerox Corporation.

one. Many people were invited to join the committee and contribute to the work. Some of these people came from General Motors, which was also carefully looking at LANs.

In order to compete with the Japanese automobile companies, GM wanted to set up a network covering all its offices, factories, dealers and suppliers. The idea was that when a customer ordered a car anywhere in the world, the dealer's computer would instantly send the order to GM, which would then send orders to its suppliers ordering the necessary steel, glass, and other raw materials.

An important part of the GM network was factory automation, in which all the robots working on the assembly lines would all be connected by LANs. Since the cars on the assembly lines move by at a fixed rate, whether the robots are ready or not, GM felt it essential to have a LAN in which the worst case transmission time had an upper bound that was known in advance. Ethernet does not have this property.

Essentially, Ethernet works by having all the machines listen to the cable. If the cable is idle, any machine may transmit. If two machines transmit at the same time, there is a collision, in which case they all stop, wait a random period of time, and try again later. In theory, there is no upper bound on the time a machine might have to wait to send a message. GM wanted a LAN called the **token bus** in which the machines took turns, round-robin, thus giving a deterministic worst-case performance rather than a statistical one.

While all this was going on, IBM announced that it had adopted yet a third LAN, the **token ring**, as its standard. The token ring was based on a prototype built at IBM's Zurich lab. IBM chose this design due to its high reliability and serviceability as well as some other technical advantages.

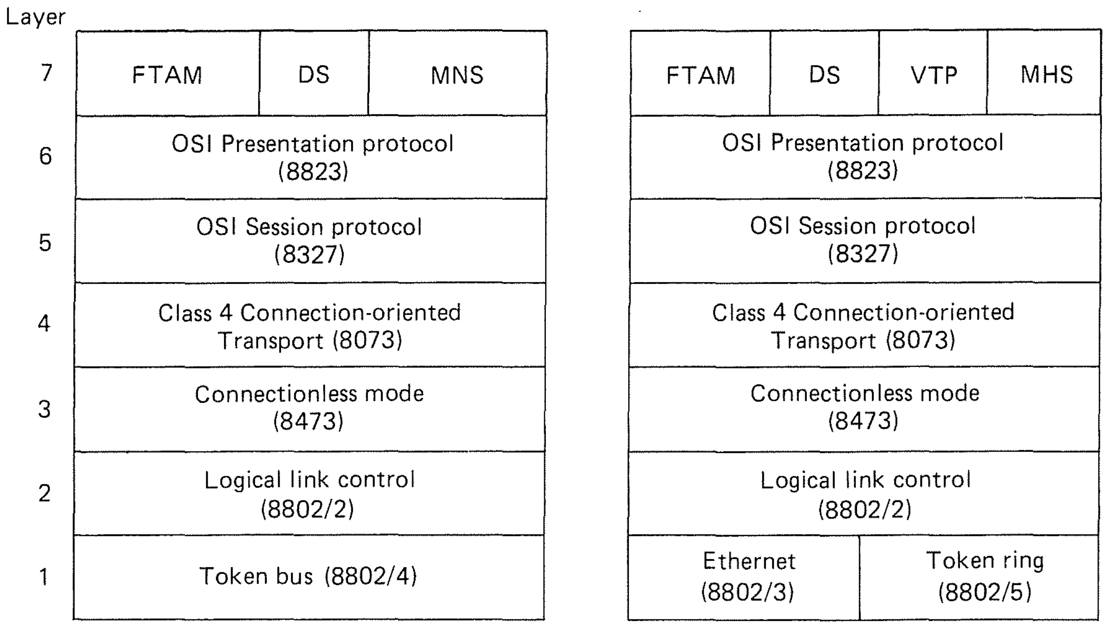
In short order, the IEEE committee had three proposals, one backed by DEC, Xerox, Intel, and the office automation people, one backed by GM, its suppliers, and other people concerned about factory automation, and one backed by IBM and many others. The necessary majority to approve a standard was simply lacking. Finally, the committee threw in the towel and approved all three LAN standards, known now as IEEE 802.3 (based on Ethernet), 802.4 (token bus), and 802.5 (token ring) on the theory that three standards were better than no standards.

Anyway, GM and other companies concerned with factory automation clearly saw the need to adopt specific protocols in each OSI layer to avoid further incompatibilities. This work led to the **MAP** (**Manufacturing Automation Protocol**) using the token bus, which was quickly adopted by many companies in the manufacturing world.

At about the same time, Boeing was interested in standards for office automation. Boeing preferred the Ethernet to the token bus because it had no real time requirements (747s do not roll down assembly lines) and Ethernet already had a huge installed base. Boeing eventually developed a set of protocols for office automation called **TOP** (**Technical and Office Protocols**) that many other companies also adopted for office automation. Although MAP and TOP differ at the lowest layers, GM and Boeing worked closely to ensure that they would be fully compatible in the middle and upper layers.

A collection of protocols, one per layer, such as MAP or TOP, is called a **protocol suite** or **protocol stack**. In addition to MAP and TOP, the protocol suites used by the public networks and the ARPANET are also widely used. Others can be expected in the future, so compatibility between different ones may become a crucial issue.

Now let us turn from the history of MAP and TOP to a closer examination of their technical contents. Both suites follow the OSI model very closely, as shown in Fig. 1-15. MAP uses the token bus for the physical medium and TOP uses the Ethernet (or token ring, which was added in 1987). Both use the IEEE 802.2 data link protocol **LLC (Logical Link Control)** in the data link layer in connectionless mode as the service available to the network layer.



Layer		
7	FTAM DS MNS	FTAM DS VTP MHS
6	OSI Presentation protocol (8823)	OSI Presentation protocol (8823)
5	OSI Session protocol (8327)	OSI Session protocol (8327)
4	Class 4 Connection-oriented Transport (8073)	Class 4 Connection-oriented Transport (8073)
3	Connectionless mode (8473)	Connectionless mode (8473)
2	Logical link control (8802/2)	Logical link control (8802/2)
1	Token bus (8802/4)	Ethernet (8802/3) Token ring (8802/5)

Fig. 1-15. TOP and MAP protocol suites.

Both suites also use a connectionless network layer protocol, ISO 8473. This protocol is very close to the ARPA internet's IP protocol, but completely different in style and details from the CCITT X.25 protocol used on public networks. The reason for this choice is that years of experience with the ARPA internet have shown that the datagram approach is much more flexible and robust when connecting multiple heterogenous networks together, an important aspect of MAP and TOP.

The MAP and TOP transport layer is ISO 8072/8073, using class 4 service. This class assumes the network layer is not completely reliable, and handles all the error control and flow control itself. By making this choice, MAP and TOP have made it possible to connect to almost any kind of network, no matter how bad. The price paid, of course, is a complex transport layer that must deal with an unreliable network service. The ARPANET, which works the same way, made this decision years ago after observing that network service *was* unreliable, so the transport protocol had to deal with it, whether or not it wanted to. In contrast, X.25 service is supposed to be highly reliable, allowing for a simpler class of transport service to be used with it.

The OSI connection-oriented standards ISO 8326/8327 and ISO 8822/8823 are used in MAP and TOP in the session and presentation layers. The OSI standards are also used in the application layer. In particular, the file transfer protocol, virtual terminal protocol, and others are used.

TOP recognizes five types of systems that may be present in a TOP network: an end system, a repeater, a bridge, a router, and a gateway. The **end system** (host) is a user machine, containing all seven OSI layers. The other four types are used to connect together different networks. They differ in the layer at which the connection is made, as shown in Fig. 1-16.

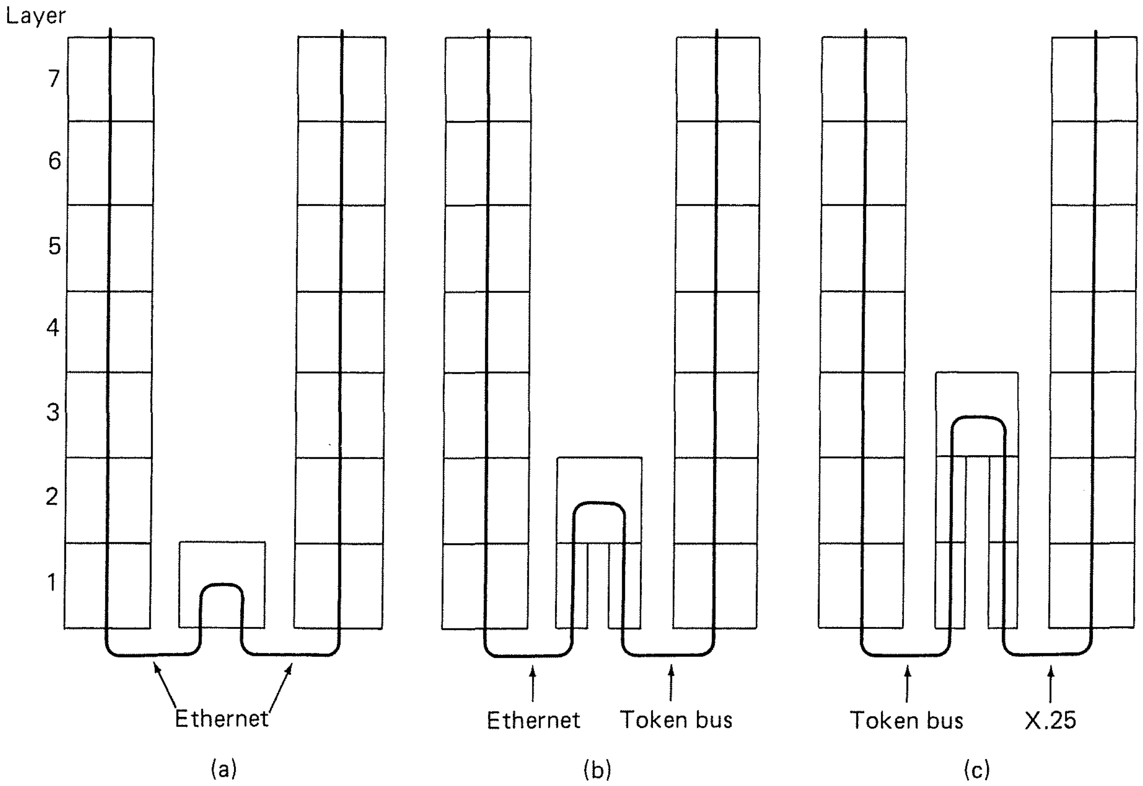


Fig. 1-16. (a) A repeater. (b) A bridge. (c) A router.

A **repeater**, as shown in Fig. 1-16(a), just forwards bits from one network to another, making the two networks look logically like one network. Networks are often split into two (or more) pieces due to maximum cable length restrictions on the individual pieces. Repeaters are dumb (no software); they just copy bits blindly without understanding what they are doing.

A **bridge**, as shown in Fig. 1-16(b), can be used to connect two networks at the data link layer. This approach is useful, for example, when the networks have different data link layers but the same network layer. A connection between an Ethernet and a token bus would normally be a bridge. The frames from the Ethernet arrive at the bridge in Ethernet form and are copied to the token bus in token bus form, or vice versa. Bridges are smart (full of software); they can be programmed to copy frames selectively and make necessary changes while doing so.

A **router**, as shown in Fig. 1-16(c), is needed when the two networks use the same transport layer, but have different network layers. A connection between a token bus and an X.25 public network would need a router to convert the token bus frames to the form required by the X.25 network.

The final type of TOP node, the **gateway**, is used to connect to a network that does not use the OSI model at all. In many cases the connection will have to be done in the application layer.

has six types of nodes, including the MAP end system, bridge, router, and gateway. It does not have the repeater (using bridges for that purpose), but it does has two additional node types, the **MINIMAP** node and the MAP/EPA gateway. These nodes are compatible with the PROWAY LAN standard, which was common in factory environments prior to MAP. They only have layers 1 and 2, and are important for critical real time work. It is expected that over a million MAP and TOP nodes will be in service by the early 1990s.

1.7.4. USENET

When UNIX first came into widespread use at Bell Laboratories, researchers there quickly realized that they needed a way to copy files from one UNIX system to another. To solve this problem, they wrote the *uucp* (UNIX to UNIX CoPy) program.

As UNIX systems acquired modems and automatic telephone dialers, it became possible to copy files automatically between distant machines using *uucp*. A number of informal networks sprung up in which one central machine with an automatic dialer would call up a group of machines late at night, one at a time, to log in and transfer files and electronic mail between them. Two machines that had modems but no automatic dialer could communicate by having the central machine call up the first one and upload all outgoing files and mail. Later, when the central machine called the destination machine, the files and mail were downloaded to it.

These networks grew extremely rapidly because all one needed to join the network was a UNIX system with a modem, something which practically every university computer science department in the Western world had. These networks have since joined up to form a single network, sometimes referred to as **UUCP** with about 10,000 machines and a million users. It is probably the largest network in the world.

Unlike the various public networks or the ARPANET, which are centrally administered, this network can best be described as near total anarchy. Each site can run any version of the software it wants to, and many of them use this freedom to the fullest. On the other hand, the user community has hundreds of competent and dedicated *wizards* (experts) who keep the network running extremely well, despite the total lack of central control.

The European segment even has an official name: EUnet, and also has a more organized structure. Each European country has a single gateway machine run by a single administrator. The administrators keep in close touch to coordinate the network. All international traffic goes between the gateways. The gateways then feed the traffic out into the national networks. Europe and the U.S. are connected by a leased line between Amsterdam and a site in Virginia called **uunet** that is also on the ARPANET. Segments also exist in Japan, Korea, Australia, and other countries.

The only service offered is electronic mail, but a companion network called **USENET**, started at Duke University and the University of North Carolina, also offers an unusual service called **network news**. While some UNIX machines in the U.S. are only on one or the other, most of them are on both. In Europe, EUnet carries both mail and news, so there is only one network. Since both networks are administered the same way and run the same protocols, throughout this book we will often treat them as a single network under the name of USENET, even though this is not strictly true.

Network news is divided into hundreds of newsgroups on a variety of topics, some technical and some not. There are groups for each popular programming language, for each common microcomputer, and for several operating systems, as well as groups for people offering or seeking jobs, people wanting to buy or sell things, and groups for many recreational activities and sports. USENET users can subscribe to whatever groups they are interested in.

Users can also post messages, which are then copied (usually by *uucp*) to all the machines in the entire world that carry that news group. A user with a question or opinion on some subject can post a message and perhaps start a discussion that may eventually involve hundreds or thousands of people all over the world. Many "bulletin board systems" run by computer hobbyists have similar features, but none of these have anywhere near the number of subscribers or the worldwide reach that USENET does.

1.7.5. CSNET

By 1980, the enormous value of the ARPANET for communication among researchers had become obvious. The main problem with the ARPANET was that it was owned by the Department of Defense, and those universities not having federal contracts were not permitted to use it. To provide networking facilities to the computer science community as a whole, NSF (National Science Foundation) set up **CSNET**, which was to be accessible to all computer science departments in the U.S. (Comer, 1983; Landweber et al., 1986; Partridge et al., 1987).

Actually, CSNET is not a real network like the ARPANET, but a metanetwork. It uses transmission facilities provided by other networks and adds a uniform protocol layer on top to make the whole thing look like a single logical network to the users. Physically, CSNET initially consisted of three components; later a fourth one was added. All the pieces are tied together by a machine called CSNET-RELAY located at a company called BBN in Cambridge, Mass. BBN was chosen to run CSNET due to its considerable experience running the ARPANET. Altogether, about 150 campuses are on CSNET.

The first component is the ARPANET. Many CS departments are already on it, and these hosts are already connected by a subnet using 56-kbps leased lines, much higher bandwidth than CSNET could possibly afford.

The second component is the public X.25 network. Those departments that are on one of the X.25 networks, such as Telenet or Uninet, can use that network to reach CSNET-RELAY.

For those departments not on the ARPANET or an X.25 network, **PHONENET** was set up. This network consists of having the PHONENET hosts simply calling up CSNET-RELAY whenever they want to. Most use automatic dialers to call every hour or two. This is similar to USENET, except here, all traffic is to or from CSNET-RELAY, whereas with USENET, there is no centralized control. Each USENET machine can directly communicate with any other machine it wants to.

The fourth component, called **CYPRESS**, is an attempt to duplicate the ARPANET technology on a lower budget (Comer and Narten, 1988). It has packet-switching nodes, called **IMPLET**s, and hosts. The IMPLETs are DEC Microvaxes connected by leased lines. To join CYPRESS, an organization has to acquire a Microvax and lease a line to some other IMPLET.

The basic service provided to all CSNET sites is electronic mail, using the ARPANET protocols and formats. Except for the PHONENET hosts, file transfer and remote login are also possible.

Having seen how well CSNET worked, NSF set up another network to provide access to supercomputers spread around the U.S. Since supercomputer programs often need large amounts of data, this network uses 1.5 Mbps microwave links for long-haul transmission. At this rate, a full 1600 bpi magnetic tape can be sent in under 5 minutes. This network, called **NSFNET** (Mills and Braun, 1987), uses the same TCP/IP protocols as the ARPA internet, CSNET, and Berkeley UNIX. By now it should be clear that the large number of networks and machines running the internet protocols means that it will be a considerable time before the whole world is converted to the OSI protocols, if ever.

1.7.6. BITNET

Another interesting network is **BITNET (Because It's Time NETwork)**, started in 1981 by City University of New York and Yale University (Landweber et al., 1986). The idea behind it was to create a university network, like CSNET, but for all departments, not just computer science. It has now spread to about 175 sites in the U.S. and about 260 sites in Europe via its counterpart there, called **EARN** (**European Academic Research Network**).

Technically, BITNET is somewhat similar to CYPRESS, with each BITNET site leasing a line to some other site. Unlike CYPRESS, the BITNET hosts themselves do the communication; there is no subnet of IMPLETs. Also unlike CYPRESS, BITNET uses a protocol and software donated to it by IBM that is not compatible with either OSI or TCP/IP (or anything else). It is based on the idea of transmitting 80-column punched card images, a frequent source of problems.

The financing of BITNET is unusual, and accounts for part of its popularity. To join, a university must lease a line to some other BITNET site and pay the rental cost of the line. It must also permit some other university to lease a line to it in the future (at the other university's expense). Finally, it must agree to forward traffic passing through it for free. Other than the loss of some computing power for forwarding third-party traffic, the only real cost is one leased line. Unlike just about every other network, there are no charges based on the volume of traffic sent.

The basic BITNET service is file transfer, which also includes electronic mail and remote job entry. Each file entered into the system contains its final destination, and may be stored and forwarded many times before it gets there. A limited amount of remote login is possible, but since the interactive traffic is stored and forwarded just like the file transfer traffic, response time is very slow and reliability is low.

Work is in progress to add a CYPRESS-like subnet to BITNET, and to try to integrate it better into the ARPA internet, USENET, and CSNET worlds.

1.7.7. SNA

No discussion of networking would be complete without at least a few words about IBM's network architecture, **SNA** (**Systems Network Architecture**). The OSI model was patterned after SNA to a considerable extent, including the concept of layering, the number of layers chosen, and their approximate functions.

SNA is a network architecture intended to allow IBM customers to construct their own private networks, both the hosts and the subnet. A bank, for example, might have one or more CPUs in its data-processing department and numerous terminals in each of its branch offices. Using SNA, all these isolated components could be transformed into a coherent system.

Prior to SNA, IBM had several hundred communication products, using three dozen teleprocessing access methods, with more than a dozen data link protocols alone. The idea behind SNA was to eliminate this chaos and to provide a coherent framework for loosely coupled distributed processing. Given the desire of many of IBM's customers to maintain compatibility with all these (mutually incompatible) programs and protocols, the SNA architecture is more complicated in places than it might have been had these constraints not been present. SNA also performs a large number of functions not found in other networks, which, although valuable for certain applications, tend to add to the overall complexity of the architecture.

SNA has evolved considerably over the years and is still evolving. The original release in 1974 permitted only centralized networks, that is, tree-shaped networks with only a single host and its terminals. From our point of view, that is no network at all. The 1976 release allowed multiple hosts with their respective trees, with intertree communication possible only between the roots of the trees. The 1979 release removed this restriction, allowing somewhat more general communication. Finally, in 1985, arbitrary topologies of hosts and LANs were supported.

An SNA network consists of a collection of machines called **nodes**, of which there are four types, approximately characterized as follows. Type 1 nodes are terminals. Type 2 nodes are controllers, machines that supervise the behavior of terminals and other peripherals. Type 4 nodes are front end processors, devices whose function is to relieve the main CPU of the work and interrupt handling associated with data communication. Type 5 nodes are the main hosts themselves, although with the advent of low-cost microprocessors, some controllers have acquired some host-like properties. Strangely enough, there are no type 3 nodes.

Each node contains one or more **NAU**s (**Network Addressable Units**). A NAU is a piece of software that allows a process to use the network. It can be regarded as a SAP plus the entities that provide the upper layer services. To use the network, a process must connect itself to a NAU, at which time it can be addressed and can address other NAUs. The NAUs are thus the entry points into the network for user processes.

There are three kinds of NAUs. An **LU** (**logical unit**) is the usual variety to which user processes can be attached. A **PU** (**physical unit** is a special administrative NAU associated with each node. The PU is used by the network to bring the node online, take it offline, test it, and perform similar network management functions. A PU provides a way for the network to address a physical device, without reference to which processes are using it. The third kind of NAU is the **SSCP** (**System Services Control Point**) of which there is normally one per type 5 node and none in the other nodes. The SSCP has complete knowledge of, and control over, all the front ends, controllers, and terminals attached to the host. The collection of hardware and software managed by an SSCP is called a **domain**. Figure 1-17 depicts a simple two-domain SNA network.

While it is possible to make a rough mapping of the SNA layers to the OSI layers, when you look at them in detail, the two models do not correspond especially well, especially in layers 3, 4, and 5. A summary of the SNA layers follows.

The lowest SNA layer, shown in Fig. 1-18 takes care of physically transporting bits from one machine to another. The protocols used in this layer generally conform to the appropriate industry standards.

The next layer, the **data link control** layer, constructs frames from the raw bit stream, detecting and recovering from transmission errors in a way transparent to higher layers. Many networks have directly or indirectly copied their layer 2

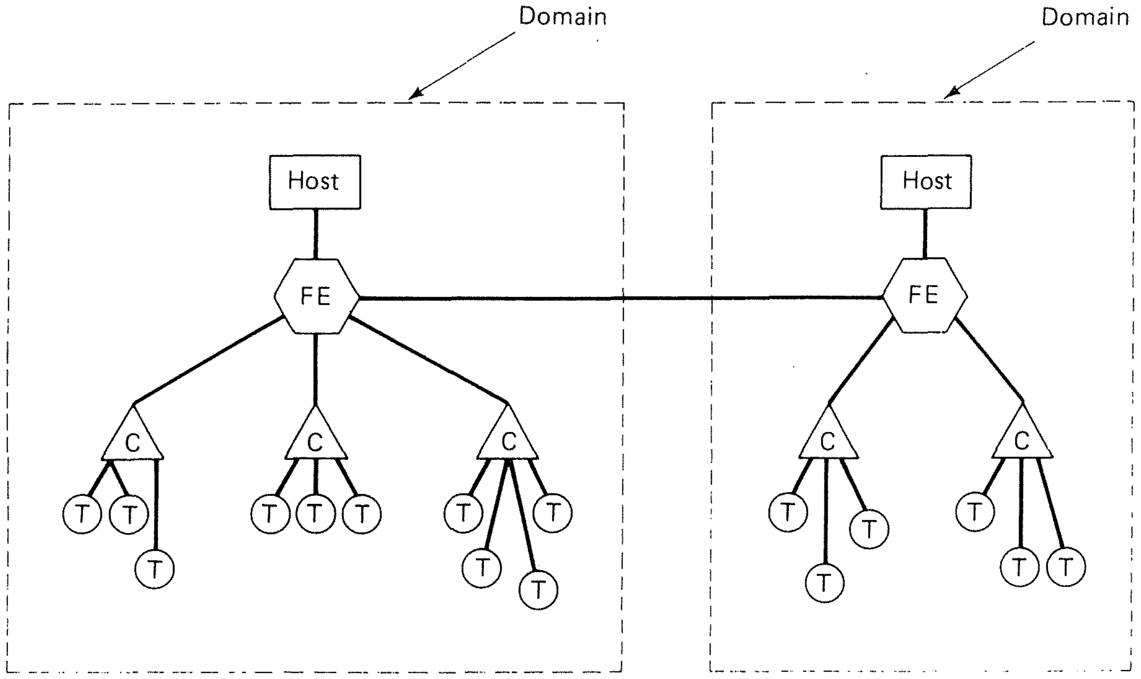


Fig. 1-17. A two domain SNA network. FE = Front End, C = Controller, T = Terminal.

protocol from SNA's layer 2 data communication protocol, **SDLC** (**Synchronous Data Link Control**). In particular, ISO's **HDLC** (**High Level-Data Link Control**) is closely patterned on SDLC. SNA also supports a token ring LAN in this layer.

Layer 3 in SNA, called **path control** by IBM, is concerned with establishing a logical path from the source NAU to the destination NAU. Many SNA networks are split up into subnetworks, called **subareas**, each of which has a special subarea node that acts as a gateway. A subarea often corresponds to a domain. This design leads to a hierarchical structure, with the subarea nodes connected together to form a backbone, and each node connected to some subarea node.

Path control consists of three sublayers. The highest sublayer does the global routing, deciding which sequence of subareas should be used to get from the source subarea to the destination subarea. This sequence is called a **virtual route**. Two subareas may be connected by several different kinds of communication lines (e.g., leased line and satellite), so the next sublayer chooses the specific lines to use, giving an **explicit route**. The lowest sublayer splits traffic among several parallel communication links of the same type to achieve greater bandwidth and reliability.

Information concerned with finding virtual and explicit routes, and managing network congestion is passed in the **transmission header**, shown in Fig. 1-18. Path control can also block unrelated packets together into large units for greater efficiency.

Above the path control layer is the **transmission control** layer whose job it is

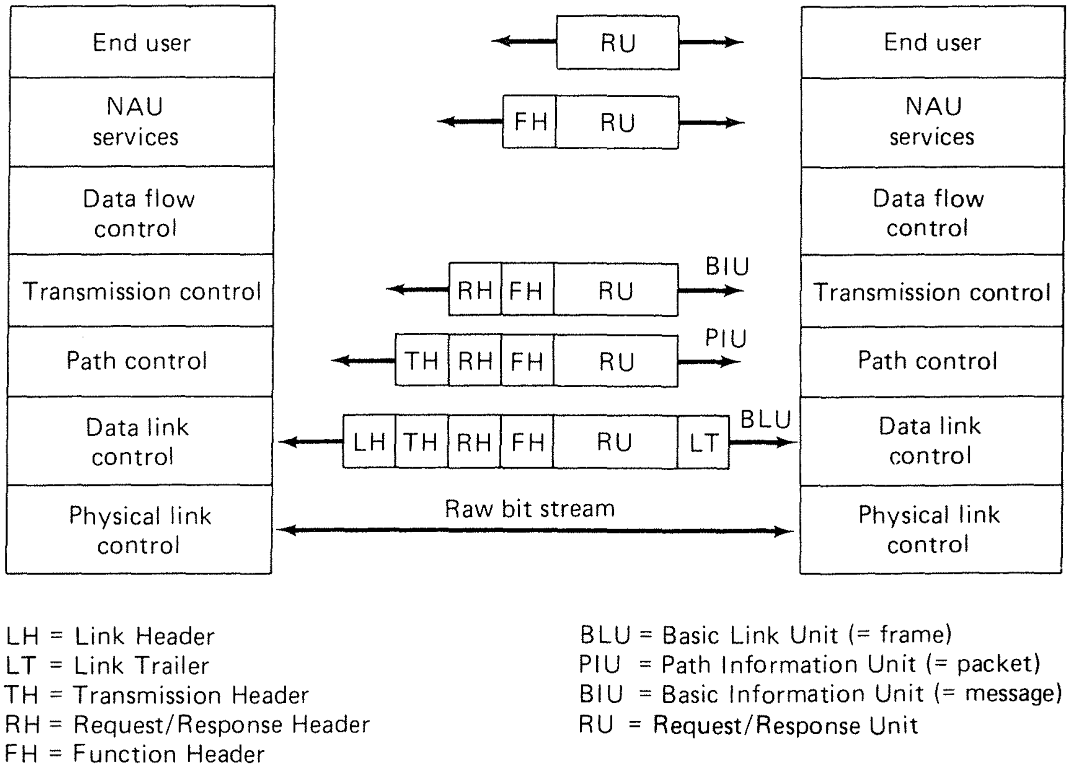


Fig. 1-18. Protocol hierarchy and units exchanged in SNA.

to create, manage, and delete transport connections (sessions). All communication in SNA uses sessions; connectionless communication is not supported. The purpose of a session in SNA, as in the OSI model, is to provide the upper layers with an error-free channel that is independent of the underlying hardware technology.

SNA distinguishes five different kinds of sessions:

1. SSCP—SSCP: for interdomain control and management messages.
2. SSCP—PU: to allow the SSCP to initialize, control, and stop PUs.
3. SSCP—LU: to enable LUs to manage sessions.
4. LU—LU: for transmitting user data.
5. PU: for network management.

In the OSI model, any process can send a message to any other process requesting that a session be established. If the called party agrees, it sends back a response that establishes the session. In SNA, the situation is much more complicated and differs for each of the session types. We will just consider user to user (i.e., LU—LU) sessions below.

To establish a session, a process must talk to the session control manager in its domain. If the destination is local (same domain), it can be established directly.

However, if the destination is in a remote domain, the SSCP must first contact the SSCP controlling the remote domain. Virtual and explicit routes must also be chosen. This mechanism is quite cumbersome, requiring the exchange of over a dozen control messages. Once a session has been established, the transmission control layer regulates the rate of flow between the processes, controls buffer allocation, manages the different message priorities, handles multiplexing and demultiplexing of data and control messages for the benefit of higher layers, and performs encryption and decryption when requested to do so.

Above transmission control comes **data flow control**, which has nothing at all to do with controlling the flow of data in the usual sense. Instead, it has to do with keeping track of which end of a session is supposed to talk next, assuming that the processes want such a service.

This layer is also heavily involved in error recovery. A somewhat unusual feature of the data flow control layer is the absence of a header used to communicate with the corresponding software on the other end. Instead, the information that would normally be communicated in a header is passed to transmission control as parameters and included in the transmission header.

The sixth layer within SNA, **NAU services**, provides two classes of services to user processes. First, there are **presentation services**, such as text compression. Second, there are **session services**, for setting up connections. In addition, there are **network services**, which have to do with the operation of the network as a whole.

SNA is a vast and complex subject. An excellent book on the subject is (Meijer, 1987).

1.8. OUTLINE OF THE REST OF THE BOOK

This book is structured according to the OSI model. While this model may or may not be a good way to organize real, live computer networks, it makes an excellent framework for organizing books about them.

Starting with Chapter 2, we begin working our way up the protocol hierarchy beginning at the bottom. The second chapter provides some background in the field of data communication. It covers analog and digital transmission, multiplexing, and switching. This material is concerned with the physical layer, although we cover only the architectural rather than the hardware aspects. Several examples of the physical layer are also discussed.

Chapter 3 concerns the Medium Access Sublayer. When the IEEE 802 committee invented this sublayer, they thought it belonged at the bottom of layer 2. ISO disagreed initially, thinking that it belonged at the top of layer 1. Either way, it goes between the physical transmission and the data link protocols, and is important enough to warrant an entire chapter. The basic question it deals with is how to determine who may use the network next when the network consists of a single shared channel, as in most LANs and some satellite networks.

Chapter 4 discusses the data link layer and its protocols by means of a number of increasingly complex examples. The analysis of these protocols is also covered.

Chapter 5 deals with the network layer, especially routing, congestion control, and internetworking. It discusses both static and dynamic routing algorithms. Broadcast routing is also covered. The effect of poor routing, congestion, is also discussed in some detail. Connecting heterogenous networks together to form internetworks leads to numerous problems that are discussed here.

Chapter 6 deals with the transport layer. Much of the emphasis is on connection-oriented protocols, since both the OSI transport protocol and TCP are of this type. An example transport service and its implementation are discussed in detail.

Chapter 7 is about the session layer. First some of the design issues concerning the OSI session layer are discussed. Like the transport layer, this is a connection-oriented service and protocol. Then comes a discussion of a connectionless session protocol, the remote procedure call. This protocol is very popular among researchers building distributed operating systems.

Chapter 8 is about the presentation layer. A large part of the chapter deals with an internationally standardized language for describing data types, such as packet formats, and how these data types are represented "on the wires." This chapter also deals with network security and privacy since the presentation layer is a reasonable place to put these functions (but not the only one).

Chapter 9 goes into some of the issues that occur in the application layer. Among these are file servers, electronic mail, virtual terminal protocols, directory services, and remote job entry.

Chapter 10 contains an annotated list of suggested readings arranged by chapter. It is intended to help those readers who would like to pursue their study of networking further. The chapter also has an alphabetical bibliography of all references cited in this book.

Throughout the book we will use four networks as running examples to illustrate the principles discussed. These are the public networks (such as X.25 networks), the ARPANET (and ARPA internet), MAP/TOP, and USENET.

1.9. SUMMARY

Networks are being developed both to connect existing machines and to take advantage of the low-cost, high-performance microprocessors the semiconductor industry is turning out. Most wide area networks have a collection of hosts communicating via a subnet. The subnet may utilize multiple point-to-point lines between its IMPs, or a single common broadcast channel, as in a satellite network. Local-area networks connect the hosts directly onto a cable using an interface chip that is somewhat analogous to the IMP in a wide area network.

Networks are always designed as a series of protocol layers, with each layer responsible for some aspect of the network's operation. The seven-layer OSI model, consisting of the physical link layer, data link layer, network layer, transport layer, session layer, presentation layer, and application layer, forms the conceptual backbone of this book.

The physical layer is concerned with standardizing network connectors and their electrical properties. The data link layer breaks the raw bit stream up into discrete units and exchanges these units using a protocol. The network layer takes care of routing. The transport layer provides reliable, end-to-end connections to the higher layers. The session layer enhances the transport layer by adding facilities to help recover from crashes and other problems. The presentation layer deals with standardizing the way data structures are described and represented. Finally, the application layer contains file transfer, electronic mail, virtual terminal, and a number of application specific protocols.

Quite a few networks are now in operation. Some of the ones we have touched upon are the public networks, the ARPANET, MAP and TOP, USENET, CSNET, BITNET, and SNA.

PROBLEMS

1. Imagine that you have trained your St. Bernard, Bernie, to carry a box of three floppy disks instead of a flask of brandy. (When your disk fills up, you consider that an emergency.) These floppy disks each contain 250,000 bytes. The dog can travel to your side, wherever you may be, at 18 km/hour. For what range of distances does Bernie have a higher data rate than a 300 bps telephone line?

2. In the future, when everyone has a home terminal connected to a computer network, instant public referendums on important pending legislation will become possible. Ultimately, existing legislatures could be eliminated, to let the will of the people to be expressed directly. The positive aspects of such a direct democracy are fairly obvious; discuss some of the negative aspects.

3. Consider $2^n - 1$ IMPs connected by the following topologies:
   (a) Star (central node is just a switch, not an IMP).
   (b) Ring.
   (c) Complete interconnection.
   For each, give the number of hops needed for the average IMP-IMP packet (no self traffic).

4. A collection of five IMPs are to be connected in a point-to-point subnet. Between each pair of IMPs, the designers may put a high-speed line, a medium-speed line, a low-speed line, or no line. If it takes 100 ms of computer time to generate and inspect each topology, how long will it take to inspect all of them?

5. A group of $2^n - 1$ IMPs are interconnected in a centralized binary tree, with an IMP at each tree node. IMP i communicates with IMP j by sending a message to the root of the tree. The root then sends the message back down to j. Derive an approximate expression for the mean number of hops per message for large n, assuming that all IMP pairs are equally likely.

6. A disadvantage of a broadcast subnet is the capacity wasted due to multiple hosts attempting to access the channel at the same time. As a simplistic example, suppose that time is divided into discrete slots, with each of the n hosts attempting to use the channel with probability p during each slot. What fraction of the slots are wasted due to collisions?

7. The president of the Specialty Paint Corp. gets the idea to work together with a local beer brewer for the purpose of producing an invisible beer can (as an anti-litter measure). The president tells his legal department to look into it, and they in turn ask engineering for help. As a result, the chief engineer calls his counterpart at the other company to discuss the technical aspects of the project. The engineers then report back to their respective legal departments, which then confer by telephone to arrange the legal aspects. Finally the two corporate presidents discuss the financial side of the deal. Is this an example of a multilayer protocol in the sense of the OSI model?

8. Redraw Fig. 1-6 for a system in which layer 3, not layer 4, splits up the messages.

9. In most networks, the data link layer handles transmission errors by requesting damaged frames to be retransmitted. If the probability of a frame being damaged is p, what is the mean number of transmissions required to send a frame if acknowledgements are never lost?

10. Which of the OSI layers handles each of the following:
(a) Breaking the transmitted bit stream into frames.
(b) Determining which route through the subnet to use.
(c) Providing synchronization.

11. In the OSI model, do TPDUs encapsulate packets or the other way around? Discuss.

12. What are the SAP addresses in FM radio broadcasting?

13. What is the principal difference between connectionless communication and connection-oriented communication?

14. Two networks both provide reliable connection-oriented service. One of them offers a reliable byte stream and the other offers a reliable message stream. Are these identical? If so, why is the distinction made? If not, give an example of how they differ.

15. What is the difference between a confirmed service and an unconfirmed service? For each of the following, tell whether it might be a confirmed service, and unconfirmed service, both or neither.
(a) Connection establishment.
(b) Data transmission.
(c) Connection release.

16. What does "negotiation" mean when discussing network protocols? Give an example of it.

17. List two advantages and two disadvantages of having international standards for network protocols.

18. When transferring a file between two computers, (at least) two acknowledgement strategies are possible. In the first one, the file is chopped up into packets, which are individually acknowledged by the receiver, but the file transfer as a whole is not acknowledged. In the second one, the packets are not acknowledged individually, but the entire file is acknowledged when it arrives. Discuss these two approaches.

19. Why is a virtual terminal protocol needed? What kinds of applications are likely to use it?

20. Ethernet is sometimes said to be inappropriate for real time computing because the worst case retransmission interval is not bounded. Under what circumstances can the same argument be leveled at the token ring? Under what circumstances does the token ring have a known worst case? Assume the number of stations on the token ring is fixed and known.

5

THE NETWORK LAYER

The network layer is concerned with getting packets from the source all the way to the destination. Getting to the destination may require making many hops at intermediate nodes along the way. This function clearly contrasts with that of the data link layer, which has the more modest goal of just moving frames from one end of a wire to the other. Thus the network layer is the lowest layer that deals with end-to-end transmission.

In order to achieve its goals, the network layer must know about the topology of the communication subnet and choose appropriate paths through it. It must also take care to choose routes to avoid overloading some of the communication lines while leaving others idle. Finally, when the source and destination are in different networks, it is up to the network layer to deal with these differences and solve the problems that result from them.

5.1. NETWORK LAYER DESIGN ISSUES

In this section we will provide an introduction to some of the issues that the designers of the network layer must grapple with. These issues include the service provided to the transport layer, routing of packets through the subnet, congestion control, and connection of multiple networks together (internetworking). We will examine the latter three subjects in greater detail later in this chapter.

5.1.1. Services Provided to the Transport Layer

The network layer provides services to the transport layer. Since in some networks (e.g., the ARPANET and X.25 networks), the network layer runs in the IMPs and the transport layer runs in the hosts, the boundary between the network and transport layers in these networks is also the boundary between the subnet and the hosts. This means that the services provided by the network layer define the services provided by the subnet.

When the subnet is run by a common carrier or PTT and the hosts are run by the users, the network layer service becomes the interface between the carrier and the users. As such, it defines the carrier's duties and responsibilities, and is thus of great importance to both the carrier and the users. As you might expect under these circumstances, there has been a fair amount of discussion and controversy over the services that should be offered. We will look at this controversy shortly.

The network layer services have been designed with the following goals in mind.

1. The services should be independent of the subnet technology.
2. The transport layer should be shielded from the number, type, and topology of the subnets present.
3. The network addresses made available to the transport layer should use a uniform numbering plan even across LANs and WANs.

Given these goals, the designers of the network layer had a lot of freedom in writing detailed specifications of the services to be offered to the transport layer. This freedom quickly degenerated into a raging battle between two warring factions. The discussion centered on the question of whether the network layer should provide connection-oriented service or connectionless service.

This same issue occurs in the data link layer, as we saw in the previous chapter, but it is not as serious there. The LLC definition provides both, and since LANs are always owned and operated by the users, they can use whichever they prefer. With the network layer, the situation is different. If the carrier only offers one type of service and the users want the other, there is often no alternative.

One camp (represented by the ARPA Internet community) argues that the subnet's job is moving bits around and nothing else. In their view (based on nearly 20 years of actual experience with a real, working network), the subnet is inherently unreliable, no matter how it is designed. Therefore, the hosts should accept the fact that it is unreliable and do error control (i.e., error detection and correction) and flow control themselves.

This viewpoint leads quickly to the conclusion that the network service should be connectionless, with primitives *SEND PACKET* and *RECEIVE PACKET*, and little else. In particular, no error checking and flow control should be done, because the hosts are going to do that any way, and there is probably little to be gained by doing it twice. Furthermore, each packet must carry the full destination address, because each packet sent is carried independently of its predecessors, if any.

The other camp (represented by the carriers) argues that the network layer (and subnet) should provide a reliable, connection-oriented service with connections having the following properties:

1. Before sending data, the source transport entity must set up a connection to the destination transport entity. This connection, which is given a special identifier, is then used until there are no more data to send, at which time the connection is explicitly released.
2. When a connection is set up, the two transport entities and the network layer providing the service can enter into a negotiation about the parameters of the service and the quality and cost of the service to be provided.
3. Communication is in both directions, and packets are delivered in sequence without errors. The conceptual model behind this is a well-behaved queue with the first-in, first-out property.
4. Flow control is provided automatically to prevent a fast sender from dumping packets into the queue at a higher rate than the receiver can take them out, thus leading to overflow.

Other properties, such as explicit confirmation of delivery and high priority packets are optional.

An analogy between connection-oriented service and connectionless service may clarify the situation. The public telephone network offers a connection-oriented service. The customer first dials a number to set up a connection. Then the parties talk (exchange data). Finally the connection is broken. Although what happens inside the telephone system is undoubtedly very complicated, the two users are presented with the illusion of a dedicated, point-to-point channel that always delivers information in the order it was sent.

In contrast, the postal system (or telegraph system) is connectionless. Each letter carries the full destination address, and is carried independently of every other letter. Letters do not necessarily arrive in the order mailed. If a letter carrier accidentally drops a letter, the post office does not time out and send a duplicate. In short, error and flow control are handled by the users themselves, outside the postal system (subnet). Figure 5-1 summarizes the differences between connection-oriented and connectionless service.

The argument between connection-oriented and connectionless service really has to do with where to put the complexity. In the connection-oriented service, it is in the network layer (subnet); in the connectionless service, it is in the transport

Issue	Connection-oriented	Connectionless
Initial setup	Required	Not possible
Destination address	Only needed during setup	Needed on every packet
Packet sequencing	Guaranteed	Not guaranteed
Error control	Done by network layer (e.g., subnet)	Done by transport layer (e.g., hosts)
Flow control	Provided by network layer	Not provided by network layer
Is option negotiation possible?	Yes	No
Are connection identifiers used?	Yes	No

Fig. 5-1. Summary of the major differences between connection-oriented and connectionless service.

layer (hosts). Supporters of connectionless service say that user computing power has become cheap, so that there is no reason not to put the complexity in the hosts (frequently in a special network coprocessor chip). Furthermore, they argue that the subnet is a major national investment that will last for decades, so it should not be cluttered up with features that may become obsolete quickly, but will have to be calculated into the price structure for many years. Furthermore, some applications, such as digitized voice and real-time data collection may regard *speedy* delivery as much more important than *accurate* delivery.

On the other hand, supporters of connection-oriented service say that most users are not interested in running complex transport layer protocols in their machines. What they want is reliable, trouble-free service, and this service can be best provided with connections.

Finally, there is the very real issue of credibility. If the users are not willing to trust the carrier's claims that the subnet only loses a packet once in a blue moon, they will implement all the error checking anyway in the transport layer to protect themselves. However, if enough people do this, then it is wasteful to go to a lot of trouble and expense in the network layer to achieve high reliability. The subnet might just as well provide a cheap, bare-bones connectionless service and tell the users to do all the work, since many of them will do it anyway, no matter what the carrier tells them.

To make a long story short, the connection-oriented supporters outnumbered the connectionless supporters in the committee that wrote the network layer service definition, so the OSI network service was originally a connection-oriented service. (It has been said that the PTTs preferred connection-oriented service because they could not charge for connect time if there were no connections.) However, the people in favor of connectionless service kept lobbying for their cause, and ISO eventually modified the service definition to include both classes of service. Both types are now permitted and protocols for supporting both types have been incorporated into the OSI framework.

The network layer is not the only battleground between the two camps. The same issues arise in all the layers. The fight was finally resolved for all time by a revision to the OSI model to allow both kinds of service to be offered all the way up to the top. In Fig. 5-2 we see how the two service types relate to each other. At the top of each layer (except the application layer) are SAPs (Service Access Points) through which the layer above accesses the services. Each SAP has a unique address that identifies it. Starting with the data link layer, these services can be connection-oriented or connectionless (the issue is moot in the physical layer, since error control and flow control are not relevant there).

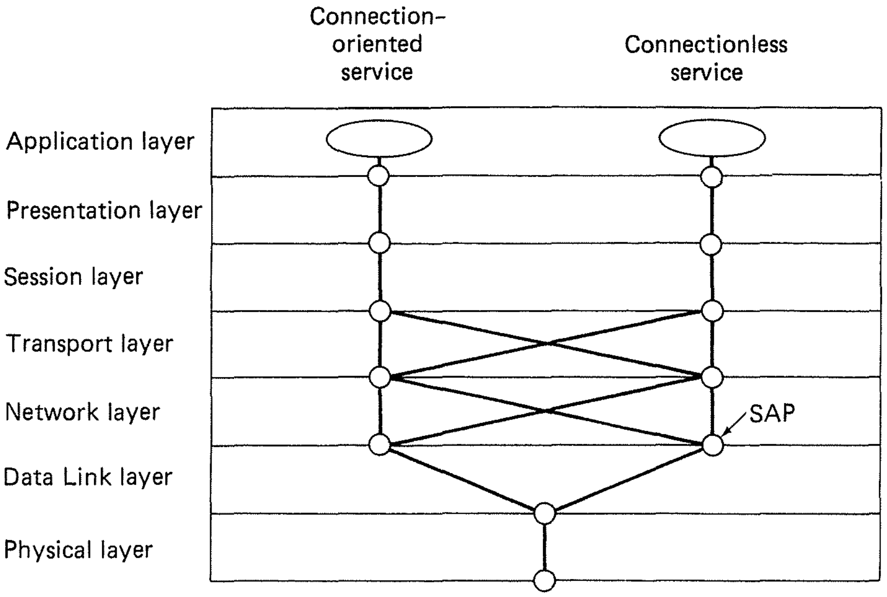


Fig. 5-2. Mixtures of connection-oriented and connectionless service in the OSI model.

Two obvious paths are fully connection-oriented service from top to bottom and fully connectionless service. However, it is also possible to have connection-oriented service be provided by the network or transport layers even though the lower layers are connectionless. In this case, the network or transport layers must handle the conversion. For example, a connection-oriented transport service could be built on a LAN with a connectionless data link service by putting the error control, flow control, and related functionality in either the network or transport layer.

The reverse mechanism, implementing a connectionless service for the upper layers on top of a connection-oriented service is also possible. Although this seems wasteful, it might be useful when connecting two fully connectionless LANs over a WAN that only offered connection-oriented network service. In any event, the conversion between the two service types can be done in either the network or transport layer, but not in higher layers.

The OSI Network Service Primitives

International Standard 8348 defines the network service by specifying the primitives that apply at the boundary between the network layer and transport layer. Both connection-oriented and connectionless primitives are provided.

The connection-oriented primitives use the model of Fig. 5-3. In this model, a connection is a pair of conceptual queues between two **NSAP**s (network addresses), one queue for traffic in each direction. Prior to establishing a connection, we have the situation of Fig. 5-3(a). After a successful establishment, we have the situation of Fig. 5-3(b). Finally, after three packets have been sent, the state of the connections might look like Fig. 5-3(c).

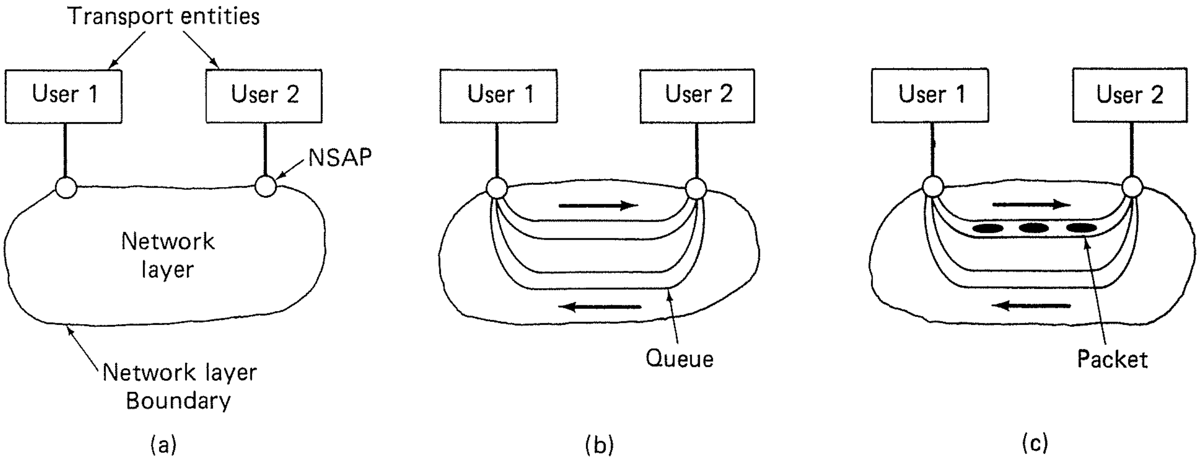


Fig. 5-3. (a) Prior to establishing a connection. (b) After establishing a connection. (c) After three packets have been sent but not yet received.

The OSI connection-oriented network service primitives are listed in Fig. 5-4(a). They can be grouped into four categories, for establishing, releasing, using, and resetting connections, respectively. Most of the primitives have parameters. The exact way the parameters are passed to the primitives is implementation dependent. The effect of each primitive can be described by the way it changes the state of the queues in Fig. 5-3.

The *N-CONNECT.request* primitive is used to set up a connection. It specifies the network address to connect to and the caller's network address. It also contains two Boolean variables used to request optional services. *Acks_wanted* is used to permit the caller to request acknowledgement of each packet sent. If the network

N-CONNECT. request (callee, caller, acks_wanted, exp_wanted, qos, user_data)
N-CONNECT. indication (callee, caller, acks_wanted, exp_wanted, qos, user_data)
N-CONNECT. response (responder, acks_wanted, exp_wanted, qos, user_data)
N-CONNECT. confirm (responder, acks_wanted, exp_wanted, qos, user_data)
N-DISCONNECT. request (originator, reason, user_data, responding_address)
N-DISCONNECT. indication (originator, reason, user_data, responding_address)
N-DATA. request (user_data)
N-DATA. indication (user_data)
N-DATA-ACKNOWLEDGE. request ( )
N-DATA-ACKNOWLEDGE. indication ( )
N-EXPEDITED-DATA. request (user_data)
N-EXPEDITED-DATA. indication (user_data)
N-RESET. request (originator, reason)
N-RESET. indication (originator, reason)
N-RESET. response ( )
N-RESET. confirm ( )

(a)

N-UNITDATA. request (source_address, destination_address, qos, user_data)
N-UNITDATA. indication (source_address, destination_address, qos, user_data)
N-FACILITY. request (qos)
N-FACILITY. indication (destination_address, qos, reason)
N-REPORT. indication (destination_address, qos, reason)

(b)

Notes on terminology:

Callee: Network address (NSAP) to be called
Caller: Network address (NSAP) used by calling transport entity
Acks_wanted: Boolean flag specifying whether acknowledgements are desired
Exp_wanted: Boolean flag specifying whether expedited data will be sent
Qos: Quality of service desired
User_data: 0 or more bytes of data transmitted but not examined
Responder: Network address (NSAP) connected to at the destination
Originator: Sepcification of who initiated the N-RESET
Reason: Specification of why the event happened

Fig. 5-4. (a) OSI connection-oriented network service primitives. (b) OSI connectionless network service primitives.

layer does not provide acknowledgements, the variable is set to *false* when delivered to the destination in the *N-CONNECT.indication* primitive. If the network layer does provide acknowledgements, but the destination does not want to use them, then it sets the flag to *false* in its *N-CONNECT.response*. Only if both transport entities and the network service provider want to use them are they used. This feature is an example of **option negotiation**.

The *exp_wanted* flag is a second example of option negotiation. If accepted by all three parties, it permits the use of **expedited data**, essentially packets may violate the normal queue ordering and skip to the head of the queue. Whether or not they actually do this, is implementation dependent. A typical example of expedited data is a user at a terminal hitting the DEL key to interrupt a running program. The DEL packet will go as expedited data.

The *qos* parameter is actually two lists of values that determine the **quality of service** provided by the connection. The first list gives the goal—what the caller really wants. The second list gives the minimum values considered acceptable. If the network service is unable to provide at least the minimum value for any parameter specified by either the caller (calling party) or the callee (called party), the connection establishment fails. The values that may be specified include throughput, delay, error rate, secrecy, and cost, among others.

The caller may include some user data in the connection request. The callee may inspect these data before deciding whether to accept or reject the request. Connection requests are accepted with the *N-CONNECT.response* primitive and rejected with the *N-DISCONNECT.request* primitive. When a request is rejected, the *reason* field allows the callee to tell why it was not willing to accept the connection and whether the condition is permanent or transient. The network layer itself may also reject attempts to establish connections, for example, if the quality of service desired is not available (permanent condition) or the subnet is currently overloaded (transient condition).

The remaining *N-CONNECT* primitives and the *N-DISCONNECT* primitives are straightforward, and need little further comment. After a connection has been established, either party can transmit data using *N-DATA.request*. When these packets arrive, an *N-DATA.indication* primitive is invoked on the receiving end. Expedited data uses primitives analogous to those for regular data.

If acknowledgements have been agreed upon, when a packet has been received, the recipient is expected to issue an *N-DATA-ACKNOWLEDGE.request*. This primitive contains no sequence number, so the party sending the original data must simply count acknowledgements. If the quality of service is low, and data and acknowledgements can be lost, this scheme is not very satisfactory. On the other hand, it is not the task of the network layer to provide an error-free service; that job is done by the transport layer. Network layer acknowledgements are merely an attempt to improve the quality of service, not make it perfect.

The *N-RESET* primitives are used to report catastrophes, such as crashes of either transport entity or the network service provider itself. After an *N-RESET* has been requested, indicated, responded to, and confirmed, the queues will be reset to their original empty state. Data present in the queues at the time of the *N-RESET* are lost. Again here, it is the job of the transport layer to recover from *N-RESET*s.

The OSI connectionless primitives are given in Fig. 5-4(b). The *N-UNITDATA* primitives are used to send up to 64,512 bytes of data (1K less than 2^{16}, to provide plenty of room for various headers and still keep the final unit less than 2^{16} bytes). The *N-UNITDATA* primitives provide no error control, no flow control, and no other control. The sender just dumps the packet into the subnet and hopes for the best.

The *N-FACILITY.request* primitive is designed to allow a network service user inquire about average delivery characteristics to the specified destination, such as percent of packets being delivered. The *N-FACILITY.indication* primitive comes from the network layer itself, not from a remote transport entity.

The *N-REPORT.indication* primitive allows the network layer to report problems back to the network service user. If, for example, a particular destination is unavailable, that fact could be reported using this primitive. The details of how the primitive is used are network dependent and not defined in the standard.

One of the functions of the network layer is to provide a uniform naming for the transport layer to use. Ideally, all the network operators in the world should get together and agree on a single name space, so that each wire running from a subnet into an office or home would have a world-wide unique address. Unfortunately, this is not going to happen.

To make the best of the existing situation, the OSI network layer addressing has been designed to incorporate today's diversity of network addressing schemes. All the network service primitives use NSAP addresses for identifying the source and destination. The format of these NSAP addresses is shown in Fig. 5-5. Each NSAP address has three fields. The first one, the **AFI** (**Authority and Format Identifier**), identifies the type of address present in the third field, the **DSP** (**Domain Specific Part**). Codes have been allocated to allow the DSP field to contain packet network addresses, telephone numbers, ISDN numbers, telex numbers, and similar existing numbering schemes, both in binary and in packed decimal. The AFI field can take on values from 10 to 99, leaving plenty of room for future numbering plans.

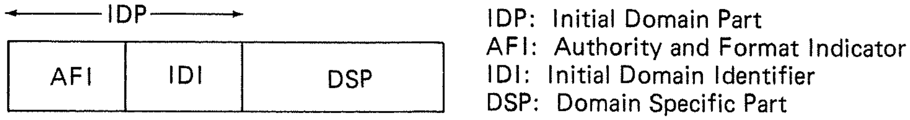

IDP
AFI
IDI
DSP
IDP: Initial Domain Part
AFI: Authority and Format Indicator
IDI: Initial Domain Identifier
DSP: Domain Specific Part

Fig. 5-5. The format of OSI network addresses (NSAPs).

The second, or **IDI** (**Initial Domain Identifier**) field, specifies the domain to which the number in the DSP part belongs. If, for example, the DSP is a telephone number, the IDI might be the country code for that number. The full NSAP address is variable length, up to 40 decimal digits or 20 bytes long.

To understand what NSAP addresses are for, it may be helpful to make an analogy with the telephone system. Most modern offices and homes nowadays are equipped with one or more sockets into which telephones can be plugged. Each socket has a wire running from it to a telephone switching office (or PBX). This socket is assigned a worldwide unique number consisting of a country code, area code, and subscriber number.

Whenever a telephone is plugged into a socket, the telephone can be reached by dialing the socket's number. Notice that the number really applies to the *socket*, not to the specific *telephone* currently plugged into it. When a telephone is moved from one office to another, it acquires the number of the new office. It does not take the old number with it. Thus the sockets give the telephone system a uniform name space, independent of which particular telephone that happens to be plugged in today, or which person will be answering it when it rings.

In this analogy, the telephone sockets are the NSAPs and the telephone numbers are the NSAP addresses. When a transport entity asks the network to make a call to a remote machine, it specifies the NSAP address (i.e., telephone number) to be called. The network layer does not care which transport entity (i.e., telephone) is currently attached (plugged in) to that NSAP, and certainly does not care which user owns the transport entity (i.e., which person will answer the telephone). How transport entities connect to NSAPs is an issue for the transport layer, not the network layer.

5.1.2. Internal Organization of the Network Layer

Having looked at the two classes of service the network layer can provide to its users, it is time to see how it works inside. Unfortunately, the OSI model does not provide a specification of the key algorithms, such as routing and congestion control. These are implementation dependent. Nevertheless, they are important and definitely worth studying in detail.

