



Computer Networking: *A Top-Down Approach Featuring the Internet*
Order the **preliminary edition** for spring 2000!

Computer Networking

A Top-Down Approach Featuring the Internet

James F. Kurose and Keith W. Ross

Preface

[Link to the Addison-Wesley WWW site for this book](#)

[Link to overheads for this book](#)

Online Forum Discussion About This Book - with Voice!

1. **Computer Networks and the Internet**

1. [What is the Internet?](#)
2. [What is a Protocol?](#)
3. [The Network Edge](#)
4. [The Network Core](#)
 - [Interactive Programs for Tracing Routes in the Internet](#)
 - [Java Applet: Message Switching and Packet Switching](#)
5. [Access Networks and Physical Media](#)
6. [Delay and Loss in Packet-Switched Networks](#)
7. [Protocol Layers and Their Service Models](#)
8. [Internet Backbones, NAPs and ISPs](#)
9. [A Brief History of Computer Networking and the Internet](#)
10. [ATM](#)
11. [Summary](#)
12. [Homework Problems and Discussion Questions](#)

2. **Application Layer**

1. [Principles of Application-Layer Protocols](#)
2. [The World Wide Web: HTTP](#)
3. [File Transfer: FTP](#)
4. [Electronic Mail in the Internet](#)

3.1 Transport Layer Services and Principles

Residing between the application and network layers, the transport layer is in the core 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 towards 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, TCP and UDP transport layer protocols.

In the previous two chapters we have 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 for the worry of the details of the physical infrastructure used to carry these messages. Figure 3.1-1 illustrates the notion of logical communication.
- As shown in Figure 3.1-1, 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.
- At 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. At 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 multiplexing/demultiplexing service, a transport protocol can possibly provide other services to invoking applications, including reliable data transfer, bandwidth guarantees, and delay guarantees.

