Larry L. Peterson & Bruce S. Davie

COMPUTER NETWORKS

A Systems Approach

Morgan Kaufmann Publishers, Inc.
San Francisco, California

Sponsoring Editor Jennifer Mann
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Assistant Jane Elliott
Cover and Text Design Ross Carron Design
Illustration ST Associates, Inc.
Copyeditor Jeff Van Bueren
Proofreader Ken DellaPenta
Composition Ed Sznyter, Babel Press
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205 USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet mkp@mkp.com
Order toll free 800/745-7323

00 99 98 97 96 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data
Peterson, Larry L.
Computer networks : a systems approach / Larry L. Peterson and Bruce S. Davie.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-368-9
1. Computer networks. I. Davie, Bruce S. II. Title.
TK5105.5.P479 1996
004.6'5—dc20 96-1590

CONTENTS

1.2 Requirements

Before trying to understand how a network that supports applications like FTP, WWW, and NV is designed and implemented, it is important to identify what we expect from a network. The short answer is that there is no single expectation; computer networks are designed and built under a large number of constraints and requirements. In fact, the requirements differ widely depending on your perspective:

- a *network user* would list the services that his or her application needs, for example, a guarantee that each message the application sends will be delivered without error within a certain amount of time;
- a *network designer* would list the properties of a cost-effective design, for example, that network resources are efficiently utilized and fairly allocated to different users; and
- a *network provider* would list the characteristics of a system that is easy to administer and manage, for example, in which faults can be easily isolated and where it is easy to account for usage.

This section attempts to distill these different perspectives into a high-level introduction to the major considerations that drive network design, and in doing so, identifies the challenges addressed throughout the rest of this book.

1.2.1 Connectivity

Starting with the obvious, a network must provide connectivity among a set of computers. Sometimes it is enough to build a limited network that connects only a few select machines. In fact, for reasons of privacy and security, many private (corporate) networks have the explicit goal of limiting the set of machines that are connected. In contrast, other networks (of which the Internet is the prime example), are designed to grow in a way that allows them the potential to connect all the computers in the world. A system that is designed to support growth to an arbitrarily large size is said to *scale*. Using the Internet as a model, this book addresses the challenge of scalability.

Links, Nodes, and Clouds

Network connectivity occurs at many different levels. At the lowest level, a network can consist of two or more computers directly connected by some physical medium, such as a coaxial cable or an optical fiber. We call such a physical medium a *link*, and we often refer to the computers it connects as *nodes*. (Sometimes a node is a more specialized piece of hardware rather than a computer, but we overlook that distinction for the purposes of this discussion.) As illustrated in Figure 1.4, physical links are sometimes limited to a pair of nodes (such a link is said to be *point-to-point*), while in other cases, more than two nodes may share a single physical link (such a link is said to be *multiple-access*). Whether a given link supports point-to-point or multiple-access connectivity depends on how the node is attached to the link. It

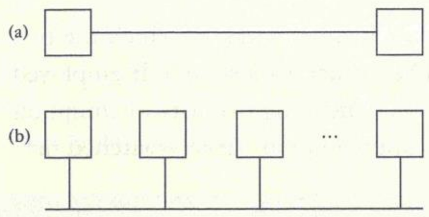

Figure 1.4 Direct links: (a) point-to-point; (b) multiple-access.

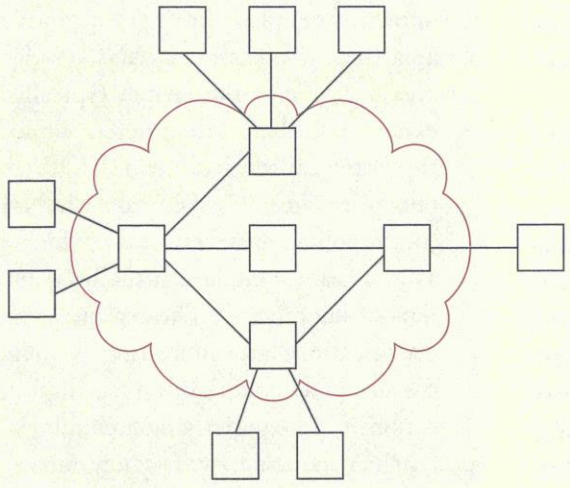

Figure 1.5 Switched network.

is also the case that multiple-access links are often limited in size, in terms of both the geographical distance they can cover and the number of nodes they can connect. The exception is a satellite link, which can cover a wide geographic area.

If computer networks were limited to situations in which all nodes are directly connected to each other over a common physical medium, then networks would either be very limited in the number of computers they could connect or the number of wires coming out of the back of each node would quickly become both unmanageable and very expensive. Fortunately, connectivity between two nodes does not necessarily imply a direct physical connection between them—indirect connectivity may be achieved among a set of cooperating nodes. Consider the following two examples of how a collection of computers can be indirectly connected.

Figure 1.5 shows a set of nodes, each of which is attached to one or more point-to-point links. Those nodes that are attached to at least two links run software that forwards data received on one link out on another. If organized in a systematic way, these forwarding nodes form a *switched network*. There are numerous types of switched networks, of which the two most common are *circuit-switched* and *packet-switched*. The former is most notably employed by the telephone system, while the latter is used for the overwhelming majority of computer networks and will be the focus of this book. The important feature of packet-switched networks is that the nodes in such a network send discrete blocks of data to each other. Think of these blocks of data as corresponding to some piece of application data such as a file, a piece of email, or an image. We call each block of data either a *packet* or a *message*, and for now we use these terms interchangeably; we discuss the reason they are not always the same in Section 1.2.2.

Packet-switched networks typically use a strategy called *store-and-forward*. As the name suggests, each node in a store-and-forward network first receives a complete packet over some link, stores the packet in its internal memory, and then forwards the complete packet to the next node. In contrast, a circuit-switched network first establishes a dedicated circuit across a sequence of links and then allows the source node to send a stream of bits across this circuit to a destination node. The major reason for using packet switching rather than circuit switching in a computer network is discussed in the next subsection.

The cloud in Figure 1.5 distinguishes between the nodes on the inside that *implement* the network (they are commonly called *switches* and their sole function is to store and forward packets) and the nodes on the outside of the cloud that *use* the network (they are commonly called *hosts* and they support users and run application programs). Also note that the cloud in Figure 1.5 is one of the most important icons of computer networking. In general, we use a cloud to denote any type of network, whether it is a single point-to-point link, a multiple-access link, or a switched network. Thus, whenever you see a cloud used in a figure, you can think of it as a placeholder for any of the networking technologies covered in this book.

DANs, LANs, MANs, and WANs

One way to characterize networks is according to their size. Two well-known examples are LANs (local area networks) and WANs (wide area networks)—the former typically extend less than 1 kilometer, while the latter can be worldwide. Other networks are classified as MANs (metropolitan area networks), which, as the name implies, usually span tens of kilometers. The reason such classifications are interesting is that the size of a network often has implications for the underlying technology that can be used, with a key factor being the amount of time it takes for data to propagate from one end of the network to the other; we discuss this issue more in later chapters.

An interesting historical note is that the term wide area network was not applied to the first WANs because there was no other sort of network to differentiate them from. When computers were incredibly rare and expensive, there was no point in

A second way in which a set of computers can be indirectly connected is shown in Figure 1.6. In this situation, a set of independent networks (clouds) are interconnected to form an *internetwork*, or internet for short. We adopt the Internet's convention of referring to a generic internetwork of networks as a lowercase i internet, and the currently operational TCP/IP Internet as the capital I Internet. A node that is connected to two or more networks is commonly called a *router* or *gateway*, and it plays much the same role as a switch—it forwards messages from one network to another. Note that an internet can itself be viewed as another kind of network, which means that an internet can be built from an interconnection of internets. Thus, we can recursively build arbitrarily large networks by interconnecting clouds to form larger clouds.

thinking about how to connect all the computers in the local area—there was only one computer in that area. Only as computers began to proliferate did LANs become necessary, and the term WAN was then introduced to describe the larger networks that interconnected geographically distant computers.

One of the most intriguing kinds of networks that is gaining attention today is the DAN (desk area network). The idea of a DAN is to open up the computer setting on your desk and to treat each component of that computer—e.g., its display, disk, CPU, as well as peripherals like cameras and printers—as a network-accessible device. In essence, the I/O bus is replaced by a network (a DAN) that can, in turn, be interconnected to other LANs, MANs, and WANs. Establishing this interconnection provides uniform access to all the resources that might be required by a network application.