As a consequence, this chapter will cover many subnet design issues that are not strictly part of the network layer, although they are related to it. We will also discuss the OSI network layer where it is appropriate. To make it clear where we are discussing subnet design (as opposed to OSI), we will use the subnet terms "IMP" and "host" instead of "network layer" and "transport layer," although the latter are equivalent in some networks.

There are basically two different philosophies for organizing the subnet, one using connections and the other working connectionless. In the context of the *internal* operation of the subnet, a connection is usually called a **virtual circuit**, in analogy with the physical circuits set up by the telephone system. The independent packets of the connectionless organization are called **datagrams**, in analogy with telegrams.

Virtual circuits are generally used in subnets whose primary service is connection-oriented, so we will describe them in that context. The idea behind virtual circuits is to avoid having to make routing decisions for every packet sent. Instead, when a connection is established, a route from the source machine to the destination machine is chosen as part of the connection setup. That route is used for all traffic flowing over the connection, exactly the same way that the telephone system works. When the connection is released, the virtual circuit is discarded.

In contrast, with a datagram subnet no routes are worked out in advance, even if the service is connection-oriented. Each packet sent is routed independently of its predecessors. Successive packets may follow different routes. While datagram subnets have to do more work, they are also more robust and adapt to failures and congestion more easily than virtual circuit subnets. We will discuss the pros and cons of the two approaches later.

If packets flowing over a given virtual circuit always take the same route through the subnet, each IMP must remember where to forward packets for each of the currently open virtual circuits passing through it. Every IMP must maintain a table with one entry per open virtual circuit. Virtual circuits not passing through IMP *X* are not entered in *X*'s table, of course. Each packet traveling through the subnet must contain a virtual circuit number field in its header, in addition to sequence numbers, checksums, and the like. When a packet arrives at an IMP, the IMP knows on which line it arrived and what the virtual circuit number is. Based only on this information, the packet must be forwarded to the correct IMP.

When a network connection is set up, a virtual circuit number not already in use on that machine is chosen as the connection identifier. Since each machine chooses virtual circuit numbers independently, the same virtual circuit number is likely to be in use on two different paths through some intermediate IMP, leading to ambiguities.

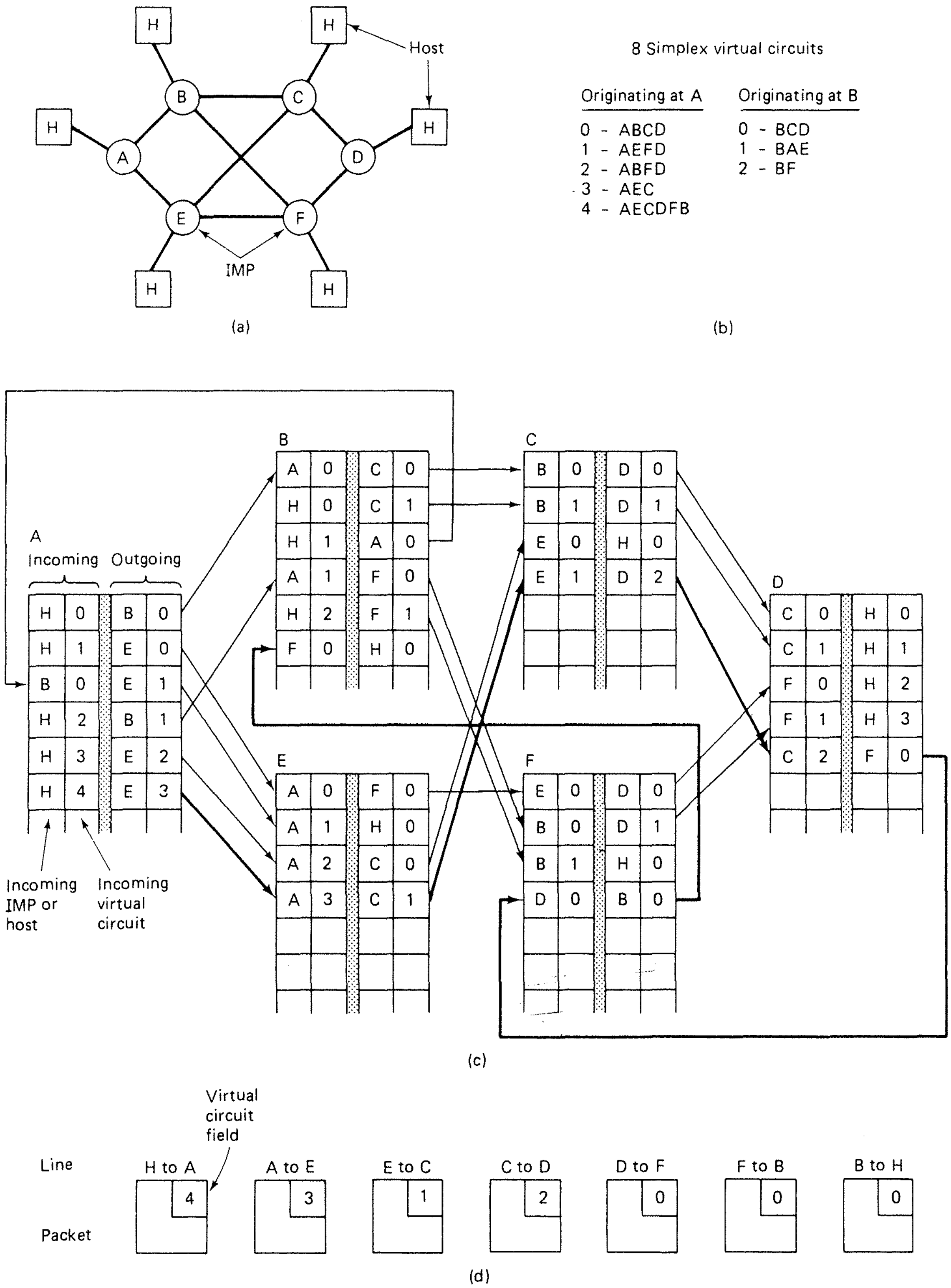
Consider the subnet of Fig. 5-6(a). Suppose that a process in *A*'s host wants to talk to a process in *D*'s host. *A* chooses virtual circuit 0. Let us assume that route *ABCD* is chosen. Simultaneously, a process in *B* decides it wants to talk to a process in *D* (not the same one as *A*). If there are no open virtual circuits starting in *B* at this point, host *B* will also choose virtual circuit 0. Further assume that route *BCD* is selected as the best one. After both virtual circuits have been set up, the process at *A* sends its first message to *D*, on virtual circuit 0. When the packet arrives at *D*, the poor host does not know which user process to give it to.

To solve this problem, whenever a host wants to create a new outbound virtual circuit, it chooses the lowest circuit number not currently in use. The IMP (say *X*) does not forward this setup packet to the next IMP (say *Y*) along the route as is. Instead, *X* looks in its table to find all the circuit numbers currently being used for traffic to *Y*. It then chooses the lowest free number and substitutes that number for the one in the packet, overwriting the number chosen by the host. Similarly, IMP *Y* chooses the lowest circuit number free between it and the next IMP.

When this setup packet finally arrives at the destination, the IMP there chooses the lowest available inbound circuit number, overwrites the circuit number found in the packet with this, and passes it to the host. As long as the destination host always sees the same circuit number on all traffic arriving on a given virtual circuit, it does not matter that the source host is consistently using a different number.

Figure 5-6(b) gives eight examples of virtual circuits pertaining to the subnet of part (a). Part (c) of the figure shows the IMPs' tables, assuming the circuits were created in the order: *ABCD*, *BCD*, *AEFD*, *BAE*, *ABFD*, *BF*, *AEC*, and *AECDFB*. The last one (*AECDFB*) may seem somewhat roundabout, but if lines *AB*, *BC*, and *EF* were badly overloaded when the routing algorithm ran, this might well have been the best choice.

Each entry consists of an incoming and an outgoing part. Each of the two parts



8 Simplex virtual circuits

Originating at A	Originating at B
0 - ABCD	0 - BCD
1 - AEFD	1 - BAE
2 - ABFD	2 - BF
3 - AEC	
4 - AECDFB	

A

Incoming		Outgoing	
H	0	B	0
H	1	E	0
B	0	E	1
H	2	B	1
H	3	E	2
H	4	E	3

Incoming IMP or host

Incoming virtual circuit

B

A	0	C	0
H	0	C	1
H	1	A	0
A	1	F	0
H	2	F	1
F	0	H	0

C

B	0	D	0
B	1	D	1
E	0	H	0
E	1	D	2

D

C	0	H	0
C	1	H	1
F	0	H	2
F	1	H	3
C	2	F	0

E

A	0	F	0
A	1	H	0
A	2	C	0
A	3	C	1

F

E	0	D	0
B	0	D	1
B	1	H	0
D	0	B	0

Fig. 5-6. (a) Example subnet. (b) Eight virtual circuits through the subnet. (c) IMP tables for the virtual circuits in (b). (d) The virtual circuit changes as a packet progresses.

has an IMP name (used to indicate a line) and a virtual circuit number. When a packet arrives, the table is searched on the left (incoming) part, using the arrival line and virtual circuit number found in the packet as the key. When a match is found, the outgoing part of the entry tells which virtual circuit number to insert into the packet and which IMP to send it to. *H* stands for the host, both on the incoming and outgoing sides.

As an example, consider a packet traveling from host *A* to host *B* on virtual circuit 4 (i.e., route *AECDFB*). When IMP *A* gets a packet from its own host, with circuit 4, it searches its table, finding a match for *H*4 at entry 5 (the top entry is 0). The outgoing part of this entry is *E*3, which means replace the circuit 4 with circuit 3 and send it to IMP *E*. IMP *E* then gets a packet from *A* with circuit 3, so it searches for *A*3 and finds a match at the third entry. The packet now goes to *C* as circuit 1. The sequence of entries used is marked by the heavy line. Figure 5-6(d) shows this sequence of packet numbers.

Because virtual circuits can be initiated from either end, a problem occurs when call setups are propagating in both directions at once along a chain of IMPs. At some point they have arrived at adjacent IMPs. Each IMP must now pick a virtual circuit number to use for the (full-duplex) circuit it is trying to establish. If they have been programmed to choose the lowest number not already in use on the link, they will pick the same number, causing two unrelated virtual circuits over the same physical line to have the same number. When a data packet arrives later, the receiving IMP has no way of telling whether it is a forward packet on one circuit or a reverse packet on the other. If circuits are simplex, there is no ambiguity.

Note that every process must be required to indicate when it is through using a virtual circuit, so that the virtual circuit can be purged from the IMP tables to recover the space. In public networks the motivation is the stick rather than the carrot: users are invariably charged for connect time as well as for data transported.

So much for the use of virtual circuits internal to the subnet. The other possibility is to use datagrams internally, in which case the IMPs do not have a table with one entry for each open virtual circuit. Instead, they have a table telling which outgoing line to use for each possible destination IMP. These tables are also needed when virtual circuits are used internally, to determine the route for a setup packet.

Each datagram must contain the full destination address (the machine and the NSAP address to which the destination process is attached). When a packet comes in, the IMP looks up the outgoing line to use and sends it on its way. Nothing in the packet is modified. Also, the establishment and release of network or transport layer connections do not require any special work on the part of the IMPs.

Comparison of Virtual Circuits and Datagrams within the Subnet

Both virtual circuits and datagrams have their supporters and their detractors. We will now attempt to summarize the arguments both ways. Inside the subnet the main trade-off between virtual circuits and datagrams is between IMP memory space and bandwidth. Virtual circuits allow packets to contain circuit numbers instead of full destination addresses. If the packets tend to be fairly short, a full destination address in every packet may represent a significant amount of overhead, and hence wasted bandwidth. The use of virtual circuits internal to the subnet becomes especially attractive when many of the "hosts" are actually interactive terminals with only a few characters per packet. The price paid for using virtual circuits internally is the table space within the IMPs. Depending upon the relative cost of communication circuits versus IMP memory, one or the other may be cheaper.

For transaction processing systems (e.g., stores calling up to verify credit card purchases), the overhead required to set up and clear a virtual circuit may easily dwarf the use of the circuit. If the majority of the traffic is expected to be of this kind, the use of virtual circuits inside the subnet makes little sense.

Virtual circuits also have a vulnerability problem. If an IMP crashes and loses its memory, even if it comes back up a second later, all the virtual circuits passing through it will have to be aborted. In contrast, if a datagram IMP goes down, only those users whose packets were queued up in the IMP at the time will suffer, and maybe not even all those, depending upon whether they have already been acknowledged or not. The loss of a communication line is fatal to virtual circuits using it, but can be easily compensated for if datagrams are used. Datagrams also allow the IMPs to balance the traffic throughout the subnet, since routes can be changed halfway through a connection.

It is worth explicitly pointing out that the service offered (connection-oriented or connectionless) is a separate issue from the subnet structure (virtual circuit or datagram). In theory, all four combinations are possible. Obviously, a virtual circuit implementation of a connection-oriented service and a datagram implementation of a connectionless service are reasonable. Implementing connections using datagrams also makes sense when the subnet is trying to provide a highly robust service.

The fourth possibility, a connectionless service on top of a virtual circuit subnet seems strange, but might happen in a subnet originally designed for connection-oriented service, with connectionless service thrown in as an afterthought. In such an arrangement, the subnet might have to set up, use, and release a virtual circuit for each packet sent, not a pleasant thought.

Figure 5-7 summarizes some of the differences between subnets using datagrams internally and subnets using virtual circuits internally.

5.1.3. Routing

The real function of the network layer is routing packets from the source machine to the destination machine. In most subnets, packets will require multiple hops to make the journey. The only notable exception is for broadcast networks,

Issue	Datagram subnet	VC subnet
Circuit setup	Not possible	Required
Addressing	Each packet contains the full source and destination address	Each packet contains a short vc number
State information	Subnet does not hold state information	Each established vc requires subnet table space
Routing	Each packet is routed independently	Route chosen when vc is set up; all packets follow this route
Effect of node failures	None, except for packets lost during the crash	All vcs that passed through the failed equipment are terminated
Congestion control	Difficult	Easy if enough buffers can be allocated in advance for each vc set up
Complexity	In the transport layer	In the network layer
Suited for	Connection-oriented and connectionless service	Connection-oriented service

Fig. 5-7. Comparison of datagram and virtual circuit subnets. Notice the similarity with the connection-oriented and connectionless services of Fig. 5-1.

but even here routing is an issue if the source and destination are not on the same network. The algorithms that choose the routes and the data structures that they use are a major area of network layer design. In this section we will just outline the problem. Later in this chapter we will examine in detail many of the algorithms that have been proposed.

The **routing algorithm** is that part of the network layer software responsible for deciding which output line an incoming packet should be transmitted on. If the subnet uses datagrams internally, this decision must be made anew for every arriving data packet. However, if the subnet uses virtual circuits internally, routing decisions are made only when a new virtual circuit is being set up. Thereafter, data packets just follow the previously established route. The latter case is sometimes called **session routing**, because a route remains in force for an entire user session (e.g., a login session at a terminal or a file transfer).

Regardless of whether routes are chosen independently for each packet or only when new connections are established, there are certain properties that are desirable in a routing algorithm: correctness, simplicity, robustness, stability, fairness, and optimality. Correctness and simplicity hardly require comment, but the need for robustness may be less obvious at first. Once a major network comes on the air, it may be expected to run continuously for years without system-wide failures. During that period there will be hardware and software failures of all kinds. Hosts, IMPs, and lines will go up and down repeatedly, and the topology will change many times. The routing algorithm must be able to cope with changes in the topology and traffic without requiring all jobs in all hosts to be aborted and the network to be rebooted every time some IMP crashes.

Stability is also an important goal for the routing algorithm. There exist routing algorithms that never converge to equilibrium, no matter how long they run. Fairness and optimality may sound like Motherhood and Apple Pie—surely no one would oppose them—but as it turns out, they are often contradictory goals. As a simple example of this conflict, look at Fig. 5-8. Suppose that there is enough traffic between *A* and *A'*, between *B* and *B'*, and between *C* and *C'* to saturate the horizontal links. To maximize the total flow, the *X* to *X'* traffic should be shut off altogether. Unfortunately, *X* and *X'* may not see it that way. Evidently, some compromise between global efficiency and fairness to individual connections is needed.

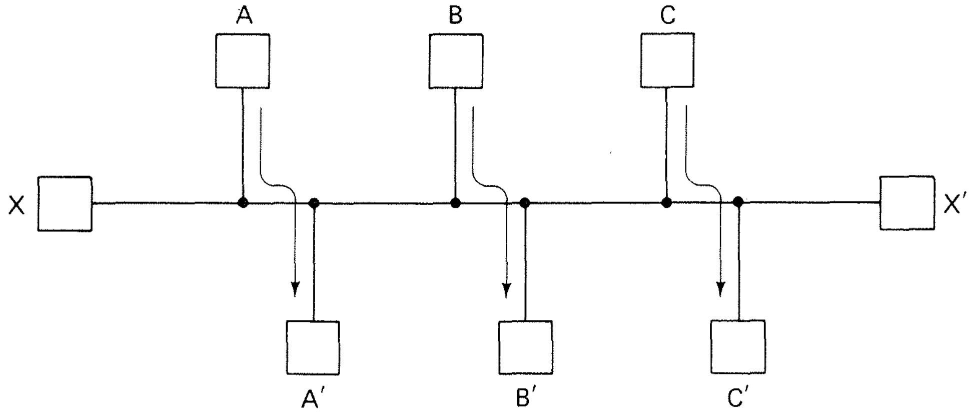


Fig. 5-8. Conflict between fairness and optimality.

Before we can even attempt to find trade-offs between fairness and optimality, we must decide what it is we seek to optimize. Minimizing mean packet delay is an obvious candidate, but so is maximizing total network throughput. Furthermore, these two goals are also in conflict, since operating any queueing system near capacity ($\rho \rightarrow 1$) implies a long queueing delay. As a compromise, many networks attempt to minimize the number of hops a packet must make, because reducing the number of hops tends to improve the delay and also reduce the amount of bandwidth consumed, which tends to improve the throughput as well.

Routing algorithms can be grouped into two major classes: nonadaptive and adaptive. Nonadaptive algorithms do not base their routing decisions on measurements or estimates of the current traffic and topology, whereas adaptive ones do. If an adaptive algorithm manages to adapt well to the traffic, it will naturally outperform an algorithm that is oblivious to what is going on in the network, but adapting well to the traffic is easier said than done. Adaptive algorithms can be further subdivided into centralized, isolated, and distributed (McQuillan, 1974), all of which are discussed in detail below.

5.1.4. Congestion

When too many packets are present in (a part of) the subnet, performance degrades. This situation is called **congestion**. Figure 5-9 depicts the symptom. When the number of packets dumped into the subnet by the hosts is within its carrying capacity, they are all delivered (except for a few that are afflicted with transmission errors), and the number delivered is proportional to the number sent. However, as traffic increases too far, the IMPs are no longer able to cope, and they begin losing packets. This tends to make matters worse. At very high traffic, performance collapses completely, and almost no packets are delivered.

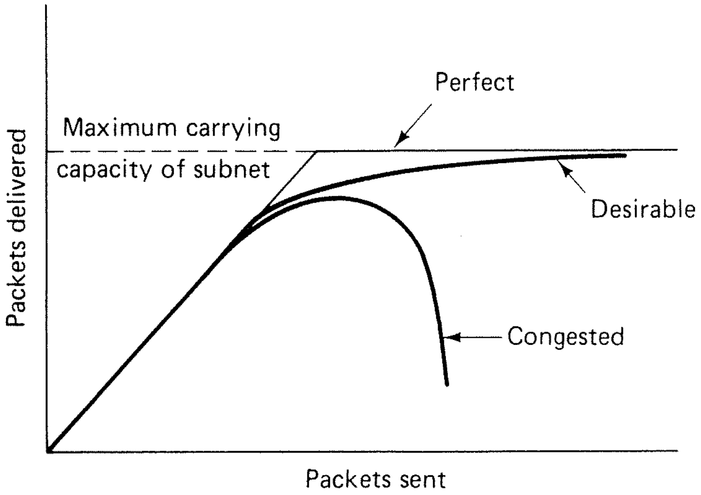


Fig. 5-9. When too much traffic is offered, congestion sets in, and performance degrades sharply.

Congestion can be brought about by several factors. If the IMPs are too slow to perform the various bookkeeping tasks required of them (queueing buffers, updating tables, etc.), queues can build up, even though there is excess line capacity. On the other hand, even if the IMP CPU is infinitely fast, queues will build up whenever the input traffic rate exceeds the capacity of the output lines. This can happen, for example, if three input lines are delivering packets at top speed, all of which need to be forwarded along the same output line. Either way, the problem boils down to not enough IMP buffers. Given an infinite supply of buffers, the IMP can always smooth over any temporary bottlenecks by just hanging onto all packets for as long as necessary. Of course, for stable operation, the hosts cannot indefinitely pump packets into the subnet at a rate higher than the subnet can absorb.

Congestion tends to feed upon itself and become worse. If an IMP has no free buffers, it must ignore newly arriving packets. When a packet is discarded, the IMP that sent the packet will time out and retransmit it, perhaps ultimately many times. Since the sending IMP cannot discard the packet until it has been acknowledged, congestion at the receiver's end forces the sender to refrain from releasing a buffer it would have normally freed. In this manner, congestion backs up, like cars approaching a toll booth.

It is definitely worth explicitly pointing out the difference between congestion control and flow control, as the authors of many books and papers on the subject do not seem to understand. Congestion control has to do with making sure the subnet is able to carry the offered traffic. It is a global issue, involving the behavior of all the hosts, all the IMPs, the store-and-forwarding processing within the IMPs, and all the other factors that tend to diminish the carrying capacity of the subnet.

Flow control, in contrast, relates to the point-to-point traffic between a given sender and a given receiver. Its job is to make sure that a fast sender cannot continually transmit data faster than the receiver can absorb it. Flow control nearly always involves some direct feedback from the receiver to the sender to tell the sender how things are doing at the other end.

To see the difference between these two concepts, consider a fiber optic network with a capacity of 1000 gigabits/sec on which a supercomputer was trying to transfer a file to a microcomputer at 100 Mbps. Although there would be no congestion at all (the network itself is not in trouble), flow control would be needed to force the supercomputer to stop frequently to give the microcomputer a chance to catch up.

At the other extreme, consider a store-and-forward network with 1-Mbps lines and 1000 large minicomputers, half of which were trying to transfer files at 100 kbps to the other half. Here the problem would not be that of fast senders overpowering slow receivers, but simply that the total offered traffic could easily exceed what the network could handle.

Later in this chapter we will examine congestion control in detail, and discuss various algorithms for dealing with it. It should be clear that congestion control and routing are closely related, with poor routing decisions being a major cause of congestion.

5.1.5. Internetworking

Just as congestion control is closely related to the primary function of the network layer, routing, so is internetworking. When the source and destination machines are in different networks, all the usual routing problems are present, only worse. For example, if the networks containing the source and destination machines are not directly connected, the routing algorithm will have to find a path through one or more intermediate networks. There may be many possible choices, all with different characteristics, advantages, and disadvantages.

Routing aside, another problem with internetworking is that not all networks use the same protocols. Different protocols imply different packet formats, headers, flow control procedures, acknowledgement rules, and more. As a consequence, when packets move from network to network, conversions are necessary. Sometimes they are straightforward, but often they are not. Just think about what happens when a packet must traverse both virtual circuit and datagram subnets on the way to the destination. Later in this chapter we will study internetworking, its problems and its solutions in detail.

5.2. ROUTING ALGORITHMS

Routing algorithms can be grouped into two major classes: nonadaptive and adaptive. **Nonadaptive algorithms** do not base their routing decisions on measurements or estimates of the current traffic and topology. Instead, the choice of the route to use to get from *i* to *j* (for all *i* and *j*) is computed in advance, off-line, and downloaded to the IMPs when the network is booted. This procedure is sometimes called **static routing**.

Adaptive algorithms, on the other hand, attempt to change their routing decisions to reflect changes in topology and the current traffic. Three different families of adaptive algorithms exist, differing in the information they use. The global algorithms use information collected from the entire subnet in an attempt to make optimal decisions. This approach is called centralized routing. The local algorithms run separately on each IMP and only use information available there, such as queue lengths. These are known as isolated algorithms. Finally, the third class of algorithms uses a mixture of global and local information. They are called distributed algorithms. All three classes are discussed in detail below. Additional information about routing can be found in Bell and Jabbour (1986).

5.2.1. Shortest Path Routing

Let us begin our study of routing algorithms with a technique that is widely used in many forms because it is simple and easy to understand. The idea is to build a graph of the subnet, with each node of the graph representing an IMP and each arc of the graph representing a communication line. To choose a route between a given pair of IMPs, the algorithm just finds the shortest path between them.

The concept **shortest path** deserves some explanation. One way of measuring path length is the number of hops. Using this metric, the paths *ABC* and *ABE* in Fig. 5-10 are equally long. Another metric is the geographic distance in kilometers, in which case *ABC* is clearly much longer than *ABE* (assuming the figure is drawn to scale).

However, many other metrics are also possible. For example, each arc could be labeled with the mean queueing and transmission delay for a standard test packet as determined by hourly or daily test runs. With this graph labeling, the shortest path is the fastest path, rather than the path with the fewest arcs or kilometers.

In the most general case, the labels on the arcs could be computed as a function of the distance, bandwidth, average traffic, communication cost, mean queue

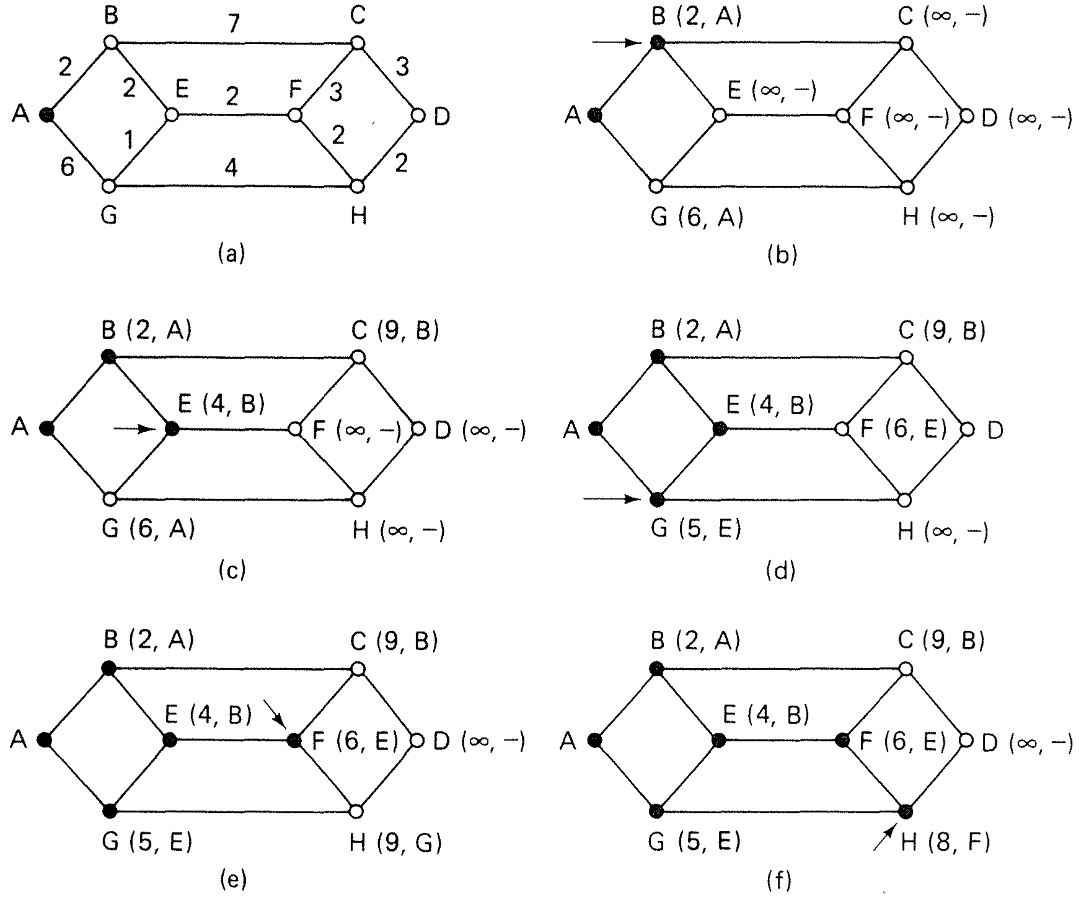


Fig. 5-10. The first five steps used in computing the shortest path from A to D. The arrows indicate the working node.

length, measured delay, and other factors. By changing the weighting function, the algorithm would then compute the "shortest" path according to any one of a number of criteria.

Several algorithms for computing the shortest path between two nodes of a graph are known. This one is due to Dijkstra (1959). Each node is labeled (in parentheses) with its distance from the source node along the best known path. Initially, no paths are known, so all nodes are labeled with infinity. As the algorithm proceeds and paths are found, the labels may change, reflecting better paths. A label may be either tentative or permanent. Initially, all labels are tentative. When it is discovered that a label represents the shortest possible path from the source to that node, it is made permanent and never changed thereafter.

To illustrate how the labeling algorithm works, look at the weighted, undirected graph of Fig. 5-10(a), where the weights represent, for example, distance. We want to find the shortest path from A to D. We start out by marking node A as permanent, indicated by a filled in circle. Then we examine, in turn, each of the nodes adjacent to A (the working node), relabeling each one with the distance to A. Whenever a node is relabeled, we also label it with the node from which the probe was made, so we can reconstruct the final path later. Having examined each of the nodes adjacent to *A*, we examine all the tentatively labeled nodes in the whole graph, and make the one with the smallest label permanent, as shown in Fig. 5-10(b). This one becomes the new working node.

We now start at *B*, and examine all nodes adjacent to it. If the sum of the label on *B* and the distance from *B* to the node being considered is less than the label on that node, we have a shorter path, so the node is relabeled.

After all the nodes adjacent to the working node have been inspected, and the tentative labels changed if possible, the entire graph is searched for the tentatively labeled node with the smallest value. This node is made permanent, and becomes the working node for the next round. Figure 5-10 shows the first five steps of the algorithm.

To see why the algorithm works, look at Fig. 5-10(c). At that point we have just made *E* permanent. Suppose that there were a shorter path than *ABE*, say *AXYZE*. There are two possibilities: either node *Z* has already been made permanent, or it has not been. If it has, then *E* has already been probed (on the round following the one when *Z* was made permanent), so the *AXYZE* path has not escaped our attention.

Now consider the case where *Z* is still tentatively labeled. Either the label at *Z* is greater than or equal to that at *E*, in which case *AXYZE* cannot be a shorter path than *ABE*, or it is less than that of *E*, in which case *Z* and not *E* will become permanent first, allowing *E* to be probed from *Z*.

This algorithm is given in Pascal in Fig. 5-11. The only difference between the program and the algorithm described above is that in Fig. 5-11, we compute the shortest path starting at the terminal node, *t*, rather than at the source node, *s*. Since the shortest path from *t* to *s* in an undirected graph is the same as the shortest path from *s* to *t*, it does not matter at which end we begin (unless there are several shortest paths, in which case reversing the search might discover a different one). The reason for searching backwards is that each node is labeled with its predecessor rather than its successor. When copying the final path into the output variable, *path*, the path is thus reversed. By reversing the search, the two effects cancel, and the answer is produced in the correct order.

5.2.2. Multipath Routing

So far we have tacitly assumed that there is a single "best" path between any pair of nodes and that all traffic between them should use it. In many networks, there are several paths between pairs of nodes that are almost equally good. Better performance can frequently be obtained by splitting the traffic over several paths, to reduce the load on each of the communication lines. The technique of using multiple routes between a single pair of nodes is called **multipath routing** or sometimes **bifurcated routing**.

Multipath routing is applicable both to datagram subnets and virtual circuit subnets. For datagram subnets, when a packet arrives at an IMP for forwarding, a

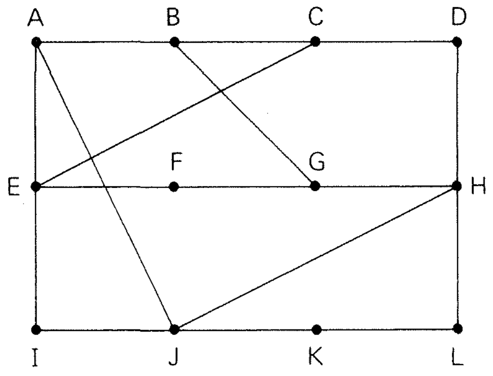
```
{Find the shortest path from the source to the sink of a given graph.}

const n = ... ;                   {number of nodes}
      infinity = ... ;            {a number larger than any possible path length}

type node = 0 .. n;
     nodelist = array [1 .. n] of node;
     matrix = array [1 .. n, 1 .. n] of integer;

procedure ShortestPath(a: matrix; s,t: node;  var path: nodelist);
{Find the shortest path from s to t in the matrix a, and return it in path.}

type lab = (perm, tent);          {is label tentative or permanent?}
     NodeLabel = record predecessor: node; length: integer; labl: lab end;
     GraphState = array[1 .. n] of NodeLabel;

var state: GraphState;  i,k: node;  min: integer;
begin                             {initialize}
  for i := 1 to n do
    with state[i] do
      begin predecessor := 0; length := infinity; labl := tent end;
  state[t].length := 0;   state[t].labl := perm;
  k := t;                         {k is the initial working node}

  repeat  {is there a better path from k?}
    for i := 1 to n do
      if (a[k,i] <> 0) and (state[i].labl = tent) then      {i is adjacent & tent.}
        if state[k].length + a[k,i] < state[i].length then
          begin
            state[i].predecessor := k;
            state[i].length := state[k].length + a[k,i]
          end;

    {Find the tentatively labeled node with the smallest label.}
    min := infinity;   k := 0;
    for i := 1 to n do
      if (state[i].labl = tent) and (state[i].length < min) then
        begin
          min := state[i].length;
          k := i                  {unless superseded, k will be next working node}
        end;
    state[k].labl := perm

  until k = s;                    {repeat until we reach the source}

  {Copy the path into the output array.}
  k := s;   i := 0;
  repeat
    i := i + 1;
    path[i] := k;
    k := state[k].predecessor;
  until k = 0
end; {ShortestPath}
```

Fig. 5-11. A procedure to compute the shortest path through a graph.

choice is made among the various alternatives for that packet, independent of the choices made for other packets to that same destination in the past. For virtual circuit subnets, whenever a virtual circuit is set up, a route is chosen, but different virtual circuits (on behalf of different users) are routed independently.

Multipath routing can be implemented as follows. Each IMP maintains a table with one row for each possible destination IMP. A row gives the best, second best, third best, etc. outgoing line for that destination, together with a relative weight. Before forwarding a packet, an IMP generates a random number and then chooses among the alternatives, using the weights as probabilities. The tables are worked out manually by the network operators, loaded into the IMPs before the network is brought up, and not changed thereafter.

As an example, consider the subnet of Fig. 5-12(a). IMP *J*'s routing table is given in Fig. 5-12(b). If *J* receives a packet whose destination is *A*, it uses the row labeled *A*. Here three choices are presented. The line to *A* is the first choice, followed by the lines to *I* and *H* respectively. To decide, *J* generates a random number between 0.00 and 0.99. If the number is below 0.63, line *A* is used; if the number is between 0.63 and 0.83, *I* is used; otherwise, *H* is used. The three weights are therefore the respective probabilities that *A*, *I*, or *H* will be used.



(a)

Destination	First choice		Second choice		Third choice	
A	A	0.63	I	0.21	H	0.16
B	A	0.46	H	0.31	I	0.23
C	A	0.34	I	0.33	H	0.33
D	H	0.50	A	0.25	I	0.25
E	A	0.40	I	0.40	H	0.20
F	A	0.34	H	0.33	I	0.33
G	H	0.46	A	0.31	K	0.23
H	H	0.63	K	0.21	A	0.16
I	I	0.65	A	0.22	H	0.13
–						
K	K	0.67	H	0.22	A	0.11
L	K	0.42	H	0.42	A	0.16

(b)

Fig. 5-12. (a) An example subnet. (b) Routing table for node *J*.

An advantage of multipath routing over shortest path routing is the possibility of sending different classes of traffic over different paths. For example, a connection between a terminal and a remote computer that consists of short packets that must be delivered quickly could be routed via terrestrial lines, whereas a long file transfer requiring high bandwidth could go via a satellite link. Not only does this method give the file transfer the high bandwidth it needs, but it also prevents short terminal packets from being delayed behind a queue of long file transfer packets.

Although multipath routing is widely used to improve performance, it can also be used to improve the reliability of the subnet. In particular, if the routing tables contain n disjoint routes between each pair of IMPs, then the subnet can withstand the loss of $n - 1$ lines without being broken into two parts.

One simple way to make sure all the alternative routes are disjoint is to first compute the shortest path between the source and destination. Then remove from the graph all the nodes and arcs used on the shortest path and compute the shortest path through the new graph. This algorithm insures that IMP or line failures on the first path will not also bring the second one down. By removing the second path from the graph as well, we can compute a third path that is completely independent of the first two. Even (1975) has devised a more sophisticated algorithm for finding disjoint paths in a graph.

5.2.3. Centralized Routing

The routing algorithms discussed above all require information about the network topology and traffic to make good decisions. If the topology is static and the traffic rarely changes, it is straightforward to build the routing tables once and for all time off-line and download them into the IMPs.

However, if IMPs and lines go down and come back up, or the traffic varies wildly throughout the day, some mechanism is needed to adapt the tables to the current circumstances. In this section we will discuss techniques for building the routing tables in a central location. In the succeeding ones we will see how this job can be done in a totally or partially decentralized fashion.

When centralized routing is used, somewhere within the network there is an **RCC** (**Routing Control Center**). Periodically, each IMP sends status information to the RCC (e.g., a list of its neighbors that are up, current queue lengths, amount of traffic processed per line since the last report, etc.). The RCC collects all this information, and then, based upon its global knowledge of the entire network, computes the optimal routes from every IMP to every other IMP, for example using the shortest path algorithm discussed above. From this information it can build new routing tables and distribute them to all the IMPs.

At first glance centralized routing is attractive: since the RCC has complete information, it can make perfect decisions. Another advantage is that it relieves the IMPs of the burden of the routing computation.

Unfortunately, centralized routing also has some serious, if not fatal, drawbacks. For one thing, if the subnet is to adapt to changing traffic, the routing calculation will have to be performed fairly often. For a large network, the calculation will take many seconds, even on a substantial CPU. If the purpose of running the algorithm is to adapt to changes in the topology rather than changes in the traffic, however, running it every minute or so may be adequate, depending on how stable the topology is.

A more serious problem is the vulnerability of the RCC. If it goes down or becomes isolated by line failures, the subnet is suddenly in trouble. One solution is to have a second machine available as a backup, but this amounts to wasting a large computer. An arbitration method is also needed to make sure that the primary and backup RCCs do not get into a fight over who is the boss.

Yet another problem with centralized routing concerns distributing the routing tables to the IMPs. The IMPs that are close to the RCC will get their new tables first and will switch over to the new routes before the distant IMPs have received their tables. Inconsistencies may arise here, so packets may be delayed. Among the packets delayed will be the new routing tables for the distant IMPs, so the problem feeds upon itself.

If the RCC computes the optimal route for each pair of IMPs and no alternates, the loss of even a single line or IMP will probably cut some IMPs off from the RCC, with disastrous consequences. If the RCC does use alternate routing, the argument in favor of having an RCC in the first place, namely that it can find the optimal routes, is weakened.

A final problem with centralized routing is the heavy concentration of routing traffic on the lines leading into the RCC. Figure 5-13 illustrates this problem. The figure was drawn by tracing the shortest path from each machine to the RCC, and placing an arrow on each line. A line with *n* arrows means that *n* IMPs are reporting to the RCC via it. The heavy load and consequent vulnerability of lines near the RCC is apparent.

As an example of how centralized routing works in practice, consider TYMNET, a commercial packet-switching network with over 1000 nodes that has been running since 1971. TYMNET is primarily used to allow terminals to log into remote computers, so the subnet offers connection-oriented service and uses virtual circuits to implement this service. The TYMNET IMPs periodically send the RCC information about their status: lines that are up or down, queue lengths, and other statistics. The RCC maintains tables keeping track of all this incoming information.

When a new user logs in and specifies which host he wants to connect to, a packet is sent to the RCC informing it of the login. The RCC then computes the best route, using all the information at its disposal. It then sends a **needle packet** back to the IMP to which the user is connected. The needle packet contains the route chosen by the RCC. This packet then threads its way through the subnet, making entries in the IMPs tables as it goes, and thus setting up the virtual circuit. When the user logs out, a similar process is used to release the virtual circuit.

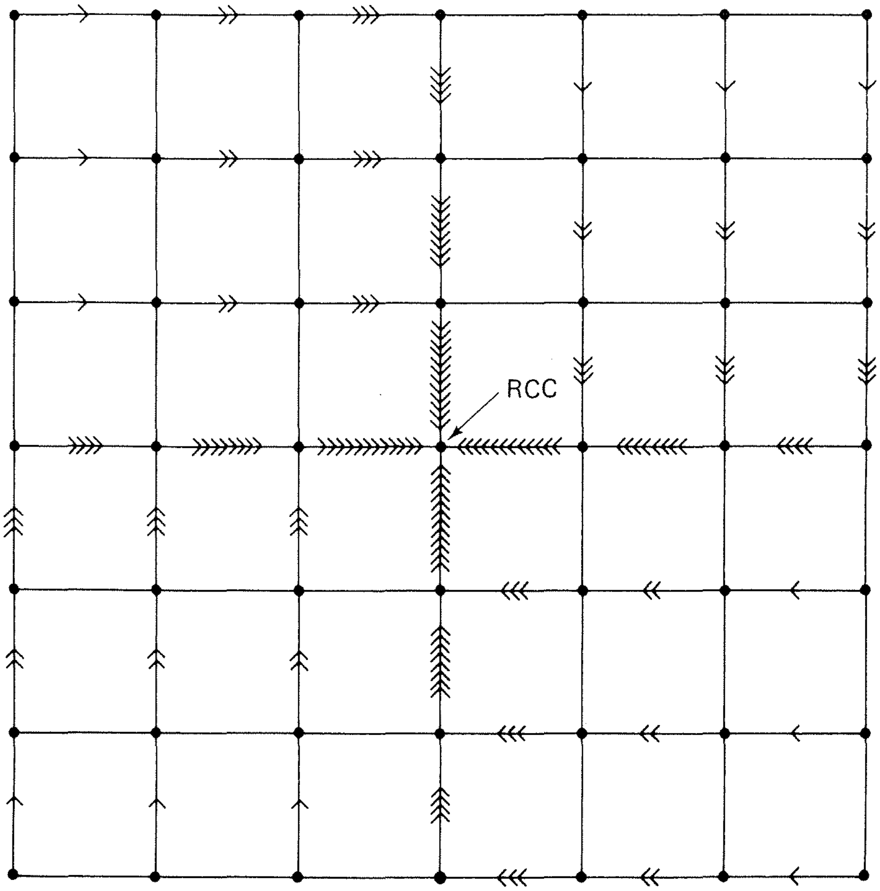


Fig. 5-13. Paths followed from the outgoing IMPs to the RCC.

5.2.4. Isolated Routing

All the problems with centralized routing algorithms suggest that decentralized algorithms might have something to offer. In the simplest decentralized routing algorithms, the IMPs make routing decisions based only upon information they themselves have gleaned; they do not exchange routing information per se with other IMPs. Nevertheless, they try to adapt to changes in topology and traffic. These are usually called **isolated adaptive** routing algorithms.

One simple isolated adaptive algorithm is Baran's (1964) **hot potato** algorithm. When a packet comes in, the IMP tries to get rid of it as fast as it can, by putting it on the shortest output queue. In other words, when a packet arrives, the IMP counts the number of packets queued up for transmission on each of the output lines. It then attaches the new packet to the end of the shortest output queue, without regard to where that line leads. In Fig. 5-14, the inside of IMP *J* from Fig. 5-12 is shown at a certain instant of time. There are four output queues, corresponding to the four output lines. Packets are queued up on each line waiting for transmission. In this example, queue *I* is the shortest, with only one packet queued up. The hot potato algorithm would therefore put the newly arrived packet on this queue.

A variation of this idea is to combine static routing with the hot potato algorithm. When a packet arrives, the routing algorithm takes into account both the static weights of the lines and the queue lengths. One possibility is to use the best static choice, unless its queue exceeds a certain threshold. Another possibility is to use the shortest queue, unless its static weight is too low. Yet another way is to rank the lines in terms of their static weights, and again in terms of their queue lengths, taking the line for which the sum of the two ranks is lowest. Whatever algorithm is chosen should have the property that under light load the line with the highest static weight is usually chosen, but as the queue for this line builds up, some of the traffic is diverted to less busy lines.

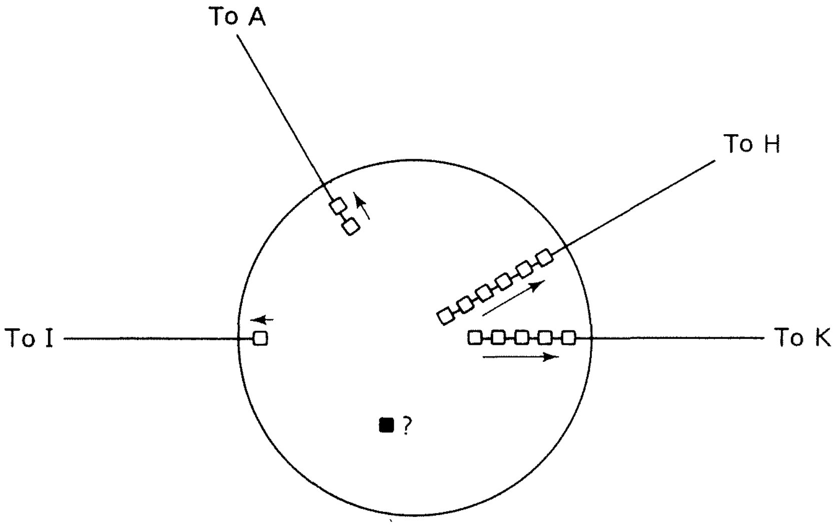


Fig. 5-14. Queueing within the IMP.

Another isolated routing algorithm, also due to Baran, is **backward learning.** In the 1950s and 1960s, when newspaper reporters from Western countries were rarely allowed to visit China, it was common to see news stories beginning with "According to travelers recently arriving in Hong Kong from China ...." The idea was that instead of the reporter going to place *X*, she could talk to people who just came from *X* and ask them what was going on there. The backward learning algorithm does the same thing.

One way to implement backward learning is to include the identity of the source IMP in each packet, together with a counter that is incremented on each hop. If an IMP sees a packet arriving on line *k* from IMP *H* with hop count 4, it knows that *H* cannot be more than four hops away via line *k*. If its current best route to *H* is estimated at more than four hops, it marks line *k* as the choice for traffic to *H* and records the estimated distance as four hops. After a while, every IMP will discover the shortest path to every other IMP.

Alas, there is a fly in the ointment. Since IMPs only record changes for the better, if a line goes down or becomes overloaded, there is no mechanism for recording the fact. Consequently, IMPs must periodically forget everything they have learned and start all over again. During the new learning period, the routing will be far from optimal. If the tables are purged frequently, the IMPs route a substantial number of packets using routes of unknown quality; if the tables are purged rarely, the adaptation process is slow.

Rudin (1976) has described an interesting hybrid between centralized routing and isolated routing, which he calls **delta routing**. In this algorithm, each IMP measures the "cost" of each line (i.e., some function of the delay, queue length, utilization, bandwidth, etc.) and periodically sends a packet to the RCC giving it these values.

Using the information sent to it by the IMPs, the RCC computes the k best paths from IMP i to IMP j, for all i and all j, where only paths that differ in their initial line are considered. Let C_{ij}^1 be the total cost of the best i-j path, C_{ij}^2 be the total cost of the next best path, etc. If $C_{ij}^n - C_{ij}^1 < \delta$, path n is declared to be equivalent to path 1, since their costs differ by so little. When the routing computation is finished, the RCC sends each IMP a list of all the equivalent paths for each of its possible destinations. (Actually, only the initial lines are needed, not the full paths.)

To do actual routing, the IMP is permitted to choose any of the equivalent paths. It may decide among them at random or use the current measured value of the line costs, that is, choose the path from the allowed set whose current initial line is cheapest. By adjusting k and δ, the network operators can transfer authority between the RCC and the IMPs. As $\delta \to 0$, all other paths are deemed inferior to the best path, and the RCC makes all the decisions. However, as $\delta \to \infty$, all the paths considered will be deemed equivalent, and the routing decisions are made in the IMPs based on local information only. Rudin's simulations have shown that δ can be chosen to provide better performance than either pure centralized routing or pure isolated routing.

5.2.5. Flooding

An extreme form of isolated routing is **flooding**, in which every incoming packet is sent out on every outgoing line except the one it arrived on. Flooding obviously generates vast numbers of duplicate packets, in fact, an infinite number unless some measures are taken to damp the process. One such measure is to have a hop counter contained in the header of each packet, which is decremented at each hop, with the packet being discarded when the counter reaches zero. Ideally, the hop counter should be initialized to the length of the path from source to destination. If the sender does not know how long the path is, it can initialize the counter to the worst-case, namely, the full diameter of the subnet.

An alternative technique for damming the flood is to have the source IMP put a sequence number in each packet it receives from its hosts. Each IMP then needs a list per source IMP telling which sequence numbers originating at that source have already been seen. To prevent the list from growing without bound, each list should be augmented by a counter, *k*, meaning that all sequence numbers through *k* have been seen. When a packet comes in, it is easy to check if the packet is a duplicate; if so, it is discarded.

Flooding is not practical in most applications, but it does have some uses. For example, in military applications, where large numbers of IMPs may be blown to bits at any instant, the tremendous robustness of flooding is highly desirable. In distributed data base applications, it is sometimes necessary to update all the data bases concurrently, in which case flooding can be useful. A third possible use of flooding is as a metric against which other routing algorithms can be compared. Flooding always chooses the shortest path, because it chooses every possible path in parallel. Consequently, no other algorithm can produce a shorter delay (if we ignore the overhead generated by the flooding process itself).

A variation of flooding that is slightly more practical is **selective flooding**. In this algorithm the IMPs do not send every incoming packet out on every line, only on those lines that are going approximately in the right direction. There is usually little point in sending a westbound packet on an eastbound line unless the topology is extremely peculiar.

Yet another nonadaptive algorithm is **random walk**. The IMP simply picks a line at random and forwards the packet on it. Here, also, the IMP can make some attempt to get the packet heading in roughly the right direction. If the subnet is richly interconnected, this algorithm has the property of making excellent use of alternative routes. It is also highly robust.

5.2.6. Distributed Routing

In this class of routing algorithms, originally used in the ARPANET, each IMP periodically exchanges explicit routing information with each of its neighbors. Typically, each IMP maintains a routing table indexed by, and containing one entry for, each other IMP in the subnet. This entry contains two parts: the preferred outgoing line to use for that destination, and some estimate of the time or distance to that destination. The metric used might be number of hops, estimated time delay in milliseconds, estimated total number of packets queued along the path, excess bandwidth, or something similar.

The IMP is assumed to know the "distance" to each of its neighbors. If the metric is hops, the distance is just one hop. If the metric is queue length, the IMP simply examines each queue. If the metric is delay, the IMP can measure it directly with special "echo" packets that the receiver just timestamps and sends back as fast as it can.

As an example, assume that delay is used as a metric and that the IMP knows the delay to each of its neighbors. Once every *T* msec each IMP sends to each neighbor a list of its estimated delays to each destination. It also receives a similar list from each neighbor. Imagine that one of these tables has just come in from neighbor X, with X_i being X's estimate of how long it takes to get to IMP i. If the IMP knows that the delay to X is m msec, it also knows that it can reach IMP i via X in $X_i + m$ msec via X. By performing this calculation for each neighbor, an IMP can find out which estimate seems the best, and use that estimate and the corresponding line in its new routing table. Note that the old routing table is not used in the calculation.

This updating process is illustrated in Fig. 5-15. Part (a) shows a subnet. The first four columns of part (b) show the delay vectors received from the neighbors of IMP J. A claims to have a 12-msec delay to B, a 25-msec delay to C, a 40-msec delay to D, etc. Suppose that J has measured or estimated its delay to its neighbors, A, I, H, and K as 8, 10, 12, and 6 msec, respectively.

Consider how J computes its new route to IMP G. It knows that it can get to A in 8 msec, and A claims to be able to get to G in 18 msec, so J knows it can count on a delay of 26 msec to G if it forwards packets bound for G to A. Similarly, it computes the delay to G via I, H, and K as 41 (31 + 10), 18 (6 + 12), and 37 (31 + 6) msec respectively. The best of these values is 18, so it makes an entry in its routing table that the delay to G is 18 msec, and that the route to use is via H. The same calculation is performed for all the other destinations, with the new routing table shown in the last column of the figure.

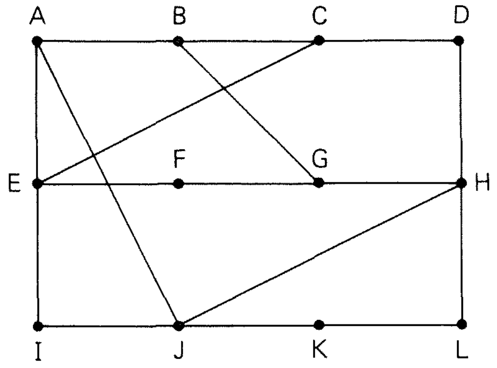
5.2.7. Optimal Routing

Even without knowing the details of the subnet topology and traffic it is possible to make some general statements about optimal routes. One such statement is known as the **optimality principle**. It states that if IMP J is on the optimal path from IMP I to IMP K, then the optimal path from J to K also falls along the same route. To see this, call the part of the route from I to J r_1 and the rest of the route r_2. If a route better than r_2 existed from J to K, it could be concatenated with r_1 to improve the route from I to K, contradicting our statement that $r_1 r_2$ is optimal.

As a direct consequence of the optimality principle, we can see that the set of optimal routes from all sources to a given destination form a tree rooted at the destination. Such a tree is called a **sink tree** and is illustrated in Fig. 5-16. Since the sink tree is indeed a tree, it does not contain any loops, so each packet will be delivered within a finite and bounded number of hops.

In general, if traffic from IMP X passes through IMP Y as it flows along the sink tree to the destination, X is said to be **upstream** from Y and Y is said to be **downstream** from X. To illustrate these notions, consider the subnet of Fig. 5-17(a), with the sink tree for destination H shown in Fig. 5-17(b). (Throughout this example we will use shortest path routing, with ties broken alphabetically; for example, B routes to H via $BCEH$ rather than $BDGH$ because $BCEH < BDGH$.)

With the sink tree in mind, consider what happens when a line goes down, blocking the path to a certain destination. An IMP cannot simply divert its traffic to



(a)

	A	I	H	K	New estimated delay from J	Line
A	0	24	20	21	8	A
B	12	36	31	28	20	A
C	25	18	19	36	28	I
D	40	27	8	24	20	H
E	14	7	30	22	17	I
F	23	20	19	40	30	I
G	18	31	6	31	18	H
H	17	20	0	19	12	H
I	21	0	14	22	10	I
J	9	11	7	10	0	—
K	24	22	22	0	6	K
L	29	33	9	9	15	K
	JA delay = 8	JI delay = 10	JH delay = 12	JK delay = 6		

(b)

Fig. 5-15. (a) A subnet. (b) Input from *A*, *I*, *H*, *K*, and the new routing table *J*.

another IMP that is upstream from it with respect to that destination. It must seek out a neighbor that is attached to another (independent) branch of the sink tree. In Chu's (1978) algorithm, each IMP maintains a routing table with one row per destination, as in Fig. 5-17(c). Each column gives the number of hops to the destination via a specific output line. For IMP *G*, the output lines are *D*, *F*, and *H*. The best route is indicated by a circle. Entries that refer to upstream IMPs are left blank.

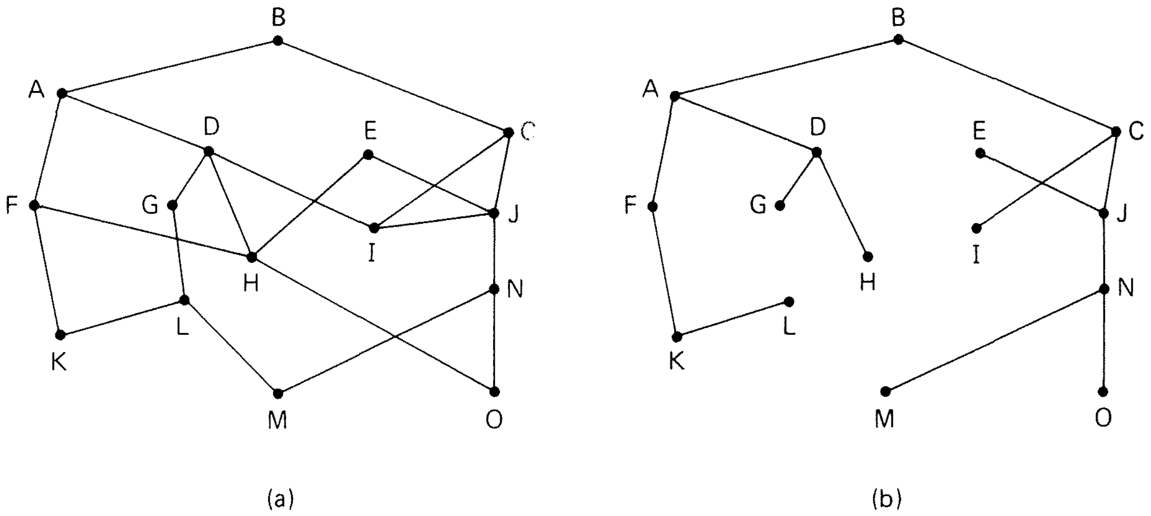
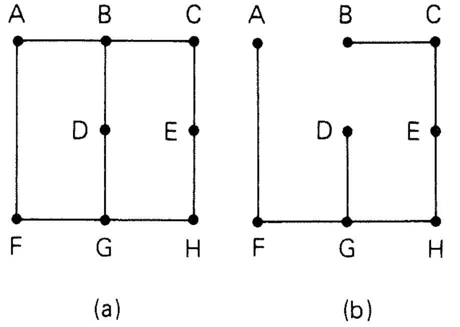


Fig. 5-16. (a) A subnet. (b) The sink tree for IMP *B*, using number of hops as metric.



Destination	Via D	Via F	Via H
A	3	②	
B	②	3	4
C	③	4	3
D	①		
E	4		②
F		①	
G	—	—	—
H			①

(c)

Fig. 5-17. (a) A subnet. (b) Sink tree for *H*. (c) Routing table used by *G*.

Note that the sink tree used to determine who is upstream and who is downstream is different for each row of the table.

Now let us see what happens if line *GH* fails. IMP *G* starts out by marking all the entries in column *H* as unusable. It then checks each row for which the best route has been wiped out to see if an alternative route is available. For destination *E*, for example, an alternative route is available via *D*. (Remember that in this example *D* routes to *E* via *B* because *DBCE* < *DGHE*.)

