Chapter 3: Transport Layer

We begin this chapter by examining the services that the transport layer can provide to network applications. The Internet provides applications a choice of two transport protocols, UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). We will see that UDP is a no-frills protocol, providing only minimal services to applications, whereas TCP is a more complex protocol, and offers a rich set of services, including reliable data transfer. We examine the fundamental principles of reliable data transfer, and study how a transport layer protocol can provide reliable data transfer even when the underlying network layer is unreliable. We also look at the fundamental principles of congestion control, which limits the amount of data an application can send into the network so as to prevent network grid lock. We study the TCP transport protocol in depth, and carefully examine how TCP provides reliable data transfer, flow control and congestion control.

Online Book

3.1: Transport-Layer Services and Principles

Residing between the application and network layers, the transport layer is a central piece of the layered network architecture. It has the critical role of providing communication services directly to the application processes running on different hosts. In this chapter we'll examine the possible services provided by a transport-layer protocol and the principles underlying various approaches toward providing these services. We'll also look at how these services are implemented and instantiated in existing protocols; as usual, particular emphasis will be given to the Internet protocols, namely, the TCP and UDP transport-layer protocols.

In the previous two chapters we touched on the role of the transport layer and the services that it provides. Let's quickly review what we have already learned about the transport layer:

A transport-layer protocol provides for **logical communication** between application processes running on different hosts. By logical communication, we mean that although the communicating application processes are not *physically* connected to each other (indeed, they may be on different sides of the planet, connected via numerous routers and a wide range of link types), from the applications' viewpoint, it is as if they were physically connected. Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages. Figure 3.1 illustrates the notion of logical communication.

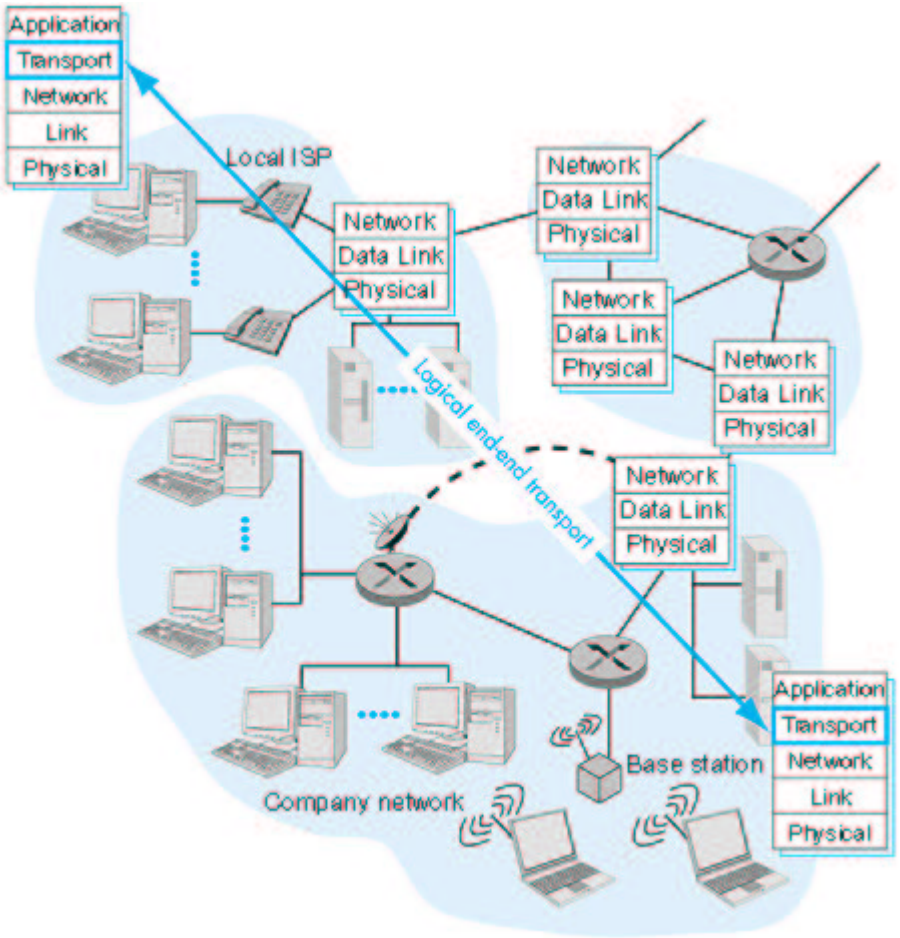


Figure 3.1: The transport layer provides logical rather than physical communication between applications.

As shown in Figure 3.1, transport-layer protocols are implemented in the end systems but not in network routers. Network routers only act on the network-layer fields of the layer-3 PDUs; they do not act on the transport-layer fields.

On the sending side, the transport layer converts the messages it receives from a sending application process into 4-PDUs (that is, transport-layer protocol data units). This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create 4-PDUs. The transport layer then passes the 4-PDUs to the network layer, where each 4-PDU is encapsulated into a 3-PDU. On the receiving side, the transport layer receives the 4-PDUs from the network layer, removes the transport header from the 4-PDUs, reassembles the messages, and passes them to a receiving application process.

A computer network can make more than one transport-layer protocol available to network applications. For example, the Internet has two protocols--TCP and UDP. Each of these protocols provides a different set of transport-layer services to the invoking application.

All transport-layer protocols provide an application multiplexing/demultiplexing service. This service will be described in detail in the next section. As discussed in Section 2.1, in addition to a multiplexing/demultiplexing service, a transport protocol can possibly provide other services to invoking applications, including reliable data transfer, bandwidth guarantees, and delay guarantees.

3.1.1: Relationship between Transport and Network Layers

The transport layer lies just above the network layer in the protocol stack. Whereas a transport-layer protocol provides *logical communication between processes* running on different hosts, a network-layer protocol provides *logical communication between hosts.* This distinction is subtle but important. Let's examine this distinction with the aid of a household analogy.

Consider two houses, one on the East Coast and the other on the West Coast, with each house being home to a dozen kids. The kids in the East Coast household are cousins of the kids in the West Coast household. The kids in the two households love to write to each other--each kid writes each cousin every week, with each letter delivered by the traditional postal service in a separate envelope. Thus, each household sends 144 letters to the other household every week. (These kids would save a lot of money if they had e-mail!) In each of the households there is one kid--Ann in the West Coast house and Bill in the East Coast house--responsible for mail collection and mail distribution. Each week Ann visits all her brothers and sisters, collects the mail, and gives the mail to a postal-service mail person who makes daily visits to the house. When letters arrive at the West Coast house, Ann also has the job of distributing the mail to her brothers and sisters. Bill has a similar job on the East Coast.

In this example, the postal service provides logical communication between the two houses--the postal service moves mail from house to house, not from person to person. On the other hand, Ann and Bill provide logical communication among the cousins--Ann and Bill pick up mail from and deliver mail to their brothers and sisters. Note that from the cousins' perspective, Ann and Bill *are* the mail service, even though Ann and Bill are only a part (the end system part) of the end-to-end delivery process. This household example serves as a nice analogy for explaining how the transport layer relates to the network layer:

> hosts (also called end systems) = houses
> processes = cousins
> application messages = letters in envelopes
> network-layer protocol = postal service (including mail persons)
> transport-layer protocol = Ann and Bill

Continuing with this analogy, observe that Ann and Bill do all their work within their respective homes; they are not involved, for example, in sorting

mail in any intermediate mail center or in moving mail from one mail center to another. Similarly, transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa; but it doesn't have any say about how the messages are moved within the network core. In fact, as illustrated in Figure 3.1, intermediate routers neither act on, nor recognize, any information that the transport layer may have appended to the application messages.

Continuing with our family saga, suppose now that when Ann and Bill go on vacation, another cousin pair--say, Susan and Harvey--substitute for them and provide the household-internal collection and delivery of mail. Unfortunately for the two families, Susan and Harvey do not do the collection and delivery in exactly the same way as Ann and Bill. Being younger kids, Susan and Harvey pick up and drop off the mail less frequently and occasionally lose letters (which are sometimes chewed up by the family dog). Thus, the cousin-pair Susan and Harvey do not provide the same set of services (that is, the same service model) as Ann and Bill. In an analogous manner, a computer network may make available multiple transport protocols, with each protocol offering a different service model to applications.

The possible services that Ann and Bill can provide are clearly constrained by the possible services that the postal service provides. For example, if the postal service doesn't provide a maximum bound on how long it can take to deliver mail between the two houses (for example, three days), then there is no way that Ann and Bill can guarantee a maximum delay for mail delivery between any of the cousin pairs. In a similar manner, the services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees for 4-PDUs sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for messages sent between processes.

Nevertheless, certain services *can* be offered by a transport protocol even when the underlying network protocol doesn't offer the corresponding service at the network layer. For example, as we'll see in this chapter, a transport protocol can offer reliable data transfer service to an application even when the underlying network protocol is unreliable, that is, even when the network protocol loses, garbles, and duplicates packets. As another example (which we'll explore in Chapter 7 when we discuss network security), a transport protocol can use encryption to guarantee that application messages are not read by intruders, even when the network layer cannot guarantee the secrecy of 4-PDUs.

3.1.2: Overview of the Transport Layer in the Internet

Recall that the Internet, and more generally a TCP/IP network, makes available two distinct transport-layer protocols to the application layer. One of these protocols is **UDP** (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application. The second

of these protocols is **TCP** (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking application. When designing a network application, the application developer must specify one of these two transport protocols. As we saw in Sections 2.6 and 2.7, the application developer selects between UDP and TCP when creating sockets.

To simplify terminology, when in an Internet context, we refer to the 4-PDU as a **segment.** We mention, however, that the Internet literature (for example, the RFCs) also refers to the PDU for TCP as a segment but often refers to the PDU for UDP as a **datagram.** But this same Internet literature also uses the terminology datagram for the network-layer PDU! For an introductory book on computer networking such as this, we believe that it is less confusing to refer to both TCP and UDP PDUs as segments, and reserve the terminology datagram for the network-layer PDU.

Before preceding with our brief introduction of UDP and TCP, it is useful to say a few words about the Internet's network layer. (The network layer is examined in detail in Chapter 4.) The Internet's network-layer protocol has a name--IP, for Internet Protocol. IP provides logical communication between hosts. The IP service model is a **best-effort delivery service.** This means that IP makes its "best effort" to deliver segments between communicating hosts, *but it makes no guarantees.* In particular, it does not guarantee segment delivery, it does not guarantee orderly delivery of segments, and it does not guarantee the integrity of the data in the segments. For these reasons, IP is said to be an **unreliable service.** We also mention here that every host has an IP address. We will examine IP addressing in detail in Chapter 4; for this chapter we need only keep in mind that each host has a *unique* IP address.

Having taken a glimpse at the IP service model, let's now summarize the service model of UDP and TCP. The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems. Extending host-to-host delivery to process-to-process delivery is called **application multiplexing** and **demultiplexing.** We'll discuss application multiplexing and demultiplexing in the next section. UDP and TCP also provide integrity checking by including error-detection fields in their headers. These two minimal transport-layer services--process-to-process data delivery and error checking--are the only two services that UDP provides! In particular, like IP, UDP is an unreliable service--it does not guarantee that data sent by one process will arrive intact to the destination process. UDP is discussed in detail in Section 3.3.

TCP, on the other hand, offers several additional services to applications. First and foremost, it provides **reliable data transfer.** Using flow control, sequence numbers, acknowledgments, and timers (techniques we'll explore in detail in this chapter), TCP ensures that data is delivered from sending process to receiving process, correctly and in order. TCP thus converts IP's unreliable service between end systems into a reliable data transport

service between processes. TCP also uses **congestion control.** Congestion control is not so much a service provided to the invoking application as it is a service for the Internet as a whole, a service for the general good. In loose terms, TCP congestion control prevents any one TCP connection from swamping the links and switches between communicating hosts with an excessive amount of traffic. In principle, TCP permits TCP connections traversing a congested network link to equally share that link's bandwidth. This is done by regulating the rate at which the sending-side TCPs can send traffic into the network. UDP traffic, on the other hand, is unregulated. An application using UDP transport can send at any rate it pleases, for as long as it pleases.
A protocol that provides reliable data transfer and congestion control is necessarily complex. We will need several sections to cover the principles of reliable data transfer and congestion control, and additional sections to cover the TCP protocol itself. These topics are investigated in Sections 3.4 through 3.8. The approach taken in this chapter is to alternate between basic principles and the TCP protocol. For example, we first discuss reliable data transfer in a general setting and then discuss how TCP specifically provides reliable data transfer. Similarly, we first discuss congestion control in a general setting and then discuss how TCP uses congestion control. But before getting into all this good stuff, let's first look at application multiplexing and demultiplexing in the next section.

3.2: Multiplexing and Demultiplexing Apps

In this section we discuss multiplexing and demultiplexing network applications. In order to keep the discussion concrete, we'll discuss this basic transport-layer service in the context of the Internet. We emphasize, however, that a multiplexing/demultiplexing service is needed for all computer networks.

Although the multiplexing/demultiplexing service is not among the most ex citing services that can be provided by a transport-layer protocol, it is absolutely critical. To understand why it is so critical, consider the fact that IP delivers data between two end systems, with each end system identified with a unique IP address. IP does *not* deliver data between the application processes that run on these end systems. Extending host-to-host delivery to process-to-process delivery is the job of application multiplexing and demultiplexing.

At the destination host, the transport layer receives segments (that is,

transport-layer PDUs) from the network layer just below. The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host. Let's take a look at an example. Suppose you are sitting in front of your computer, and you are downloading Web pages while running one FTP session and two Telnet sessions. You therefore have four network application processes running--two Telnet processes, one FTP process, and one HTTP process. When the transport-layer in your computer receives data from the network layer below, it needs to direct the received data to one of these four processes. Let's now examine how this is done.

Each transport-layer segment has a set of fields that determine the process to which the segment's data is to be delivered. At the receiving end, the transport layer can then examine these fields to determine the receiving process and then direct the segment to that process. This job of delivering the data in a transport-layer segment to the correct application process is called **demultiplexing.** The job of gathering data at the source host from different application processes, enveloping the data with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called **multiplexing.** Multiplexing and demultiplexing are illustrated in Figure 3.2.

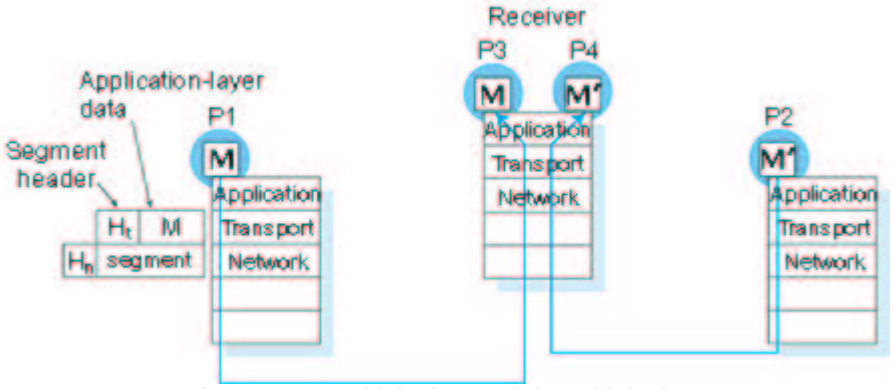


Figure 3.2: Multiplexing and demultiplexing

To illustrate the demultiplexing job, let us return to the household saga in the previous section. Each of the kids is distinguished by his or her name. When Bill receives a batch of mail from the mail person, he performs a demultiplexing operation by observing to whom the letters are addressed and then hand delivering the mail to his brothers and sisters. Ann performs a multiplexing operation when she collects letters from her brothers and sisters and gives the collected mail to the mail person.

UDP and TCP perform the demultiplexing and multiplexing jobs by including two special fields in the segment headers: the **source port-number field** and the **destination port-number field.** These two fields are illustrated in Figure 3.3. When taken together, the fields uniquely identify an application process running on the destination host. (The UDP and TCP segments have other fields as well, and they will be addressed in the

subsequent sections of this chapter.)

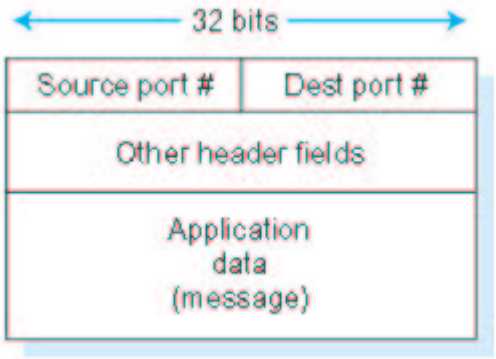


Figure 3.3: Source and destination port-number fields in a transport layer segment

The notion of port numbers was briefly introduced in Sections 2.6-2.7, in which we studied application development and socket programming. The port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called **well-known port numbers** and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP and FTP. HTTP uses port number 80; FTP uses port number 21. The list of well-known port numbers is given in RFC 1700. When we develop a new application (such as one of the applications developed in Sections 2.6-2.8), we must assign the application a port number.

Given that each *type* of application running on an end system has a unique port number, then why is it that the transport-layer segment has fields for two port numbers, a source port number and a destination port number? The answer is simple: An end system may be running two processes of the same type at the same time, and thus the port destination number of an application may not suffice to identify a specific process. For example, a Web server may spawn a new HTTP process for every request it processes; whenever such a Web server is servicing more than one request (which is by no means uncommon), the server is running more than one process with port number 80. Therefore, in order to uniquely identify the process to which data is destined, a second port number is needed.

How is this second port number created? Which port number goes in the source port-number field of a segment? Which goes in the destination port-number field of a segment? To answer these questions, recall from Section 2.1 that networking applications are organized around the client/server model. Typically, the host that initiates the application is the client and the other host is the server. Now let us look at a specific example. Suppose the application has port number 23 (the port number for Telnet). Consider a transport-layer segment leaving the client (that is, the host that initiated the Telnet session) and destined for the server. What are the destination and source port numbers for this segment? For the destination port number, this segment has the port number of the application, namely, 23. For the source

port number, the client uses a number that is not being used by any other host processes. (This is done automatically by the transport-layer software running on the client and is transparent to the application developer.) Let's say the client chooses port number *x*. Then each segment that this process sends to the Telnet server will have its source port number set to *x* and destination port number set to 23. When the segment arrives at the server, the source and destination port numbers in the segment enable the server host to pass the segment's data to the correct application process. The destination port number 23 identifies a Telnet process and the source port number *x* identifies the specific Telnet process.

The situation is reversed for the segments flowing from the server to the client. The source port number is now the application port number, 23. The destination port number is now *x* (the *same x* used for the source port number for the segments sent from client to server). When a segment arrives at the client, the source and destination port numbers in the segment will enable the client host to pass the data of the segment to the correct application process, which is identified by the port number pair. Figure 3.4 summarizes the discussion.

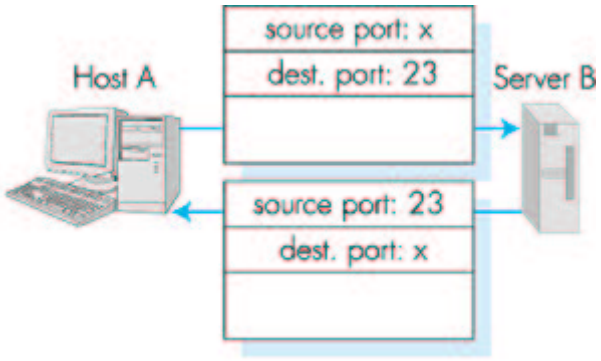


Figure 3.4: Use of source and destination port numbers in a client/server application

Now you may be wondering, what happens if two different clients establish a session to a server and each of these clients choose the same source port number *x*? This might well happen with a busy www server that handles many Web clients simultaneously. How will the server be able to demultiplex the segments when the two sessions have exactly the same port-number pair? The answer to this question is that the server also uses the IP addresses in the IP datagrams carrying these segments. (We will discuss IP datagrams and addressing in detail in Chapter 4.) The situation is illustrated in Figure 3.5, in which host C initiates two HTTP sessions to server B, and host A initiates one HTTP session to B. Hosts A, C, and server B each have their own unique IP address, A, C, and B, respectively. Host C assigns two different source port (SP) numbers (*x* and *y*) to the two HTTP connections emanating from host A. But because host A is choosing source port numbers independently from C, it can also assign SP = *x* to its HTTP connection. Nevertheless, server B is still able to correctly demultiplex the two connections since the two connections have different

source IP addresses. In summary, we see that when a destination host receives data from the network layer, the triplet (source IP address, source port number, destination port number) is used to forward the data to the appropriate process.

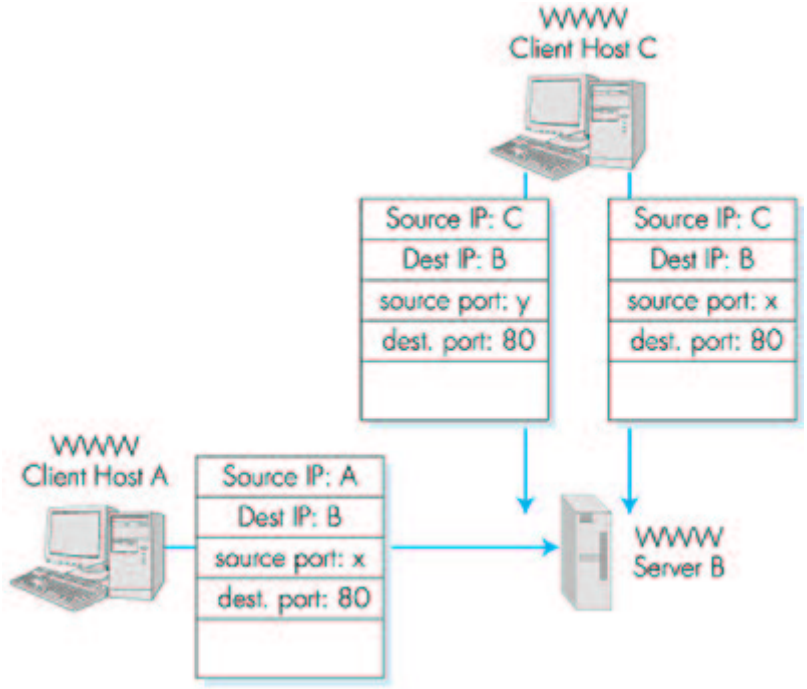


Figure 3.5: Two clients, using the same port numbers to communicate with the same server application

Now that we've shown how the transport layer can multiplex and demultiplex network applications, let's move on and discuss one of the Internet's transport protocols, UDP. In the next section we'll see that UDP adds little more to the network-layer protocol than a multiplexing/demultiplexing service.

Online Book

3.3: Connectionless Transport: UDP

In this section, we take a close look at UDP, how UDP works, and what it does. The reader is encouraged to refer back to material in Section 2.1, which includes an overview of the UDP service model, and to the material in Section 2.7, which discusses socket programming over UDP.

To motivate our discussion about UDP, suppose you were interested in designing a no-frills, bare-bones transport protocol. How might you go

about doing this? You might first consider using a vacuous transport protocol. In particular, on the sending side, you might consider taking the messages from the application process and passing them directly to the network layer; and on the receiving side, you might consider taking the messages arriving from the network layer and passing them directly to the application process. But as we learned in the previous section, we have to do a little more than nothing. At the very least, the transport layer has to provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct process.

UDP, defined in RFC 768, does just about as little as a transport protocol can do. Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP. In fact, if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the port numbers and the IP destination address to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be *connectionless.*

DNS is an example of an application-layer protocol that uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to a UDP socket (see Section 2.7). Without performing any handshaking, UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply (possibly because the underlying network lost the query or the reply), it either tries sending the query to another nameserver, or it informs the invoking application that it can't get a reply. We mention that the DNS specification permits DNS to run over TCP instead of UDP; in practice, however, DNS almost always runs over UDP.

Now you might be wondering why an application developer would ever choose to build an application over UDP rather than over TCP. Isn't TCP always preferable to UDP since TCP provides a reliable data transfer service and UDP does not? The answer is no, as many applications are better suited for UDP for the following reasons:

- *No connection establishment.* As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just

blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP--DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text. But, as we briefly discussed in Section 2.2, the TCP connection-establishment delay in HTTP is an important contributor to the "world wide wait."

- *No connection state.* TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. We will see in Section 3.5 that this state information is needed to implement TCP's reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

- *Small packet header overhead.* The TCP segment has 20 bytes of header overhead in every segment, whereas UDP only has 8 bytes of overhead.

- *Unregulated send rate.* TCP has a congestion control mechanism that throttles the sender when one or more links between sender and receiver become excessively congested. This throttling can have a severe impact on real-time applications, which can tolerate some packet loss but require a minimum send rate. On the other hand, the speed at which UDP sends data is only constrained by the rate at which the application generates data, the capabilities of the source (CPU, clock rate, and so on) and the access bandwidth to the Internet. We should keep in mind, however, that the receiving host does not necessarily receive all the data. When the network is congested, some of the data could be lost due to router buffer overflow. Thus, the receive rate can be limited by network congestion even if the sending rate is not constrained.