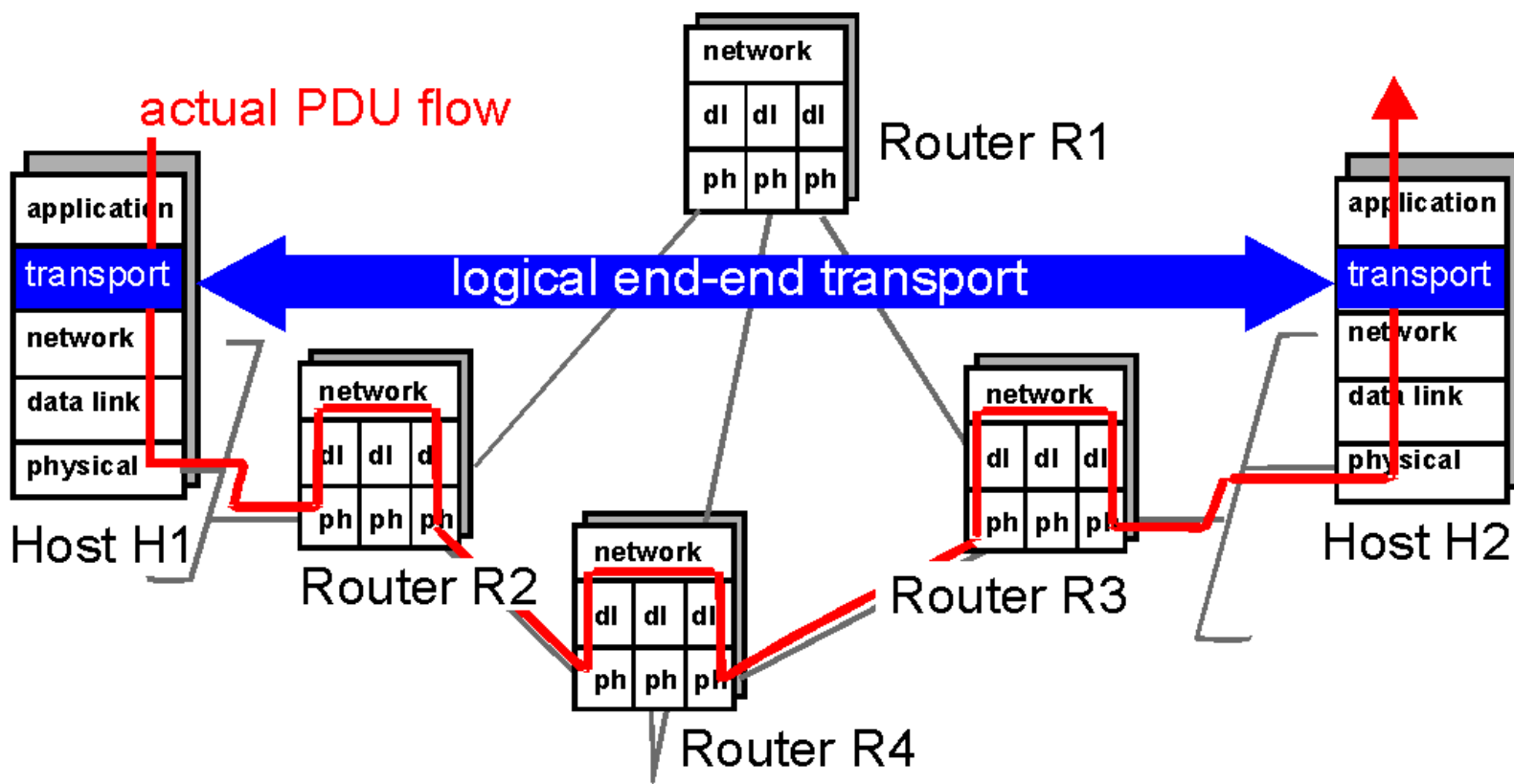


Figure 3.1-1: The transport layer provides logical rather than physical communication between applications.

3.1.1 Relationship between Transport and Network Layers

From the perspective of network applications, the transport layer *is* the underlying communication infrastructure. Of course, there is more to the communication infrastructure than just the transport layer. For example, the network layer lies just below the transport 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 with the kids in the West Coast households. The kids in the two households love to write 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 -- Alice in the West Coast house and Bob in the East Coast house -- responsible for mail collection and mail distribution. Each week Alice 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 to the West Coast house, Alice also has the job of distributing the mail to her brothers and sisters. Bob 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, Alice and Bob provide logical communication between the cousins -- Alice and Bob pick up mail from and deliver mail to, their brothers and sisters. Note that, from the cousins' perspective, Alice and Bob *are* the mail service, even though Alice and Bob 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 envelope
- *network layer protocol* = postal service (including mail persons)
- *transport layer protocol* = Alice and Bob

Continuing with this analogy, observe that Alice and Bob 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 (i.e., 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 3.1-1, 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 Alice and Bob 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 Alice and Bob. 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 (i.e., the same service model) as Alice and Bob. 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 Alice and Bob 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 (e.g., three days), then there is no way that Alice and Bob 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 can not provide delay or bandwidth guarantees for the 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

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 (e.g., 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 one, 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, which abbreviates "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 its header. These two minimal transport-layer services -- host-to-host 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 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's guarantee of reliable data transfer 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 traffic 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 alternative between the 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.

[Return to Table of Contents](#)

Copyright 1996-2000 Keith W. Ross and James F. Kurose

3.3 Connectionless Transport: UDP



The Internet makes two transport protocols available to its applications, UDP and TCP. In this section we take a close look at UDP: how it 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 must provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application.

UDP, defined in [\[RFC 768\]](#), does just about as little as a transport protocol can. 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 talking almost directly with IP. UDP takes messages from application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other fields of minor importance, 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 source and destination addresses to deliver the data in the segment 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 (see section 2.5) 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 a 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 UDP 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 shall discuss in Section 3.5, 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 principle 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 segment 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 becomes 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, etc.) 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, a significant fraction of the UDP-transmitted data could be lost due to router buffer overflow. Thus, the receive rate is limited by network congestion even if the sending rate is not constrained.

Table 3.1-1 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 -- 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, so that lost updates are replaced by more up-to-date updates. UDP is used to carry network management (SNMP - see Chapter 8) data. UDP is preferred to TCP in this case, since network management 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 3.1-1: Popular Internet applications and their underlying transport protocols.

As shown in Figure 3.1-1, 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 shall 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, interactive real-time applications, such as Internet phone and video conferencing, react very poorly to TCP's congestion control. For these reasons, developers of multimedia applications often choose to run the 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 lacks any form of congestion control. But congestion control is needed to prevent the network from entering a congested 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. Many researchers have proposed new mechanisms to force all sources, including UDP sources, to perform adaptive congestion control [\[Mahdavi\]](#).