For destination *H* the situation is worse because both neighbors are upstream from *G*. Consequently, *G* sends each of them a control packet saying "I cannot reach *H* any more. Please help me." Upon receiving the packet each neighbor checks to see if it has an alternative (previously suboptimal) route. *D* has such an alternative, and sends a reply back announcing it. *F*, however, has no alternative, because its only other neighbor, *A*, is upstream from it, so it acts in much the same way that it would have if line *FG* had gone down, namely, by asking *A* for help. Control packets continue propagating upstream until someone finds an alternate route. When *G* finally receives replies from all its upstream neighbors, it can make new entries in its routing table and choose the best. The only condition under which the algorithm fails is when none of the IMPs upstream from the failed link can make contact with any IMP on another branch of the sink tree. This condition occurs only when the subnet has been broken into two separate components.

5.2.8. Flow-Based Routing

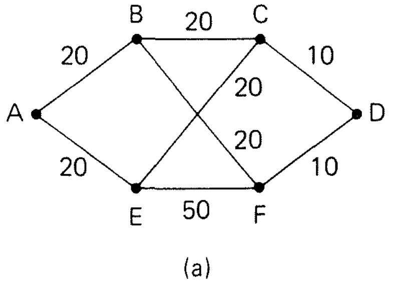
To some extent, all the algorithms discussed so far are largely empirical, rather than being derived from some fundamental theory. However, under certain limited conditions it is possible to find routing algorithms that are provably optimal. In this section we will give a brief introduction to this subject. Bertsekas and Gallager (1987) give a more comprehensive treatment.

In some networks, the mean data flow between each pair of nodes is relatively stable and predictable. For example, in a corporate network for a retail store chain, each store might send orders, sales reports, inventory updates, and other well-defined types of messages to known sites in a pre-defined pattern, so that the total volume of traffic varied little from day to day. Under conditions in which the average traffic from i to j is known in advance and, to a reasonable approximation, constant in time, it is possible to analyze the flows mathematically to optimize the routing.

The basic idea behind the analysis is that for a given line, if the capacity and average flow are known, it is possible to compute the mean packet delay on that line from queueing theory. From the mean delays on all the lines, it is straightforward to calculate a flow-weighted average to get the mean packet delay for the whole network. The routing problem then reduces to finding the routing algorithm that produces the minimum average delay for the network.

To use this technique, certain information must be known in advance. First the network topology must be known. Second, the traffic matrix, F_{ij}, must be given. Third, the line capacity matrix, C_{ij}, specifying the capacity of each line in bps must be available. Finally, a (possibly tentative) routing algorithm must be chosen.

As an example of this method, consider the full-duplex network of Fig. 5-18(a). The weights on the arcs give the capacities, C_{ij}, in each direction in kbps. The matrix of Fig. 5-18(b) has an entry for each source-destination pair. The entry for source i to destination j shows the route to be used for i - j traffic, and also the number of packets/sec to be sent from i to j. For example, 3 packets/sec go from *B* to *D*, and they use route *BFD*. Notice that some routing algorithm has already been applied to derive the routes shown in the matrix.



Destination

Source	A	B	C	D	E	F
A	▨	9 AB	4 ABC	1 ABFD	7 AE	4 AEF
B	9 BA	▨	8 BC	3 BFD	2 BFE	4 BF
C	4 CBA	8 CB	▨	3 CD	3 CE	2 CEF
D	1 DFBA	3 DFB	3 DC	▨	3 DCE	4 DF
E	7 EA	2 EFB	3 EC	3 ECD	▨	5 EF
F	4 FEA	4 FB	2 FEC	4 FD	5 FE	▨

(b)

Fig. 5-18. (a) A network with line capacities shown in kbps. (b) The traffic in packets/sec and the routing matrix.

Given this information, it is straightforward to calculate the total in line i λ_i. For example, the *B*-*D* traffic contributes 3 packets/sec to the *BF* line and also 3 packets/sec to the *FD* line. Similarly, the *A*-*D* traffic contributes 1 packet/sec to each of three lines. The total traffic in each eastbound line is shown in the λ_i column of Fig. 5-19. In this example, all the traffic is symmetric, that is the *XY* traffic is identical to the *YX* traffic, for all *X* and *Y*. The figure also shows the mean number of packets/sec on each line, μC_{ij} assuming a mean packet size of $\mu = 800$ bits.

i	Line	λ_i (pkts/sec)	C_i (kbps)	μC_i (pkts/sec)	T_i (ms)
1	AB	14	20	25	91
2	BC	12	20	25	77
3	CD	6	10	12.5	154
4	AE	11	20	25	71
5	EF	13	50	62.5	20
6	FD	8	10	12.5	222
7	BF	10	20	25	67
8	EC	8	20	25	59

Fig. 5-19. Analysis of the network of Fig. 5-18 using a mean packet size of 800 bits. The reverse traffic (*BA*, *CB*, etc.) is the same as the forward traffic.

The last column of Fig. 5-19 gives the mean delay for each line derived from the queueing theory formula

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

where μ is the mean packet size in bits, C is the capacity in bps, and λ is the mean flow in packets/sec. For example, with a capacity $\mu C = 25$ packets/sec and an actual flow $\lambda = 14$ packets/sec, the mean delay is 91 msec. Note that with $\lambda = 0$, the mean delay is still 40 msec, corresponding to the fact that the capacity is 25 packets/sec. In other words, the "delay" includes both queueing time and service time.

To compute the mean delay time for the entire network, we take the weighted sum of each of the eight links, with the weight being the fraction of the total traffic using that link. In this example, the mean turns out to be 114 msec.

To evaluate a different routing algorithm, we can repeat the entire process, only with different flows to get a new average delay. If we restrict ourselves to only single path routing algorithms, there are only a finite number of ways to route packets from each source to each destination. It is simple enough to write a program to simply try them all and find out which one has the smallest mean delay. This one is then the best routing algorithm.

If we allow multipath routing, the situation is more complicated because there are then an infinite number of ways to route traffic from A to D, for example, a fraction α of the traffic via $ABCD$ and the rest via $AEFD$, where α can take on any real value between 0 and 1. However, in this case we can write down the flow in each line as a function of the load-splitting parameter α (and similar parameters for other flows). This approach yields an expression for the mean delay as a function of all the parameters. Various techniques exist for minimizing this expression to find the optimum value for α and the other parameters, which effectively determines the routing algorithm.

5.2.9. Hierarchical Routing

As networks grow in size, the IMP routing tables grow proportionally. Not only is IMP memory consumed by ever increasing tables, but more CPU time is needed to scan them and more bandwidth is needed to send status reports about them. At a certain point the network may grow to the point where it is no longer feasible for every IMP to have an entry for every other IMP, so the routing will have to be done hierarchically, as it is in the telephone network.

When hierarchical routing is used, the IMPs are divided into **regions**, with each IMP knowing all the details about how to route packets to destinations within its own region, but knowing nothing about the internal structure of other regions. When different networks are connected together, it is natural to regard each one as a separate region in order to free the IMPs in one network from having to know the topological structure of the other ones.

For huge networks, a two-level hierarchy may be insufficient; it may be necessary to group the regions into clusters, the clusters into zones, the zones into groups, and so on, until we run out of names for aggregations. As an example of a multilevel hierarchy, consider how a packet might be routed from Berkeley, California to Malindi, Kenya. The Berkeley IMP would know the detailed topology within California, but would send all out-of-state traffic to the Los Angeles IMP. The Los Angeles IMP would be able to route traffic to other domestic IMPs, but would send foreign traffic to New York. The New York IMP would be programmed to direct all traffic to the IMP in the destination country responsible for handling foreign traffic, say in Nairobi. Finally, the packet would work its way down the tree in Kenya until it got to Malindi.

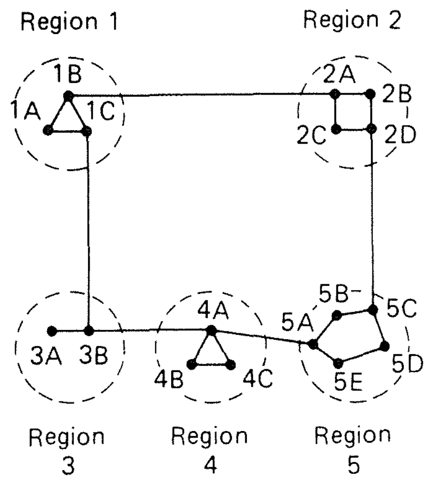
Figure 5-20 gives a quantitative example of routing in a two-level hierarchy with five regions. The full routing table for IMP $1A$ has 17 entries, as shown in Fig. 5-20(b). When routing is done hierarchically, as in Fig. 5-20(c), there are entries for all the local IMPs as before, but all other regions have been condensed into a single IMP, so all traffic for region 2 goes via the $1B-2A$ line, but the rest of the remote traffic goes via the $1C-3B$ line. Hierarchical routing has reduced the table from 17 to 7 entries. As the ratio of the number of regions to the number of IMPs within a region grows, the savings in table space grow proportionally.

Unfortunately, these gains in space are not free. There is a penalty to be paid, and this penalty is in the form of increased path length. For example, the best route from $1A$ to $5C$ is via region 2, but with hierarchical routing all traffic to region 5 goes via region 3, because that is a better choice for most destinations in region 5.

When a single network becomes very large, an interesting question is how many levels should the hierarchy have? For example, consider a subnet with 720 IMPs. If there is no hierarchy, each IMP needs 720 routing table entries. If the subnet is partitioned into 24 regions of 30 IMPs each, each IMP needs 30 local entries plus 23 remote entries for a total of 53 entries. If a three-level hierarchy is chosen, with eight clusters, each containing 9 regions of 10 IMPs, each IMP needs 10 entries for local IMPs, 8 entries for routing to other regions within its own cluster, and seven entries for distant clusters, for a total of 25 entries. Kamoun and Kleinrock (1979) have discovered that the optimal number of levels for an N IMP subnet is $\ln N$, requiring a total of $e \ln N$ entries per IMP. They have also discovered that the increase in effective mean path length caused by hierarchical routing is sufficiently small that it is not objectionable.

5.2.10. Broadcast Routing

For some applications, hosts need to send messages to all other hosts. Typical examples might be for scheduling: a host wants to find out which other hosts are willing and able to perform a certain task for it, or distributed data base updates. In



(a)

Full table for 1A

Dest.	Line	Hops
1A	—	—
1B	1B	1
1C	1C	1
2A	1B	2
2B	1B	3
2C	1B	3
2D	1B	4
3A	1C	3
3B	1C	2
4A	1C	3
4B	1C	4
4C	1C	4
5A	1C	4
5B	1C	5
5C	1B	5
5D	1C	6
5E	1C	5

(b)

Hierarchical table for 1A

Dest	Line	Hops
1A	—	—
1B	1B	1
1C	1C	1
2	1B	2
3	1C	2
4	1C	3
5	1C	4

(c)

Fig. 5-20. Hierarchical routing.

some networks the IMPs may also need such a facility, for example to distribute routing table updates. Sending a packet to all destinations simultaneously is called **broadcasting**, and various methods have been proposed for implementing it. Our treatment is based on the work of Dalal and Metcalfe (1978).

One broadcasting method that requires no special features from the subnet is for the source to simply send a distinct packet to each destination. Not only is the method wasteful of bandwidth but it also requires the source to have a complete list of all destinations. In practice this may be the only possibility, but it is the least desirable of the methods.

Flooding is another obvious candidate. Although flooding is ill-suited for ordinary point-to-point communication, for broadcasting it might rate serious consideration, especially if none of the methods described below are applicable. The problem with flooding as a broadcast technique is the same problem it has as a point-to-point routing algorithm: it generates too many packets and consumes too much bandwidth.

A third algorithm is **multidestination routing**. If this method is used, each packet contains either a list of destinations or a bit map indicating the desired destinations. When a packet arrives at an IMP, the IMP checks all the destinations to determine the set of output lines that will be needed. (An output line is needed if it is the best route to at least one of the destinations.) The IMP generates a new copy of the packet for each output line to be used and includes in each packet only those destinations that are to use the line. In effect, the destination set is partitioned among the output lines. After a sufficient number of hops, each packet will carry only one destination and can be treated as a normal packet. Multidestination routing is like separately addressed packets, except that when several packets must follow the same route, one of them pays full fare and the rest ride free.

A fourth broadcast algorithm makes explicit use of the sink tree for the IMP initiating the broadcast, or any other convenient spanning tree for that matter. (A **spanning tree** is a subset of the subnet that includes all the IMPs but contains no loops.) If each IMP knows which of its lines belong to the spanning tree, it can copy an incoming broadcast packet onto all the spanning tree lines except the one it arrived on. This method makes excellent use of bandwidth, generating the absolute minimum number of packets necessary to do the job. The only problem is that each IMP must have knowledge of some spanning tree for it to be applicable, and many of the routing algorithms we have studied do not have such knowledge.

Our last broadcast algorithm is an attempt to approximate the behavior of the previous one, even when the IMPs do not know anything at all about spanning trees. The idea is remarkably simple once it has been pointed out. When a broadcast packet arrives at an IMP, the IMP checks to see if the packet arrived on the line that is normally used for sending packets *to* the source of the broadcast. If so, there is an excellent chance that the broadcast packet itself followed the best route from the IMP and is therefore the first copy to arrive at the IMP. This being the case, the IMP forwards copies of it onto all lines except the one it arrived on. If, however, the broadcast packet arrived on a line other than the preferred one for reaching the source, the packet is discarded as a likely duplicate.

An example of the algorithm, called **reverse path forwarding**, is shown in Fig. 5-21. Part (a) shows a subnet, part (b) shows a sink tree for IMP *I* of that subnet, and part (c) shows how the reverse path algorithm works. On the first hop, *I* sends packets to *F*, *H*, *J*, and *N*, as indicated by the second row of the tree. Each of these packets arrives on the preferred path to *I* (assuming that the preferred path falls along the sink tree) and is so indicated by a circle around the letter. On the second hop, eight packets are generated, two by each of the IMPs that received a packet on the first hop. As it turns out, all eight of these arrive at previously unvisited IMPs, and all but one arrive along the preferred line. Of the nine packets generated on the third hop, only two arrive on the preferred path (at *C* and *L*), and so only these generate further packets. After five hops and 24 packets, the broadcasting terminates, compared with four hops and 14 packets had the sink tree been followed exactly.

The principal advantage of reverse path forwarding is that it is both reasonably efficient and easy to implement. It does not require IMPs to know about spanning trees, nor does it have the overhead of a destination list or bit map in each broadcast packet as does multidestination addressing. Nor does it require any special mechanism to stop the process, as flooding does (either a hop counter in each

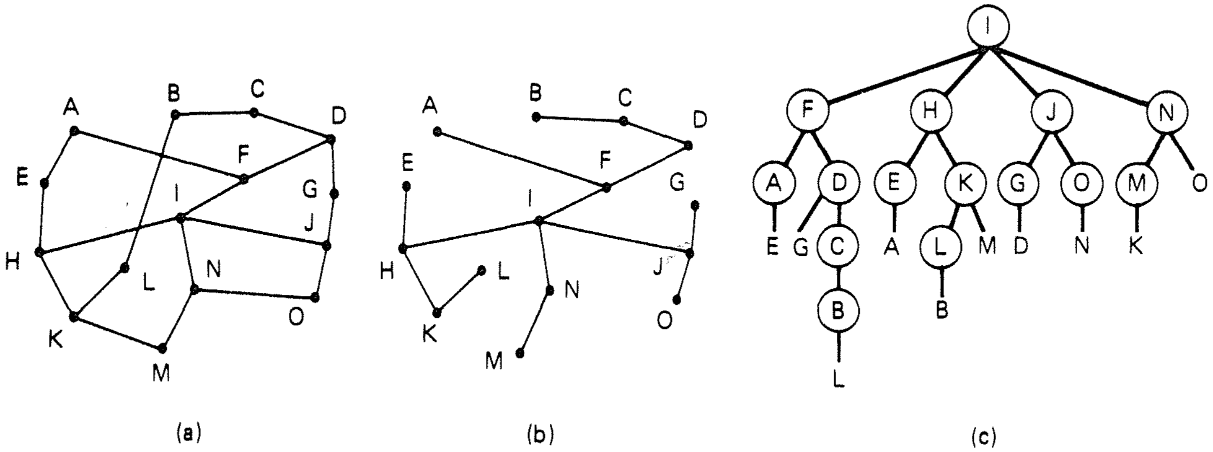


Fig. 5-21. Reverse path forwarding.

packet and a priori knowledge of the subnet diameter, or a list of packets already seen per source).

5.3. CONGESTION CONTROL ALGORITHMS

In this section we will examine five strategies for controlling congestion. These strategies involve allocating resources in advance, allowing packets to be discarded when they cannot be processed, restricting the number of packets in the subnet, using flow control to avoid congestion, and choking off input when the subnet is overloaded.

5.3.1. Preallocation of Buffers

If virtual circuits are used inside the subnet, it is possible to solve the congestion problem altogether, as follows. When a virtual circuit is set up, the call request packet wends its way through the subnet, making table entries as it goes. When it has arrived at the destination, the route to be followed by all subsequent data traffic has been determined and entries made in the routing tables of all the intermediate IMPs.

Normally, the call request packet does not reserve any buffer space in the intermediate IMPs, just table slots. However, a simple modification to the setup algorithm could have each call request packet reserve one or more data buffers as well. If a call request packet arrives at an IMP and all the buffers are already reserved, either another route must be found or a "busy signal" must be returned to the caller. Even if buffers are not reserved, some aspiring virtual circuits may have to be rerouted or rejected for lack of table space, so reserving buffers does not add any new problems that were not already there.

By permanently allocating buffers to each virtual circuit in each IMP, there will always be a place to store any incoming packet until it can be forwarded. First consider the case of a stop-and-wait IMP-IMP protocol. One buffer per virtual circuit per IMP is sufficient for simplex circuits, and one for each direction is sufficient for full-duplex circuits. When a packet arrives, the acknowledgement is not sent back to the sending IMP until the packet has been forwarded. In effect, an acknowledgement means that the receiver not only received the packet correctly, but also that it has a free buffer and is willing to accept another one. If the IMP-IMP protocol allows multiple outstanding packets, each IMP will have to dedicate a full window's worth of buffers to each virtual circuit to completely eliminate the possibility of congestion.

When each virtual circuit passing through each IMP has a sufficient amount of buffer space dedicated to it, packet switching becomes quite similar to circuit switching. In both cases an involved setup procedure is required. In both cases substantial resources are permanently allocated to specific connections, whether or not there is any traffic. In both cases congestion is impossible because all the resources needed to process the traffic have already been reserved. And in both cases there is a potentially inefficient use of resources, because resources not being used by the connection to which they are allocated are nevertheless unavailable to anyone else.

Because dedicating a complete set of buffers to an idle virtual circuit is expensive, some subnets may use it only where low delay and high bandwidth are essential, for example on virtual circuits carrying digitized speech. For virtual circuits where low delay is not absolutely essential all the time, a reasonable strategy is to associate a timer with each buffer. If the buffer lays idle for too long, it is released, to be reacquired when the next packet arrives. Of course, acquiring a buffer might take a while, so packets will have to be forwarded without dedicated buffers until the chain of buffers can be set up again.

5.3.2. Packet Discarding

Our second congestion control mechanism is just the opposite of the first one. Instead of reserving all the buffers in advance, nothing is reserved in advance. If a packet arrives and there is no place to put it, the IMP simply discards it. If the subnet offers datagram service to the hosts, that is all there is to it: congestion is resolved by discarding packets at will. If the subnet offers virtual circuit service, a copy of the packet must be kept somewhere so that it can be retransmitted later. One possibility is for the IMP sending the discarded packet to keep timing out and retransmitting the packet until it is received. Another possibility is for the sending IMP to give up after a certain number of tries, and require the source IMP to time out and start all over again.

Discarding packets at will can be carried too far. It is clearly stupid in the extreme to ignore an incoming packet containing an acknowledgement from a neighboring IMP. That acknowledgement would allow the IMP to abandon a by-now-received packet and thus free up a buffer. However, if the IMP has no spare buffers, it cannot acquire any more incoming packets to see if they contain acknowledgements. The solution is to permanently reserve one buffer per input line to allow all incoming packets to be inspected. It is quite legitimate for an IMP to examine a newly arrived packet, make use of any piggybacked acknowledgement, and then discard the packet anyway. Alternatively, the bearer of good tidings could be rewarded by keeping it, using the just freed buffer as the new input buffer.

If congestion is to be avoided by discarding packets, a rule is needed to tell when to keep a packet and when to discard it. Irland (1978) studied this problem and came up with a simple, yet effective heuristic for discarding packets. In the absence of any explicit rule to the contrary, a single output line might hog all the available buffers in an IMP, since they are simply assigned first come, first served. Figure 5-22(a) shows an IMP with a total of 10 buffers. Three of these are permanently assigned to the input lines. The remaining seven are holding packets queued for transmission on one of the output lines.

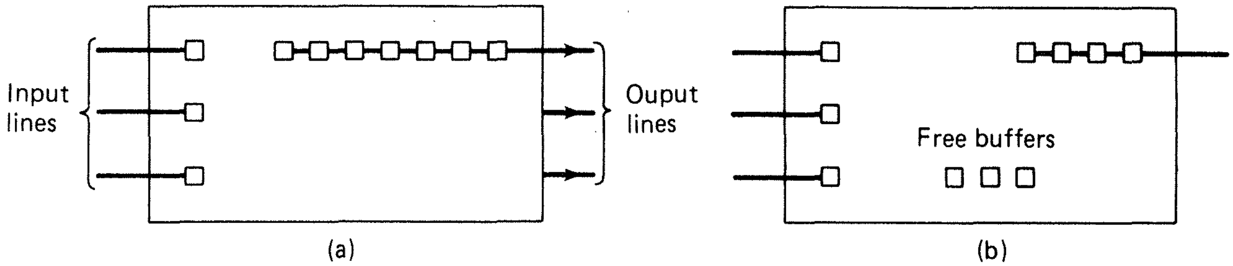


Fig. 5-22. Congestion can be reduced by putting an upper bound on the number of buffers queued on an output line.

Even though two output lines are idle, any incoming packets destined for these lines must be discarded because there are no spare buffers. This is obviously wasteful. Irland's idea is to limit the number of buffers that may be attached to any one output queue. For example, if the limit were set at four, the situation of Fig. 5-22(b) would prevail: three unassigned buffers. A newly arrived packet wanting to go out on the first output line would be discarded rather than allowing it to increase the queue length to five.

This strategy is not really as drastic as it may appear. After all, that output line is already running at maximum capacity. Having seven packets queued instead of four will not pump the bits out any faster, but it will allow traffic for the other lines to be forwarded immediately, possibly doubling or tripling the output rate of the IMP. In any case, the discarded packet will be retransmitted shortly. If the system is well tuned, it will even be retransmitted before the queue empties, so its initial rejection will not even be noticed.

Irland studied several different algorithms for determining maximum queue length, m, for an IMP with k buffers (buffers permanently dedicated for input do not count). The uncontrolled case is $m = k$. If there are s output lines, the case $m = k/s$ effectively means that each buffer is dedicated to a given output line. No line may borrow even one buffer from an idle line, ever. Intuitively this is not efficient, and the study bears this out.

It turns out that the optimal value of m is a complicated function of the mean traffic. Although the IMP could attempt to measure its traffic and continually adjust m, if the traffic were bursty, this probably would not work well. Irland did, however, discover a simple rule of thumb that usually gives good, but not optimal, performance: $m = k/\sqrt{s}$. For example, for seven pool buffers and three lines, $m = 7/\sqrt{3}$, so 4 buffers would be allocated.

A related idea, due to Kamoun (1976), directly prevents any line or lines from starving: a minimum number of buffers is dedicated to each line. If there is no traffic, the empty buffers are reserved. Irland's method can be combined with Kamoun's by having a minimum and a maximum number of buffers for each line. The ARPANET uses this method.

Although discarding packets is easy, it has some disadvantages. Chief among these is the extra bandwidth needed for the duplicates. If the probability of a packet being discarded is p, the expected number of transmissions before it is accepted is $1/(1-p)$. A related issue is how long the timeout interval should be. If it is too short, duplicates will be generated when they are not needed, making the congestion worse. If it is too long, the delay will suffer.

One way to minimize the amount of bandwidth wasted on the retransmission of discarded packets is to systematically discard packets that have not yet traveled far and hence do not represent a large investment in resources. The limiting case of this strategy is to discard newly arrived packets from hosts in preference to discarding transit traffic. For example, IMPs could refuse or discard new packets from attached hosts whenever the number of buffers tied up by new packets (or total packets) exceeds some threshold.

5.3.3. Isarithmic Congestion Control

Congestion occurs when there are too many packets in the subnet. A direct approach to controlling it is to limit the number of packets in the subnet. Davies (1972) proposed a method that enforces precisely such a limit.

In this method, called **isarithmic** because it keeps the number of packets constant, there exist **permits**, which circulate about within the subnet. Whenever an IMP wants to send a packet just given to it by its host, it must first capture a permit and destroy it. When the destination IMP finally removes the packet from the subnet, it regenerates the permit. These simple rules ensure that the number of packets in the subnet will never exceed the number of permits initially present.

However, this method has some problems. First, although it does guarantee that the subnet as a whole will never become congested, it does not guarantee that a given IMP will not suddenly be swamped with packets.

Second, how to distribute the permits is far from obvious. To prevent a newly generated packet from suffering a long delay while the local IMP tries to scout up a permit, the permits must be uniformly distributed, so that every IMP has some. On the other hand, to permit high-bandwidth file transfer, it is undesirable for the sending IMP to have to go hunting all over the place to find enough permits. It would be nicer if they were all centralized, so that requests for substantial numbers could be honored quickly. Some compromise must be found, such as having a maximum number of permits that may be present at any IMP, with excess permits required to hunt for an IMP with space. Note that the random walk of the excess permits itself puts a load on the subnet.

Third, and by no means least, if permits ever get destroyed for any reason, (e.g., transmission errors, malfunctioning IMPs, being discarded by a congested IMP), the carrying capacity of the network will be forever reduced. There is no easy way to find out how many permits still exist while the network is running.

5.3.4. Flow Control

Some networks (notably the ARPANET) have attempted to use flow control mechanisms to eliminate congestion. Although flow control schemes can be used by the transport layer to keep one host from saturating another, and flow control schemes can be used to prevent one IMP from saturating its neighbors, it is difficult to control the total amount of traffic in the network using end-to-end flow control rules. Still, if the hosts are forced to stop transmitting due to strict flow control rules, the subnet will not be as heavily loaded.

Flow control cannot really solve congestion problems for a good reason: computer traffic is bursty. Most of the time an interactive user sits at his terminal scratching his head, but once in a while he may want to scan a large file. The potential peak traffic is vastly higher than the mean rate. Any flow control scheme which is adjusted so as to restrict each user to the mean rate will provide bad service when the user wants a burst of traffic. On the other hand, if the flow control limit is set high enough to permit the peak traffic to get through, it has little value as congestion control when several users demand the peak at once. (If half the people in the world suddenly picked up their telephones to call the other half, there would be a lot of busy signals; the telephone system is also designed for average traffic, not worst case.)

When flow control is used in an attempt to quench congestion, it can apply to the traffic between pairs of:

1. User processes (e.g., one outstanding message per virtual circuit).
2. Hosts, irrespective of the number of virtual circuits open.
3. Source and destination IMPs, without regard to hosts.

In addition, the number of virtual circuits open can be restricted.

5.3.5. Choke Packets

Although limiting the volume of traffic between each pair of IMPs or hosts may indirectly alleviate congestion it does so at the price of potentially reducing throughput even when there is no threat of congestion. What is really needed is a mechanism that is triggered only when the system is congested.

One way is to have each IMP monitor the percent utilization of each of its output lines. Associated with each line is a real variable, u, whose value, between 0.0 and 1.0, reflects the recent utilization of that line. To maintain a good estimate of u, a sample of the instantaneous line utilization, f (either 0 or 1), can be made periodically and u updated according to

$$u_{\text{new}} = au_{\text{old}} + (1 - a)f$$

where the constant a determines how fast the IMP forgets recent history.

Whenever u moves above the threshold, the output line enters a "warning" state. Each newly arriving packet is checked to see if its output line is in warning state. If so, the IMP sends a **choke packet** back to the source host, giving it the destination found in the packet. The packet itself is tagged (a header bit is turned on) so that it will not generate any more choke packets later, and is forwarded in the usual way.

When the source host gets the choke packet, it is required to reduce the traffic sent to the specified destination by X percent. Since other packets aimed at the same destination are probably already under way and will generate yet more choke packets, the host should ignore choke packets referring to that destination for a fixed time interval. After that period has expired, the host listens for more choke packets for another interval. If one arrives, the line is still congested, so the host reduces the flow still more and begins ignoring choke packets again. If no choke packets arrive during the listening period, the host may increase the flow again. The feedback implicit in this protocol should prevent congestion, yet not throttle any flow unless trouble occurs.

Several variations on this congestion control algorithm have been proposed. For one, the IMPs could maintain two critical levels. Above the first level, choke packets are sent back. Above the second, incoming traffic is just discarded, the theory being that the host has probably been warned already. Without extensive tables it is difficult for the IMP to know which hosts have been warned recently about which destinations, and which hosts have not.

Another variation is to use queue lengths instead of line utilization as the trigger signal. The same exponential weighting can be used with this metric as with u, of course. Yet another possibility is to have the IMPs propagate congestion information along with routing information, so that the trigger is not based on only one IMPs observations, but on the fact that somewhere along the path there is a bottleneck. By propagating congestion information around the subnet, choke packets can be sent earlier, before too many more packets are under way, thus preventing congestion from building up.

5.3.6. Deadlocks

The ultimate congestion is a **deadlock**, also called a **lockup**. The first IMP cannot proceed until the second IMP does something, and the second IMP cannot proceed because it is waiting for the first IMP to do something. Both IMPs have ground to a complete halt and will stay that way forever. Deadlocks are not considered a desirable property to have in your network.

The simplest lockup can happen with two IMPs. Suppose that IMP *A* has five buffers, all of which are queued for output to IMP *B*. Similarly, IMP *B* has five buffers, all of which are occupied by packets needing to go to IMP *A* [see Fig. 5-23(a)]. Neither IMP can accept any incoming packets from the other. They are both stuck. This situation is called **direct store-and-forward lockup**. The same thing can happen on a larger scale, as shown in Fig. 5-23(b). Each IMP is trying to send to a neighbor, but nobody has any buffers available to receive incoming packets. This situation is called **indirect store-and-forward lockup**. Note that when an IMP is locked up, all its lines are effectively blocked, including those not involved in the lockup.

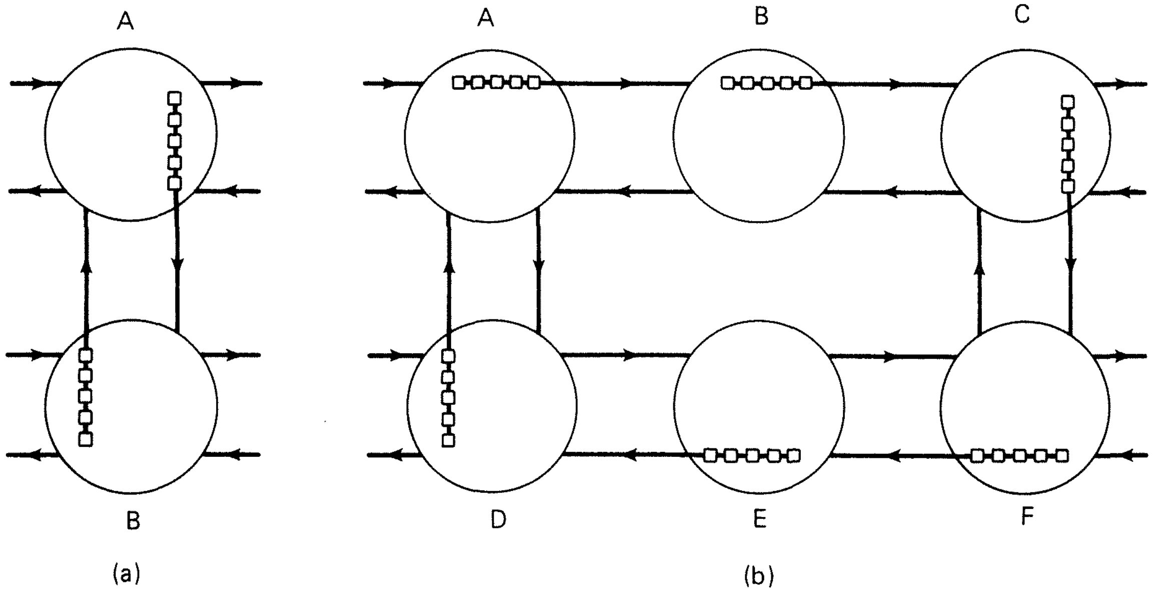


Fig. 5-23. Store-and-forward lockup. (a) Direct. (b) Indirect.

Merlin and Schweitzer (1980) have presented a solution to the problem of store-and-forward lockup. In their scheme, a directed graph is constructed, with the buffers being the nodes of the graph. Arcs connect pairs of buffers in the same IMP or adjacent IMPs. The graph is constructed in such a way that if all packets move from buffer to buffer along the arcs of the graph, then no deadlocks can occur.

As a simple example of their method, consider a subnet with N IMPs in which the longest route from any source to any destination is of length M hops. Each IMP needs $M + 1$ buffers, numbered from 0 to M. The buffer graph is now constructed by drawing an arc from buffer i in each node to buffer $i + 1$ in each of the adjacent nodes. The legal routes from buffer i at IMP A are those to a buffer labeled $i + 1$ at IMPs adjacent to A, and then to a buffer labeled $i + 2$ at IMPs two hops from A, etc.

A packet from a host can only be admitted to the subnet if buffer 0 at the source IMP is empty. Once admitted, this packet can only move to a buffer labeled 1 in an adjacent IMP, and so on, until either it reaches its destination and is removed from the subnet, or it reaches a buffer labeled M, in which case it is discarded. (If M is chosen longer than the longest route, then only looping packets will be discarded.) A packet in buffer i in some IMP may only be moved if buffer $i + 1$ in the IMP chosen by the routing algorithm is free. Note that numbering the buffers does not restrict the choice of routing algorithm, which can be static or dynamic.

To see that this algorithm is deadlock free, consider the state of all buffers labeled M at some instant. Each buffer is in one of three states: empty, holding a packet destined for a local host, or holding a packet for a distant host. In the second case the packet can be delivered, in the third case the packet is looping and must be dropped. In all three cases the buffer can be made free. Consequently, all the packets in buffers labeled $M - 1$ can now be moved forward, one at a time, to be delivered or discarded. Once all the buffers labeled $M - 1$ are free, the packets in buffers labeled $M - 2$ can be moved forward and delivered or discarded. Eventually, all packets can be delivered or discarded. If the routing algorithm guarantees that packets cannot loop, then M can be set to the longest path length, and all packets will be correctly delivered with no discards and no deadlocks.

Merlin and Schweitzer have also presented many improvements to this simple strategy to reduce the number of buffers needed and to improve the performance. For example, a packet that has already made i hops (i.e., is currently in a buffer labeled i), can be put in any available higher numbered buffer at the next hop, not just in buffer $i + 1$. As long as the sequence of buffer numbers is monotonically increasing, there can be no deadlocks.

Although this algorithm avoids deadlock completely, it has the disadvantage that under normal circumstances many buffers will be wasted due to lack of the appropriate packets. Furthermore, lines will be frequently idle because the buffer that the packet at the head of the queue happens to need is not available at the moment.

A completely different deadlock prevention algorithm that does not have these properties has been published by Blazewicz et al. (1987a). In their algorithm, each packet bears a globally unique timestamp. This stamp contains the time the packet was created in the high-order bits and the machine number in the low-order bits. It is not essential that all the clocks be synchronized, although the algorithm tends to give fairer service if the clocks are not too far apart.

The algorithm requires each IMP to reserve one buffer per input line as a special receive buffer. All the other buffers can be used for holding packets in transit. These packets are queued in timestamp order in a separate queue for each output line, as illustrated in Fig. 5-24. In the absence of any deadlock prevention algorithm, the three IMPs in Fig. 5-24 would be deadlocked because although each one can receive a packet from its neighbors, it has nowhere to store it while the packet waits its turn for the next hop.

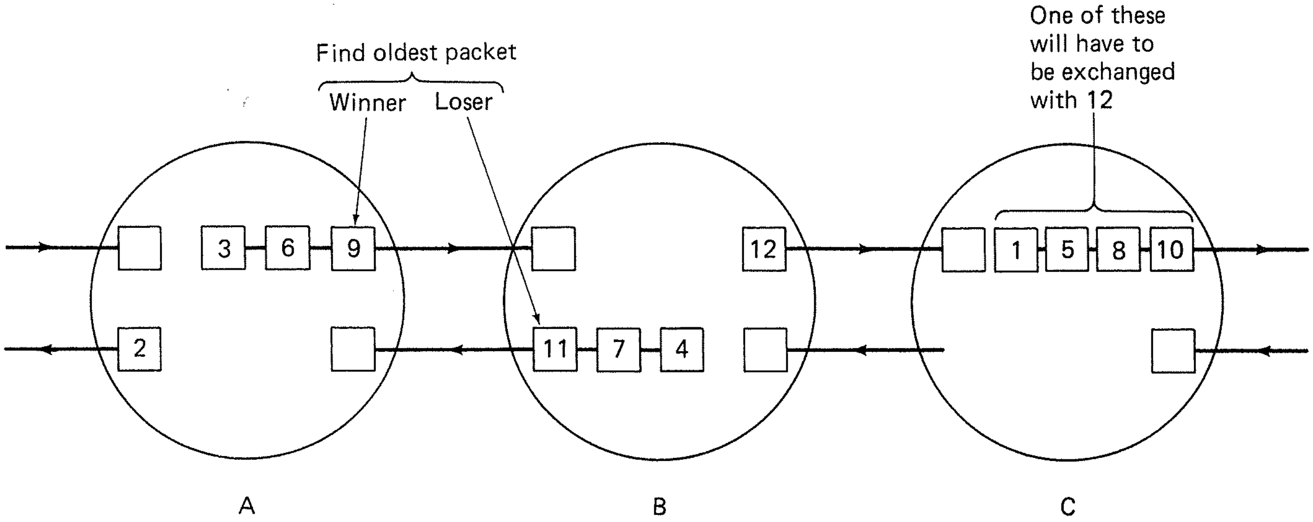


Fig. 5-24. Three potentially deadlocked IMPs.

The essence of the algorithm is that it makes a distinction between three conditions. In all three cases we assume that *A* has an important packet that it wants to send to *B*. The conditions are:

1. *B* has a free buffer.
2. *B* does not have a free buffer, but it does have a packet for *A*.
3. *B* has neither a free buffer nor a packet for *A*.

In case 1, *A* just sends its packet. In case 2, *A* and *B* exchange packets, so that the newly arrived packet can be put in the buffer just freed by the packet going the other way. In case 3, *B* is forced to choose a packet not destined for *A* (preferably, one that is at least going in the same general direction as *A*) and this packet is exchanged with *A* as in case 2.

Now we can explain the algorithm and prove that it is deadlock free. Whenever a line goes idle, the two IMPs at the ends of it exchange control packets giving the timestamp of its oldest packet that wants to use the line. The one with the lowest number (oldest packet) wins and that packet is sent, even if case 3 applies and the losing IMP is forced to send another packet in the wrong direction to free up buffer space for the winner. At any instant, one packet in the subnet has the honor of being the oldest. This packet will always come on top in any IMP-IMP discussion about age, so this packet will make nonstop progress to its destination. As soon as it has been delivered, some other packet becomes the oldest, and it can then proceed unobstructed to its destination. As time marches on, every packet eventually becomes the oldest and is delivered.

Two criticisms can be leveled at this algorithm. First, a young packet may be sent far out of its way before it acquires enough seniority to start consistently winning the IMP-IMP timestamp comparisons. However, in practice, case 3 only occurs when the subnet is heavily loaded (all buffers full). If an IMP has, say, 3 or 4 outgoing lines and 20 to 50 full buffers, chances are that none of the outgoing queues will be empty, so that the losing IMP will nearly always be able to find a good packet to exchange with the winner.

The second criticism has to do with clock synchronization. If one IMP's clock is a few seconds ahead of all the others, its packets will be at a competitive disadvantage when timestamps are compared. Still, within a few seconds, its packets will get delivered, so the condition is annoying but not fatal. If each pair of adjacent IMPs periodically exchanges clock values, each IMP can then set its time to the average of its neighbor's times. In this way, no clock can get very far out of step with the rest.

Store-and-forward lockups are not the only kind of deadlocks that plague the subnet. Consider the situation of an IMP with 20 buffers and lines to four other IMPs, as shown in Fig. 5-25. Four of the buffers are dedicated to the four input lines, to help alleviate congestion. Assume that a sliding window protocol is being used, with window size seven. Further, assume that packets are accepted out of order by the IMP, but must be delivered to the host in order. At the time of the snapshot, four virtual circuits, 0, 1, 2, and 3, are open between the host and other hosts.

Rather than dedicate a full window load of buffers to each open virtual circuit, buffers are simply assigned on a first come, first served basis. As soon as the next sequence number expected by the host becomes available, that packet is passed to the host (with each virtual circuit independent of all the others). However higher number packets within the window are buffered in the usual way.

In Fig. 5-25 *v* and *s* indicate the virtual circuit and sequence number of each packet, respectively. The host is waiting for sequence number 0 on all four virtual circuits, but none of them have arrived undamaged yet. Nevertheless, all the buffers are occupied.

If packet 0 should arrive, it would have to be discarded due to lack of buffer space. As a result, no more packets can be passed to the host and no buffers freed. This deadlock occurred in the ARPANET in a slightly different form, and was called **reassembly lockup**.

In the ARPANET there are multipacket messages (i.e., messages too large to fit into a single packet). By allowing hosts to pass relatively large chunks of information to the IMPs in a single transfer, the number of host interrupts could be reduced. The sending IMP splits multipacket messages into individual packets and sends

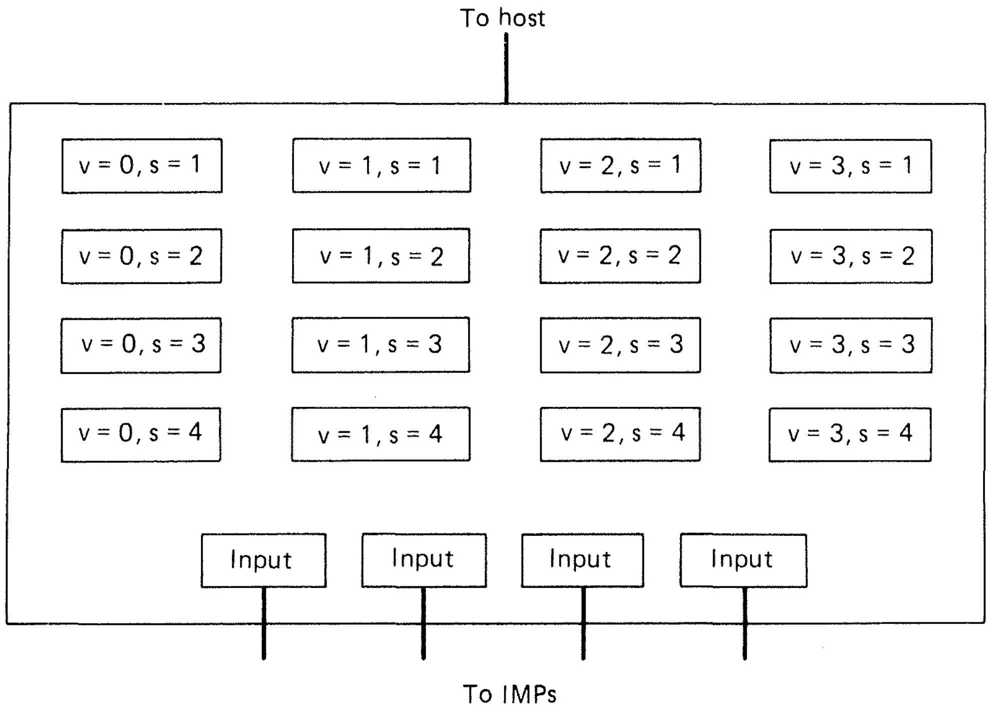


Fig. 5-25. Reassembly lockup.

each one separately. The destination IMP must put all the pieces together before passing the reassembled message to the host. If pieces of several multipacket messages have managed to commandeer all the available buffer space in the destination IMP, the missing pieces will have to be rejected and the IMP (and host) will be deadlocked.

The ARPANET solution is to have the source IMP ask the destination IMP permission to send a multipacket message. The destination IMP then dedicates enough buffers to reassemble the full message before telling the source to go ahead. The sliding window version of the deadlock can only be prevented by dedicating a full window's worth of buffers to each open virtual circuit. The reason this strategy is more attractive in the ARPANET case is that there most (96%) messages are single packet, not multipacket, whereas in the sliding window case, in effect, all messages are multipacket.

Two other problems discovered in the ARPANET are worth mentioning here. Both were caused by malfunctioning IMPs. All of a sudden, one IMP announced that it had zero delay to all other IMPs in the entire subnet. The adaptive routing algorithm spread the good news far and wide. The other IMPs were ecstatic. Within a few seconds, practically all of the traffic in the entire subnet was headed toward the only IMP that was not working properly. Although not a deadlock, this certainly brought the whole network to a grinding halt.

The other problem was also caused by a failing memory. One fine day the Aberdeen IMP (on the East Coast) decided that it was the UCLA IMP (on the West

Coast). The consequences of this case of mistaken identity can be easily imagined, especially for the UCLA and Aberdeen hosts.

The fix applied to the IMP software was to have each IMP periodically compute a software checksum of its own code and tables, in hope of discovering failing memory words, such as those that caused the above problems. Nevertheless the nagging question of how you prevent one bad IMP from bringing the whole network down remains. One of the arguments in favor of computer networking is that higher reliability can be achieved so: if one machine goes down, there are still plenty of others around. However, if a single failure at one site often pollutes the entire nationwide (or worldwide) system, there may be no advantage at all.

An exhaustive catalog of other network deadlocks and possible solutions to them is given by Lai (1982). Other deadlock prevention methods are discussed by Blazewicz et al. (1987b), and Gopal (1985).

5.4. INTERNETWORKING

Up until now, we have implicitly assumed that there is a single homogeneous network, with each machine using the same protocol in each layer. Unfortunately, this assumption is wildly optimistic. Many different networks exist. More than 20,000 SNA networks, more than 2000 DECNET networks, and uncountably many LANs of every kind imaginable are in daily operation all over the world (Green, 1986). Very few of these use the OSI model. In this section we will take a careful look at the issues that arise when it becomes necessary to interconnect two or more networks together to form an **internet**. Additional detail can be found in (Burg et al. 1984; Hinden et al. 1983; Israel and Weissberger, 1987; Postel 1980; Schneidewind 1983; Stallings 1987; and Weissberger and Israel, 1987).

Enormous controversy exists about the question of whether today's abundance of network types is a temporary condition that will go away as soon as everyone realizes how wonderful OSI is, or whether it is an inevitable, but permanent feature of the world that is here to stay. We believe that a variety of different networks will always be around, for the following reasons. First of all, the installed base of non OSI systems is already very large and growing rapidly. IBM is still selling new SNA systems. Most UNIX shops run TCP/IP. LANs are rarely OSI. This trend will continue for years because not all vendors perceive it in their interest for their customers to be able to easily migrate to another vendor's system.

Second, as computers and networks get cheaper, the place where decisions get made moves downward. Many companies have a policy to the effect that purchases costing over a million dollars have to be approved by top management, purchases costing over 100,000 dollars have to be approved by middle management, but purchases under 100,000 dollars can be made by department heads without any higher approval. This can easily lead to the accounting department installing an

Ethernet, the engineering department installing a token bus, and the personnel department installing a token ring.

Third, different networks (e.g., LAN and satellite) have radically different technology, so it should not be surprising that as new hardware developments occur, new software will be needed that does not fit the OSI model.

Let us assume that multiple, incompatible networks are going to be a fact of life for a while, and take a look at some circumstances where it is desirable to connect them together. At most universities, the computer science and electrical engineering departments have their own LANs, often different. These LANs have numerous personal computers, workstations, and minicomputers on them. People interested in number crunching (physicists) or letter crunching (poets) frequently use the computer center's mainframe, in the former case due to the computing power available, and in the latter due to a lack of interest in maintaining hardware. Both the departmental LANs and the mainframes are often connected to national or international WANs, as well as to each other.

The following scenarios are easy to imagine:

1. LAN-LAN: A computer scientist downloading a file to engineering.
2. LAN-WAN: A computer scientist sending mail to a distant physicist.
3. WAN-WAN: Two poets exchanging sonnets.
4. LAN-WAN-LAN: Engineers at different universities communicating.

Figure 5-26 illustrates these four types of connections as dotted lines. In each case, it is necessary to insert a "black box" at the junction between two networks, to handle the necessary conversions as packets move from one network to the other. The generic term for these devices is **relay**. We will discuss the various types (bridges and gateways) later in this chapter. Relays can be **bilateral**, connecting just two networks, or **multilateral**, connecting several networks.

5.4.1. OSI and Internetworking

In the OSI model, internetworking is done in the network layer. In all honesty, this is not one of the areas in which ISO has devised a model that has met with universal acclaim (network security is another one). From looking at the documents, one gets the feeling that internetworking was hastily grafted onto the main structure at the last minute. In particular, the objections from the ARPA Internet community did not carry as much weight as they perhaps should have, inasmuch as DARPA had 10 years experience running an internet with hundreds of interconnected networks, and had a good idea of what worked in practice and what did not.

ISO is not alone in this failing, however. When CCITT drew up its international network numbering plan, for example, it decided that four decimal digits

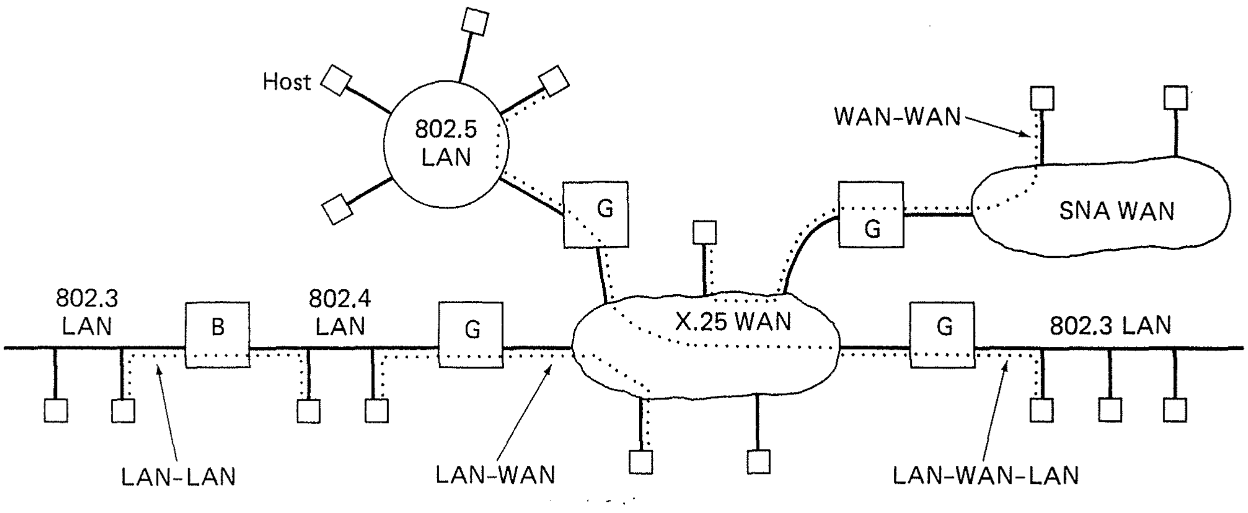


Fig. 5-26. Network interconnection. The boxes marked B are bridges. Those marked G are gateways.

(i.e., 10,000 networks) would be enough for the entire world for years to come. With 20,000 SNA networks and probably even more LANs already around, four decimal digits is grossly inadequate.

The problem is not one of poor estimation; it is a question of mentality. In CCITT's view, each country *ought* to have just one or two public networks, run by the national PTT (or carriers such as TELENET and TYMNET in the U.S.). All the private networks do not count for very much in CCITT's vision. However, not all the users share this viewpoint. Even if all 20,000 SNA networks could somehow miraculously be converted overnight to OSI, it is very unlikely that their owners would be willing to give up administrative control and merge them all into one big, homogeneous public network.

All this said, let us see how internetworking is handled in the OSI model. Where needed, the network layer can be divided into three sublayers: the **subnet access sublayer**, the **subnet enhancement sublayer**, and the **internet sublayer**, as shown in Fig. 5-27. The purpose of the subnet access layer is to handle the network layer protocol for the specific subnet being used. It generates and receives data and control packets and performs the ordinary network layer functions. The software is designed to interface to the real subnet available. There is no guarantee that it will also work with other subnets.

The subnet enhancement sublayer is designed to harmonize subnets that offer different services. In the relay of Fig. 5-27, the upper boundary of 3a is different from 3a′. However, the upper boundary of 3b and 3b′ are the same, so that 3c can work with either subnet.

Subnets can differ in many ways. As one example, consider addressing. The internet sublayer uses NSAPs for addressing. Remember that an NSAP address refers not only to a specific machine, but also to a specific access point within that machine to which a transport process can attach itself. Thus NSAP addresses are

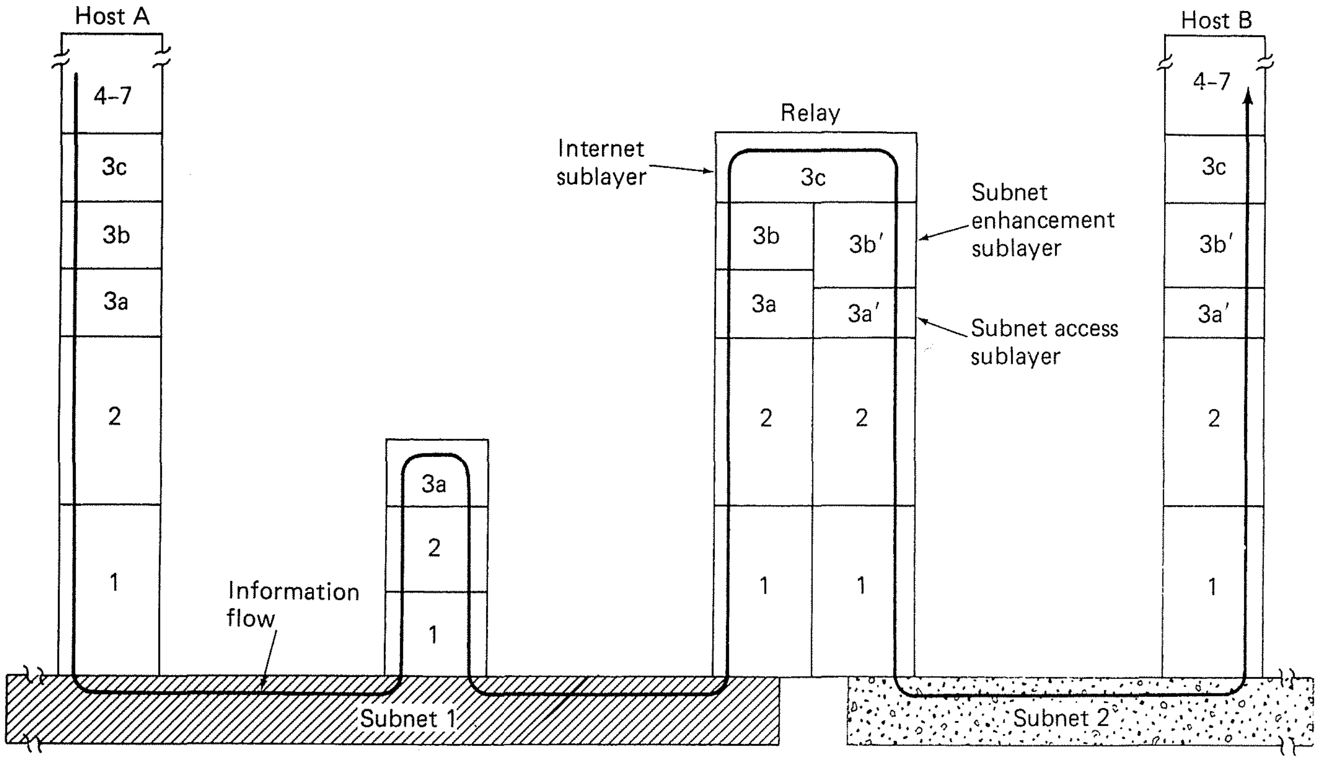


Fig. 5-27. The internal structure of the network layer. The solid line shows how information flows from host *A* to host *B*. The relay is connected to both subnets.

ultimately used to refer to transport layer processes, not to machines. The *N-CONNECT* primitives all use NSAP addresses as parameters (see Fig. 5-4).

Let us first see how a network connection is established in a subnet that conforms to the OSI model. When the connection request comes in from above, the internet sublayer passes it down to the subnet access sublayer (the enhancement sublayer is null for OSI subnets since they need not be enhanced—they are fine as is). The subnet access sublayer constructs a *CALL REQUEST* packet containing the caller and callee's NSAP addresses and gives it to the data link layer for transmission. Later it receives a reply and the connection is established.

Now let us consider what happens if the subnet does not conform to the OSI model, but, for example, uses the 1980 version of the X.25 protocol. This protocol has a *CALL REQUEST* packet, but the addresses used in it are machine addresses, not NSAP addresses. There is no convenient place to put the NSAP addresses. What happens is that the subnet enhancement sublayer first sets up a network layer connection to the proper machine. Then it sends a special data packet containing the NSAP addresses. The result of this extra exchange is that the subnet enhancement sublayer can offer a service (connection to a specific NSAP) that the subnet access sublayer cannot. As packets move along this connection, the subnet enhancement sublayer intercepts each one, routing it to the proper NSAP.

In this manner the actual subnet service is brought up to the level demanded by the internet sublayer. The effect of this enhancement is that the internet sublayer can assume that the subnet provides OSI service, even if it does not. Since internetworking frequently involves connecting one or more nonstandard networks, having a structural way to deal with strange subnets is essential.

In the previous example, the subnet service was not good enough (it lacked NSAP addressing). It can also happen that the subnet service is too good, and must be de-enhanced (degraded?) to match up to what the internet sublayer requires. One example is a relay between a datagram subnet and a virtual circuit subnet.

The internet sublayer can be designed with either type of service in mind. If datagram service has been chosen, then it is up to the subnet enchancement layer on the virtual circuit side to hide the virtual circuits and just provide datagram service to the internet sublayer. If it cannot devise any better strategy, for every datagram offered to it, it can establish a virtual circuit, send the datagram, and then release the virtual circuit. In practice, the subnet enhancement sublayer would not release the virtual circuit until it had been idle for several minutes because there is a high probability it could be used again.

The principal task of the internet sublayer is end-to-end routing. When a packet arrives at a relay, it works its way up to the internet sublayer, which examines it and decides whether to forward it, and if so, using which subnet (a multilateral relay may have several subnets to choose from). To a first approximation, routing across multiple subnets is similar to routing within a single subnet, and the techniques we have studied earlier are relevant. For a large internet, hierarchical routing is an obvious candidate, since it frees the relays from having to know about the internal structure of distant subnets.