Just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. The final requirement is that each node must be able to say which of the other nodes on the network it wants to communicate with. This is done by assigning an *address* to each node. An address is a byte string that identifies a node; i.e., the network can use a node's address to distinguish it from the other nodes connected to the network. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the switches and routers of the network use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward the destination node based on its address is called *routing*.

This brief introduction to addressing and routing has presumed that the source node wants to send a message to a single destination node (*unicast*). While this is the most common scenario, it is also possible that the source node might want to *broadcast* a message to all the nodes on the network. Or a source node might want to send a message to some subset of

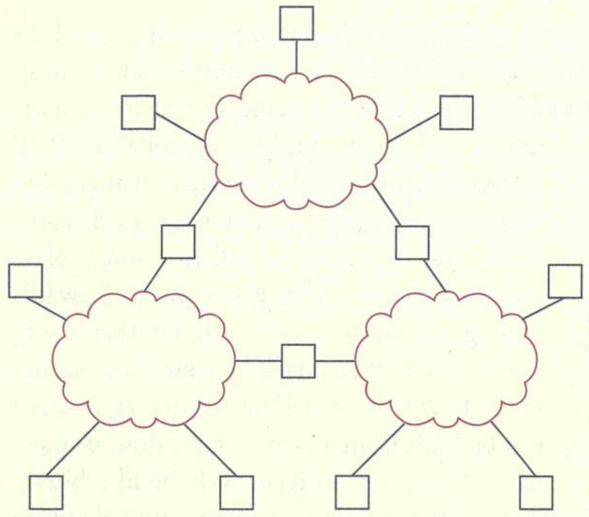

Figure 1.6 Interconnection of networks.

the other nodes, but not all of them, a situation called *multicast*. Thus, in addition to node-specific addresses, another requirement of a network is that it support multicast and broadcast addresses.

The main thing to take away from this discussion is that we can define a *network* recursively as consisting of two or more nodes connected by a physical link, or as two or more networks connected by one or more nodes. In other words, a network can be constructed from a nesting of networks, where at the bottom level, the network is implemented by some physical medium. One of the key challenges in providing network connectivity is to define an address for each node that is reachable on the network (including support for broadcast and multicast connectivity), and to be able to use this address to route messages toward the appropriate destination node(s).

1.2.2 Cost-Effective Resource Sharing

As stated above, this book focuses on packet-switched networks. This section explains the key requirement of computer networks—in short, efficiency—that leads us to packet switching as the strategy of choice.

Given a collection of nodes indirectly connected by a nesting of networks, it is possible for any pair of hosts to send messages to each other across a sequence of links and nodes. Of course, we want to do more than support just one pair of communicating hosts—we want to provide all pairs of hosts with the ability to exchange messages. The question, then, is how do all the hosts that want to communicate share the network, especially if they want to use it at the same time? And, as if that problem isn't hard enough, how do several hosts share the same *link* when they all want to use it at the same time?

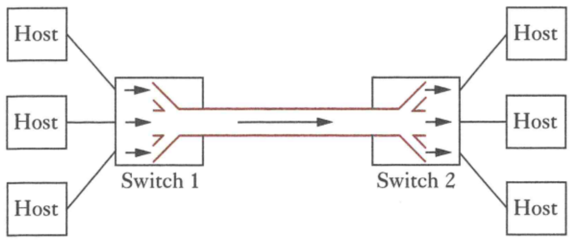

Figure 1.7 Multiplexing multiple logical flows over a single physical link.

To understand how hosts share a network, we need to introduce a fundamental concept, *multiplexing*, which means that a system resource is shared among multiple users. At an intuitive level, multiplexing can be explained by analogy to a timesharing computer system, where a single physical CPU is shared (multiplexed) among multiple jobs, each of which believes it has its own private processor. Similarly, data being sent by multiple users can be multiplexed over the physical links that make up a network.

To see how this might work, consider the simple network illustrated in Figure 1.7, where the three hosts on the left side of the network are sending data to the three hosts on the right by sharing a switched network that contains only one physical link. (For simplicity, assume that the top host on the left is communicating with the top host on the right, and so on.) In this situation, three flows of data—corresponding to the three pairs of hosts—are multiplexed onto a single physical link by Switch 1 and then *demultiplexed* back into separate flows by Switch 2. Note that we are being intentionally vague about exactly what a "flow of data" corresponds to. For the purposes of this discussion, assume that each host on the left has a large supply of data that it wants to send to its counterpart on the right.

There are several different methods for multiplexing multiple flows onto one physical link. One method, which is commonly used in the telephone network, is *synchronous time-division multiplexing* (STDM). The idea of STDM is to divide time into equal-sized quanta, and in a round-robin fashion, give each flow a chance to send its data over the physical link. In other words, during time quantum 1, data from the first flow is transmitted; during time quantum 2, data from the second flow is transmitted; and so on. This process continues until all the flows have had a turn, at which time the first flow gets to go again, and the process repeats. Another common method is *frequency-division multiplexing* (FDM). The idea of FDM is to transmit each flow over the physical link at a different frequency, much the same way that the signals for different TV stations are transmitted at a different frequency on a physical cable TV link.

Although simple to understand, both STDM and FDM are limited in two ways. First, if one of the flows (host pairs) does not have any data to send, its share of the physical link—i.e., its time quantum or its frequency—remains idle, even if one of the other flows has data to transmit. For computer communication, the amount of time that a link is idle can be very large—for example, consider the amount of time you spend reading a Web page (leaving the link idle) compared to the time you spend fetching the page. Second, both STDM and FDM are limited to situations in which the maximum number of flows is fixed and known ahead of time. It is not practical to resize the quantum or to add additional quanta in the case of STDM or to add new frequencies in the case of FDM.

The form of multiplexing that we make most use of in this book is called *statistical multiplexing*. Although the name is not all that helpful for understanding the concept, statistical multiplexing is really quite simple; it involves two key ideas. First, it is like STDM in that the physical link is shared over time—first data from one flow is transmitted over the physical link, then data from another flow is transmitted, and so on. Unlike STDM, however, data is transmitted from each flow on demand rather than during a predetermined time slot. Thus, if only one flow has data to send, it gets to transmit that data without waiting for its quantum to come around and thus without having to watch the quanta assigned to the other flows go by unused. It is this avoidance of idle time that gives packet switching its efficiency.

As defined so far, however, statistical multiplexing has no mechanism to ensure that all the flows eventually get their turn to transmit over the physical link. That is, once a flow begins sending data, we need some way to limit the transmission, so that the other flows can have a turn. To account for this need, statistical multiplexing defines an upper bound on the size of the block of data that each flow is permitted to transmit at a given time. This limited-size block of data is typically referred to as a *packet*, to distinguish it from the arbitrarily large *message* that an application program might want to transmit. The fact that a packet-switched network limits the maximum size of packets means that a host may not be able to send a complete message in one packet—the source may need to fragment the message into several packets, with the receiver reassembling the packets back into the original message.

In other words, each flow sends a sequence of packets over the physical link, with a decision made on a packet-by-packet basis as to which flow's packet to send next. Notice that if only one flow has data to send, then it can send a sequence of packets back to back. However, should more than one of the flows have data to send, then their packets are interleaved on the link. Figure 1.8 depicts a switch multiplexing packets from multiple sources onto a single shared link.

The decision as to which packet to send next on a shared link can be made in a number of different ways. For example, in a network consisting of switches interconnected by links such as the one in Figure 1.7, the decision would be made by the switch that transmits packets onto the shared link. (As we will see later, not all packet-switched networks actually involve switches, and they may use other mechanisms to determine whose packet goes onto the link next.) Each switch in a packet-switched network makes this decision independently, on a packet-by-packet basis. One of the issues that faces a network designer is how to make this decision in a fair manner. For example, a switch could be designed to service the different flows in a round-robin manner, just as in STDM. However, statistical multiplexing does not

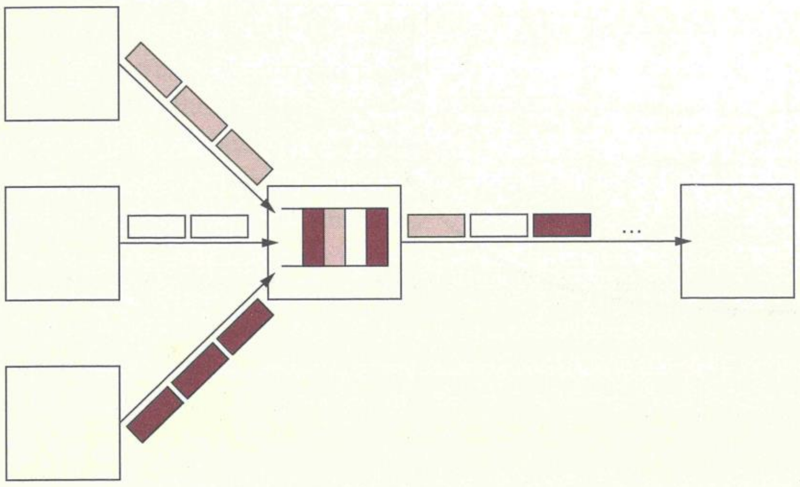

Figure 1.8 A switch multiplexing packets from multiple sources onto one shared link.

require a round-robin approach. In fact, another equally valid choice would be to service each flow's packets on a first-in-first-out (FIFO) basis.