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 (e.g., by adding acknowledgement and retransmission mechanisms, such as those we shall study in the next section). But this a non-trivial 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 being constrained by the transmission rate constraints imposed by TCP's congestion control mechanism. Application-level reliability also allows an application to tailor its own application-specific form of error control. An interactive real-time may occasionally choose to retransmit a lost message, provided that

round trip network delays are small enough to avoid adding significant playout delays [[Papadopoulos 1996](#)].

Many of today's proprietary streaming applications do just this -- they run over UDP, but they have built acknowledgements and retransmissions into the application in order reduce packet loss.

UDP Segment Structure

The UDP segment structure, shown in Figure 3.3-2, is defined in [[RFC 768](#)].

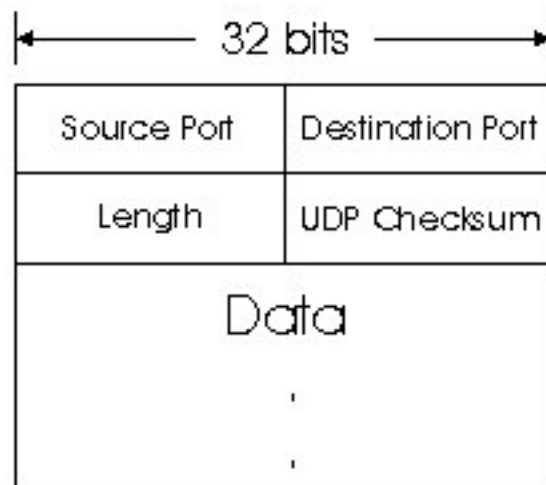


Figure 3.3-2: UDP segment structure

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 four 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 that host (i.e., perform the demultiplexing function). The checksum is used by the receiving host to check if errors have been introduced into the segment during the course of its transmission from source to destination. (Basic principles of error detection are described in Section 5.2.).

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. (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.) When the segment arrives (if it arrives!) at the receiving host, all 16-bit words are added together, including the checksum. If this sum equals 1111111111111111, then the segment has no detected errors. If one of the bits is a zero, then we know that errors have been introduced into the segment.

Here we give a simple example of the checksum calculation. You can find details about efficient implementation of the calculation in the [\[RFC 1071\]](#). 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 00111010100110101, which becomes the checksum. At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the segment, 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 segment. In section 5.1, we'll see that the Internet checksum is not foolproof -- even if the sum equals 1111111111111111, it is still possible that there are undetected errors in the segment. For this reason, a number of protocols use more sophisticated error detection techniques than simple checksumming.

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.

References

[**Papadopoulos 1996**] C. Papadopoulos and G. Parulkar, "Retransmission-Based Error Control for Continuous Media Applications," Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), April 1996.

[**Mahdavi**] J. Mahdavi and S. Floyd, "The TCP-Friendly Website,"

http://www.psc.edu/networking/tcp_friendly.html

[**RFC 768**] J. Postel, "User Datagram Protocol," [RFC 768](#), August 1980.

[**RFC 1071**] R. Braden, D. Borman, C. Partridge, "Computing The Internet Checksum," [RFC 1071](#), September 1988.

Search RFCs and Internet Drafts

If you are interested in an Internet Draft relating to a certain subject or protocol enter the keyword(s) here.

Query:

Press button to submit your query or reset the form:

Query Options:

Case insensitive

Maximum number of hits:

[Return to Table Of Contents](#)

Copyright Keith W. Ross and James F. Kurose 1996-2000

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.