The relay of Fig. 5-27 extends up as far as layer 3 and moves packets between networks in that layer. In the general (non OSI) case, relaying can be done in any layer. Four common types of relays are as follows:

Layer 1: **Repeaters** copy individual bits between cable segments.

Layer 2: **Bridges** store and forward frames between LANs.

Layer 3: **Gateways** store and forward packets between dissimilar networks.

Layer 4: **Protocol converters** provide interfacing in higher layers.

Repeaters are low-level devices that amplify just electrical signals. They are needed to provide current to drive long cables. In 802.3, for example, the timing properties of the MAC protocol (the value of τ chosen) allow cables up to 2.5 km, but the transceiver chips can only provide enough power to drive 500 meters. The solution is to use repeaters to extend the cable length where that is desired.

Unlike repeaters, which copy the bits as they arrive, bridges are store-and-forward devices. A bridge accepts an entire frame and passes it up to the data link layer where the checksum is verified. Then the frame is sent down to the physical layer for forwarding on a different subnet. Bridges can make minor changes to the frame before forwarding it, such as adding or deleting some fields from the frame header. Since they are data link layer devices, they do not deal with headers at layer 3 and above, and cannot make changes or decisions that depend on them.

Gateways are conceptually similar to bridges, except that they are found in the network layer. The relay of Fig. 5-27 is a gateway. Some people use the term gateway in a generic sense, applicable to any layer, and the term **router** for a network layer gateway. In this chapter "gateway" will refer to the network layer.

As a general rule, the networks connected by a gateway can differ much more than those connected by a bridge. In Fig. 5-26, the LANs are connected by a bridge; the LAN-WAN and WAN-WAN relays are gateways. A major advantage of gateways over bridges is that they can connect networks with incompatible addressing formats, for example, an 802 LAN using 48-bit binary addresses and an X.25 network using 14 decimal digit X.121 addresses.

At the transport layer and above, the relays are usually called protocol converters, although the term "gateway" is used by some people, as mentioned above. The job of a protocol converter is much more complex than that of a gateway. The protocol converter must convert from one protocol to another without losing too much meaning in the process. An example of a protocol converter is a relay that translates the OSI transport protocol to the protocol used in the ARPA Internet (TCP). Another example of protocol conversion is converting OSI mail messages (MOTIS) to ARPA Internet format (RFC 822).

Regardless of which layer the relaying is done in, the complexity of the job depends mostly on how similar the two networks are in terms of frames, packets, messages, and protocols. Some of the ways networks can differ are frame, packet, and message size, checksum algorithms, maximum packet lifetimes, connection-oriented vs. connectionless protocols, and timer values. Sometimes the conversion is not even possible, for example, when trying to forward expedited data (someone hit the DEL key) through a network not having any concept of expedited data.

5.4.2. Bridges

In this section we will look at bridge design. In the following ones we will study gateways. In a sense, the material about bridges might logically have been covered in Chapter 3 or Chapter 4, but as a convenience to the reader, we have decided to put all the material on internetworking in one place. Most bridges connect 802 LANs, so we will concentrate primarily on 802 bridges.

Before getting into the technology of bridges, it is worthwhile taking a look at some common situations in which bridges are used. We will mention six reasons why a single organization may end up with multiple LANs. First, many university and corporate departments have their own LANs, primarily to connect their own personal computers, workstations, and minicomputers. Since the goals of the various departments differ, different departments choose different LANs, without regard to what other departments are doing. Sooner or later, there is a need for interaction, so a bridge is needed. In this example, multiple LANs came into existence due to the autonomy of their owners.

Second, the organization may be geographically spread over several buildings separated by considerable distances. It may be cheaper to have separate LANs in each building and connect them with bridges and infrared links than to run a single coaxial cable over the entire campus.

Third, it may be necessary to split what is logically a single LAN into separate LANs to accommodate the load. At Carnegie-Mellon University, for example, thousands of workstations are available for student and faculty computing (Morris, 1988; Morris et al., 1986). Files are normally kept on file server machines, and are downloaded to users' machines upon request. The enormous scale of this system precludes putting all the workstations on a single LAN—the total bandwidth needed is far too high. Instead multiple LANs connected by bridges are used, as shown in Fig. 5-28. Each LAN contains a cluster of workstations with its own file server, so that most traffic is restricted to a single LAN.

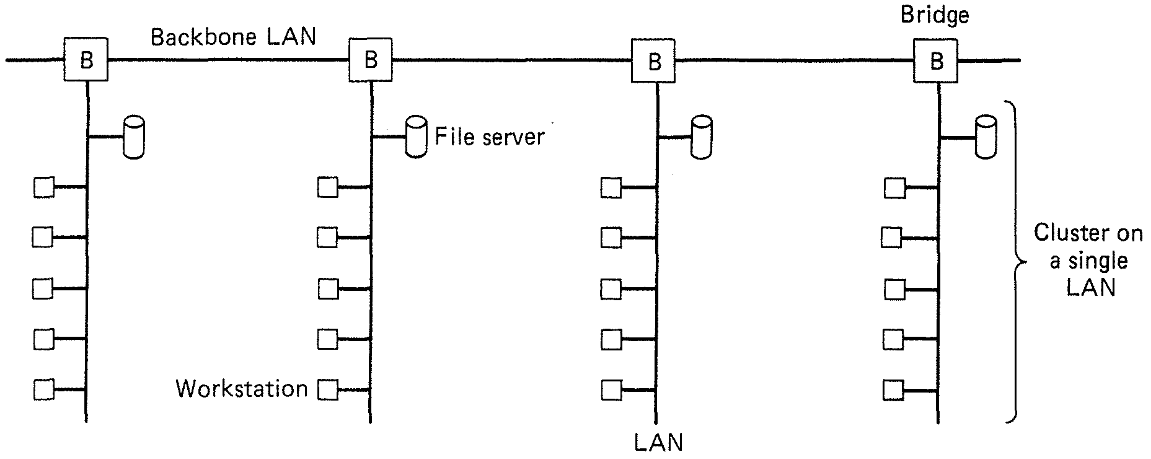


Fig. 5-28. Multiple LANs connected by a backbone to handle a total load higher than the capacity of a single LAN.

Fourth, in some situations, a single LAN would be adequate in terms of the load, but the physical distance between the most distant machines is too great (e.g., more than 2.5 km for 802.3). Even if laying the cable is easy to do, the network would not work due to the excessively long round-trip delay. The only solution is to partition the LAN and install bridges between the segments.

Fifth, there is the matter of reliability. On a single LAN, a defective node that keeps outputting a continuous stream of garbage will cripple the LAN. Bridges can be inserted at critical places, like fire doors in a building, to prevent a single node gone berserk from bringing down the entire system. Unlike a repeater, which just copies whatever it sees, a bridge can be programmed to exercise some discretion about what it forwards and what it does not forward.

Sixth, and last, bridges can contribute to the organization's security. Most LAN interfaces have a **promiscuous mode**, in which *all* packets are given to the computer, not just those addressed to it. Spies and busybodies love this feature. By inserting bridges at various places and being careful not to forward sensitive traffic, it is possible to isolate parts of the network so that its traffic cannot escape.

Having seen why bridges are needed, let us now turn to the question of how they work. Figure 5-29 illustrates the operation of a simple bilateral bridge. Host *A* has a packet to send. The packet descends into the LLC sublayer and acquires an LLC header. Then it passes into the MAC sublayer and an 802.3 header is prepended to it (also a trailer, not shown in the figure). This unit goes out onto the cable and eventually is passed up to the MAC sublayer in the bridge, where the 802.3 header is stripped off. The bare packet (with LLC header) is then handed off to the LLC sublayer in the bridge. In this example, the packet is destined for an 802.4 subnet connected to the bridge, so it works its way down the 802.4 side of the bridge and off it goes. Note that a bridge connecting *k* different LANs will have *k* different MAC sublayers and *k* different physical layers, one for each type.

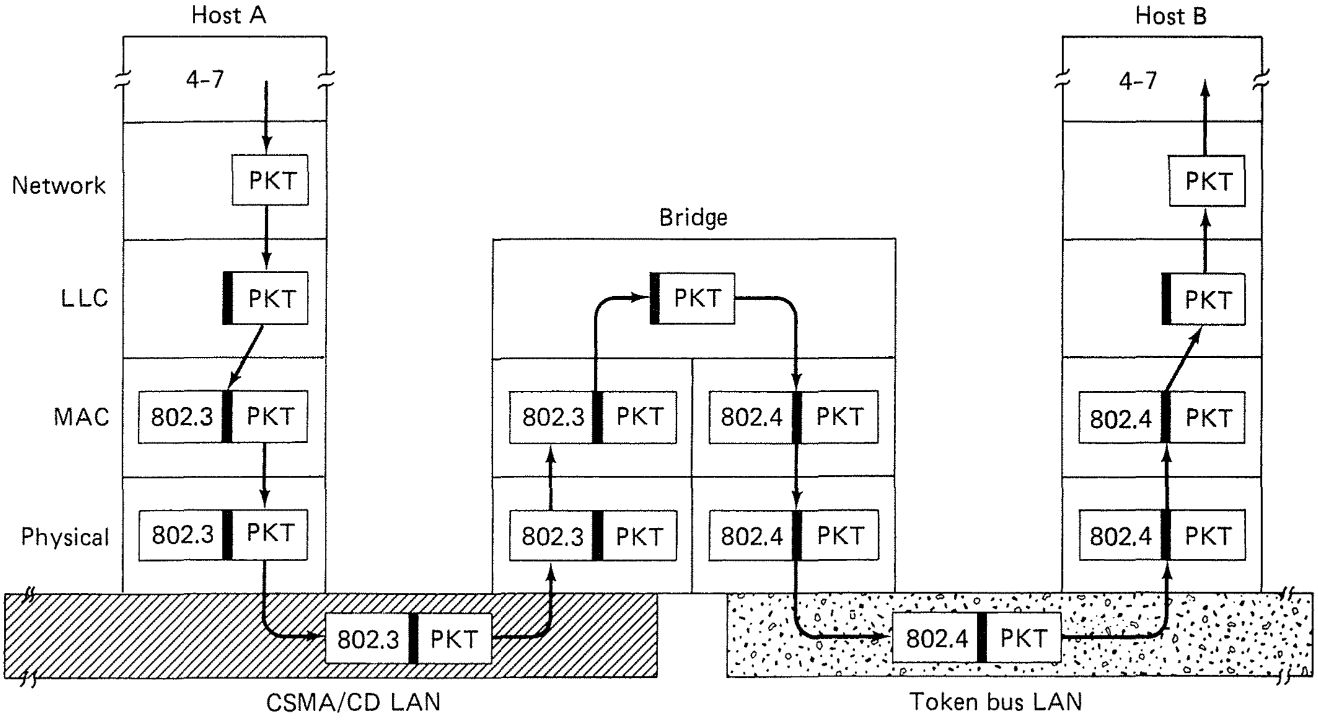


Fig. 5-29. Operation of a LAN bridge from 802.3 to 802.4.

Bridges from 802.x to 802.y

You might naively think that a bridge from one 802 LAN to another one would be completely trivial. Such is not the case. In the remainder of this section we will point out some of the difficulties that will be encountered when trying to build a bridge between the various 802 LANs. More details can be found in Berntsen et al. (1985) and Hawe et al. (1984).

Each of the nine combinations of 802.x to 802.y has its own unique set of problems. However, before dealing with these one at a time, let us look at some general problems common to all the bridges. To start with, each of the LANs uses a different frame format (see Fig. 5-30). There is no valid technical reason for this incompatibility. It is just that none of the corporations supporting the three standards (Xerox, GM, and IBM) wanted to change *theirs*. As a result, any copying between different LANs requires reformatting, which takes CPU time, requires a new checksum calculation, and introduces the possibility of undetected errors due to bad bits in the bridge's memory. None of this would have been necessary if the three committees had been able to agree on a single format.

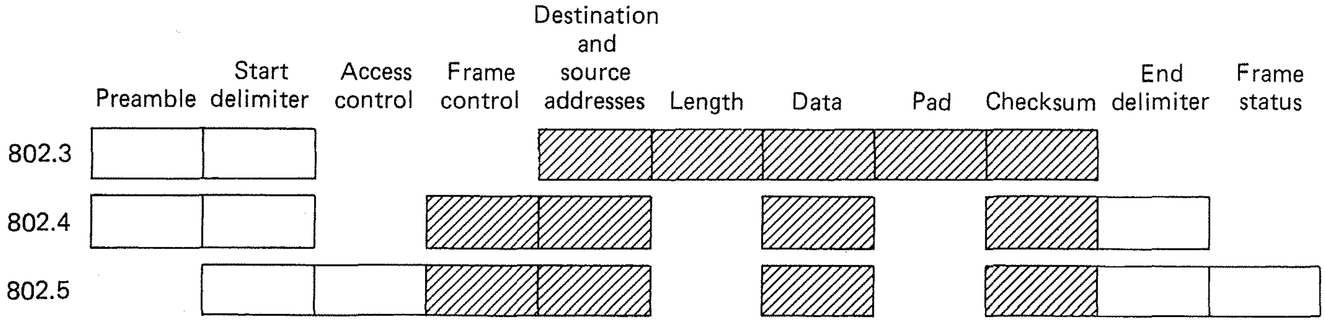


Fig. 5-30. The IEEE 802 frame formats.

A second, and far more serious problem, is that interconnected LANs do not necessarily run at the same data rate. Each of the standards allows a variety of speeds. The 802.3 standard allows 1 to 20 Mbps; the 802.4 standard permits various speeds from 1 to 10 Mbps; finally, the 802.5 standard calls for 1 or 4 Mbps. In practice, 802.3 is 10 Mbps, 802.4 is frequently 10 Mbps, and 802.5 is 4 Mbps.

When forwarding a long run of back-to-back frames from 802.3 or 802.4 to 802.5, a bridge will not be able to get rid of the frames as fast as they come in. It will have to buffer them, hoping not to run out of memory. The problem also exists from 802.4 to 802.3 to some extent because some of 802.3's bandwidth is lost to collisions. It does not really have 10 Mbps, whereas 802.4 really does.

A subtle, but important problem related to the bridge-as-bottleneck problem is the value of timers in the higher layers. Suppose the network layer on an 802.4 LAN is trying to send a very long message as a sequence of frames. After sending the last one it starts a timer to wait for an acknowledgement. If the message has to transit a bridge to an 802.5 LAN, there is a danger that the timer will go off before the last frame has been forwarded onto the slower LAN. The network layer will assume the problem is due to a lost frame, and just retransmit the entire sequence again. After *n* failed attempts it may give up and tell the transport layer that the destination is dead. Precisely this problem with mismatched speeds at a gateway was reported by Nagle (1984) in a slightly different context.

A third, and potentially most serious problem of all, is that all three 802 LANs have a different maximum frame length. For 802.3 it depends on the parameters of the configuration, but for the standard 10-Mbps system it is 1518 bytes. For 802.4 it is fixed at 8191 bytes. For 802.5 there is no upper limit, except that a station may not transmit longer than the token holding time. With the default value of 10 msec, the maximum frame length is 5000 bytes.

The obvious problem is what happens when a long frame must be forwarded onto a LAN that cannot accept it? Splitting the frame into pieces is out of the question in this layer. All the protocols assume that frames either arrive or they do not. There is no provision for reassembling frames out of smaller units. This is not to say that such protocols could not be devised. They could be and have been. It is just that 802 does not provide this feature. Basically, there is no solution. Frames that are too large to be forwarded must be discarded. So much for transparency.

Now let us briefly consider each of the nine cases of 802.x to 802.y bridges to see what other problems are lurking in the shadows. From 802.3 to 802.3 is easy. The only thing that can go wrong is that the destination LAN is so heavily loaded that frames keep pouring into the bridge, but the bridge cannot get rid of them. If this situation persists long enough, the bridge might run out of buffer space and begin dropping frames. Since this problem is always potentially present when forwarding onto an 802.3 LAN, we will not mention it further. With the other two LANs, each station, including the bridge is guaranteed to acquire the token periodically, and cannot be shut out for long intervals.

From 802.4 to 802.3 two problems exist. First, 802.4 frames carry priority bits that 802.3 frames do not have. As a result, if two 802.4 LANs communicate via an 802.3 LAN, the priority will be lost the intermediate LAN.

The second problem is caused by a specific feature in 802.4: temporary token handoff. It is possible for an 802.4 frame to have a header bit set to 1 to temporarily pass the token to the destination, to let it send an acknowledgement frame. However, if such a frame is forwarded by a bridge, what should the bridge do? If it sends an acknowledgement frame itself, it is lying because the frame really has not been delivered yet. In fact, the destination may be dead.

On the other hand, if it does not generate the acknowledgement, the sender will almost assuredly conclude that the destination is dead and report back failure to its superiors. There does not seem to be any way to solve with this problem.

From 802.5 to 802.3 we have a similar problem. The 802.5 frame format has *A* and *C* bits in the frame status byte. These bits are set by the destination to tell the sender whether the station addressed saw the frame, and whether it copied it. Again here, the bridge can lie and say the frame has been copied, but if it later turns out that the destination is down, serious problems may arise. In essence, the insertion of a bridge into the network has changed the semantics of the bits.

From 802.3 to 802.4 we have the problem of what to put in the priority bits. A good case can be made for having the bridge retransmit all frames at the highest priority, because they have probably suffered enough delay already.

From 802.4 to 802.4 the only problem is what to do with the temporary token handoff. At least here we have the possibility of the bridge managing to forward the frame fast enough to get the response before the timer runs out. Still it is a gamble. By forwarding the frame at the highest priority, the bridge is telling a little white lie, but it thereby increases the probability of getting the response in time.

From 802.5 to 802.4 we have the same problem with the *A* and *C* bits as before. Also, the definition of the priority bits is different for the two LANs, but beggars can't be choosers. At least the two LANs have the same number of priority bits. All the bridge can do is copy the priority bits across and hope for the best.

From 802.3 to 802.5 the bridge must generate priority bits, but there are no other special problems. From 802.4 to 802.5 there is a potential problem with frames that are too long and the token handoff problem is present again. Finally, from 802.5 to 802.5 the problem is what to do with the *A* and *C* bits again. Figure 5-31 summarizes the various problems we have been discussing.

		Destination LAN		
		802.3(CSMA/CD)	802.4 (Token bus)	802.5 (Token ring)
Source LAN	802.3		1, 4	1, 2, 4, 8
	802.4	1, 5, 9, 8, 10	9	1, 2, 3, 8, 9, 10
	802.5	1, 2, 5, 6, 7, 10	1, 2, 3, 6, 7	6, 7

Actions:
1. Reformat the frame and compute new checksum.
2. Reverse the bit order.
3. Copy the priority, meaningful or not.
4. Generate a ficticious priority.
5. Discard priority.
6. Drain the ring (somehow).
7. Set A and C bits (by lying).
8. Worry about congestion (fast LAN to slow LAN).
9. Worry about token handoff ACK being delayed or impossible.
10. Panic if frame is too long for destination LAN.

Parameters assumed:

802.3:	1518-byte frames,	10 Mbps (minus collisions)
802.4:	8191-byte frames	10 Mbps
802.5:	5000-byte frames	4 Mbps

Fig. 5-31. Problems encountered in building bridges from 802.x to 802.y.

When the IEEE 802 committee set out to come up with a LAN standard, it was unable to agree on a single standard, so it produced *three* incompatible standards, as we have just seen in some detail. For this failure, it has been roundly criticized. When it was later assigned the job of designing a standard for bridges to interconnect its three incompatible LANs, it resolved to do better. It did. It came up with *two* incompatible bridge designs. So far nobody has asked it to design a gateway standard to connect its two incompatible bridges, but at least the trend is in the right direction.

This section has dealt with the problems encountered in connected two LANs via a single bridge. The next two sections deal with the problems of connecting large internetworks containing many LANs and many bridges and the two IEEE approaches to designing these bridges.

Transparent Bridges

The first 802 bridge is a **transparent bridge** or **spanning tree bridge** (Backes, 1988). The overriding concern of the people who supported this design was complete transparency. In their view, a site with multiple LANs should be able to go out and buy bridges designed to the IEEE standard, plug the connectors into the bridges, and everything should work perfectly, instantly. There should be no hardware changes required, no software changes required, no setting of address switches, no downloading of routing tables or parameters, nothing. Just plug in the cables and walk away. The existing LANs should not be affected by the bridges at all. Surprisingly enough, they actually succeeded.

The transparent bridge operates in promiscuous mode, accepting every frame transmitted on all the LANs to which it is attached. As an example, consider the configuration of Fig. 5-32. Bridge 1 is connected to LANs 1 and 2, and bridge 2 is connected to LANs 2, 3, and 4. A frame arriving at bridge 1 on LAN 1 destined for *A* can be discarded immediately, because it is already on the right LAN, but a frame arriving on LAN 1 for *B*, *C*, or *D* must be forwarded.

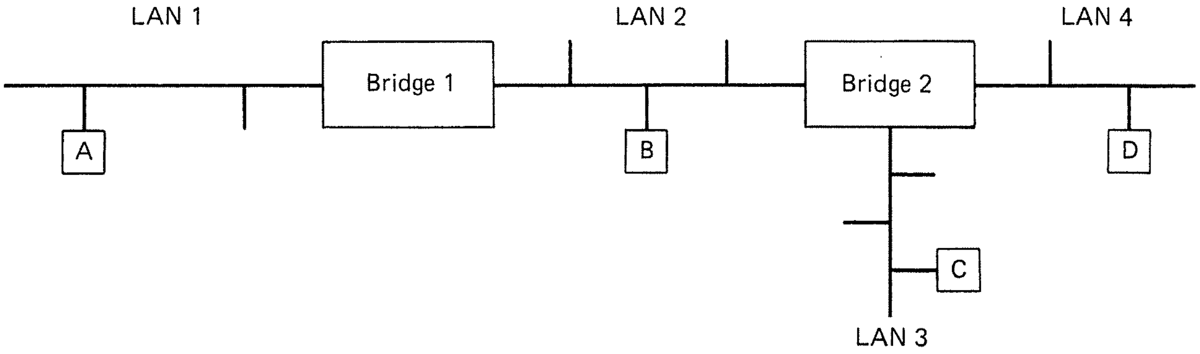


Fig. 5-32. A configuration with four LANs and two bridges.

When a frame arrives, a bridge must decide whether to discard or forward it, and if the latter, on which LAN to put the frame. This decision is made by looking up the destination address in a big (hash) table inside the bridge. The table can list each possible destination, and tell which output line (LAN) it belongs on. For example, bridge 2's table would list *A* as belonging to LAN 2, since all bridge 2 has to know is which LAN to put frames for *A* on. That, in fact, more forwarding happens later is not of interest to it.

When the bridges are first plugged in, all the hash tables are empty. None of the bridges know where any of the destinations are, so they use the flooding algorithm: every incoming frame for an unknown destination is output on all the LANs to which the bridge is connected except the one it arrived on. As time goes on, the bridges learn where destinations are, as described below. Once a destination is known, frames destined for it are put on only the proper LAN, and are not flooded.

The algorithm used by the transparent bridges is Baran's backward learning. As mentioned above, the bridges operate in promiscuous mode, so they see every frame sent on any of their LANs. By looking at the source address, they can tell which machine is accessible on which LAN. For example, if bridge 1 in Fig. 5-32 sees a frame on LAN 2 coming from *C* it knows that *C* must be reachable via LAN 2, so it makes an entry in its hash table noting that frames going to *C* should use LAN 2. Any subsequent frame addressed to *C* coming in on LAN 1 will be forwarded, but a frame for *C* coming in on LAN 2 will be discarded.

The topology of the internetwork can change as machines and bridges are powered up and down and moved around. To handle dynamic topologies, whenever a hash table entry is made, the arrival time of the frame is noted in the entry. Whenever a frame that is already in the table arrives, its entry is updated with the current time. Thus the time associated with every entry tells the last time a frame from that machine was seen.

Periodically, a process in the bridge scans the hash table and purges all entries more than a few minutes old. In this way, if a computer is unplugged from its LAN, moved around the building, and replugged in somewhere else, within a few minutes it will be back in normal operation, without any manual intervention. This algorithm also means that if a machine is quiet for a few minutes, any traffic sent to it will have to be flooded, until it next sends a frame itself.

The routing procedure for an incoming frame depends on the LAN it arrives on (the source LAN) and the LAN its destination is on (the destination LAN), as follows:

1. If destination and source LANs are the same, discard the frame.
2. If the destination and source LANs are different, forward the frame.
3. If the destination LAN is unknown, use flooding.

As each frame arrives, this algorithm must be applied. Special purpose VLSI chips exist to do the lookup and update the table entry, all in a few microseconds.

To increase reliability, some sites use two or more bridges in parallel between pairs of LANs, as shown in Fig. 5-33. This arrangement, however, also introduces some additional problems because it creates loops in the topology.

A simple example of these problems can be seen by observing how a frame, *F*, with unknown destination is handled in Fig. 5-33. Each bridge, following the normal rules for handling unknown destinations, uses flooding, which in this example, just means copying it to LAN 2. Shortly thereafter, bridge 1 sees F_2, a frame with an unknown destination, which it copies to LAN 1, generating F_3. Similarly, bridge

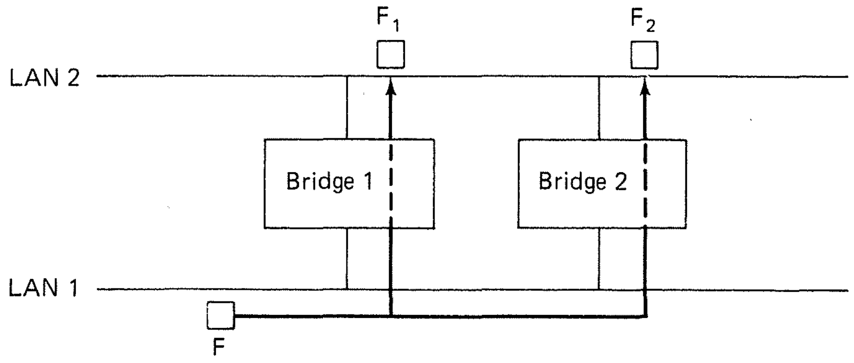


Fig. 5-33. Two parallel transparent bridges.

2 copies F_1 to LAN 1 generating F_4. Bridge 1 now forwards F_4 and bridge 2 copies F_3. This cycle goes on forever.

The solution to this difficulty is for the bridges to communicate with each other and overlay the actual topology with a spanning tree that reaches every LAN. Fig. 5-16(b) shows one of the many spanning trees that can be overlayed on the network of Fig. 5-16(a). Once the bridges have agreed on the spanning tree, all forwarding between LANs follows the spanning tree. Since there is a unique path from each source to each destination, loops are impossible.

To build the spanning tree, every few seconds each bridge broadcasts its identity (e.g., a serial number installed by the manufacturer and guaranteed to be unique) and the list of all other bridges it knows about on its LANs. A distributed algorithm is then used to select one bridge as the root of the tree, for example, the bridge with the lowest serial number. Once the root is selected, the tree is constructed by having each bridge choose the shortest path to the root. In case of ties, the lowest serial number wins.

The result of this algorithm is that a unique path is established from every LAN to the root, and thus to every other LAN. Although the tree spans alls the LAN, not all the bridge are necessarily present in the tree (to prevent loops). Even after the spanning tree has been established, the algorithm continues to run in order to automatically detect topology changes and update the tree.

Bridges can also be used to connect LANs that are widely separated. In this model, each site consists of a collection of LANs and bridges, one of which has a connection to a WAN. Frames for remote LANs travel over the WAN. The basic spanning tree algorithm can be used, preferably with certain optimizations to select a tree that minimizes the amount of WAN traffic. Hart (1988) discusses bridging over WANs in more detail.

When the internetwork becomes very large, problems of scale appear. For example, when each of the 150 million telephones in the United States is eventually replaced by an intelligent telephone, the basic spanning tree algorithm will take much too long to run. An algorithm that can handle large networks by partitioning them into multiple communicating spanning trees is described by Sincoskie and Cotton (1988).

Source Routing Bridges

Transparent bridges have the advantage of being easy to install. You just plug them in and walk away. On the other hand, they do not make optimal use of the bandwidth, since they only use a subset of the topology (the spanning tree). The relative importance of these two (and other) factors led to a split within the 802 committees (Pitt, 1988). The CSMA/CD and token bus people chose the transparent bridge. The ring people (with encouragement from IBM) preferred a scheme called **source routing**, which we will now describe. For addition details, see (Dixon and Pitt, 1988; Hamner and Samsen, 1988; and Pitt and Winkler, 1987).

Reduced to its barest essentials, source routing assumes that the sender of each frame knows whether or not the destination is on its own LAN. When sending a frame to a different LAN, the source machine sets the high-order bit of the destination address to 1, to mark it. Furthermore, it includes in the frame header the exact path that the frame is to follow.

This path is constructed as follows. Each LAN has a unique 12-bit number, and each bridge has a 4-bit number that uniquely identifies it in the context of its LANs. Thus, two bridges far apart may both have number 3, but two bridges on the same LAN must have different bridge numbers. A route is then a sequence of bridge, LAN, bridge, LAN, ... numbers. Referring to Fig. 5-32, the route from *A* to *C* would be (B1, L2, B2, L3), where we have added the codes *B* and *L* for convenience, to show which items are bridges and which are LANs.

A source routing bridge is only interested in those frames with the high-order bit of the destination set to 1. For each such frame that it sees, it scans the route looking for the number of the LAN on which the frame arrived. If this LAN number is followed by its own bridge number, the bridge forwards the frame onto the LAN whose number follows its bridge number in the route. If the incoming LAN number is followed by the number of some other bridge, it does not forward the frame.

This algorithm lends itself to three possible implementations:

1. Software: the bridge runs in promiscuous mode, copying all frames to its memory to see if they have the high-order destination bit set to 1. If so, the frame is inspected further, otherwise it is not.

2. Hybrid: the bridge's LAN interface inspects the high-order destination bit and only gives it frames with the bit set. This interface is easy to build into hardware and greatly reduces the number of frames the bridge must inspect.

3. Hardware: the bridge's LAN interface not only checks the high-order destination bit, but it also scans the route to see if this bridge must do forwarding. Only frames that must actually be forwarded are given to the bridge. This implementation require the most complex hardware, but wastes no bridge CPU cycles because all irrelevant frames are screened out.

These three implementations vary in their cost and performance. The first one has no additional hardware cost for the interface, but may require a very fast CPU to handle all the frames. The last one requires a special VLSI chip, but offloads much of the processing from the bridge to the chip, so that a slower CPU can be used, or alternatively, the bridge can handle more LANs.

Implicit in the design of source routing is that every machine in the internetwork knows the exact path to every other machine. How these routes are discovered is an important part of the source routing algorithm. The basic idea is that if a destination is unknown, the source issues a broadcast frame asking where it is. This **discovery frame** is copied by every bridge so that it reaches every LAN on the internetwork. When the reply comes back, the bridges record their identity in it, so that the original sender can see the exact route taken and ultimately choose the best route.

While this algorithm clearly finds the best route (it finds *all* routes), it suffers from a frame explosion. Consider the configuration of Fig. 5-34, with N LANs linearly connected by triple bridges. Each discovery frame sent by machine A is copied by each of the three bridges on LAN 1, yielding three discovery frames on LAN 2. Each of these is copied by each of the bridges on LAN 2, resulting in nine frame on LAN 3. By the time we reach LAN N, $3^N - 1$ frames are circulating. If a dozen sets of bridges are traversed, more than half a million discovery frames will have to be injected into the last ring, causing severe congestion.

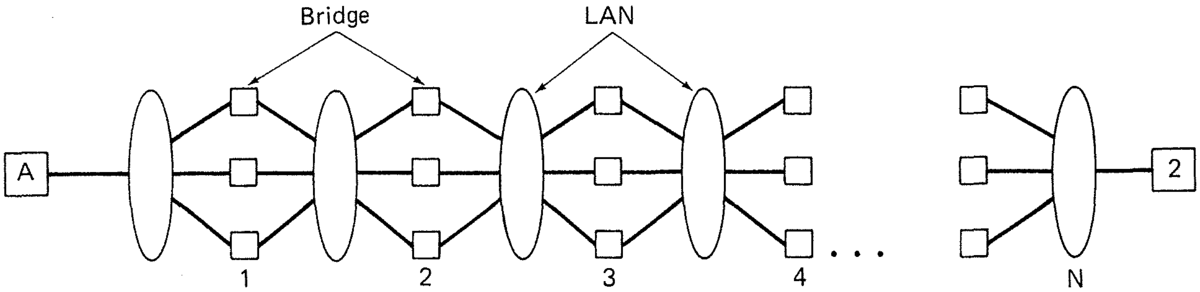


Fig. 5-34. A series of LANs connected by triple bridges.

A somewhat analogous process happens with the transparent bridge, only not nearly so severe. When an unknown frame arrives, it is flooded, but only along the spanning tree, so the total volume of frames sent is linear with the size of the network, not exponential.

Once a host has discovered a route to a certain destination, it stores the route in a cache, so that the discovery process will not have to be run next time. While this approach greatly limits the impact of the frame explosion, it does put some administrative burden on all the hosts, and the whole algorithm is definitely not transparent.

Comparison of 802 Bridges

The transparent and source routing bridges each have advantages and disadvantages. In this section we will discuss some of the major ones. They are summarized in Fig. 5-35 and covered in more detail in (Soha and Perlman, 1988; and Zhang, 1988). Be warned, however, that every one of the points is highly contested.

Issue	Transparent bridge	Source routing bridge
Orientation	Connectionless	Connection-oriented
Transparency	Fully transparent	Not transparent
Configuration	Automatic	Manual
Routing	Suboptimal	Optimal
Locating	Backward learning	Discovery frames
Failures	Handled by the bridges	Handled by the hosts
Complexity	In the bridges	In the hosts

Fig. 5-35. Comparison of transparent and source routing bridges.

At the heart of the difference between the two bridge types is the distinction between connectionless and connection-oriented networking. The transparent bridges have no concept of a virtual circuit at all, and route each frame independently from all the others. The source routing bridges, in contrast, determine a route using discovery frames and then use that route thereafter.

The transparent bridges are completely invisible to the hosts and are fully compatible with all existing 802 products. The source bridges are neither transparent nor compatible. To use source routing, hosts must be fully aware of the bridging scheme, and must actively participate in it.

When using transparent bridges, no network management is needed. The bridges configure themselves to the topology automatically. With source routing bridges, the network manager must manually install the LAN and bridge numbers. Mistakes, such as duplicating a LAN or bridge number, can be very difficult to detect, as they may cause some frames to loop, but not others on different routes. Furthermore, when connecting two previously disjoint internetworks, with transparent bridges there is nothing to do except connect them, whereas with source routing, it may be necessary to manually change many LAN numbers to make them unique in the combined internetwork.

One of the few advantages of source routing is that, in theory, it can use optimal routing, whereas transparent bridging is ,estricted to the spanning tree. Furthermore, source routing can also make good use of parallel bridges between two LANs to split the load. Whether actual bridges will be clever enough to make use of these theoretical advantages is questionable.

Locating destinations is done using backward learning in the transparent bridge and using discovery frames in source routing bridges. The disadvantage of backward learning is that the bridges have to wait until a frame from a particular machine happens to come along in order to learn where that machine is. The disadvantage of discovery frames is the exponential explosion in moderate to large internetworks with parallel bridges.

Failures handling is quite different in the two schemes. Transparent bridges learn about bridge and LAN failures and other topology changes quickly and automatically, just from listening to each other's control frames. Hosts do not notice these changes at all.

With source routing, the situation is quite different. When a bridge fails, machines that are routing over it initially notice that their frames are no longer being acknowledged, so they time out and try over and over. Finally, they conclude that something is wrong, but they still do not know if the problem is with the destination itself, or with the current route. Only by sending another discovery frame can they see if the destination is available. Unfortunately, when a major bridge fails, a large number of hosts will have to experience timeouts and send new discovery frames before the problem is resolved, even if an alternative route is available. This greater vulnerability to failures is one of the major weaknesses of all connection-oriented systems.

Finally, we come to complexity and cost, a very controversial topic. If source routing bridges have a VLSI chip that reads in only those frames that must be forwarded, these bridges will experience a lighter frame processing load and deliver a better performance for a given investment in hardware. Without this chip they will do worse because the amount of processing per frame (searching the route in the frame header) is substantially more.

In addition, source routing puts extra complexity in the hosts: they must store routes, send discovery frames, and copy route information into each frame. All of these things require memory and CPU cycles. Since there are typically one to two orders of magnitude more hosts than bridges, it may be better to put the extra cost and complexity into a few bridges, rather than in all the hosts.

5.4.3. Gateways

In contrast to bridges, gateways operate at the network level. This gives them more flexibility, for example, in translating addresses between very dissimilar networks, but it also makes them slower. As a consequence, gateways are commonly used in WANs, where no one expects them to handle more than 10,000 packets/sec, a common requirement for LAN bridges.

Two styles of gateways are common, one for connection-oriented networks and one for connectionless networks. We will study both of these in turn below.

Connection-Oriented Gateways

The OSI model permits two styles of internetworking: a connection-oriented concatenation of virtual circuit subnets, and a datagram internet style. In this section we will look at the virtual circuit approach; in the next one we will examine the datagram method.

The virtual circuit method differs from interconnection using bridges in that it happens in the network layer (see Fig. 5-27) rather than the data link layer, but it also differs in other ways. One of these differences stems from the nature of a bridge. It is a small minicomputer or microcomputer. When the bridge and all the LANs are owned by the same organization, ownership and operation of the bridge do not generate any special problems. However, when a gateway is between two WANs run by different organizations, possibly in different countries, the joint operation of a small minicomputer can lead to a lot of finger pointing.

To eliminate these problems, a slightly different approach is taken. The relay of Fig. 5-27 is effectively ripped apart in the middle and the two parts of 3c are connected with a wire, as shown in Fig. 5-36. Each of the halves is called a **half-gateway** and each one is owned and operated by one of the network operators. The whole problem of gatewaying then reduces to agreeing to a common protocol to use on the wire. As long as both parties use the common protocol on the wire, they can arrange their sublayers 3a, 3b, and 3c any way that is convenient for them.

The protocol that the half-gateways speak over the wire is CCITT's **X.75** protocol, which is almost identical to X.25 (discussed later in this chapter). The X.75 model is based on the idea of building up an internetwork connection by concatenating a series of intranetwork and half-gateway to half-gateway virtual circuits. The model is shown in Fig. 5-37(a) The connection between the source host, in one network, and the destination host, in another network, is composed of five adjacent virtual circuits, marked VC 1-5. VC 1 goes from the source host to a half-gateway (called a **signaling terminal** or **STE** by CCITT), in its own network. VC 2 goes from the half-gateway in the source network to a half-gateway in an intermediate network, assuming that there is no direct connection between the source and destination networks. VCs 3 and 5 are also intranet, like VC 1, and VC 4 is internet, like VC 2.

In this model a connection to a host in a distant network is set up the same way normal connections are established. The subnet sees that the destination is remote, chooses an appropriate (half-) gateway and builds a virtual circuit to the gateway. The first gateway records the existence of the virtual circuit in its tables and

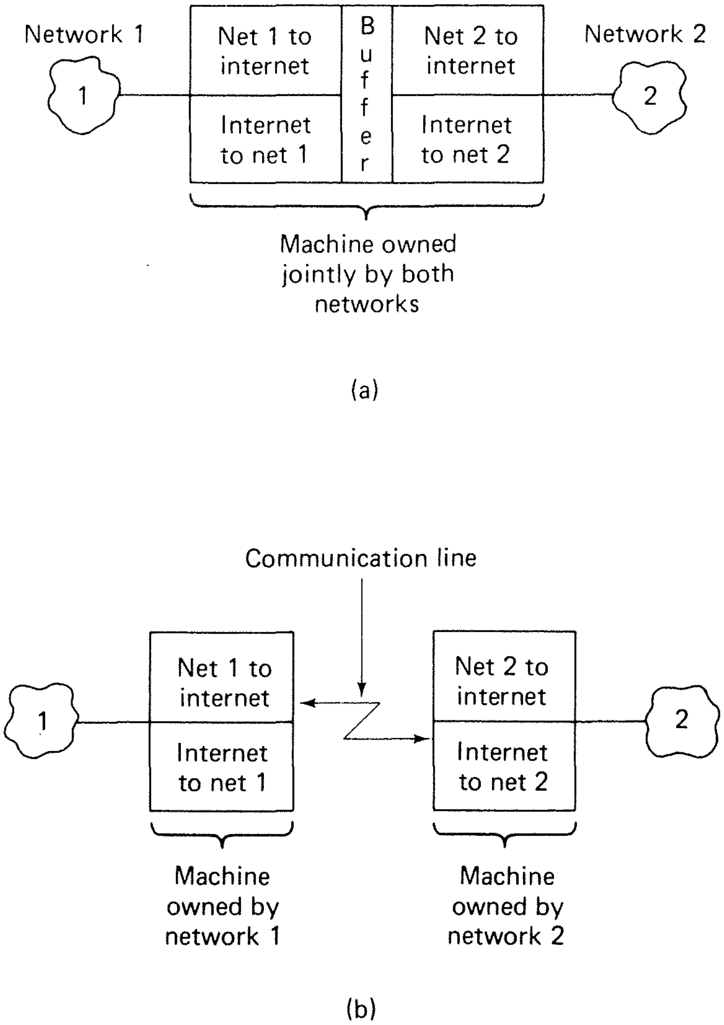
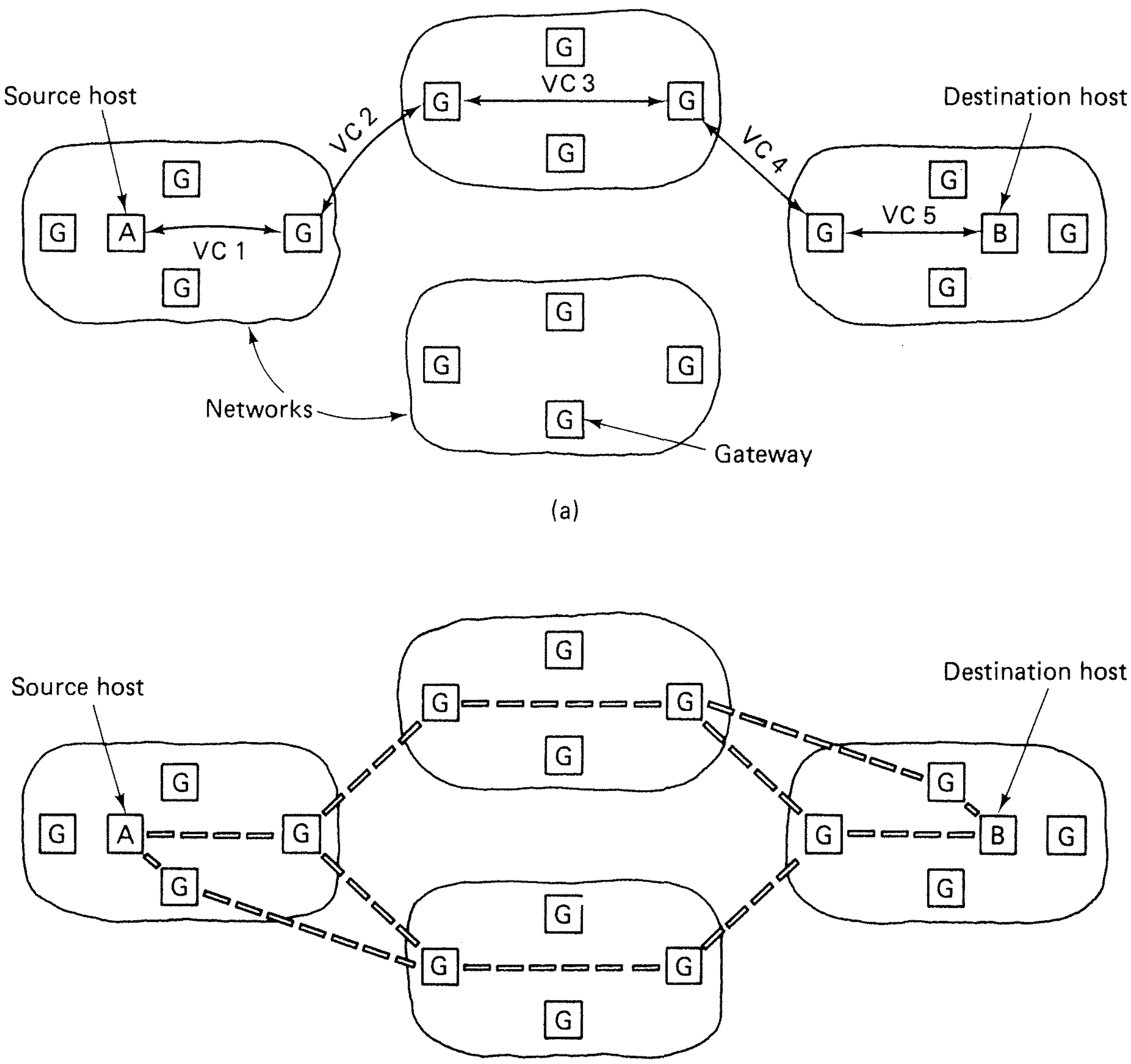


Fig. 5-36. (a) A full gateway. (b) Two half-gateways.

proceeds to build another virtual circuit to the next gateway. This process continues until the destination host has been reached.

Once data packets begin flowing along the path, each gateway relays incoming packets, converting between packet formats and virtual circuit numbers as needed. Clearly all data packets must traverse the same sequence of gateways, although VCs 1, 3, and 5 in Fig. 5-37(a) might be implemented internally using datagrams, so that not all packets need follow precisely the same route from source to destination. In practice, networks that use virtual circuits *between* networks also use them internally.

The essential feature of this approach is that a sequence of virtual circuits is set up from the source through one or more gateways to the destination. Each gateway maintains tables telling which virtual circuits pass through it, where they are to be routed, and what the new virtual circuit number is. The whole arrangement is similar to the fixed routing of Fig. 5-6, except that only the sequence of gateways is fixed, not (necessarily) the full sequence of IMPs.



(a)

(b)

Fig. 5-37. (a) Internetworking using concatenated virtual circuits. (b) Internetworking using datagrams.

Figure 5-38 shows the protocols used on the various lines when two X.25 networks are connected. The source host talks to the subnet using X.25. The internal IMP-IMP protocol is not specified by CCITT and probably will be different in every network. The protocol used between the gateways and the rest of the subnet is also left open, but here there are clearly two distinct choices: the internal IMP-IMP protocol, or X.25. If this protocol is the IMP-IMP protocol, the network regards the gateway as an IMP. If it is X.25, the subnet regards the gateway as a host.

Although the X.75 protocol itself only applies to the gateway-to-gateway lines, it effectively dictates the concatenated virtual circuit architecture by requiring all packets on a connection to pass over the same gateway-to-gateway virtual circuit in sequence. It is hard to see how different packets could use different gateways and

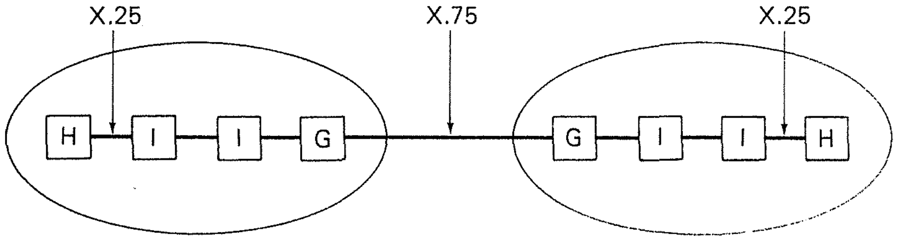


Fig. 5-38. Protocols in the CCITT internetwork model.

still meet this requirement. The X.75 protocol only differs in minor ways from X.25, such as the range of facilities available.

Connectionless Gateways

The alternative internetwork model to CCITT's is the datagram model, shown in Fig. 5-37(b). In this model, the only service the network layer offers to the transport layer is the ability to inject datagrams into the subnet, and hope for the best. There is no notion of a virtual circuit at all in the network layer, let alone a concatenation of them. This model does not require all packets belonging to one connection to traverse the same sequence of gateways. In Fig. 5-37(b) datagrams from *A* to *B* are shown taking a wide variety of different routes through the internetwork. On the other hand, no one guarantees that the packets arrive at the destination in order, assuming that they arrive at all.

To see how the internet datagram model works, let us follow the path of the transport message of Fig. 5-39 from host *A* to host *B*. Datagrams have a fixed maximum size, so if the message is longer than the limit, the transport layer chops it up into as many datagrams as necessary. It is these datagrams that move from network to network on their way from source to destination. Only at the end of their journey does the transport layer at the destination machine reassemble the datagrams back into the original message.

In order for a datagram to go from gateway to gateway in the internet, the datagram must be encapsulated in the data link format for each network it passes through. In Fig. 5-39, the datagram is given a header and a trailer to form frame 1, which is then passed transparently through network 1. When the datagram reaches gateway 1, the data link header and trailer are stripped off, and the naked datagram emerges again, like a butterfly coming out of its cocoon. While traveling through network 2, the wrapping is different, but the same datagram emerges again at the next gateway. This process of wrapping and unwrapping the datagram occurs repeatedly until the datagram reaches the destination host.

Each network imposes some maximum size on its packets. These limits have various causes, among them:

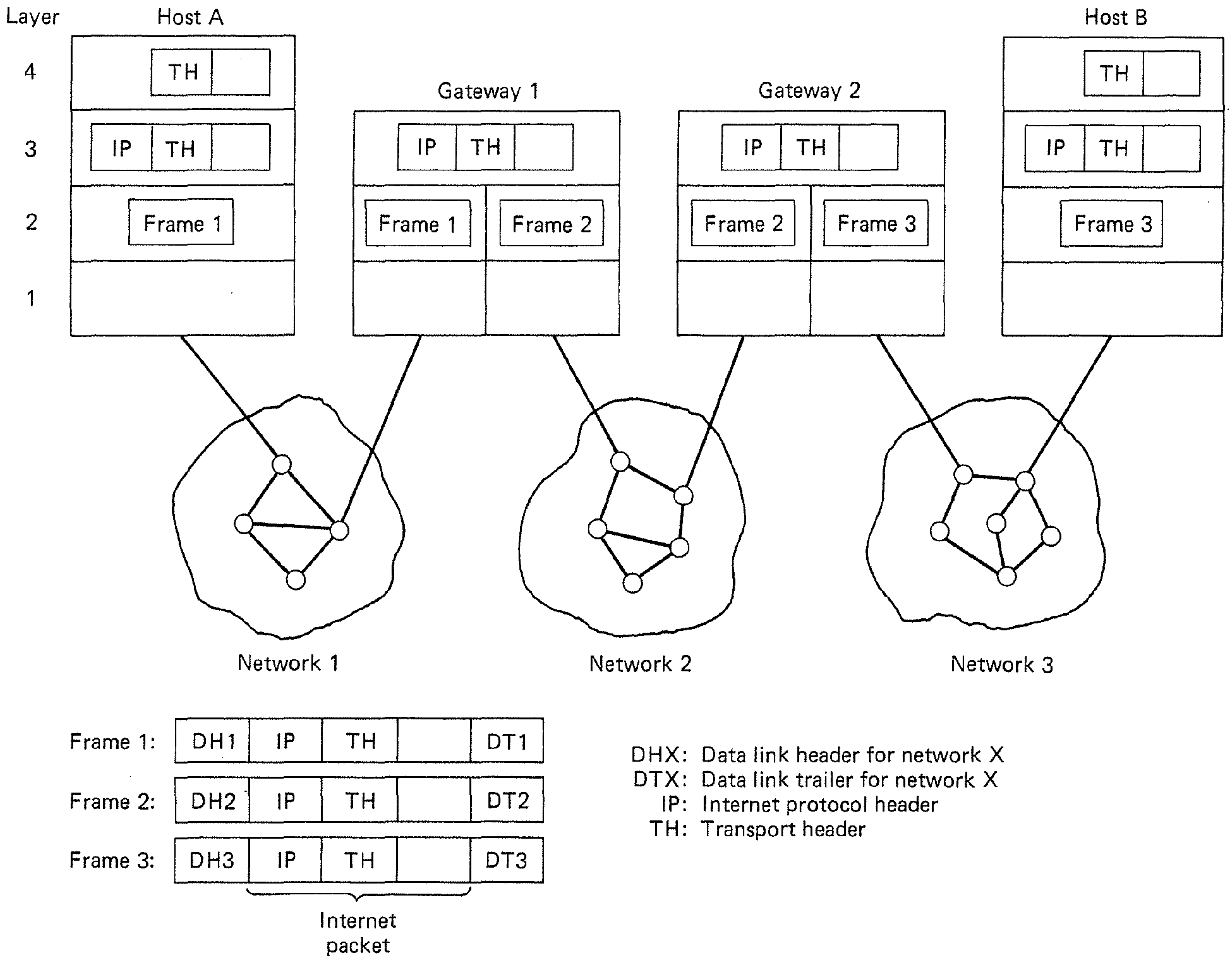


Fig. 5-39. A datagram moving from network to network.

1. Hardware (e.g., the width of a TDM transmission slot).
2. Operating system (e.g., all buffers are 512 bytes).
3. Protocols (e.g., the number of bits in the packet length field).
4. Compliance with some (inter)national standard.
5. Desire to reduce error induced retransmissions to some level.
6. Desire to prevent one packet from occupying the channel too long.

The result of all these factors is that the network designers are not free to choose any maximum packet size they wish. Some typical examples of maximum packet lengths in actual networks are:

1. HDLC: in principle infinite.
2. 802.4: 65,528 bits.
3. X.25: 32768 bits.
4. ARPA Packet Radio Network: 2032 bits.
5. ARPANET: 1008 bits.
6. University of Hawaii ALOHANET: 640 bits.

An obvious problem appears when a large packet wants to travel through a network whose maximum packet size is too small. This problem has received a great deal of attention in the literature and some proposals for solving it have been made.

One solution is to make sure the problem does not occur in the first place. In other words, the internet should use a routing algorithm that avoids sending packets through networks that cannot handle them. However, this solution is no solution at all. What happens if the original source packet is too large to be handled by the destination network? The routing algorithm can hardly bypass the destination. Nevertheless, intelligent routing can minimize the extent of the problem.

Basically, the only solution to the problem is to allow gateways to break packets up into **fragments**, sending each fragment as a separate internet packet. However, as every parent of a small child knows, converting a large object into small fragments is considerably easier than the reverse process. (Physicists have even given this effect a name: the second law of thermodynamics.) Packet-switching networks, too, have trouble putting the fragments back together again.

Two opposing strategies exist for recombining the fragments back into the original packet. The first strategy is to make fragmentation caused by a "small-packet" network transparent to any subsequent networks through which the packet must pass on its way to the ultimate destination. This option is shown in Fig. 5-40(a). When an oversized packet arrives at a gateway, the gateway breaks it up into fragments. Each fragment is addressed to the same exit gateway, where the pieces are recombined. In this way passage through the small-packet network has been made transparent. Subsequent networks are not even aware that fragmentation has occurred.

Transparent fragmentation is simple but has some problems. For one thing, the exit gateway must know when it has received all the pieces, so that either a count field or an "end of packet" bit must be included in each packet. For another thing, all packets must exit via the same gateway. By not allowing some fragments to follow one route to the ultimate destination, and other fragments a disjoint route, some performance may be lost. A third problem is possible reassembly lockup at the exit gateway. A last problem is the overhead required to repeatedly reassemble and then refragment a large packet passing through a series of small-packet networks.

The other fragmentation strategy is to refrain from recombining fragments at

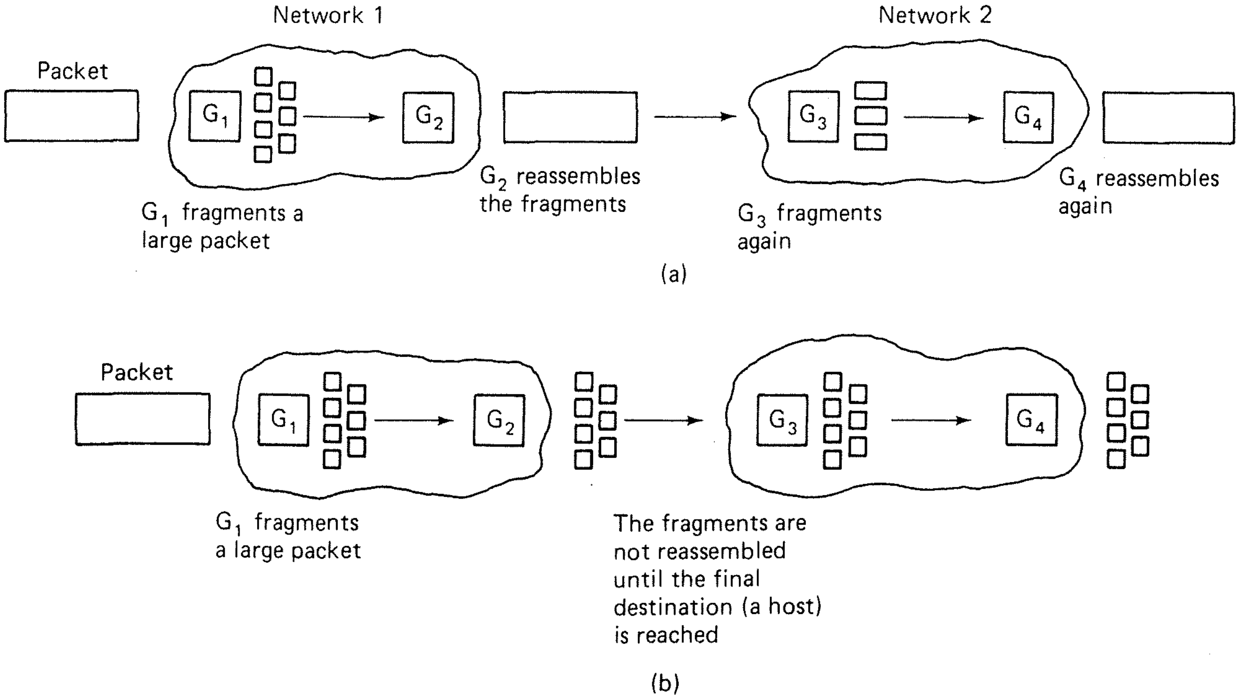

Network 1
Network 2
Packet
G1
G2
G3
G4
G1 fragments a large packet
G2 reassembles the fragments
G3 fragments again
G4 reassembles again
(a)
Packet
G1
G2
G3
G4
G1 fragments a large packet
The fragments are not reassembled until the final destination (a host) is reached
(b)

Fig. 5-40. (a) Transparent fragmentation. (b) Nontransparent fragmentation.

any intermediate gateways. Once a packet has been fragmented, each fragment is treated as though it were an original packet. All fragments are passed through the exit gateway (or gateways), as shown in Fig. 5-40(b). Recombination occurs only at the destination host.

Nontransparent fragmentation also has some problems. For example, it requires *every* host to be able to do reassembly. Yet another problem is that when a large packet is fragmented the total overhead increases, because each fragment must have a header. Whereas in the first method this overhead disappears as soon as the small packet network is exited, in this method the overhead remains for the rest of the journey. An advantage of this method, however, is that multiple exit gateways can now be used. Of course, if the concatenated virtual circuit model is being used, this advantage is of no use.