Figure 3.6 lists popular Internet applications and the transport protocols that they use. As we expect, e-mail, remote terminal access, the Web, and file transfer run over TCP--all these applications need the reliable data transfer service of TCP. Nevertheless, many important applications run over UDP rather than TCP. UDP is used for RIP routing table updates (see Chapter 4 on the network layer), because the updates are sent periodically (typically every five minutes), so that lost updates are replaced by more recent updates. UDP is used to carry network management (SNMP; see Chapter 8) data. UDP is preferred to TCP in this case, since network management applications must often run when the network is in a stressed state--

precisely when reliable, congestion-controlled data transfer is difficult to achieve. Also, as we mentioned earlier, DNS runs over UDP, thereby avoiding TCP's connection-establishment delays.

Application	**Application-layer protocol**	**Underlying transport protocol**
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	typically UDP
Streaming multimedia	proprietary	typically UDP
Internet telephony	proprietary	typically UDP
Network management	SNMP	typically UDP
Routing protocol	RIP	typically UDP
Name translation	DNS	typically UDP

Figure 3.6: Popular Internet applications and their underlying transport protocols

As shown in Figure 3.6, UDP is also commonly used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video. We'll take a close look at these applications in Chapter 6. We just mention now that all of these applications can tolerate a small fraction of packet loss, so that reliable data transfer is

not absolutely critical for the success of the application. Furthermore, real-time applications, like Internet phone and video conferencing, react very poorly to TCP's congestion control. For these reasons, developers of multimedia applications often choose to run their applications over UDP instead of TCP. Finally, because TCP cannot be employed with multicast, multicast applications run over UDP.

Although commonly done today, running multimedia applications over UDP is controversial to say the least. As we mentioned above, UDP has no congestion control. But congestion control is needed to prevent the network from entering a state in which very little useful work is done. If everyone were to start streaming high-bit-rate video without using any congestion control, there would be so much packet overflow at routers that no one would see anything. Thus, the lack of congestion control in UDP is a potentially serious problem [Floyd 1999]. Many researchers have proposed new mechanisms to force all sources, including UDP sources, to perform adaptive congestion control [Mahdavi 1997; Floyd 2000].

Before discussing the UDP segment structure, we mention that it is possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms, such as those we shall study in the next section). But this is a nontrivial task that would keep an application developer busy debugging for a long time. Nevertheless, building reliability directly into the application allows the application to "have its cake and eat it too." That is, application processes can communicate reliably without having to succumb to the transmission-rate constraints imposed by TCP's congestion-control mechanism. Many of today's proprietary streaming applications do just this--they run over UDP, but they have built acknowledgments and retransmissions into the application in order to reduce packet loss; see, for example, [Rhee 1998].

3.3.1: UDP Segment Structure

The UDP segment structure, shown in Figure 3.7, is defined in RFC 768. The application data occupies the data field of the UDP datagram. For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field. The UDP header has only four fields, each consisting of two bytes. As discussed in the previous section, the port numbers allow the destination host to pass the application data to the correct process running on the destination (that is, the demultiplexing function). The checksum is used by the receiving host to check if errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment. But we ignore this detail in order to see the forest through the trees. We shall discuss the checksum calculation below. Basic principles of error detection are described in Section 5.1. The length field specifies the length of the UDP segment, including the header, in bytes.

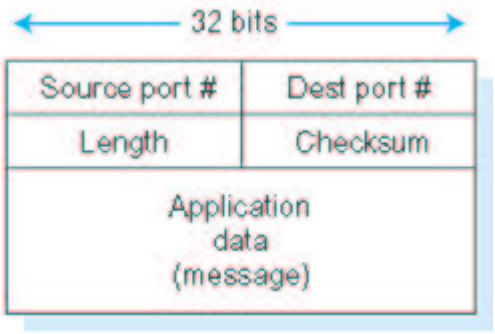


Figure 3.7: UDP segment structure

3.3.2: UDP Checksum

The UDP checksum provides for error detection. UDP at the sender side performs the one's complement of the sum of all the 16-bit words in the segment. This result is put in the checksum field of the UDP segment. Here we give a simple example of the checksum calculation. You can find details about efficient implementation of the calculation in the RFC 1071 and performance over real data in [Stone 1998 and Stone 2000]. As an example, suppose that we have the following three 16-bit words:

0110011001100110
0101010101010101
0000111100001111

The sum of first of these 16-bit words is

0110011001100110
0101010101010101
1011101110111011

Adding the third word to the above sum gives

1011101110111011
0000111100001111
1100101011001010

The 1's complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1's complement of the sum 1100101011001010 is 0011010100110101, which becomes the checksum. At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a zero, then we know that errors have been introduced into the packet.

You may wonder why UDP provides a checksum in the first place, as many link-layer protocols (including the popular Ethernet protocol) also provide error checking. The reason is that there is no guarantee that all the links between source and destination provide error checking--one of the links may use a protocol that does not provide error checking. Because IP is supposed to run over just about any layer-2 protocol, it is useful for the

transport layer to provide error checking as a safety measure. Although UDP provides error checking, it does not do anything to recover from an error. Some implementations of UDP simply discard the damaged segment; others pass the damaged segment to the application with a warning.
That wraps up our discussion of UDP. We will soon see that TCP offers reliable data transfer to its applications as well as other services that UDP doesn't offer. Naturally, TCP is also more complex than UDP. Before discussing TCP, however, it will be useful to step back and first discuss the underlying principles of reliable data transfer, which we do in the subsequent section. We will then explore TCP in Section 3.5, where we will see that TCP has its foundations in these underlying principles.

Online Book

3.4: Principles of Reliable Data Transfer

In this section, we consider the problem of reliable data transfer in a general context. This is appropriate since the problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well. The general problem is thus of central importance to networking. Indeed, if one had to identify a "top-10" list of fundamentally important problems in all of networking, this would be a top candidate to lead that list. In the next section we will examine TCP and show, in particular, that TCP exploits many of the principles that we are about to describe.

Figure 3.8 illustrates the framework for our study of reliable data transfer. The service abstraction provided to the upper layer entities is that of a reliable channel through which data can be transferred. With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.

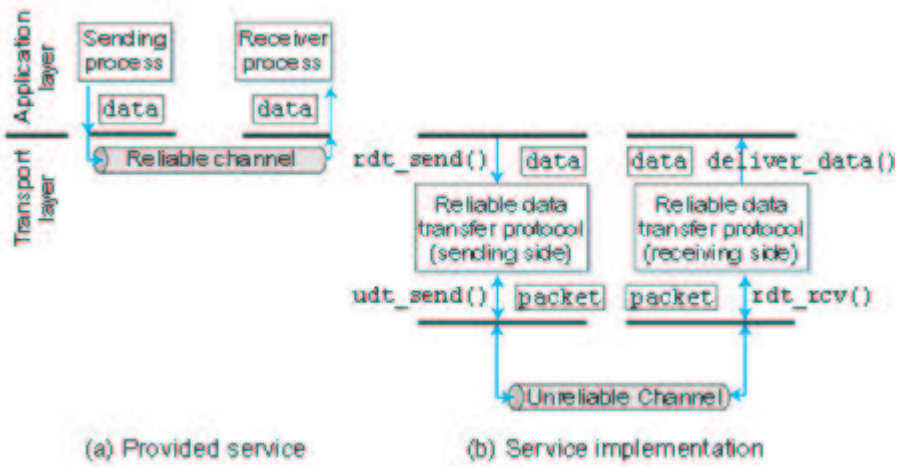


Figure 3.8: Reliable data transfer: Service model and service implementation

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that the layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-end network layer. More generally, the layer beneath the two reliably communicating endpoints might consist of a single physical link (for example, as in the case of a link-level data transfer protocol) or a global internetwork (for example, as in the case of a transport-level protocol). For our purposes, however, we can view this lower layer simply as an unreliable point-to-point channel.

In this section, we will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel. Figure 3.8(b) illustrates the interfaces for our data transfer protocol. The sending side of the data transfer protocol will be invoked from above by a call to `rdt_send()`. It will be passed the data to be delivered to the upper layer at the receiving side. (Here `rdt` stands for "reliable data transfer" protocol and `_send` indicates that the sending side of `rdt` is being called. The first step in developing any protocol is to choose a good name!) On the receiving side, `rdt_rcv()` will be called when a packet arrives from the receiving side of the channel. When the `rdt` protocol wants to deliver data to the upper layer, it will do so by calling `deliver_data()`. In the following we use the terminology "packet" rather than "segment" for the protocol data unit. Because the theory developed in this section applies to computer networks in general and not just to the Internet transport layer, the generic term "packet" is perhaps more appropriate here.

In this section we consider only the case of **unidirectional** data transfer, that is, data transfer from the sending to receiving side. The case of reliable **bidirectional** (that is, full duplex) data transfer is conceptually no more difficult but considerably more tedious to explain. Although we consider only unidirectional data transfer, it is important to note that the sending and

receiving sides of our protocol will nonetheless need to transmit packets in *both* directions, as indicated in Figure 3.8. We will see shortly that, in addition to exchanging packets containing the data to be transferred, the sending and receiving sides of rdt will also need to exchange control packets back and forth. Both the send and receive sides of rdt send packets to the other side by a call to udt_send() (where udt stands for unreliable data transfer).

3.4.1: Building a Reliable Data-Transfer Protocol

We now step through a series of protocols, each one becoming more complex, arriving at a flawless reliable data transfer protocol.

Reliable Data Transfer Over a Perfectly Reliable Channel: rdt1.0

We first consider the simplest case in which the underlying channel is completely reliable. The protocol itself, which we'll call rdt1.0, is trivial. The **finite-state machine (FSM)** definitions for the rdt1.0 sender and receiver are shown in Figure 3.9. The sender and receiver FSMs in Figure 3.9 each have just one state. The arrows in the FSM description indicate the transition of the protocol from one state to another. (Since each FSM in Figure 3.9 has just one state, a transition is necessarily from the one state back to itself; we'll see more complicated state diagrams shortly.) The event causing the transition is shown above the horizontal line labeling the transition, and the action(s) taken when the event occurs are shown below the horizontal line.

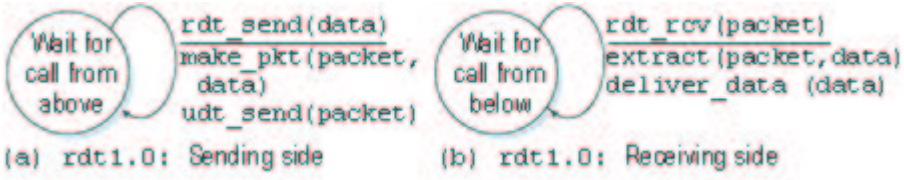


Figure 3.9: rdt1.0--A protocol for a completely reliable channel

The sending side of rdt simply accepts data from the upper layer via the rdt_send(data) event, puts the data into a packet (via the action make_pkt (packet,data)) and sends the packet into the channel. In practice, the rdt_send(data) event would result from a procedure call (for example, to rdt_send()) by the upper-layer application.

On the receiving side, rdt receives a packet from the underlying channel via the rdt_rcv(packet) event, removes the data from the packet (via the action extract(packet,data)) and passes the data up to the upper layer. In practice, the rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lower-layer protocol.

In this simple protocol, there is no difference between a unit of data and a packet. Also, all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong! Note that we have also assumed that the receiver is able to receive data as fast as the sender

happens to send data. Thus, there is no need for the receiver to ask the sender to "slow down!"

Reliable Data Transfer Over a Channel with Bit Errors: rdt2.0

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted. Such bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered. We'll continue to assume for the moment that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

Before developing a protocol for reliably communicating over such a channel, first consider how people might deal with such a situation. Consider how you yourself might dictate a long message over the phone. In a typical scenario, the message taker might say "OK" after each sentence has been heard, understood, and recorded. If the message taker hears a garbled sentence, you're asked to repeat the garbled sentence. This message-dictation protocol uses both **positive acknowledgments** ("OK") and **negative acknowledgments** ("Please repeat that."). These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating. In a computer network setting, reliable data transfer protocols based on such retransmission are known **ARQ** (Automatic Repeat reQuest) **protocols.**

Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

- *Error detection.* First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. Recall from the previous section that UDP uses the Internet checksum field for exactly this purpose. In Chapter 5 we'll examine error detection and correction techniques in greater detail; these techniques allow the receiver to detect and possibly correct packet bit errors. For now, we need only know that these techniques require that extra bits (beyond the bits of original data to be transferred) be sent from the sender to receiver; these bits will be gathered into the packet checksum field of the `rdt2.0` data packet.

- *Receiver feedback.* Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world (in this case, whether or not a packet was received correctly) is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message dictation scenario are examples of such feedback. Our `rdt2.0` protocol will similarly send ACK and NAK packets back from the receiver to the sender. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

- *Retransmission.* A packet that is received in error at the receiver will be retransmitted by the sender.

Figure 3.10 shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

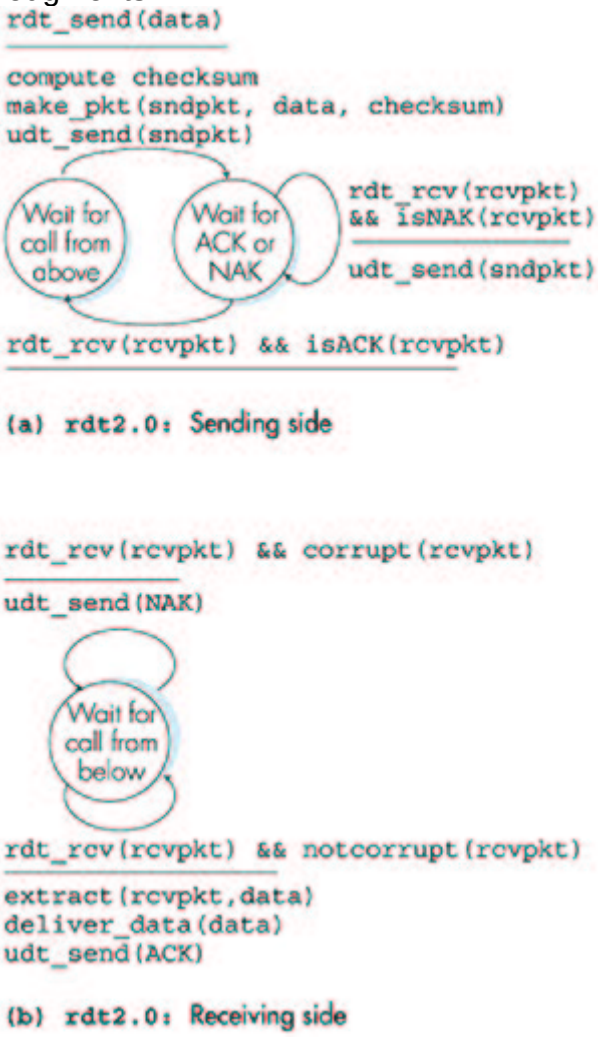


Figure 3.10: rdt2.0--A protocol for a channel with bit errors

The send side of rdt2.0 has two states. In one state, the send-side protocol is waiting for data to be passed down from the upper layer. In the other state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation rdt_rcv(rcvpkt) && isACK (rcvpkt) in Figure 3.10 corresponds to this event), the sender knows the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer. If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet. It is important to note that when the receiver is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that will

only happen after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as rdt2.0 are known as **stop-and-wait** protocols.

The receiver-side FSM for rdt2.0 still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted. In Figure 3.10, the notation rdt_rcv(rcvpkt) && corrupt(rcvpkt) corresponds to the event in which a packet is received and is found to be in error.

Protocol rdt2.0 may look as if it works but, unfortunately, it has a fatal flaw. In particular, we haven't accounted for the possibility that the ACK or NAK packet could be corrupted! (Before proceeding on, you should think about how this problem may be fixed.) Unfortunately, our slight oversight is not as innocuous as it may seem. Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors. The more difficult question is how the protocol should recover from errors in ACK or NAK packets. The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data.

Consider three possibilities for handling corrupted ACKs or NAKs:

- For the first possibility, consider what a human might do in the message dictation scenario. If the speaker didn't understand the "OK" or "Please repeat that" reply from the receiver, the speaker would probably ask "What did you say?" (thus introducing a new type of sender-to-receiver packet to our protocol). The speaker would then repeat the reply. But what if the speaker's "What did you say?" is corrupted? The receiver, having no idea whether the garbled sentence was part of the dictation or a request to repeat the last reply, would probably then respond with "What did *you* say?" And then, of course, that response might be garbled. Clearly, we're heading down a difficult path.

- A second alternative is to add enough checksum bits to allow the sender to not only detect, but to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.

- A third approach is for the sender to simply resend the current data packet when it receives a garbled ACK or NAK packet. This method, however, introduces **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know a priori whether an arriving packet contains new data or is a retransmission!

A simple solution to this new problem (and one adopted in almost all existing data-transfer protocols including TCP) is to add a new field to the

data packet and have the sender number its data packets by putting a **sequence number** into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission. For this simple case of a stop-and-wait protocol, a one-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet (the sequence number changes, moving "forward" in modulo-2 arithmetic). Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging. The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

Figures 3.11 and 3.12 show the FSM description for rdt2.1, our fixed version of rdt2.0. The rdt2.1 sender and receiver FSM's each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1. Note that the actions in those states where a 0-numbered packet is being sent or expected are mirror images of those where a 1-numbered packet is being sent or expected; the only differences have to do with the handling of the sequence number.

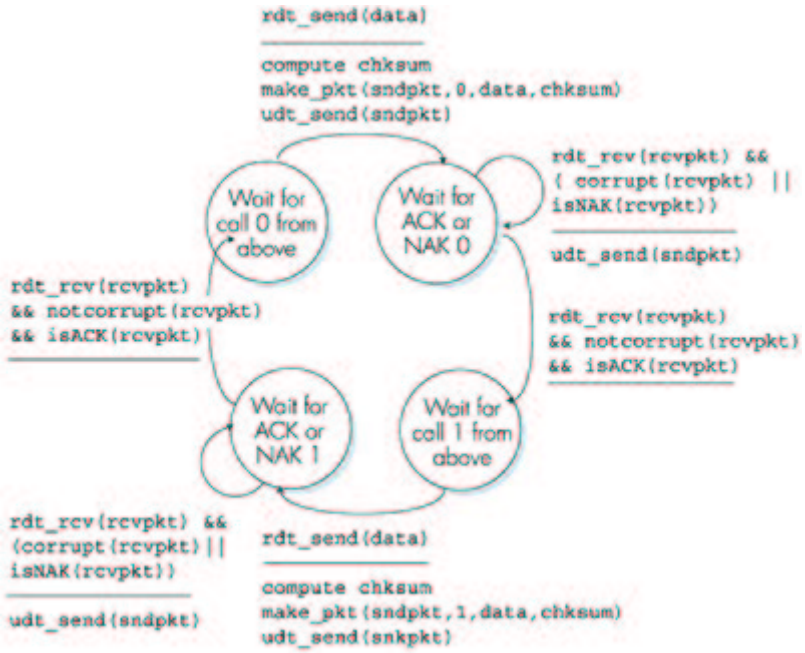


Figure 3.11: rdt2.1 sender

Figure 3.12: rdt2.1 receiver

Protocol rdt2.1 uses both positive and negative acknowledgments from the receiver to the sender. A negative acknowledgment is sent whenever a corrupted packet or an out-of-order packet is received. We can accomplish the same effect as a NAK if, instead of sending a NAK, we instead send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet (that is, receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Many TCP implementations use the receipt of so-called "triple duplicate ACKs" (three ACK packets all acknowledging the same already-acknowledged packet) to trigger a retransmission at the sender. Our NAK-free reliable data transfer protocol for a channel with bit errors is rdt2.2, shown in Figures 3.13 and 3.14.

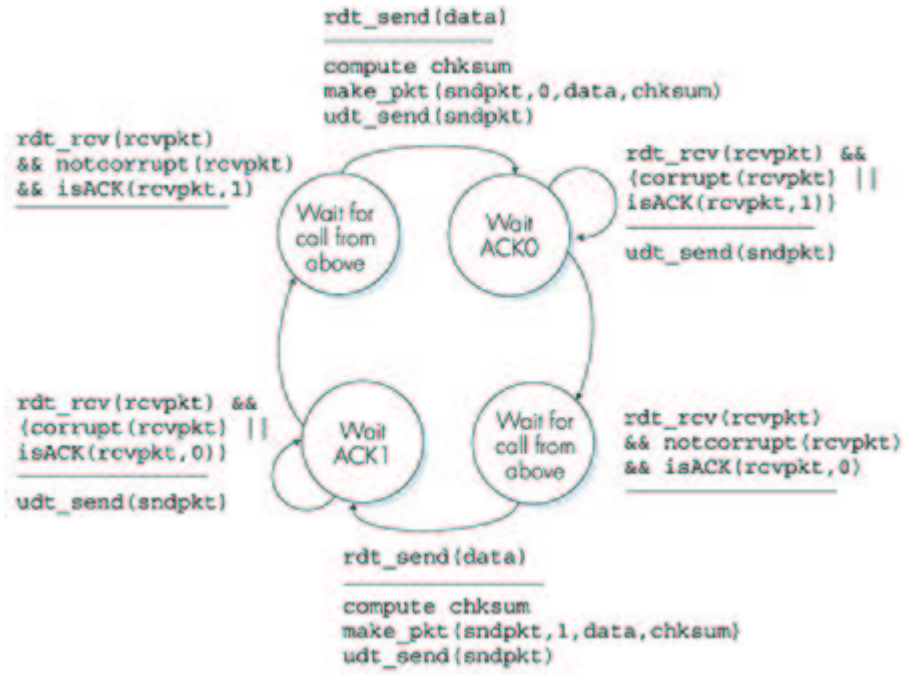


Figure 3.13: rdt2.2 sender

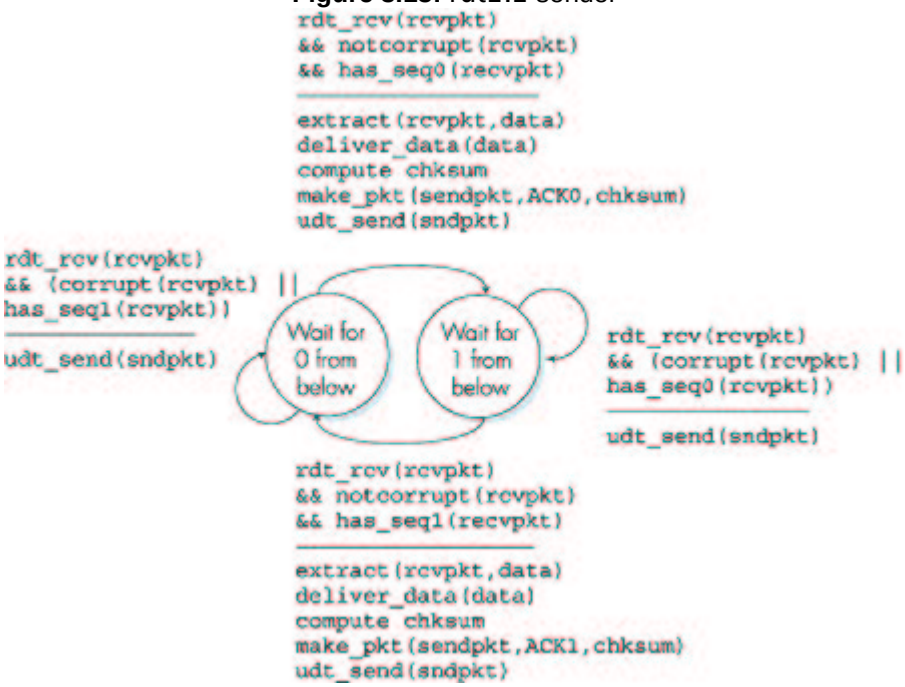


Figure 3.14: rdt2.2 receiver

Reliable Data Transfer Over a Lossy Channel with Bit Errors: rdt3.0

Suppose now that in addition to corrupting bits, the underlying channel can *lose* packets as well, a not-uncommon event in today's computer networks (including the Internet). Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions--the techniques already developed in rdt2.2--will allow us to answer the latter concern. Handling the first concern will require adding a

new protocol mechanism.

There are many possible approaches toward dealing with packet loss (several more of which are explored in the exercises at the end of the chapter). Here, we'll put the burden of detecting and recovering from lost packets on the sender. Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is *certain* that a packet has been lost, it can simply retransmit the data packet. You should convince yourself that this protocol does indeed work.

But how long must the sender wait to be certain that something has been lost? The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers or gateways) plus whatever amount of time is needed to process a packet at the receiver. In many networks, this worst-case maximum delay is very difficult to even estimate, much less know with certainty. Moreover, the protocol should ideally recover from packet loss as soon as possible; waiting for a worst-case delay could mean a long wait until error recovery is initiated. The approach thus adopted in practice is for the sender to "judiciously" choose a time value such that packet loss is likely, although not guaranteed, to have happened. If an ACK is not received within this time, the packet is retransmitted. Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost. This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Happily, protocol `rdt2.2` already has enough functionality (that is, sequence numbers) to handle the case of duplicate packets.