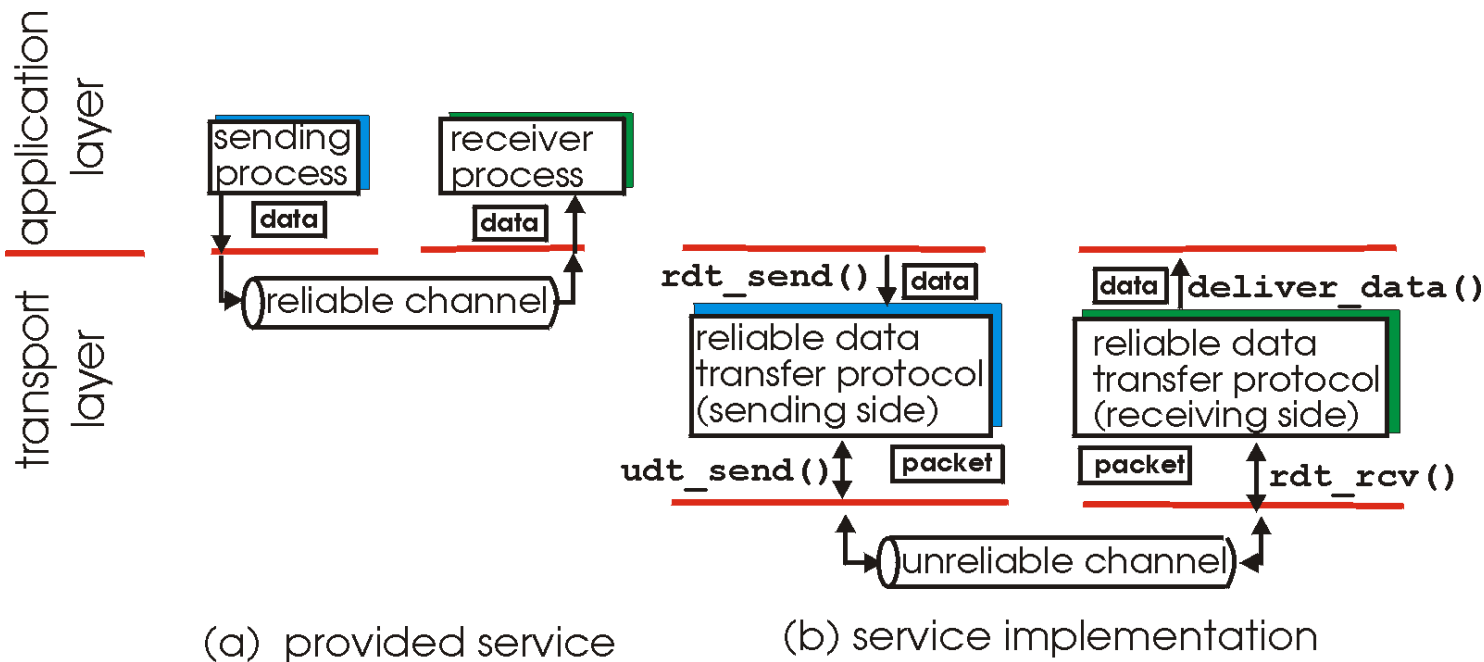


Figure 3.4-1: Reliable data transfer: service model and service implementation.

Figure 3.4-1 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.

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that 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 (e.g., as in the case of a link-level data transfer protocol) or a global internetwork (e.g., 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 3.4-1(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, i.e., data transfer from the sending to receiving side. The

case of reliable **bidirectional** (i.e., full duplex) data transfer is conceptually no more difficult but considerably more tedious. 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 3.4-1. 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()` (**un**reliable **d**ata transfer).

3.4.1 Building a 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 will call `rdt1.0`, is trivial. The **finite state machine** (FSM) definitions for the `rdt1.0` sender and receiver are shown in Figure 3.4-2. The sender and receiver FSMs in Figure 3.4-2 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 3.4-2 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.

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 (e.g., 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 (e.g., 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!