Shoch (1979) has proposed having each packet carry a bit specifying whether or not the ultimate destination is prepared to reassemble fragments. If it is not, any gateway fragmenting a packet must arrange for some other gateway to reassemble the pieces. If the ultimate destination is prepared to do reassembly itself, fragmentation may or may not be transparent, at each gateway's discretion.

When a packet is fragmented, the fragments must be numbered in such a way that the original data stream can be reconstructed. One way of numbering the fragments is to use a tree. If packet 0 must be split up, the pieces are called 0.0, 0.1, 0.2, etc. If these fragments must themselves be fragmented at a subsequent gateway, the pieces are numbered 0.0.0, 0.0.1, 0.0.2, . . . , 0.1.0, 0.1.1, 0.1.2, etc. If enough fields have been reserved in the header for the worst case and if there are no duplicates generated anywhere, this scheme is sufficient to ensure that all the pieces can be correctly reassembled at the destination, no matter what order they arrive in.

However, if even one network loses or discards packets, there is a need for end-to-end retransmissions, with unfortunate effects for the numbering system. Suppose that a 1024-bit packet is initially fragmented into four equal-sized fragments, 0.0, 0.1, 0.2, and 0.3. Fragment 0.1 is lost, but the other parts arrive at the destination. Eventually, the source times out and retransmits the original packet again. Only this time the route taken passes through a network with a 512-bit limit, so two fragments are generated. When the new fragment 0.1 arrives at the destination, the receiver will think that all four pieces are now accounted for and reconstruct the packet incorrectly.

A completely different (and better) numbering system is for the internetwork protocol to define an elementary fragment size small enough that the elementary fragment can pass through every network. When a packet is fragmented, all the pieces are equal to the elementary fragment size except the last one, which may be shorter. An internet packet may contain several fragments, for efficiency reasons. The internet header must provide the original packet number, and the number of the (first) elementary fragment contained in the frame. As usual, there must also be a bit indicating that the last elementary fragment contained within the internet packet is the last one of the original packet.

This approach requires two sequence fields in the internet header: the original packet number, and the fragment number. There is clearly a trade-off between the size of the elementary fragment and the number of bits in the fragment number. Because the elementary fragment size is presumed to be acceptable to every network, subsequent fragmentation of an internet packet containing several fragments causes no problem. The ultimate limit here is to have the elementary fragment be a single bit or byte, with the fragment number then being the bit or byte offset within the original packet, as shown in Fig. 5-41.

Some internet protocols take this method even further, and consider the entire transmission on a virtual circuit to be one giant packet, so that each fragment contains the absolute byte number of the first byte within the fragment. Some other issues relating to fragmentation are discussed in (Kent and Mogul, 1987).

5.4.4. Comparison of Connection-Oriented and Connectionless Gateways

The concatenated virtual circuit and datagram approaches have different strengths and weaknesses. The concatenated virtual circuit model has essentially the same advantages as using virtual circuits within a single subnet: buffers can be reserved in advance (in the gateways) to ease congestion, sequencing can be guaranteed, short headers can be used, and the troubles caused by delayed duplicate packets can be avoided.

It also has the same disadvantages: table space required in the gateways for each open connection, whether or not there is any traffic, no alternate routing to

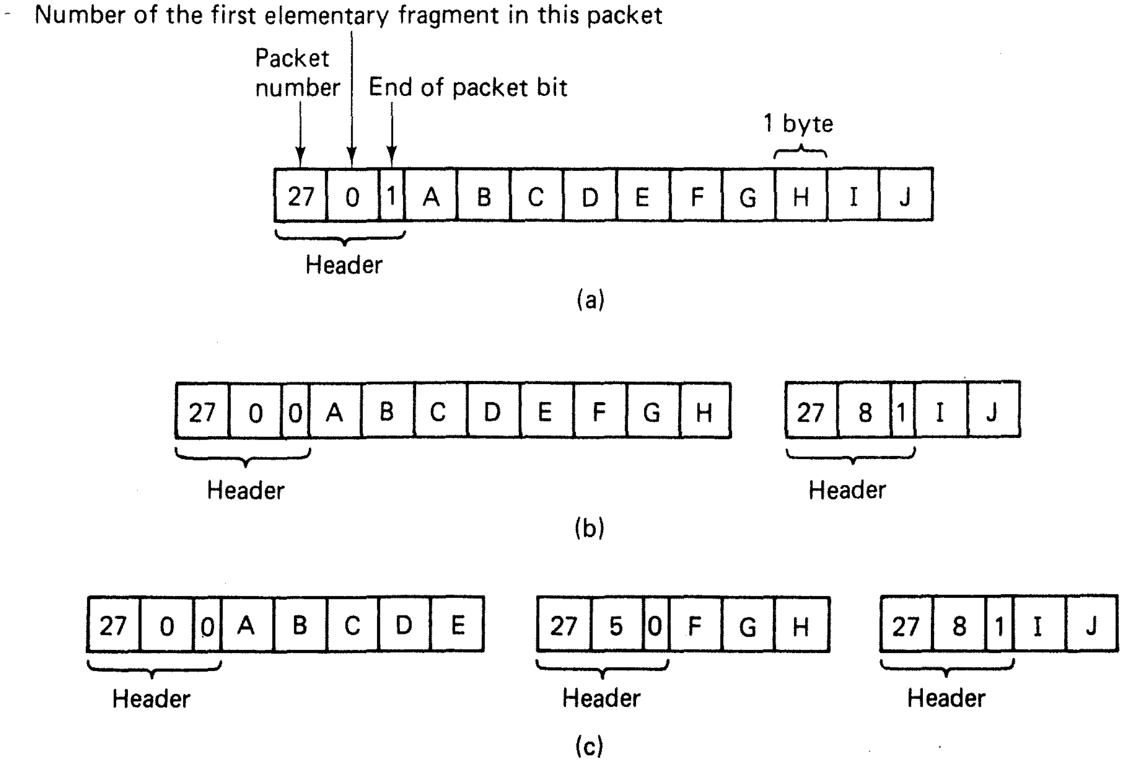


Fig. 5-41. Fragmentation when the elementary data size is 1 byte. (a) Original packet, containing 10 data bytes. (b) Fragments after passing through a network with maximum packet size of 8 bytes. (c) Fragments after passing through a size 5 gateway.

avoid congested areas, and vulnerability to gateway failures anywhere along the path. It also has the disadvantage of being difficult, if not impossible, to implement if one of the networks involved is an unreliable datagram network.

The properties of the datagram approach to internetworking are the same as those of datagram subnets: more potential for congestion, but also more potential for adapting to it, robustness in the face of gateway failures, and longer headers needed. Various adaptive routing algorithms are possible in the internet, just as they are within a single datagram network.

A major advantage of the datagram approach to internetworking is that it can be used over subnets that do not use virtual circuits inside. Many LANs, mobile networks (e.g., aircraft and naval fleets), and even some WANs fall into this category. When an internet includes one of these, serious problems occur if the internetworking strategy is based on virtual circuits.

5.4.5. Bridge and Gateway Software

The software used in a bridge or gateway is quite different from ordinary host software, and as such it is worth looking at a little bit. The reason for this difference is not hard to find. The performance requirements here are critical. Bridges and gateways are expected to accept and forward traffic from one network to another in real time without slowing down the operation of either network.

Consider the worst case requirements for a LAN bridge. Imagine that a continuous stream of 64-byte packets must be forwarded from one 10-Mbps LAN to another. The 64-byte packets arrive at a rate of one frame per 51 μsec (almost 20,000 frames/sec). Thus the bridge must get an interrupt, process a packet and retransmit it, all in a time of 51 μsec. If packets can arrive from both sides simultaneously, the required throughput is doubled. Even with intelligent front-end processors that collect the bits of the incoming packets and assemble them in memory before causing an interrupt, 51 μsec is not much time. Bridges are nearly always CPU bound.

As a consequence of the heavy workload, the software must be carefully structured for maximum efficiency. At one extreme, the entire bridge or gateway could be interrupt driven. In this model, each arriving packet causes an interrupt, which disables further interrupts and processes the packet to completion right in the interrupt routine. While this approach is very fast, it leads to poorly structured software and should be avoided.

At the other extreme, the software is divided into processes, with each process having a well-defined task. Figure 5-42 shows a simple example of a trilateral gateway (or bridge). Each of the nine processes runs in its own address space. When a packet arrives from network 1, 2, or 3, process 2, 3, or 5, respectively, is made runnable. That process verifies the checksum, converts the packet to an internal network-independent format if needed, and deposits it in the work queue for process 7, the routing process. These input processes must have the highest priority to avoid losing input packets.

When none of the high priority processes, 2, 3, or 5, or medium priority processes, 1, 4, or 6 are able to run and process 7 has work in its queue, it takes the first queued packet, determines how to route it, and puts the packet in the work queue for one of the output processes (1, 4, or 6), which converts it to the appropriate output format and starts the transmitter (if it was idle). As soon as packet transmission has started, the output process terminates to let another process run. When the interrupt comes in at the end of the transmission, the output process is enabled for running, at which time it dequeues the packet just transmitted and starts the next one.

Packet filtering is also done by the routing process. If the packet under examination is a control packet or a packet whose destination is in the same network as its source, it is not queued for output, but just ignored.

The background processes 8 and 9 run periodically at the lowest priority. This process-oriented approach is well structured but slow due to the process switching overhead (each process switch requires setting up a new memory map).

A reasonable compromise is to keep the process model, but have all the processes run within a single address space in kernel mode. For a general-purpose operating system this approach would not work, but since all the gateway processes

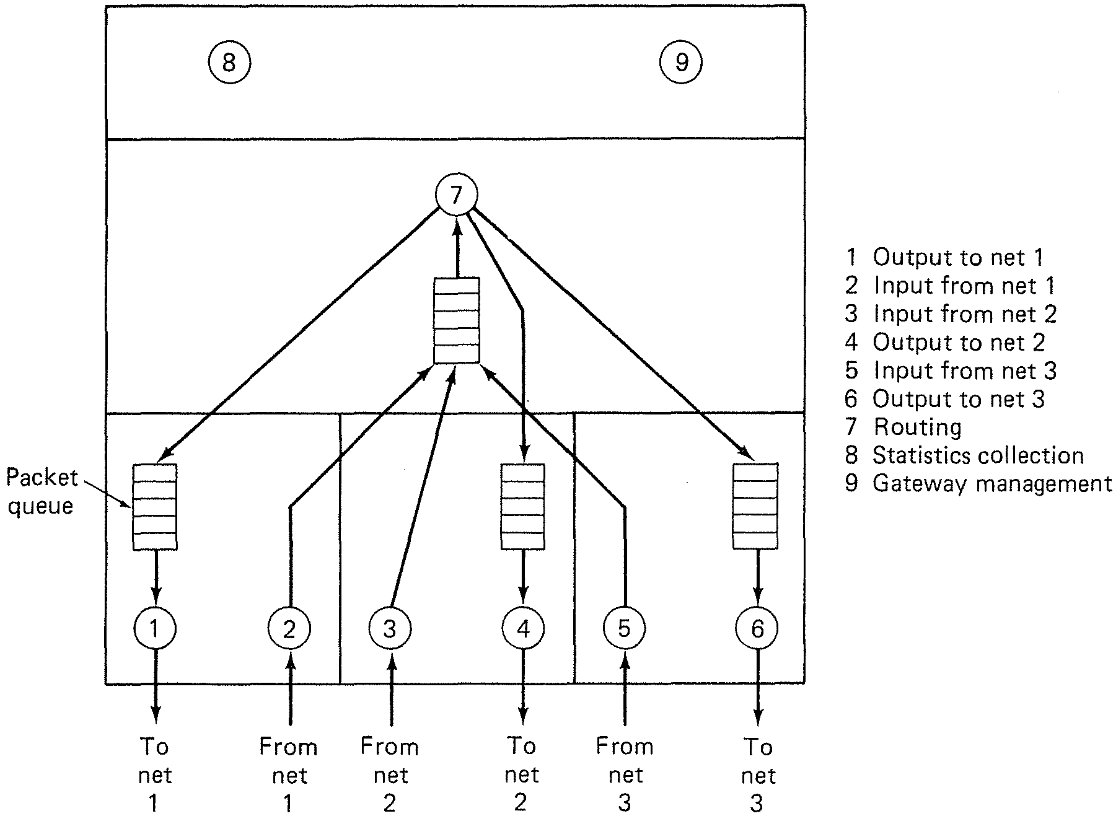


Fig. 5-42. A process-structured trilateral gateway.

can be assumed to be well-behaved, the gain in speed is worth the loss of security.

The gateway scheduling algorithm is also critical. Rather than using round-robin scheduling or some other form of time-slicing, it is probably better to let each process run to completion. Not only does run-to-completion reduce the number of process switches, but when a process finishes with the current packet it has no state information to remember so its registers and stack need not be saved. In fact, it can simply schedule its own successor by finding the highest priority runnable process and jumping to it. That process thus inherits the registers and empty stack of its predecessor. Only one stack is needed, no matter how many processes there are.

On some machines, interrupt processing overhead is substantial. One way to avoid it is to run the gateway with interrupts disabled. When a process finishes, it examines (polls) each network interface to see if a transmission has finished. If so, the corresponding process is marked as runnable, and the highest priority runnable process is executed.

The interprocess communication between the processes is critical. Actually passing messages between processes by copying them would be unbearably slow. It is better to pass buffer pointers. However, if one process allocates a packet buffer and puts a pointer to it in another process' work queue, how does the first process know when to deallocate the buffer? In general, it does not, so the receiving process must either reuse the buffer itself, or explicitly free it.

Buffer management is complicated by the fact that packets may change size as they move around the gateway, with headers being attached and stripped in various places. This effect is most pronounced if multiple protocol layers must be traversed within the gateway. One way to solve the problem is to set up the packet receive hardware to read packets into buffers not at the start of the buffer, but at a distance from the start equal to the size of the largest header. For example, if networks 1, 2, and 3 of Fig. 5-42 have headers of 8, 16, and 24 bytes, respectively, then all packets read in from network 1 should begin at byte 16, as shown in Fig. 5-43(a). That way, if the packet must eventually be transmitted onto network 3, with a 24-byte header, there is room in the buffer for the new header, as shown in Fig. 5-43(b). If packet fragmentation is potentially required, the situation becomes much more complex and a more sophisticated buffer management strategy is needed.

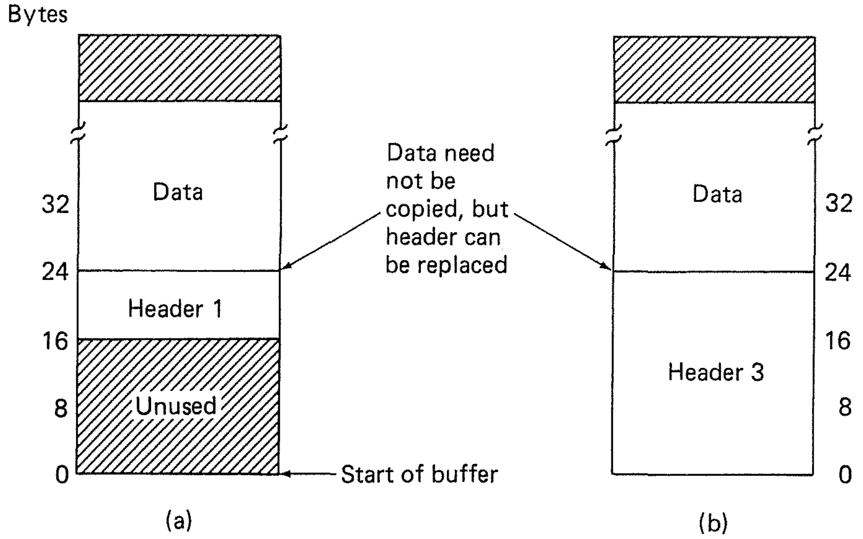


Fig. 5-43. Packets should be read in at an offset from the start of the buffer to allow enough room for the worst case header to be prepended to the data.

Timer management is another important issue. Depending on the layer and nature of the gateway, it may be necessary to wait until a packet has been acknowledged before releasing its buffer. Because most packets are not lost, most timers do not expire. Therefore, what is important is that setting a timer does not take much time. The amount of time required to handle a timer that has gone off is less critical.

Some gateways must reverse the order of the bits in a byte (e.g., for transmission between 802.3 and 802.5). The fastest way to invert the bits is to hardwire bit 0 on the input side to bit 7 on the output side, bit 1 on the input side to bit 6 on the output side, and so forth, so that the reversal is done during transmission between the interface chip and memory. If this special hardware is not available, the next best thing to do is have a 256-entry table indexed by byte value. The *i*-th entry in this table is the inverted bit pattern for byte *i*. For inversion in both directions, two 256-byte tables are needed. However, bit reversal is very expensive, even with table lookup, because it means that every byte must be examined; it is even worse than copying, since that can usually be done with a single BLOCK MOVE instruction.

It is worth noting the design of our example gateway is not all that different from the design of an ordinary IMP. There too, packets come in on various input lines, are routed, and are retransmitted. The main difference is that no format conversions or header changes are needed in the IMPs.

More details about gateways, their software, and their problems are given in (Benhamou and Estrin, 1983; Bosack and Hedrick, 1988; and Seifert, 1988). LAN-WAN gateways are described in (Folts, 1984; and Spiegelhalter, 1984).

5.5. EXAMPLES OF THE NETWORK LAYER

The above material should give you a good feeling for the choices and options available to subnet designers. In the remainder of this chapter we will examine our usual example networks to see what choices their designers actually made in practice.

5.5.1. The Network Layer in Public Networks

To prevent networks in different countries from developing mutually incompatible interfaces, in 1974 CCITT proposed international standard network access protocols for layers 1, 2, and 3. These protocols were revised in 1976, 1980, 1984, and so on, *ad infinitum.* These standards are collectively known as **X.25** The layer 3 protocol is often called the **X.25 PLP** (**Packet Layer Protocol**) to distinguish it from the lower two layers. The significance of X.25 PLP is that it is widely used as the connection-oriented network layer protocol in the OSI model.

X.25 defines the interface between the host, called a **DTE** (**Data Terminal Equipment**) by CCITT, and the carrier's equipment, called a **DCE** (**Data Circuit-terminating Equipment**) by CCITT. An IMP is known as a **DSE** (**Data Switching Exchange**). In this section we will adhere to the CCITT terminology because it is in widespread use in public network circles. X.25 defines the format and meaning of the information exchanged across the DTE - DCE interface for the layer 1, 2, and 3 protocols (see Fig. 5-44). Since this interface separates the carrier's equipment (the DCE) from the user's equipment (the DTE), it is important that the interface be very carefully defined. Notice that CCITT's use of the word "interface" differs from the OSI usage, as we have already pointed out in Fig. 2-30.

X.25 layer 1 deals with the electrical, mechanical, procedural, and functional interface between the DTE and DCE. Actually, X.25 does not define these things, but rather references two other standards, X.21 and X.21 *bis*, which define the digital and analog interfaces, respectively. We discussed X.21 in Chapter 2. X.21 *bis*

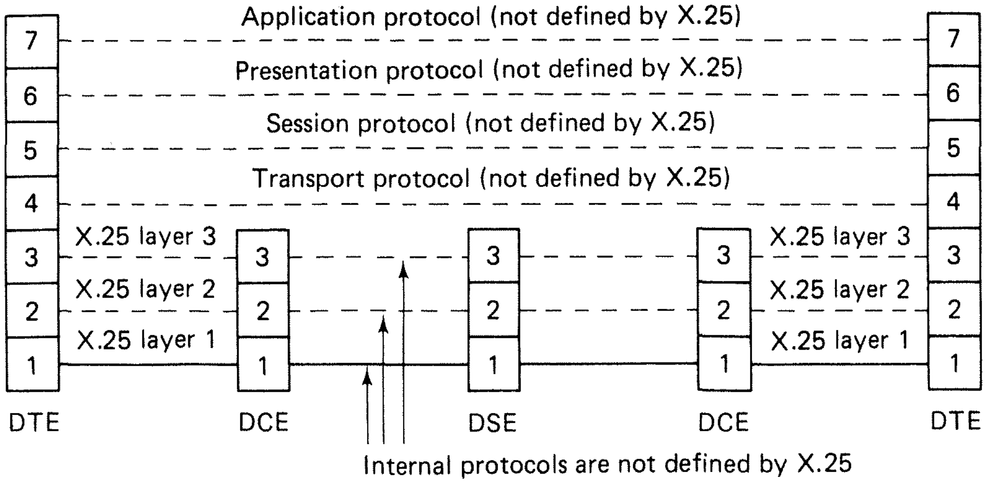

Application protocol (not defined by X.25)
Presentation protocol (not defined by X.25)
Session protocol (not defined by X.25)
Transport protocol (not defined by X.25)
X.25 layer 3
X.25 layer 2
X.25 layer 1
7
6
5
4
3
2
1
DTE
DCE
DSE
DCE
DTE
Internal protocols are not defined by X.25

Fig. 5-44. The place of X.25 in the protocol hierarchy.

is an interim standard to be used on analog networks until digital networks become widely available. It is essentially RS-232-C.

The job of layer 2 is to ensure reliable communication between the DTE and the DCE, even though they may be connected by a noisy telephone line. The protocols used are LAP and LAPB, as discussed in Chapter 4.

Layer 3 manages connections between a pair of DTEs. Two forms of connection are provided, **virtual calls** and **permanent virtual circuits**. A virtual call is like an ordinary telephone call: a connection is established, data are transferred, and then the connection is released. In contrast, a permanent virtual circuit is like a leased line. It is always present, and the DTE at either end can just send data whenever it wants to, without any setup. Permanent virtual circuits are normally used in situations with a high volume of data. Since data transfer on a permanent virtual circuit is the same as on a virtual call, we will not discuss permanent virtual circuits any further.

Connections (virtual calls in CCITT terminology) are made as follows. When a DTE wants to communicate with another DTE, it must first set up a connection. To do this, the DTE builds a *CALL REQUEST* packet and passes it to its DCE. The subnet then delivers the packet to the destination DCE, which then gives it to the destination DTE. If the destination DTE wishes to accept the call, it sends a *CALL ACCEPTED* packet back. When the originating DTE receives the *CALL ACCEPTED* packet, the virtual circuit is established. (Actually, when a packet comes into the originating DTE it is called a *CALL CONNECTED* packet, but it is in fact identical to the *CALL ACCEPTED* packet sent by the remote DTE.)

At this point both DTEs may use the full-duplex connection to exchange data packets. When either side has had enough, it sends a *CLEAR REQUEST* packet to the other side, which then sends a *CLEAR CONFIRMATION* packet back as an acknowledgement. The three phases of an X.25 connection are shown in Fig. 5-45.

The originating DTE may choose any idle virtual circuit number to identify the

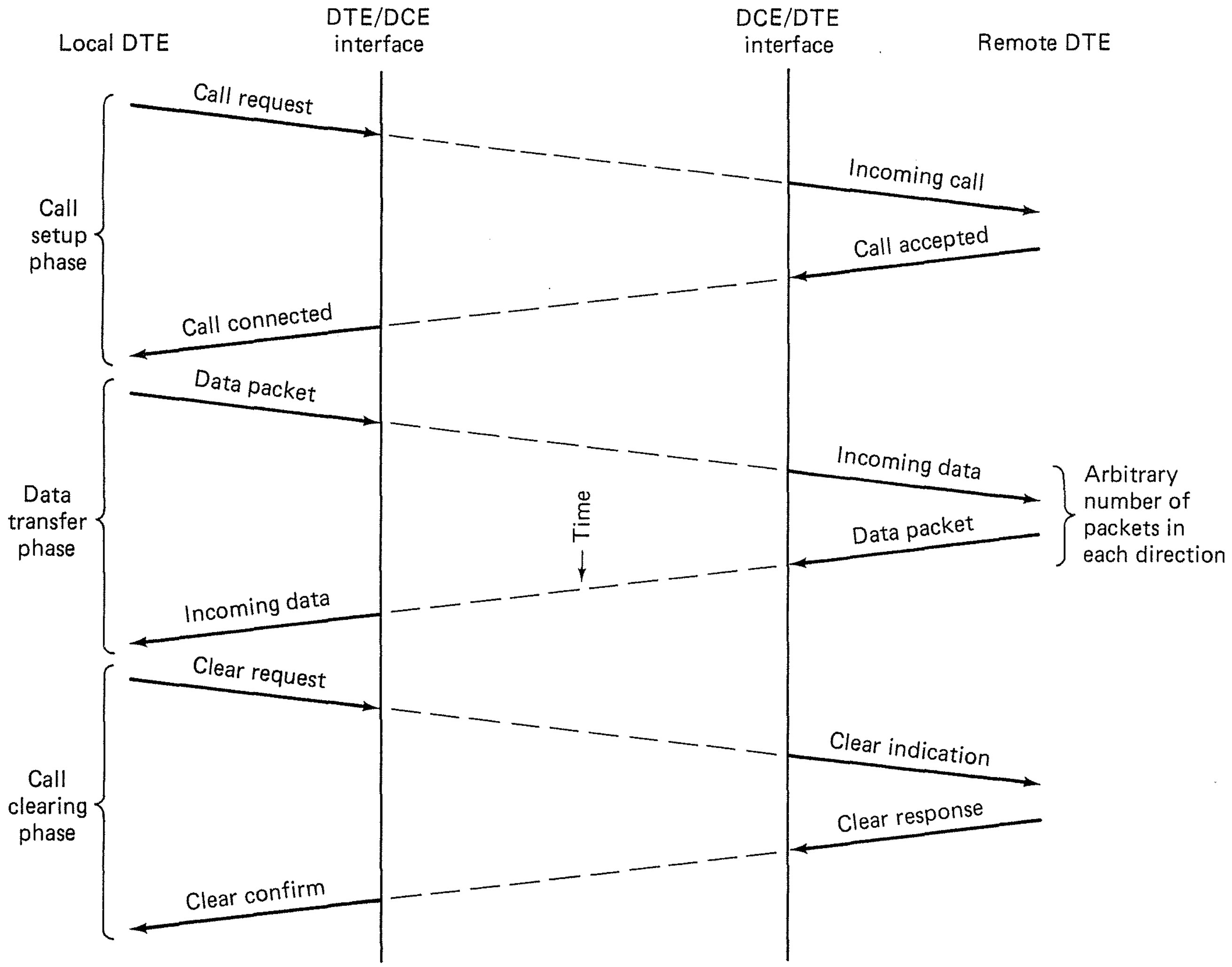
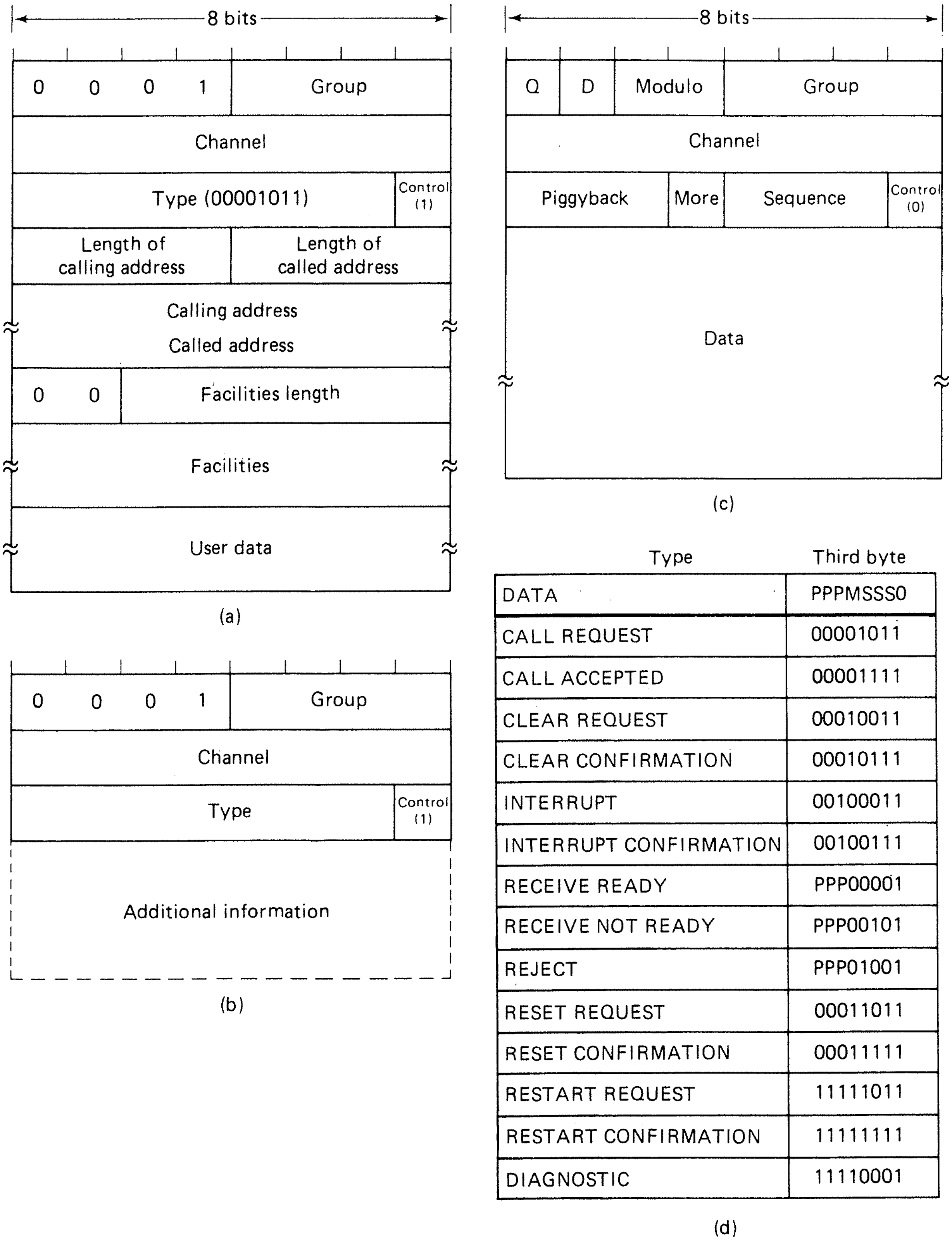


Fig. 5-45. The three phases of an X.25 connection.

connection. If this virtual circuit number is in use at the destination DTE, the destination DCE must replace it by an idle one before delivering the packet. Thus the choice of circuit number on outgoing calls is determined by the DTE, and on incoming calls by the DCE. It might happen that both simultaneously choose the same one, leading to a **call collision**. X.25 specifies that in the event of a call collision, the outgoing call is put through and the incoming one is canceled. Many networks will attempt to put the incoming call through shortly thereafter using a different virtual circuit. To minimize the chance of getting a call collision, the DTE normally chooses the highest available identifier for outgoing calls and the DCE chooses the lowest one for incoming calls.

The format of the *CALL REQUEST* packet is shown in Fig. 5-46(a). This packet, as well as all other X.25 packets, begins with a 3-byte header. (CCITT calls bytes **octets**.)

The *Group* and *Channel* fields together form a 12-bit virtual circuit number.



Type	Third byte
DATA	PPPMSSS0
CALL REQUEST	00001011
CALL ACCEPTED	00001111
CLEAR REQUEST	00010011
CLEAR CONFIRMATION	00010111
INTERRUPT	00100011
INTERRUPT CONFIRMATION	00100111
RECEIVE READY	PPP00001
RECEIVE NOT READY	PPP00101
REJECT	PPP01001
RESET REQUEST	00011011
RESET CONFIRMATION	00011111
RESTART REQUEST	11111011
RESTART CONFIRMATION	11111111
DIAGNOSTIC	11110001

Fig. 5-46. X.25 packet formats. (a) Call request format. (b) Control packet format. (c) Data packet format. (d) Type field (P = Piggyback, S = Sequence, M = More).

Virtual circuit 0 is reserved for future use, so in principle, a DTE may have up to 4095 virtual circuits open simultaneously. The *Group* and *Channel* fields individually have no particular significance.

The *Type* field in the *CALL REQUEST* packet, and in all other control packets, identifies the packet type. The *Control* bit is set to 1 in all control packets and to 0 in all data packets. By first inspecting this bit the DTE can tell whether a newly arrived packet contains data or control information.

We are now finished with the (3-byte) header. The remaining fields of Fig. 5-46(a) are unique to the *CALL REQUEST* packet. The next two fields tell how long the calling and called addresses are, respectively. Both addresses are encoded as decimal digits, 4 bits per digit. Old habits die hard in the telephone industry.

The addressing system used in X.25 is defined in CCITT recommendation **X.121**. This system is similar to the public switched telephone network, with each host identified by a decimal number consisting of a country code, a network code, and an address within the specified network. The full address may contain up to 14 decimal digits, of which the first three indicate the country, and the next one indicates the network number. For countries expected to have many public networks engaged in international traffic, multiple country codes have been allocated e.g., the United States has been allocated country codes 310 through 329, allowing up to 200 networks, Canada has been allocated 302 through 307, allowing up to 60 networks, but the Kingdom of Tonga has been allocated only code 539, allowing for 10 networks. Country codes with initial digits of 0 and 1 are reserved for future use, and initial digits 8 and 9 are used for connecting to the public telex and telephone networks, respectively. The division of the remaining 10 digits is not specified by X.121, permitting each network to allocate the 10 billion addresses itself.

The *Facilities length* field tells how many bytes worth of facilities field follow. The *Facilities* field itself is used to request special features for this connection. The specific features available may vary from network to network. One possible feature is reverse charging (collect calls). This facility is especially important to organizations with thousands of remote terminals that initiate calls to a central computer. If all terminals always request reverse charging, the organization only gets one "network phone bill" instead of thousands of them. High-priority delivery is also a possibility. Yet another feature is a simplex, instead of a full-duplex, virtual circuit.

The caller can also specify a maximum packet length and a window size rather than using the defaults of 128 bytes and two packets, respectively. If the callee does not like the proposed maximum packet length or window size he may make a counterproposal in the facilities field of the *CALL ACCEPTED* packet. The counterproposal may only change the original one to bring it closer to the default values, not further away. Figure 5-47 lists some common facilities offered by many networks.

Some facilities may be selected when the customer becomes a network subscriber rather than on a call-by-call basis. These include closed user groups (no

Use extended (7-bit) sequence numbers
Set nonstandard window size
Set nonstandard packet size
Set throughput class (75 bps to 48 kbps)
Request reverse charging
Accept reverse charging
Select carrier (e.g., TELENET or TYMNET)
Outgoing data only (no incoming data)
Incoming data only (no outgoing data)
Select go-back-n vs. selective repeat
Use fast select

Fig. 5-47. Examples of X.25 facilities.

user can call outside the group, for security reasons), maximum window sizes smaller than seven (for terminals with limited buffer space), line speed (e.g., 2400 bps, 4800 bps, 9600 bps), and prohibition of outgoing calls or incoming calls (terminals place calls but do not accept them).

The *User data* field allows the DTE to send up to 16 bytes of data together with the *CALL REQUEST* packet. The DTEs can decide for themselves what to do with this information. They might decide, for example, to use it for indicating which process in the DTE the caller wants to be connected with. Alternatively, it might contain a login password.

The format of the other control packets is shown in Fig. 5-46(b). Some have only a header; others have an additional byte or two. The fourth byte of the *CLEAR REQUEST* packet, for example, tells why the connection is being cleared. *CLEAR REQUEST* packets are automatically generated by the subnet when a *CALL REQUEST* cannot be put through. When this happens, the cause is recorded here. Typical causes are: callee refuses to accept reverse charging, the number is busy, the destination is down, or the network is congested.

Because X.25 makes a distinction between *CLEAR REQUEST* and *CLEAR CONFIRMATION*, there is the possibility of a **clear collision** (i.e., both sides decide to terminate the connection simultaneously). However, it is always obvious what is happening, so there is no ambiguity, and the connection can be simply cleared.

The data packet format is shown in Fig. 5-46(c). The *Q* bit indicates Qualified data. The standard is silent as to what distinguishes qualified from unqualified data, but the intention is to allow protocols in the transport and higher layers to set this bit to 1 to separate their control packets from their data packets. The *Control* field is always 0 for data packets. The *Sequence* and *Piggyback* fields are used for flow control, using a sliding window. The sequence numbers are modulo 8 if *Modulo* is 01, and modulo 128 if *Modulo* is 10. (00 and 11 are illegal.) If modulo 128 sequence numbers are used, the header is extended with an extra byte to accommodate the longer *Sequence* and *Piggyback* fields. The meaning of the *Piggyback* field is determined by the setting of the D bit. If $D = 0$, a subsequent acknowledgement means only that the local DCE has received the packet, not that the remote DTE has received it. If $D = 1$, the acknowledgement is a true end-to-end acknowledgement, and means that the packet has been successfully delivered to the remote DTE.

Even if delivery is not guaranteed (i.e., $D = 0$), the *Piggyback* field can be useful. Consider, for example, a carrier that offers a high delay service for bargain hunters. Incoming packets are written onto a magnetic tape, which is then mailed to the destination the next day. In this case the *Piggyback* field is used strictly for flow control. It tells the DTE that the DCE is prepared to accept the next packet, and nothing more.

One point worth noting about acknowledgements in X.25 is that instead of returning the number of the last packet correctly received (as in our examples of Chapter 4), the DTEs are required to return the number of the next packet expected (i.e., one higher). This choice is an arbitrary one, but to be X.25-compatible, DTEs must play the game according to CCITT's rules.

The *More* field allows a DTE to indicate that a group of packets belong together. In a long message, each packet except the last one would have the *More* bit on. Only a full packet may have this bit set. The subnet is free to repackage data in different length packets if it needs to, but it will never combine data from different messages (as indicated by the *More* bit) into one packet.

The standard says that all carriers are required to support a maximum packet length of 128 data bytes. However, it also allows carriers to provide optional maximum lengths of 16, 32, 64, 256, 512, 1024, 2048, and 4096 bytes. In addition, maximum packet length can be negotiated when a connection is set up. The point of maximum packet lengths longer than 128 is for efficiency. The point of maximum packet lengths shorter than 128 is to allow terminals with little buffer space to be protected against long incoming packets.

The other kinds of control packets are listed in Fig. 5-46(d). *INTERRUPT* packets allow a short (32-byte) signal to be sent out of sequence. Since control packets do not bear sequence numbers, they can be delivered as soon as they arrive, without regard to how many sequenced data packets are queued up ahead of them. A typical use for this packet is to convey the fact that a terminal user has hit the quit or break key. An *INTERRUPT* packet is acknowledged by an *INTERRUPT CONFIRMATION* packet.

The *RECEIVE READY (RR)* packet is used to send separate acknowledgements when there is no reverse traffic to piggyback onto. The *PPP* field tells which packet is expected next. When sequence numbers are modulo 128, an extra byte is needed for this packet.

The *RECEIVE NOT READY (RNR)* packet allows a DTE to tell the other side to stop sending packets to it for a while. *RECEIVE READY* can then be used to tell the DCE to proceed.

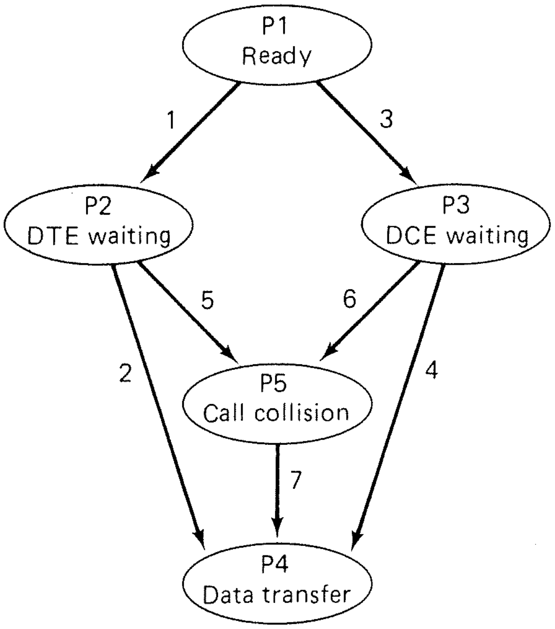
The *REJECT* packet allows a DTE to request retransmission of a series of packets. The *PPP* field gives the first sequence number desired.

The *RESET* and *RESTART* packets are used to recover from varying degrees of trouble. A *RESET REQUEST* applies to a specific connection and has the effect of reinitializing the window parameters to 0. A common use of *RESET REQUEST* is for the DCE to inform the DTE that the subnet has crashed. After receiving a *RESET REQUEST*, the DTE has no way of knowing if packets that were outstanding at the time have been delivered. Recovery must be done by the transport layer. Up to two extra bytes in the *RESET REQUEST* packet allow the requester to try to explain what the cause of the reset is. The DTE can also initiate a *RESET REQUEST*, of course.

A *RESTART REQUEST* is much more serious. It is used when a DTE or DCE has crashed and is forced to abandon all of its connections. A single *RESTART REQUEST* is equivalent to sending a *RESET REQUEST* separately for each virtual circuit.

A *DIAGNOSTIC* packet is also provided, to allow the network to inform the user of problems, including errors in the packets sent by the user (e.g., an illegal *Type* field).

The X.25 standard contains several state diagrams to describe event sequences such as call setup and call clearing. The diagram of Fig. 5-48 shows the subphases of call setup. Initially, the interface is in state *P1*. A *CALL REQUEST* or *INCOMING CALL* (i.e., the arrival of a *CALL REQUEST* packet) changes the state to *P2* or *P3*, respectively. From these states the data transfer state, *P4*, can be reached, either directly, or via *P5*. Similar diagrams are provided for call clearing, resetting, and restarting.



Transition	Caused by	Packet sent
1	DTE	CALL REQUEST
2	DCE	CALL CONNECTED
3	DCE	INCOMING CALL
4	DTE	CALL ACCEPTED
5	DCE	INCOMING CALL
6	DTE	CALL REQUEST
7	DCE	CALL CONNECTED

Fig. 5-48. Call setup in X.25.

The original (1976) X.25 standard was more-or-less as we have described it so far (minus the *D* bit, DIAGNOSTIC packet, the packet length and window size negotiation, and a few other minor items). However, there was considerable demand for a datagram facility, in addition to the virtual circuits. Both the United States and Japan made (conflicting) proposals for the architecture of the datagram service. In the great tradition of international bureaucracies, CCITT accepted *both* of them in 1980, making the protocol even more complicated than it already was.

However, the PTTs, with 100 years of telephone-mentality behind them, did not implement datagrams. Several years later it was noticed that no one was using datagrams, so they were dropped in the 1984 standard.

Nevertheless, the demand for a connectionless service remained. Applications such as point-of-sale terminals, credit card verification, and electronic funds transfer all have the property that the calling party initiates a call with a request, and the called party provides a response. No connection is needed thereafter. To satisfy this demand, CCITT agreed to add a **fast select** feature in 1984.

When fast select is used, the *CALL REQUEST* packet is also expanded to include 128 bytes of user data. Fast select is requested as a facility. As far as the network is concerned, the packet really is an attempt to establish a virtual circuit. When fast select is used, the called DTE may reject the attempted call with a *CLEAR REQUEST* packet, which has also been expanded to include 128 bytes of reply data. However, it may also accept the call, in which case the virtual circuit is set up normally.

Using X.25 for the Network Layer Connection-Oriented Service

When the X.25 protocol is used to provide the OSI connection-oriented network service, the OSI service primitives must be mapped onto the various features of X.25. ISO standard 8878 describes how this mapping is to be done. We will describe it briefly below.

In general, when the transport layer issues a request, a packet is sent. The arrival of this packet at the destination causes an indication primitive to happen. Responses are analogous to requests. Figure 5-49 shows the mapping between primitives and X.25 actions. It is more-or-less straightforward, except that the *N-DATA-ACKNOWLEDGEMENT* primitives do not have any exact correspondence in X.25. They can be handled using the sequence numbers in the RR and RNR packets.

5.5.2. The Network Layer in the ARPANET (IP)

The network layer in the ARPANET has gone through several iterations as problems appeared and were solved. The current network layer protocol, **IP (Internet Protocol)**, was introduced in the early 1980s and has been running ever since. Many other networks have since adopted it, usually in conjunction with the ARPANET transport protocol, TCP, which we will study in the chapter on the transport layer.

Network Service Primitive	X.25 Action
N-CONNECT.request	Send CALL REQUEST
N-CONNECT.indication	CALL REQUEST arrives
N-CONNECT.response	Send CALL ACCEPTED
N-CONNECT.confirm	CALL ACCEPTED arrives
N-DISCONNECT.request	Send CLEAR REQUEST
N-DISCONNECT.indication	CLEAR REQUEST arrives
N-DATA.request	Send Data packet
N-DATA.indication	Data packet arrives
N-DATA-ACKNOWLEDGE.request	No packet
N-DATA-ACKNOWLEDGE.indication	No packet
N-EXPEDITED-DATA.request	Send INTERRUPT
N-EXPEDITED-DATA.indication	INTERRUPT arrives
N-RESET.request	Send RESET REQUEST
N-RESET.indication	RESET REQUEST arrives
N-RESET.response	none
N-RESET.confirm	none

Fig. 5-49. Mapping of network layer connection-oriented services onto X.25

Unlike the X.25 protocol, which is connection-oriented, the ARPANET network layer protocol is connectionless. It is based on the idea of internet datagrams that are transparently, but not necessarily reliably, transported from the source host to the destination host, possibly traversing several networks while doing so, as shown in Fig. 5-39.

The decision to have the network layer provide an unreliable connectionless service evolved gradually from an earlier reliable connection-oriented service as the ARPANET itself evolved into the ARPA Internet, which contains many networks, not all of them reliable. By putting all the reliability mechanisms into the transport layer, it was possible to have reliable end-to-end connections even when some of the underlying networks were not very dependable.

The IP protocol works as follows. The transport layer takes messages and breaks them up into datagrams of up to 64K bytes each. Each datagram is transmitted through the Internet, possibly being fragmented into smaller units as it goes. When all the pieces finally get to the destination machine, they are reassembled by the transport layer to reform the original message.

An IP datagram consists of a header part and a text part. The header has a 20-byte fixed part and a variable length optional part. The header format is given in Fig. 5-50. The *Version* field keeps track of which version of the protocol the datagram belongs to. By including the version in each datagram, the possibility of changing protocols while the network is operating is not excluded.

Since the header length is not constant, a field in the header, *IHL*, is provided to tell how long the header is, in 32-bit words. The minimum value is 5.

The *Type of service* field allows the host to tell the subnet what kind of service

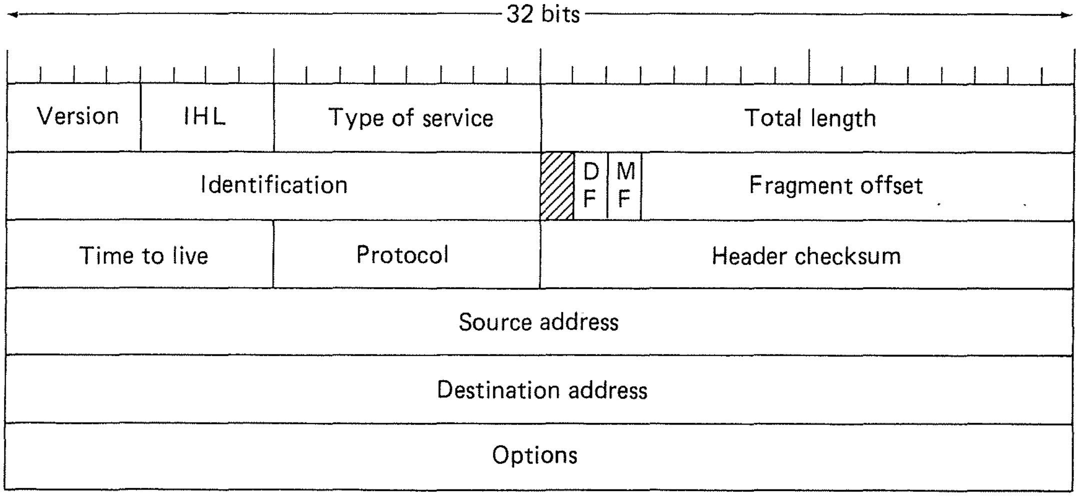


Fig. 5-50. The IP (Internet Protocol) header.

it wants. Various combinations of reliability and speed are possible. For digitized voice, fast delivery is far more important than correcting transmission errors. For file transfer, accurate transmission is far more important than speedy delivery. Various other combinations are also possible from routine traffic to flash override.

The *Total length* includes everything in the datagram—both header and data. The maximum length is 65,536 bytes.

The *Identification* field is needed to allow the destination host to determine which datagram a newly arrived fragment belongs to. All the fragments of a datagram contain the same *identification* value.

Next comes an unused bit and then two 1-bit fields. *DF* stands for Don't Fragment. It is an order to the gateways not to fragment the datagram because the destination is incapable of putting the pieces back together again. For example, a datagram being downloaded to a small micro for execution might be marked with *DF* because the micro's ROM expects the whole program in one datagram. If the datagram cannot be passed through a network, it must either be routed around the network or discarded.

MF stands for More Fragments. All fragments except the last one must have this bit set. It is used as a double check against the *Total length* field, to make sure that no fragments are missing and that the whole datagram is reassembled.

The *Fragment offset* tells where in the current datagram this fragment belongs. All fragments except the last one in a datagram must be a multiple of 8 bytes, the elementary fragment unit. Since 13 bits are provided, there is a maximum of 8192 fragments per datagram, giving a maximum datagram length of 65,536 bytes, in agreement with the *Total length* field.

The *Time to live* field is a counter used to limit packet lifetimes. When it becomes zero, the packet is destroyed. The unit of time is the second, allowing a maximum lifetime of 255 sec.

When the network layer has assembled a complete datagram, it needs to know what to do with it. The *Protocol* field tells which of the various transport processes the datagram belongs to. TCP is certainly one possibility, but there may be others.

The *Header checksum* verifies the header only. Such a checksum is useful because the header may change at a gateway (e.g., fragmentation may occur).

The *Source address* and *Destination address* indicate the network number and host number. Four different formats are used, as illustrated in Fig. 5-51. The four schemes allow for up to 128 networks with 16 million hosts each, 16,384 networks with up to 64K hosts, 2 million networks, presumably LANs, with up to 256 hosts each, and multicast, in which a datagram is directed at a group of hosts. Nearly 1000 networks are currently part of the ARPA Internet, at research, DOD, other governmental, and commercial sites. Addresses beginning with 1111 are reserved for future use.

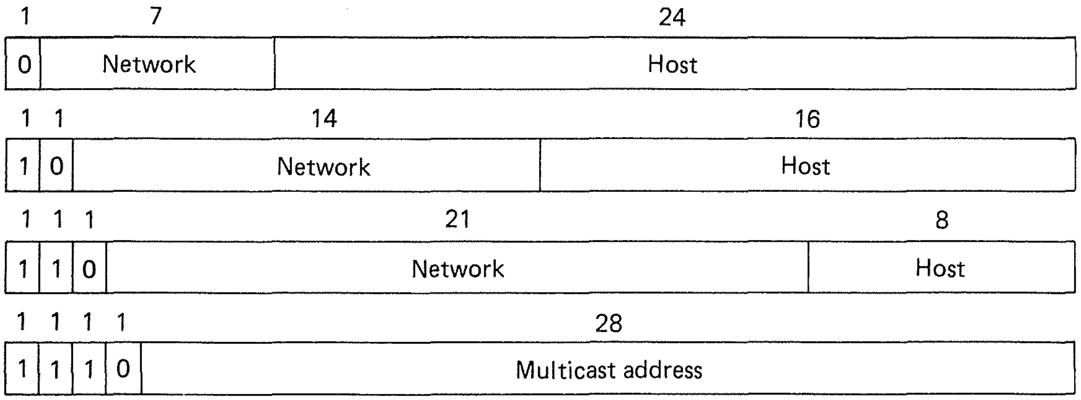


Fig. 5-51. Source and destination address formats in IP.

The *Options* field is used for security, source routing, error reporting, debugging, time stamping, and other information. Basically it provides an escape to allow subsequent versions of the protocol to include information not present in the original design, to allow experimenters to try out new ideas, and to avoid allocating header bits to information that is rarely needed.

The operation of the ARPANET is monitored closely by the IMPs and gateways. When something suspicious occurs, the event is reported by the **ICMP (Internet Control Message Protocol)**, which is also used to test the Internet. About a dozen types of ICMP messages are defined. Each message type is encapsulated in an IP packet.

The *DESTINATION UNREACHABLE* message is used when the subnet or a gateway cannot locate the destination, or a packet with the *DF* bit cannot be delivered because a "small-packet" network stands in the way.

The *TIME EXCEEDED* message is sent when a packet is dropped due to its counter reaching zero. This event is a symptom that packets are looping, that there is enormous congestion, or that the timer values are being set too low.

The *PARAMETER PROBLEM* message indicates that an illegal value has been detected in a header field. This problem indicates a bug in the sending host's IP software, or possibly in the software of a gateway transited.

The *SOURCE QUENCH* message is used to throttle hosts that are sending too many packets. When a host receives this message, it is expected to slow down.

The *REDIRECT* message is used when a gateway notices that a packet seems to be routed wrong. For example, if a gateway in Los Angeles sees a packet that came from New York and is headed for Boston, this message would be used to report the event and help get the routing straightened out.

The *ECHO REQUEST* and *ECHO REPLY* messages are used to see if a given destination is reachable and alive. Upon receiving the *ECHO* message, the destination is expected to send an *ECHO REPLY* message back. The *TIMESTAMP REQUEST* and *TIMESTAMP REPLY* messages are similar, except that the arrival time of the message and the departure time of the reply are recorded in the reply. This facility is used to measure network performance.

In addition to these messages, there are four others that deal with internet addressing, to allow hosts to discover their network numbers and to handle the case of multiple LANs sharing a single IP address.

Originally, the ARPANET used the distributed routing algorithm described in Sec. 5.2.6. After about 10 years, that algorithm was replaced because it caused some packets to loop for a long time and did not use alternate routes. Furthermore, the ARPANET had grown to such a size that the traffic generated by routing table exchanges was getting so large as to interfere with the regular traffic.

In the successor algorithm, each IMP maintains internally a representation of the entire ARPANET, including the delays on each line. Using this data base, every IMP computes the shortest path from itself to every other IMP, using delay as the metric. Since every IMP runs the shortest path algorithm on (almost) the same data base, the paths are consistent and there is little looping.

To provide adaptation to traffic and topology changes, each IMP measures the delay on each of its lines averaged over a 10-sec period. The results of these measurements, along with an update sequence number, are then broadcast to all other IMPs using a flooding algorithm.

5.5.3. The Network Layer in MAP and TOP

The network layer in both MAP and TOP provides the OSI connectionless service to the transport layer. The network protocol used is ISO 8473, which is the ISO connectionless network layer protocol based on the ARPANET's IP protocol. Thus the transport layer in MAP and TOP builds and sends internet datagrams.

The choice of IP by MAP and TOP is both surprising and not surprising. It is surprising because most large organizations tend to accept X.25 at the network layer as a fact of life, whether they like it or not. It is not surprising because both MAP and TOP are very concerned about internetworking, especially between 802 LANs and various WANs. Running all the 802 LANs as X.25 networks and then connecting them with the virtual circuit oriented X.75 is unattractive. Furthermore, both GM and Boeing clearly wanted a system that would actually work, and the fact that the ARPANET had a decade of experience running IP over a very heterogeneous collection of networks clearly made an impression on them.

However, a problem arises when trying to run IP over public networks that offer only connection-oriented X.25 service. The solution is to set up a connection between two X.25 end points, and then just use the public network as a big, dumb wire to transmit the IP packets. In this mode of operation, the subnet enhancement sublayer of Fig. 5-27 effectively de-enhances X.25 to get rid of its connection orientation and just provide a raw connectionless service to the transport layer. When running IP over an 802 LAN, the subnet enhancement layer is null.

The OSI internet packet format described in ISO 8473 contains almost exactly the same fields as the ARPA Internet IP packet (Fig. 5-50), although the order is slightly different. Two differences, however, are the format of the address fields and an extra flag bit. The ARPA Internet uses 32-bit source and destination addresses, which are certainly not adequate for a worldwide network. The corresponding OSI fields are variable length, with count fields telling how long they are, just as in X.25. The extra flag bit can be set to request that if the packet is discarded along the way (e.g., due to congestion), a report is sent back to the source.

The MAP and TOP standards specify the form of the addresses to be used and how they relate to routing. Basically, these addresses have three subparts, the company, the LAN, and the machine number. This numbering plan leads to a hierarchical routing, with packets first being routed to the proper company, then to the proper LAN, and finally to the destination.

5.5.4. The Network Layer in USENET

USENET does not have a network layer in the sense of something that offers a service to the transport layer. However, with something like 10,000 hosts on the network, most of them only connecting to a handful of other hosts, routing is a major issue. The routing algorithm has gone through several iterations, which we will describe below.

The initial routing algorithm was pure source routing. Each mail message contained a line of the form:

To: ucbvax!decvax!mcvax!marvin

In this example, the originating machine would call up, or wait to be called up by *ucbvax* (a VAX computer at Berkeley) to transfer the message to it. The next time *ucbvax* and *decvax* were in contact, the message would be forwarded to *decvax*, which in turn would eventually forward it to *mcvax* where it would be put in Marvin's mailbox.

This algorithm had the advantage of not requiring any routing on the part of the network. Users had to figure out how to route their own messages, with the aid of large maps of the network topology. On the other hand, as the network grew to thousands of hosts, the map got very large, and paths with dozens of hops became increasingly common. Users often made typing mistakes, which resulted in problems when their messages got to the point in the path where the typing error occurred. If the host holding the message did not recognize the name of the host it was supposed to forward the message to, all it could do was discard it or return it to the sender.

To further complicate matters, the network topology changed daily, and the maps were always out of date. (Actually, this was not so much of a problem because they were unreadable anyway.) Finally, sending mail between USENET, BITNET, CSNET, and ARPANET was difficult because each network had its own naming and routing conventions.

To bring some order to this chaos, it was decided to introduce a uniform naming scheme to all four networks. The idea was to group all the hosts into **domains**, which in turn could be divided into subdomains, which also could be subdivided. In effect, all the hosts would be organized into a naming tree.

At the top level in the U.S. the following domains exist:

1. COM- Companies, including nonprofit corporations
2. EDU- Educational institutions such as universities and high schools
3. GOV- Nonmilitary city, state, and federal government agencies
4. MIL - Military organizations
5. ORG- Everything else

In addition, each foreign country is also a top-level domain, for example, NL for The Netherlands, UK for the United Kingdom, and so on. It is sometimes possible to generate multiple names for the same site. The IBM research laboratory in Switzerland, for example, has chosen to be in the COM domain, rather than in the Swiss domain.

Each domain is divided into subdomains. For example, EDU has subdomains BERKELEY.EDU, MIT.EDU, and many others for other universities. These second level domains are typically divided into third level domains for departments or major projects. Thus CS.HARVARD.EDU could be the computer science department at Harvard University. In the domain system, mail is addressed not by giving an explicit route to follow, but rather by giving a name and a domain, as in MARVIN@CS.HARVARD.EDU.

While domain naming makes it much easier for people to type addresses correctly, it completely ruins the USENET routing algorithm. Instead, a new method was devised. This method consists of a program that can run on any host, read the network topology map stored online there, and produce a file giving the shortest path to each known destination. When it is time to forward a piece of mail, the mailing system can look in this file to find out what the first hop along the path is, and forward the message there.