Also, notice in Figure 1.8 that since the switch has to multiplex three incoming packet streams onto one outgoing link, it is possible that the switch will receive packets faster than the shared link can accommodate. In this case, the switch is forced to buffer these packets in its memory. Should a switch receive packets faster than it can send them for an extended period of time, then the switch will eventually run out of buffer space, and some packets will have to be dropped. When a switch is operating in this state, it is said to be *congested*.

The bottom line is that statistical multiplexing defines a cost-effective way for multiple users (e.g., host-to-host flows of data) to share network resources (links and nodes) in a fine-grained manner. It defines the packet as the granularity with which the links of the network are allocated to different flows, with each switch able to schedule the use of the physical links it is connected to on a per-packet basis. Fairly allocating link capacity to different flows and dealing with congestion when it occurs are the key challenges of statistical multiplexing.

1.2.3 Functionality

While the previous section outlined the challenges involved in providing cost-effective connectivity among a group of hosts, it is overly simplistic to view a computer network as simply delivering packets among a collection of computers. It is more accurate to think of a network as providing the means for a set of application processes that are distributed over those computers to communicate. In other words, the next requirement of a computer network is that the application programs running on the hosts connected to the network must be able to communicate in a meaningful way.

6.1 Simple Demultiplexer (UDP)

The simplest possible transport protocol is one that extends the host-to-host delivery service of the underlying network into a process-to-process communication service. There are likely to be many processes running on any given host, so the protocol needs to add a level of demultiplexing, thereby allowing multiple application processes on each host to share the network. Aside from this requirement, the transport protocol adds no other functionality to the best-effort service provided by the underlying network. The Internet's User Datagram Protocol (UDP) is an example of such a transport protocol. So is A Simple Protocol (ASP) described in Chapter 2.

The only interesting issue in such a protocol is the form of the address used to identify the target process. Although it is possible for processes to *directly* identify each other with an OS-assigned process id (pid), such an approach is only practical in a "closed" distributed system in which a single OS runs on all hosts and assigns each process a unique id. A more common approach, and the one used by both ASP and UDP, is for processes to *indirectly* identify each other using an abstract locater, often called a *port* or *mailbox*. The basic idea is for a source process to send a message to a port and for the destination process to receive the message from a port.

The header for an end-to-end protocol that implements this demultiplexing function typically contains an identifier (port) for both the sender (source) and the receiver (destination) of the message. For example, the UDP header is given in Figure 6.1. Notice that the UDP port field is only 16 bits long. This means that there are up to 64-K possible ports, clearly not enough to identify all the processes on all the hosts in the Internet. Fortunately, ports are not interpreted across the entire Internet, but only on a single host. That is, a process is really identified by a port on some particular host—a ⟨port, host⟩ pair. In fact, this pair constitutes the demultiplexing key for the UDP protocol.

The next issue is how a process learns the port for the process to which it wants to send a message. Typically, a client process initiates a message exchange with a server process. Once a client has contacted a server, the server knows the client's port (it was contained in the message header) and can reply to it. The real problem, therefore, is how the client learns the server's port in the first place. A common approach is for the server to accept messages at a *well-known port*. That is, each server receives its messages at some fixed port that is widely published, much like the emergency telephone service is available at the well-known phone number 911. In the Internet, for example, the Domain Name Server (DNS) receives messages at well-known port 53 on each host, the Unix talk program accepts messages at well-known port 517, and so on. This mapping is published periodically in an RFC and is available on most Unix systems in file /etc/services. Sometimes a well-known port is just the starting point for communication: the client and server use the well-known port to agree on some other port that they will use for subsequent communication, leaving the well-known port free for other clients.

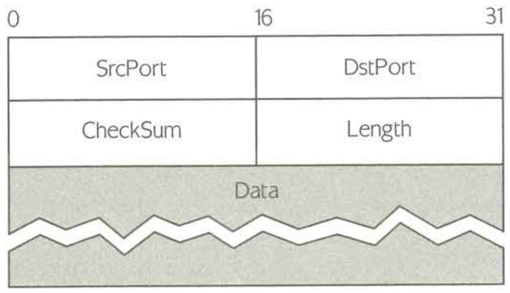

Figure 6.1 Format for UDP header.

As just mentioned, a port is purely an abstraction. Exactly how it is implemented differs from system to system, or more precisely, from OS to OS. Typically, a port is implemented by a message queue. When a message arrives, the protocol (e.g., UDP) appends the message to the end of the queue. Should the queue be full, the message is discarded. There is no flow-control mechanism that tells the sender to slow down. When an application process wants to receive a message, one is removed from the front of the queue. If the queue is empty, the process blocks until a message becomes available.

Finally, although UDP does not implement flow control or reliable/ordered delivery, it does a little more work than to simply demultiplex messages to some application process—it also ensures the correctness of the message by the use of a checksum. (The UDP checksum is optional in the current Internet, but will become mandatory with IPv6.) UDP computes its checksum over the UDP header, the contents of the message body, and something called the *pseudoheader*. The pseudoheader consists of three fields from the IP header: length, source IP address, and destination IP address. UDP uses the same checksum algorithm as IP, as defined in Section 3.4.3. The motivation behind having the pseudoheader is to verify that this message has been delivered between the correct two endpoints. For example, if the destination IP address was modified while the packet was in transit, causing the packet to be misdelivered, this fact would be detected by the UDP checksum.

6.2 Reliable Byte Stream (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol is one that offers a connection-oriented, reliable byte-stream service. Such a service has proven useful to a wide assortment of applications because it frees the application from having to worry about missing or reordered data. The Internet's Transmission Control Protocol (TCP) is probably the most widely used protocol of this type. It is also the most carefully optimized, which makes it an interesting protocol to study.

In terms of the properties of transport protocols given in the problem statement at the start of this chapter, TCP guarantees the reliable, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit. Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.

In addition to the above features, TCP also implements a highly tuned congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from overrunning the receiver, but so as to keep the sender from overloading the network. A description of TCP's congestion-control mechanism is postponed until Chapter 8, where we discuss it in the larger context of how network resources are fairly allocated.

Since many people confuse congestion control and flow control, we restate the difference. *Flow control* involves preventing senders from overrunning the capacity of receivers. *Congestion control* involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded. Thus, flow control is an end-to-end issue, while congestion control is more of an issue of how hosts and networks interact.

6.2.1 End-to-End Issues

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm we saw in Section 3.5.2, because TCP runs over the Internet rather than a point-to-point link there are many important differences. This subsection identifies these differences and explains how they complicate TCP. The following five subsections then describe how TCP addresses these complications.

First, whereas the sliding window algorithm presented in Section 3.5.2 runs over a single physical link that always connects the same two computers, TCP supports logical connections between processes that are running

TCP Extensions

We have mentioned at three different points in this section that proposed extensions to TCP may help to mitigate some problem that TCP is facing. These proposed extensions are designed to have as small an impact on TCP as possible. In particular, they are realized as options that can be added to the TCP header. (We glossed over this point earlier, but the reason that the TCP header has a HdrLen field is that the header can be of variable length; the variable part of the TCP header contains the options that have been added.) The significance of adding these extensions as options rather than changing the core of the TCP header is that hosts can still communicate using TCP even if they do not implement the options. Hosts that do implement the optional extensions, however, can take advantage of them. The two sides agree that they will use the options during TCP's connection-establishment phase.