From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. In all cases, the action is the same: retransmit. In order to implement a time-based retransmission mechanism, a **countdown timer** will be needed that can interrupt the sender after a given amount of timer has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet, or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

The existence of sender-generated duplicate packets and packet (data, ACK) loss also complicates the sender's processing of any ACK packet it receives. If an ACK is received, how is the sender to know if it was sent by the receiver in response to its (sender's) own most recently transmitted packet, or is a delayed ACK sent in response to an earlier transmission of a different data packet? The solution to this dilemma is to augment the ACK packet with an **acknowledgment field.** When the receiver generates an ACK, it will copy the sequence number of the data packet being acknowledged into this acknowledgment field. By examining the contents of the acknowledgment field, the sender can determine the sequence number

of the packet being positively acknowledged.

Figure 3.15 shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets. Figure 3.16 shows how the protocol operates with no lost or delayed packets, and how it handles lost data packets.

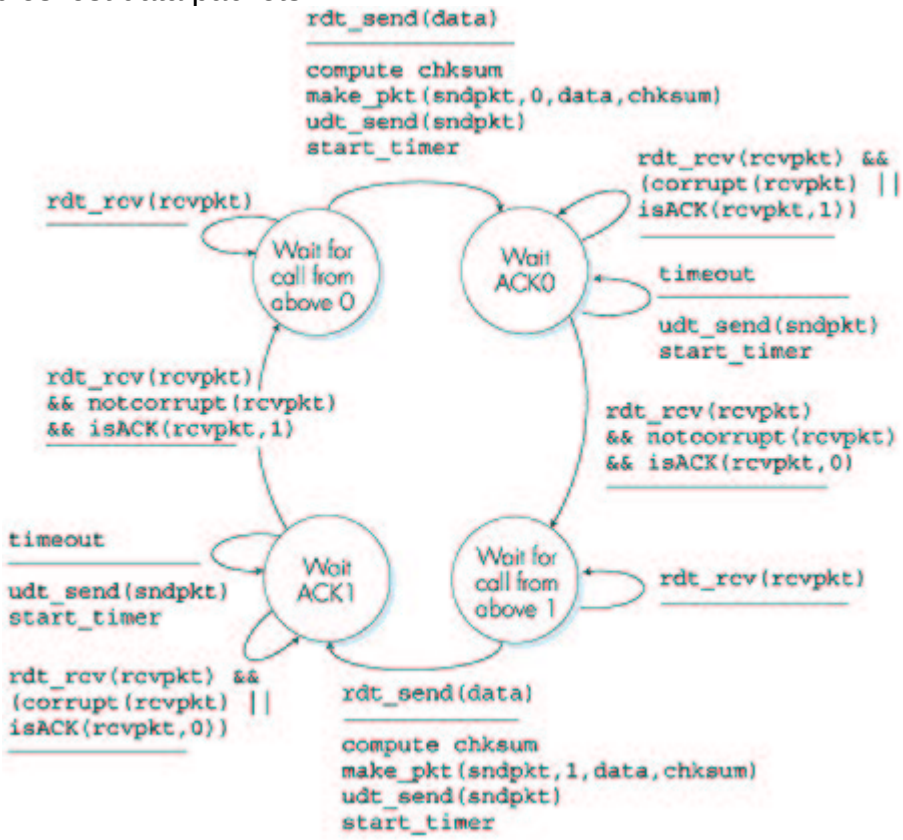


Figure 3.15: rdt3.0 sender FSM

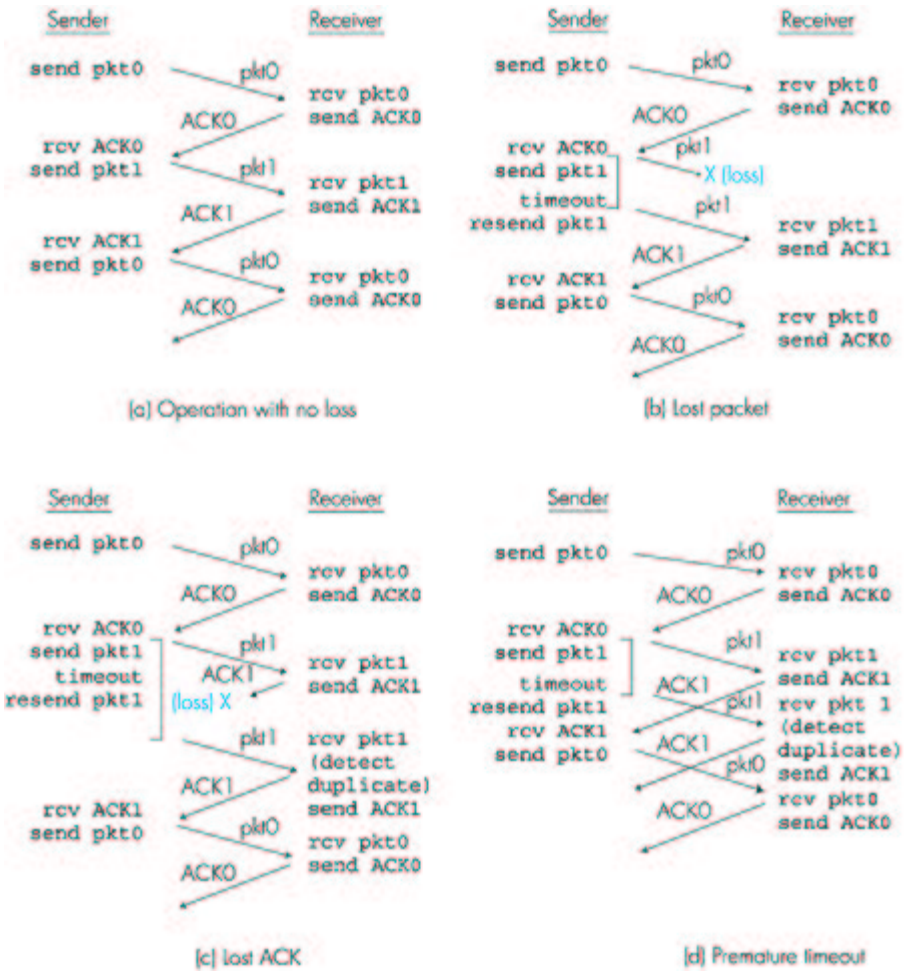


Figure 3.16: Operation of rdt3.0, the alternating-bit protocol

In Figure 3.16, time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. In Figures 3.16b-d, the send-side brackets indicate the times at which a timer is set and later times out. Several of the more subtle aspects of this protocol are explored in the exercises at the end of this chapter. Because packet sequence numbers alternate between 0 and 1, protocol rdt3.0 is sometimes known as the **alternating-bit protocol.** We have now assembled the key elements of a data-transfer protocol. Checksums, sequence numbers, timers, and positive and negative acknowledgment packets each play a crucial and necessary role in the operation of the protocol. We now have a working reliable data-transfer protocol!

3.4.2: Pipelined Reliable Data Transfer Protocols

Protocol rdt3.0 is a functionally correct protocol, but it is unlikely that anyone would be happy with its performance, particularly in today's high-speed networks. At the heart of rdt3.0's performance problem is the fact that it is a stop-and-wait protocol.

To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two end hosts, one located on the West Coast of the United States and the other located on the East Coast. The speed-of-light propagation delay, T_{prop}, between these two end systems is approximately 15 milliseconds. Suppose that they are connected by a channel with a capacity, C, of 1 gigabit (10^9 bits) per second. With a packet size, SP, of 1 Kbytes per packet including both header fields and data, the time needed to actually transmit the packet into the 1Gbps link is

$$T_{\text{trans}} = \frac{SP}{C} = \frac{8\,\text{KBits/packet}}{10^9\,\text{bits/sec}} = 8\,\text{microseconds}$$

With our stop-and-wait protocol, if the sender begins sending the packet at $t = 0$, then at $t = 8$ microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, as depicted in Figure 3.17a, with the last bit of the packet emerging at the receiver at $t = 15.008$ msec.

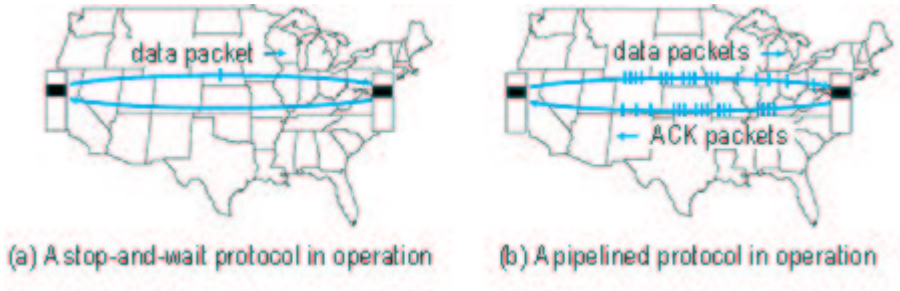


Figure 3.17: Stop-and-wait versus pipelined protocol

Assuming for simplicity that ACK packets are the same size as data packets and that the receiver can begin sending an ACK packet as soon as the last bit of a data packet is received, the last bit of the ACK packet emerges back at the receiver at $t = 30.016$ msec. Thus, in 30.016 msec, the sender was busy (sending or receiving) for only 0.016 msec. If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits via the channel, we have a rather dismal sender utilization, U_{sender}, of

$$U_{\text{sender}} = \frac{.008}{30.016} = 0.00015$$

That is, the sender was busy only 1.5 hundredths of one percent of the time. Viewed another way, the sender was only able to send 1 kilobyte in 30.016 milliseconds, an effective throughput of only 33 kilobytes per second--even though a 1 gigabit per second link was available! Imagine the unhappy network manager who just paid a fortune for a gigabit capacity link but manages to get a throughput of only 33 kilobytes per second! This is a graphic example of how network protocols can limit the capabilities provided by the underlying network hardware. Also, we have neglected lower-layer protocol-processing times at the sender and receiver, as well as the processing and queuing delays that would occur at any intermediate routers between the sender and receiver. Including these effects would only

serve to further increase the delay and further accentuate the poor performance.
The solution to this particular performance problem is a simple one: rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as shown in Figure 3.17(b). Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining.** Pipelining has several consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.

- The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted, but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.

The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: **Go-Back-N** and **selective repeat.**

3.4.3: Go-Back-N (GBN)

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline. Figure 3.18 shows the sender's view of the range of sequence numbers in a GBN protocol. If we define base to be the sequence number of the oldest unacknowledged packet and nextseqnum to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval [0,base-1] correspond to packets that have already been transmitted and acknowledged. The interval [base,nextseqnum-1] corresponds to packets that have been sent but not yet acknowledged. Sequence numbers in the interval [nextseqnum,base+N-1] can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged.

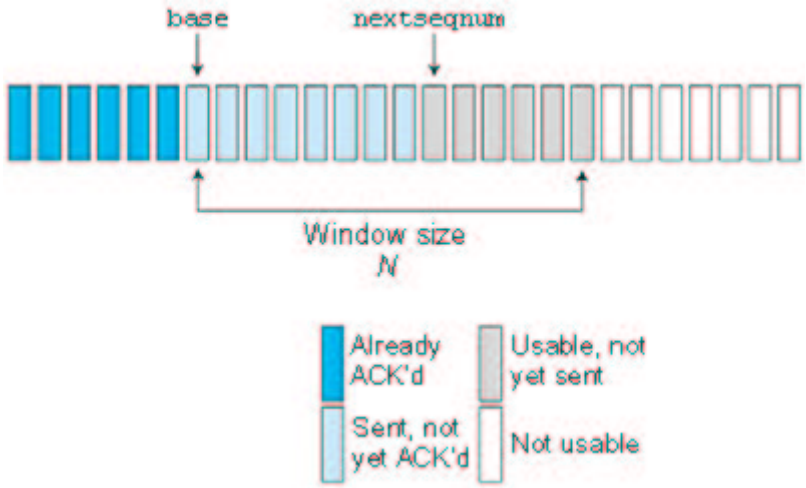


Figure 3.18: Sender's view of sequence numbers in Go-Back-N

As suggested by Figure 3.18, the range of permissible sequence numbers for transmitted but not-yet-acknowledged packets can be viewed as a "window" of size *N* over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol.** You might be wondering why we would even limit the number of outstanding, unacknowledged packets to a value of *N* in the first place. Why not allow an unlimited number of such packets? We will see in Section 3.5 that flow control is one reason to impose a limit on the sender. We'll examine another reason to do so in Section 3.7, when we study TCP congestion control.

In practice, a packet's sequence number is carried in a fixed-length field in the packet header. If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0,2^k - 1]$. With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2^k arithmetic. (That is, the sequence number space can be thought of as a ring of size 2^k, where sequence number $2^k - 1$ is immediately followed by sequence number 0.) Recall that `rdt3.0` had a one-bit sequence number and a range of sequence numbers of [0,1]. Several of the problems at the end of this chapter explore consequences of a finite range of sequence numbers. We will see in Section 3.5 that TCP has a 32-bit sequence-number field, where TCP sequence numbers count bytes in the byte stream rather than packets.

Figures 3.19 and 3.20 give an extended-FSM description of the sender and receiver sides of an ACK-based, NAK-free, GBN protocol. We refer to this FSM description as an **extended FSM** since we have added variables (similar to programming language variables) for `base` and `nextseqnum` and also added operations on these variables and conditional actions involving these variables. Note that the extended FSM specification is now beginning to look somewhat like a programming language specification. [Bochman 1984] provides an excellent survey of additional extensions to FSM techniques as well as other programming language-based techniques for

specifying protocols.

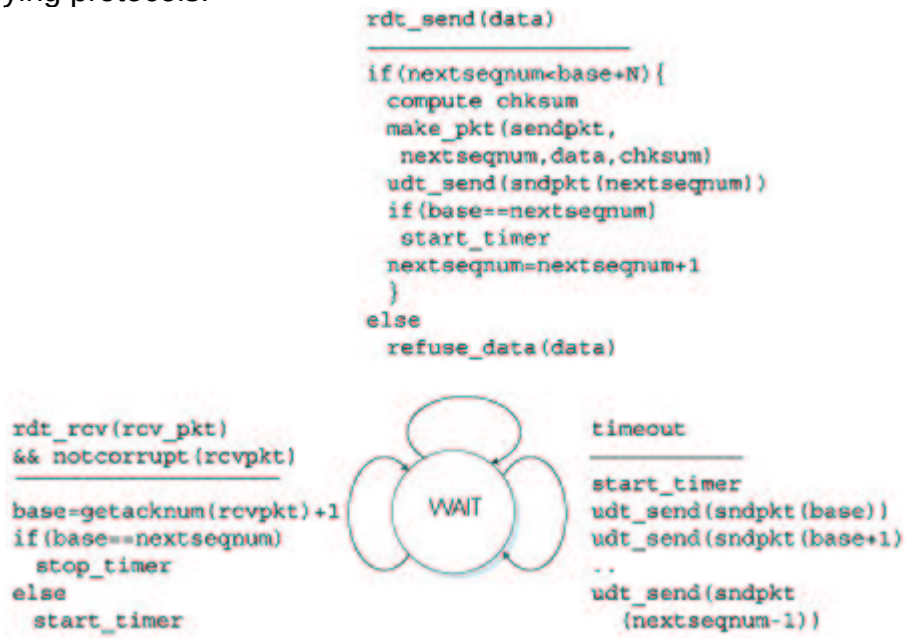


Figure 3.19: Extended FSM description of GBN sender

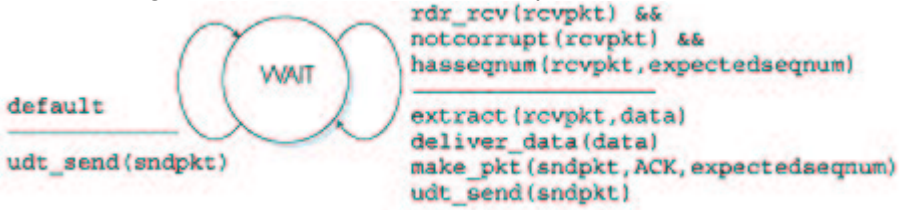


Figure 3.20: Extended FSM description of GBN receiver

The GBN sender must respond to three types of events:

- *Invocation from above.* When `rdt_send()` is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call `rdt_send()` only when the window is not full.

- *Receipt of an ACK.* In our GBN protocol, an acknowledgment for packet with sequence number *n* will be taken to be a **cumulative acknowledgment,** indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver. We'll come back to this issue shortly when we examine the receiver side of GBN.

- *A timeout event.* The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. Our sender in Figure 3.19 uses only a single timer, which can be thought of as a timer for the oldest transmitted-but-not-yet-acknowledged packet. If an ACK is received but there are still additional transmitted-but-yet-to-be-acknowledged packets, the timer is restarted. If there are no outstanding unacknowledged packets, the timer is stopped.

The receiver's actions in GBN are also simple. If a packet with sequence number *n* is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number *n* - 1), the receiver sends an ACK for packet *n* and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. Note that since packets are delivered one-at-a-time to the upper layer, if packet *k* has been received and delivered, then all packets with a sequence number lower than *k* have also been delivered. Thus, the use of cumulative acknowledgments is a natural choice for GBN.

In our GBN protocol, the receiver discards out-of-order packets. Although it may seem silly and wasteful to discard a correctly received (but out-of-order) packet, there is some justification for doing so. Recall that the receiver must deliver data, in order, to the upper layer. Suppose now that packet *n* is expected, but packet *n* + 1 arrives. Since data must be delivered in order, the receiver *could* buffer (save) packet *n* + 1 and then deliver this packet to the upper layer after it had later received and delivered packet *n*. However, if packet *n* is lost, both it and packet *n* + 1 will eventually be retransmitted as a result of the GBN retransmission rule at the sender. Thus, the receiver can simply discard packet *n* + 1. The advantage of this approach is the simplicity of receiver buffering--the receiver need not buffer *any* out-of-order packets. Thus, while the sender must maintain the upper and lower bounds of its window and the position of `nextseqnum` within this window, the only piece of information the receiver need maintain is the sequence number of the next in-order packet. This value is held in the variable `expectedseqnum`, shown in the receiver FSM in Figure 3.20. Of course, the disadvantage of throwing away a correctly received packet is that the subsequent retransmission of that packet might be lost or garbled and thus even more retransmissions would be required.

Figure 3.21 shows the operation of the GBN protocol for the case of a window size of four packets. Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding. As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward

and the sender can transmit one new packet (pkt4 and pkt5, respectively). On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out-of-order and are discarded.

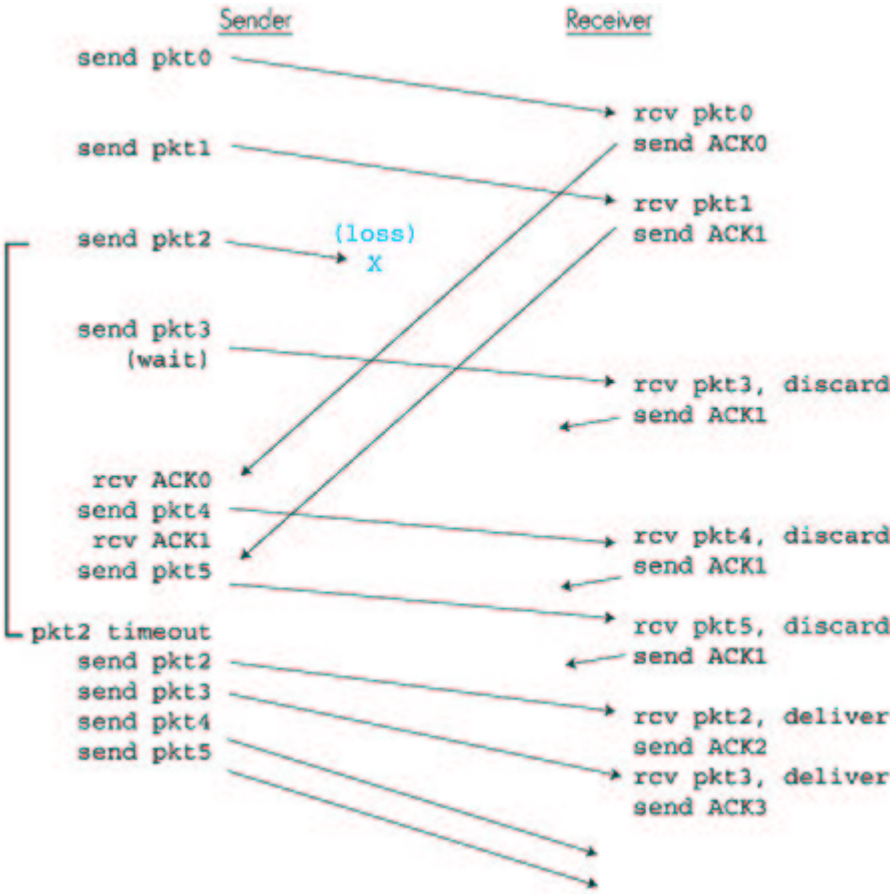


Figure 3.21: Go-Back-N in operation

Before closing our discussion of GBN, it is worth noting that an implementation of this protocol in a protocol stack would likely be structured similar to that of the extended FSM in Figure 3.19. The implementation would also likely be in the form of various procedures that implement the actions to be taken in response to the various events that can occur. In such **event-based programming,** the various procedures are called (invoked) either by other procedures in the protocol stack, or as the result of an interrupt. In the sender, these events would be (1) a call from the upper-layer entity to invoke `rdt_send()`, (2) a timer interrupt, and (3) a call from the lower layer to invoke `rdt_rcv()` when a packet arrives. The programming exercises at the end of this chapter will give you a chance to actually implement these routines in a simulated, but realistic, network setting.

We note here that the GBN protocol incorporates almost all of the techniques that we will encounter when we study the reliable data-transfer components of TCP in Section 3.5. These techniques include the use of sequence numbers, cumulative acknowledgments, checksums, and a time-out/retransmit operation. In this sense, TCP has a number of elements of a GBN-style protocol. There are, however, some differences between GBN

and TCP. Many TCP implementations will buffer correctly received but out-of-order segments [Stevens 1994]. A proposed modification to TCP, the so-called selective acknowledgment [RFC 2581], will also allow a TCP receiver to selectively acknowledge a single out-of-order packet rather than cumulatively acknowledge the last correctly received packet. The notion of a selective acknowledgment is at the heart of the second broad class of pipelined protocols: the so-called selective-repeat protocols that we study below. TCP is thus probably best categorized as a hybrid of Go-Back-N and selective-repeat protocols.

3.4.4: Selective Repeat (SR)

The GBN protocol allows the sender to potentially "fill the pipeline" in Figure 3.17 with packets, thus avoiding the channel utilization problems we noted with stop-and-wait protocols. There are, however, scenarios in which GBN itself will suffer from performance problems. In particular, when the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many of which may be unnecessary. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions. Imagine in our message dictation scenario, if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated. The dictation would be slowed by all of the reiterated words.

As the name suggests, Selective-Repeat (SR) protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as-needed, retransmission will require that the receiver *individually* acknowledge correctly received packets. A window size of *N* will again be used to limit the number of outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the sender will have already received ACKs for some of the packets in the window. Figure 3.22 shows the SR sender’s view of the sequence number space. Figure 3.23 details the various actions taken by the SR sender.

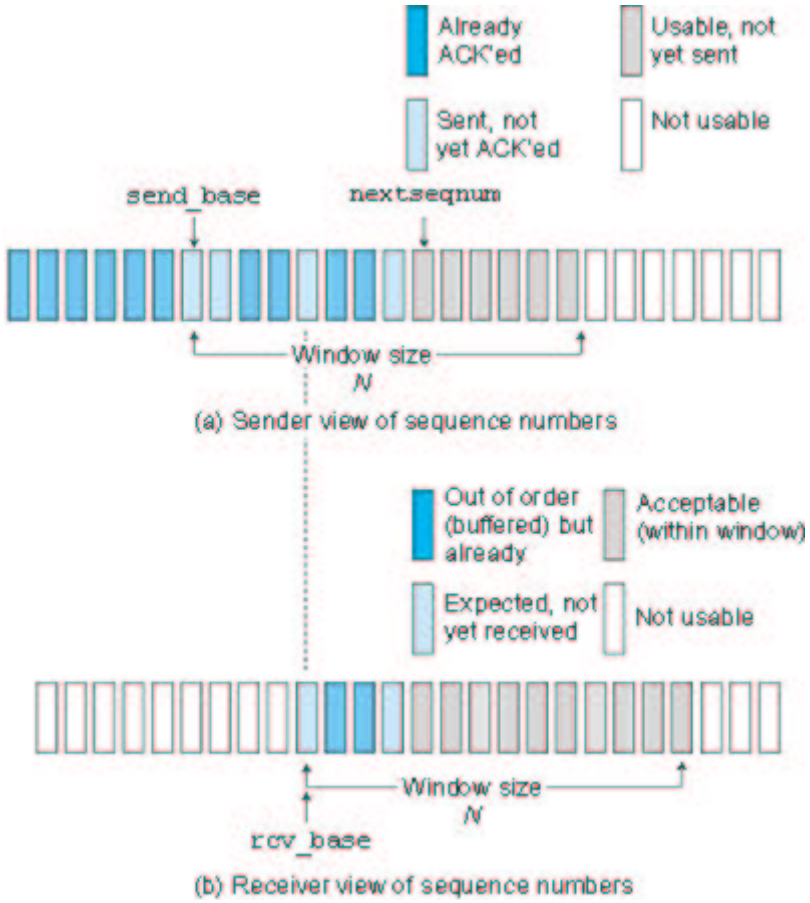


Figure 3.22: Selective Repeat (SR) sender and receiver views of sequence-number space

1. *Data received from above.* When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.