Although this routing algorithm sounds simple, there are a few points worth making. To start with, the idea of precomputing all the shortest paths in advance rather than doing it per message is really essential. With over 10,000 hosts and 30,000 links in the map, computing the shortest path to any given host requires a substantial amount of computation.

A more serious issue is how to assign costs to links. The connections differ in terms of bandwidth (1200 bps, 2400 bps, etc.), telephone cost per minute, and frequency of connection. Suppose mail to host *X* can go via a 9600 bps telephone line that is activated once a day, seven days a week and costs 20 cents/minute or via a 300 bps X.25 connection that is activated twice a day on weekdays only and costs 0.1 cent per byte and 2 cents/minute for connect time. Which route is shorter? To make a long story short, the weights on the arcs of the graph have been assigned empirically to make the results reasonable, but not optimal. The routing procedure is described in more detail by Partridge (1986).

5.6. SUMMARY

The network layer provides services to the transport layer. These services can be either connection-oriented or connectionless. The connection-oriented services have primitives for establishing, using and releasing connections, whereas the connectionless ones just have primitives for sending and receiving data. The OSI model supports both styles.

Functionally, the main task of the network layer is routing packets from source to destination. Some networks do routing at the time a connection is set up, and use that route for all packets on the connection. Others route each packet individually. Each approach has its own advantages and disadvantages.

Many routing algorithms are known. Shortest path routing is conceptually simple and widely used. Multipath routing spreads the traffic over several routes, both to gain improved performance and higher reliability. In centralized routing, a single routing machine computes all the routes and downloads them to the IMPs. In isolated routing, each IMP makes its own decisions based on local traffic conditions. In between is distributed routing, in which each IMP makes local routing decisions, but exchanges information with its neighbors. Hierarchical routing is important in large networks.

If too many packets are present in the subnet, it may become congested and the throughput will drop off. Congestion can be dealt with by preallocating buffers, discarding packets, or limiting the number of packets in the subnet by various means such as the isarithmic scheme or having IMPs send choke packets to slow down the input rate when they get overloaded. Various kinds of deadlocks can occur, but algorithms for avoiding them are known.

Internetworking involves connecting two networks together. The interconnection can occur in the data link layer (bridges) or network layer (gateways). IEEE has standardized two types of 802 bridges, a spanning tree bridge and a source routing bridge. In the former, the bridges are fully transparent and acquire routing information via backward learning. In the latter, the hosts are involved, and they learn about the topology using discovery frames.

In the network layer, two types of interconnections have evolved: concatenated virtual circuits (X.75) and internet datagrams (IP). As with virtual circuits and datagrams within a single subnet, the virtual circuit approach is simple and avoids congestion, but is inflexible and wastes resources. Similarly, the datagram approach is flexible and robust, but can suffer from congestion.

Two network layer protocols are in widespread use around the world. The X.25 protocol, which is connection-oriented, is used in public data networks. The IP protocol, which is connectionless, is used in the ARPA Internet, at universities, and at most UNIX installations.

PROBLEMS

1. Give two example applications for which connection-oriented service is appropriate. Now give two examples for which connectionless service is best.

2. Are there any circumstances when a virtual circuit service will (or should) deliver packets out of order? Explain.

3. Give three examples of protocol parameters that might be negotiated when a connection is set up.

4. Referring to Fig. 5-6, what new table entries would be needed to add a path *AEFD*?

5. Consider the following design problem concerning implementation of virtual circuit service. If virtual circuits are used internal to the subnet, each data packet must have a 3-byte header, and each IMP must tie up 8 bytes of storage for circuit identification. If datagrams are used internally, 15-byte headers are needed but no IMP table space is required. Transmission capacity costs 1 cent per 10^6 bytes, per hop. IMP memory can be purchased for 1 cent per byte and is depreciated over 2 years (business hours only). The statistically average session runs for 1000 sec, in which time 200 packets are transmitted. The mean packet requires four hops. Which implementation is cheaper, and by how much?

6. Assuming that all IMPs and hosts are working properly and that all software in both is free of all errors, is there any chance, however small, that a packet will be delivered to the wrong destination?

7. What is the difference, if any, between static routing using two equally weighted alternatives, and selective flooding using only the two best paths?

8. Give a simple algorithm for finding two paths through a network from a given source to a given destination that can survive the loss of any communication line. The IMPs are considered reliable enough, so it is not necessary to worry about the possibility of IMP crashes.

9. A certain network uses hot potato routing, that is, incoming packets are put on the shortest queue. One of the IMPs has only two outgoing lines, hence two queues. If the queues are equally long, packets are put on a queue at random. Write down the equation expressing the conservation of flow into and out of the state in which the length of queue 1 is i and the length of queue 2 is j, with $i > j + 1$ and $j > 1$.

10. Propose a good routing algorithm for a network in which each IMP knows the path length in hops to each destination for each output line, and also the queue lengths for each line. For simplicity assume that time is discrete and that a packet can move one hop per time interval.

11. An IMP uses a combination of hot potato and static routing. When a packet comes in, it goes onto the first choice queue if and only if that queue is empty and the line is idle, otherwise it uses the second-choice queue. There is no third choice. If the arrival rate at the IMP for a certain destination is λ packets/sec and the service rate is μ packets/sec, what fraction of the packets get routed via the first choice queue? Assume Poisson arrivals and service times.

12. If delays are recorded as 8-bit numbers in a 50 IMP network, and delay vectors are exchanged twice a second, how much bandwidth per (full-duplex) line is chewed up by the distributed routing algorithm? Assume that each IMP has 3 lines to other IMPS.

13. For hierarchical routing with 4800 IMPs, what region and cluster sizes should be chosen to minimize the size of the routing table for a three layer hierarchy?

14. Looking at the subnet of Fig. 5-16, how many packets are generated by a broadcast from B, using
(a) reverse path forwarding?
(b) the sink tree?

15. Irland's method for controlling congestion requires discarding packets if the required queue exceeds a certain length. Use an M/M/1 finite buffer queueing model to derive an expression telling what fraction of the packets will be discarded.

16. As a possible congestion control mechanism in a subnet using virtual circuits internally, an IMP could refrain from acknowledging a received packet until (1) it knows its last transmission along the virtual circuit was received successfully and (2) it has a free buffer. For simplicity, assume that the IMPs use a stop-and-wait protocol and that each virtual circuit has a one buffer dedicated to it for each direction of traffic. If it takes T sec to transmit a packet (data or acknowledgement), and there are n IMPs on the path, what is the rate at which packets are delivered to the destination host? Assume that transmission errors are rare, and that the host-IMP connection is infinitely fast.

17. A datagram subnet allows IMPs to drop packets whenever they need to. The probability of an IMP discarding a packet is p. Consider the case of a source host connected to the source IMP, which is connected to the destination IMP, and then to the destination host. If either of the IMPs discards a packet, the source host eventually times out and tries again. If both host-IMP and IMP-IMP lines are counted as hops, what is the mean number of
(a) hops a packet makes per transmission?
(b) transmissions a packet makes?
(c) hops required per received packet?

18. Does the Merlin-Schweitzer buffering algorithm guarantee that every packet will be delivered within a finite time? If so, prove it, if not, what is needed to fix it?

19. The Blazewicz et al. algorithm uses timestamps as unique numbers so that some packet will have the lowest value and this move unimpeded to its destination. Would checksums do just as well as timestamps?

20. Consider the operation of a bridge between two LANs using LAPB as the data link protocol. Which of the problems cited in Fig. 5-32 would occur in this bridge and which would not?

21. Imagine two LAN bridges, both connecting a pair of 802.4 networks. The first bridge is faced with 1000 512-byte packets per second that must be forwarded. The second is faced with 200 4096-byte packets per second. Which bridge do you think will need the faster CPU? Discuss.

22. Suppose that the two bridges of the previous problem each connected an 802.4 LAN to an 802.5 LAN. Would that change have any influence on the previous answer?

23. Why do you think CCITT provided flow control in X.25 in both LAPB and PLP instead of just in one layer?

24. When an X.25 DTE and DCE both decide to put a call through at the same time, a call collision occurs and the incoming call is canceled. When both sides try to clear simultaneously, the clear collision is resolved without canceling either request. Do you think that simultaneous resets are handled like call collisions or clear collisions? Defend your answer.

25. An IP datagram of 1024 bytes is fragmented into pieces. Each piece is sent as a separate fragment over an X.25 network whose packet size allows 128 bytes of data per packet. How many fragments are needed, and what is the efficiency of the transmission, counting both the X.25 and IP packet overhead, but ignoring that of lower layers?

26. Describe a way to do reassembly of IP fragments at the destination.

27. Write a program to simulate routing using flooding. Each packet should contain a counter that is decremented on each hop. When the counter gets to zero, the packet is discarded. Time is discrete, with each line handling one packet per time interval. Make three versions of the program: all lines are flooded, all lines except the input line are flooded, and only the (statically chosen) best k lines are flooded. Compare flooding with deterministic routing ($k = 1$) in terms of delay and bandwidth used.

28. Simulate routing using a combination of hot potato and static routing. Time is discrete as above. Use the statically best route unless the queue length is k or more, in which case use the next best route, etc. Investigate the effect of k on mean delay.

29. Write a program that simulates a computer network using discrete time. The first packet on each IMP queue makes one hop per time interval. Each IMP has only a finite number of buffers. If a packet arrives and there is no room for it, it is discarded and not retransmitted. Instead, there is an end-to-end protocol, complete with timeouts and acknowledgement packets, that eventually regenerates the packet from the source IMP. Plot the throughput of the network as a function of the end-to-end timeout interval, parametrized by error rate.

SECOND EDITION

COMPUTER NETWORKS

By Andrew S. Tanenbaum

A comprehensive introduction to computer networks, this book emphasizes network protocols and algorithms, from the physical layer to the application layer and from local area networks to satellite networks. Assuming only a general familiarity with computer systems and programming, **Computer Networks** presents the full spectrum of basic protocols, concepts, algorithms, software, and technologies.

Revised to reflect the latest advances in the field, this second edition features material on LANs (incuding IEEE 802), ISDN (Integrated Services Digital Networks), and fiber optics networks. It also has detailed coverage of the upper layers of the OSI model (transport, session, presentation, and application) as well as extended discussions of MAP, TOP, and USENET.

Outstanding content highlights include:

• The Physical Layer • The Data Link Layer II—Medium Access • The Data Link Layer II—Protocols • The Network Layer • The Transport Layer • The Session Layer • The Presentation Layer • The Application Layer • Suggested Reading and Bibliography •

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-162959-X

6

THE TRANSPORT LAYER

The transport layer is not just another layer. It is really the heart of the whole protocol hierarchy. Its task is to provide reliable, cost-effective data transport from the source machine to the destination machine, independent of the physical network or networks currently in use. Without the transport layer, the whole concept of layered protocols would make little sense. In this chapter we will study the transport layer in detail, including its design, services, and protocols.

Many networking applications need only a method to reliably transmit a stream of bits from one machine to another. For example, pipes between machines in a distributed UNIX system just need bit transport. They do not need or want any session or presentation services to get in the way. In fact, there are a surprisingly large number of applications that do not need any services from the session layer or above, but it is considered in poor taste to point this out publicly. The ARPANET does not even have session or presentation layers, and few complaints have been voiced about their absence.

This chapter will rely on the OSI model and its terminology more than any of the previous ones. This change in emphasis is due to the fact that many networks were already in operation before the OSI model was designed. These networks generally were well thought out up through the network layer, but began to get excessively fuzzy around the transport layer. As a result, a substantial body of terminology, literature, and operational experience for layers 1 through 3 was well established before the OSI model came along, and is likely to continue developing for years to come. Most of the previous chapters have drawn heavily on this experience (e.g., the term "packet" had been long established before the OSI term "NPDU" came along). This chapter will also emphasize connection-oriented transport, since that is by far the most common type.

Starting with the transport layer, however, the pre-OSI influence has been much weaker. Only one pre-OSI transport protocol (the ARPANET's TCP) is well-established, and even that one may eventually be replaced by its OSI equivalent. Relatively little pre-OSI terminology relating to the transport layer (and almost none relating to the session and presentation layers) is in common use. Considering all these factors, this chapter will have much more of an OSI flavor than its predecessors, a harbinger of things to come. However, we will still treat the general principles first and the details of the various example protocols in a separate section at the end of the chapter, as before.

6.1. TRANSPORT LAYER DESIGN ISSUES

In this section we will provide an introduction to some of the issues that the designers of the transport layer must grapple with. They include the kind of service provided to the session layer, the quality of this service, and the transport layer primitives provided to invoke the service. Finally, we will conclude the section with an initial discussion of the protocols needed to realize the transport service.

6.1.1. Services Provided to the Session Layer

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective service to its users, normally entities (e.g., processes) in the session layer. To achieve this goal, the transport layer makes use of the services provided by the network layer. The hardware and/or software within the transport layer that does the work is called the **transport entity**. The relationship of the network, transport, and session layers is illustrated in Fig. 6-1.

Just as there are two types of network service, there are also two types of transport service: connection-oriented and connectionless. The connection-oriented transport service is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release. Addressing and flow control are also similar in both layers. Furthermore, the connectionless transport service is also very similar to the connectionless network service.

The obvious question is then: "If the transport layer service is so similar to the network layer service, why are there two distinct layers? Why is one layer not adequate?" The answer is subtle, but crucial, and goes back to Fig. 1-7. In this figure we can see that the network layer is part of the communication subnet and is run by the carrier (at least for WANs). What happens if the network layer offers

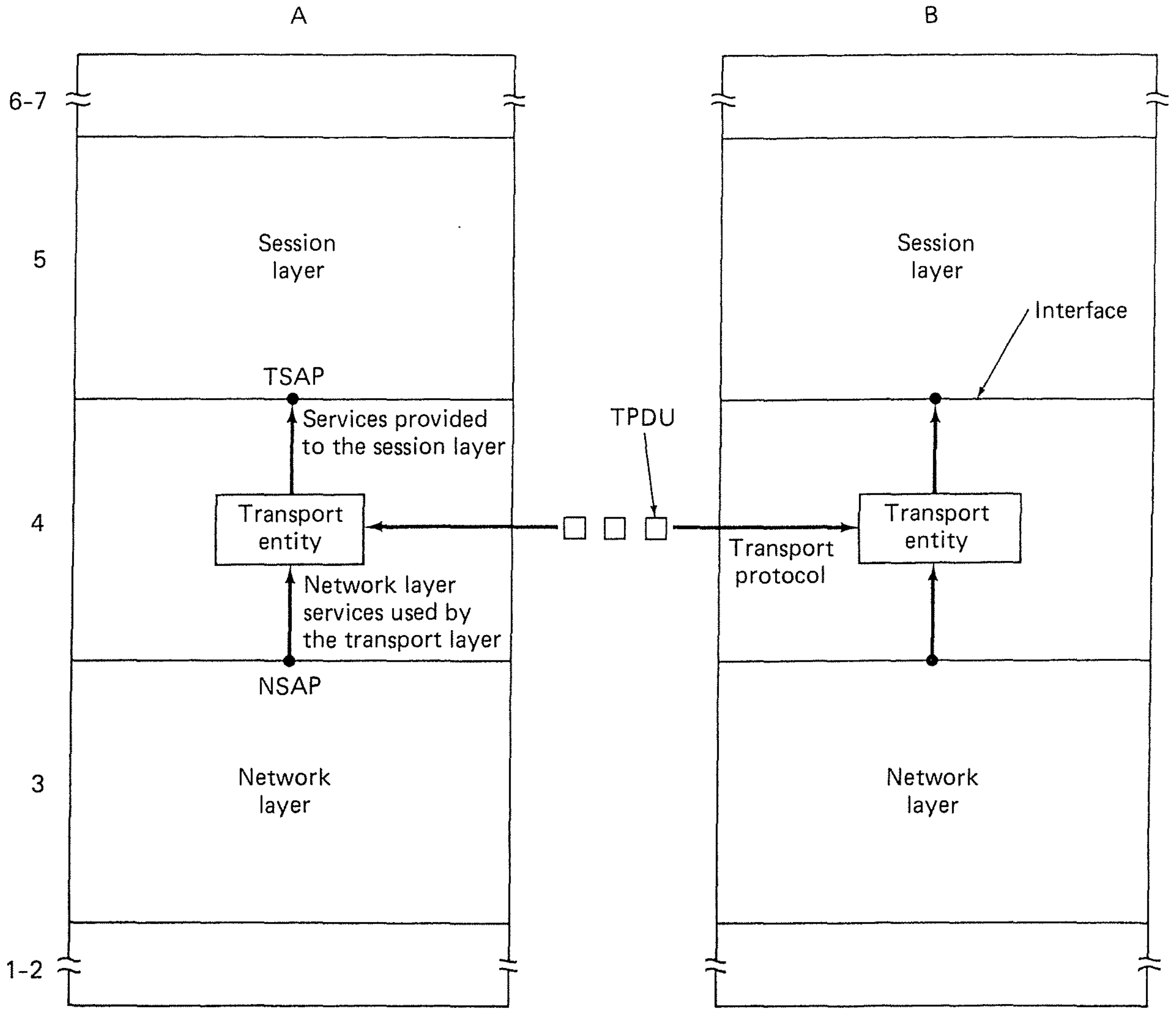


Fig. 6-1. The network, transport, and session layers.

connection-oriented service, but is unreliable? Suppose it frequently loses packets? What happens if it crashes or issues *N-RESET*s all the time?

Since the users have no control over the subnet, they cannot solve the problem of poor service by using better IMPs or putting more error handling in the data link layer. The only possibility is to put another layer on top of the network layer that improves the quality of the service. If a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity. Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and then pick up from where it left off.

In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network service. Lost packets, mangled data, and even network *N-RESET*s can be detected and compensated for by the transport layer. Furthermore, the transport service primitives can be designed to be independent of the network service primitives, which may vary considerably from network to network (e.g., connectionless LAN service may be quite different than connection-oriented WAN service).

Thanks to the transport layer, it is possible for application programs to be written using a standard set of primitives, and to have these programs work on a wide variety of networks, without having to worry about dealing with different subnet interfaces and unreliable transmission. If all real networks were flawless and all had the same service primitives, the transport layer would probably not be needed. However, in the real world it fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the subnet.

For this reason, many people have made a distinction between layers 1 through 4 on the one hand, and layers 5 through 7 on the other. The bottom four layers can be seen as the **transport service provider**, whereas the upper three layers are the **transport service user**. This distinction of provider vs. user has a considerable impact on the design of the layers, and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service.

6.1.2. Quality of Service

Another way of looking at the transport layer is to regard its primary function as enhancing the **QOS** (**Quality Of Service**) provided by the network layer. If the network service is impeccable, the transport layer has an easy job. If, however, the network service is poor, the transport layer has to bridge the gap between what the transport users want and what the network layer provides.

While at first glance, quality of service might seem like a vague concept (getting everyone to agree what constitutes "good" service is a nontrivial exercise), QOS can be characterized by a number of specific parameters. The OSI transport service allows the user to specify preferred, acceptable, and unacceptable values for these parameters at the time a connection is set up. Some of the parameters also apply to connectionless transport. It is up to the transport layer to examine these parameters, and depending on the kind of network service or services available to it, determine whether it can provide the required service. In the remainder of this section we will discuss the QOS parameters. They are summarized in Fig. 6-2.

The **connection establishment delay** is the amount of time elapsing between a transport connection being requested and the confirm being received by the user of the transport service. It includes the processing delay in the remote transport entity. As with all parameters measuring a delay, the shorter the delay, the better the service.

The **connection establishment failure probability** is the chance of a connection not being established within the maximum establishment delay time, for example, due to network congestion, lack of table space, or other internal problems.

Connection establishment delay
Connection establishment failure probability
Throughput
Transit delay
Residual error rate
Transfer failure probability
Connection release delay
Connection release failure probability
Protection
Priority
Resilience

Fig. 6-2. Transport layer quality of service parameters.

The **throughput** parameter measures the number of bytes of user data transferred per second, measured over some recent time interval. The throughput is measured separately for each direction. Actually, there are two kinds of throughput: the actual measured throughput and the throughput that the network is capable of providing. The actual throughput may be lower than the network's capacity because the user has not been sending data as fast as the network is willing to accept it.

The **transit delay** measures the time between a message being sent by the transport user on the source machine and its being received by the transport user on the destination machine. As with throughput, each direction is handled separately.

The **residual error rate** measures the number of lost or garbled messages as a fraction of the total sent in the sampling period. In theory, the residual error rate should be zero, since it is the job of the transport layer to hide all network layer errors. In practice it may have some (small) finite value.

The **transfer failure probability** measures how well the transport service is living up to its promises. When a transport connection is established, a given level of throughput, transit delay, and residual error rate are agreed upon. The transfer failure probability gives the fraction of times that these agreed upon goals were not met during some observation period.

The **connection release delay** is the amount of time elapsing between a transport user initiating a release of a connection, and the actual release happening at the other end.

The **connection release failure probability** is the fraction of connection release attempts that did not complete within the agreed upon connection release delay interval.

The **protection** parameter provides a way for the transport user to specify interest in having the transport layer provide protection against unauthorized third parties (wiretappers) reading or modifying the transmitted data.

The **priority** parameter provides a way for a transport user to indicate that some of its connections are more important than other ones, and in the event of congestion, to make sure that the high-priority connections get serviced before the low-priority ones.

Finally, the **resilience** parameter gives the probability of the transport layer itself spontaneously terminating a connection due to internal problems or congestion.

The QOS parameters are specified by the transport user when a connection is requested. Both the desired and minimum acceptable values can be given. In some cases, upon seeing the QOS parameters, the transport layer may immediately realize that some of them are unachievable, in which case it tells the caller that the connection attempt failed, without even bothering to contact the destination. The failure report specifies the reason for the failure.

In other cases, the transport layer knows it cannot achieve the desired goal (e.g., 1200 bytes/sec throughput), but it can achieve a lower, but still acceptable rate (e.g., 600 bytes/sec). It then sends the lower rate and the minimum acceptable rate to the remote machine, asking to establish a connection. If the remote machine cannot handle the proposed value, but it can handle a value above the minimum, it may lower the parameter to its value. If it cannot handle any value above the minimum, it rejects the connection attempt. Finally, the originating transport user is informed of whether the connection was established or rejected, and if it was established, the values of the parameters agreed upon.

This process is called **option negotiation**. Once the options have been negotiated, they remain that way throughout the life of the connection. The OSI Transport Service Definition (ISO 8072) does not give the encoding or allowed values for the QOS parameters. These are normally agreed upon between the carrier and the customer at the time the customer subscribes to the network service. To keep customers from being too greedy, most carriers have the tendency to charge more money for better quality service.

6.1.3. The OSI Transport Service Primitives

The OSI transport service primitives provide for both connection-oriented and connectionless service. The transport primitives are listed in Fig. 6-3. A comparison of Fig. 6-3 with Fig. 5-4 will show that the transport and network services are (intentionally) similar.

Despite the similarities with the network service, there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks (e.g., X.25 networks), warts and all. These networks can lose packets and can spontaneously issue *N-RESET*s due to internal network problems. Thus the network service provides a way for its users to deal with acknowledgements and *N-RESET*s.

The transport service, in contrast, does not mention either acknowledgements or

```
T-CONNECT.request(callee, caller, exp_wanted, qos, user_data)
T-CONNECT.indication(callee, caller, exp_wanted, qos, user_data)
T-CONNECT.response(qos, responder, exp_wanted, user_data)
T-CONNECT.confirm(qos, responder, exp_wanted, user_data)
```

```
T-DISCONNECT.request(user_data)
T-DISCONNECT.indication(reason, user_data)
```

```
T-DATA.request(user_data)
T-DATA.indication(user–data)
T-EXPEDITED-DATA.request(user_data)
T-EXPEDITED-DATA.indication(user_data)
```

(a)

```
T-UNITDATA.request(callee, caller, qos, user_data)
T-UNITDATA.indication(callee, caller, qos, user_data)
```

(b)

Notes on terminology:
Callee: Transport address (TSAP) to be called
Caller: Transport address (NSAP) used by calling transport entity
Exp_wanted: Boolean flag specifying whether expedited data will be sent
Qos: Quality of service desired
User_data: 0 or more bytes of data transmitted but not examined
Reason: Why did it happen
Responder: Transport address connected to at the destination

Fig. 6-3. (a) OSI connection-oriented transport service primitives. (b) OSI connectionless transport service primitives.

*N-RESET*s. From the transport user's point of view, the service is error-free. Of course real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network. The acknowledgements and *N-RESET*s that come from the network service are intercepted by the transport entities and the errors are recovered from by the transport protocol. If a network connection is reset, the transport layer can just establish a new one, and continue from where it left off with the old one.

Another important difference between the network service and transport service is who the services are intended for. The network service is used by the transport entities, which normally are part of the operating system or located on a special hardware board or chip. Few users write their own transport entities, and thus few users or programs ever see the bare network service.

In contrast, many users have no use for the session and presentation layers, and do see the transport primitives. As we mentioned earlier, the ARPANET does not even have session or presentation layers, so all programs that use the network interact with the transport primitives (which are different from the OSI transport primitives, but are roughly comparable). To illustrate this point, consider processes connected by pipes in UNIX. They assume the connection between them is perfect.

They do not want to know about acknowledgements or *N-RESET*s or network congestion or anything like that. What they want is a perfect connection. Process *A* puts data into one end of the pipe, and process *B* takes it out of the other. This is what the connection-oriented transport service is all about—hiding the imperfections of the network service so that user processes can just assume the existence of an error-free bit stream.

Just for the record, the situation is not quite so black-and-white as we have just sketched it. The quality of service parameter provides a large number of gray values between perfect service and perfectly awful service. Still, the basic point we have made remains: the transport service is designed to relieve programs that use it from having to worry about dealing with network errors.

Figure 6-4 shows the relationship among the OSI primitives. In each of the eight parts of the figure, one transport user is shown to the left of the double lines, the other transport user is shown to the right of the double lines, and the transport service provider (i.e., the transport layer itself) is shown between the double lines. Furthermore, time runs downward, so that events at the top occur before events at the bottom.

The normal connection setup is illustrated in Fig. 6-4(a). Four primitives are used. One of the transport entities executes a *T-CONNECT.request* primitive to signal its desire to establish a connection with the transport user attached to the **transport service access point** (**TSAP**) address named in the *CONNECT.request* primitive. This primitive results in a *T-CONNECT.indication* occurring at the destination. The transport user attached to the TSAP addressed gets the indication and can either accept it with a *T-CONNECT.response*, as shown in Fig. 6-4(a), or reject it with a *T-DISCONNECT.request*, as shown in Fig. 6-4(b). The result of an acceptance comes back to the initiator as a *T-CONNECT.confirm*. The result of a rejection comes back as a *T-DISCONNECT.indication*.

One other scenario is also possible for an attempt to establish a connection. This scenario is illustrated in Fig. 6-4(c), and occurs when the transport service provider itself rejects the connection. Such a rejection may be the transport user's fault (e.g., a bad parameter in the *T-CONNECT.request* primitive) or the transport provider's fault (e.g., the transport provider has run out of internal table space). In this scenario, nothing is transmitted across the network, so the remote site does not even hear about the failed attempt.

In Fig. 6-4(d)-(f) we see three ways a connection can be released. The normal way is that one of the parties issues a *T-DISCONNECT.request*, which is signaled to the other party as a *T-DISCONNECT.indication*. Either the calling or called party may initiate the release of the connection. If both parties simultaneously issue a *T-DISCONNECT.request* primitive, the connection is released without either side getting an indication.

Finally, the transport provider itself can terminate a connection by issuing *T-DISCONNECT.indication* primitives on both ends, as shown in Fig. 6-4(f). In a way, the last scenario is a little like the network layer issuing an *N-RESET.indication*.

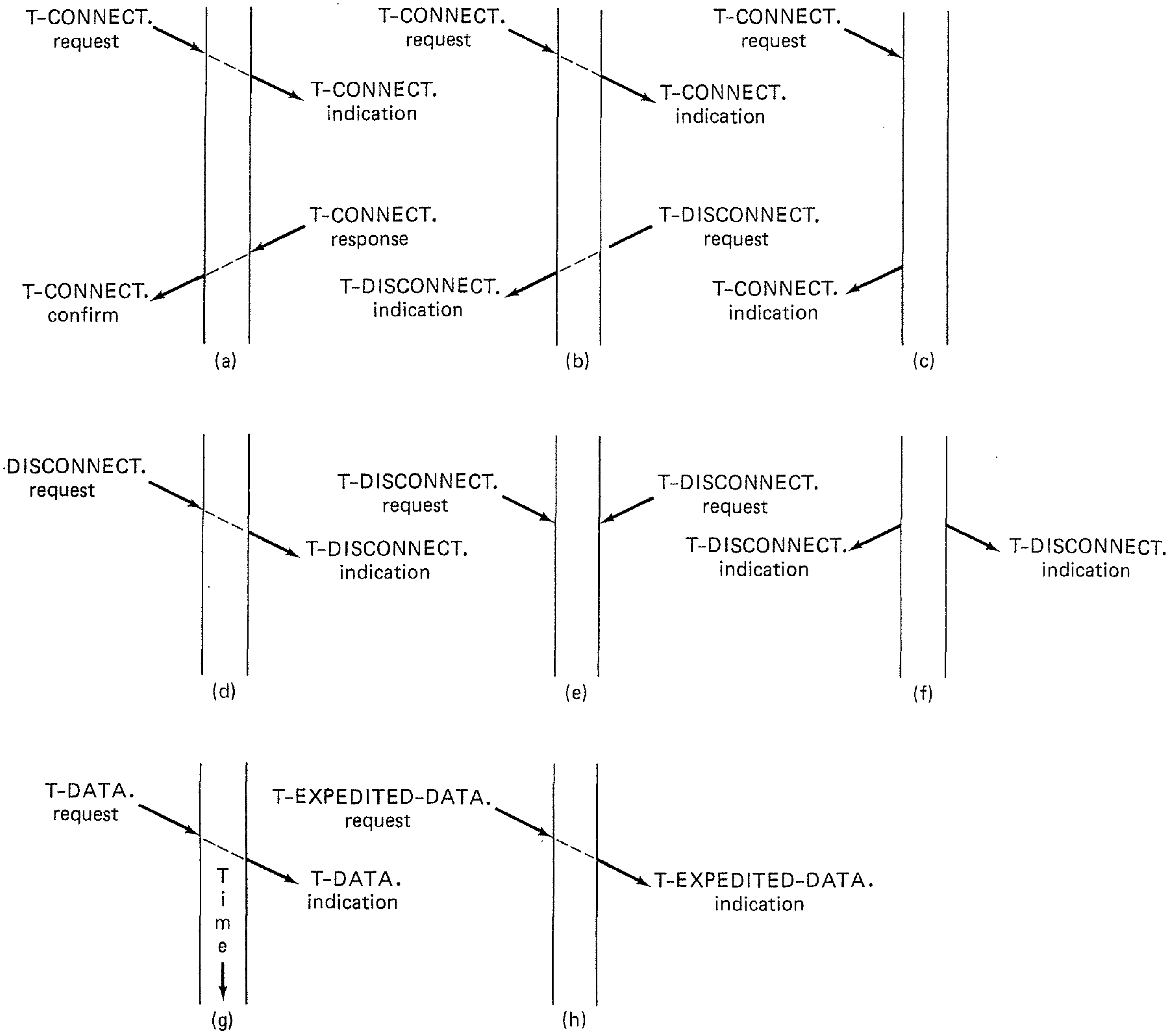


Fig. 6-4. Some valid sequences of OSI transport primitives. (a) Connection setup. (b) Connection rejected by called user. (c) Connection rejected by transport layer. (d) Normal connection release. (e) Simultaneous release by both sides. (f) Transport layer initiated release. (g) Normal data transfer. (h) Expedited data transfer.

The connection is terminated. Clearly, a well-designed transport service provider should not issue spontaneous *T-DISCONNECT.indication* primitives too lightly, but there are circumstances in which no other course of action is possible. For example, if the underlying network crashes and refuses to react to repeated attempts to communicate with it, there is little else the transport service provider can do than break all the connections. Unless the session layer has taken special precautions against this problem, the failure will have to be reported back to the highest level, and may require human intervention to retry the failed command.

]

The final two diagrams in Fig. 6-4 show normal and expedited data transport. In neither case is an explicit acknowledgement or other indication provided back to the sender. The transport layer uses the same queue model (Fig. 5-3) as the network layer, with data normally delivered in sequence.

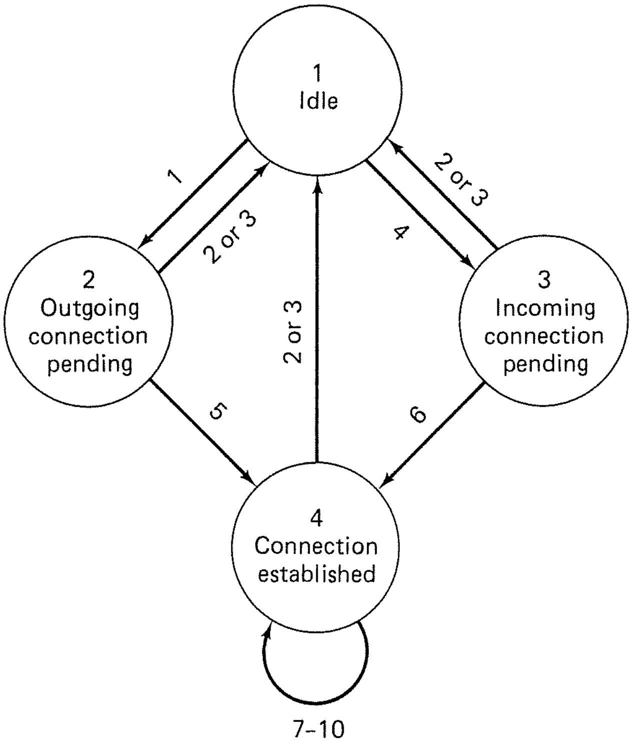
However, the *T-EXPEDITED-DATA* primitive can be used to send data that skips ahead of other data already in the queue. This primitive is normally only used to transmit the BREAK, DEL, or interrupt key that people can type from their terminals to interrupt the current program. If there were no expedited data, consider what would happen if a user started a program from a remote terminal connected to the host by a transport connection, and then typed a line ahead while waiting for the program just started to terminate. If that program got into an infinite loop and the user typed a BREAK character, the BREAK would be carefully added to the tail of the queue, and would not be delivered to the host until the running program terminated and read the line queued ahead of the BREAK. The use of *T-EXPEDITED-DATA.request* causes the BREAK to be delivered immediately, no matter what is in the queue.

Strict rules govern the order in which the transport primitives may be used. For example, it is not permitted to issue a *T-DISCONNECT.request* when no connection is established (or at least in the process of being established). The diagram of Fig. 6-5 shows the four legal states that a connection endpoint may have at a TSAP and the relations among the states. The two endpoints are not necessarily in the same state. During the establishment phase, for example, the initiating party goes from the IDLE state to the OUTGOING CONNECTION PENDING state before anything happens at the other endpoint.

The four states are:

1. IDLE—No connection is established or trying to be established. Both incoming and outgoing connections are possible.
2. OUTGOING CONNECTION PENDING—A *T-CONNECT.request* has been done. The reply from the remote peer has not yet been received.
3. INCOMING CONNECTION PENDING—A *T-CONNECT.indication* has come in. It has not yet been accepted or rejected.
4. CONNECTION ESTABLISHED—A valid connection has been established. The establishment phase is completed and data transfer can begin.

Two ways exist to get from the IDLE state to the CONNECTION ESTABLISHED state. The left-hand route (1-2-4) is used when an outgoing call is being placed. The intermediate state (2) is entered after the *T-CONNECT.request* has been issued, and holds until the *T-CONNECT.confirm* comes in. The remote side rejects the



Transitions:
1. T-CONNECT.request received from transport user
2. T-DISCONNECT.indication received from transport service provider
3. T-DISCONNECT.request received from transport user
4. T-CONNECT.indication received from transport service provider
5. T-CONNECT.confirm received from transport service provider
6. T-CONNECT.response received from transport user
7. T-DATA.request received from transport user
8. T-DATA.indication received from transport service provider
9. T-EXPEDITED-DATA.request received from transport user
10. T-EXPEDITED-DATA.indication received from transport service provider

Fig. 6-5. Transport connection endpoint states and transport primitives.

connection or the caller changes its mind and issues a *T-DISCONNECT.request*, the IDLE state is entered again.

The right-hand route (1-3-4) is used when an incoming call is received. The intermediate state is entered after the *T-CONNECT.indication* comes in. If the connection is intentionally rejected or a *T-DISCONNECT.indication* comes in (either from a fickle caller or from the transport layer itself), we return to the IDLE state. However, if the connection is accepted, state 4 is reached.

From state 4 there are two ways back to the IDLE state. These correspond to the local transport user releasing the connection and the transport layer releasing the connection. The latter event may happen either upon request of the remote transport user or of necessity by the transport service provider.

The set of states and transitions governing connection endpoints shown in Fig. 6-5 is a finite state machine. This one is called the **transport protocol**

machine. Similar protocol machines exist for the other layers. This one is particularly simple. For some of the upper layer protocols, the protocol machine has dozens of states and dozens of transitions, typically one transition for each primitive that can be invoked and each PDU that can arrive from the outside.

6.1.4. Transport Protocols

The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols we studied in detail in Chapter 4. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 6-6. At the data link layer, two IMPs communicate directly via a physical channel, whereas at the transport layer, this physical channel is replaced by the entire subnet. This difference has many important implications for the protocols.

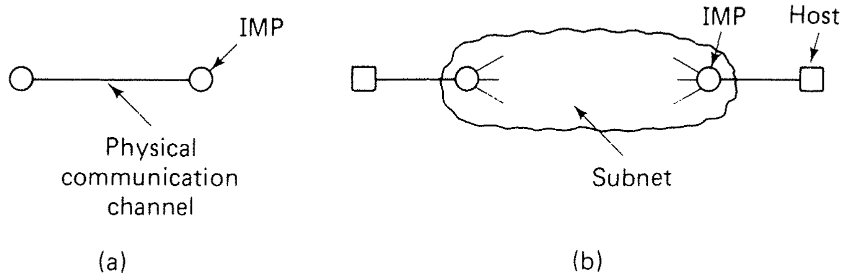


Fig. 6-6. (a) Environment of the data link layer. (b) Environment of the transport layer.

For one thing, in the data link layer, it is not necessary for an IMP to specify which IMP it wants to talk to—each outgoing line uniquely specifies a particular IMP. In the transport layer, explicit addressing of destinations is required.

For another thing, the process of establishing a connection over the wire of Fig. 6-6(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do. In the transport layer, initial connection establishment is more complicated, as we will see.

Another exceedingly annoying, difference between the data link layer and the transport layer is the potential existence of storage capacity in the subnet. When an IMP sends a frame, it may arrive or be lost, but it cannot bounce around for a while, go into hiding in a far corner of the world, and then suddenly emerge at an inopportune moment 30 sec later. If the subnet uses datagrams and adaptive routing inside, there is a nonnegligible probability that a packet may be stored for a number of seconds and then delivered later. The consequences of this ability of the subnet to store packets can be disastrous, and require the use of special protocols.

A final difference between the data link and transport layers is one of amount rather than of kind. Buffering and flow control are needed in both layers, but the presence of a large and dynamically varying number of connections in the transport layer may require a different approach than we used in the data link layer. In Chapter 4, some of the protocols allocate a fixed number of buffers to each line, so that when a frame arrives there is always a buffer available. In the transport layer, the larger number of connections that must be managed make the idea of dedicating many buffers to each one less attractive.

From the transport protocol designer's point of view, the actual properties of the subnet (Fig. 6-6) are less important than the *service* offered by the network layer, although the latter is, of course, strongly influenced by the former. To some extent, however, the network layer service may mask the least desirable aspects of the subnet and provide a better interface.

For the purposes of studying transport protocols, we will group the various kinds of network services into three categories as shown in Fig. 6-7. The first category, type A, consists of service that is essentially perfect. The fraction of packets lost, duplicated, or garbled is negligible. *N-RESET*s are so rare that they can be ignored. In effect, the environment in which the transport layer operates is that of Fig. 6-6(a). Public WANs offering type A service are scarcer than hen's teeth, but some LANs come fairly close. The transport protocols needed to operate over a type A network are straightforward.

Network type	Description
A	Flawless, error-free service with no N-RESETS
B	Perfect packet delivery, but with N-RESETS
C	Unreliable service with lost and duplicated packets and possibly N-RESETS

Fig. 6-7. Types of service that can be offered by the network layer.

A more common situation for WANs is type B service. Individual packets are rarely, if ever, lost (because the network and data link layer protocols recover from these losses transparently) but from time to time the network layer issues *N-RESET*s, either due to internal congestion, hardware problems, or software bugs. It is then up to the transport protocol to pick up the pieces, establish a new network connection, resynchronize, and continue, so that the *N-RESET* is completely hidden from the transport user. Most public X.25 networks are type B. Transport protocols for type B networks are more complex than those for type A.

The third type is the network service that is not reliable enough to be trusted at all. WANs offering pure connectionless (datagram) service, packet radio networks, and many internetworks fall into this category. Transport protocols that must live with type C service are the most complex of all, and must solve all the problems that we saw in the data link layer, and many more as well.

Thus different transport protocols will be needed for different situations. The worse the network service, the more complex the transport protocol. OSI has recognized this problem, and devised a transport protocol with five variants, as listed in Fig. 6-8.

Protocol class	Network type	Name
0	A	Simple class
1	B	Basic error recovery class
2	A	Multiplexing class
3	B	Error recovery and multiplexing class
4	C	Error detection and recovery class

Fig. 6-8. Transport protocol classes.

Class 0 is the simplest class. It sets up a network connection for each transport connection requested and assumes the network connection does not make errors. The transport protocol does no sequencing or flow control, relying on the underlying network layer to get everything right. It does however, provide mechanisms for establishing and releasing transport connections.

Class 1 is like class 0 except that it has been designed to recover from *N-RESET*s. If the network connection being used for a given transport connection is ever subject to an *N-RESET*, the two transport entities resynchronize and continue from where they left off. To achieve resynchronization, they must use and keep track of sequence numbers, something not required with class 0. Other than the ability to recover from *N-RESET*s, class 1 does not provide any error control or flow control on top of what the network layer itself provides.

Class 2, like class 0 is designed to be used with reliable networks (type A). It differs from class 0 in that two or more transport connections may be sent (multiplexed) over the same network connection. This feature is useful when there are many transport connections open, each with relatively little traffic, and the carrier has a high charge for connect time for each open network connection. For example, in an office full of airline reservation terminals, each terminal might have a separate transport connection to a remote computer, with all the transport connections going over one (or a few) network connections, to reduce networking costs. We will look at multiplexing in more detail later in this chapter.

Class 3 combines the features of classes 1 and 2. It allows multiplexing and can also recover from *N-RESET*s. It also uses explicit flow control.

Class 4 is designed for type C network service. It is completely paranoic and takes Murphy's Law (If something can go wrong, it will) as a given. It must therefore be able to handle lost, duplicate, and garbled packets, *N-RESET*s, and everything else the network can throw at it. Needless to say, class 4 protocols are much more complex than the other ones. We will study some of the problems that class 4 protocols must deal with later in this chapter.

It should be pointed out that having a simple-minded connectionless network service and putting all the complexity in the transport protocol is not necessarily a bad idea. Doing a lot of work to make the service almost reliable in the data link and network layers, and then discovering that it is not quite good enough to satisfy the transport layer, so that class 4 must be used there anyway, may be inefficient.

The choice of which protocol class will be used on any given connection is determined by the transport entities at the time the connection is established. The initiator can propose a preferred class, and zero or more alternative classes. The responder then chooses the protocol class to use from the list supplied. If none of the choices offered are acceptable, the connection is rejected.

This negotiation is required because different users may have different concepts of what "reliable" means. To a casual user behind a terminal who wants to access a remote computer to see if any electronic mail has arrived, a network that loses an average of one packet per week may well be considered a type A (perfect) network, and the class 0 protocol may be sufficient. A bank doing multimillion dollar funds transfers all day long may view the same network as definitely type C, and may insist upon the heavy-duty class 4 protocol. Empirical studies of the transport protocols have been made by Meister (1987) and Cole and Lloyd (1986).

6.1.5. Elements of Transport Protocols

The exact features provided by a transport protocol depend on the environment in which it operates (e.g., the type of network service available) and the type of service it must provide. Nevertheless, it is possible to give a list of basic elements which are common to many transport protocols. Figure 6-9 gives such a list, and furthermore shows which features are applicable to each of the five OSI protocol classes. This list should not be taken too literally because the details of the features sometimes differ among the various protocol classes, and not all the alternatives and variants are mentioned in the list.

In the following paragraphs we will briefly discuss each of the protocol elements listed in Fig. 6-9. All connection-oriented protocols must provide a mechanism for establishing connections. Furthermore all of them provide a way for the called party to accept or refuse a requested connection. Establishing and releasing connections will be discussed in detail in the next section.

To actually move bits across the network, the transport entities normally establish a network connection, and keep track of the mapping between transport

	Class				
Protocol element	0	1	2	3	4
Connection establishment	x	x	x	x	x
Connection refusal	x	x	x	x	x
Assignment to network connection	x	x	x	x	x
Splitting long messages into TPDUs	x	x	x	x	x
Association of TPDUs with connection	x	x	x	x	x
TPDU transfer	x	x	x	x	x
Normal release	x	x	x	x	x
Treatment of protocol errors	x	x	x	x	x
Concatenation of TPDUs to the user		x	x	x	x
Error release	x		x		
TPDU numbering		x	o	x	x
Expedited data transfer		o	o	x	x
Transport layer flow control			o	x	x
Resynchronization after a RESET		x		x	x
Retention of TPDUs until ack		x		x	x
Reassignment after network disconnect		x		x	x
Frozen references		x		x	x
Multiplexing			x	x	x
Use of multiple network connections					x
Retransmission upon timeout					x
Resequencing of TPDUs					x
Inactivity timer					x
Transport layer checksum					o

x = present o = optional (blank) = absent

Fig. 6-9. Elements of transport protocols and their relationships to the five OSI connection-oriented transport protocol classes.

connections and network connections. However, it is also possible for the transport entities to use a connectionless network protocol for data transport, provided that the transport protocol is class 4 (or a non-OSI protocol with the same functionality).

Before going on to the next item, let us first say a few words about terminology. When discussing the data link layer we called the units exchanged "frames." In the network layer we called them "packets." Both of these terms are widely used (e.g., in the CCITT X.25 recommendation). For the "transport packet" there is no comparable word, so we will use the OSI term **TPDU** (**Transport Protocol Data Unit**). We will call the item of information passed by the transport user to the transport provider a **message** since the OSI term, **TSDU** (**Transport Service Data Unit**), is rarely used. In some cases, the distinction between a message and a TPDU is not important, in which case we will use whichever term seems most appropriate in the context.

The messages to be transmitted may be of any length, so it is up to the transport layer to split them into TPDUs for transport. If a TPDU does not fit in a single packet, each TPDU may have to be split as well. One of the tasks of all five OSI protocol classes is to split long messages into the TPDU size used by the protocol, and then reassemble the pieces transparently at the other end.

If multiple connections are open on a machine, the transport entities will have to give each connection a number, and put the connection number in each TPDU, so that when a TPDU arrives at the other end, the receiving transport entity will know which connection to associate it with. Needless to say, transport of TPDUs is also a feature of all transport protocols.

Normal release of a connection is also found in all protocols, although it works slightly differently in class 0. In this class, there is always a one-to-one mapping between transport connections and network connections. The transport connection is released implicitly by just releasing the underlying network connection. With the other classes, releasing is explicit, by exchange of control TPDUs.

All protocols must deal with protocol errors. If an invalid TPDU arrives, there must be rules governing what to do. In some cases the action may be to ignore it; in others it may be to release some connection (or all connections). Protocol errors are not supposed to happen, of course, but allowing a transport entity just to crash if one occurs is not a good idea.

The remaining items on the list do not apply to all five OSI protocol classes. For example, concatenation of TPDUs applies to classes 1 through 4, but not 0. This feature allows the transport entity to collect several TPDUs and send them together as a single packet, thus reducing the number of calls to the network layer.

Error release refers to the fact that for protocol classes 0 and 2, an *N-RESET* or *N-DISCONNECT* terminates the transport connection(s) using that network connection. No attempt is made to recover.

TPDU numbering is used to keep track of the TPDUs. By assigning successive TPDUs on a connection successively higher sequence numbers, explicit acknowledgements and flow control are possible, and there is a way to figure out after an *N-RESET* which TPDU was the last one received. Class 0 (and optionally class 2) do not use sequence numbers.

Expedited data transfer is a possibility in the four upper protocol classes, but is not available in class 0.

Transport layer flow control consists of having an explicit part of the transport protocol dealing with how many TPDUs may be sent at any instant. A sliding window scheme can be used, but there are also other possibilities. If no explicit flow control scheme is used at the transport layer, the underlying flow control of the network connection is used.

Resynchronization after an *N-RESET* is done in classes 1, 3, and 4 to allow each side to discover which of the TPDUs it sent have arrived. Closely related to resynchronization is the necessity for transport entities to retain copies of TPDUs sent until they have been acknowledged, so that they can be retransmitted in the event of an *N-RESET*. Since classes 0 and 2 give an error release after an *N-RESET*, rather than attempting to resynchronize, they do not have to handle retention of TPDUs until they are acknowledged.

Reassignment after a network disconnect is related to the above problems. If the network breaks the connection altogether, rather than just resetting it, then it is up to the transport layer to establish a new connection on which to work.

Frozen references are important in networks that can store packets for a non-negligible time. The idea here is to refrain from giving a TPDU an identification that is identical to that of an older TPDU that is still in existence, just in case the old one pops up in an unexpected moment. We will discuss this subtle matter in detail later.

Next come multiplexing and use of multiple network connections. Both of these have to do with having several transport connections on one network connection or vice versa. We will also study these later.

Retransmission upon timeout is only needed in class 4, because this is the only class in which lost packets are common enough to require error control in the transport layer. We saw how the timer mechanism worked in the data link layer. It is essentially the same in the transport layer, although choosing good timeout values is more difficult (Karn and Partridge, 1987).

If TPDUs may be lost (class 4 again), the destination may have received TPDUs in the wrong order, and has to put them together again. This mechanism was also present in protocol 6 in Chapter 4.

The inactivity timer is different from the timer used to detect lost TPDUs. It is used to detect a dead network connection. If there is no sign of life from the network layer for the period of time governed by this timer, despite repeated attempts to communicate with it, the transport layer must assume something is radically wrong. The normal recovery procedure is to try to establish a new network connection.

Finally, the last item is the use of software checksums. TPDUs can be checksummed in software, to guard against networks whose own lower layer checksumming is inadequate. The checksum algorithm (Fletcher, 1982) has been designed to be easy to compute in software (basically, adding up the bytes modulo 256).

6.2. CONNECTION MANAGEMENT

As we mentioned before, in many ways transport protocols resemble data link protocols. Sliding window protocols, for example, can be used in both layers. One significant difference, however, is how connections are managed. In the data link layer, there is little connection management. The lines between the IMPs are always there and always ready for use. In the transport layer, the establishment, release, and general management of connections is much more complex. In this section we will look at some of the issues surrounding connection management in detail. We will see that special protocols are sometimes needed.

6.2.1. Addressing

When a (transport) user wishes to set up a connection to another user, he must specify which remote user to connect to. (Connectionless transport has the same problem: to whom should each message be sent?) The method normally used is to define transport service access points (TSAPs) to which processes can attach themselves and wait for connection requests to arrive. TSAPs are completely analogous to the NSAPs we saw in the previous chapter, only they exist at the top of the transport layer, rather than at the top of the network layer.

Figure 6-10 illustrates the relationship between the NSAP, TSAP, network connection, and transport connection. A possible connection scenario is as follows.

1. A time-of-day server process on machine *B* attaches itself to TSAP 122 to wait for a *T-CONNECT.indication*. How a process attaches itself to a TSAP is outside the OSI model and depends entirely on the local operating system.
2. A process on machine *A* wants to find out the time-of-day, so it issues a *T-CONNECT.request* specifying TSAP 6 as the source and TSAP 122 as the destination.
3. The transport entity on *A* selects an NSAP on its machine and on the destination machine, and sets up a network connection (e.g., an X.25 virtual circuit) between them. Using this network connection, it can talk to the transport entity on *B*.
4. The first thing the transport entity on *A* says to its peer on *B* is: "Good morning. I would like to establish a transport connection between my TSAP 6 and your TSAP 122. What do you say?"
5. The transport entity on *B* then issues the *T-CONNECT.indication*, and if the time-of-day server at TSAP 122 is agreeable, the transport connection is established.

Note that the transport connection goes from TSAP to TSAP, whereas the network connection only goes part way, from NSAP to NSAP (e.g., from X.121 address to X.121 address).

The picture painted above is fine, except we have swept one little problem under the rug: How does the user process on *A* know that the time-of-day server is attached to TSAP 122? One possibility is that the time-of-day server has been attaching itself to TSAP 122 for years, and gradually all the network users have learned this. In this model, services have stable TSAP addresses which can be printed on paper and given to new users when they join the network.

While stable TSAP addresses might work for a small number of key services

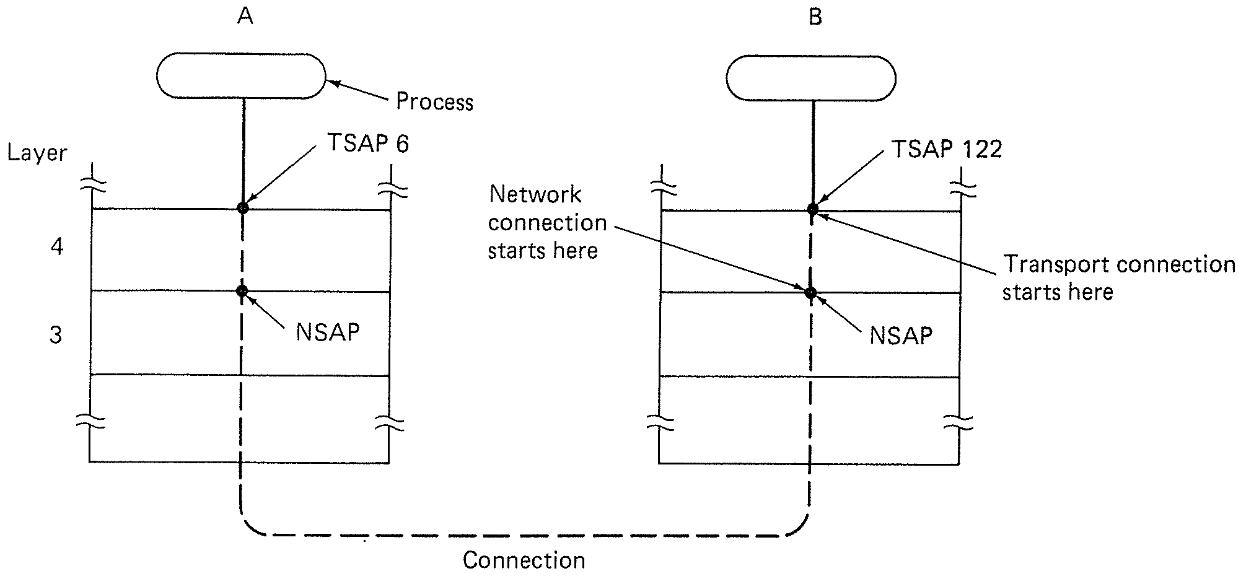

A
B
Process
Layer
TSAP 6
TSAP 122
Network connection starts here
4
Transport connection starts here
3
NSAP
NSAP
Connection

Fig. 6-10. TSAPs, NSAPs, and connections.

that never change, in general, user processes often want to talk to other user processes that only exist for a short time and do not have a TSAP address that is known in advance. Furthermore, if there are potentially many server processes, most of which are rarely used, it is wasteful to have each of them active and listening to a stable TSAP address all day long. In short, a better scheme is needed.

One such scheme, pioneered in the ARPANET, is shown in Fig. 6-11 in a simplified form. It is known as the ARPANET **initial connection protocol.** Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer service to remote users has a special **process server** (or **logger**) through which all services must be requested. Whenever the process server is idle, it listens to a well-known TSAP. Potential users of any service must begin by doing a *T-CONNECT.request*, specifying the TSAP address of the process server.

Once the connection has been established, the user sends the process server a message telling which program it wants to run (e.g., the time-of-day program). The process server then chooses an idle TSAP and spawns a new process, telling the new process to listen to the chosen TSAP. Finally, the process server sends the remote user the address of the chosen TSAP, terminates the connection, and goes back to listening on its well-known TSAP.

At this point the new process is listening on a TSAP that the user now knows, so it is possible for the user to release the connection to the process server and connect to the new process. Once this connection has been set up, the new process executes the desired program, the name of which was passed to it by the process server, together with address of the TSAP to listen to. When the server has performed its job, it releases the connection and terminates itself.

While the ARPANET initial connection protocol works fine for those servers that can be created as they are needed, there are many situations in which services do exist independently of the process server. A file server, for example, needs to

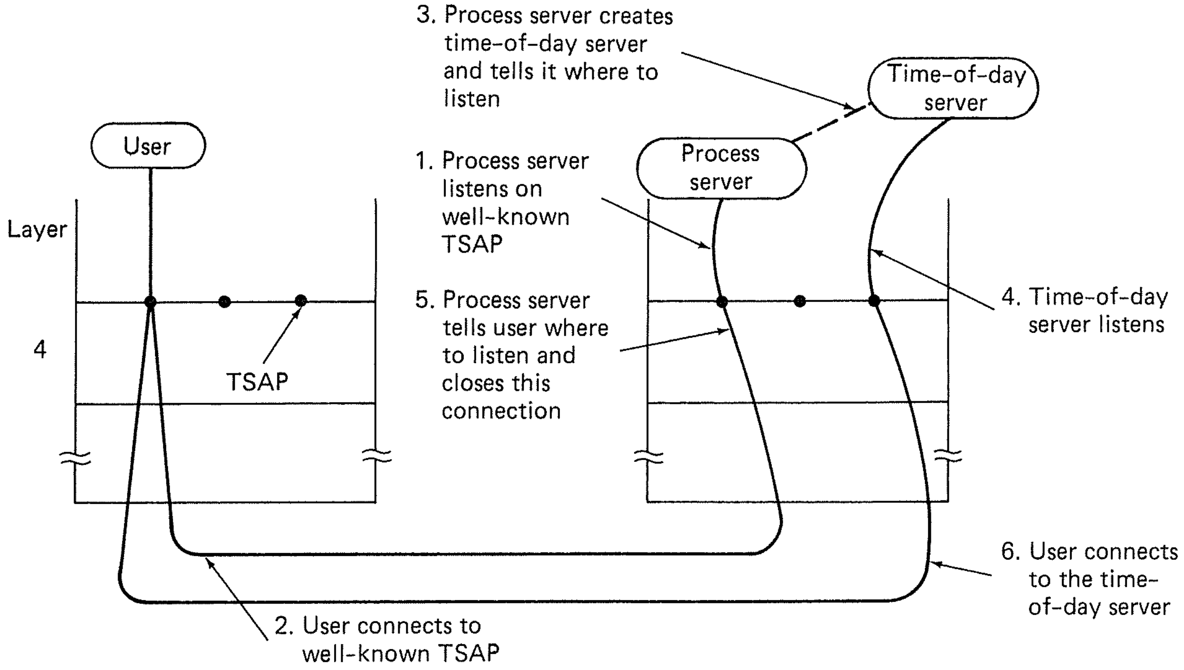

3. Process server creates time-of-day server and tells it where to listen
Time-of-day server
User
1. Process server listens on well-known TSAP
Process server
Layer
4
5. Process server tells user where to listen and closes this connection
4. Time-of-day server listens
TSAP
6. User connects to the time-of-day server
2. User connects to well-known TSAP

Fig. 6-11. How a user process in host *A* establishes a connection with a time-of-day server in host *B*.

run on special hardware (a machine with a disk) and cannot just be created on-the-fly when someone wants to talk to it.