The first extension helps to improve TCP's timeout mechanism. Instead of measuring the RTT using a coarse-grained event, TCP can read the actual system clock when it is about to send a segment, and put on any two computers in the Internet. This means that TCP needs an explicit connection-establishment phase during which the two sides of the connection agree to exchange data with each other. This difference is analogous to having to dial up the other party, rather than having a dedicated phone line. TCP also has an explicit connection-teardown phase. One of the things that happens during connection establishment is that the two parties establish some shared state to enable the sliding window algorithm to begin.

Second, whereas a single physical link that always connects the same two computers has a fixed RTT, TCP connections have highly variable round-trip times. For example, a TCP connection between a host in Tucson and a host in New York, which are separated by several thousand kilometers, might have an RTT of 100 ms, while a TCP connection between a host in Tucson and a host in Phoenix, only a few hundred kilometers away, might have an RTT of only 10 ms. The same TCP protocol must be able to support both of these connections. To make matters worse, the TCP connection between hosts in Tucson and New York might have an RTT of 100 ms at 3 a.m., but an RTT of 500 ms at 3 p.m. Variations in the RTT are even possible during a single TCP connection that lasts only a few minutes. What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive. (Certainly, the timeout for a point-to-point link must be a settable parameter, but it is not necessary to adapt this timer frequently.)

The third difference is also related to the variable RTT of a logical connection across the Internet, but it is concerned with the pathological situation in which a packet is delayed in the network for an extended period of time. Recall from Section 5.2 that the time to live (TTL) field of the IP header limits the number of hops that a packet can traverse. (In IPv6, the TTL field is renamed the HopLimit field.) TCP makes use of the fact that limiting the number of hops in-

this time—think of it as a 32-bit *timestamp*—in the segment's header. The receiver then echoes this timestamp back to the sender in its acknowledgment, and the sender subtracts this timestamp from the current time to measure the RTT. In essence, the timestamp option provides a convenient place for TCP to "store" the record of when a segment was transmitted; it stores the time in the segment itself. Note that the endpoints in the connection do not need synchronized clocks, since the timestamp is written and read at the same end of the connection.

The second extension addresses the problem of TCP's 32-bit SequenceNum field wrapping around too soon on a high-speed network. Rather than define a new 64-bit sequence number field, TCP uses the 32-bit timestamp just described to effectively extend the sequence number space. In other words, TCP decides whether to accept or reject a segment based on a 64-bit identifier that has the SequenceNum field in the low-order 32 bits and the timestamp in the high-order 32 bits. Since the timestamp is always directly limits how long a packet can circulate in the Internet. Specifically, TCP assumes that each packet has a maximum lifetime of no more than 60 seconds. Keep in mind that IP does not directly enforce this 60-second value; it is simply a conservative estimate that TCP makes of how long a packet might live in the Internet. This sort of delay is simply not possible in a point-to-point link—a packet put into one end of the link must appear at the other end in an amount of time closely related to the speed of light. The implication of this difference is significant—TCP has to be prepared for very old packets to suddenly show up at the receiver, potentially confusing the sliding window algorithm.

Fourth, the computers connected to a point-to-point link are generally engineered to support the link. For example, if a link's delay × bandwidth product is computed to be 8 KB—meaning that a window size is selected to allow up to 8 KB of data to be unacknowledged at a given time—then it is likely that the computers at either end of the link have the ability to buffer up to 8 KB of data. Designing the system otherwise would be silly. On the other hand, almost any kind of computer can be connected to the Internet, making the amount of resources dedicated to any one TCP connection highly variable, especially considering that any one host can potentially support hundreds of TCP connections at the same time. This means that TCP must include a mechanism that each side uses to "learn" what resources (e.g., how much buffer space) the other side is able to apply to the connection.

Fifth, because the transmitting side of a directly connected link cannot send any faster than the bandwidth of the link allows, and only one host is pumping data into the link, it is not possible to unknowingly congest the link. Said another way, the load on the link is visible in the form of a queue of packets at the sender. In contrast, the sending side of a TCP connection has no idea what links will be traversed to reach the destination. For example, the sending

increasing, it serves to distinguish between two different incarnations of the same sequence number. Note that the timestamp is being used in this setting only to protect against wraparound; it is not treated as part of the sequence number for the purpose of ordering or acknowledging data.

The third extension allows TCP to advertise a larger window, thereby allowing it to fill larger delay × bandwidth pipes that are made possible by high-speed networks. This extension involves an option that defines a *scaling factor* for the advertised window. That is, rather than interpreting the number that appears in the AdvertisedWindow field as indicating how many bytes the sender is allowed to have unacknowledged, this option allows the two sides of TCP to agree that the AdvertisedWindow field counts larger chunks (e.g., how many 16-byte units of data the sender can have unacknowledged). In other words, the window scaling option specifies how many bits each side should left-shift the AdvertisedWindow field before using its contents to compute an effective window.

machine might be directly connected to a relatively fast Ethernet—and so, capable of sending data at a rate of 10 Mbps—but somewhere out in the middle of the network, a 1.5-Mbps T1 link must be traversed. And to make matters worse, data being generated by many different sources might be trying to traverse this same slow link. This leads to the problem of network congestion. Discussion of this topic is delayed until Chapter 8.

We conclude this discussion of end-to-end issues by comparing TCP's approach to providing a reliable/ordered delivery service with the approach used by X.25 networks. In TCP, the underlying IP network is assumed to be unreliable and to deliver messages out of order; TCP uses the sliding window algorithm on an end-to-end basis to provide reliable/ordered delivery. In contrast, X.25 networks use the sliding window protocol within the network, on a hop-by-hop basis. The assumption behind this approach is that if messages are delivered reliably and in order between each pair of nodes along the path between the source host and the destination host, then the end-to-end service also guarantees reliable/ordered delivery.

The problem with this latter approach is that a sequence of hop-by-hop guarantees does not necessarily add up to an end-to-end guarantee. First, if a heterogeneous link (say, across an Ethernet) is added to one end of the path, then there is no guarantee that this hop will preserve the same service as the other hops. Second, just because the sliding window protocol guarantees that messages are delivered correctly from node A to node B, and then from node B to node C, it does not guarantee that node B behaves perfectly. For example, network nodes have been known to introduce errors into messages while transferring them from an input buffer to an output buffer. They have also been known to accidentally reorder messages. As a consequence of these small windows of vulnerability, it is still necessary to provide true end-to-end checks to guarantee reliable/ordered service, even though the lower levels of the system also implement that functionality.

This discussion serves to illustrate one of the most important principles in system design—the *end-to-end argument*. In a nutshell, the end-to-end argument says that a function (in our example, providing reliable/ordered delivery) should not be provided in the lower levels of the system unless it can be completely and correctly implemented at that level. Therefore, this rule argues in favor of the TCP/IP approach. This rule is not absolute, however. It does allow for functions to be incompletely provided at a low level as a performance optimization. This is why it is perfectly consistent with the end-to-end argument to perform error detection (e.g., CRC) on a hop-by-hop basis; detecting and retransmitting a single corrupt packet across one hop is preferable to having to retransmit an entire file end-to-end.

6.2.2 Segment Format

TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection. Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet. Instead, TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into

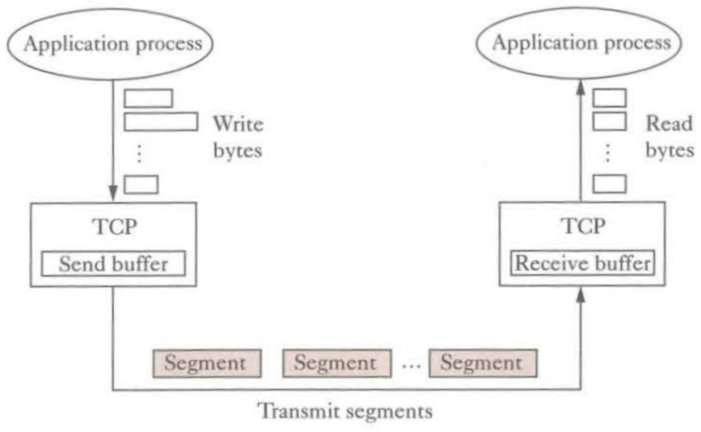

Figure 6.2 How TCP manages a byte stream.

a receive buffer, and the receiving process reads from this buffer at its leisure. This situation is illustrated in Figure 6.2, which, for simplicity, shows data flowing in only one direction. Remember that, in general, a single TCP connection supports byte streams flowing in both directions.