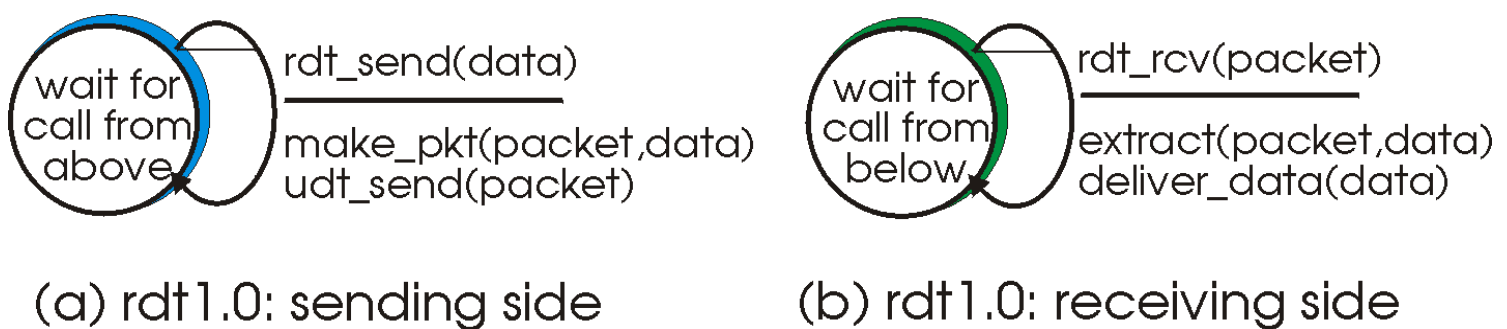


Figure 3.4-2: `rdt1.0` - a protocol for a completely reliable channel

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 acknowledgements** ("OK") and **negative acknowledgements** ("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, two 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 Sections 3.3 that the UDP transport protocol 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 acknowledgement (NAK) replies in the message dictation scenario are an example 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, e.g., a zero value could indicate a NAK and a value of 1 could indicate an ACK.

Figure 3.4-3 shows the FSM representation of `rdt2.0`, a data transfer protocol employing error detection, positive acknowledgements (ACKs), and negative acknowledgements (NAKs).

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 3.4-3 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 can *not* 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 3.4-3, the notation `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` corresponds to the event where a packet is received and is found to be in error.

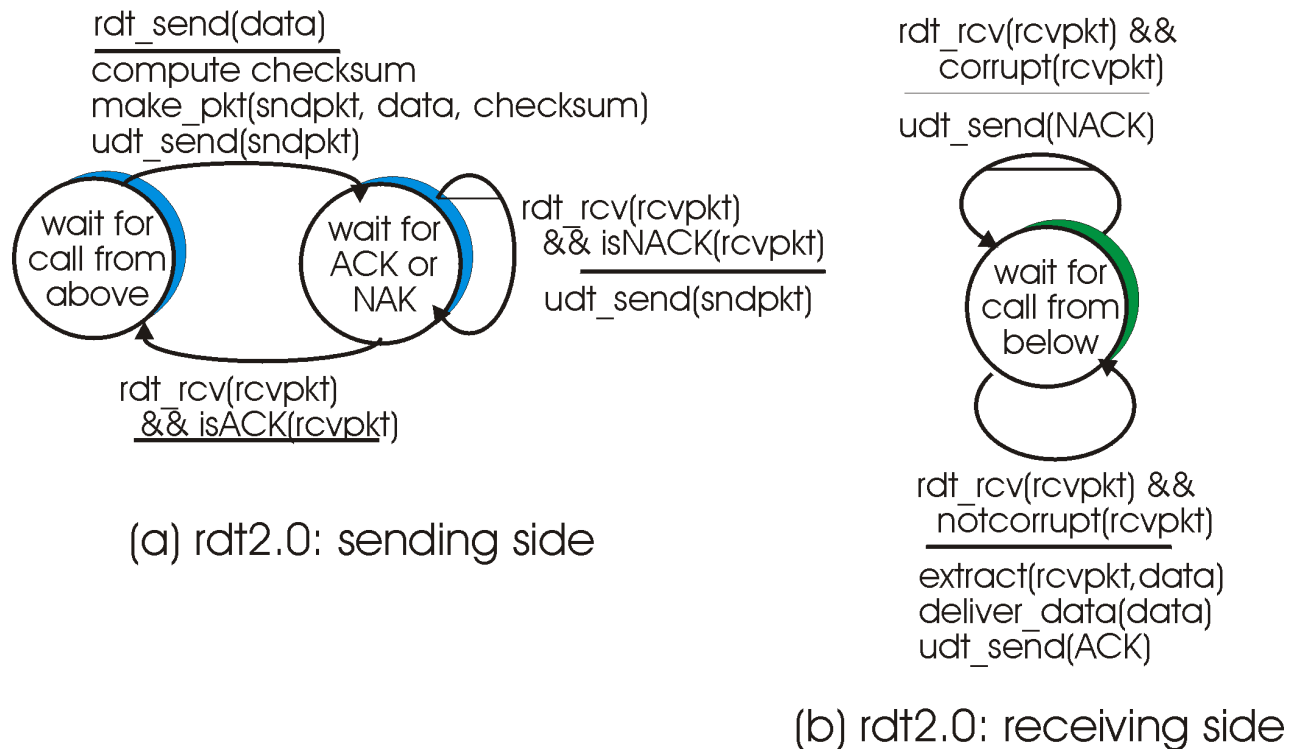


Figure 3.4-3: rdt2.0 - a protocol for a channel with bit-errors

Protocol rdt2.0 may look as if it works but unfortunately 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 recover from, bit errors. This solves the immediate problem for a channel which 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, 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 can not 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 1-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, i.e., moves "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 ACKing or NAKing, since 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.