2. *Timeout.* Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers [Varghese 1997].

3. *ACK received.* If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to `send_base`, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

Figure 3.23: Selective Repeat (SR) sender events and actions

The SR receiver will acknowledge a correctly received packet whether or not it is in-order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in-order to the upper layer. Figure 3.24 itemizes the various actions taken by the SR receiver. Figure 3.25 shows an example of SR operation in the presence of lost packets. Note that in Figure 3.25, the receiver initially buffers packets 3 and 4, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

1. *Packet with sequence number in* [rcv_base, rcv_base+N-1] *is correctly received.* In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (rcv_base in Figure 3.22), then this packet, and any previously buffered and consecutively numbered (beginning with rcv_base) packets are delivered to the upper layer. The receive window is then moved forward by the number of packets delivered to the upper layer. As an example, consider Figure 3.25. When a packet with a sequence number of rcv_base=2 is received, it and packets rcv_base+1 and rcv_base+2 can be delivered to the upper layer.

2. *Packet with sequence number in* [rcv_base-N, rcv_base-1] *is received.* In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.

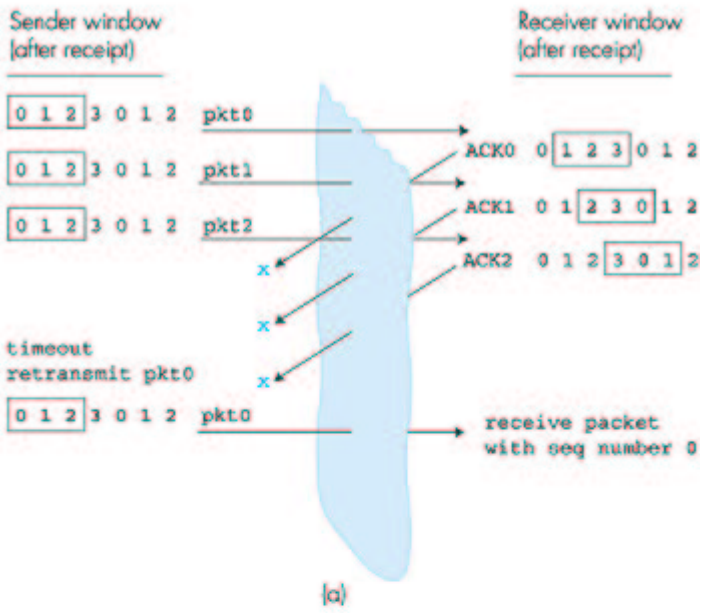
3. *Otherwise.* Ignore the packet.

Figure 3.24: Selective Repeat (SR) receiver events and actions

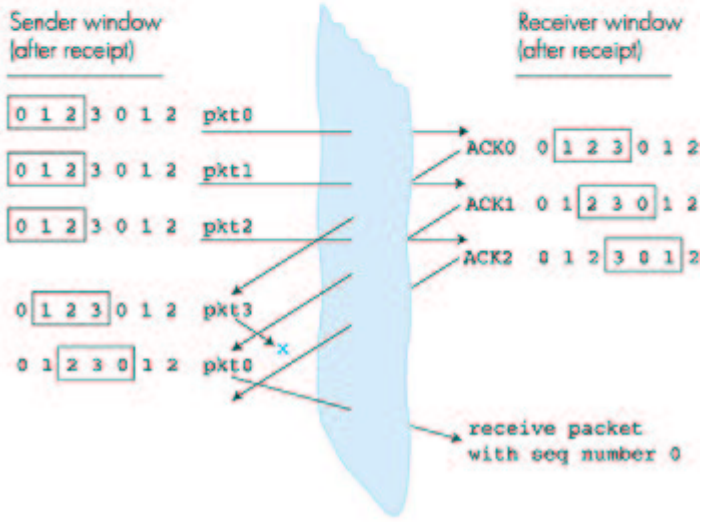
Figure 3.25: SR operation

It is important to note that in step 2 in Figure 3.24, the receiver re-acknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base. You should convince yourself that this re-acknowledgment is indeed needed. Given the sender and receiver sequence-number spaces in Figure 3.22 for example, if there is no ACK for packet `send_base` propagating from the receiver to the sender, the sender will eventually retransmit packet `send_base`, even though it is clear (to us, not the sender!) that the receiver has already received that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward! This example illustrates an important aspect of SR protocols (and many other protocols as well). The sender and receiver will not always have an identical view of what has been received correctly and what has not. For SR protocols, this means that the sender and receiver windows will not always coincide.

The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3 and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios. In the first scenario, shown in Figure 3.26a, the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0--a copy of the first packet sent.



(a)



(b)

Figure 3.26: SR receiver dilemma with too-large windows: A new packet or retransmission?

In the second scenario, shown in Figure 3.26b, the ACKs for the first three packets are all delivered correctly. The sender thus moves its window forward and sends the fourth, fifth, and sixth packets, with sequence numbers 3, 0, 1, respectively. The packet with sequence number 3 is lost, but the packet with sequence number 0 arrives--a packet containing *new* data.

Now consider the receiver's viewpoint in Figure 3.26, which has a figurative curtain between the sender and the receiver, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the

sequence of messages it receives from the channel and sends into the channel. As far as it is concerned, the two scenarios in Figure 3.26 are *identical.* There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet. Clearly, a window size that is 1 less than the size of the sequence number space won't work. But how small must the window size be? A problem at the end of the chapter asks you to show that the window size must be less than or equal to half the size of the sequence-number space for SR protocols.

Let us conclude our discussion of reliable data transfer protocols by considering one remaining assumption in our underlying channel model. Recall that we have assumed that packets cannot be reordered within the channel between the sender and receiver. This is generally a reasonable assumption when the sender and receiver are connected by a single physical wire. However, when the "channel" connecting the two is a network, packet reordering can occur. One manifestation of packet ordering is that old copies of a packet with a sequence or acknowledgment number of *x* can appear, even though neither the sender's nor the receiver's window contains *x*. With packet reordering, the channel can be thought of as essentially buffering packets and spontaneously emitting these packets at *any* point in the future. Because sequence numbers may be reused, some care must be taken to guard against such duplicate packets. The approach taken in practice is to ensure that a sequence number is not reused until the sender is relatively "sure" than any previously sent packets with sequence number *x* are no longer in the network. This is done by assuming that a packet cannot "live" in the network for longer than some fixed maximum amount of time. A maximum packet lifetime of approximately three minutes is assumed in the TCP extensions for high-speed networks [RFC 1323]. [Sunshine 1978] describes a method for using sequence numbers such that reordering problems can be completely avoided.

Online Book

3.5: Connection-Oriented Transport: TCP

Now that we have covered the underlying principles of reliable data transfer, let's turn to TCP--the Internet's transport-layer, connection-oriented, reliable transport protocol. In this section, we'll see that in order to provide reliable data transfer, TCP relies on many of the underlying principles discussed in the previous section, including error detection, retransmissions, cumulative acknowledgments, timers, and header fields for sequence and acknowledgment numbers. TCP is defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581.

3.5.1: The TCP Connection

TCP provides multiplexing, demultiplexing, and error detection in exactly the same manner as UDP. Nevertheless, TCP and UDP differ in many ways. The most fundamental difference is that UDP is **connectionless,** while TCP is **connection-oriented.** UDP is connectionless because it sends data without ever establishing a connection. TCP is connection-oriented because before one application process can begin to send data to another, the two processes must first "handshake" with each other--that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. As part of the TCP connection establishment, both sides of the connection will initialize many TCP "state variables" (many of which will be discussed in this section and in Section 3.7) associated with the TCP connection.

The TCP "connection" is not an end-to-end TDM or FDM circuit as in a circuit-switched network. Nor is it a virtual circuit (see Chapter 1), as the connection state resides entirely in the two end systems. Because the TCP protocol runs only in the end systems and not in the intermediate network elements (routers and bridges), the intermediate network elements do not maintain TCP connection state. In fact, the intermediate routers are completely oblivious to TCP connections; they see datagrams, not connections.

A TCP connection provides for **full duplex** data transfer. If there is a TCP connection between process A on one host and process B on another host, then application-level data can flow from A to B at the same time as application-level data flows from B to A. A TCP connection is also always **point-to-point,** that is, between a single sender and a single receiver. So called "multicasting" (see Section 4.8)--the transfer of data from one sender to many receivers in a single send operation--is not possible with TCP. With TCP, two hosts are company and three are a crowd!

Let’s now take a look at how a TCP connection is established. Suppose a process running in one host wants to initiate a connection with another process in another host. Recall that the process that is initiating the connection is called the client process, while the other process is called the server process. The client application process first informs the client TCP that it wants to establish a connection to a process in the server. Recall from Section 2.6, a Java client program does this by issuing the command:

```
Socket clientSocket = new Socket("hostname", port number);
```

The transport layer in the client then proceeds to establish a TCP connection with the TCP in the server. We will discuss in some detail the connection establishment procedure at the end of this section. For now it suffices to know that the client first sends a special TCP segment; the server responds with a second special TCP segment; and finally the client responds again with a third special segment. The first two segments contain no "payload," that is, no application-layer data; the third of these segments may carry a payload. Because three segments are sent between the two hosts, this connection establishment procedure is often referred to as a **three-way handshake.**

Once a TCP connection is established, the two application processes can send data to each other; because TCP is full-duplex they can send data at the same time. Let us consider the sending of data from the client process to the server process. The client process passes a stream of data through the socket (the door of the process), as described in Section 2.6. Once the data passes through the door, the data is now in the hands of TCP running in the client. As shown in Figure 3.27, TCP directs this data to the connection's **send buffer,** which is one of the buffers that is set aside during the initial three-way handshake. From time to time, TCP will "grab" chunks of data from the send buffer. Interestingly, the TCP specification [RFC 793] is very "laid back" about specifying when TCP should actually send buffered data, stating that TCP should "send that data in segments at its own convenience." The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS).** The MSS depends on the TCP implementation (determined by the operating system) and can often be configured; common values are 1,500 bytes, 536 bytes, and 512 bytes. (These segment sizes are often chosen in order to avoid IP fragmentation, which will be discussed in the next chapter.) Note that the MSS is the maximum amount of application-level data in the segment, not the maximum size of the TCP segment including headers. (This terminology is confusing, but we have to live with it, as it is well entrenched.)

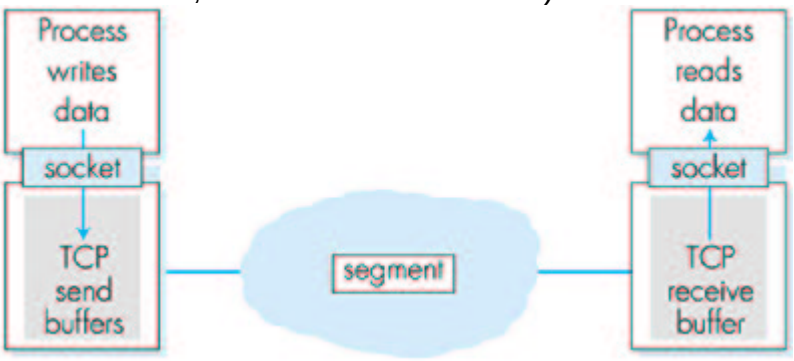


Figure 3.27: TCP send and receive buffers

TCP encapsulates each chunk of client data with a TCP header, thereby forming **TCP segments.** The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams. The IP datagrams are then sent into the network. When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's **receive buffer.** The application reads the stream of data from this buffer. Each side of the connection has its own send buffer and its own receive buffer. The send and receive buffers for data flowing in one direction are shown in Figure 3.27.

We see from this discussion that a TCP connection consists of buffers, variables, and a socket connection to a process in one host, and another set of buffers, variables, and a socket connection to a process in another host. As mentioned earlier, no buffers or variables are allocated to the connection in the network elements (routers, bridges, and repeaters) between the hosts.

Case History

Vinton Cerf, Robert Kahn, and TCP/IP

In the early 1970s, packet-switched networks began to proliferate, with the ARPAnet--the precursor of the Internet--being just one of many networks. Each of these networks had its own protocol. Two researchers, Vinton Cerf and Robert Kahn, recognized the importance of interconnecting these networks, and invented a cross-network protocol called TCP/IP, which stands for Transmission Control Protocol/ Internet Protocol. Although Cerf and Kahn began by seeing the protocol as a single entity, it was later split into its two parts, TCP and IP, which operated separately. Cerf and Kahn published a paper on TCP/IP in May 1974 in *IEEE Transactions on Communications Technology.*

The TCP/IP protocol, which is the bread and butter of today's Internet, was devised before PCs and workstations, before the proliferation of Ethernets and other local area network technologies, before the Web, streaming audio, and chat. Cerf and Kahn saw the need for a networking protocol that, on the one hand, provides broad support for yet-to-be-defined applications, and, on the other hand, allows arbitrary hosts and link layer protocols to interoperate.

3.5.2: TCP Segment Structure

Having taken a brief look at the TCP connection, let's examine the TCP segment structure. The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. As mentioned above, the MSS limits the maximum size of a segment's data field. When TCP sends a large file, such as an encoded image as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS). Interactive applications, however, often transmit data chunks that are smaller than the MSS; for example, with remote login applications like Telnet, the data field in the TCP segment is often only one byte. Because the TCP header is typically 20 bytes (12 bytes more than the UDP header), segments sent by Telnet may only be 21 bytes in length. Figure 3.28 shows the structure of the TCP segment. As with UDP, the header includes **source and destination port numbers,** that are used for multiplexing/ demultiplexing data from/to upper layer applications. Also, as with UDP, the header includes a **checksum field.**

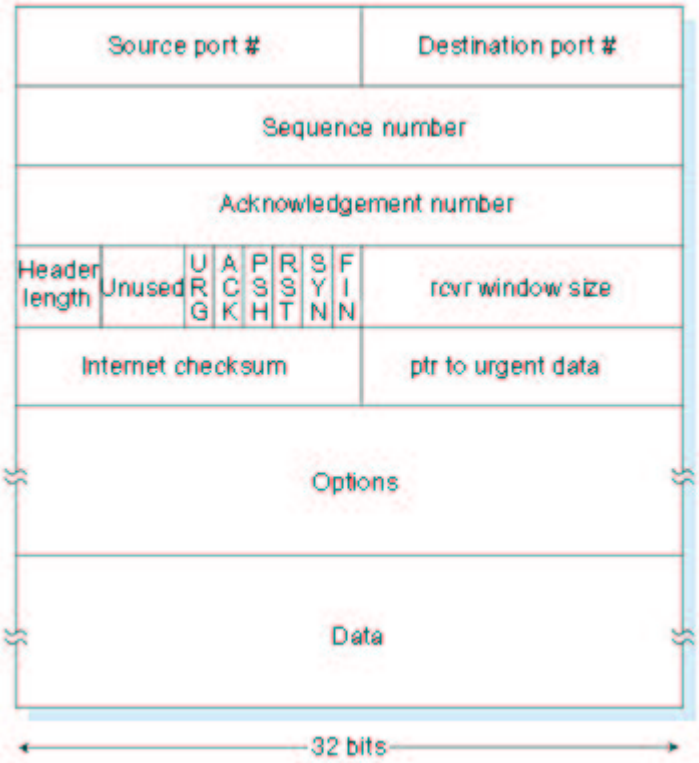


Figure 3.28: TCP segment structure

A TCP segment header also contains the following fields:

- The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data-transfer service, as discussed below.
- The 16-bit **window-size** field is used for flow control. We will see shortly that it is used to indicate the number of bytes that a receiver is willing to accept.
- The 4-bit **length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field, discussed below. (Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)
- The optional and variable length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A timestamping option is also defined. See RFC 854 and RFC 1323 for additional details.
- The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid. The **RST, SYN,** and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section. When the **PSH** bit is set, this is an indication that the receiver should pass the data to the upper layer immediately.

Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper layer entity has marked as "urgent." The location of the last byte of this urgent data is indicated by the 16-bit urgent data pointer. TCP must inform the receiving-side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data. (In practice, the PSH, URG, and pointer to urgent data are not used. However, we mention these fields for completeness.)

3.5.3: Sequence Numbers and Acknowledgment Numbers

Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field. These fields are a critical part of TCP's reliable data transfer service. But before discussing how these fields are used to provide reliable data transfer, let us first explain what exactly TCP puts in these fields.

TCP views data as an unstructured, but ordered, stream of bytes. TCP's use of sequence numbers reflects this view in that sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments. The **sequence number for a segment** is the byte-stream number of the first byte in the segment. Let's look at an example. Suppose that a process in host A wants to send a stream of data to a process in host B over a TCP connection. The TCP in host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered zero. As shown in Figure 3.29, TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1000, the third segment gets assigned sequence number 2000, and so on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.

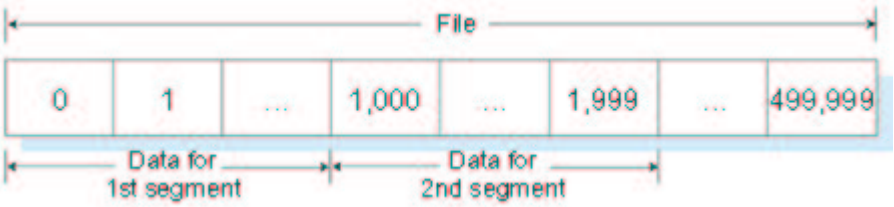


Figure 3.29: Dividing file data into TCP segments

Now let us consider acknowledgment numbers. These are a little trickier than sequence numbers. Recall that TCP is full-duplex, so that host A may be receiving data from host B while it sends data to host B (as part of the same TCP connection). Each of the segments that arrive from host B have a sequence number for the data flowing from B to A. *The acknowledgment number that host A puts in its segment is the sequence number of the next byte host A is expecting from host B.* It is good to look at a few examples to understand what is going on here. Suppose that host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to host B. In other words, host A is waiting for byte 536 and all the subsequent bytes in host B's data stream. So host A puts 536 in the

acknowledgment number field of the segment it sends to B.
As another example, suppose that host A has received one segment from host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason host A has not yet received bytes 536 through 899. In this example, host A is still waiting for byte 536 (and beyond) in order to recreate B's data stream. Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments.**
This last example also brings up an important but subtle issue. Host A received the third segment (bytes 900 through 1,000) before receiving the second segment (bytes 536 through 899). Thus, the third segment arrived out of order. The subtle issue is: What does a host do when it receives out-of-order segments in a TCP connection? Interestingly, the TCP RFCs do not impose any rules here and leave the decision up to the people programming a TCP implementation. There are basically two choices: either (1) the receiver immediately discards out-of-order bytes; or (2) the receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps. Clearly, the latter choice is more efficient in terms of network bandwidth, whereas the former choice simplifies the TCP code. Throughout the remainder of this introductory discussion of TCP, we focus on the former implementation, that is, we assume that the TCP receiver discards out-of-order segments.
In Figure 3.29 we assumed that the initial sequence number was zero. In truth, both sides of a TCP connection randomly choose an initial sequence number. This is done to minimize the possibility that a segment that is still present in the network from an earlier, already-terminated connection between two hosts is mistaken for a valid segment in a later connection between these same two hosts (who also happen to be using the same port numbers as the old connection) [Sunshine 1978].

3.5.4: Telnet: A Case Study for Sequence and Acknowledgment Numbers

Telnet, defined in RFC 854, is a popular application-layer protocol used for remote login. It runs over TCP and is designed to work between any pair of hosts. Unlike the bulk-data transfer applications discussed in Chapter 2, Telnet is an interactive application. We discuss a Telnet example here, as it nicely illustrates TCP sequence and acknowledgment numbers.
Suppose host A initiates a Telnet session with host B. Because host A initiates the session, it is labeled the client, and host B is labeled the server. Each character typed by the user (at the client) will be sent to the remote host; the remote host will send back a copy of each character, which will be displayed on the Telnet user's screen. This "echo back" is used to ensure that characters seen by the Telnet user have already been received and processed at the remote site. Each character thus traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor.

Now suppose the user types a single letter, 'C', and then grabs a coffee. Let's examine the TCP segments that are sent between the client and server. As shown in Figure 3.30, we suppose the starting sequence numbers are 42 and 79 for the client and server, respectively. Recall that the sequence number of a segment is the sequence number of the first byte in the data field. Thus, the first segment sent from the client will have sequence number 42; the first segment sent from the server will have sequence number 79. Recall that the acknowledgment number is the sequence number of the next byte of data that the host is waiting for. After the TCP connection is established but before any data is sent, the client is waiting for byte 79 and the server is waiting for byte 42.

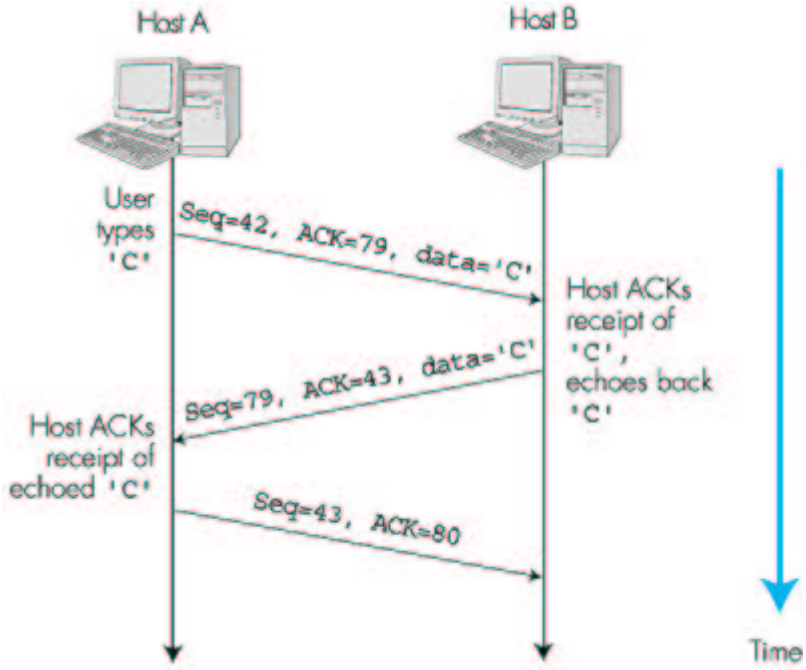


Figure 3.30: Sequence and acknowledgement numbers for simple Telnet application over TCP

As shown in Figure 3.30, three segments are sent. The first segment is sent from the client to the server, containing the one-byte ASCII representation of the letter 'C' in its data field. This first segment also has 42 in its sequence number field, as we just described. Also, because the client has not yet received any data from the server, this first segment will have 79 in its acknowledgment number field.

The second segment is sent from the server to the client. It serves a dual purpose. First it provides an acknowledgment for the data the server has received. By putting 43 in the acknowledgment field, the server is telling the client that it has successfully received everything up through byte 42 and is now waiting for bytes 43 onward. The second purpose of this segment is to echo back the letter 'C'. Thus, the second segment has the ASCII representation of 'C' in its data field. This second segment has the sequence number 79, the initial sequence number of the server-to-client data flow of this TCP connection, as this is the very first byte of data that the server is sending.

Note that the acknowledgment for client-to-server data is carried in a segment carrying server-to-client data; this acknowledgment is said to be **piggybacked** on the server-to-client data segment.
The third segment is sent from the client to the server. Its sole purpose is to acknowledge the data it has received from the server. (Recall that the second segment contained data--the letter 'C'--from the server to the client.) This segment has an empty data field (that is, the acknowledgment is not being piggybacked with any client-to-server data). The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 onward. You might think it odd that this segment also has a sequence number since the segment contains no data. But because TCP has a sequence number field, the segment needs to have some sequence number.