The packets exchanged between TCP peers in Figure 6.2 are called *segments*, since each one carries a segment of the byte stream. One question you might ask is, how does TCP decide that it has enough bytes to send a segment? The answer is that TCP has three mechanisms to trigger the transmission of a segment. First, TCP maintains a threshold variable, typically called the maximum segment size (MSS), and it sends a segment as soon as it has collected MSS bytes from the sending process. MSS is usually set to the size of the largest segment TCP can send without causing the local IP to fragment. That is, MSS is set to the MTU of the directly connected network, minus the size of the TCP and IP headers. The second thing that triggers TCP to transmit a segment is that the sending process has explicitly asked it to do so. Specifically, TCP supports a *push* operation, and the sending process invokes this operation to effectively flush the buffer of unsent bytes. (This push operation is not the same as the the x-kernel's xPush.) This operation is used in terminal emulators like Telnet because each byte has to be sent as soon as it is typed. The final trigger for transmitting a segment is a timer that periodically fires; the resulting segment contains as many bytes as are currently buffered for transmission.

Each TCP segment contains the header schematically depicted in Figure 6.3. The relevance of most of these fields will become apparent throughout this section. For now, we simply introduce them.

The SrcPort and DstPort fields identify the source and destination ports, respectively, just as in UDP. These two fields, plus the source and destination IP addresses, combine to uniquely identify each TCP connection. That is, TCP's demux key is given by the 4-tuple:

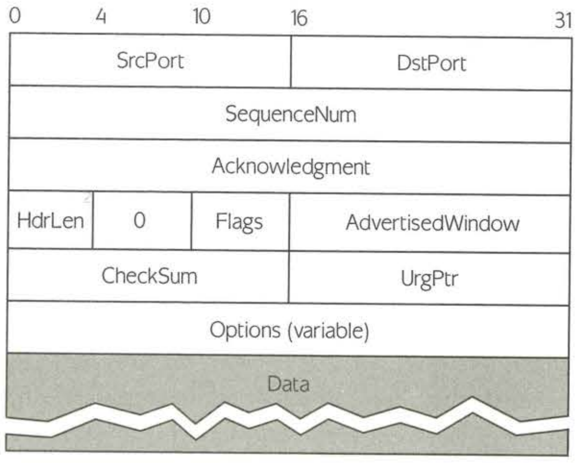

Figure 6.3 TCP header format.

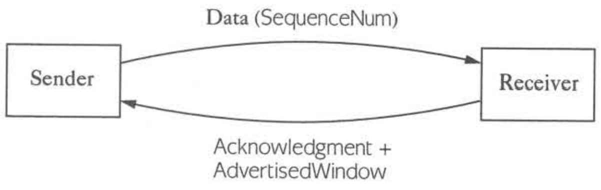

Figure 6.4 Simplified illustration (showing only one direction) of TCP process, with data flow in one direction and ACKs in the other.

⟨ SrcPort, SrcIPAddr, DstPort, DstIPAddr ⟩.

Note that because TCP connections come and go, it is possible for a connection between a particular pair of ports to be established, used to send and receive data, and closed, and then at a later time for the same pair of ports to be involved in a second connection. We sometimes refer to this situation as two different *incarnations* of the same connection.

The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm. Because TCP is a byte-oriented protocol, each byte of data has a sequence number; the SequenceNum field contains the sequence number for the first byte of data carried in that segment. The Acknowledgment and AdvertisedWindow fields carry information about the flow of data going in the other direction. To simplify our discussion, we ignore the fact that data can flow in both directions, and we concentrate on data that has a particular SequenceNum flowing in one direction and Acknowledgment and AdvertisedWindow values flowing in the opposite direction, as illustrated in Figure 6.4. The use of these three fields is described more fully in Section 6.2.4.

The 6-bit Flags field is used to relay control information between TCP peers. The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK. The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively. Their use is described in Section 6.2.3. The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it. The URG flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the non-urgent data contained in this segment begins. The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment. The PUSH flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact. We discuss these last two features more in Section 6.2.6. Finally, the RESET flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

Finally, the CheckSum field is used in exactly the same way as in UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header. The checksum is required for TCP in both IPv4 and IPv6. Also, since the TCP header is of variable length (options can be attached after the mandatory fields), a HdrLen field is included that gives the length of the header in 32-bit words. This field is also known as the Offset field, since it measures the offset from the start of the packet to the start of the data.

6.2.3 Connection Establishment and Termination

A TCP connection begins with a client (caller) doing an active open to a server (callee). Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection. (Recall from Chapter 2 that a party wanting to initiate a connection performs an active open, while a party willing to accept a connection does a passive open.) Only after this connection-establishment phase is over do the two sides begin sending data. Likewise, as soon as a participant is done sending data, it closes its half of the connection, which causes TCP to initiate a round of connection-termination messages. Notice that while connection setup is an asymmetric activity—one side does a passive open and the other side does an active open—connection teardown is symmetric—each side has to close the connection independently. Therefore, it is possible for one side to have done a close, meaning that it can no longer send data, but for the other side to keep its half of the bidirectional connection open and to continue sending data.

Three-Way Handshake

The algorithm used by TCP to establish and terminate a connection is called a *three-way handshake*. We first describe the basic algorithm and then show how it is used by TCP. The three-way handshake involves the exchange of three messages between the client and the server, as illustrated by the timeline given in Figure 6.5.

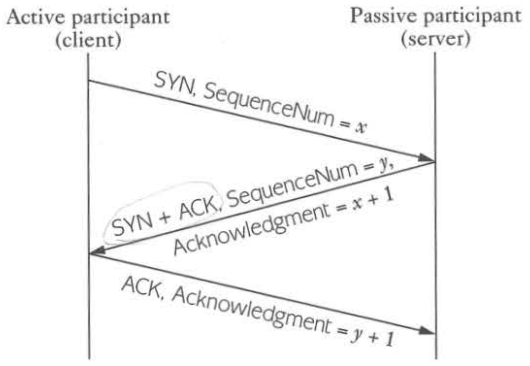
Active participant (client)
Passive participant (server)
SYN, SequenceNum = x
SYN + ACK, SequenceNum = y, Acknowledgment = x + 1
ACK, Acknowledgment = y + 1

Figure 6.5 Timeline for three-way handshake algorithm.

The idea is that two parties want to agree on a set of parameters, which, in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective byte streams. In general, the parameters might be any facts that each side wants the other to know about. First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (Flags = SYN, SequenceNum = x). The server then responds with a single segment that both acknowledges the client's sequence number (Flags = ACK, Ack = x + 1) and states its own beginning sequence number (Flags = SYN, SequenceNum = y). That is, both the SYN and ACK bits are set in the Flags field of this second message. Finally, the client responds with a third segment that acknowledges the server's sequence number (Flags = ACK, Ack = y + 1). The reason that each side acknowledges a sequence number that is one larger than the one sent is that the Acknowledgment field actually identifies the "next sequence number expected," thereby implicitly acknowledging all earlier sequence numbers. Although not shown in this timeline, a timer is scheduled for each of the first two segments, and if the expected response is not received, the segment is retransmitted.

You may be asking yourself why the client and server have to exchange starting sequence numbers with each other at connection setup time. It would be simpler if each side simply started at some "well-known" sequence number, such as 0. In fact, the TCP specification requires that each side of a connection select an initial starting sequence number at random. The reason for this is to protect against two incarnations of the same connection reusing the same sequence numbers too soon, that is, while there is still a chance that a segment from an earlier incarnation of a connection might interfere with a later incarnation of the connection.