To handle this situation, an alternative scheme is often used. In this model, there exists a special process called a **name server** or sometimes a **directory server**. To find the TSAP address corresponding to a given service name, such as "time-of-day," a user sets up a connection to the name server (which listens to a well-known TSAP). The user then sends a message specifying the service name, and the name server sends back the TSAP address. Then the user releases the connection with the name server and establishes a new one with the desired service.

In this model, when a new service is created, it must register itself with the name server, giving both its service name (typically an ASCII string) and the address of its TSAP. The name server records this information in its internal data base, so that when queries come in later, it will know the answers.

The function of the name server is analogous to the directory assistance operator in the telephone system—it provides a mapping of names onto numbers. Just as in the telephone system, it is essential that the address of the well-known TSAP used by the name server (or the process server in the initial connection protocol) is indeed well-known. If you do not know the number of the information operator, you cannot call the information operator to find it out. If you think the number you dial for information is obvious, try it in a foreign country some time.

Now let us suppose that the user has successfully located the address of the TSAP to be connected to. Another interesting question is how does the local transport entity know on which machine that TSAP is located? More specifically, how does the transport entity know which NSAP to use to set up a network connection to the remote transport entity that manages the TSAP requested?

The answer depends on the structure of TSAP addresses. One possible structure is that TSAP addresses are **hierarchical addresses**. With hierarchical addresses, the address consists of a sequence of fields used to disjointly partition the address space. For example, a truly universal TSAP address might have the following structure:

address = <galaxy> <star> <planet> <country> <network> <host> <port>

With this scheme, it is straightforward to locate a TSAP anywhere in the known universe. Equivalently, if a TSAP address is a concatenation of an NSAP address and a port (a local identifier specifying one of the local TSAPs), then when a transport entity is given a TSAP address to connect to, it uses the NSAP address contained in the TSAP address to reach the proper remote transport entity.

As a simple example of a hierarchical address, consider the telephone number 19076543210. This number can be parsed as 1-907-654-3210, where 1 is a country code (U.S. + Canada), 907 is an area code (Alaska), 654 is an end office in Alaska, and 3210 is one of the "ports" (subscriber lines) in that end office.

The alternative to a hierarchical address space is a **flat address space**. If the TSAP addresses are not hierarchical, a second level of mapping is needed to locate the proper machine. There would have to be a name server that took TSAP addresses as input and returned NSAP addresses as output. Alternatively, in some situations, it might be possible to broadcast a query asking the destination machine to please identify itself.

6.2.2. Establishing a Connection

Establishing a connection sounds easy, but it is actually surprisingly tricky, especially in a type C network. At first glance, it would seem sufficient for one transport entity to just send a *CR (CONNECTION REQUEST)* TPDU to the destination and wait for a *CC (CONNECTION CONFIRM)* reply. The problem occurs when the network can lose, store, and duplicate packets.

Imagine a subnet that is so congested that acknowledgements never get back in time, and each packet times out and is retransmitted two or three times. Suppose the subnet uses datagrams inside, and every packet follows a different route. Some of the packets might get stuck in traffic jams and take a long time to arrive, that is, they are stored in the subnet and pop out much later.

The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not entirely trustworthy person, and then releases the connection. Unfortunately, each packet in the scenario is duplicated and stored in the subnet. After the connection has been released, all the packets pop out of the subnet and arrive at the destination in order, asking the bank to establish a new connection, transfer money (again), and release the connection. The bank has no way of telling that these are duplicates, assumes this is a second, independent transaction, and transfers the money again. For the remainder of this section we will study the problem of delayed duplicates, with special emphasis on algorithms for establishing connections in a reliable way, so that nightmares like the one above cannot happen.

The crux of the problem is the existence of delayed duplicates. It can be attacked in various ways, none of them very satisfactory. One way is to use throwaway TSAP addresses. In this approach, each time a TSAP address is needed, a new, unique address is generated, typically based on the current time. When a connection is released, the addresses are discarded forever. This strategy makes the process server model of Fig. 6-11 impossible.

Another possibility is to give each connection a connection identifier (i.e., a sequence number incremented for each connection established), chosen by the initiating party, and put in each TPDU, including the one requesting the connection. After each connection is released, each transport entity could update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request came in, it could be checked against the table, to see if it belonged to a previously released connection.

Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely. If a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used.

Instead, we need to take a different tack. Rather than allowing packets to live forever within the subnet, we must devise a mechanism to kill off very old packets that are still wandering about. If we can ensure that no packet lives longer than some known time, the problem becomes somewhat more manageable.

Packet lifetime can be restricted to a known maximum using one of the following techniques:

1. Restricted subnet design.
2. Putting a hop counter in each packet.
3. Time stamping each packet.

The first method includes any method that prevents packets from looping, combined with some way of bounding congestion delay over the (now known) longest possible path. The second method consists of having the hop count incremented each time the packet is forwarded. The data link protocol simply discards any packet whose hop counter has exceeded a certain value. The third method requires each packet to bear the time it was created, with the IMPs agreeing to discard any packet older than some agreed upon time. This latter method requires the IMP clocks to be synchronized, which itself is a nontrivial task unless synchronization is achieved external to the network, for example by listening to WWV or some other radio station that broadcasts the exact time periodically.

In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are also dead, so we will now introduce T, which is some small multiple of the true maximum packet lifetime. The multiple is protocol-dependent and simply has the effect of making T longer. If we wait a time T after a packet has been sent, we can be sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear out of the blue to complicate matters.

With packet lifetimes bounded, it is possible to devise a foolproof way to establish connections safely. The method described below is due to Tomlinson (1975). It solves the problem, but introduces some peculiarities of its own. The method was further refined by Sunshine and Dalal (1978).

To get around the problem of a machine losing all memory of where it was after a crash, Tomlinson proposed equipping each host with a time of day clock. The clocks at different hosts need not be synchronized. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals. Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence numbers. Last, and most important, the clock is assumed to continue running even if the host goes down.

The basic idea is to ensure that two identically numbered TPDUs are never outstanding at the same time. When a connection is set up, the low-order k bits of the clock are used as the initial sequence number (also k bits). Thus, unlike our protocols of Chapter 4, each connection starts numbering its TPDUs with a different sequence number. The sequence space should be so large (e.g., 32 bits) that by the time sequence numbers wrap around, old TPDUs with the same sequence number are long gone. This linear relation between time and initial sequence numbers is shown in Fig. 6-12.

Once both transport entities have agreed on the initial sequence number, any sliding window protocol can be used for data flow control. In reality, the initial sequence number curve (shown by the heavy line) is not really linear, but a staircase, since the clock advances in discrete steps. For simplicity we will ignore this detail.

A problem occurs when a host crashes. When it comes up again, its transport entity does not know where it was in the sequence space. One solution is to require transport entities to be idle for T sec after a recovery to let all old TPDUs die off. However, in a complex internetwork T may be large, so this strategy is unattractive.

To avoid requiring T sec of dead time after a crash, it is necessary to introduce a new restriction on the use of sequence numbers. We can best see the need for this restriction by means of an example. Let T, the maximum packet lifetime, be 60 sec, and let the clock tick once per second. As shown in Fig. 6-12, the initial sequence number for a connection opened at time x will be x. Imagine that at $t = 30$ sec, an

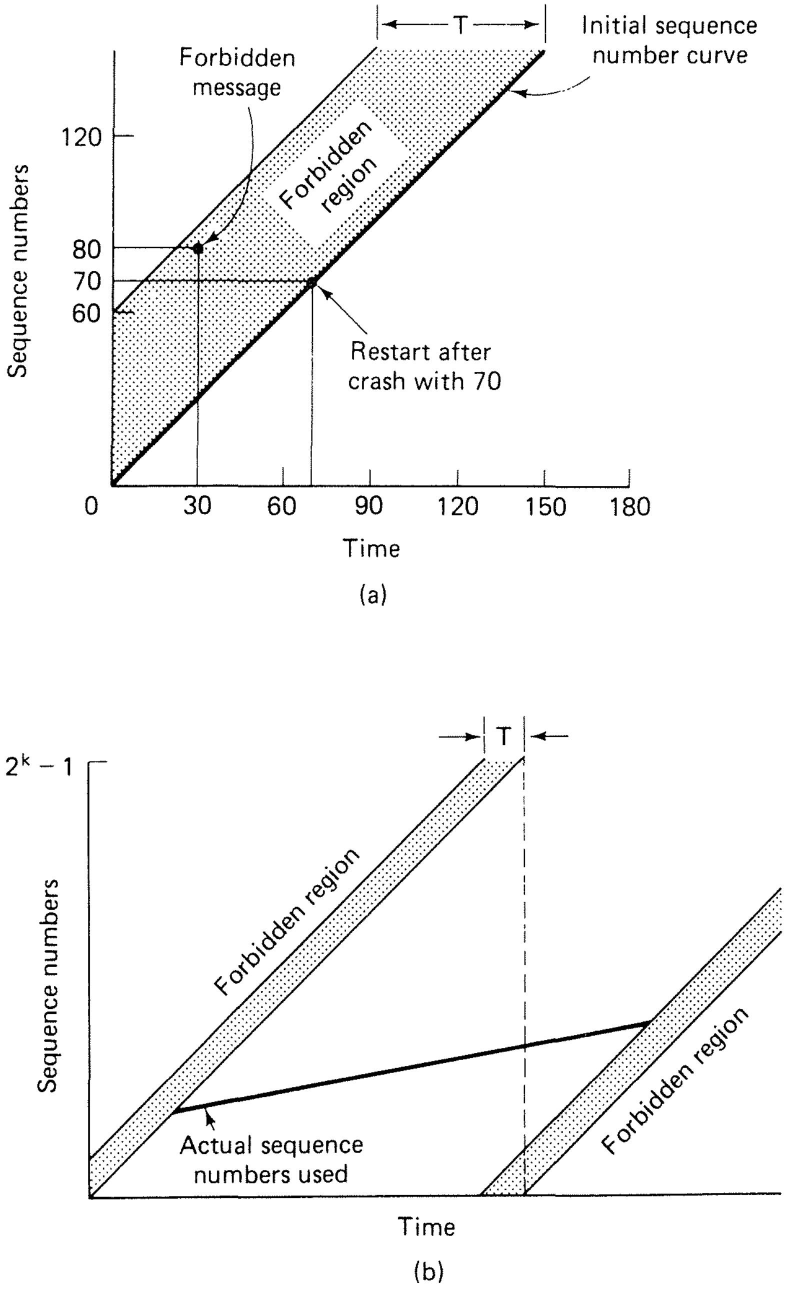


Fig. 6-12. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

ordinary data TPDU being sent on (a previously opened) connection 5 is given sequence number 80. Call this TPDU *X*. Immediately after sending TPDU *X*, the host crashes and then quickly restarts. At $t = 60$, it begins reopening connections 0 through 4. At $t = 70$, it reopens connection 5, using initial sequence number 70 as required. Within the next 15 sec it sends data TPDUs 70 through 80. Thus at $t = 85$ a new TPDU with sequence number 80 and connection 5 has been injected into the subnet. Unfortunately, TPDU *X* still exists. If it should arrive at the receiver before the new TPDU 80, it will be accepted and the correct TPDU rejected.

To prevent such problems, we must prevent sequence numbers from being used (i.e., assigned to new TPDUs) for a time *T* before their potential use as initial sequence numbers. The illegal combinations of time and sequence number are shown as the **forbidden region** in Fig. 6-12(a). Before sending any TPDU on any connection, the transport entity must read the clock and check to see that it is not in the forbidden region.

The protocol can get itself into trouble in two different ways. If a host sends too much data too fast on a newly opened connection, the actual sequence number versus time curve may rise more steeply than the initial sequence number versus time curve. This means that the maximum data rate on any connection is one TPDU per clock tick. It also means that the transport entity must wait until the clock ticks before opening a new connection after a crash restart, lest the same number be used twice. Both of these points argue for a short clock tick (a few milliseconds).

Unfortunately, entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-12(b), it should be clear that at any data rate less than the clock rate, the curve of actual sequence numbers used versus time will eventually run into the forbidden region from the left. The greater the slope of the actual sequence number curve, the longer this event will be delayed. As we stated above, just before sending every TPDU, the transport entity must check to see if it is about to enter the forbidden region, and if so, either delay the TPDU for *T* sec or resynchronize the sequence numbers.

The clock based method solves the delayed duplicate problem for data TPDUs, but for this method to be useful, a connection must first be established. Since control TPDUs may also be delayed, there is a potential problem in getting both sides to agree on the initial sequence number. Suppose, for example, that connections are established by having one machine *A*, send a *CR* TPDU containing the proposed initial sequence number and destination port number to a remote peer, *B*. The receiver, *B*, then acknowledges this request by sending a *CC* TPDU back. If the *CR* TPDU is lost but a delayed duplicate *CR* suddenly shows up at *B*, the connection will be established incorrectly.

To solve this problem, Tomlinson (1975) has introduced the **three-way handshake**. This establishment protocol does not require both sides to begin sending with the same sequence number, so it can be used with synchronization methods other than the global clock method. The setup procedure when *A* initiates is shown in Fig. 6-13. In this figure, *A* is the transport entity (not transport user) to the left of the vertical lines, and *B* is the transport entity to the right. The arrows denote TPDUs sent and received, not transport user primitives, as was the case when we were discussing the service rather than the protocol. *A* chooses a sequence number somehow, say *x*, and sends it to *B*. *B* replies with a *CC* TPDU acknowledging *x* and announcing its own initial sequence number, *y* (which may be equal to *x*, of course). Finally, *A* acknowledges *B*'s choice of an initial sequence number in its first data TPDU.

Now let us see how the three-way handshake works in the presence of delayed

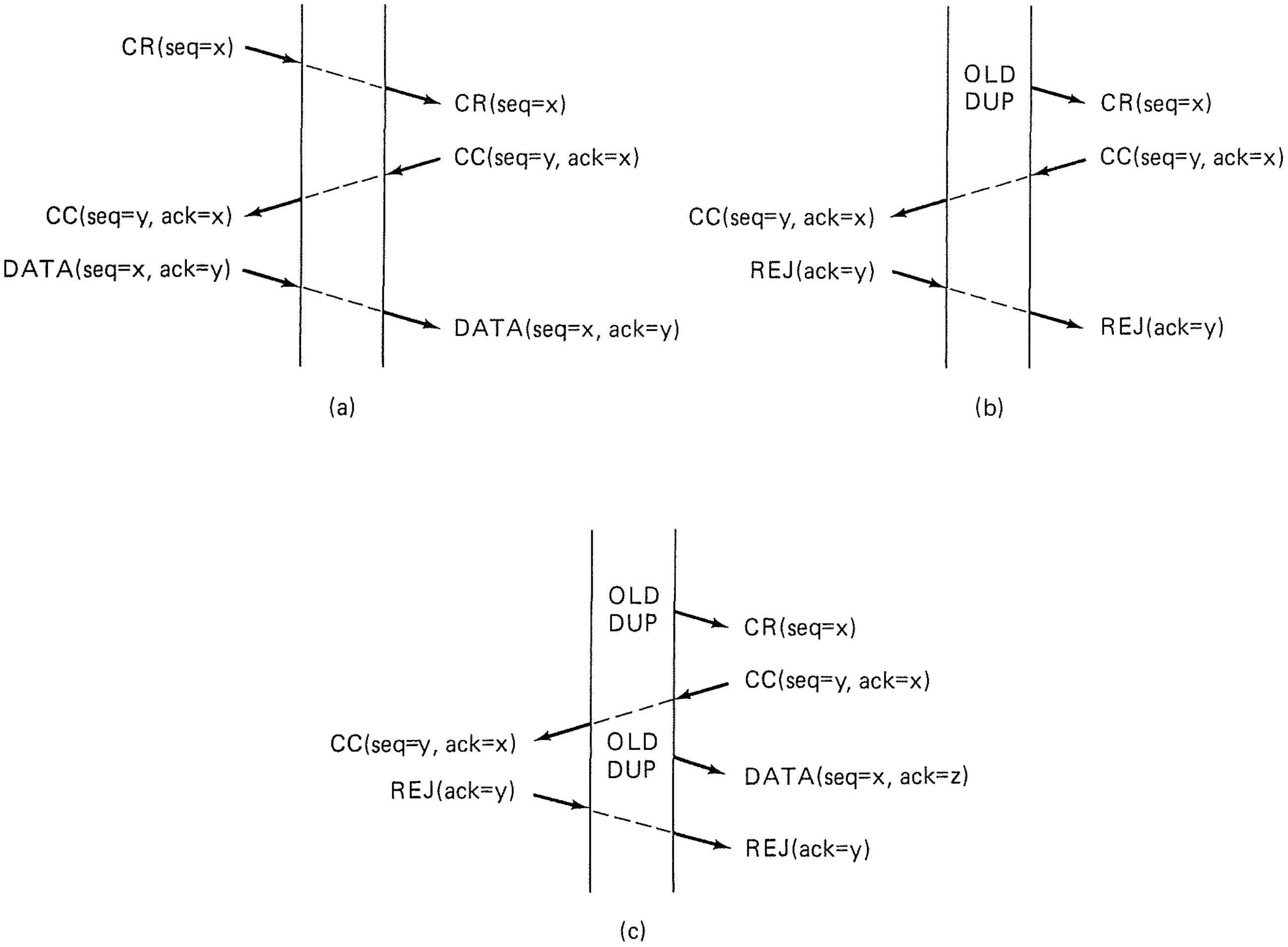


Fig. 6-13. Three protocol scenarios for establishing a connection using a three-way handshake. (a) Normal operation. (b) Old duplicate *CR* appearing out of nowhere. (c) Duplicate *CR* and duplicate ACK.

duplicate control TPDUs. In Fig. 6-13(b), the first TPDU is a delayed duplicate *CR* from a connection since released. This TPDU arrives at *B* without *A*'s knowledge. *B* reacts to this TPDU by sending *A* a *CC* TPDU, in effect asking for verification that *A* was indeed trying to set up a new connection. When *A* rejects *B*'s attempt to establish, *B* realizes that it was tricked by a delayed duplicate and abandons the connection.

The worst case is when both a delayed *CR* and an acknowledgement to a *CC* are floating around in the subnet. This case is shown in Fig. 6-13(c). As in the previous example, *B* gets a delayed *CR* and replies to it. At this point it is crucial to realize that *B* has proposed using *y* as the initial sequence number for *B* to *A* traffic, knowing full well that no TPDUs containing sequence number *y* or acknowledgements to *y* are still in existence. When the second delayed TPDU arrives at *B*, the fact that *z* has been acknowledged rather than *y* tells *B* that this, too, is an old duplicate.

6.2.3. Releasing a Connection

Releasing a connection is much easier than establishing one. Nevertheless, there are more pitfalls than one might expect. In Fig. 6-4(d)-(f) we see three ways that a connection can be released. The first one is the most common, with one of the users issuing a *T-DISCONNECT.request* primitive. The transport layer then generates a *T-DISCONNECT.indication* on the other end, and the connection is released.

The second way that release can happen is when both users issue a *T-DISCONNECT.request* simultaneously. The third way is when the transport layer throws in the towel and issues *T-DISCONNECT.indications* on both ends of the connection. Theoretically, there is also a fourth way, in which one user issues a *T-DISCONNECT.request* and before the *T-DISCONNECT.indication* can happen at the other end, the transport layer itself disconnects the other end.

All of these forms of disconnect have one feature in common: they are abrupt and may result in data loss. Consider the scenario of Fig. 6-14. After the connection is established, *A* sends a TPDU that arrives properly at *B*. Then *A* sends another TPDU and disconnects. Unfortunately, *B* issues a *T-DISCONNECT.request* before the second TPDU arrives. The result is that the connection is released and data are lost.

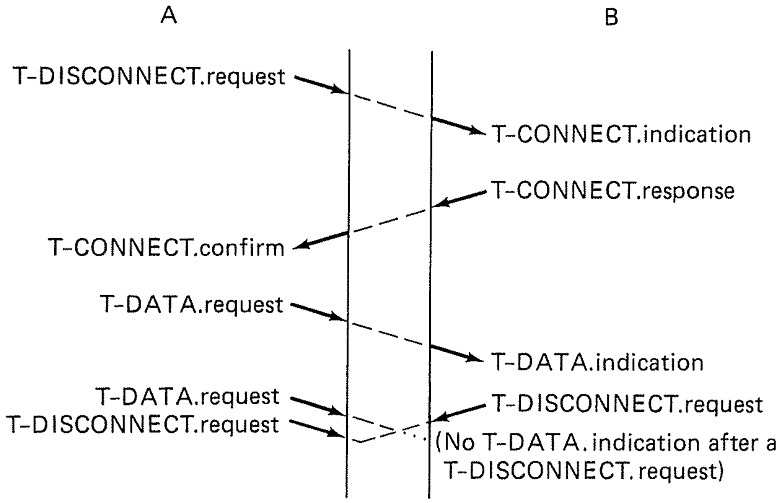


Fig. 6-14. Abrupt disconnection with loss of data.

Clearly a more sophisticated release protocol is required to avoid data loss. An obvious way to handle release is not to have either side issue a *T-DISCONNECT.request* until it is sure that the other side has received all data that has been sent, and has no more data to send itself. In other words, the protocol could be something like *A* saying: "I am done. Are you done too?" If *B* responds: "I am done too. Goodbye." the connection can be safely released.

Unfortunately, this protocol does not always work. There is a famous problem that deals with this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-15. On both of the surrounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together they are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

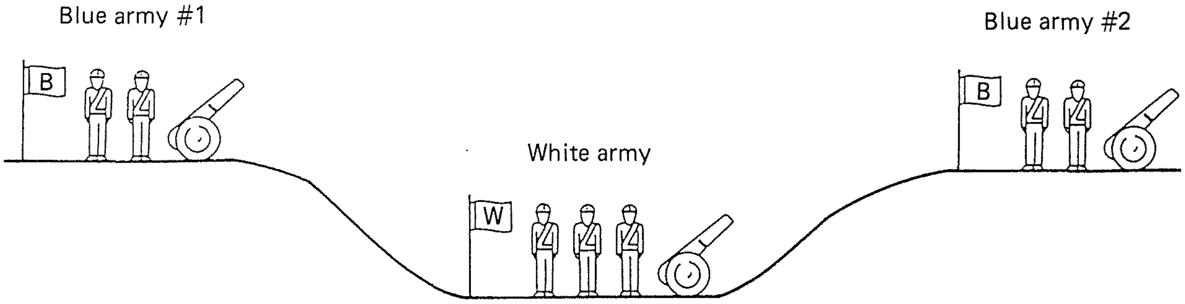


Fig. 6-15. The two-army problem.

The blue armies obviously want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel). The question is, does a protocol exist that allows the blue armies to win?

Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?" Now suppose that the message arrives, and the commander of blue army #2 agrees, and that his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

Now let us improve the protocol by making it a three-way handshake. The initiator of the original proposal must acknowledge the response. Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate. After all, he does not know if his acknowledgement got through, and if it did not, he knows that blue army #2 will not attack. We could now make a four-way handshake protocol, but that does not help either.

In fact, it can easily be proven that no protocol exists that works. Suppose that some protocol did exist. Either the last message of the protocol is essential or it is not. If it is not, remove it (and any other unessential messages) until we are left with a protocol in which every message is essential. What happens if the final message does not get through? We just said that it was essential, so if it is lost, the attack does not take place. Since the sender of the final message can never be sure of its arrival, he will not risk attacking. Worse yet, the other blue army knows this, so it will not attack either.

To see the relevance of the two-army problem to releasing connections, just substitute "disconnect" for "attack." If neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

In practice, one is usually prepared to take more risks when releasing connections than when attacking white armies, so the situation is not entirely hopeless. Figure 6-16 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate. Note that it shows the TPDUs sent and received by the transport entities, not the primitives seen by the transport users.

In Fig. 6-16(a), we see the normal case in which one of the users sends a *DR (DISCONNECT REQUEST)* TPDU in order to initiate the connection release. When it arrives, the recipient sends back a *DC (DISCONNECT CONFIRM)* TPDU and starts a timer, just in case the *DC* is lost. When the *DC* arrives, the original sender sends back an *ACK* TPDU and deletes the connection. Finally, when the ACK arrives, the receiver also deletes the connection. Note that "delete connection" in this context means that the transport entity removes the information about the connection from its table of open connections, and signals the connection's owner (the transport user) somehow. This action is completely different from a transport user issuing a *T-DISCONNECT.request* primitive.

If the final ACK is lost, as shown in Fig. 6-16(b), the situation is saved by the timer. When the timer expires, the connection is deleted anyway.

Now consider the case of the *DC* (or *DR)* being lost. The user initiating the disconnect will not receive the *DC,* will time out, and will start all over again. In Fig. 6-16(c) we see how this works, assuming that the second time no TPDUs are lost.

Our last scenario, Fig. 6-16(d), is the same as Fig. 6-16(c) except that now we assume all the repeated attempts to retransmit the *DR* also fail due to lost TPDUs. After *n* retries, the sender just gives up and deletes the connection. Meanwhile, the receiver times out and also exits.

While this protocol usually suffices, in theory it can fail if the initial DR and *n* retransmissions are all lost. The sender will give up and delete the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection.

We could have avoided this problem by not allowing the sender to give up after *n* retries, but forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, then the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, then the protocol hangs in Fig. 6-16(b).

One way to kill off half-open connections is to have a rule saying that if no TPDUs have arrived for a certain number of seconds, the connection is automatically disconnected. That way, if one side ever disconnects, the other side will detect the lack of activity and also disconnect. Of course, if this rule is introduced, it is necessary for each transport entity to have a timer that is stopped and then re-

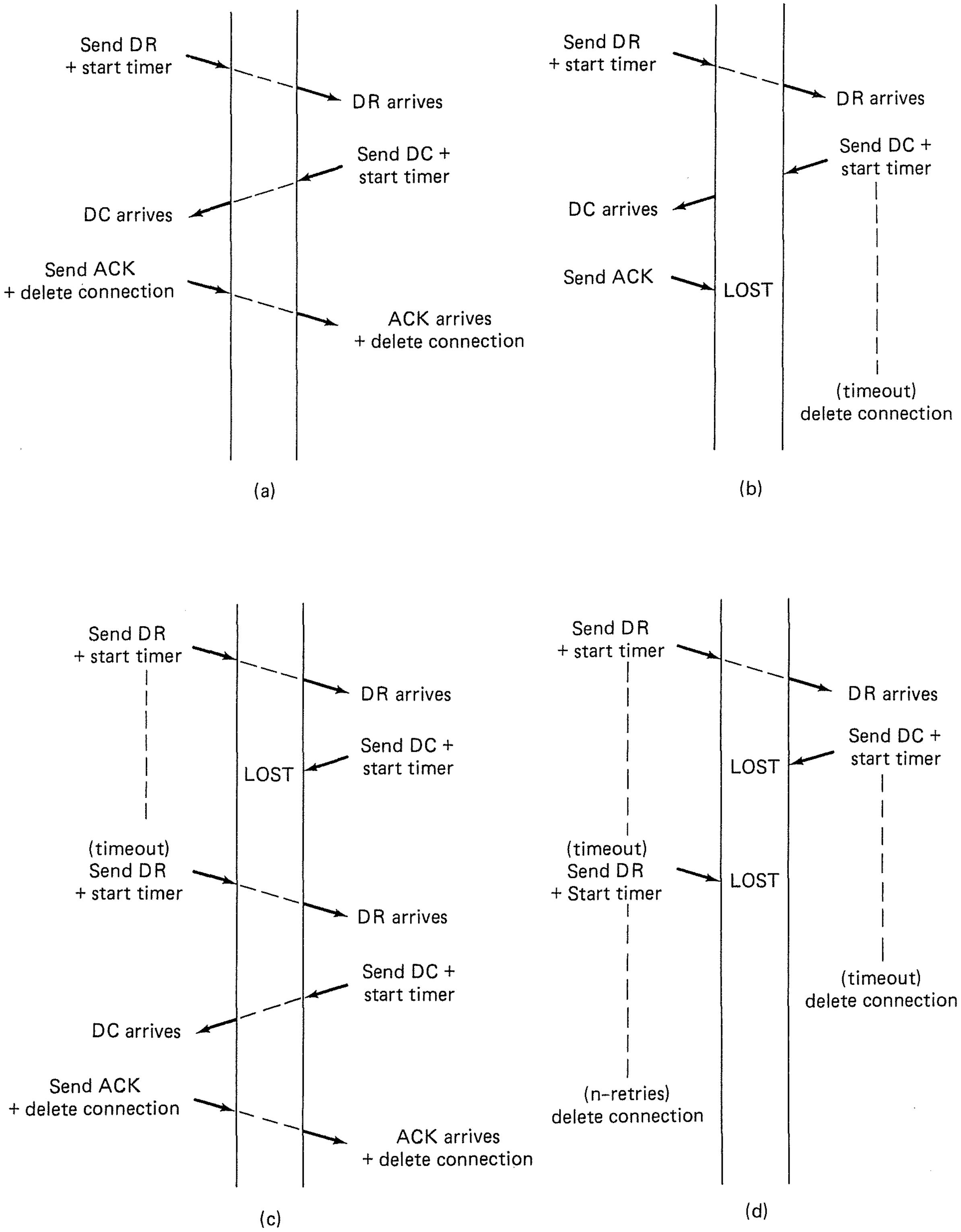


Fig. 6-16. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final *ACK* lost. (c) *DC* lost. (d) *DC* lost and subsequent *DRs* lost.

started whenever a TPDU is sent. If this timer expires, a dummy TPDU is transmitted, just to keep the other side from disconnecting. On the other hand, if the automatic disconnect rule is used and too many dummy TPDUs in a row are lost on an otherwise idle connection, first one side, then the other side will automatically disconnect.

We will not belabor this point any more, but by now it should be clear that releasing a connection is not nearly as simple as it at first appears.

6.2.4. Timer-Based Connection Management

Let us take a step back. The original problem we posed was how to avoid the nightmare of old, duplicate packets containing *CR, DATA,* and *DR* TPDUs suddenly appearing from nowhere and being accepted as legitimate. One way is for each transport entity to assign a unique connection identifier to each connection so it can recognize TPDUs from previous connections. A second way is to use three-way handshakes for establishing connections. Fletcher and Watson (1978) and Watson (1981) have proposed a third way based on timers. We will describe their method in this section.

The basic idea is closely related to the proposal earlier to have a connection simply time out if there is no activity. What Fletcher and Watson have done, is arrange for the transport entity to refrain from deleting information about a connection until all the TPDUs relating to it have died off. Thus in their scheme, a transport entity's table entry about a connection is not deleted when the connection is released, but when a carefully chosen time interval has expired.

The heart of their scheme is this: when a sender wishes to send a stream of consecutive TPDUs to a receiver, it creates a **connection record** internally. This connection record keeps track of which TPDUs have been sent, which acknowledgements have been received, and so on. Whenever a connection record is created a timer is started. Whenever a TPDU is sent using a previously created connection record, the timer is started all over again. If the timer expires (meaning nothing has been sent for a certain interval), the connection record is deleted. The time intervals for sender and receiver are different and carefully chosen so that the receiver will always time out and delete its connection record well before the sender.

The first TPDU in the stream contains a 1-bit flag called *DRF* (Data Run Flag), which identifies it as the first in a run of TPDUs. When any TPDU is sent, a timer is started. If the TPDU is acknowledged, the timer is stopped. If the timer goes off, the TPDU is retransmitted. If, after *n* retransmissions, there is still no acknowledgement, the sender gives up. This give-up time plays an important role in the protocol.

When a TPDU with the *DRF* flag set arrives at the receiver, the receiver notes its sequence number and creates a connection record. Subsequent TPDUs will only be accepted if they are in sequence. If the first TPDU to arrive at the receiver does not have the *DRF* flag set, it is discarded. Eventually the first TPDU (original or retransmission) will arrive and the connection record can be created. In other words, a connection record is only created when a TPDU with *DRF* set arrives.

Whenever a TPDU arrives in sequence, it can be passed to the transport user and an acknowledgement returned to the sender. If a TPDU arrives out of sequence when a connection record exists, it may be buffered (like protocol 6 in Chapter 4). It may also be acknowledged. However, an acknowledgement does not imply that all the previous TPDUs have been received as well, unless a flag, *ARF* (Acknowledgment Run Flag), is set.

Let us first consider a simple case of how this protocol works. A sequence of TPDUs is sent and all are received in order and acknowledged. When the sender gets the acknowledgement of the final TDPU sent and sees the *ARF* flag, it stops all the retransmission timers. If no more data are forthcoming from the transport user, eventually the receiver's connection record times out and then later the sender's does too. No explicit establishment or release TPDUs are needed (although the *DRF* flag is somewhat analogous to a *CR* TPDU).

Now consider what happens if the TPDU bearing the *DRF* flag is lost. The receiver will not create a connection record. Eventually the sender will time out and retransmit the TPDU, repeatedly if necessary, until it is acknowledged. Once a connection record has been created by the receiver, a gap in the TPDU stream is easily detected and repaired by sender timeouts and retransmissions.

Suppose a stream of TPDUs is sent and correctly received, but some of the acknowledgements are lost. Subsequent retransmissions are also lost, so the receiver's connection record times out and is deleted. What is to prevent an old duplicate of the TPDU with the *DRF* flag from now appearing at the receiver and triggering a new connection record? The trick is to make the receiver's connection record timer much longer than the sender's give-up timer plus the maximum TPDU lifetime. Thus once the initial TPDU is accepted, the connection record will be kept in existence until the receiver is certain that the sender has stopped sending the TPDU with the *DRF* flag (either because it got an acknowledgement or it gave up). In this way, once the initial TPDU has been accepted, there is no danger that it will appear after the connection record has been deleted—all copies of it are definitely gone. The other TPDUs in the run are harmless because the receiver will not accept them (only TPDUs with the *DRF* flag are acceptable when the receiver has no connection record).

Furthermore, if some TPDUs remain unacknowledged, the sender will keep retransmitting them, thus keeping its connection record alive. As long as the connection record shows unacknowledged TPDUs outstanding, no new TPDU with *DRF* set will be sent.

Now let us see what happens if the transport user sends messages in bursts. Suppose a burst of TPDUs is sent and all are acknowledged. When the transport user finally gets around to sending a new message, one of three situations must hold:

1. Both sender and receiver still have their connection records.
2. The sender has its connection record but the receiver does not.
3. Both connection records have been deleted.

It cannot happen that the sender's record has been deleted while the receiver's is still around because the timer for the sender's record is intentionally longer to prevent just this case. In case 1, the next TPDU will carry a *DRF* flag and will start a new run; it will be numbered one higher than the previous TPDU because the connection record still exists. The receiver will accept it without problems. In case 2, the receiver will create a new connection record and regard it as the start of a new run, since it has already forgotten the previous run. In case 3, the sender will create a new connection record, which means that the TPDU will not be numbered in sequence with the previous ones. However, the receiver no longer knows what the previous ones were, so it does not matter. It should be clear now that the fourth case (receiver knowing which TPDU to expect but sender not knowing which one to send) has been carefully forbidden for good reason.

In summary, this protocol has an interesting mixture of connectionless and connection-oriented properties. The minimum exchange is two TPDUs, which makes it as efficient as a connectionless protocol for query-response applications. Unlike a connectionless protocol, however, if it turns out that a sequence of TPDUs must be sent, they are guaranteed to be delivered in order. Finally, connections are released automatically by the clever use of timers.

6.2.5. Flow Control and Buffering

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. One of the key issues has come up before: flow control. In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different. The basic similarity is that in both layers a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver. The main difference is that the IMP usually has relatively few lines whereas the host may have numerous connections. This difference makes it impractical to implement the data link buffering strategy in the transport layer.

In the data link protocols of Chapter 4, frames were buffered at both the sending IMP and at the receiving IMP. In protocol 6, for example, both sender and receiver are required to dedicate *MaxSeq* + 1 buffers to each line, half for input and half for output. For a host with a maximum of, say, 64 connections, and a 4-bit sequence number, this protocol would require 1024 buffers.

In the data link layer, the sending side must buffer outgoing frames because they might have to be retransmitted. If the subnet provides datagram service, the sending transport entity must also buffer, and for the same reason. If the receiver knows that the sender buffers all TPDUs until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit. The receiver may, for example, maintain a single buffer pool shared by all connections. When a TPDU comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the TPDU is accepted, otherwise it is discarded. Since the sender is prepared to retransmit TPDUs lost by the subnet, no harm is done by having the receiver drop TPDUs. The sender just keeps trying until it gets an acknowledgement.

In summary, if the network service is unreliable (i.e., type B or C), the sender must buffer all TPDUs sent, just as in the data link layer. However, with reliable (type A) network service, other trade-offs become possible. In particular, if the sender knows that the receiver always has buffer space, it need not retain copies of the TPDUs it sends. However, if the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway. In the latter case, the sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the TPDU arrived, not that it was accepted. We will come back to this important point later.

Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size. If most TPDUs are nearly the same size, it is natural to organize the buffers as a pool of identical size buffers, with one TPDU per buffer, as in Fig. 6-17(a). However, if there is wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen equal to the largest possible TPDU, space will be wasted whenever a short TPDU arrives. If the buffer size is chosen less than the maximum TPDU size, multiple buffers will be needed for long TPDUs, with the attendant complexity.

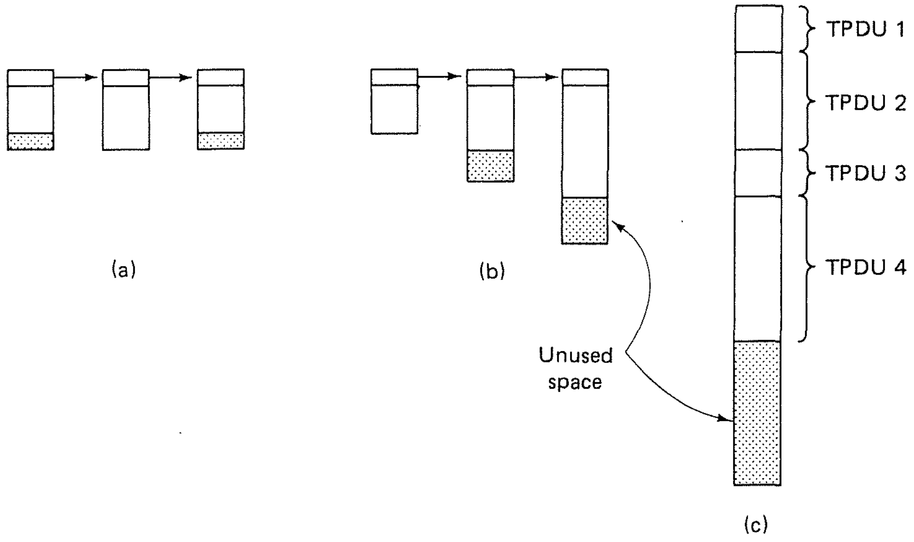


Fig. 6-17. (a) Chained fixed-size buffers. (b) Chained variable-size buffers. (c) One large circular buffer per connection.

Another approach to the buffer size problem is to use variable-size buffers, as in Fig. 6-17(b). The advantage here is better memory utilization, at the price of far more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-17(c). This system also makes good use of memory, provided that all connections are heavily loaded, but is poor if some connections are lightly loaded.

The optimum trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection. For low-bandwidth bursty traffic, such as that produced by an interactive terminal, it is better not to dedicate any buffers, but rather to acquire them dynamically at both ends. Since the sender cannot be sure the receiver will be able to acquire a buffer, the sender must retain a copy of the TPDU until it is acknowledged. On the other hand, for file transfer and other high-bandwidth traffic, it is better if the receiver does dedicate a full window of buffers, to allow the data to flow at maximum speed. Thus for low-bandwidth bursty traffic, it is better to buffer at the sender, and for high-bandwidth, smooth traffic it is better to buffer at the receiver.

As connections are opened and closed, and as the traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations. Consequently, the transport protocol should allow a sending host to request buffer space at the other end. Buffers could be allocated per connection, or collectively, for all the connections running between the two hosts. Alternatively, the receiver, knowing its buffer situation (but not knowing the offered traffic), could tell the sender "I have reserved *X* buffers for you." If the number of open connections should increase, it may be necessary for an allocation to be reduced, so the protocol should provide for this possibility.

A reasonably general way to manage dynamic buffer allocation is to decouple the buffering from the acknowledgements, in contrast to the sliding window protocols of Chapter 4. Dynamic buffer management means, in effect, a variable-sized window. Initially, the sender requests a certain number of buffers, based on its perceived needs. The receiver then grants as many of these as it can afford. Every time the sender transmits a TPDU, it must decrement its allocation, stopping altogether when the allocation reaches zero. The receiver then separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic.

Figure 6-18 shows an example of how dynamic window management might work in a datagram subnet with 4-bit sequence numbers. Assume that buffer allocation information travels in separate TPDUs, as shown, and is not piggybacked onto reverse traffic. Initially, *A* wants eight buffers, but is only granted four of these. It then sends three TPDUs, of which the third is lost. TPDU 6 acknowledges receipt of all TPDUs up to and including sequence number 1, thus allowing *A* to release those buffers, and furthermore informs *A* that it has permission to send three more TPDUs starting beyond 1 (i.e., TPDUs 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send TPDUs 3 and 4, which it proceeds to do. At this point it is blocked, and must wait for more buffer allocation. Timeout induced retransmissions (line 9), however, may occur while blocked, since they use buffers that have already been allocated. In line 10, *B* acknowledges receipt of all TPDUs up to and including 4, but refuses to let *A* continue. Such a situation is impossible with the fixed window protocols of Chapter 4. The next TPDU from *B* to *A* allocates another buffer and allows *A* to continue.

	A	Message	B	Comments
1	→	< request 8 buffers >	→	A wants 8 buffers
2	←	< ack = 15, buf = 4 >	←	B grants messages 0-3 only
3	→	< seq = 0, data = m0 >	→	A has 3 buffers left now
4	→	< seq = 1, data = m1 >	→	A has 2 buffers left now
5	→	< seq = 2, data = m2 >	. . .	Message lost but A thinks it has 1 left
6	←	< ack = 1, buf = 3 >	←	B acknowledges 0 and 1, permits 2-4
7	→	< seq = 3, data = m3 >	→	A has 1 buffer left
8	→	< seq = 4, data = m4 >	→	A has 0 buffers left, and must stop
9	→	< seq = 2, data = m2 >	→	A times out and retransmits
10	←	< ack = 4, buf = 0 >	←	Everything acknowledged, but A still blocked
11	←	< ack = 4, buf = 1 >	←	A may now send 5
12	←	< ack = 4, buf = 2 >	←	B found a new buffer somewhere
13	→	< seq = 5, data m5>	→	A has 1 buffer left
14	→	< seq = 6, data = m6 >	→	A is now blocked again
15	←	< ack = 6, buf = 0 >	←	A is still blocked
16	. . .	< ack = 6, buf = 4 >	←	Potential deadlock

Fig. 6-18. Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU.

Potential problems with buffer allocation schemes of this kind can arise in datagram networks if control TPDUs can get lost. Look at line 16. *B* has now allocated more buffers to *A*, but the allocation TPDU was lost. Since control TPDUs are not sequenced or timed out, *A* is now deadlocked. To prevent this situation, each host should periodically send control TPDUs giving the acknowledgement and buffer status on each connection. That way, the deadlock will be broken, sooner or later.

Up until now we have tacitly assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver. As memory prices continue to fall dramatically, it may become feasible to equip hosts with so much memory that lack of buffers is rarely, if ever, a problem.

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the subnet. If adjacent IMPs can exchange at most *x* frames/sec and there are *k* disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than *kx* TPDUs/sec, no matter how much buffer space is available at each end. If the sender pushes too hard (i.e., sends more than *kx* TPDUs/sec), the subnet will become congested, because it will be unable to deliver TPDUs as fast as they are coming in.

What is needed is a mechanism based on the subnet's carrying capacity rather than on the receiver's buffering capacity. Clearly, the flow control mechanism must be applied at the sender, to prevent it from having too many unacknowledged TPDUs outstanding at once. Belsnes (1975) has proposed using a sliding window flow control scheme in which the sender dynamically adjusts the window size to match the network's carrying capacity. If the network can handle *c* TPDUs/sec and the cycle time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is *r*, then the sender's window should be *cr*. With a window of this size the sender normally operates with the pipeline full. Any small decrease in network performance will cause it to block.

In order to adjust the window size periodically, the sender could monitor both parameters and then compute the desired window size. The carrying capacity can be determined by simply counting the number of TPDUs acknowledged during some time period and then dividing by the time period. During the measurement, the sender should send as fast as it can, to make sure that the network's carrying capacity, and not the low input rate, is the factor limiting the acknowledgement rate. The time required for a transmitted TPDU to be acknowledged can be measured exactly and a running mean maintained. Since the capacity of the network depends on the amount of traffic in it, the window size should be adjusted frequently, to track changes in the carrying capacity.

6.2.6. Multiplexing

Multiplexing several conversations onto connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer the need for multiplexing can arise in a number of ways. For example, in networks that use virtual circuits within the subnet, each open connection consumes some table space in the IMPs for the entire duration of the connection. If buffers are dedicated to the virtual circuit in each IMP as well, a user who left his terminal logged into a remote machine during a coffee break is nevertheless consuming expensive resources. Although this implementation of packet switching defeats one of the main reasons for having packet switching in the first place—to bill the user based on the amount of data sent, not the connect time—a number of PTTs have chosen this approach, presumably because it so closely resembles the circuit switching model to which they have grown accustomed over the decades.

The consequence of a price structure that heavily penalizes installations for having many virtual circuits open for long periods of time is to make multiplexing of different transport connections onto the same network connection attractive. This form of multiplexing, called **upward multiplexing**, is shown in Fig. 6-19(a). In this figure, four distinct transport connections all use the same network connection (e.g., X.25 virtual circuit) to the remote host. When connect time forms the major component of the carrier's bill, it is up to the transport layer to group transport connections according to their destination and map each group onto the minimum number of network connections. If too many transport connections are mapped onto one network connection, the performance will be poor, because the window will usually be full, and users will have to wait their turn to send one message. If too few transport connections are mapped onto one network connection, the service will be expensive. When upward multiplexing is used with X.25, we have the ironic (tragic?) situation of having to identify the connection using a field in the transport header, even though X.25 provides more than 4000 virtual circuit numbers expressly for that purpose.

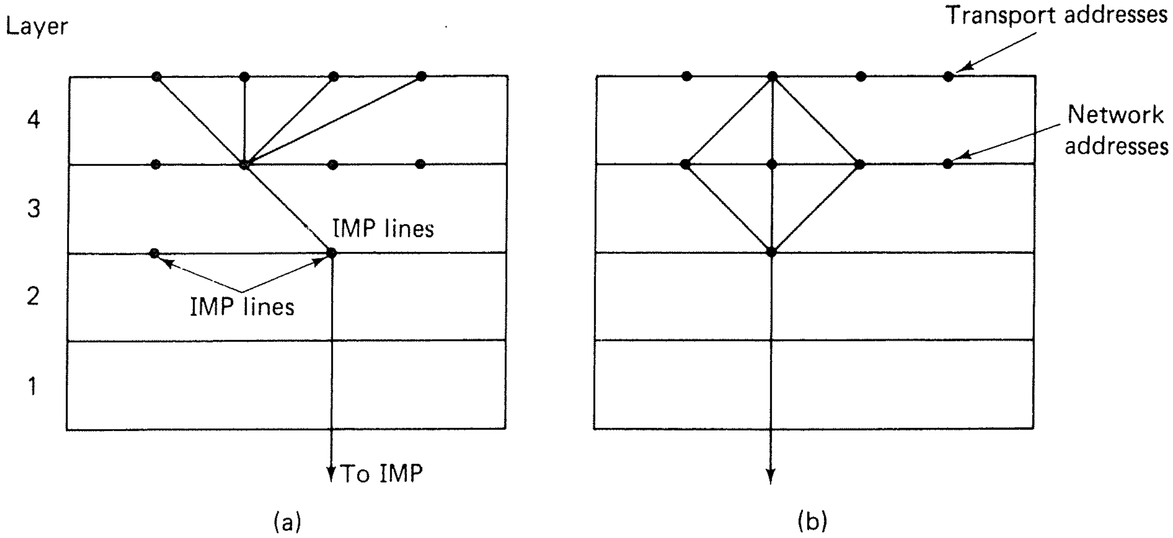


Fig. 6-19. (a) Upward multiplexing. (b) Downward multiplexing.

Multiplexing can also be useful in the transport layer for another reason, related to carrier technical decisions rather than carrier pricing decisions. Suppose, for example, that a certain important user needs a high-bandwidth connection from time to time. If the subnet enforces a sliding window flow control with a 3-bit sequence number, the user must stop sending as soon as seven packets are outstanding, and must wait for the packets to propagate to the remote host and be acknowledged. If the physical connection is via a satellite, the user is effectively limited to seven packets every 540 msec. With 128-byte packets, the usable bandwidth is about 13 kbps, even though the physical channel bandwidth is more than 1000 times higher.

One possible solution is to have the transport layer open multiple network connections, and distribute the traffic among them on a round-robin basis, as indicated in Fig. 6-19(b). This modus operandi is called **downward multiplexing**. With k network connections open, the effective bandwidth is increased by a factor of k. With 4095 X.25 virtual circuits, 128-byte packets, and a 3-bit sequence number, it is theoretically possible to achieve data rates in excess of 50 Mbps. Of course, this performance can be achieved only if the host-IMP line can support 50 Mbps, because all 4095 virtual circuits are still being sent out over one physical line, at least in Fig. 6-19(b). If multiple host-IMP lines are available, downward multiplexing can also be used to increase the performance even more.

6.2.7. Crash Recovery

If hosts and IMPs are subject to crashes, recovery from these crashes becomes an issue. If the network layer issues an *N-RESET*, for example, the transport entities must exchange information after the crash to determine which TPDUs were received and which were not. In effect, after a crash host *A* can ask host *B*: "I have four unacknowledged TPDUs outstanding, 2, 3, 4, and 5; have you received any of them?" Based on the answer, *A* can retransmit the appropriate TPDUs, provided that it has kept copies of them. If the host simply assumes that the subnet is reliable and does not keep copies, it will not be able to recover in this manner.

A more troublesome problem is how to recover from host crashes. To illustrate the difficulty, let us assume that one host, the sender, is sending a long file to another host, the receiver, using a simple stop-and-wait protocol. The transport layer on the receiving host simply passes the incoming TPDUs to the transport user, one by one. Part way through the transmission the receiver crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the receiver might send a broadcast TPDU to all other hosts, announcing that it had just crashed and requesting the other hosts to inform it of the status of all open connections. The sender can be in one of two states: one TPDU outstanding, *S1*, or no TPDUs outstanding, *S0*. Based on only this state information, the sender must decide whether or not to retransmit the most recent TPDU.

At first glance it would seem obvious that the sender should retransmit only if it has an unacknowledged TPDU outstanding (i.e., is in state *S1*) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation when the receiving host first sends an acknowledgement, and then, when the acknowledgement has been sent, performs the write. Writing a TPDU onto the output stream and sending an acknowledgement are considered as two distinct indivisible events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent, but before the write has been done, the other host will receive the acknowledgement and thus be in state *S0* when the crash recovery announcement arrives. The sender will therefore not retransmit, thinking the TPDU has arrived correctly, leading to a missing TPDU.

At this point you may be thinking: "That problem can be solved easily. All you have to do is reprogram the transport entity to first do the write, and then send the acknowledgement." Try again. Imagine that the write has been done but the crash occurs before the acknowledgement can be sent. The sender will be in state *S1* and thus retransmit, leading to an undetected duplicate TPDU in the output stream.

No matter how the sender and receiver are programmed, there are always situations where the protocol fails to recover properly. The receiver can be programmed in one of two ways: acknowledge first or write first. The sender can be programmed in one of four ways: always retransmit the last TPDU, never retransmit the last TPDU, retransmit only in state *S0*, or retransmit only in state *S1*. This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.

Three events are possible at the receiver: sending an acknowledgement (*A*), writing to the output process (*W*), and crashing (*C*). The three events can occur in six different orderings: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC*, and *WC*(*A*), where the parentheses are used to indicate that neither *A* nor *W* may follow *C* (i.e., once it has crashed, it has crashed). Figure 6-20 shows all eight combinations of sender and receiver strategy and the valid event sequences for each one. Notice that for each strategy there is some sequence of events that causes the protocol to behave incorrectly. For example, if the sender always retransmits, the *AWC* event will generate an undetected duplicate, even though the other two events work properly.

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Fig. 6-20. Different combinations of sender and receiver strategy.

Making the protocol more elaborate does not help. Even if the sender and receiver exchange several TPDUs before the receiver attempts to write, so that the sender knows exactly what is about to happen, the sender has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events, host crash/recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as recovery from a layer *N* crash can only be done by layer *N* + 1, and then only if the higher layer retains enough status information. As mentioned above, the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where it is.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote data base. Suppose that the remote transport entity is programmed to first pass TPDUs to the next layer up, and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user's machine does not necessarily mean that the remote host stayed up long enough to actually update the data base. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done, and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).

6.3. A SIMPLE TRANSPORT PROTOCOL ON TOP OF X.25

To make the ideas discussed so far more concrete, in this section we will study an example transport layer in detail. The example has been carefully chosen to be reasonably realistic, yet still simple enough to be easy to understand. The abstract service primitives are the OSI connection-oriented primitives, with the exception of the expedited data feature, which just adds complexity without providing any new insight into how the transport layer works.

6.3.1. The Example Service Primitives

Our first problem is how to express the OSI transport primitives in Pascal. *CONNECT.request* is easy: we will just have a library procedure *connect*, that can be called with the appropriate parameters necessary to establish a connection. However, *CONNECT.indication* is much harder. How do we signal the called transport user that there is an incoming call? In essence, an incoming call is an interrupt, a difficult concept to deal with in a high-level language and poor programming practice as well. The *CONNECT.indication* primitive is an excellent way of modeling how telephones work (telephones really do generate interrupts, by ringing), but is a not a good way of modeling how computers work.

To provide a reasonable interface to our transport layer, we will have to do what all real networks do, and invent a different, and much more computer-oriented, model for connection establishment. In our model, there are two procedures available, *listen* and *connect*. When a process (i.e., a transport user) wants to be able to accept incoming calls, it calls *listen*, specifying a particular TSAP to listen to. The process then blocks (i.e., goes to sleep) until some remote process attempts to establish a connection to its TSAP.

The other procedure, *connect*, can be used when a process wants to initiate the establishment of a connection. The caller specifies the local and remote TSAPs, and is blocked while the transport layer tries to set up the connection. If the connection succeeds, both parties are unblocked, and can start exchanging data.

Note that this model is highly asymmetric. One side is passive, executing a *listen* and waiting until something happens. The other side is active and initiates the connection. An interesting question arises of what to do if the active side begins first. One strategy is to have the connection attempt fail if there is no listener at the remote TSAP. Another strategy is to have the initiator block (possibly forever) until a listener appears.

A compromise, used in our example, is to hold the connection request at the receiving end for a certain time interval. If a process on that host calls *listen* before the timer goes off, the connection is established; otherwise, it is rejected and the caller is unblocked.

To release a connection, we will use a procedure *disconnect*. When both sides have disconnected, the connection is released.

Data transmission has precisely the same problem as connection establishment: although *T-DATA.request* can be implemented directly with a call to a library procedure, *T-DATA.indication* cannot be. We will use the same solution for data transmission as for connection establishment, an active call *send* that transmits data, and a passive call *receive* that blocks until a message has arrived.

Our concrete service definition thus consists of five primitives: *CONNECT*, *LISTEN*, *DISCONNECT*, *SEND*, and *RECEIVE*. Each primitive corresponds exactly with a library procedure that executes the primitive (unlike the OSI model, in which there is barely any correspondence at all between the primitives and the library procedures). The parameters for the service primitives and library procedures are as follows:

```
connum = CONNECT(local, remote)
connum = LISTEN(local)
status = DISCONNECT(connum)
status = SEND(connum, buffer, bytes)
status = RECEIVE(connum, buffer, bytes)
```

The *CONNECT* primitive takes two parameters, a local TSAP (i.e., transport address), *local*, and a remote TSAP, *remote*, and tries to establish a transport connection between the two. If it succeeds, it returns in *connum* a nonnegative number used to identify the connection on subsequent calls. If it fails, the reason for failure is put in *connum* as a negative number. In our simple model, each TSAP may participate in only one transport connection, so a possible reason for failure is that one of the transport addresses is currently in use. Some other reasons are: remote host down, illegal local address, and illegal remote address.

The *LISTEN* primitive announces the caller's willingness to accept connection requests directed at the indicated TSAP. The user of the primitive is blocked until an attempt is made to connect to it. There is no timeout.

The *DISCONNECT* primitive terminates a transport connection. The parameter *connum* tells which one. Possible errors are: *connum* belongs to another process, or *connum* is not a valid connection identifier. The error code, or 0 for success, is returned in *status*.

The *SEND* primitive transmits the contents of the buffer as a message on the indicated transport connection, possibly in several units if it is too big. Possible errors, returned in *status*, are: no connection, illegal buffer address, or negative count.

The *RECEIVE* primitive indicates the caller's desire to accept data. The size of the incoming message is placed in *bytes*. If the remote process has released the connection or the buffer address is illegal (e.g., outside the user's program), *status* is set accordingly to an error code.

6.3.2. The Example Transport Entity

Before looking at the code of the example transport entity, please be sure you realize that this example is analogous to the early examples presented in Chapter 4: it is more for pedagogical purposes than a serious proposal. Many of the technical details (such as extensive error checking) that would be needed in a production system have been omitted for the sake of simplicity. Nevertheless, most of the basic ideas found in the transport entity for the class 0 OSI protocol are present in our example.

The transport layer makes use of the network service primitives to send and receive TPDUs. Just as the OSI transport service primitives cannot be mapped directly onto library procedures, neither can the network service procedures. In this example we get around this problem by using X.25 as the network layer interface. Each TPDU will be carried in one packet and each packet will correspond to one TPDU. We will call these units "packets" below. In this example we will assume that X.25 is completely reliable (type A), neither losing packets nor resetting the circuit. Figure 6-21 gives an example program for implementing our transport service. Such a program (effectively the code of the transport entity) may be part of the host's operating system or it may be a package of library routines running within the user's address space. It may also be contained on a co-processor chip or network board plugged into the host's backplane. For simplicity, the example of Fig. 6-21 has been programmed as though it were a library package, but the changes needed to make it part of the operating system are minimal (primarily how user buffers are accessed).

It is worth noting, however, that in this example, the "transport entity" is not really a separate entity at all, but part of the user process. In particular, when the user executes a primitive that blocks, such as *LISTEN*, the entire transport entity blocks as well. While this design is fine for a host with only a single user process, on a host with multiple users, it would be more natural to have the transport entity be a separate process, distinct from all the user processes.