3.5.5: Reliable Data Transfer

Recall that the Internet's network layer service (IP service) is unreliable. IP does not guarantee datagram delivery, does not guarantee in-order delivery of datagrams, and does not guarantee the integrity of the data in the datagrams. With IP service, datagrams can overflow router buffers and never reach their destination, datagrams can arrive out of order, and bits in the datagram can get corrupted (flipped from 0 to 1 and vice versa). Because transport-layer segments are carried across the network by IP datagrams, transport-layer segments can also suffer from these problems as well.
TCP creates a **reliable data-transfer service** on top of IP's unreliable best-effort service. TCP's reliable data-transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence, that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection. In this subsection, we provide an overview of how TCP provides a reliable data transfer. We'll see that the reliable data transfer service of TCP uses many of the principles that we studied in Section 3.4.
Figure 3.31 shows the three major events related to data transmission/ retransmission at a simplified TCP sender. Let's consider a TCP connection between host A and B and focus on the data stream being sent from host A to host B. At the sending host (A), TCP is passed application-layer data, which it frames into segments and then passes on to IP. The passing of data from the application to TCP and the subsequent framing and transmission of a segment is the first important event that the TCP sender must handle. Each time TCP releases a segment to IP, it starts a timer for that segment. If this timer expires, an interrupt event is generated at host A. TCP responds to the timeout event, the second major type of event that the TCP sender must handle, by retransmitting the segment that caused the timeout.

```
/*assume sender is not constrained by TCP flow or congestion control,
  that data from above is less than MSS in size, and that data
  transfer is in one direction only
*/
```

```
sendbase=initial_sequence number /*see Figure 3.18*/
nextseqnum=initial_sequence number

loop (forever) {
 switch(event)

   event: data received from application above
     create TCP segment with sequence number nextseqnum
     start timer for segment nextseqnum
     pass segment to IP
     nextseqnum=nextseqnum+length(data)
     break;  /* end of event data received from above */

   event: timer timeout for segment with sequence number y
     retransmit segment with sequence number y
     compute new timeout interval for segment y
     restart timer for sequence number y
     break;  /* end of timeout event */

   event: ACK received, with ACK field value of y
     if (y > sendbase) {/* cumulative ACK of all date up to y */
       cancel all timers for segments with sequence numbers < y
       sendbase=y
       }
     else { /* a duplicate ACK for already ACKed segment */
       increment number of duplicate ACKs received for y
       if (number of duplicate ACKs received for y==3) {
         /* TCP fast retransmit */
         resend segment with sequence number y
         restart timer for segment y
       }
       break; /* end of ACK received event */
} /* end of loop forever */
```

Figure 3.31: Simplified TCP sender

The third major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver (more specifically, a segment containing a valid ACK field value). Here, the sender's TCP must determine whether the ACK is a **first-time ACK** for a segment for which the sender has yet to receive an acknowledgment, or a so-called **duplicate ACK** that re-acknowledges a segment for which the sender has already received an earlier acknowledgment. In the case of the arrival of a first-time ACK, the sender now knows that *all* data up to the byte being acknowledged has been received correctly at the receiver. The sender can thus update its TCP state

variable that tracks the sequence number of the last byte that is known to have been received correctly and in order at the receiver.
To understand the sender's response to a duplicate ACK, we must look at why the receiver sends a duplicate ACK in the first place. Table 3.1 summarizes the TCP receiver's ACK generation policy. When a TCP receiver receives a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream--that is, a missing segment. Since TCP does not use negative acknowledgments, the receiver cannot send an explicit negative acknowledgment back to the sender. Instead, it simply re-acknowledges (that is, generates a duplicate ACK for) the last in-order byte of data it has received. If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost. In this case, TCP performs a **fast retransmit** [RFC 2581], retransmitting the missing segment *before* that segment's timer expires.

Table 3.1: TCP ACK generation recommendations [RFC 1122, RFC 2581]

Event	TCP receiver action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged. No gaps in the received data.	Delayed ACK. Wait up to 500 ms for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission. No gaps in the received data.	Immediately send single cumulative ACK. ACKing both in-order segments
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte.
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

TCP provides reliable data transfer by using positive acknowledgments and timers in much the same way that we studied in Section 3.4. TCP acknowledges data that has been received correctly, and it then retransmits segments when segments or their corresponding acknowledgments are thought to be lost or corrupted. Certain versions of TCP also have an implicit NAK mechanism--with TCP's fast retransmit mechanism, the receipt of three duplicate ACKs for a given segment serves as an implicit NAK for the following segment, triggering retransmission of that segment before timeout. TCP uses sequences of numbers to allow the receiver to identify lost or duplicate segments. Just as in the case of our reliable data transfer protocol, `rdt3.0`, TCP cannot itself tell for certain if a segment, or its ACK, is lost, corrupted, or overly delayed. At the sender, TCP's response will be the same: retransmit the segment in

question.

TCP also uses pipelining, allowing the sender to have multiple transmitted but yet-to-be-acknowledged segments outstanding at any given time. We saw earlier that pipelining can greatly improve a session's throughput when the ratio of the segment size to round-trip delay is small. The specific number of outstanding unacknowledged segments that a sender can have is determined by TCP's flow-control and congestion-control mechanisms. TCP flow control is discussed at the end of this section; TCP congestion control is discussed in Section 3.7. For the time being, we must simply be aware that the TCP sender uses pipelining.

A Few Interesting Scenarios

We end this discussion by looking at a few simple scenarios. Figure 3.32 depicts the scenario in which host A sends one segment to host B. Suppose that this segment has sequence number 92 and contains 8 bytes of data. After sending this segment, host A waits for a segment from B with acknowledgment number 100. Although the segment from A is received at B, the acknowledgment from B to A gets lost. In this case, the timer expires, and host A retransmits the same segment. Of course, when host B receives the retransmission, it will observe from the sequence number that the segment contains data that has already been received. Thus, TCP in host B will discard the bytes in the retransmitted segment.

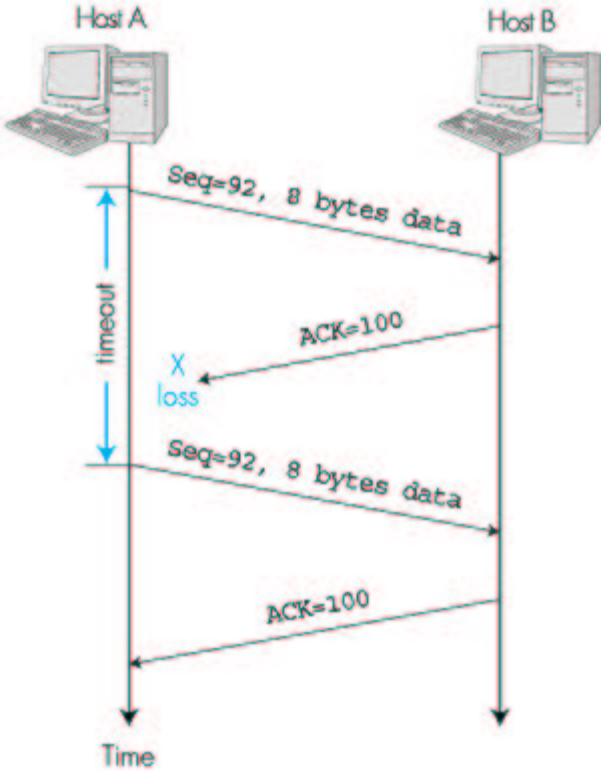


Figure 3.32: Retransmission due to a lost acknowledgement

In a second scenario, host A sends two segments back to back. The first segment has sequence number 92 and 8 bytes of data, and the second segment has sequence number 100 and 20 bytes of data. Suppose that both segments arrive intact at B, and B sends two separate acknowledgments for each of these segments. The first of these acknowledgments has acknowledgment number 100; the second has acknowledgment number 120. Suppose now that neither of the acknowledgments arrive at host A before the

timeout of the first segment. When the timer expires, host A resends the first segment with sequence number 92. Now, you may ask, does A also resend the second segment? According to the rules described above, host A resends the segment only if the timer expires before the arrival of an acknowledgment with an acknowledgment number of 120 or greater. Thus, as shown in Figure 3.33, if the second acknowledgment does not get lost and arrives before the timeout of the second segment, A does not resend the second segment.

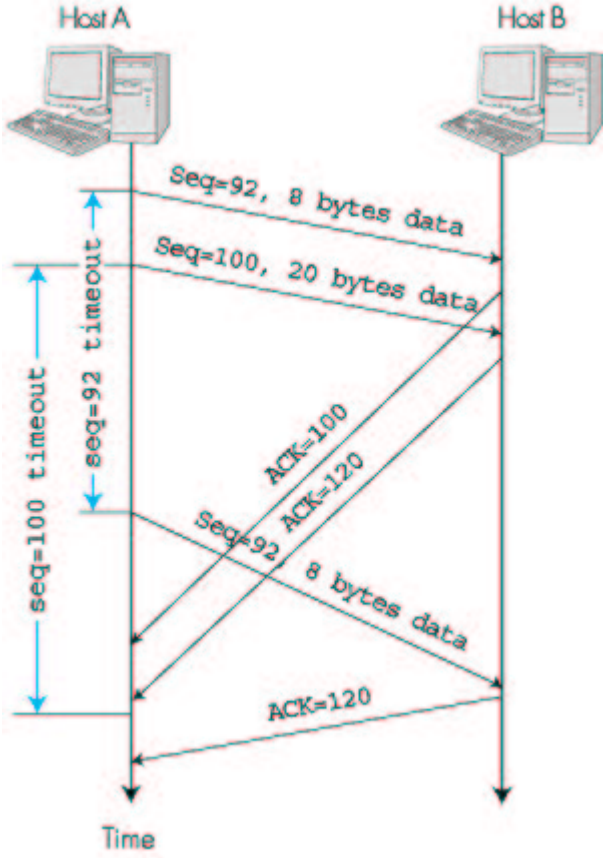


Figure 3.33: Segment is not retransmitted because its acknowledgement arrives before the timeout

In a third and final scenario, suppose host A sends the two segments, exactly as in the second example. The acknowledgment of the first segment is lost in the network, but just before the timeout of the first segment, host A receives an acknowledgment with acknowledgment number 120. Host A therefore knows that host B has received *everything* up through byte 119; so host A does not resend either of the two segments. This scenario is illustrated in the Figure 3.34.

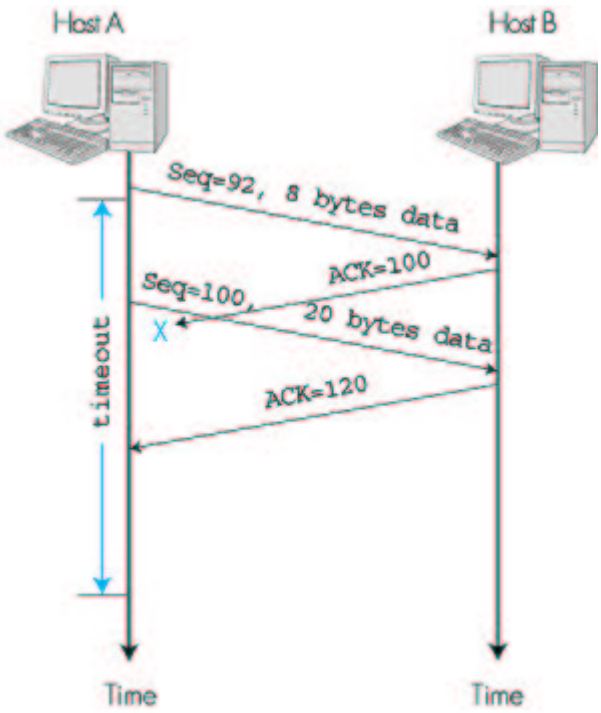


Figure 3.34: A cumulative acknowledgement avoids retransmission of first segment

Recall that in the previous section we said that TCP is a Go-Back-N style protocol. This is because acknowledgments are cumulative and correctly received but out-of-order segments are not individually ACKed by the receiver. Consequently, as shown in Figure 3.31 (see also Figure 3.18), the TCP sender need only maintain the smallest sequence number of a transmitted but unacknowledged byte (`sendbase`) and the sequence number of the next byte to be sent (`nextseqnum`). But the reader should keep in mind that although the reliable data-transfer component of TCP resembles Go-Back-N, it is by no means a pure implementation of Go-Back-N. To see that there are some striking differences between TCP and Go-Back-N, consider what happens when the sender sends a sequence of segments *1, 2, . . ., N,* and all of the segments arrive in order without error at the receiver. Further suppose that the acknowledgment for packet $n < N$ gets lost, but the remaining $N - 1$ acknowledgments arrive at the sender before their respective timeouts. In this example, Go-Back-N would retransmit not only packet n, but also all of the subsequent packets $n + 1, n + 2, \ldots, N$. TCP, on the other hand, would retransmit at most, one segment, namely, segment n. Moreover, TCP would not even retransmit segment n if the acknowledgment for segment $n + 1$ arrives before the timeout for segment n.

There have recently been several proposals [RFC 2018; Fall 1996; Mathis 1996] to extend the TCP ACKing scheme to be more similar to a selective repeat protocol. The key idea in these proposals is to provide the sender with explicit information about which segments have been received correctly, and which are still missing at the receiver.

3.5.6: Flow Control

Recall that the hosts on each side of a TCP connection each set aside a receive buffer for the connection. When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer. The associated application process will read data from this buffer, but not necessarily at the instant the data arrives. Indeed, the receiving application may be busy with some other task and may not even attempt to read the data until long after it has arrived. If the application is relatively slow at reading the data, the sender can very easily overflow the connection's receive buffer by sending too much data too quickly. TCP thus provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed matching service--matching the rate at which the sender is sending to the rate at which the receiving application is reading. As noted earlier, a TCP sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as **congestion control,** a topic we will explore in detail in Sections 3.6 and 3.7. Even though the actions taken by flow and congestion control are similar (the throttling of the sender), they are obviously taken for very different reasons. Unfortunately, many authors use the term interchangeably, and the savvy reader would be careful to distinguish between the two cases. Let's now discuss how TCP provides its flow-control service.

TCP provides flow control by having the sender maintain a variable called the **receive window.** Informally, the receive window is used to give the sender an idea about how much free buffer space is available at the receiver. In a full-duplex connection, the sender at each side of the connection maintains a distinct receive window. The receive window is dynamic; that is, it changes throughout a connection's lifetime. Let's investigate the receive window in the context of a file transfer. Suppose that host A is sending a large file to host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by RcvBuffer. From time to time, the application process in host B reads from the buffer. Define the following variables:

> LastByteRead = the number of the last byte in the data stream read from the buffer by the application process in B.
>
> LastByteRcvd = the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.

Because TCP is not permitted to overflow the allocated buffer, we must have:

$$LastByteRcvd - LastByteRead <= RcvBuffer$$

The receive window, denoted RcvWindow, is set to the amount of spare room in the buffer:

$$RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Because the spare room changes with time, RcvWindow is dynamic. The variable RcvWindow is illustrated in Figure 3.35.

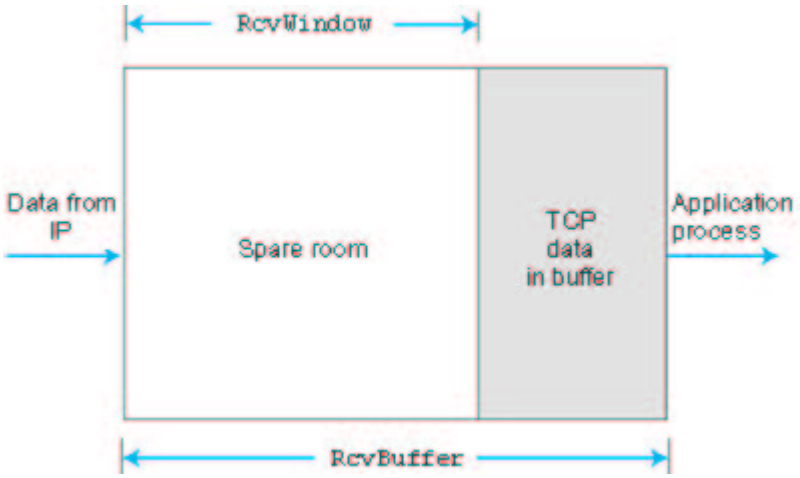


Figure 3.35: The receive window (RcvWindow) and the receive buffer (RcvBuffer)

How does the connection use the variable RcvWindow to provide the flow control service? Host B tells host A how much spare room it has in the connection buffer by placing its current value of RcvWindow in the window field of every segment it sends to A. Initially, host B sets RcvWindow = RcvBuffer. Note that to pull this off, host B must keep track of several connection-specific variables.

Host A in turn keeps track of two variables, LastByteSent and LastByteAcked, which have obvious meanings. Note that the difference between these two variables, LastByteSent - LastByteAcked, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of RcvWindow, host A is assured that it is not overflowing the receive buffer at host B. Thus, host A makes sure throughout the connection's life that

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

There is one minor technical problem with this scheme. To see this, suppose host B's receive buffer becomes full so that RcvWindow = 0. After advertising RcvWindow = 0 to host A, also suppose that B has *nothing* to send to A. As the application process at B empties the buffer, TCP does not send new segments with new RcvWindows to host A--TCP will send a segment to host A only if it has data to send or if it has an acknowledgment to send. Therefore, host A is never informed that some space has opened up in host B's receive buffer: host A is blocked and can transmit no more data! To solve this problem, the TCP specification requires host A to continue to send segments with one data byte when B's receive window is zero. These segments will be acknowledged by the receiver. Eventually the buffer will begin to empty and the acknowledgments will contain a nonzero RcvWindow value.

Having described TCP's flow-control service, we briefly mention here that UDP does not provide flow control. To understand the issue here, consider sending a series of UDP segments from a process on host A to a process on host B. For a typical UDP implementation, UDP will append the segments (more precisely, the data in the segments) in a finite-size queue that

"precedes" the corresponding socket (that is, the door to the process). The process reads one entire segment at a time from the queue. If the process does not read the segments fast enough from the queue, the queue will overflow and segments will get lost.
Following this section we provide an interactive Java applet that should provide significant insight into the TCP receive window. Click here to open it in a new window, or select it from the menu bar at the left.

3.5.7: Round Trip Time and Timeout

Recall that when a host sends a segment into a TCP connection, it starts a timer. If the timer expires before the host receives an acknowledgment for the data in the segment, the host retransmits the segment. The time from when the timer is started until when it expires is called the **timeout** of the timer. A natural question is, how large should timeout be? Clearly, the timeout should be larger than the connection's round-trip time, that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent. But the timeout should not be much larger than the round-trip time; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, and it would thereby introduce significant data transfer delays into the application. Before discussing the timeout interval in more detail, let's take a closer look at the round-trip time (RTT). The discussion below is based on the TCP work in [Jacobson 1988].

Estimating the Average Round-Trip Time

The sample RTT, denoted SampleRTT, for a segment is the amount of time from when the segment is sent (that is, passed to IP) until an acknowledgment for the segment is received. Each segment sent will have its own associated SampleRTT. Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given SampleRTT value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values. TCP maintains an average, called EstimatedRTT, of the SampleRTT values. Upon receiving an acknowledgment and obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:

$$\text{EstimatedRTT} = (1 - x) \cdot \text{EstimatedRTT} + x \cdot \text{SampleRTT}.$$

The above formula is written in the form of a programming language statement--the new value of EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT. A typical value of x is $x = 0.125$ (i.e., 1/8), in which case the above formula becomes:

$$\text{EstimatedRTT} = 0.875\ \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT}.$$

Note that EstimatedRTT is a weighted average of the SampleRTT values. As we will see in the homework, this weighted average puts more weight on recent samples than on old samples, This is natural, as the more recent samples better reflect the current congestion in the network. In statistics, such an average is called an **exponential weighted moving average** (EWMA). The

word "exponential" appears in EWMA because the weight of a given SampleRTT decays exponentially fast as the updates proceed. In the homework problems you will be asked to derive the exponential term in EstimatedRTT. Figure 3.36 shows the SampleRTT values (dotted line) and EstimatedRTT (solid line) for a value of $x = 1/8$ for a TCP connection between void.cs.umass.edu (in Amherst, Massachusetts) to maria.wustl.edu (in St. Louis, Missouri). Clearly, the variations in the SampleRTT are smoothed out in the computation of the EstimatedRTT.

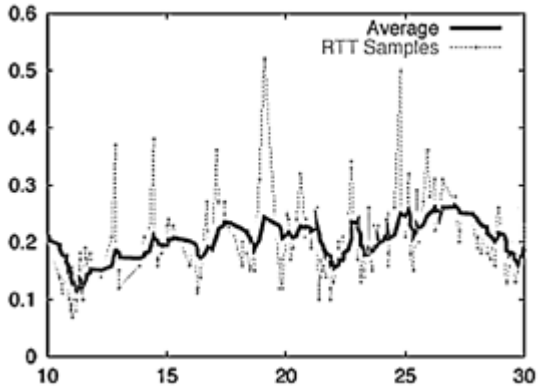


Figure 3.36: RTT samples and RTT estimate

Setting the Timeout

The timeout should be set so that a timer expires early (that is, before the delayed arrival of a segment's ACK) only on rare occasions. It is therefore natural to set the timeout equal to the EstimatedRTT plus some margin. The margin should be large when there is a lot of fluctuation in the SampleRTT values; it should be small when there is little fluctuation. TCP uses the following formula:

$$\text{Timeout} = \text{EstimatedRTT} + 4 \cdot \text{Deviation},$$

where Deviation is an estimate of how much SampleRTT typically deviates from EstimatedRTT:

$$\text{Deviation} = (1 - x) \cdot \text{Deviation} + x \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Note that Deviation is an EWMA of how much SampleRTT deviates from EstimatedRTT. If the SampleRTT values have little fluctuation, then Deviation is small and Timeout is hardly more than EstimatedRTT; on the other hand, if there is a lot of fluctuation, Deviation will be large and Timeout will be much larger than EstimatedRTT. "A Quick Tour around TCP" [Cela 2000] provides nifty interactive applets illustrating RTT variance estimation.

3.5.8: TCP Connection Management

In this subsection, we take a closer look at how a TCP connection is established and torn down. Although this topic may not seem particularly exciting, it is important because TCP connection establishment can significantly add to perceived delays (for example, when surfing the Web).

Let's now take a look at how a TCP connection is established. Suppose a process running in one host (client) wants to initiate a connection with another process in another host (server). The client application process first informs the client TCP that it wants to establish a connection to a process in the server. The TCP in the client then proceeds to establish a TCP connection with the TCP in the server in the following manner:

- **Step 1.** The client-side TCP first sends a special TCP segment to the server-side TCP. This special segment contains no application-layer data. It does, however, have one of the flag bits in the segment's header (see Figure 3.28), the so-called SYN bit, set to 1. For this reason, this special segment is referred to as a **SYN segment.** In addition, the client chooses an initial sequence number (client_isn) and puts this number in the sequence number field of the initial TCP SYN segment. This segment is encapsulated within an IP datagram and sent to the server.

- **Step 2.** Once the IP datagram containing the TCP SYN segment arrives at the server host (assuming it does arrive!), the server extracts the TCP SYN segment from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to client TCP. This connection-granted segment also contains no application-layer data. However, it does contain three important pieces of information in the segment header. First, the SYN bit is set to 1. Second, the acknowledgment field of the TCP segment header is set to client_isn+1. Finally, the server chooses its own initial sequence number (server_isn) and puts this value in the sequence number field of the TCP segment header. This connection granted segment is saying, in effect, "I received your SYN packet to start a connection with your initial sequence number, client_isn. I agree to establish this connection. My own initial sequence number is server_isn." The connection-granted segment is sometimes referred to as a **SYNACK** segment.

- **Step 3.** Upon receiving the connection-granted segment, the client also allocates buffers and variables to the connection. The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment (the client does so by putting the value server_isn+1 in the acknowledgment field of the TCP segment header). The SYN bit is set to 0, since the connection is established.

Once the previous three steps have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero. Note, that in order to establish the connection, three packets are sent between the two hosts, as illustrated in Figure 3.37. For this reason, this connection establishment procedure is often referred to as a **three-way handshake.** Several aspects of the TCP three-way handshake are explored in the homework problems (Why are initial sequence

numbers needed? Why is a three-way handshake, as opposed to a two-way handshake, needed?)

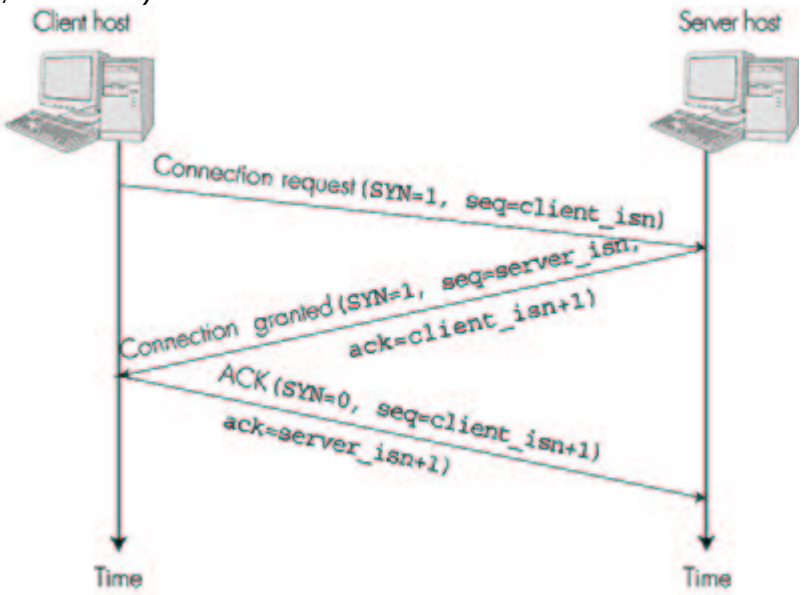


Figure 3.37: TCP three-way handshake: segment exchange

All good things must come to an end, and the same is true with a TCP connection. Either of the two processes participating in a TCP connection can end the connection. When a connection ends, the "resources" (that is, the buffers and variables) in the hosts are de-allocated. As an example, suppose the client decides to close the connection, as shown in Figure 3.38. The client application process issues a close command. This causes the client TCP to send a special TCP segment to the server process. This special segment has a flag bit in the segment's header, the so-called FIN bit (see Figure 3.38), set to 1. When the server receives this segment, it sends the client an acknowledgment segment in return. The server then sends its own shut-down segment, which has the FIN bit set to 1. Finally, the client acknowledges the server's shut-down segment. At this point, all the resources in the two hosts are now de-allocated.