State-Transition Diagram

TCP is complex enough that its specification includes a state-transition diagram. A copy of this diagram is given in Figure 6.6. This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything

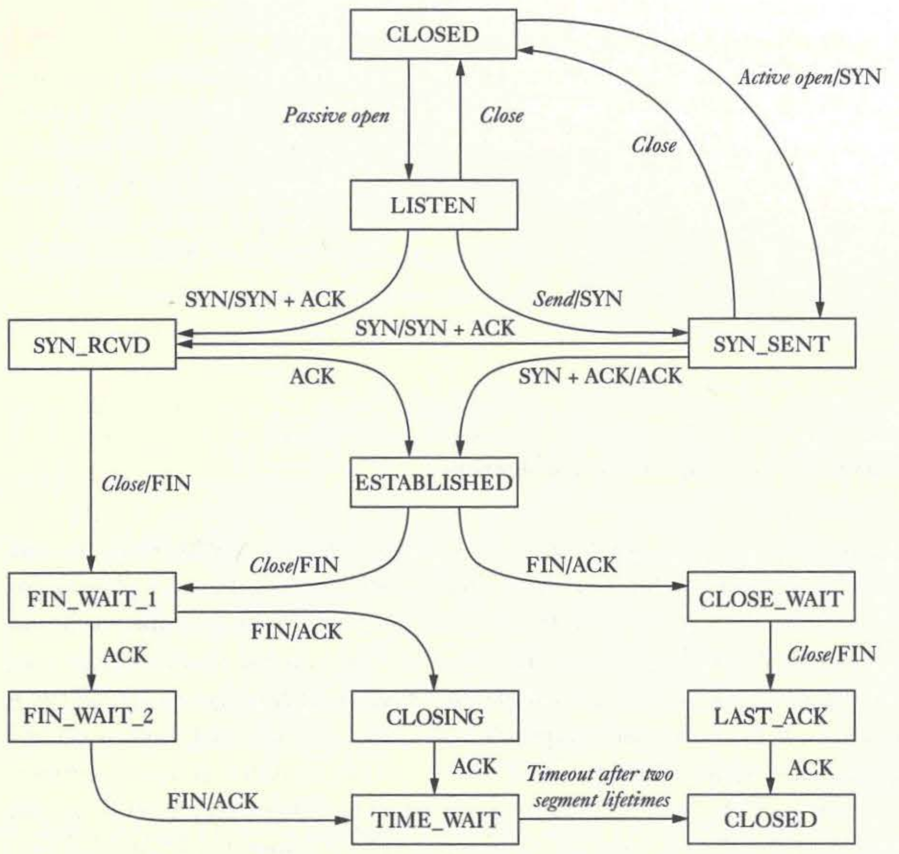

Figure 6.6 TCP state-transition diagram.

below ESTABLISHED). Everything that goes on while a connection is open—i.e., the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.

TCP's state-transition diagram is fairly easy to understand. Each circle denotes a state that any TCP connection can find itself in. All connections start in the CLOSED state. As the connection progresses, the connection moves from state to state according to the arcs. Each arc is labeled with a tag of the form *event/action*. Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the **SYN** flag set), the connection makes a transition to the SYN_RCVD state and takes the action of replying with an ACK + SYN segment.

Notice that two kinds of events trigger a state transition: (1) a segment arrives from the peer (e.g., the event on the arc from LISTEN to SYN_RCVD), and (2) the local application process invokes an operation on TCP (e.g., the *active open* event on the arc from CLOSE to SYN_SENT). In other words, TCP's state-transition diagram effectively defines the *semantics* of both its peer-to-peer interface and its service interface, as defined in Section 1.3.1. The *syntax* of these two interfaces is given by the segment format (as illustrated in Figure 6.3), and by some application programming interface (an example of which is given in Section 6.4), respectively.

Now let's trace the typical transitions taken through the diagram in Figure 6.6. Keep in mind that at each end of the connection, TCP makes different transitions from state to state. When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state. At some later time, the client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN_SENT state. When the SYN segment arrives at the server, it moves to the SYN_RCVD state and responds with a SYN+ACK segment. The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server. When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.

There are three things to notice about the connection-establishment half of the state-transition diagram. First, if the client's ACK to the server is lost, corresponding to the third leg of the three-way handshake, then the connection still functions correctly. This is because the client side is already in the ESTABLISHED state, so the local application process can start sending data to the other end. Each of these data segments will have the ACK flag set, and the correct value in the Acknowledgment field, so the server will move to the ESTABLISHED state when the first data segment arrives. This is actually an important point about TCP—every segment reports what sequence number the sender is expecting to see next, even if this repeats the same sequence number contained in one or more previous segments.

The second thing to notice about the state-transition diagram is that there is a funny transition out of the LISTEN state whenever the local process invokes a *send* operation on TCP. That is, it is possible for a passive participant to identify both ends of the connection (i.e., itself and the remote participant that it is willing to have connect to it), and then for it to change its mind about waiting for the other side and instead actively establish the connection. To the best of our knowledge, this is a feature of TCP that no system-specific interface allows the application process to take advantage of.

The final thing to notice about the diagram is the arcs that are not shown. Specifically, most of the states that involve sending a segment to the other side also schedule a timeout that eventually causes the segment to be resent if the expected response does not happen. These retransmissions are not depicted in the state-transition diagram.

Turning our attention now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection. This complicates the state-transition diagram because it must account for the possibility that the two sides invoke the *close* operator at the same time, as well as the possibility that first one side invokes close and then at some later time, the other side invokes close. Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:

- This side closes first:
  ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.
- The other side closes first:
  ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSED.
- Both sides close at the same time:
  ESTABLISHED → FIN_WAIT_1 → CLOSING → TIME_WAIT → CLOSED.

There is actually a fourth, although rare, sequence of transitions that lead to the CLOSED state; it follows the arc from FIN_WAIT_1 to TIME_WAIT. We leave it as an exercise for you to figure out what combination of circumstances leads to this fourth possibility.

The main thing to recognize about connection teardown is that a connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

6.2.4 Sliding Window Revisited

We are now ready to discuss TCP's variant of the sliding window algorithm. As discussed in Section 3.5.2, the sliding window serves several purposes: (1) it guarantees the reliable delivery of data, (2) it ensures that data is delivered in order, and (3) it enforces flow control between the sender and the receiver. TCP's use of the sliding window algorithm is the same as we saw in Section 3.5.2 in the case of the first two of these three functions. Where TCP differs from the earlier algorithm is that it folds the flow-control function in as well. In particular, rather than having a fixed-sized sliding window, the receiver *advertises* a window size to the sender. This is done using the AdvertisedWindow field in the TCP header. The sender is then limited to having no more than a value of AdvertisedWindow bytes of unacknowledged data at any given time. The receiver selects a suitable value for AdvertisedWindow based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from overrunning the receiver's buffer. We discuss this at greater length below.

Reliable and Ordered Delivery

To see how the sending and receiving sides of TCP interact with each other to implement reliable and ordered delivery, consider the situation illustrated in Figure 6.7. TCP on the sending

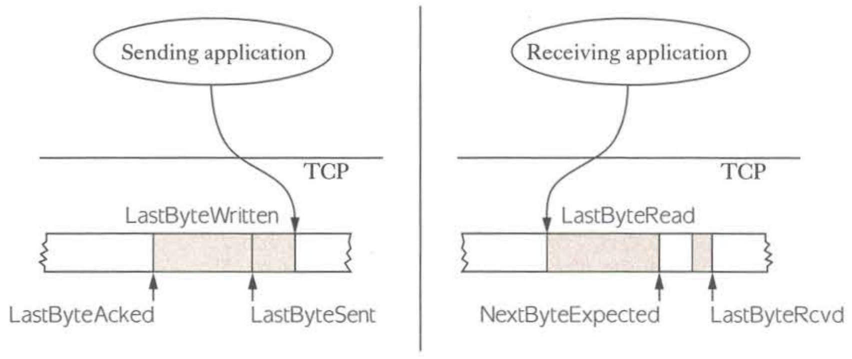

Figure 6.7 Relationship between TCP send buffer (left) and receive buffer (right).

side (pictured on the left) maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application, but not transmitted. On the receiving side, TCP maintains a receive buffer. This buffer holds data that arrives out of order, as well as data that is in the correct order (i.e., there are no missing bytes earlier in the stream) but that the application process has not yet had the chance to read.

To make the following discussion simpler to follow, we initially ignore the fact that both the buffers and the sequence numbers are of some finite size, and hence will eventually wrap around. Also, we do not distinguish between a pointer into a buffer where a particular byte of data is stored and the sequence number for that byte.

Looking first at the sending side, three pointers are maintained into the send buffer, each with an obvious meaning: LastByteAcked, LastByteSent, and LastByteWritten. Clearly,

$$\text{LastByteAcked} \leq \text{LastByteSent}$$

since the receiver cannot have acknowledged a byte that has not yet been sent, and

$$\text{LastByteSent} \leq \text{LastByteWritten}$$

since TCP cannot send a byte that the application process has not yet written. Also note that none of the bytes to the left of LastByteAcked need to be saved in the buffer because they have already been acknowledged, and none of the bytes to the right of LastByteWritten need to be buffered because they have not yet been generated.