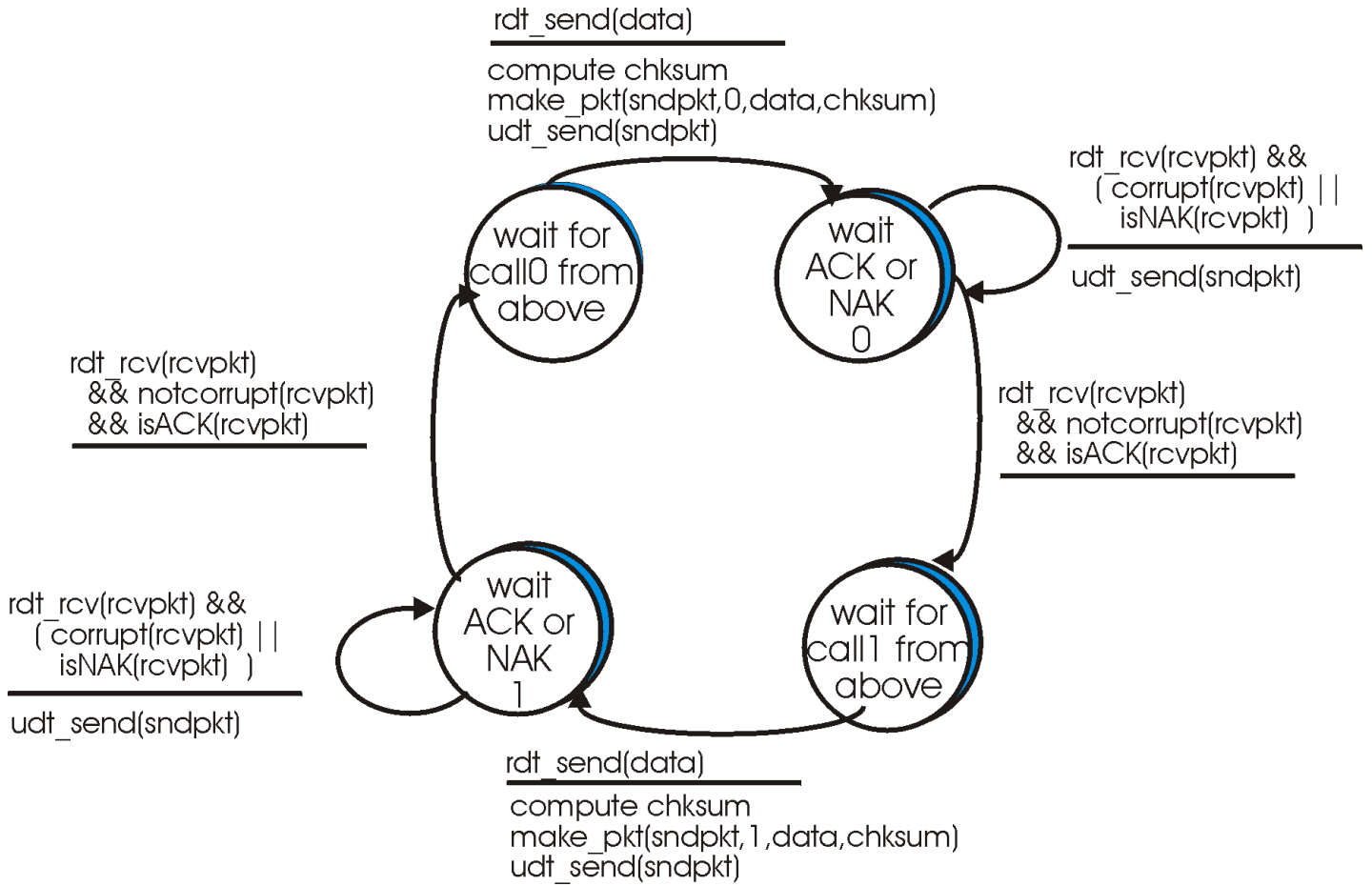


Figure 3.4-4: rdt2.1 sender

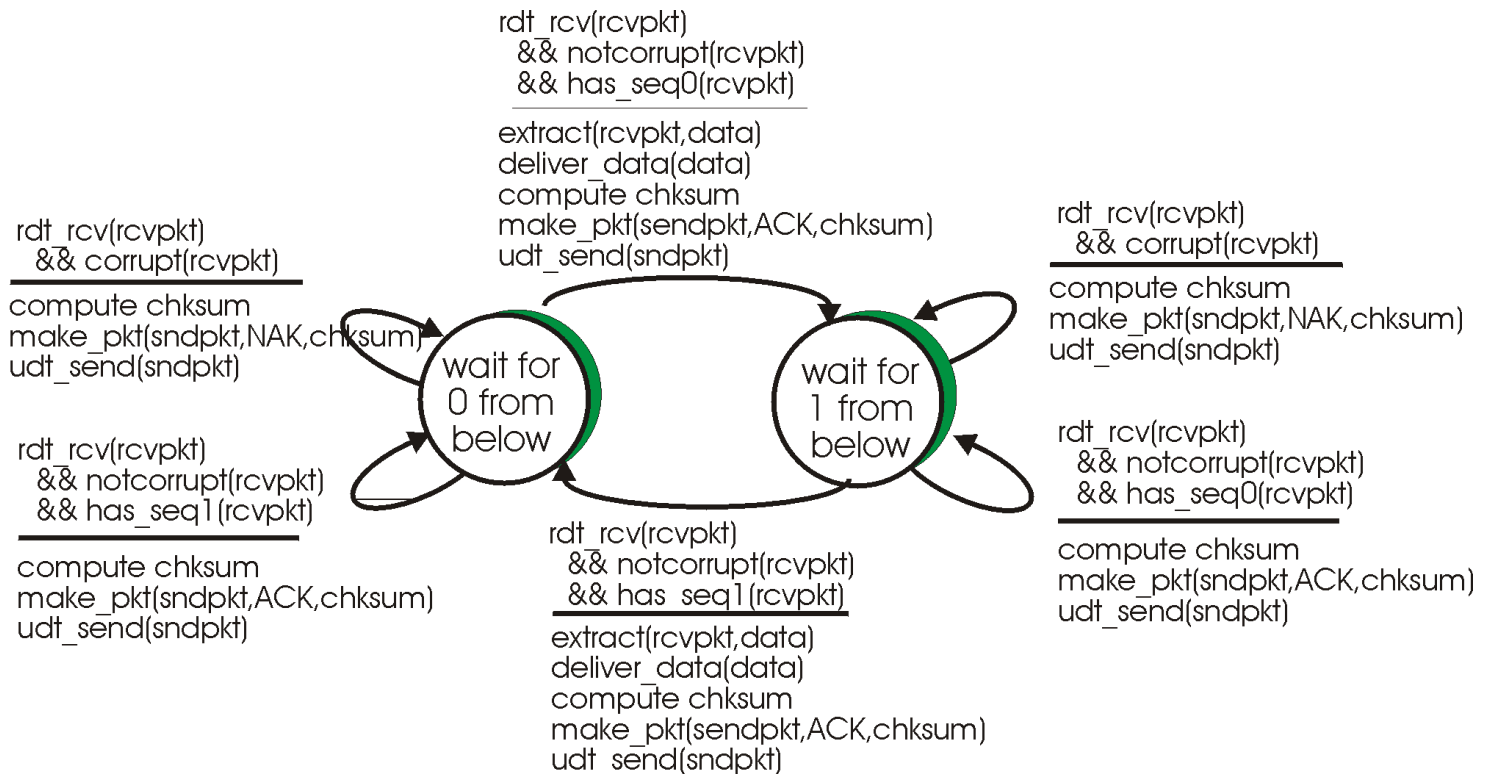


Figure 3.4-5: rdt2.1 receiver

Figures 3.4-4 and 3.4-5 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.