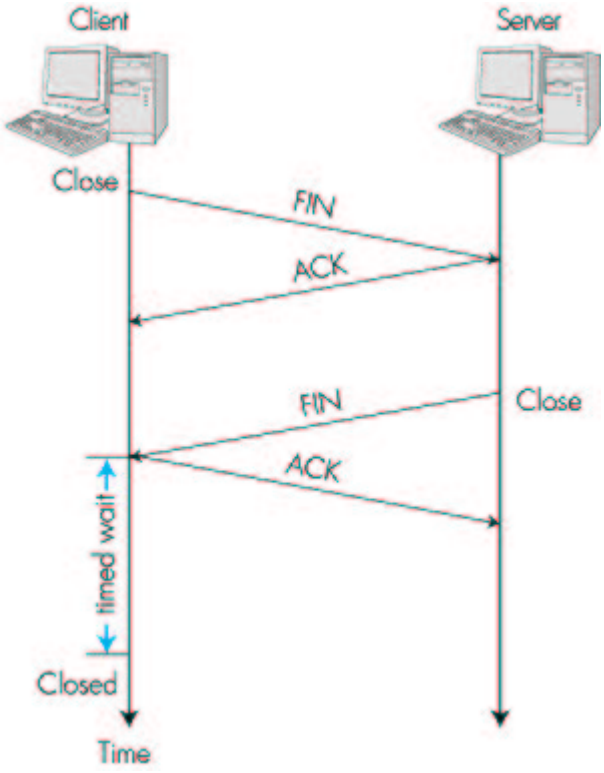


Figure 3.38: Closing a TCP connection

During the life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states.** Figure 3.39 illustrates a typical sequence of TCP states that are visited by the *client* TCP. The client TCP begins in the closed state. The application on the client side initiates a new TCP connection (by creating a Socket object in our Java examples from Chapter 2). This causes TCP in the client to send a SYN segment to TCP in the server. After having sent the SYN segment, the client TCP enters the SYN_SENT state. While in the SYN_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client's previous segment as well as the SYN bit set to 1. Once having received such a segment, the client TCP enters the ESTABLISHED state. While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload (that is, application-generated) data.

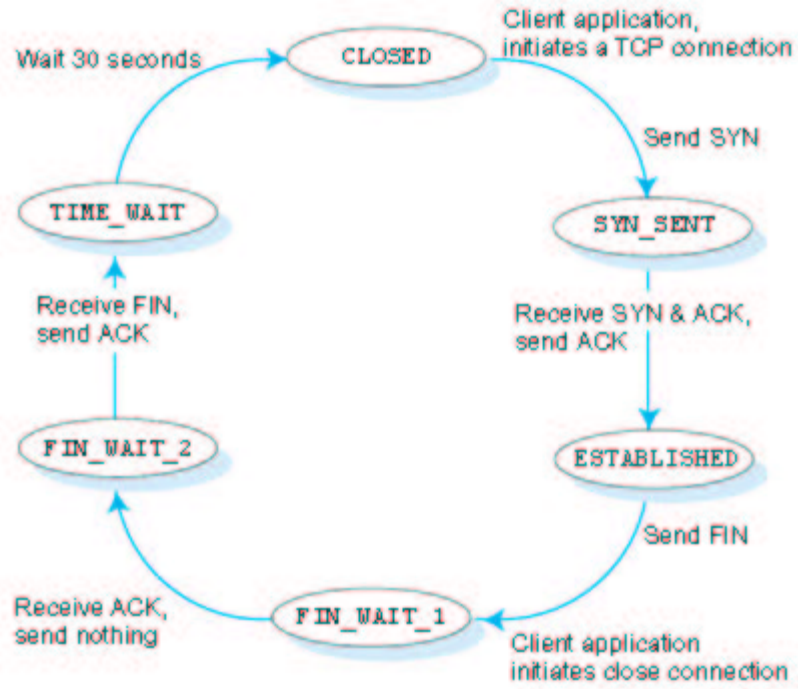


Figure 3.39: A typical sequence of TCP states visited by a client TCP

Suppose that the client application decides it wants to close the connection. (Note that the server could also choose to close the connection.) This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN_WAIT_1 state. While in the FIN_WAIT_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment. When it receives this segment, the client TCP enters the FIN_WAIT_2 state. While in the FIN_WAIT_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server's segment and enters the TIME_WAIT state. The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost. The time spent in the TIME_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes. After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.

Figure 3.40 illustrates the series of states typically visited by the server-side TCP, assuming the client begins connection tear down. The transitions are self-explanatory. In these two state-transition diagrams, we have only shown how a TCP connection is normally established and shut down. We have not described what happens in certain pathological scenarios, for example, when both sides of a connection want to shut down at the same time. If you are interested in learning about this and other advanced issues concerning TCP, you are encouraged to see Stevens' comprehensive book [Stevens 1994].

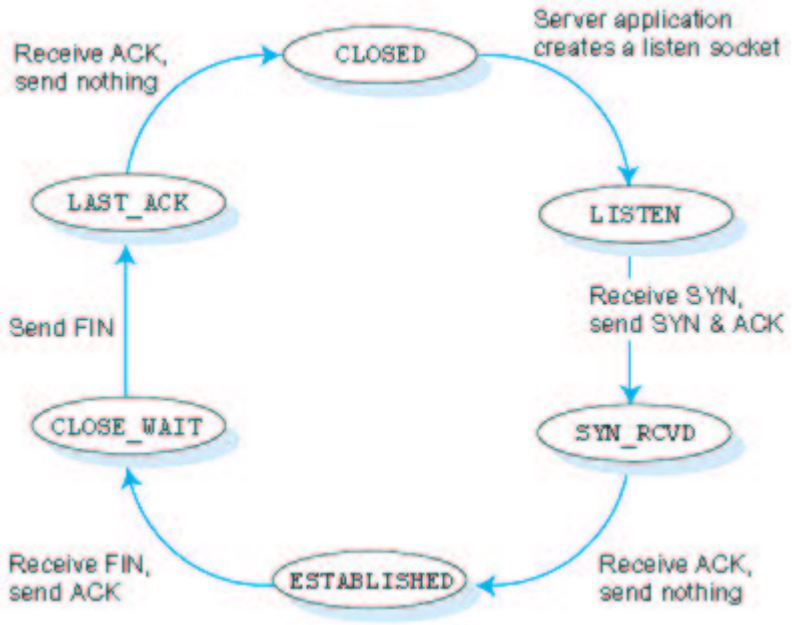


Figure 3.40: A typical sequence of TCP states visited by a server-side TCP

This completes our introduction to TCP. In Section 3.7 we will return to TCP and look at TCP congestion control in some depth. Before doing so, however, we first step back and examine congestion-control issues in a broader context.

Online Book

3.6: Principles of Congestion Control

In the previous sections, we've examined both the general principles and specific TCP mechanisms used to provide for a reliable data-transfer service in the face of packet loss. We mentioned earlier that, in practice, such loss typically results from the overflowing of router buffers as the network becomes congested. Packet retransmission thus treats a symptom of network congestion (the loss of a specific transport-layer segment) but does not treat the cause of network congestion--too many sources attempting to send data at too high a rate. To treat the *cause* of network congestion, mechanisms are needed to throttle senders in the face of network congestion.

In this section, we consider the problem of congestion control in a general context, seeking to understand *why* congestion is a "bad thing," *how* network congestion is manifested in the performance received by upper-layer applications, and various approaches that can be taken to avoid, or react to, network congestion. This more general study of congestion control

is appropriate since, as with reliable data transfer, it is high on the "top-10" list of fundamentally important problems in networking. We conclude this section with a discussion of congestion control in the ABR service in asynchronous transfer mode (ATM) networks. The following section contains a detailed study of TCP's congestion-control algorithm.

3.6.1: The Causes and the Costs of Congestion

Let's begin our general study of congestion control by examining three increasingly complex scenarios in which congestion occurs. In each case, we'll look at why congestion occurs in the first place and at the cost of congestion (in terms of resources not fully utilized and poor performance received by the end systems).

Scenario 1: Two Senders, a Router with Infinite Buffers

We begin by considering perhaps the simplest congestion scenario possible: two hosts (A and B) each have a connection that shares a single hop between source and destination, as shown in Figure 3.41.

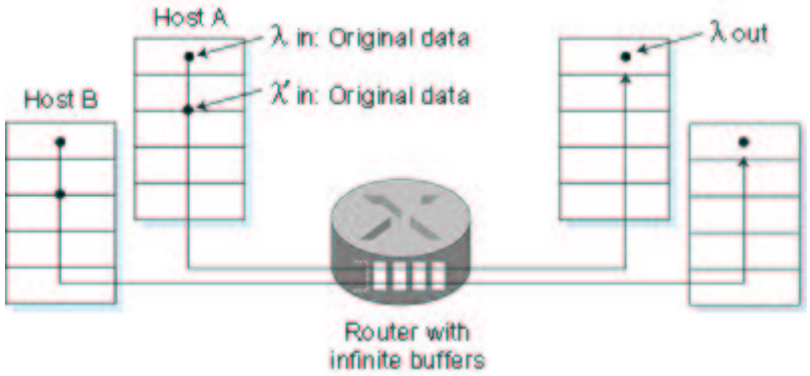


Figure 3.41: Congestion scenario 1: Two connections sharing a single hop with infinite buffers

Let's assume that the application in Host A is sending data into the connection (for example, passing data to the transport-level protocol via a socket) at an average rate of λ_{in} bytes/sec. These data are "original" in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a simple one. Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed. Host B operates in a similar manner, and we assume for simplicity that it too is sending at a rate of λ_{in} bytes/sec. Packets from hosts A and B pass through a router and over a shared outgoing link of capacity *R*. The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity. In this first scenario, we'll assume that the router has an infinite amount of buffer space.

Figure 3.42 plots the performance of Host A's connection under this first scenario. The left graph plots the **per-connection throughput** (number of bytes per second at the receiver) as a function of the connection sending rate. For a sending rate between 0 and *R*/2, the throughput at the receiver equals the sender's sending rate--everything sent by the sender is received

at the receiver with a finite delay. When the sending rate is above *R*/2, however, the throughput is only *R*/2. This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds *R*/2. No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than *R*/2.

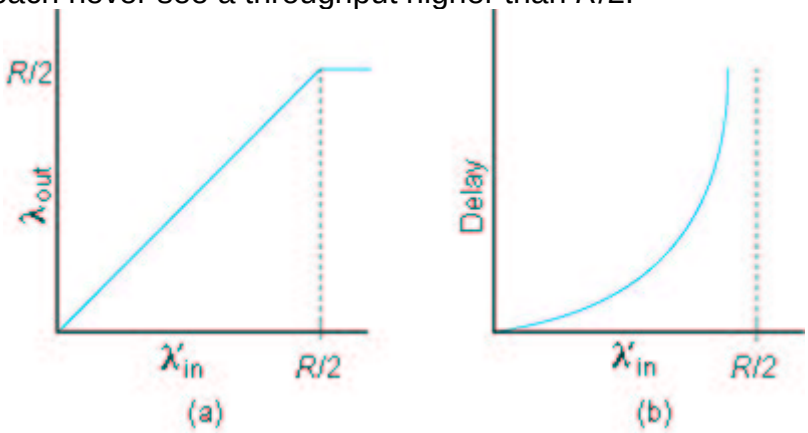


Figure 3.42: Congestion scenario 1: Throughput and delay as a function of host sending rate

Achieving a per-connection throughput of *R*/2 might actually appear to be a "good thing," as the link is fully utilized in delivering packets to their destinations. The right-hand graph in Figure 3.42, however, shows the consequences of operating near link capacity. As the sending rate approaches *R*/2 (from the left), the average delay becomes larger and larger. When the sending rate exceeds *R*/2, the average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite (assuming that the connections operate at these sending rates for an infinite period of time). Thus, while operating at an aggregate throughput of near *R* may be ideal from a throughput standpoint, it is far from ideal from a delay standpoint. *Even in this (extremely) idealized scenario, we've already found one cost of a congested network--large queuing delays are experienced as the packet-arrival rate nears the link capacity.*

Scenario 2: Two Senders, a Router with Finite Buffers

Let us now slightly modify scenario 1 in the following two ways (see Figure 3.43). First, the amount of router buffering is assumed to be finite. Second, we assume that each connection is reliable. If a packet containing a transport-level segment is dropped at the router, it will eventually be retransmitted by the sender. Because packets can be retransmitted, we must now be more careful with our use of the term "sending rate." Specifically, let us again denote the rate at which the application sends original data into the socket by λ_{in} bytes/sec. The rate at which the transport layer sends segments (containing original data *or* retransmitted data) into the network will be denoted λ_{in}' bytes/sec. λ_{in}' is sometimes referred to as the **offered load** to the network.

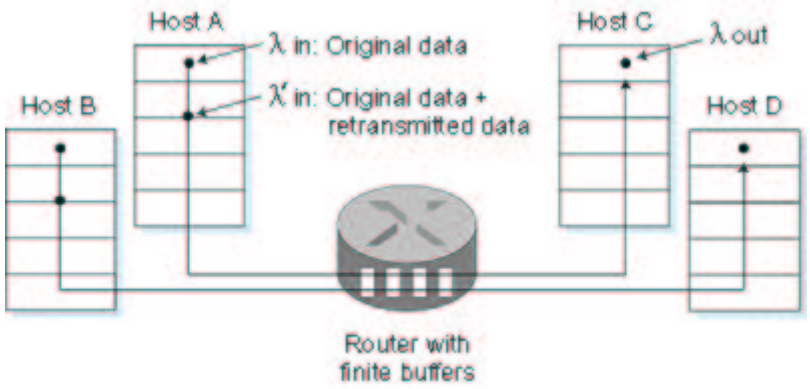


Figure 3.43: Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

The performance realized under scenario 2 will now depend strongly on how retransmission is performed. First, consider the unrealistic case that Host A is able to somehow (magically!) determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free. In this case, no loss would occur, λ_{in} would be equal to λ_{in}', and the throughput of the connection would be equal to λ_{in}. This case is shown by the upper curve in Figure 3.44(a). From a throughput standpoint, performance is ideal--everything that is sent is received. Note that the average host sending rate cannot exceed *R*/2 under this scenario, since packet loss is assumed never to occur.

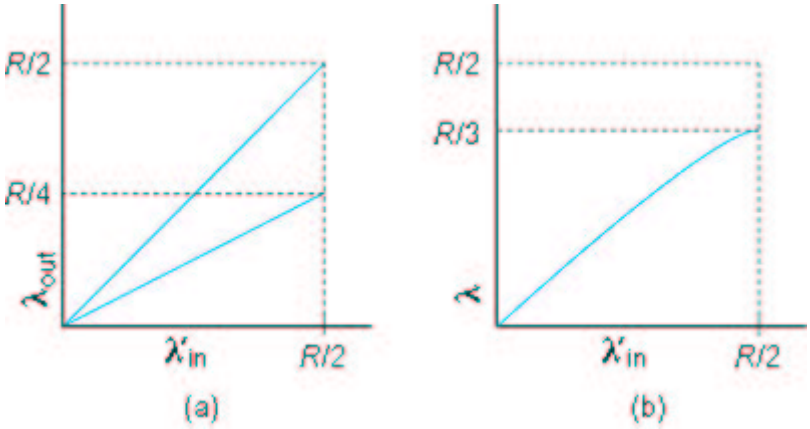


Figure 3.44: Scenario 2 performance

Consider next the slightly more realistic case that the sender retransmits only when a packet is known for certain to be lost. (Again, this assumption is a bit of a stretch. However, it is possible that the sending host might set its timeout large enough to be virtually assured that a packet that has not been acknowledged has been lost.) In this case, the performance might look something like that shown in Figure 3.44(b). To appreciate what is happening here, consider the case that the offered load, λ_{in}' (the rate of original data transmission plus retransmissions), equals 0.5*R*. According to Figure 3.44(b), at this value of the offered load, the rate at which data are delivered to the receiver application is *R*/3. Thus, out of the 0.5*R* units of data transmitted, 0.333*R* bytes/sec (on average) are original data and

0.266*R* bytes per second (on average) are retransmitted data. *We see here another cost of a congested network--the sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.*

Finally, let us consider the case that the sender may timeout prematurely and retransmit a packet that has been delayed in the queue, but not yet lost. In this case, both the original data packet and the retransmission may both reach the receiver. Of course, the receiver needs but one copy of this packet and will discard the retransmission. In this case, the "work" done by the router in forwarding the retransmitted copy of the original packet was "wasted," as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead. *Here then is yet another cost of a congested network--unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet.* The lower curve in Figure 3.44(a) shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router. Since each packet is forwarded twice, the throughput achieved will be given by the line segment in Figure 3.44(a) with the asymptotic value of *R*/4.

Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths

In our final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in Figure 3.45. We again assume that each host uses a timeout/ retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of λ_{in}, and that all router links have capacity *R* bytes/sec.

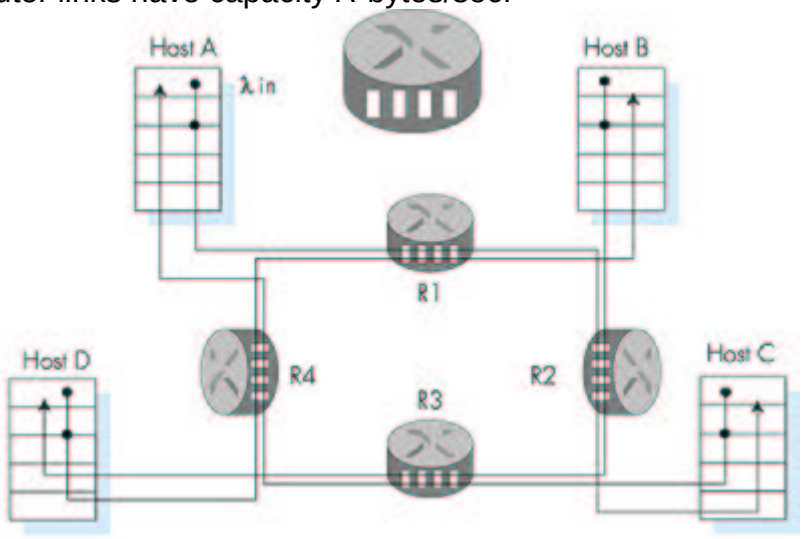


Figure 3.45: Four senders, routers with finite buffers, and multihop paths

Let us consider the connection from Host A to Host C, passing through Routers R1 and R2. The A-C connection shares router R1 with the D-B connection and shares router R2 with the B-D connection. For extremely small values of λ_{in}, buffer overflows are rare (as in congestion scenarios 1

and 2), and the throughput approximately equals the offered load. For slightly larger values of λ_{in}, the corresponding throughput is also larger, as more original data is being transmitted into the network and delivered to the destination, and overflows are still rare. Thus, for small values of λ_{in}, an increase in λ_{in} results in an increase in λ_{out}.

Having considered the case of extremely low traffic, let us next examine the case that λ_{in} (and hence λ_{in}') is extremely large. Consider router R2. The A-C traffic arriving to router R2 (which arrives at R2 after being forwarded from R1) can have an arrival rate at R2 that is at most *R,* the capacity of the link from R1 to R2, regardless of the value of λ_{in}. If λ_{in}' is extremely large for all connections (including the B-D connection), then the arrival rate of B-D traffic at R2 can be much larger than that of the A-C traffic. Because the A-C and B-D traffic must compete at router R2 for the limited amount of buffer space, the amount of A-C traffic that successfully gets through R2 (that is, is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B-D gets larger and larger. In the limit, as the offered load approaches infinity, an empty buffer at R2 is immediately filled by a B-D packet, and the throughput of the A-C connection at R2 goes to zero. This, in turn, *implies that the A-C end-end throughput goes to zero* in the limit of heavy traffic. These considerations give rise to the offered load versus throughput tradeoff shown in Figure 3.46.

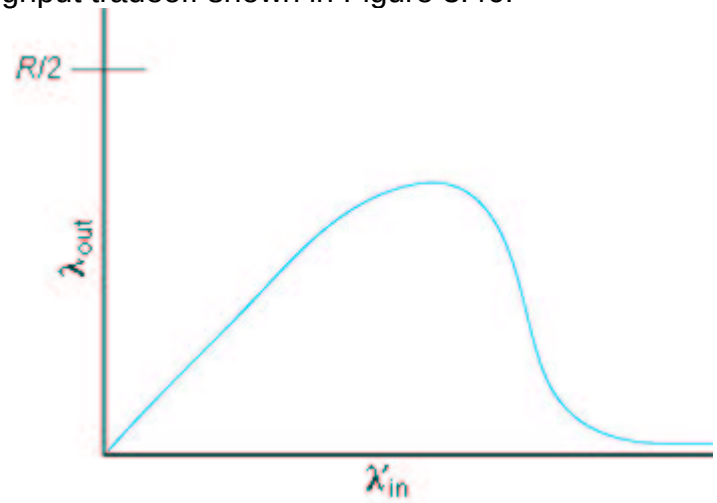


Figure 3.46: Scenario 3 performance with finite buffers and multihop paths

The reason for the eventual decrease in throughput with increasing offered load is evident when one considers the amount of wasted "work" done by the network. In the high-traffic scenario outlined above, whenever a packet is dropped at a second-hop router, the "work" done by the first-hop router in forwarding a packet to the second-hop router ends up being "wasted." The network would have been equally well off (more accurately, equally bad off) if the first router had simply discarded that packet and remained idle. More to the point, the transmission capacity used at the first router to forward the packet to the second router could have been much more profitably used to transmit a different packet. (For example, when selecting a packet for transmission, it might be better for a router to give priority to packets that

have already traversed some number of upstream routers.) *So here we see yet another cost of dropping a packet due to congestion--when a packet is dropped along a path, the transmission capacity that was used at each of the upstream routers to forward that packet to the point at which it is dropped ends up having been wasted.*

3.6.2: Approaches toward Congestion Control

In Section 3.7, we'll examine TCP's specific approach towards congestion control in great detail. Here, we identify the two broad approaches that are taken in practice toward congestion control, and discuss specific network architectures and congestion-control protocols embodying these approaches.

At the broadest level, we can distinguish among congestion-control approaches based on whether or not the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

- *End-end congestion control.* In an end-end approach toward congestion control, the network layer provides *no explicit support* to the transport layer for congestion-control purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). We will see in Section 3.7 that TCP must necessarily take this end-end approach toward congestion control, since the IP layer provides no feedback to the end systems regarding network congestion. TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly. We will also see that new proposals for TCP use increasing round-trip delay values as indicators of increased network congestion.

- *Network-assisted congestion control.* With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. This approach was taken in the early IBM SNA [Schwartz 1982] and DEC DECnet [Jain 1989; Ramakrishnan 1990] architectures, was recently proposed for TCP/IP networks [Floyd TCP 1994; RFC 2481], and is used in ATM available bit-rate (ABR) congestion control as well, as discussed below. More sophisticated network-feedback is also possible. For example, one form of ATM ABR congestion control that we will study shortly allows a router to explicitly inform the sender of the transmission rate it (the router) can support on an outgoing link.

For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in Figure 3.47. Direct feedback may be sent from a network router to the

sender. This form of notification typically takes the form of a **choke packet** (essentially saying, "I'm congested!"). The second form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. Note that this latter form of notification takes at least a full round-trip time.