A similar set of pointers (sequence numbers) are maintained on the receiving side: LastByteRead, NextByteExpected, and LastByteRcvd. The inequalities are a little less intuitive, however, because of the problem of out-of-order delivery. The first relationship

$$\text{LastByteRead} < \text{NextByteExpected}$$

is true because a byte cannot be read by the application until it is received *and* all preceding bytes have also been received. NextByteExpected points to the byte immediately after the latest byte to meet this criterion. Second, NextByteExpected $\leq$ LastByteRcvd + 1

since, if data has arrived in order, NextByteExpected points to the byte after NextByteExpected, whereas if data has arrived out of order, NextByteExpected points to the start of the first gap in the data, as in Figure 6.7. Note that bytes to the left of LastByteRead need not be buffered because they have already been read by the local application process, and bytes to the right of LastByteRcvd need not be buffered because they have not yet arrived.

Flow Control

Most of the above discussion is similar to that found in Section 3.5.2, the only real difference being that this time we elaborated on the fact that the sending and receiving application processes are filling and emptying their local buffer, respectively. (The earlier discussion glossed over the fact that data arriving from an upstream node was filling the send buffer, and data being transmitted to a downstream node was emptying the receive buffer).

You should make sure you understand this much before proceeding, because now comes the point where the two algorithms differ more significantly. In what follows, we reintroduce the fact that both buffers are of some finite size, denoted MaxSendBuffer and MaxRcvBuffer, although we don't worry about the details of how they are implemented. In other words, we are only interested in the number of bytes being buffered, not in where those bytes are actually stored.

Recall that in a sliding window protocol, the size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver. Thus, the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer. Observe that TCP on the receive side must keep

LastByteRcvd − NextByteRead $\leq$ MaxRcvBuffer

to avoid overflowing its buffer. It therefore advertises a window size of

AdvertisedWindow = MaxRcvBuffer − (LastByteRcvd − NextByteRead),

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding bytes have also arrived. In addition, LastByteRcvd moves to the right (is incremented), meaning that the advertised window potentially shrinks. Whether or not it shrinks depends on how fast the local application process is consuming data. If the local process is reading data just as fast as it arrives (causing NextByteRead to be incremented at the same rate as LastByteRcvd), then the advertised window stays open (i.e., AdvertisedWindow = MaxRcvBuffer). If, however, the receiving process falls behind, perhaps because it performs a very expensive operation on each byte of data that it reads, then the advertised window grows smaller with every segment that arrives, until it eventually goes to 0.

TCP on the send side must then adhere to the advertised window it gets from the receiver. This means that at any given time, it must ensure that

LastByteSent − LastByteAcked $\leq$ AdvertisedWindow.

Said another way, the sender computes an *effective* window that limits how much data it can send:

$$\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

Clearly, EffectiveWindow must be greater than 0 before the source can send more data. It is possible, therefore, that a segment arrives acknowledging x bytes, thereby allowing the sender to increment LastByteAcked by x, but because the receiving process was not reading any data, the advertised window is now x bytes smaller than the time before. In such a situation, the sender would be able to free buffer space, but not to send any more data.

All the while this is going on, the send side must also make sure that the local application process does not overflow the send buffer, that is, that

$$\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}.$$

If the sending process tries to write y bytes to TCP, but

$$(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer},$$

then TCP blocks the sending process and does not allow it to generate more data.

It is now possible to understand how a slow receiving process ultimately stops a fast sending process. First, the receive buffer fills up, which means the advertised window shrinks to 0. An advertised window of 0 means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As soon as the receiving process starts to read data again, the receive-side TCP is able to open its window back up, which allows the send-side TCP to transmit data out of its buffer. When this data is eventually acknowledged, LastByteAcked is incremented, the buffer space holding this acknowledged data becomes free, and the sending process is unblocked and allowed to proceed.

There is only one remaining detail that must be resolved—how does the sending side know that the advertised window is no longer 0? As mentioned above, TCP *always* sends a segment in response to a received data segment, and this response contains the latest values for the Acknowledge and AdvertisedWindow fields, even if these values have not changed since the last time they were sent. The problem is this. Once the receive side has advertised a window size of 0, the sender is not permitted to send any more data, which means it has no way to discover that the advertised window is no longer 0 at some time in the future. TCP on the receive side does not spontaneously send nondata segments; it only sends them in response to an arriving data segment.

TCP deals with this situation as follows. Whenever the other side advertises a window size of 0, the sending side persists in sending a segment with 1 byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments triggers a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a nonzero advertised window.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

Table 6.1 Time until 32-bit sequence number space wraps around.

Note that the reason the sending side periodically sends this probe segment is that TCP is designed to make the receive side as simple as possible—it simply responds to segments from the sender, and it never initiates any activity on its own. This is an example of a well-recognized (although not universally applied) protocol design rule, which, for lack of a better name, we call the *smart sender/dumb receiver* rule. Recall that we saw another example of this rule when we discussed the use of NAKs in Section 3.5.2.

Keeping the Pipe Full

We now turn our attention to the size of the SequenceNum and AdvertisedWindow fields and the implications of their sizes on TCP's correctness and performance. TCP's SequenceNum field is 32 bits long and its AdvertisedWindow field is 16 bits long, meaning that TCP has easily satisfied the requirement of the sliding window algorithm that the sequence number space be twice as big as the window size: $2^{32} >> 2 \times 2^{16}$. However, this requirement is not the interesting thing about these two fields. Consider each field in turn.

The relevance of the 32-bit sequence number space is that the sequence number used on a given connection might wrap around—a byte with sequence number x could be sent at one time, and then at a later time, a second byte with the same sequence number x might be sent. Once again, we assume that packets cannot survive in the Internet for longer than 60 seconds. Thus, we need to make sure that the sequence number does not wrap around within a 60-second period of time. Whether or not this happens depends on how fast data can be transmitted over the Internet, that is, how fast the 32-bit sequence number space can be consumed. (This discussion assumes that we are trying to consume the sequence number space as fast as possible, but of course we will be if we are doing our job of keeping the pipe full.) Table 6.1 shows how long it takes for the sequence number to wrap around on networks with various bandwidths.

As you can see, the 32-bit sequence number space is adequate for today's networks, but it won't be long (STS-12) until a larger sequence number space is needed. The IETF is already

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
FDDI (100 Mbps)	1.2 MB
STS-3 (155 Mbps)	1.8 MB
STS-12 (622 Mbps)	7.4 MB
STS-24 (1.2 Gbps)	14.8 MB

Table 6.2 Required window size for 100 ms RTT.

working on an extension to TCP that effectively extends the sequence number space to protect against the sequence number wrapping around.

The relevance of the 16-bit AdvertisedWindow field is that it must be big enough to allow the sender to keep the pipe full. Clearly, the receiver is free to not open the window as large as the AdvertisedWindow field allows; we are interested in the situation in which the receiver has enough buffer space to handle as much data as the largest possible AdvertisedWindow allows.

In this case, it is not just the network bandwidth but the delay × bandwidth product that dictates how big the AdvertisedWindow field needs to be—the window needs to be opened far enough to allow a full delay × bandwidth product's worth of data to be transmitted. Assuming an RTT of 100 ms (a typical number for a crosscountry connection in the U.S.), Table 6.2 gives the delay × bandwidth product for several network technologies.

As you can see, TCP's AdvertisedWindow field is in even worse shape than its SequenceNum field—it is not big enough to handle even a T3 connection across the continental U.S., since a 16-bit field allows us to advertise a window of only 64 KB. The very same TCP extension mentioned above provides a mechanism for effectively increasing the size of the advertised window.

6.2.5 Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We now describe this mechanism and how it has evolved over time as the Internet community has gained more experience using TCP.

Original Algorithm

We begin with a simple algorithm for computing a timeout value between a pair of hosts. This is the algorithm that was originally described in the TCP specification—and the following description presents it in those terms—but it could be used by any end-to-end protocol.