The interface to the network layer (X.25) is via the procedures *ToNet* and *FromNet* (not shown). Each has six parameters: the connection identifier, which maps one-to-one onto network virtual circuits; the X.25 *Q* and *M* bits, which indicate control message and more data from this message follows in the next packet, respectively; the packet type, chosen from the set *CALL REQUEST, CALL*

```
const MaxConn = ... ;  MaxMsg = ... ;  MaxPkt = ... ;
      TimeOut = ... ; cred = ... ;
      q0 = 0; q1 = 1; m0 = 0; m1 = 1; ok = 0;
      ErrFull = -1;  ErrReject = -2;  ErrClosed = -3;  LowErr = -3;

type bit = 0..1;
     TransportAddress = integer;
     ConnId = 0 .. MaxConn;          {connection identifier}
     PktType = (CallReq, CallAcc, ClearReq, ClearConf, DataPkt, credit);
     cstate = (idle, waiting, queued, established, sending, receiving, disconnecting);
     message = array [0 .. MaxMsg] of 0 .. 255;
     msgptr = ↑message;        {pointer to a message}
     ErrorCode = LowErr .. 0;
     ConnIdOrErr = LowErr .. MaxConn;
     PktLength = 0 .. MaxPkt;
     packet = array[PktLength] of 0 .. 255;

var ListenAddress: TransportAddress; {local address being listened to}
    ListenConn: ConnId;              {connection identifier for listen}
    data: packet;                    {scratch area for packet data}
    conn: array[ConnId] of record
      LocalAddress, RemoteAddress: TransportAddress;
      state: cstate;            {state of this connection}
      UserBufferAddress: msgptr;     {pointer to receive buffer}
      ByteCount: 0 .. MaxMsg;        {send/receive count}
      ClrReqReceived: boolean;       {set when CLEAR REQUEST packet received}
      timer: integer;                {used to time out CALL REQUEST packets}
      credits: integer               {number of messages that may be sent}
    end;

function listen(t: TransportAddress): ConnIdOrErr;
{User wants to listen for a connection.  See if CALLREQ has already arrived.}
var i: integer; found: boolean;
begin
  i := 1;
  found := false;

  while (i <= MaxConn) and not found do
    if (conn[i].state = queued) and (conn[i].LocalAddress = t)
      then found := true
      else i := i + 1;

    if not found then
      begin    {no CALLREQ is waiting.  Go to sleep until arrival or timeout.}
        ListenAddress := t;  sleep;  i := ListenConn
      end;
    conn[i].state := established;    {connection is established}
    conn[i].timer := 0;              {timer is not used}
    listen := i;                     {return connection identifier}
    ListenConn := 0;                 {0 is assumed to be an invalid address}
    ToNet(i, q0, m0, CallAcc, data, 0)      {tell net to accept connection}
end; {listen}
```

Fig. 6-21. A sample transport entity.

```
function connect(l, r: TransportAddress): ConnIdOrErr;
{User wants to connect to a remote process.  Send CALLREQ packet.}
var i: integer;
begin i := MaxConn;             {search table backwards}
  data[0] := r;   data[1] := l;         {CALL REQUEST packet needs these}
  while (conn[i].state <> idle) and (i > 1) do i := i - 1;
  if conn[i].state = idle then
    with conn[i] do
      begin                     {make a table entry that CALLREQ has been sent}
        LocalAddress := l;   RemoteAddress := r;   state := waiting;
        ClrReqReceived := false;   credits := 0;   timer := 0;
        ToNet(i, q0, m0, CallReq, data, 2);
        sleep;                  {wait for CALLACC or CLEARREQ}
        if state = established then connect := i;
        if ClrReqReceived then
          begin                 {other side refused call}
            connect := ErrReject;
            state := idle;      {back to idle state}
            ToNet(i, q0, m0, ClearConf, data, 0)
          end
      end
  else connect := ErrFull       {reject CONNECT: no table space}
end; {connect}

function send(cid: ConnId; bufptr: msgptr; bytes: integer): ErrorCode;
{User wants to send a message.}
var i, count: integer;   m: bit;
begin
  with conn[cid] do
    begin                       {enter sending state}
      state := sending;
      ByteCount := 0;
      if (not ClrReqReceived) and (credits = 0) then sleep;
      if not ClrReqReceived then
        begin {credit available; split message into packets}
          repeat
            if bytes - ByteCount > MaxPkt
              then begin count := MaxPkt;  m := 1 end
              else begin count := bytes - ByteCount;  m := 0 end;
            for i := 0 to count - 1 do data[i] := bufptr↑[ByteCount + i];
            ToNet(cid, q0, m, DataPkt, data, count);
            ByteCount := ByteCount + count;
          until ByteCount = bytes;   {loop until whole message sent}
          credits := credits - 1;    {one credit used up}
          send := ok
        end
      else send := ErrClosed;        {SEND failed: peer wants to disconnect}
      state := established
    end
end; {send}
```

```
function receive(cid: ConnId; bufptr: msgptr; var bytes: integer): ErrorCode;
{User is prepared to receive a message.}
begin
  with conn[cid] do
    begin
      if not ClrReqReceived then
        begin                      {connection still established; try to receive}
          state := receiving;
          UserBufferAddress := bufptr;   ByteCount := 0;
          data[0] := cred;   data[1] := 1;
          ToNet(cid, q1, m0, credit, data, 2);       {send credit}
          sleep;                   {block awaiting data}
          bytes := ByteCount
        end;
      if ClrReqReceived then receive := ErrClosed else receive := ok;
      state := established
    end
end; {receive}

function disconnect(cid: ConnId): ErrorCode;
{User wants to release a connection.}
begin
  with conn[cid] do
    if ClrReqReceived
      then begin state := idle;  ToNet(cid, q0, m0, ClearConf, data, 0) end
      else begin state := disconnecting;  ToNet(cid, q0, m0, ClearReq, data, 0) end;
  disconnect := ok
end; {disconnect}

procedure PacketArrival;
{A packet has arrived, get and process it.}
var cid: ConnId;                   {connection on which packet arrived}
    q, m: bit;
    ptype: PktType;                {CallReq, CallAcc, ClearReq, ClearConf, DataPkt, credit}
    data: packet;                  {data portion of the incoming packet}
    count: PktLength;              {number of data bytes in packet}
    i: integer;                    {scratch variable}

begin
  FromNet(cid, q, m, ptype, data, count);  {go get it}
  with conn[cid] do
  case ptype of
  CallReq:                         {remote user wants to establish connection}
    begin
      LocalAddress := data[0];  RemoteAddress := data[1];
      if LocalAddress = ListenAddress
        then begin ListenConn := cid;  state := established;  wakeup end
        else begin state := queued;  timer := TimeOut end;
      ClrReqReceived := false;   credits := 0
    end;
  CallAcc:                  {remote user has accepted our CALL REQUEST}
    begin
      state := established;
      wakeup
    end;

  ClearReq:                 {remote user wants to disconnect or reject call}
    begin
      ClrReqReceived := true;
      if state = disconnecting then state := idle;    {clear collision}
      if state in [waiting, receiving, sending] then wakeup
    end;

  ClearConf:                {remote user agrees to disconnect}
    state := idle;

  credit:                   {remote user is waiting for data}
    begin
      credits := credits + data[1];
      if state = sending then wakeup
    end;

  Datapkt:                  {remote user has sent data}
    begin
      for i := 0 to count − 1 do UserBufferAddress↑[ByteCount + i] := data[i];
      ByteCount := ByteCount + count;
      if m = 0 then wakeup
    end
  end;
end; {PacketArrival}

procedure clock;
{The clock has ticked, check for timeouts of queued connect requests.}
var i: ConnId;
begin
  for i := 1 to MaxConn do
    with conn[i] do
      if timer > 0 then
        begin {timer was running}
          timer := timer − 1;
          if timer = 0 then
            begin    {timer has expired}
              state := idle;
              ToNet(i, q0, m0, ClearReq, data, 0)
            end
        end
end; {clock}
```

ACCEPTED, *CLEAR REQUEST*, *CLEAR CONFIRMATION*, *DATA*, and *CREDIT*; a pointer to the data itself; and the number of bytes of data.

On calls to *ToNet*, the transport entity (i.e., some procedure in Fig. 6-21) fills in all the parameters for the network layer to read; on calls to *FromNet*, the network layer dismembers an incoming packet for the transport entity. By passing information as procedure parameters rather than passing the actual outgoing or incoming packet itself, the transport layer is shielded from the details of the network layer protocol. If the transport entity should attempt to send a packet when the underlying virtual circuit's sliding window is full, it is suspended within *ToNet* until there is room in the window. This mechanism is transparent to the transport entity and is controlled by the network layer using commands like *EnableTransportLayer* and *DisableTransportLayer* analogous to those described in the protocols of Chapter 4. The management of the X.25 packet layer window is also done by the network layer.

In addition to this transparent suspension mechanism, there are also explicit *sleep* and *wakeup* procedures (not shown) called by the transport entity. The procedure *sleep* is called when the transport entity is logically blocked waiting for an external event to happen, generally the arrival of a packet. After *sleep* has been called, the transport entity (and user process, of course) stop executing.

Each connection maintained by the transport entity of Fig. 6-21 is always in one of seven states, as follows:

1. Idle—Connection not established yet.
2. Waiting—*CONNECT* has been executed and *CALL REQUEST* sent.
3. Queued—A *CALL REQUEST* has arrived; *LISTEN* has not been done.
4. Established—The connection has been established.
5. Sending—The user is waiting for permission to transmit a packet.
6. Receiving—A *RECEIVE* has been done.
7. Disconnecting—A *DISCONNECT* has been done locally.

Transitions between states can occur when primitives are executed, when packets arrive, or when the timer expires.

The collection of procedures shown in Fig. 6-21 are of two types. Most are directly callable by user programs. *PacketArrival* and *clock* are different, however. They are spontaneously triggered by external events: the arrival of a packet and the clock ticking, respectively. In effect, they are interrupt routines. We will assume that they are never invoked while a transport entity procedure is running. Only when the user process is sleeping or executing outside the transport entity may they be called. This property is crucial to the correct functioning of the transport entity.

The existence of the *Q* (Qualifier) bit in the X.25 header allows us to avoid the overhead of a transport protocol header. Ordinary data messages are sent as X.25 data packets with $Q = 0$. Transport protocol control messages, of which there is only one *(CREDIT)* in our example, are sent as X.25 data packets with $Q = 1$. These control messages are detected and processed by the receiving transport entity, of course.

The main data structure used by the transport entity is the array *conn*, which has one record for each potential connection. The record maintains the state of the connection, including the transport addresses at either end, the number of messages sent and received on the connection, the current state, the user buffer pointer, the number of bytes of the current messages sent or received so far, a bit indicating that the remote user has issued a *DISCONNECT*, a timer, and a permission counter used to enable sending of messages. Not all of these fields are used in our simple example, but a complete transport entity would need all of them, and perhaps more. Each *conn* entry is assumed initialized to the *idle* state.

When the user calls *CONNECT*, the network layer is instructed to send a *CALL REQUEST* packet to the remote machine, and the user is put to sleep. When the *CALL REQUEST* packet arrives at the other side, the transport entity is interrupted to run *PacketArrival* to check if the local user is listening on the specified address. If so, a *CALL ACCEPTED* packet is sent back and the remote user is awakened; if not, the *CALL REQUEST* is queued for *TimeOut* clock ticks. If a *LISTEN* is done within this period, the connection is established; otherwise, it times out and is rejected with a *CLEAR REQUEST* packet. This mechanism is needed to prevent the initiator from blocking forever in the event that the remote process does not want to connect to it.

Although we have eliminated the transport protocol header, we still need a way to keep track of which packet belongs to which transport connection, since multiple connections may exist simultaneously. The simplest approach is to use the X.25 virtual circuit number as the transport connection number as well. Furthermore, the virtual circuit number can also be used as the index into the *conn* array. When a packet comes in on X.25 virtual circuit k, it belongs to transport connection k, whose state is in the record *conn*[k]. For connections initiated at a host, the connection number is chosen by the originating transport entity. For incoming calls, the X.25 network makes the choice, choosing any unused virtual circuit number.

To avoid having to provide and manage buffers within the transport entity, a flow control mechanism different from the traditional sliding window is used here. Instead, when a user calls *RECEIVE*, a special **credit message** is sent to the transport entity on the sending machine and is recorded in the *conn* array. When *SEND* is called, the transport entity checks to see if a credit has arrived on the specified connection. If so, the message is sent (in multiple packets if need be) and the credit decremented; if not, the transport entity puts itself to sleep until a credit arrives. This mechanism guarantees that no message is ever sent unless the other side has already done a *RECEIVE*. As a result, whenever a message arrives there is guaranteed to be a buffer available into which it can be put. The scheme can easily be generalized to allow receivers to provide multiple buffers and request multiple messages.

You should keep the simplicity of Fig. 6-21 in mind. A realistic transport entity would normally check all user supplied parameters for validity, handle recovery from network resets, deal with call collisions, and support a more general transport service including such facilities as interrupts, datagrams, and nonblocking versions of the *SEND* and *RECEIVE* primitives.

6.3.3. The Example as a Finite State Machine

Writing a transport entity is difficult and exacting work, especially for the higher transport protocol classes. To reduce the chance of making an error, it is often useful to represent the state of the protocol as a finite state machine.

We have already seen that our example protocol has seven states per connection. It is also possible to isolate 12 events that can happen to move a connection from one state to another. Five of these events are the five service primitives. Another six are the arrivals of the six legal packet types. The last one is the expiration of the timer. Figure 6-22 shows the main protocol actions in matrix form. The columns are the states and the rows are the 12 events.

Each entry in the matrix (i.e., the finite state machine) of Fig. 6-22 has up to three fields: a predicate, an action, and a new state. The predicate indicates under what conditions the action is taken. For example, in the upper left-hand entry, if a *LISTEN* is executed and there is no more table space (predicate *P1*), the *LISTEN* fails and the state does not change. On the other hand, if a *CALL REQUEST* packet has already arrived for the transport address being listened to (predicate *P2*), the connection is established immediately. Another possibility is that *P2* is false, that is, no *CALL REQUEST* has come in, in which case the connection remains in the *Idle* state, awaiting a *CALL REQUEST* packet.

It is worth pointing out that the choice of states to use in the matrix is not entirely fixed by the protocol itself. In this example, there is no state *listening*, which might have been a reasonable thing to have following a *LISTEN*. There is no *listening* state because a state is associated with a connection record entry, and no connection record is created by *LISTEN*. Why not? Because we have decided to use the X.25 virtual circuit numbers as the connection identifiers, and for a *LISTEN*, the virtual circuit number is ultimately chosen by the X.25 network when the *CALL REQUEST* packet arrives.

The actions *A1* through *A12* are the major actions, such as sending packets and starting timers. Not all the minor actions, such as initializing the fields of a connection record are listed. If an action involves waking up a sleeping process, 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*

Status

		Idle	Waiting	Queued	Established	Sending	Receiving	Disconnecting
Primitives	LISTEN	P1: ~/Idle P2: A1/Estab $\overline{\text{P2}}$: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle $\overline{\text{P1}}$: A3/Wait						
	DISCONNECT				P4: A5/Idle $\overline{\text{P4}}$: A6/Disc			
	SEND				P5: A7/Estab $\overline{\text{P5}}$: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	CallReq	P3:A1/Estab $\overline{\text{P3}}$:A4/Queu'd						
	CallAcc		~/Estab					
	ClearReq		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	ClearConf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			~/Idle				

Predicates

P1: Connection table full
P2: CallReq pending
P3: LISTEN pending
P4: ClearReq pending
P5: Credit available

Actions

A1: Send CallAcc
A2: Wait for CallReq
A3: Send CallReq
A4: Start timer
A5: Send ClearConf
A6: Send ClearReq
A7: Send message
A8: Wait for credit
A9: Send credit
A10: Set ClrReqReceived flag
A11: Record credit
A12: Accept message

Fig. 6-22. The example protocol as a finite state machine. Each entry has an optional predicate, and optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicates the negation of the predicate. Blank entries correspond to impossible or invalid events.

REQUEST. After each action is performed, the connection may move to a new state, as shown in Fig. 6-22.

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. 6-22 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 that should be checked for.

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, including the OSI connection-oriented transport protocol standard (ISO 8073), the protocols are given as finite state machines of the type of Fig. 6-22. 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. 6-23.

6.4. EXAMPLES OF THE TRANSPORT LAYER

In this section we will once again examine our running examples to see what their transport layers are like. As usual, we will emphasize the protocol aspects in these examples.

6.4.1. The Transport Layer in Public Networks

Nearly all public networks use the connection-oriented OSI transport service (ISO 8072) and OSI transport protocols (ISO 8073). We have already looked at the OSI transport service; let us now look at the protocols.

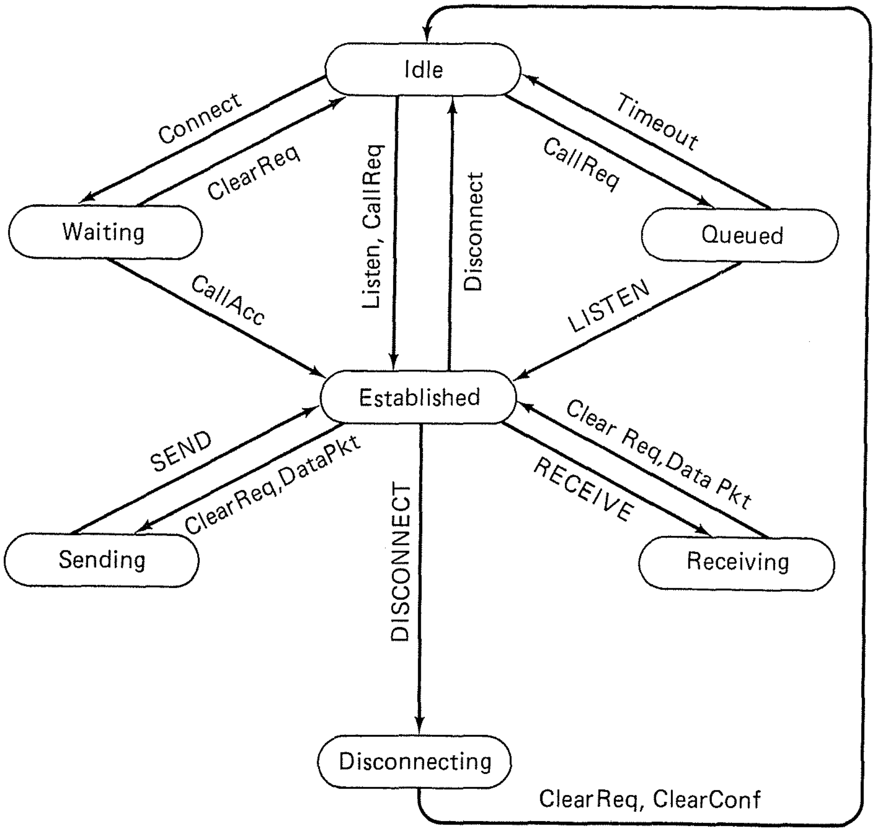


Fig. 6-23. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

As mentioned earlier, the OSI transport protocol has five variations, Classes 0 through 4. Each variation is intended for a specific type of network reliability, ranging to perfect to completely unreliable. We will briefly review the five protocol classes below. Class 0 (Simple class) is intended for use with type A (perfect) networks. It is primarily concerned with connection establishment and release, data transfer, and breaking large messages up into smaller TPDUs, if necessary. It uses the underlying network connection to do all the rest of the work (flow control, error control, and so on).

Class 1 (Basic error recovery class) differs from Class 0 in its ability to recover from *N-RESET*s generated by the network layer. After a network layer connection is broken, the transport entities establish a new network layer connection and continue from where they left off. In order to accomplish this recovery, the transport entities number the TPDUs, a feature not present in Class 0. Class 0 just gives up if the network connection breaks.

Class 2 (Multiplexing class) is the same as Class 0, except that it also supports multiplexing of multiple transport connections onto one network connection. Also, it permits explicit flow control using a credit scheme analogous to the one used in the example of Sec. 6-3. Finally, Class 2 also permits the use of expedited data.

Class 3 (Error recovery and multiplexing class) includes the features of Classes

1 and 2, namely, it can recover from network layer failures and it can also support the multiplexing of multiple transport connections onto one network connection.

Class 4 (Error detection and recovery class) is the most sophisticated and most interesting class. It has been designed to deal with unreliable network service, for example, datagram subnets that can lose, store, and duplicate packets. It is this class that we will primarily examine below.

The OSI transport protocol has 10 different TPDU types. Each TPDU has up to four parts:

1. A 1-byte field giving the length of the fixed plus variable headers.
2. A fixed part of the header (length depends on the TPDU type).
3. A variable part of the header (length depends on the parameters).
4. A user data.

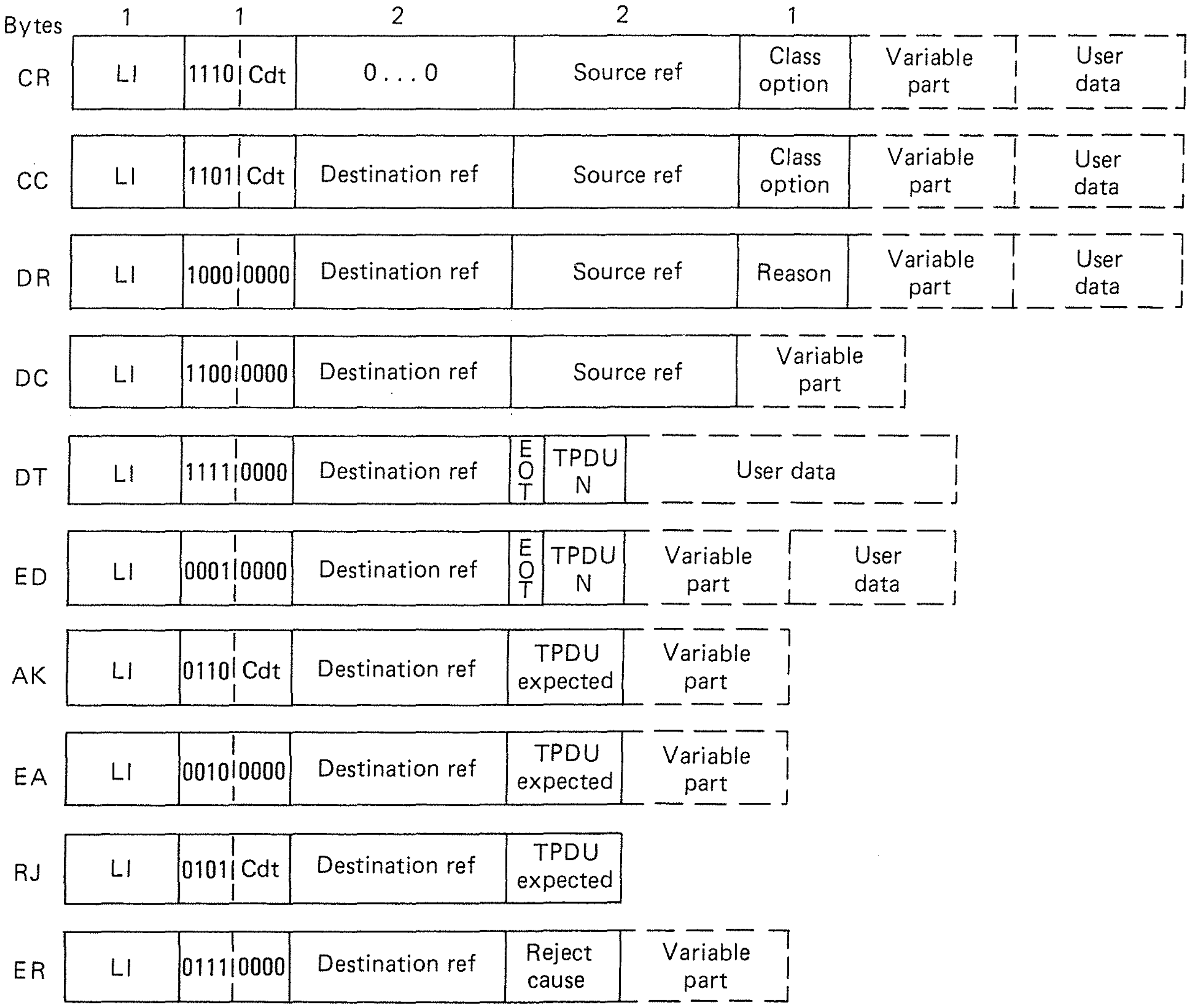
The first byte of each TPDU is the *LI* (*Length Indicator*) field. It gives the total header length (fixed plus variable parts) in bytes, excluding the *LI* field itself, up to a maximum of 254 bytes. Code 255 is reserved for future use.

Next comes the fixed part of the header. These fields are important enough that they are included in each TPDU of the relevant type. The fields are TPDU dependent. In other words, the fixed part of all *CONNECTION REQUEST* TPDUs have the same fields, but these are different from the fields in a *DATA* TPDU.

Following the fixed part of the header is the variable part. This part of the header is used for options that are not always needed. The recipient of a TPDU can tell how many bytes of variable-part header are present by looking at the *LI* field and subtracting off the length of the fixed-part header for the TPDU type. The variable part is divided into fields, each starting with a 1-byte type field, then a 1-byte length field, followed by the data itself. For example, when setting up a transport connection, the initiating transport entity can use the variable part to propose a nonstandard maximum TPDU size.

Following the header come the user data. The *DATA* TPDU obviously contains user data, but some of the other TPDUs also contain a limited amount too. The formats of the 10 TPDU types are shown in Fig. 6-24. Some of these have minor variants that are not shown.

The *CONNECTION REQUEST, CONNECTION CONFIRM, DISCONNECT REQUEST,* and *DISCONNECT CONFIRM* TPDUs are completely analogous to the *CALL REQUEST, CALL ACCEPTED, CLEAR REQUEST,* and *CLEAR CONFIRM* packets used in X.25. To establish a connection, the initiating transport entity sends a *CONNECTION REQUEST* TPDU and the peer replies with *CONNECTION CONFIRM*. Similarly, to release a connection, either one of the transport entities sends a *DISCONNECT REQUEST* and the peer replies with *DISCONNECT CONFIRM*. The *DATA* and *EXPEDITED DATA* TPDUs are used for regular and expedited data,



CR: Connection request
CC: Connection confirm
DR: Disconnect request
DC: Disconnect confirm
DT: Data

ED: Expedited data
AK: Data acknowledgement
EA: Expedited data acknowledgement
RJ: Reject
ER: Error

Fig. 6-24. The OSI transport protocol TPDUs.

respectively. These two types are acknowledged by the *DATA ACKNOWLEDGEMENT* and *EXPEDITED DATA ACKNOWLEDGEMENT* TPDUs, respectively. Finally, the *REJECT* and *ERROR* TPDUs are used for error handling.

Let us now look at the various TPDU types one at a time in more detail. *CONNECTION REQUEST* is used to establish a connection. Like all TPDUs, it contains a 1-byte *LI* field giving the total header length (excluding the *LI* field itself).

Next comes a byte containing a 4-bit TPDU type and the *cdt* (credit) field. The Class 4 protocol uses a credit scheme for flow control, rather than a sliding window scheme, and this field tells the remote transport entity how many TPDU's it may initially send.

The *Destination reference* and *Source reference* fields identify transport connections. They are needed because in Classes 2, 3, and 4 it is possible to multiplex several transport connections over one network connection. When a packet comes in from the network layer, the transport entities use these fields to determine which transport connection the TPDU in the packet belongs to. The *CONNECTION REQUEST* TPDU provides an identifier in the *Source reference* field that will be used by the initiator. The *CONNECTION CONFIRM* TPDU adds to that identifier the *Destination reference*, which is used by the destination for connection identification.

The *Class option* field is used by the transport entities for negotiating the protocol class to be used. The initiator makes a proposal, which the responder can either accept or reject. The responder can also make a counterproposal, suggesting a lower, but not a higher, protocol class. The field also contains two bits that are used to enable 4-byte TPDU sequence numbers instead of the standard 1-byte numbers, and enable or disable explicit flow control in Class 2.

The *Variable part* of the *CONNECTION REQUEST* TPDU may contain any of the following options:

1. TSAP to be connected to at the remote host.
2. TSAP being connected to at the local host.
3. Proposed maximum TPDU size (128 to 8192 bytes, in powers of 2).
4. Version number.
5. Protection parameter (e.g., an encryption key).
6. Checksum.
7. Some option bits (e.g., use of expedited data, use of checksum).
8. Alternative protocol classes that are acceptable to the initiator.
9. Maximum delay before acknowledging a TPDU, in milliseconds.
10. Throughput expected (average desired and minimum acceptable).
11. Residual error rate (average desired and maximum acceptable).
12. Priority (0 to 65535, with 0 being the highest priority).
13. Transit delay (average and maximum acceptable, in milliseconds).
14. How long to keep trying to recover after an *N-RESET*.

Some of the parameters, such as the alternative protocol classes, are intended for the remote peer. However, others, such as the quality of service parameters (throughput, residual error rate, etc.), are aimed at the network layer. If the network layer is unable to provide at least the minimum service required, then the network layer itself rejects the connection with a *DISCONNECT REQUEST*, rather than putting it through.

The *User data* field may contain up to 32 bytes of data in Classes 1 through 4. It is not permitted in Class 0. This field may be used for any purpose the users wish (e.g., a password for remote login).

When the *CONNECTION REQUEST* TPDU arrives at the remote host, the transport entity there causes a *CONNECT.indication* primitive to the transport user. If the user decides to accept the incoming call, the remote transport layer replies to the initiator with a *CONNECTION CONFIRM* TPDU. The format and options in the *CONNECTION CONFIRM* TPDU are the same as those in the *CONNECTION REQUEST* TPDU.

In type A and B networks, this establishment procedure is sufficient, but in type C networks it is not, due to the possibility of delayed duplicate packets. To eliminate the possibility of old packets interfering with a connection establishment, a three-way handshake is used, with the *CONNECTION CONFIRM* TPDU itself being acknowledged with an *ACK* TPDU. Furthermore, after a connection is released, the *Source reference* and *Destination reference* are considered **frozen references** and not reused for an interval long enough to guarantee that all old duplicates have died out. Unlike Tomlinson's clock scheme, the OSI transport protocol does not describe how to deal with crashes. The safest way is just to wait for the maximum packet lifetime before rebooting.

To release a connection, a transport entity sends a *DISCONNECT REQUEST* TPDU to its peer. The format of the fixed part of the header is the same as for the *CONNECTION REQUEST* TPDU, except that the *Class option* field is replaced with the *Reason* field telling why the connection is being released. Among other possibilities are: the transport user executed a *DISCONNECT.request* primitive, there was a bad parameter in a TPDU, the TSAP to be connected to does not exist, or the network is congested. The *Variable part* of the header and the *User data* field (up to 64 bytes) can provide additional explanations.

The required response to a *DISCONNECT REQUEST* TPDU is a *DISCONNECT CONFIRM* TPDU. The only field that may be present in the *Variable part* is the checksum, if checksums are being used on the connection.

The formats of the *DATA* TPDU and *EXPEDITED DATA* TPDU shown in Fig. 6-24 are the normal types for Class 4. An additional format with a 4-byte *TPDU Nr* is also permitted to make sure that TPDU sequence numbers will not wrap around for a very long time. The *EOT* flag is set to 1 to indicate End Of Transport message. This flag is needed so the remote transport entity knows when to stop reassembling TPDUs and pass the resulting message to the remote transport user. The only *Variable part* field is the checksum, when it is in use.

The *DATA ACKNOWLEDGEMENT* and *EXPEDITED DATA ACKNOWLEDGEMENT* TPDUs acknowledge the receipt of TPDUs up to, but not including, the one whose sequence number is given in the acknowledgement. Thus that one is the one expected next. The *Variable part* of both contain the checksum, if checksums are being used.

In addition, the *DATA ACKNOWLEDGEMENT* TPDU also contains information about the flow control window. In particular, the receiver can explicitly specify the lower edge and size of the window of sequence numbers that the sender is permitted to send. It is explicitly permitted for the receiver to reduce the window size (i.e., take back credits). For example, suppose the receiver has acknowledged TPDU 3 and given a window size of 8, thus permitting TPDUs 4 through 11. Shortly thereafter, the receiver notices that it is short on buffer space and decides to reduce the window to 3 by sending a new acknowledgement for 3 with a window of 2. This action permits only TPDUs 4 and 5 and forbids 6 through 11, even though these were previously permitted.

With a type C network, a problem arises if the two acknowledgements are delivered in the wrong order. The sender will wrongfully conclude that it may send TPDUs up to 11. To eliminate this problem, the protocol allows a *subsequence* field in the *Variable part*. Normally this field is not used, but when a second acknowledgement is sent for the same TPDU, a *subsequence* field is included with the value 1. A third acknowledgement carries a *subsequence* value of 2, and so on. This method allows the sender to deduce the order in which the acknowledgements were sent, and only accept the last one as valid.

The *REJECT* TPDU is only used in Classes 1 and 3, and is primarily used when resynchronizing after an *N-RESET*. Its function is to signal a problem and invite the peer to retransmit all TPDUs starting at the sequence number indicated. The credit value is also reset.

The final TPDU type is used to report protocol errors. The *Reject cause* field tells what was wrong. Possibilities include: invalid parameter code in the *Variable part*, invalid TPDU type, and invalid TPDU value.

As we mentioned earlier, the OSI transport layer defines a connectionless service and protocol, in addition to the connection-oriented service and protocol we have been describing at length. The purpose of the OSI connectionless transport service is to allow its users to send messages with the *T-UNITDATA* primitive without the overhead of first establishing and later releasing a connection.

The OSI connectionless transport service can work using either connection-oriented network service or connectionless network service. To a large extent, putting connectionless transport service on top of connection-oriented network service defeats the purpose of having connectionless transport service, but in some cases the only network service available is connection-oriented (e.g., a public X.25 network), so that case has been provided for.

When operating the connectionless transport service on top of a connectionless network service, each TPDU goes into a single packet. These TPDUs are not acknowledged and are not guaranteed to be reliably delivered. No promise is given about the order in which TPDUs are delivered.

The OSI connectionless service uses a connectionless protocol. Only one TPDU format is used, as shown in Fig. 6-25. This TPDU is similar to those used in the connection-oriented service. The *Variable part* contains the source and destination TSAP addresses, and optionally a checksum.

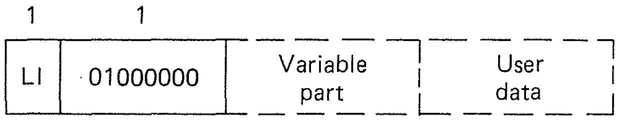


Fig. 6-25. The OSI connectionless TPDU format.

Work on the implementation of OSI and related transport protocols is discussed in (Chong, 1986; and Watson and Mamrak, 1987).

6.4.2. The Transport Layer in the ARPANET (TCP)

In the original ARPANET design, the subnet was assumed to offer virtual circuit service (i.e., be perfectly reliable). The first transport layer protocol, **NCP (Network Control Protocol)**, was designed with a perfect subnet in mind. It just passed TPDUs (called messages) to the network layer and assumed that they would all be delivered in order to the destination. Experience showed that the ARPANET was indeed reliable enough for this protocol to be completely satisfactory for traffic within the ARPANET itself.

However, as time went on, and the ARPANET grew into the ARPA Internet, which included many LANs, a packet radio subnet, and several satellite channels, the end-to-end reliability of the subnet decreased. This development forced a major change in the transport layer and led to the gradual introduction of a new transport layer protocol, **TCP (Transmission Control Protocol)**, which was specifically designed to tolerate an unreliable subnet (type C in OSI terms). Associated with TCP was a new network layer protocol, **IP**, which we studied in the previous chapter. Today TCP/IP is not only used in the ARPANET and ARPA Internet, but in many commercial systems as well.

A TCP transport entity accepts arbitrarily long messages from user processes, breaks them up into pieces not exceeding 64K bytes, and sends each piece as a separate datagram. The network 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.

Every byte of data transmitted by TCP has its own private sequence number. The sequence number space is 32 bits wide to make sure that old duplicates have long since vanished by the time the sequence numbers have wrapped around. TCP does, however, explicitly deal with the problem of delayed duplicates when attempting to establish a connection, using the three-way handshake for this purpose.

Figure 6-26 shows the header used by TCP. The first thing that strikes one is that the minimum TCP header is 20 bytes. Unlike OSI Class 4, with which it is roughly comparable, TCP has only one TPDU header format. Let us dissect this large header field by field. The *Source port* and *Destination port* fields identify the end points of the connection (the TSAP addresses in OSI terminology). Each host may decide for itself how to allocate its ports.

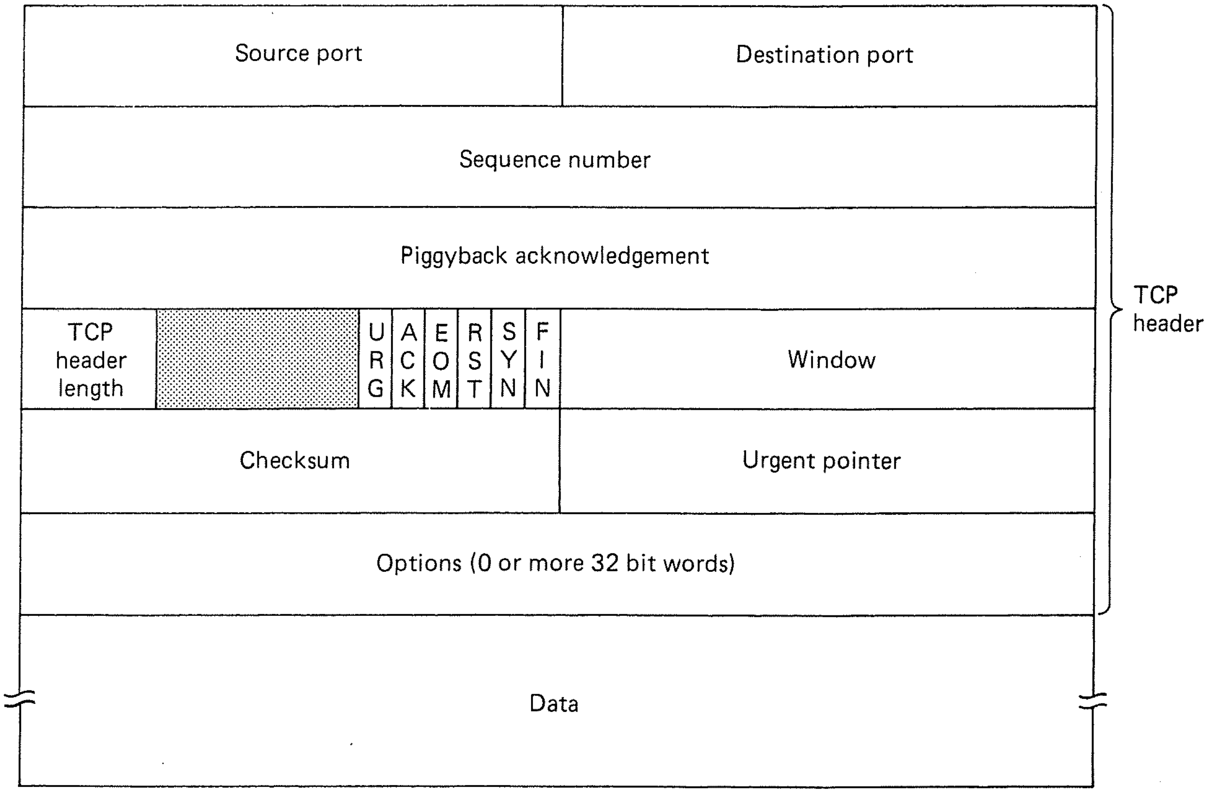


Fig. 6-26. The TCP TPDU structure.

The *Sequence number* and *Piggyback acknowledgement* fields perform their usual functions. They are 32 bits long because every byte of data is numbered in TCP.

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 variable length, so the header is too.

Next 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. 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 CONFIRM*, 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. After closing a connection, a process may continue to receive data indefinitely. The *RST* bit is used to reset a connection that has become confused due to delayed duplicate *SYN*s or host crashes. The *EOM* bit indicates End Of Message.

Flow control in TCP is handled using a variable-size sliding window. A 16-bit field is needed, because *Window* tells how many bytes may be sent beyond the byte acknowledged, not how many TPDUs.

A *Checksum* is also provided for extreme reliability. The checksum algorithm is simply to add up all the data, regarded as 16-bit words, and then to take the 1's complement of the sum.

The *Options* field is used for miscellaneous things, for example to communicate buffer sizes during the setup procedure.

Unlike most of the non-OSI protocols, TCP has a well-defined service interface. Calls are provided to actively and passively initiate connections (effectively, the *CONNECT* and *LISTEN* primitives from our example transport protocol, except that the *LISTEN* has two variants, one to accept connection requests from any caller and one to accept connection requests only from some specified caller). Calls are also provided to send and receive data, gracefully and abruptly terminate connections, and ask for the status of a connection.

Like OSI, the ARPANET also provides a connectionless transport protocol, **UDP (User Datagram Protocol)**. Like its OSI counterpart, it allows users to send messages without connection establishment and without any guarantee of delivery or sequencing. In effect, UDP is simply a user interface to IP. Its format is shown in Fig. 6-27.

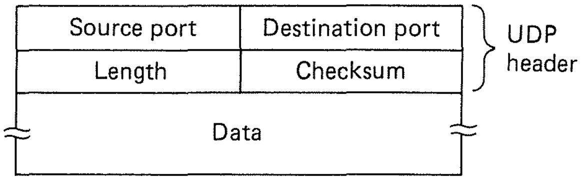


Fig. 6-27. The ARPANET UDP format.

Comparison of OSI Class 4 and TCP

The OSI Class 4 transport protocol (often called **TP4**), and TCP have numerous similarities but also some differences. In this section we will examine these similarities and differences (Groenbaek, 1986). Let us start with the points on which the two protocols are the same. Both protocols are designed for providing a reliable, connection-oriented, end-to-end transport service on top of an unreliable network that can lose, garble, store, and duplicate packets. Both must deal with the worst case problems, such as a subnet that can store a valid sequence of packets and then "play them back" later.

The two protocols are also alike in that both have a connection establishment phase, a data transfer phase, and then a connection release phase. The general concepts of establishing, using, and releasing connections are also similar, although some of the details differ. In particular, both TP4 and TCP use three-way handshakes to eliminate potential difficulties caused by old packets suddenly emerging and causing trouble.

However, the two protocols also have some notable differences, which are listed in Fig. 6-28. First, TP4 uses nine different TPDU types, whereas TCP only has one. This difference results in TCP being simpler, but it also needs a larger header, because all fields must be present in all TPDUs. The minimum size of the TCP header is 20 bytes; the minimum size of the TP4 header is 5 bytes. Both protocols allow optional fields that can increase the header size above the minimum.

Feature	OSI TP4	TCP
Number of TPDU types	9	1
Connection collision	2 connections	1 connection
Addressing format	Not defined	32 bits
Quality of service	Open ended	Specific options
User data in CR	Permitted	Not permitted
Stream	Messages	Bytes
Important data	Expedited	Urgent
Piggybacking	No	Yes
Explicit flow control	Sometimes	Always
Subsequence numbers	Permitted	Not permitted
Release	Abrupt	Graceful

Fig. 6-28. Differences between the OSI TP4 protocol and TCP.

A second difference is what happens if two processes simultaneously attempt to set up connections between the same two TSAPs (i.e., a connection collision). With TP4, two independent, full-duplex connections are established. With TCP, a connection is identified by a pair of TSAPs, so only one connection is established.

A third difference is in the addressing format used. TP4 does not specify the exact format of a TSAP address; TCP uses 32-bit numbers as TSAPs.

The issue of quality of service is also handled differently in the two protocols and forms the fourth difference. TP4 has a rather elaborate and open-ended mechanism for a three-way negotiation of the quality of service. This negotiation involves the calling process, the called process, and the transport service itself. Many parameters can be specified, and both target and minimum acceptable values can be given. TCP, in contrast, does not have a quality of service field at all, but the underlying IP service has an 8-bit field that allows a choice to be made out of a limited number of combinations of speed and reliability.

A fifth difference is that TP4 allows user data to be carried in the *CR* TPDU, but TCP does not allow user data in the initial TPDU. The initial data (e.g., a password) might be necessary to decide whether or not a connection should be established. With TCP, making establishment depend on user data is not possible.

The previous four differences relate to the connection establishment phase. The next five concern the data transfer phase. A very basic difference is the model of data transport. The TP4 model is that of an ordered series of messages (TSDUs in OSI terminology). The TCP model is that of a continuous stream of bytes, without any explicit message boundaries. In practice, however, the TCP model is not really a pure byte stream because a library procedure, *push*, can be called to flush out any data buffered but not yet sent. When the remote user does a read, data from before and after the *push* will not be combined, so in a sense *push* can be thought of as defining a sort of message boundary.

The seventh difference concerns how important data that need special processing (such as BREAK characters) are dealt with. TP4 has two independent message streams, regular data and expedited data, multiplexed together. Only one expedited message may be outstanding at any instant. TCP uses the *Urgent* field to indicate that some number of bytes within the current TPDU are special and should be processed out of order.

The eighth difference is the absence of piggybacking in TP4 and its presence in TCP. This difference is not quite as significant as it may first appear since it is possible for a transport entity to put two TPDUs, for example, DT and AK in a single network packet.

The ninth difference is the way flow control is handled. TP4 can use a credit scheme, but it can also rely on the window scheme of the network layer to regulate the flow. TCP always uses an explicit flow control mechanism with the window size specified in each TPDU.

The tenth difference relates to this window scheme. In both protocols the receiver is entitled to reduce the window at will. This possibility potentially gives rise to problems if the grant of a large window and its subsequent retraction arrive in the wrong order. In TCP there is no solution to this problem. In TP4 it is solved by the subsequence number that is included in the retraction, thus allowing the sender to determine that the small window followed, rather then preceded, the large one.

Finally, our eleventh and last difference between the two protocols is in the way connections are released. TP4 uses an abrupt disconnection in which a series of data TPDUs may be followed directly by a *DR* TPDU. If the data TPDUs are lost, they will not be recovered by the protocol and information will be lost. TCP uses a three-way handshake to avoid data loss upon disconnection. The OSI model handles this problem in the session layer. It is worth noting that the U.S. National Bureau of Standards was so displeased with this property of TP4 that it introduced extra TPDUs into the transport protocol to allow disconnection without data loss. As a consequence, the U.S. and international versions of TP4 are not identical.

NETWORKING IN UNIX

A large number of machines on the ARPANET and the ARPA Internet run Berkeley UNIX. For this reason, it is worth taking a short look at how it handles networking. Berkeley UNIX supports TCP/IP, which is accessed through a set of primitives.

In contrast to the OSI primitives, which are highly abstract, the Berkeley primitives are much more specific. They are implemented as a set of system calls that allow users to access the transport service. The major system calls are listed in Fig. 6-29. Several minor calls and some variants of the major calls are not shown, for simplicity.

Socket	Create a TSAP of a given type
Bind	Associate an ASCII name to a previously created socket
Listen	Create a queue to store incoming connection requests
Accept	Remove a connection request from the queue or wait for one
Connect	Initiate a connection with a remote socket
Shutdown	Terminate the connection on a socket
Send	Send a message through a given socket
Recv	Receive a message on a given socket
Select	Check a set of sockets to see if any can be read or written

Fig. 6-29. The principal transport service calls in Berkeley UNIX.

Central to the service interface is the concept of a **socket**, which is similar to an OSI TSAP. Sockets are end points to which connections can be attached from the bottom (the operating system side) and to which processes can be attached from the top (the user side). The *socket* system call creates a socket (a data structure within the operating system). The parameters to the call specify the address format (e.g., an ARPA Internet name), the socket type (e.g., connection-oriented or connectionless), and the protocol (e.g., TCP/IP).

Once a socket has been created, buffer space can be allocated to it for storing incoming connection requests. This space is allocated using the *listen* call. A socket specified in a *listen* call then becomes a passive end point waiting for a connection request to arrive from outside.

In order for a remote user to send a connection request to a socket, the socket has to have a name (TSAP address). Names can be attached to sockets by the *bind* system call. Once bound, a name can be published or distributed in some way, so that remote processes can address the socket.

The way a user process attaches itself to a socket to passively await the arrival of a connection request is the *accept* call (essentially the same as the *listen* call in our example). If a request has already arrived, it will be taken from the socket's queue; otherwise, the process will block until a request comes in (unless the socket has been specified as nonblocking). Either way, when a request is available, a new socket is created and the new socket is used for the connection end point. In this manner, a single well-known port can be used to establish many connections.

To initiate a connection to a remote socket, a process can make the *connect* system call specifying a local socket and a remote name as parameters. This call establishes a connection between the two sockets. Alternatively, if the sockets are of the connectionless type, the operating system records an association between the two, so that *send*s on the local socket result in messages being sent to the remote one, even though no formal connection exists.

To terminate a connection or association the *shutdown* call is used. The two directions of a full duplex connection can be independently shut down.

The calls *send* and *recv* are used to send and receive messages. Several variations of the basic call are present.

Finally, we have the *select* call. This system call is useful for processes that have several connections established. In many cases such a process wants to do a *recv* on any socket that has a message waiting for it. Unfortunately it does not know which sockets have messages pending and which do not. If it picks one socket at random, it may end up blocking for a long time waiting for a message, while messages are waiting on several other sockets. The *select* call allows it to block until reads (or writes) are possible on some set of sockets specified by the parameters. For example, a process can say that it wants to block until a message is available on any one of a given set of sockets. When the call terminates, the caller is told which sockets have messages pending and which do not.

6.4.3. The Transport Layer in MAP and TOP

MAP and TOP use the OSI transport protocol. Since they use a connectionless protocol (ISO 8473) in the network layer, they are forced to use the TP4 variant in the transport layer.

TP4 contains several options that can be negotiated at the time a connection is established. Specific choices have been made for some of them. To start with, the computer initiating the connection is required to specify the use of Class 4 in the *CR* TPDU, and the responder is required to accept it in the *CC* TPDU. If either side is unable or unwilling to do this, the connection cannot be established.

Both the 7-bit and 31-bit sequence number options must be supported. Implementations are encouraged to use the 31-bit option all the time, except in those circumstances in which the underlying network cannot support it.

Expedited data must be supported by all implementations, even though the OSI standard says that it is optional. If during the connection establishment either peer rejects this option, the connection must be rejected because some of the upper layers use this facility.

Finally, all MAP and TOP implementations must be prepared to use the software checksum option, in which transport entities checksum each TPDU before sending it or after receiving it. Doing the checksum in software is very expensive in terms of CPU time, but guards against the possibility of losing data due to memory errors. Use of this option for any given connection is optional, but if it is desired, it must be available.

6.4.4. The Transport Layer in USENET

USENET does not have an official transport protocol. Each pair of communicating machines can negotiate the use of any desired transport protocol (or none at all). Many pairs of machines use TCP/IP, but X.25 and the *uucp* protocol described in Chap. 4 are also widely used.

The upper layers do not require any specific transport protocol, although they do need a way to establish a reliable connection between machines for the purpose of logging in. If this goal can be achieved, each pair of machines can use any mutually agreeable transport protocol, or none at all, if the underlying communication is reliable enough (e.g., over a LAN).

6.5. SUMMARY

The purpose of the transport layer is to bridge the gap between what the network layer offers and what the transport user wants. It also serves to isolate the upper layers from the technology of the network layer by providing a standardized service definition. In this way, changes in the network technology will not require changes to software in the higher layers.

The OSI transport service definition views a connection as having three phases: establishment, use, and release. For each phase, service primitives are available to perform the required actions. Connectionless service is also provided for.

Network layer service can be categorized as A, B, or C, depending on how reliable it is. For type A (reliable) networks, simple protocols can be used. For type B (almost reliable) networks, the transport protocol must be able to recover from *N-RESET*s. For type C networks, the transport protocol must use complex mechanisms to deal with many subtle errors that may occur.

Connection management is a key responsibility of the transport layer. For type C networks, connection establishment needs to be quite elaborate, usually involving a three-way handshake. Connection release can also be a problem, as our example of the two-army problem demonstrated. One possible solution is to use timer-based connections.

After finishing with connection management, we studied a sample transport layer using X.25 as the network service. In this example we saw one possible way of relating the abstract OSI service primitives to executable procedures. We also saw how the protocol can be represented as a finite state machine.

Finally, we examined the transport layer in our usual running examples and looked at the two main transport protocols currently in use, the OSI transport protocol and TCP. The similarities and differences between these two protocols were pointed out.

PROBLEMS

1. The dynamic buffer allocation scheme of Fig. 6-18 tells the sender how many buffers it has beyond the acknowledged message. An alternative way of conveying the same information would be for the buffer field to simply tell how many additional buffers, if any, had been allocated. In this method the sender maintains a counter that is incremented by the contents of the buffer field in arriving messages, and decremented when a message is sent for the first time. Are the two methods equally good?

2. A user process sends a stream of 128-byte messages to another user process over a connection. The receiver's main loop consists of two actions, fetch message and process message. The time required to fetch and process a message has an exponential probability density, with mean 10 msec. The window mechanism allows up to 16 outstanding messages at any instant. All communication lines in the subnet are 230 kbps, but due to delays in the subnet, the arrival pattern at the receiver is approximately Poisson. Measurements show that the time for an acknowledgement to get back to the sender, measured from the first bit of transmission, is 200 msec. Use queueing theory to determine the mean number of bytes of buffer space required at the receiving host.

3. A group of N users located in the same building are all using the same remote computer via an X.25 network. The average user generates L lines of traffic (input + output) per hour, on the average, with the mean line length being P bytes, excluding the X.25 headers. The packet carrier charges C cents per byte of user data transported, plus X cents per hour for each X.25 virtual circuit open. Under what conditions is it cost effective to multiplex all N transport connections onto the same X.25 virtual circuit, if such multiplexing adds 2 bytes of data to each packet? Assume that even one X.25 virtual circuit has enough bandwidth for all the users.

4. Class 0 of the OSI transport protocol does not have any explicit flow control procedure. Does this mean that a fast sender can transmit data can drown a slow receiver in data?

5. Imagine a generalized n-army problem, in which the agreement of any two of the armies is sufficient for victory. Does a protocol exist that allows blue to win?

6. In a network that has a maximum packet size of 128 bytes, a maximum packet lifetime

of 30 sec, and an 8-bit packet sequence number, what is the maximum data rate per connection?

7. Suppose that the clock-driven scheme for generating initial sequence numbers is used with a 15-bit wide clock counter. The clock ticks once every 100 msec, and the maximum packet lifetime is 60 sec. How often need resynchronization take place
(a) in the worst case?
(b) when the data consumes 240 sequence numbers/min?

8. Why does the maximum packet lifetime, *T*, have to be large enough to ensure that not only the packet, but also its acknowledgements have vanished?

9. Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

10. Consider the problem of recovering from host crashes (i.e., Fig. 6-20). If the interval between writing and sending an acknowledgement, or vice versa, can be made relatively small, what are the two best sender-receiver strategies for minimizing the chance of a protocol failure?

11. Are deadlocks possible with the transport entity described in the text?

12. What happens when the user of the transport entity given in Fig. 6-21 sends a zero length message? Discuss the significance of your answer.

13. Out of curiosity, the implementer of the transport entity of Fig. 6-21 has decided to put counters inside the *sleep* procedure to collect statistics about the *conn* array. Among these are the number of connections in each of the seven possible states, n_i $(i = 1, \ldots, 7)$. After writing a massive FORTRAN program to analyze the data, our implementer discovered that the relation $\sum n_i = MaxConn$ appears to always be true. Are there any other invariants involving only these seven variables?

14. For each event that can potentially occur in the transport entity of Fig. 6-21, tell whether it is legal or not when the user is sleeping in *sending* state.

15. The X.25 protocol does not use subsequence numbers like TP4. Is this simply because X.25 was invented years ago, before the idea of subsequence numbers had been thought of, or is there a different reason?

16. Consider the problem of internetworking with type C connectionless subnets. If a TPDU passes through a network whose maximum packet size is smaller than the standard TPDU size, will this cause problems with TP4? How about with TCP?

17. TCP only allows one connection to exist between any pair of TSAPs. Do you think this is also true of TP4? Discuss your answer.

18. Discuss the advantages and disadvantages of credits versus sliding window protocols.

19. TCP uses a single transport protocol header, whereas OSI has many of them. Discuss the advantages and disadvantages of each method.

20. Datagram fragmentation and reassembly is handled by IP, and is invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

21. Modify the program of Fig. 6-21 to do error recovery. Add a new packet type, *reset*, that can arrive after a connection has been opened by both sides but closed by neither. This event, which happens simultaneously on both ends of the connection, means that any packets that were in transit have either been delivered or destroyed, but in either case no longer are in the subnet.

22. Modify the program of Fig. 6-21 to multiplex all transport connections onto a single X.25 virtual circuit. This change will probably require you to create and manage an explicit transport header to keep track of which packet belongs to which connection.

23. Write a program that simulates buffer management in a transport entity using a sliding window for flow control rather than the credit system of Fig. 6-21. Let higher layer processes randomly open connections, send data, and close connections. To keep it simple, have all the data travel from machine *A* to machine *B*, and none the other way. Experiment with different buffer allocation strategies at *B*, such as dedicating buffers to specific connections versus a common buffer pool, and measure the total throughput achieved by each one.

SECOND EDITION

COMPUTER NETWORKS

By Andrew S. Tanenbaum

A comprehensive introduction to computer networks, this book emphasizes network protocols and algorithms, from the physical layer to the application layer and from local area networks to satellite networks. Assuming only a general familiarity with computer systems and programming, **Computer Networks** presents the full spectrum of basic protocols, concepts, algorithms, software, and technologies.

Revised to reflect the latest advances in the field, this second edition features material on LANs (incuding IEEE 802), ISDN (Integrated Services Digital Networks), and fiber optics networks. It also has detailed coverage of the upper layers of the OSI model (transport, session, presentation, and application) as well as extended discussions of MAP, TOP, and USENET.

Outstanding content highlights include:

• The Physical Layer • The Data Link Layer II—Medium Access • The Data Link Layer II—Protocols • The Network Layer • The Transport Layer • The Session Layer • The Presentation Layer • The Application Layer • Suggested Reading and Bibliography •

Other fine titles by Andrew S. Tanenbaum available from Prentice Hall:

Structured Computer Organization, Third Edition

The third edition of this best selling book has been thoroughly revised to discuss the technology and techniques that will be important in the 1990s. The book covers the CPU, memory, I/O, digital logic, microprogramming, virtual memory, multiprogramming, and assembly language programming, among many other topics. A large amount of new material has been included in this third edition on RISC machines and parallel processors. The Intel 80386, Motorola 68030, SPARC, and MIPS chips are used as running examples to illustrate the principles of both traditional and RISC architectures.

Operating Systems: Design and Implementation

This complete step-by-step guide to the principles and practices of today's operating systems examines interprocess communication, scheduling and paging algorithms, deadlocks, input/output, memory management, file systems, and security. To illustrate these principles, this practical text also contains the MINIX operating system source code (developed by the author), enabling readers to apply the general principles of operating systems to a UNIX-like system.

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-162959-X

TANENBAUM COMPUTER NETWORKS