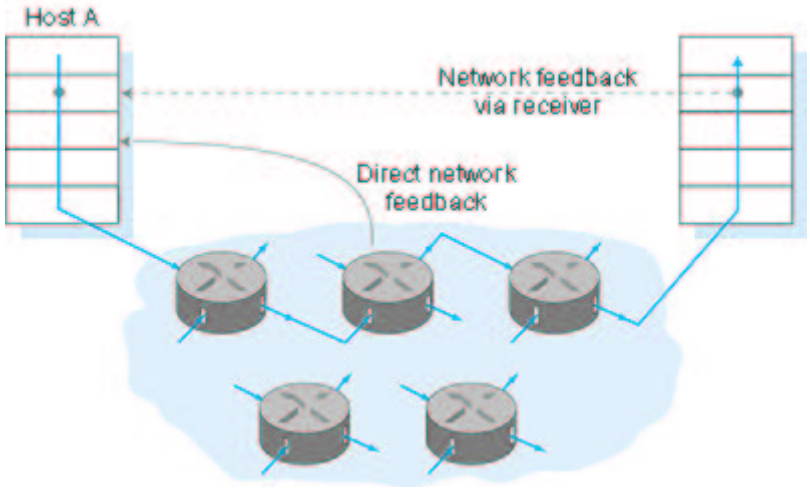


Figure 3.47: Two feedback pathways for network-indicated congestion information

3.6.3: ATM ABR Congestion Control

Our detailed study of TCP congestion control in Section 3.7 will provide an in-depth case study of an end-end approach toward congestion control. We conclude this section with a brief case study of the network-assisted congestion-control mechanisms used in ATM ABR service. ABR has been designed as an elastic data transfer service in a manner reminiscent of TCP. When the network is underloaded, ABR service should be able to take advantage of the spare available bandwidth; when the network is congested, ABR service should throttle its transmission rate to some predetermined minimum transmission rate. A detailed tutorial on ATM ABR congestion control and traffic management is provided in [Jain 1996]. Figure 3.48 shows the framework for ATM ABR congestion control. In our discussion below we adopt ATM terminology (for example, using the term "switch" rather than "router," and the term "cell" rather than "packet"). With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches. Interspersed with the data cells are so-called resource-management cells, **RM cells;** we will see shortly that these RM cells can be used to convey congestion-related information among the hosts and switches. When an RM cell is at a destination, it will be "turned around" and sent back to the sender (possibly after the destination has modified the contents of the RM cell). It is also possible for a switch to generate an RM cell itself and send this RM cell directly to a source. RM cells can thus be used to provide both direct network feedback and network-feedback-via-the-receiver, as shown in Figure 3.48.

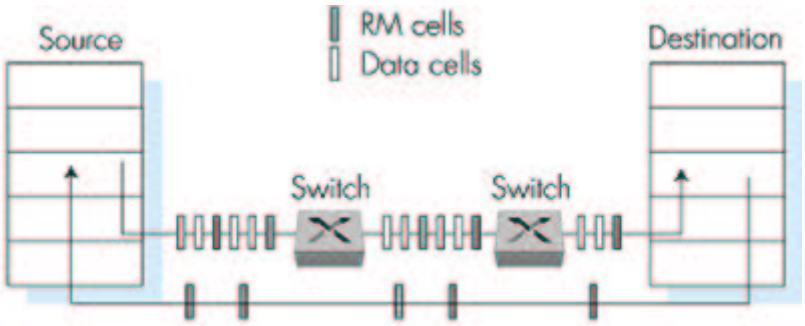


Figure 3.48: Congestion control framework for ATM ABR service

ATM ABR congestion control is a rate-based approach. That is, the sender explicitly computes a maximum rate at which it can send and regulates itself accordingly. ABR provides three mechanisms for signaling congestion-related information from the switches to the receiver:

- *EFCI bit.* Each *data cell* contains an **EFCI** (**explicit forward congestion-indication**) **bit.** A congested network switch can set the EFCI bit in a data cell to 1 to signal congestion to the destination host. The destination must check the EFCI bit in all received data cells. When an RM cell arrives at the destination, if the most recently received data cell had the EFCI bit set to 1, then the destination sets the congestion-indication bit (the CI bit) of the RM cell to 1 and sends the RM cell back to the sender. Using the EFCI in data cells and the CI bit in RM cells, a sender can thus be notified about congestion at a network switch.

- *CI and NI bits.* As noted above, sender-to-receiver RM cells are interspersed with data cells. The rate of RM cell interspersion is a tunable parameter, with one RM cell every 32 data cells being the default value. These RM cells have a CI (congestion indication) bit and an NI (no increase) bit that can be set by a congested-network switch. Specifically, a switch can set the NI bit in a passing RM cell to 1 under mild congestion and can set the CI bit to 1 under severe congestion conditions. When a destination host receives an RM cell, it will send the RM cell back to the sender with its CI and NI bits intact (except that CI may be set to 1 by the destination as a result of the EFCI mechanism described above).

- *ER setting.* Each RM cell also contains a two-byte **ER (explicit rate) field.** A congested switch may lower the value contained in the ER field in a passing RM cell. In this manner, the ER field will be set to the minimum supportable rate of all switches on the source-to-destination path.

An ATM ABR source adjusts the rate at which it can send cells as a function of the CI, NI, and ER values in a returned RM cell. The rules for making this rate adjustment are rather complicated and a bit tedious. The interested reader is referred to] for details.

Online Book

3.7: TCP Congestion Control

In this section we return to our study of TCP. As we learned in Section 3.5, TCP provides a reliable transport service between two processes running on different hosts. Another extremely important component of TCP is its congestion-control mechanism. As we indicated in the previous section, TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion. Before diving into the details of TCP congestion control, let's first get a high-level view of TCP's congestion-control mechanism, as well as the overall goal that TCP strives for when multiple TCP connections must share the bandwidth of a congested link.

A TCP connection controls its transmission rate by limiting its number of transmitted-but-yet-to-be-acknowledged segments. Let us denote this number of permissible unacknowledged segments as *w*, often referred to as the TCP **window size.** Ideally, TCP connections should be allowed to transmit as fast as possible (that is, to have as large a number of outstanding unacknowledged segments as possible) as long as segments are not lost (dropped at routers) due to congestion. In very broad terms, a TCP connection starts with a small value of *w* and then "probes" for the existence of additional unused link bandwidth at the links on its end-to-end path by increasing *w*. A TCP connection continues to increase *w* until a segment loss occurs (as detected by a timeout or duplicate acknowledgments). When such a loss occurs, the TCP connection reduces *w* to a "safe level" and then begins probing again for unused bandwidth by slowly increasing *w*.

An important measure of the performance of a TCP connection is its throughput--the rate at which it transmits data from the sender to the receiver. Clearly, throughput will depend on the value of *w*. If a TCP sender transmits all *w* segments back to back, it must then wait for one round-trip time (RTT) until it receives acknowledgments for these segments, at which point it can send *w* additional segments. If a connection transmits *w* segments of size *MSS* bytes every *RTT* seconds, then the connection's throughput, or transmission rate, is $(w \cdot MSS)/RTT$ bytes per second.

Suppose now that *K* TCP connections are traversing a link of capacity *R*. Suppose also that there are no UDP packets flowing over this link, that each TCP connection is transferring a very large amount of data and that none of these TCP connections traverse any other congested link. Ideally, the window sizes in the TCP connections traversing this link should be such that each connection achieves a throughput of R/K. More generally, if a

connection passes through N links, with link n having transmission rate R_n and supporting a total of K_n TCP connections, then ideally this connection should achieve a rate of R_n/K_n on the nth link. However, this connection's end-to-end average rate cannot exceed the minimum rate achieved at all of the links along the end-to-end path. That is, the end-to-end transmission rate for this connection is $r = \min\{R_1/K_1, \ldots, R_N/K_N\}$. We could think of the goal of TCP as providing this connection with this end-to-end rate, r. (In actuality, the formula for r is more complicated, as we should take into account the fact that one or more of the intervening connections may be bottlenecked at some other link that is not on this end-to-end path and hence cannot use their bandwidth share, R_n/K_n. In this case, the value of r would be higher than $\min\{R_1/K_1, \ldots, R_N/K_N\}$. See [Bertsekas 1991].)

3.7.1: Overview of TCP Congestion Control

In Section 3.5 we saw that each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, RcvWin, and so on.) The TCP congestion-control mechanism has each side of the connection keep track of two additional variables: the **congestion window** and the **threshold.** The congestion window, denoted CongWin, imposes an additional constraint on how much traffic a host can send into a connection. Specifically, the amount of unacknowledged data that a host can have within a TCP connection may not exceed the minimum of CongWin and RcvWin, that is:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{CongWin}, \text{RcvWin}\}$$

The threshold, which we discuss in detail below, is a variable that affects how CongWin grows.

Let us now look at how the congestion window evolves throughout the lifetime of a TCP connection. In order to focus on congestion control (as opposed to flow control), let us assume that the TCP receive buffer is so large that the receive window constraint can be ignored. In this case, the amount of unacknowledged data that a host can have within a TCP connection is solely limited by CongWin. Further let's assume that a sender has a very large amount of data to send to a receiver.

Once a TCP connection is established between the two end systems, the application process at the sender writes bytes to the sender's TCP send buffer. TCP grabs chunks of size MSS, encapsulates each chunk within a TCP segment, and passes the segments to the network layer for transmission across the network. The TCP congestion window regulates the times at which the segments are sent into the network (that is, passed to the network layer). Initially, the congestion window is equal to one MSS. TCP sends the first segment into the network and waits for an acknowledgment. If this segment is acknowledged before its timer times out, the sender increases the congestion window by one MSS and sends out two maximum-size segments. If these segments are acknowledged before their timeouts, the sender increases the congestion window by one

MSS for each of the acknowledged segments, giving a congestion window of four MSS, and sends out four maximum-sized segments. This procedure continues as long as (1) the congestion window is below the threshold and (2) the acknowledgments arrive before their corresponding timeouts.
During this phase of the congestion-control procedure, the congestion window increases exponentially fast. The congestion window is initialized to one MSS; after one RTT, the window is increased to two segments; after two round-trip times, the window is increased to four segments; after three round-trip times, the window is increased to eight segments, and so forth. This phase of the algorithm is called **slow start** because it begins with a small congestion window equal to one MSS. (The transmission rate of the connection starts slowly but accelerates rapidly.)
The slow-start phase ends when the window size exceeds the value of threshold. Once the congestion window is larger than the current value of threshold, the congestion window grows linearly rather than exponentially. Specifically, if *w* is the current value of the congestion window, and *w* is larger than threshold, then after *w* acknowledgments have arrived, TCP replaces *w* with *w* + 1. This has the effect of increasing the congestion window by 1 in each RTT for which an entire window's worth of acknowledgments arrives. This phase of the algorithm is called **congestion avoidance.**
The congestion-avoidance phase continues as long as the acknowledgments arrive before their corresponding timeouts. But the window size, and hence the rate at which the TCP sender can send, cannot increase forever. Eventually, the TCP rate will be such that one of the links along the path becomes saturated, at which point loss (and a resulting timeout at the sender) will occur. When a timeout occurs, the value of threshold is set to half the value of the current congestion window, and the congestion window is reset to one MSS. The sender then again grows the congestion window exponentially fast using the slow-start procedure until the congestion window hits the threshold.
In summary:

- When the congestion window is below the threshold, the congestion window grows exponentially.
- When the congestion window is above the threshold, the congestion window grows linearly.
- Whenever there is a timeout, the threshold is set to one-half of the current congestion window and the congestion window is then set to 1.

If we ignore the slow-start phase, we see that TCP essentially increases its window size by 1 each RTT (and thus increases its transmission rate by an additive factor) when its network path is not congested, and decreases its window size by a factor of 2 each RTT when the path is congested. For this reason, TCP is often referred to as an **additive-increase, multiplicative-**

decrease (AIMD) algorithm.
The evolution of TCP's congestion window is illustrated in Figure 3.49. In this figure, the threshold is initially equal to 8 • *MSS.* The congestion window climbs exponentially fast during slow start and hits the threshold at the third transmission. The congestion window then climbs linearly until loss occurs, just after transmission 7. Note that the congestion window is 12 • *MSS* when loss occurs. The threshold is then set to 0.5 • CongWin = 6 • *MSS* and the congestion window is set 1. And the process continues. This congestion-control algorithm is due to V. Jacobson [Jacobson 1988]; a number of modifications to Jacobson's initial algorithm are described in Stevens (1994) and in RFC 2581.

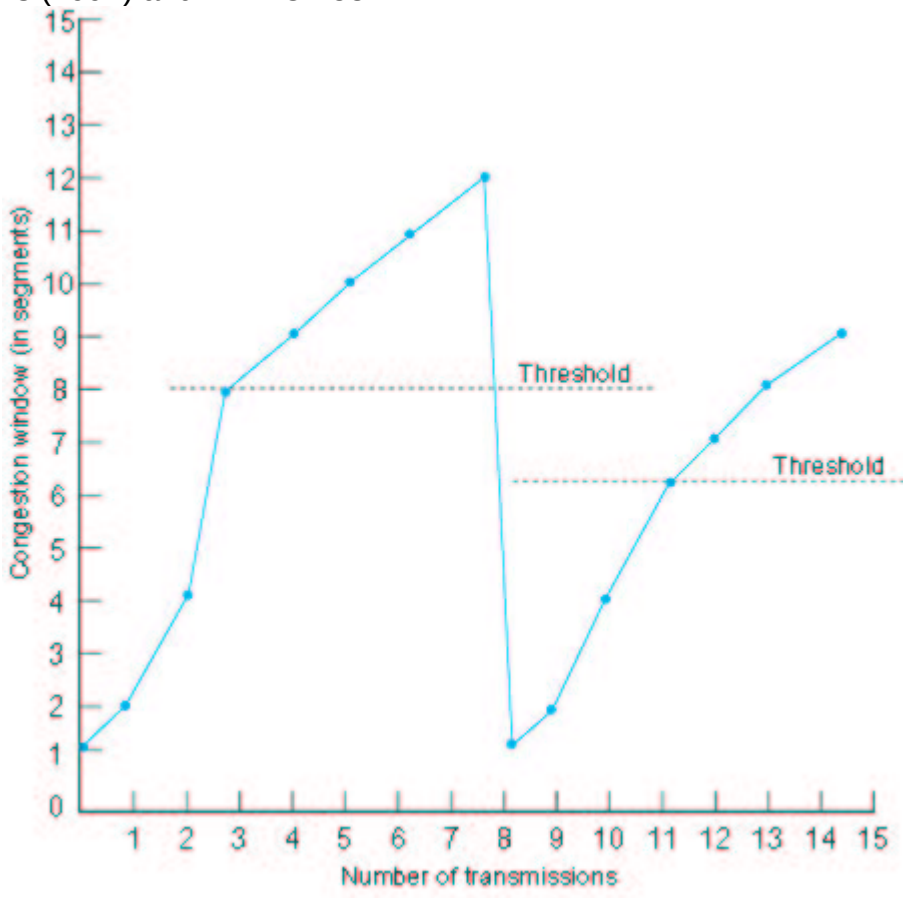


Figure 3.49: Evolution of TCP's congestion window

We note briefly here that the description of TCP slow start is an idealized one. An initial window of up to two MSS's is a proposed standard [RFC 2581] and it is actually used in some implementations.

A Trip to Nevada: Tahoe, Reno, and Vegas

The TCP congestion-control algorithm just described is often referred to as **Tahoe.** One problem with the Tahoe algorithm is that, when a segment is lost, the sender side of the application may have to wait a long period of time for the timeout. For this reason, a variant of Tahoe, called **Reno,** is implemented by most operating systems. Like Tahoe, Reno sets its congestion window to one segment upon the expiration of a timer.

However, Reno also includes the fast retransmit mechanism that we examined in Section 3.5. Recall that fast retransmission triggers the transmission of a dropped segment if three duplicate ACKs for a segment are received before the occurrence of the segment's timeout. Reno also employs a **fast-recovery** mechanism that essentially cancels the slow-start phase after a fast retransmission. The interested reader is encouraged to see [Stevens 1994] and in [RFC 2581] for details. [Cela 2000] provides interactive animations of congestion avoidance, slow start, fast retransmit, and fast recovery in TCP.

Most TCP implementations currently use the Reno algorithm. There is, however, another algorithm in the literature, the Vegas algorithm, that can improve Reno's performance. Whereas Tahoe and Reno react to congestion (that is, to overflowing router buffers), Vegas attempts to avoid congestion while maintaining good throughput. The basic idea of Vegas is to (1) detect congestion in the routers between source and destination *before* packet loss occurs and (2) lower the rate linearly when this imminent packet loss is detected. Imminent packet loss is predicted by observing the round-trip times. The longer the round-trip times of the packets, the greater the congestion in the routers. The Vegas algorithm is discussed in detail in [Brakmo 1995]; a study of its performance is given in [Ahn 1995]. As of 1999, Vegas is not a part of the most popular TCP implementations.

We emphasize that TCP congestion control has evolved over the years, and is still evolving. What was good for the Internet when the bulk of the TCP connections carried SMTP, FTP, and Telnet traffic is not necessarily good for today's Web-dominated Internet or for the Internet of the future, which will support who-knows-what kinds of services.

Does TCP Ensure Fairness?

In the above discussion, we noted that a goal of TCP's congestion-control mechanism is to share a bottleneck link's bandwidth evenly among the TCP connections that are bottlenecked at that link. But why should TCP's additive-increase, multiplicative-decrease algorithm achieve that goal, particularly given that different TCP connections may start at different times and thus may have different window sizes at a given point in time? [Chiu 1989] provides an elegant and intuitive explanation of why TCP congestion control converges to provide an equal share of a bottleneck link's bandwidth among competing TCP connections.

Let's consider the simple case of two TCP connections sharing a single link with transmission rate R, as shown in Figure 3.50. We'll assume that the two connections have the same MSS and RTT (so that if they have the same congestion window size, then they have the same throughput), that they have a large amount of data to send, and that no other TCP connections or UDP datagrams traverse this shared link. Also, we'll ignore the slow-start phase of TCP and assume the TCP connections are operating in congestion-avoidance mode (additive-increase, multiplicative-decrease) at all times.

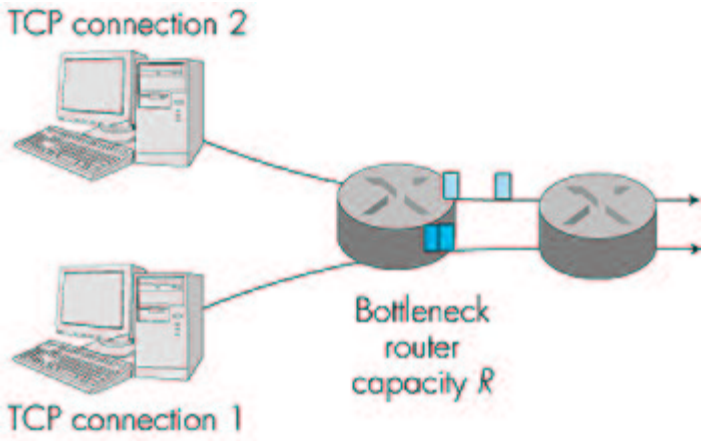


Figure 3.50: Two TCP connections sharing a single bottleneck link

Figure 3.51 plots the throughput realized by the two TCP connections. If TCP is to equally share the link bandwidth between the two connections, then the realized throughput should fall along the 45-degree arrow ("equal bandwidth share") emanating from the origin. Ideally, the sum of the two throughputs should equal *R*. (Certainly, each connection receiving an equal, but zero, share of the link capacity is not a desirable situation!) So the goal should be to have the achieved throughputs fall somewhere near the intersection of the "equal bandwidth share" line and the "full bandwidth utilization" line in Figure 3.51.

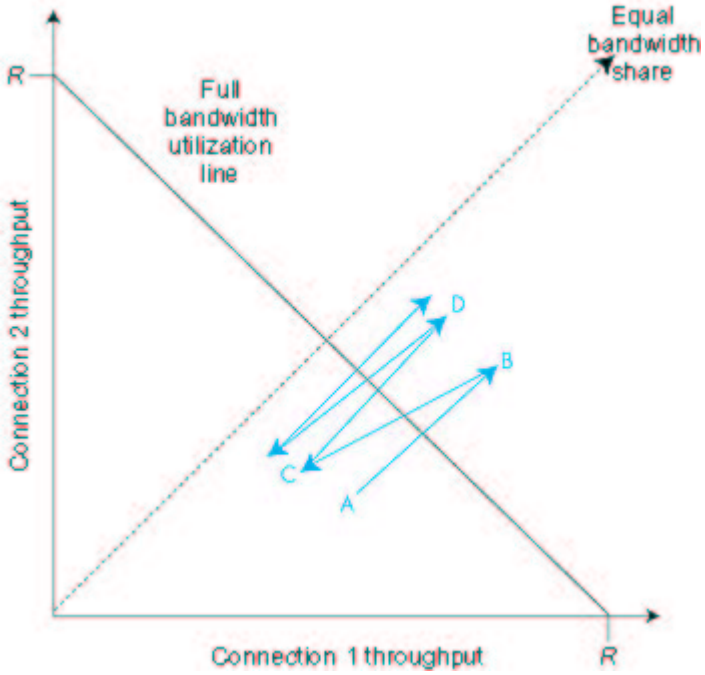


Figure 3.51: Throughput realized by TCP connections 1 and 2

Suppose that the TCP window sizes are such that at a given point in time, connections 1 and 2 realize throughputs indicated by point *A* in Figure 3.51. Because the amount of link bandwidth jointly consumed by the two connections is less than *R*, no loss will occur, and both connections will increase their window by 1 per RTT as a result of TCP's congestion-

avoidance algorithm. Thus, the joint throughput of the two connections proceeds along a 45-degree line (equal increase for both connections) starting from point *A*. Eventually, the link bandwidth jointly consumed by the two connections will be greater than *R* and eventually packet loss will occur. Suppose that connections 1 and 2 experience packet loss when they realize throughputs indicated by point *B*. Connections 1 and 2 then decrease their windows by a factor of two. The resulting throughputs realized are thus at point *C*, halfway along a vector starting at *B* and ending at the origin. Because the joint bandwidth use is less than *R* at point *C*, the two connections again increase their throughputs along a 45-degree line starting from *C*. Eventually, loss will again occur, for example, at point *D*, and the two connections again decrease their window sizes by a factor of two, and so on. You should convince yourself that the bandwidth realized by the two connections eventually fluctuates along the equal bandwidth share line. You should also convince yourself that the two connections will converge to this behavior regardless of where they are in the two-dimensional space! Although a number of idealized assumptions lay behind this scenario, it still provides an intuitive feel for why TCP results in an equal sharing of bandwidth among connections.

In our idealized scenario, we assumed that only TCP connections traverse the bottleneck link, and that only a single TCP connection is associated with a host-destination pair. In practice, these two conditions are typically not met, and client/ server applications can thus obtain very unequal portions of link bandwidth.

Many network applications run over TCP rather than UDP because they want to make use of TCP’s reliable transport service. But an application developer choosing TCP gets not only reliable data transfer but also TCP congestion control. We have just seen how TCP congestion control regulates an application’s transmission rate via the congestion-window mechanism. Many multimedia applications do not run over TCP for this very reason--they do not want their transmission rate throttled, even if the network is very congested. In particular, many Internet telephone and Internet video conferencing applications typically run over UDP. These applications prefer to pump their audio and video into the network at a constant rate and occasionally lose packets, rather than reduce their rates to "fair" levels at times of congestion and not lose any packets. From the perspective of TCP, the multimedia applications running over UDP are not being fair--they do not cooperate with the other connections nor adjust their transmission rates appropriately. A major challenge in the upcoming years will be to develop congestion-control mechanisms for the Internet that prevent UDP traffic from bringing the Internet’s throughput to a grinding halt, [Floyd 1999].

But even if we could force UDP traffic to behave fairly, the fairness problem would still not be completely solved. This is because there is nothing to stop an application running over TCP from using multiple parallel connections. For example, Web browsers often use multiple parallel TCP connections to

transfer a Web page. (The exact number of multiple connections is configurable in most browsers.) When an application uses multiple parallel connections, it gets a larger fraction of the bandwidth in a congested link. As an example, consider a link of rate *R* supporting nine ongoing client/server applications, with each of the applications using one TCP connection. If a new application comes along and also uses one TCP connection, then each application gets approximately the same transmission rate of *R*/10. But if this new application instead uses 11 parallel TCP connections, then the new application gets an unfair allocation of more than *R*/2. Because Web traffic is so pervasive in the Internet, multiple parallel connections are not uncommon.

Macroscopic Description of TCP Dynamics

Consider sending a very large file over a TCP connection. If we take a macroscopic view of the traffic sent by the source, we can ignore the slow-start phase. Indeed, the connection is in the slow-start phase for a relatively short period of time because the connection grows out of the phase exponentially fast. When we ignore the slow-start phase, the congestion window grows linearly, gets chopped in half when loss occurs, grows linearly, gets chopped in half when loss occurs, and so on. This gives rise to the saw-tooth behavior of TCP [Stevens 1994] shown in Figure 3.49. Given this saw-tooth behavior, what is the average throughput of a TCP connection? During a particular round-trip interval, the rate at which TCP sends data is a function of the congestion window and the current *RTT*. When the window size is *w* • *MSS* and the current round-trip time is *RTT,* then TCP's transmission rate is (*w* • *MSS*)/*RTT.* During the congestion-avoidance phase, TCP probes for additional bandwidth by increasing *w* by one each *RTT* until loss occurs. (Denote by *W* the value of *w* at which loss occurs.) Assuming that *RTT* and *W* are approximately constant over the duration of the connection, the TCP transmission rate ranges from

$$\frac{W \cdot MSS}{2RTT} \text{ to } \frac{W \cdot MSS}{RTT}$$

These assumptions lead to a highly simplified macroscopic model for the steady-state behavior of TCP. The network drops a packet from the connection when the connection's window size increases to *W* • *MSS;* the congestion window is then cut in half and then increases by one MSS per round-trip time until it again reaches *W*. This process repeats itself over and over again. Because the TCP throughput increases linearly between the two extreme values, we have:

$$\text{Average throughput of a connection} = \frac{0.75 \cdot W \cdot MSS}{RTT}$$

Using this highly idealized model for the steady-state dynamics of TCP, we can also derive an interesting expression that relates a connection's loss rate to its available bandwidth [Mahdavi 1997]. This derivation is outlined in the homework problems.