The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again, and then takes the difference between these two times as a **SampleRTT**. TCP then computes an **EstimatedRTT** as a weighted average between the previous estimate and this new sample. That is,

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + \beta \times \text{SampleRTT}$$

where

$$\alpha + \beta = 1.$$

The parameters α and β are selected to *smooth* the **EstimatedRTT**. A large β value tracks changes in the RTT but is perhaps too heavily influenced by temporary fluctuations. On the other hand, a large α value is more stable but perhaps not quick enough to adapt to real changes. The original TCP specification recommended a setting of α between 0.8 and 0.9 and β between 0.1 and 0.2. TCP then uses **EstmatedRTT** to compute the timeout in a rather conservative way:

$$\text{TimeOut} = 2 \times \text{EstimatedRTT}.$$

Karn/Partridge Algorithm

After several years of use on the Internet, a rather obvious flaw was discovered in this simple algorithm. The problem was that an ACK does not really acknowledge a transmission; it actually acknowledges the receipt of data. In other words, whenever a segment is retransmitted and then an ACK arrives at the sender, it is impossible to determine if this ACK should be associated with the first or the second transmission of the segment for the purpose of measuring the sample RTT. It is necessary to know which transmission to associate it with so as to compute an accurate **SampleRTT**. As illustrated in Figure 6.8, if you assume that the ACK is for the original transmission but it was really for the second, then the **SampleRTT** is too large (a), while if you assume that the ACK is for the second transmission but it was actually for the first, then the **SampleRTT** is too small (b).

The solution is surprisingly simple. Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures **SampleRTT** for segments that have been sent only once. This solution is known as the Karn/Partridge algorithm, after its inventors. Their proposed fix also includes a second small change to TCP's timeout mechanism. Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last **EstimatedRTT**. That is, Karn and Partridge proposed that TCP use exponential backoff, just as the Ethernet does.

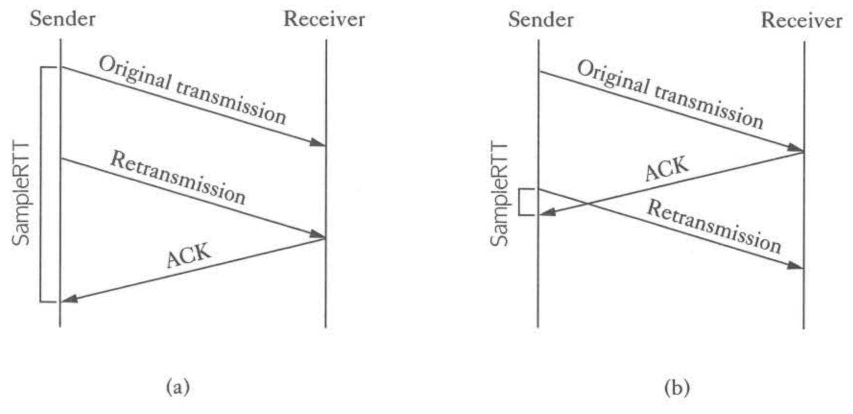

Figure 6.8 Associating the ACK with original transmission (a) versus retransmission (b).

Jacobson/Karels Algorithm

The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion. Their approach was designed to fix some of the causes of that congestion, and although it was an improvement, the congestion was not eliminated. A couple of years later, two other researchers—Jacobson and Karels—proposed a more drastic change to TCP to battle congestion. The bulk of that proposed change is described in Chapter 8. Here, we focus on the aspect of that proposal that is related to deciding when to timeout and retransmit a segment.

As an aside, it should be clear how the timeout mechanism is related to congestion—if you timeout too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network. As we will see in Chapter 8, the other reason for needing an accurate timeout value is that a timeout is taken to imply congestion, which triggers a congestion-control mechanism. Finally, note that there is nothing about the Jacobson/Karels timeout computation that is specific to TCP. It could be used by any end-to-end protocol.

The main problem with the original computation is that it does not take the variance of the sample RTTs into account. Intuitively, if the variation among samples is small, then the EstimatedRTT can be better trusted and there is no reason for multiplying this estimate by 2 to compute the timeout. On the other hand, a large variance in the samples suggests that the timeout value should not be too tightly coupled to the EstimatedRTT.

In the new approach, the sender measures a new SampleRTT as before. It then folds this new sample into the timeout calculation as follows:

$$\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$$
$$\text{EstimatedRTT} = \text{EstimatedRTT} + (\delta \times \text{Difference})$$
$$\text{Deviation} = \text{Deviation} + \delta\,(\,|\text{Difference}| - \text{Deviation})$$

where δ is a fraction between 0 and 1. That is, we calculate both the mean RTT and the variation in that mean.

TCP then computes the timeout value as a function of both EstimatedRTT and Deviation as follows:

$$\text{TimeOut} = \mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}$$

where based on experience, μ is typically set to 1 and ϕ is set to 4. Thus, when the variance is small, TimeOut is close to EstimatedRTT, while a large variance causes the Deviation term to dominate the calculation.

Implementation

There are two items of note regarding the implementation of timeouts in TCP. The first is that it is possible to implement the calculation for EstimatedRTT and Deviation without using floating-point arithmetic. Instead, the whole calculation is scaled by 2^n, with δ selected to be $1/2^n$. This allows us to do integer arithmetic, implementing multiplication and division using shifts, thereby achieving higher performance. The resulting calculation is given by the following code fragment, where $n = 3$ (i.e., $\delta = 1/8$). Note that EstimatedRTT and Deviation are stored in their scaled up forms, while the value of SampleRTT at the start of the code and of TimeOut at the end are real, unscaled values. If you find the code hard to follow, you might want to try plugging some real numbers into it and verifying that it gives the same results as the equations above.

```
{
    SampleRTT -= (EstimatedRTT >> 3);
    EstimatedRTT += SampleRTT;
    if (SampleRTT < 0)
        SampleRTT = -SampleRTT;
    SampleRTT -= (Deviation >> 3);
    Deviation += SampleRTT;
    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
}
```

The second point of note is that Jacobson and Karels's algorithm is only as good as the clock used to read the current time. On a typical Berkeley Unix implementation, the clock granularity is as large as 500 ms, which is significantly larger than the average crosscountry RTT of somewhere between 100 and 200 ms. To make matters worse, the Berkeley Unix implementation of TCP only checks to see if a timeout should happen every time this 500-ms clock ticks, and it only takes a sample of the round-trip time once per RTT. The combination of these two factors quite often means that a timeout happens 1 second after the segment was transmitted. Once again, the proposed extensions to TCP include a mechanism that makes this RTT calculation a bit more precise.

6.2.6 Record Boundaries

As mentioned earlier in this section, TCP is a byte-stream protocol. This means that the number of bytes written by the sender are not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side, the application reads 5 bytes at a time inside a loop that iterates 6 times. TCP does not interject record boundaries between the eighth and ninth bytes, nor between the tenth and eleventh bytes. This is in contrast to a message-oriented protocol, such as UDP, in which the message that is sent is exactly the same length as the message that is received.

Even though TCP is a byte-stream protocol, it has two different features that can be used by the sender to effectively insert record boundaries into this byte stream, thereby informing the receiver how to break the stream of bytes into records. (Being able to mark record boundaries is useful, for example, in many database applications.) Both of these features were originally included in TCP for completely different reasons; they have only come to be used for this purpose over time.

The first mechanism is the *push* operation. Originally, this mechanism was designed to allow the sending process to tell TCP that it should send whatever bytes it had collected to its peer. This was, and still is, used in terminal emulators like Telnet because each byte has to be sent as soon as it is typed. However, push can be used to implement record boundaries because the specification says that TCP should inform the receiving application that a push was performed; this is the reason for the PUSH flag in the TCP header. This act of informing the receiver of a push can be interpreted as marking a record boundary.

The second mechanism for inserting end-of-record markers into a byte stream is the urgent data feature, as implemented by the URG flag and the UrgPtr field in the TCP header. Originally, the urgent data mechanism was designed to allow the sending application to send *out-of-band* data to its peer. By "out of band" we mean data that is separate from the normal flow of data, e.g., a command to interrupt an operation already under way. This out-of-band data was identified in the segment using the UrgPtr field and was to be delivered to the receiving process as soon as it arrived, even if that meant delivering it before data with an earlier sequence number. Over time, however, this feature has not been used, so instead of signifying "urgent" data, it has come to be used to signify "special" data, such as a record marker. This use has developed because, as with the push operation, TCP on the receiving side must inform the application that "urgent data" has arrived. That is, the urgent data in itself is not important. It is the fact that the sending process can effectively send a signal to the receiver that is important.

Of course, the application program is always free to insert record boundaries without any assistance from TCP. For example, it can send a field that indicates the length of a record that is to follow, or it can insert its own record boundary markers into the data stream.

6.3 Remote Procedure Call

As discussed in Chapter 1, a common pattern of communication used by application programs is the request/reply paradigm, also called message transaction: a client sends a request message to a server, the server responds with a reply message, and the client blocks (suspends