3.7.2: Modeling Latency: Static Congestion Window

Many TCP connections transport relatively small files from one host to another. For example, with HTTP/1.0, each object in a Web page is transported over a separate TCP connection, and many of these objects are small text files or tiny icons. When transporting a small file, TCP connection establishment and slow start may have a significant impact on the latency. In this section we present an analytical model that quantifies the impact of connection establishment and slow start on latency. For a given object, we define the **latency** as the time from when the client initiates a TCP connection until the time at which the client receives the requested object in its entirety.

The analysis presented here assumes that the network is uncongested, that is, that the TCP connection transporting the object does not have to share link bandwidth with other TCP or UDP traffic. (We comment on this assumption below.) Also, in order to not obscure the central issues, we carry out the analysis in the context of the simple one-link network as shown in Figure 3.52. (This link might model a single bottleneck on an end-to-end path. See also the homework problems for an explicit extension to the case of multiple links.)

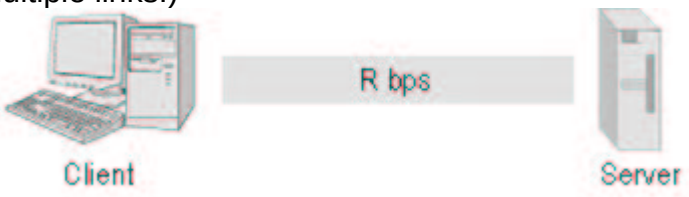


Figure 3.52: A simple one-link network connecting a client and a server

We also make the following simplifying assumptions:

- The amount of data that the sender can transmit is solely limited by the sender's congestion window. (Thus, the TCP receive buffers are large.)
- Packets are neither lost nor corrupted, so that there are no retransmissions.
- All protocol header overheads--including TCP, IP, and link-layer headers--are negligible and ignored.
- The object (that is, file) to be transferred consists of an integer number of segments of size MSS (maximum segment size).
- The only packets that have non-negligible transmission times are packets that carry maximum-size TCP segments. Request messages, acknowledgments, and TCP connection establishment segments are small and have negligible transmission times.
- The initial threshold in the TCP congestion-control mechanism is a large value that is never attained by the congestion window.

We also introduce the following notation:

- The size of the object to be transferred is *O* bits.
- The MSS (maximum size segment) is *S* bits (for example, 536 bytes).
- The transmission rate of the link from the server to the client is *R* bps.
- The round-trip time is denoted by *RTT.*

In this section we define the *RTT* to be the time elapsed for a small packet to travel from client to server and then back to the client, *excluding the transmission time of the packet.* It includes the two end-to-end propagation delays between the two end systems and the processing times at the two end systems. We shall assume that the RTT is also equal to the roundtrip time of a packet beginning at the server.

Although the analysis presented in this section assumes an uncongested network with a single TCP connection, it nevertheless sheds insight on the more realistic case of multilink congested network. For a congested network, *R* roughly represents the amount of bandwidth received in steady state in the end-to-end network connection, and RTT represents a round-trip delay that includes queuing delays at the routers preceding the congested links. In the congested network case, we model each TCP connection as a constant-bit-rate connection of rate *R* bps preceded by a single slow-start phase. (This is roughly how TCP Tahoe behaves when losses are detected with triple duplicate acknowledgments.) In our numerical examples, we use values of *R* and *RTT* that reflect typical values for a congested network.

Before beginning the formal analysis, let us try to gain some intuition. Let us consider what would be the latency if there were no congestion-window constraint; that is, if the server were permitted to send segments back-to-back until the entire object is sent. To answer this question, first note that one RTT is required to initiate the TCP connection. After one RTT, the client sends a request for the object (which is piggybacked onto the third segment in the three-way TCP handshake). After a total of two RTTs, the client begins to receive data from the server. The client receives data from the server for a period of time *O*/*R*, the time for the server to transmit the entire object. Thus, in the case of no congestion-window constraint, the total latency is 2 *RTT* + *O*/*R*. This represents a lower bound; the slow-start procedure, with its dynamic congestion window, will of course elongate this latency.

Static Congestion Window

Although TCP uses a dynamic congestion window, it is instructive to first analyze the case of a static congestion window. Let *W*, a positive integer, denote a fixed-size static congestion window. For the static congestion window, the server is not permitted to have more than *W* unacknowledged outstanding segments. When the server receives the request from the

client, the server immediately sends *W* segments back-to-back to the client. The server then sends one segment into the network for each acknowledgment it receives from the client. The server continues to send one segment for each acknowledgment until all of the segments of the object have been sent. There are two cases to consider:

1. *WS*/*R* > *RTT* + *S*/*R*. In this case, the server receives an acknowledgment for the first segment in the first window before the server completes the transmission of the first window.

2. *WS*/*R* < *RTT* + *S*/*R*. In this case, the server transmits the first window's worth of segments before the server receives an acknowledgment for the first segment in the window.

Let us first consider case 1, which is illustrated in Figure 3.53. In this figure the window size is *W* = 4 segments.

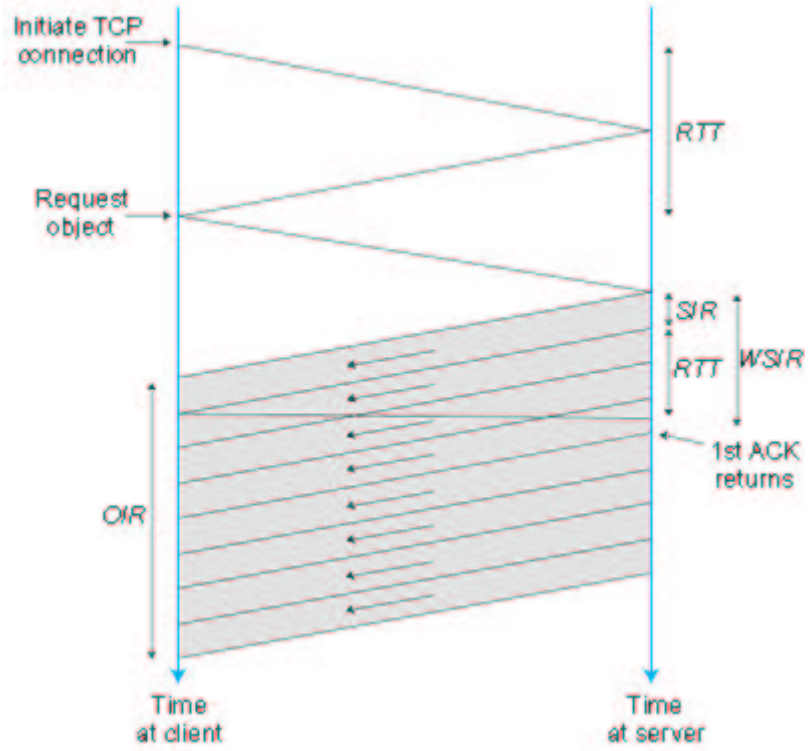


Figure 3.53: The case that *WS/R* > *RTT* + *S/R*

One RTT is required to initiate the TCP connection. After one RTT, the client sends a request for the object (which is piggybacked onto the third segment in the three-way TCP handshake). After a total of two RTTs, the client begins to receive data from the server. Segments arrive periodically from the server every *S*/*R* seconds, and the client acknowledges every segment it receives from the server. Because the server receives the first acknowledgment before it completes sending a window's worth of segments, the server continues to transmit segments after having transmitted the first window's worth of segments. And because the acknowledgments arrive periodically at the server every *S*/*R* seconds from

the time when the first acknowledgment arrives, the server transmits segments continuously until it has transmitted the entire object. Thus, once the server starts to transmit the object at rate *R*, it continues to transmit the object at rate *R* until the entire object is transmitted. The latency therefore is 2 *RTT* + *O*/*R*.

Now let us consider case 2, which is illustrated in Figure 3.54. In this figure, the window size is *W* = 2 segments.

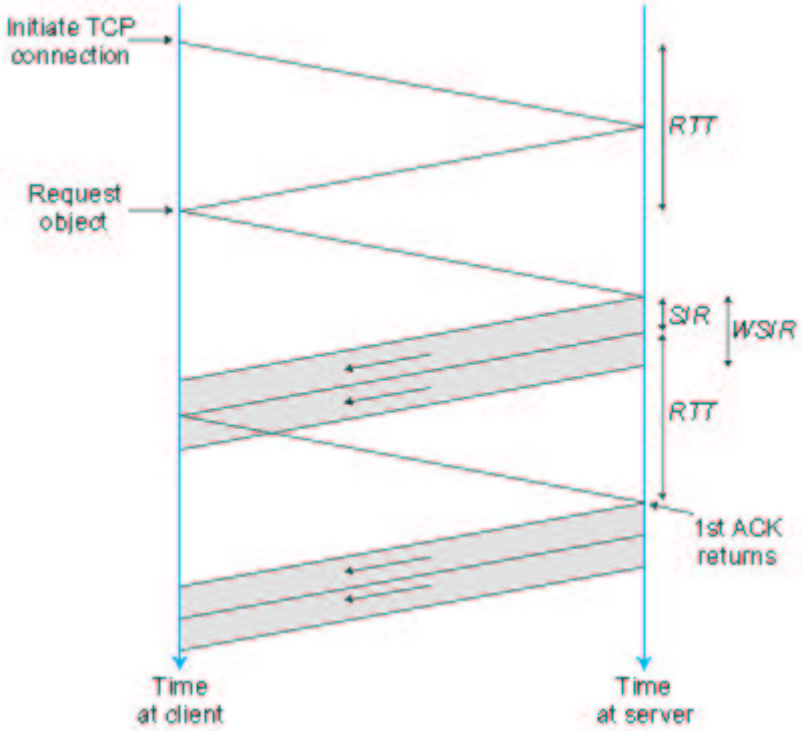


Figure 3.54: The case that *WS/R* < *RTT* + *S/R*

Once again, after a total of two RTTs, the client begins to receive segments from the server. These segments arrive periodically every *S*/*R* seconds, and the client acknowledges every segment it receives from the server. But now the server completes the transmission of the first window before the first acknowledgment arrives from the client. Therefore, after sending a window, the server must stall and wait for an acknowledgment before resuming transmission. When an acknowledgment finally arrives, the server sends a new segment to the client. Once the first acknowledgment arrives, a window's worth of acknowledgments arrive, with each successive acknowledgment spaced by *S*/*R* seconds. For each of these acknowledgments, the server sends exactly one segment. Thus, the server alternates between two states: a transmitting state, during which it transmits *W* segments, and a stalled state, during which it transmits nothing and waits for an acknowledgment. The latency is equal to 2 *RTT* plus the time required for the server to transmit the object, *O*/*R*, plus the amount of time that the server is in the stalled state. To determine the amount of time the server is in the stalled state, let *K* = *O*/*WS*; if *O*/*WS* is not an integer, then round *K* up to the nearest integer. Note that *K* is the number of windows of data there are in the object of size *O*. The server is in the stalled state

between the transmission of each of the windows, that is, for K - 1 periods of time, with each period lasting RTT - (W - 1)S/R (see Figure 3.54). Thus, for case 2,

$$\text{Latency} = 2\,RTT + O/R + (K - 1)\,[S/R + RTT - WS/R]$$

Combining the two cases, we obtain

$$\text{Latency} = 2\,RTT + O/R + (K - 1)\,[S/R + RTT - WS/R]^{+}$$

where $[x]^{+} = \max(x,0)$.

This completes our analysis of static windows. The following analysis for dynamic windows is more complicated, but parallels that for static windows.

3.7.3: Modeling Latency: Dynamic Congestion Window

We now investigate the latency for a file transfer when TCP's dynamic congestion window is in force. Recall that the server first starts with a congestion window of one segment and sends one segment to the client. When it receives an acknowledgment for the segment, it increases its congestion window to two segments and sends two segments to the client (spaced apart by S/R seconds). As it receives the acknowledgments for the two segments, it increases the congestion window to four segments and sends four segments to the client (again spaced apart by S/R seconds). The process continues, with the congestion window doubling every RTT. A timing diagram for TCP is illustrated in Figure 3.55.

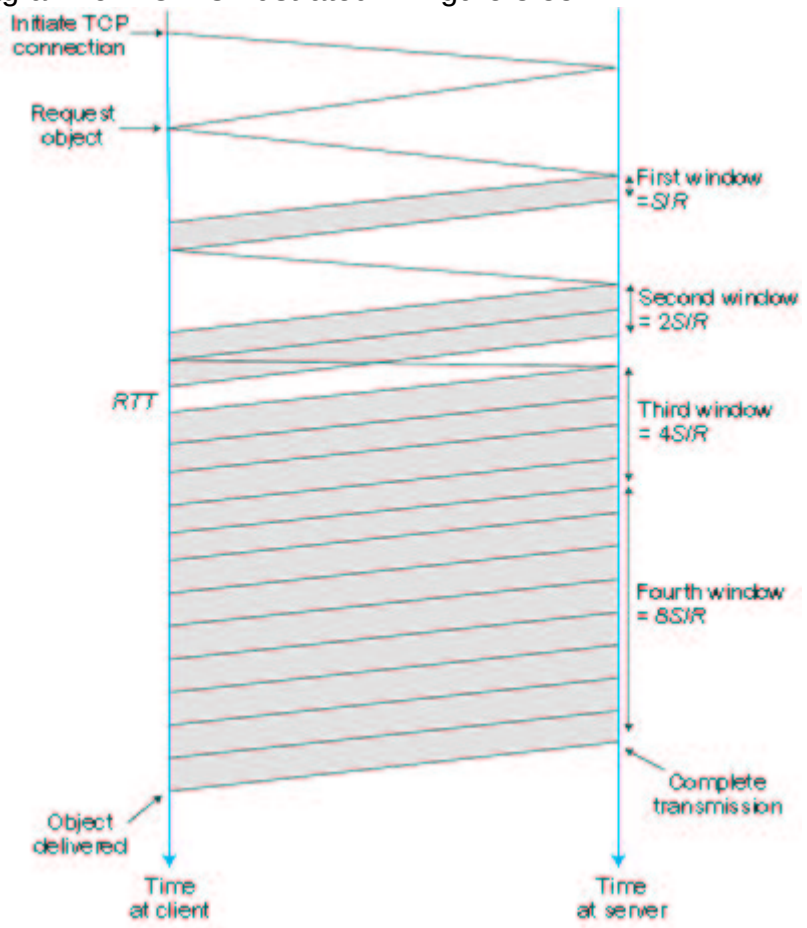


Figure 3.55: TCP timing during slow start

Note that O/S is the number of segments in the object; in the above diagram, $O/S = 15$. Consider the number of segments that are in each of the windows. The first window contains one segment, the second window contains two segments, and the third window contains four segments. More generally, the kth window contains 2^{k-1} segments. Let K be the number of windows that cover the object; in the preceding diagram, $K = 4$. In general, we can express K in terms of O/S as follows:

$$\begin{aligned} K &= \min\left\{k: 2^0 + 2^1 + \ldots + 2^{k-1} \geq \frac{O}{S}\right\} \\ &= \min\left\{k: 2^k - 1 \geq \frac{O}{S}\right\} \\ &= \min\left\{k: k \geq \log_2\left(\frac{O}{S} + 1\right)\right\} \\ &= \left\lceil \log_2\left(\frac{O}{S} + 1\right)\right\rceil \end{aligned}$$

After transmitting a window's worth of data, the server may stall (that is, stop transmitting) while it waits for an acknowledgment. In Figure 3.55, the server stalls after transmitting the first and second windows, but not after transmitting the third. Let us now calculate the amount of stall time after transmitting the kth window. The time the server begins to transmit the kth window until the time when the server receives an acknowledgment for the first segment in the window is $S/R + RTT$. The transmission time of the kth window is $(S/R)\ 2^{k-1}$. The stall time is the difference of these two quantities, that is,

$$[S/R + RTT - 2^{k-1}\ (S/R)]^+.$$

The server can potentially stall after the transmission of each of the first $k - 1$ windows. (The server is done after the transmission of the kth window.) We can now calculate the latency for transferring the file. The latency has three components: 2 RTT for setting up the TCP connection and requesting the file, O/R, the transmission time of the object, and the sum of all the stalled times. Thus,

$$\text{Latency} = 2RTT + \frac{O}{R} + \sum_{k=1}^{K-1}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^+$$

The reader should compare the above equation for the latency equation for static congestion windows; all the terms are exactly the same except that the term WS/R for static windows has been replaced by $2^{k-1}(S/R)$ for dynamic windows. To obtain a more compact expression for the latency, let Q be the number of times the server would stall if the object contained an infinite number of segments:

$$
\begin{aligned}
Q &= \max\left\{k : RTT + \frac{S}{R} - \frac{S}{R}2^{k-1} \geq 0\right\} \\
&= \max\left\{k : 2^{k-1} \leq 1 + \frac{RTT}{S/R}\right\} \\
&= \max\left\{k : k \leq \log_2\left(1 + \frac{RTT}{S/R}\right) + 1\right\} \\
&= \left\lceil \log_2\left(1 + \frac{RTT}{S/R}\right)\right\rceil + 1.
\end{aligned}
$$

The actual number of times the server stalls is $P = \min\{Q, K-1\}$. In Figure 3.55, $P = Q = 2$. Combining the above two equations gives

$$
\text{Latency} = \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left(RTT + \frac{S}{R} - \frac{S}{R}2^{k-1}\right)
$$

We can further simplify the above formula for latency by noting

$$
\sum_{k=1}^{P} 2^{k-1} = 2^P - 1
$$

Combining the above two equations gives the following closed-form expression for the latency:

$$
\text{Latency} = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - \left(2^P - 1\right)\frac{S}{R}
$$

Thus to calculate the latency, we simply must calculate K and Q, set $P = \min\{Q, K-1\}$, and plug P into the above formula.

It is interesting to compare the TCP latency to the latency that would occur if there were no congestion control (that is, no congestion window constraint). Without congestion control, the latency is $2\,RTT + O/R$, which we define to be the *minimum latency.* It is a simple exercise to show that

$$
\frac{\text{Latency}}{\text{MinimumLatency}} \leq 1 + \frac{P}{[(O/R)/RTT] + 2]}
$$

We see from the above formula that TCP slow start will not significantly increase latency if $RTT << O/R$, that is, if the round-trip time is much less than the transmission time of the object. Thus, if we are sending a relatively large object over an uncongested high-speed link, then slow start has an insignificant effect on latency. However, with the Web, we are often transmitting many small objects over congested links, in which case slow start can significantly increase latency (as we'll see in the following subsection).

Let us now take a look at some example scenarios. In all the scenarios we set $S = 536$ bytes, a common default value for TCP. We'll use an RTT of 100 msec, which is not an atypical value for a continental or intercontinental

delay over moderately congested links. First consider sending a rather large object of size O = 100 Kbytes. The number of windows that cover this object is K = 8. For a number of transmission rates, the following table examines the effect of the slow-start mechanism on the latency.

R
O/R
P
Minimum Latency: *O/R* + 2 *RTT*
Latency with slow start

28 Kbps
28.6 sec
1
28.8 sec
28.9 sec

100 Kbps
8 sec
2
8.2 sec
8.4 sec

1 Mbps
800 msec
5
1 sec
1.5 sec

10 Mbps
80 msec
7
0.28 sec
0.98 sec

We see from the above chart that for a large object, slow-start adds appreciable delay only when the transmission rate is high. If the transmission rate is low, then acknowledgments come back relatively quickly, and TCP quickly ramps up to its maximum rate. For example, when R = 100 Kbps, the number of stall periods is P = 2 whereas the number of windows to transmit is K = 8; thus the server stalls only after the first two of eight windows. On the other hand, when R = 10 Mbps, the server stalls between each window, which causes a significant increase in the delay. Now consider sending a small object of size O = 5 Kbytes. The number of windows that cover this object is K = 4. For a number of transmission rates, the following table examines the effect of the slow-start mechanism.

R
O/R
P
Minimum latency: *O/R* + 2 *RTT*
Latency with slow start

28 Kbps
1.43 sec

1
1.63 sec
1.73 sec

100 Kbps
0.4 sec
2
0.6 sec
0.757 sec

1 Mbps
40 msec
3
0.24 sec
0.52 sec

10 Mbps
4 msec
3
0.20 sec
0.50 sec

Once again, slow start adds an appreciable delay when the transmission rate is high. For example, when *R* = 1 Mbps, the server stalls between each window, which causes the latency to be more than twice that of the minimum latency.
For a larger *RTT,* the effect of slow start becomes significant for small objects for smaller transmission rates. The following table examines the effect of slow start for RTT = 1 second and *O* = 5 Kbytes (*K* = 4).
R
O/R
<STRONG*P*
Minimum latency: *O/R* + 2 *RTT*
Latency with slow start

28 Kbps
1.43 sec
3
3.4 sec
5.8 sec

100 Kbps
0.4 sec
3
2.4 sec
5.2 sec

1 Mbps
40 msec
3
2.0 sec
5.0 sec

10 Mbps

4 msec
3
2.0 sec
5.0 sec

In summary, slow start can significantly increase latency when the object size is relatively small and the RTT is relatively large. Unfortunately, this is often the scenario when sending objects over the World Wide Web.

An Example: HTTP

As an application of the latency analysis, let's now calculate the response time for a Web page sent over nonpersistent HTTP. Suppose that the page consists of one base HTML page and *M* referenced images. To keep things simple, let us assume that each of the *M* + 1 objects contains exactly *O* bits.

With nonpersistent HTTP, each object is transferred independently, one after the other. The response time of the Web page is therefore the sum of the latencies for the individual objects. Thus

$$\text{Response time} = (M+1)\left\{2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}\right\}$$

Note that the response time for nonpersistent HTTP takes the form:

Response time = (*M* + 1)*O*/*R* + 2(*M* + 1)RTT + latency due to TCP slow-start for each of the *M* + 1 objects.

Clearly, if there are many objects in the Web page and if *RTT* is large, then non-persistent HTTP will have poor response-time performance. In the homework problems, we will investigate the response time for other HTTP transport schemes, including persistent connections and nonpersistent connections with parallel connections. The reader is also encouraged to see [a related analysis.

Online Book

3.8: Summary

We began this chapter by studying the services that a transport-layer protocol can provide to network applications. At one extreme, the transport-layer protocol can be very simple and offer a no-frills service to applications, providing only the multiplexing/demultiplexing function for communicating processes. The Internet's UDP protocol is an example of such a no-frills transport-layer protocol. At the other extreme, a transport-layer protocol can provide a variety of guarantees to applications, such as reliable delivery of data, delay guarantees, and bandwidth guarantees. Nevertheless, the services that a transport protocol can provide are often

constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees to transport-layer segments, then the transport-layer protocol cannot provide delay or bandwidth guarantees for the messages sent between processes.

We learned in Section 3.4 that a transport-layer protocol can provide reliable data transfer even if the underlying network layer is unreliable. We saw that providing reliable data transfer has many subtle points, but that the task can be accomplished by carefully combining acknowledgments, timers, retransmissions, and sequence numbers.

Although we covered reliable data transfer in this chapter, we should keep in mind that reliable data transfer can be provided by link, network, transport, or application-layer protocols. Any of the upper four layers of the protocol stack can implement acknowledgments, timers, retransmissions, and sequence numbers and provide reliable data transfer to the layer above. In fact, over the years, engineers and computer scientists have independently designed and implemented link, network, transport, and application-layer protocols that provide reliable data transfer (although many of these protocols have quietly disappeared).

In Section 3.5 we took a close look at TCP, the Internet's connection-oriented and reliable transport-layer protocol. We learned that TCP is complex, involving connection management, flow control, round-trip time estimation, as well as reliable data transfer. In fact, TCP is actually more complex than our description--we intentionally did not discuss a variety of TCP patches, fixes, and improvements that are widely implemented in various versions of TCP. All of this complexity, however, is hidden from the network application. If a client on one host wants to reliably send data to a server on another host, it simply opens a TCP socket to the server and then pumps data into that socket. The client/server application is blissfully unaware all of TCP's complexity.

In Section 3.6 we examined congestion control from a broad perspective, a
congestion control. We learned that congestion control is imperative for the
network can easily become gridlocked, with little or no data being transport
implements an end-to-end congestion-control mechanism that additively inc
connection's path is judged to be congestion-free, and multiplicatively decre
mechanism also strives to give each TCP connection passing through a co
also examined in some depth the impact of TCP connection establishment
important scenarios, connection establishment and slow start significantly c
more that while TCP congestion control has evolved over the years, it rema
continue to evolve in the upcoming years.

In Chapter 1 we said that a computer network can be partitioned into the "network edge" and the "network core." The network edge covers everything that happens in the end systems. Having now covered the

application layer and the transport layer, our discussion of the network edge is now complete. It is time to explore the network core! This journey begins in the next chapter, where we'll study the network layer, and continues into Chapter 5, where we'll study the link layer.