Protocol rdt2.1 uses both positive and negative acknowledgements from the receiver to the sender. A negative acknowledgement 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 (i.e., receives **duplicate ACKs**) knows that the receiver did not correctly receive the packet following the packet that is being ACK'ed twice. Many TCP implementations use the receipt of so-called "triple duplicate ACKs" (three ACK packets all ACK'ing the same 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 Figure 3.4-6 and 3.4-7.

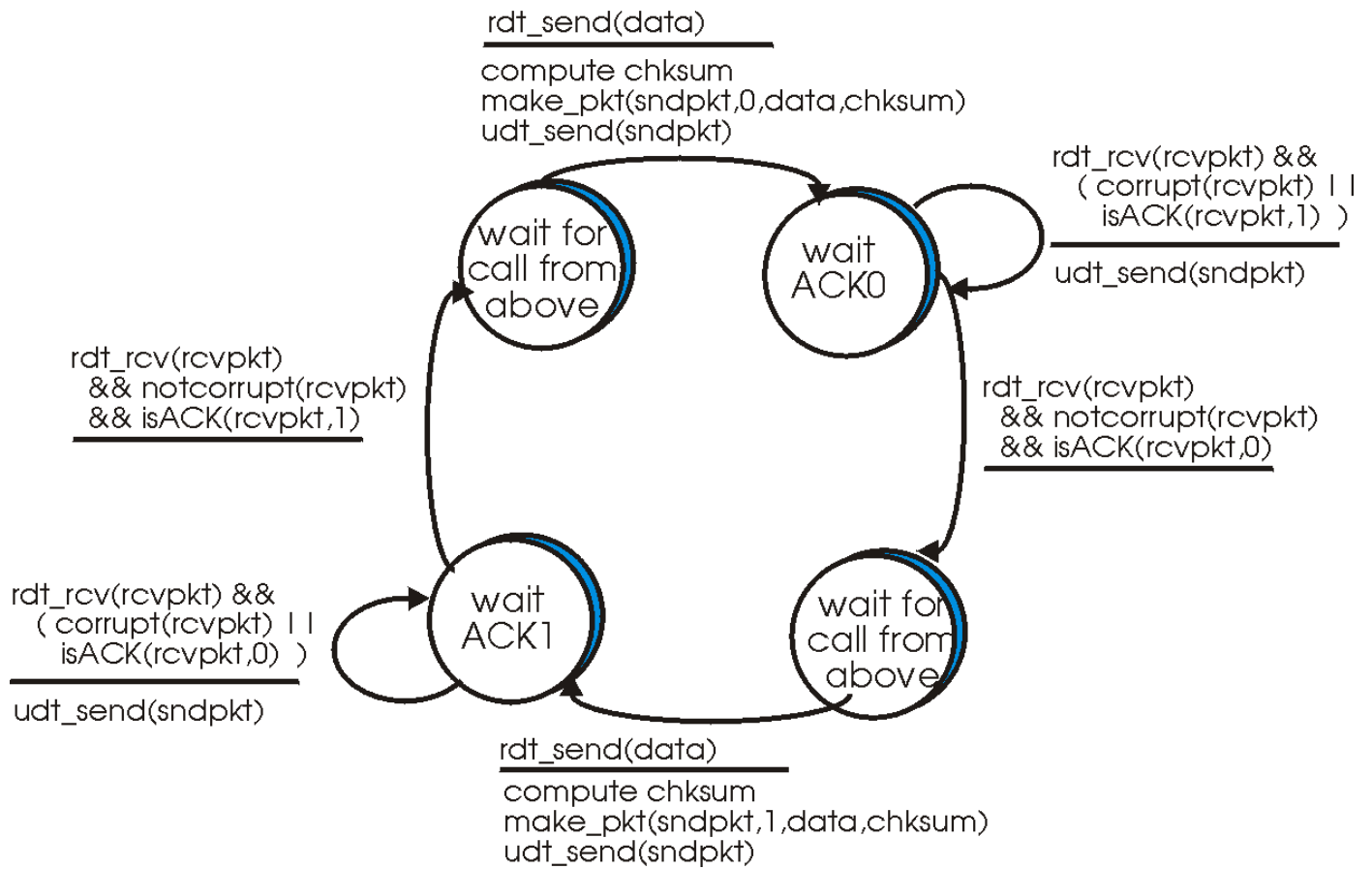


Figure 3.4-6: rdt2.2 sender

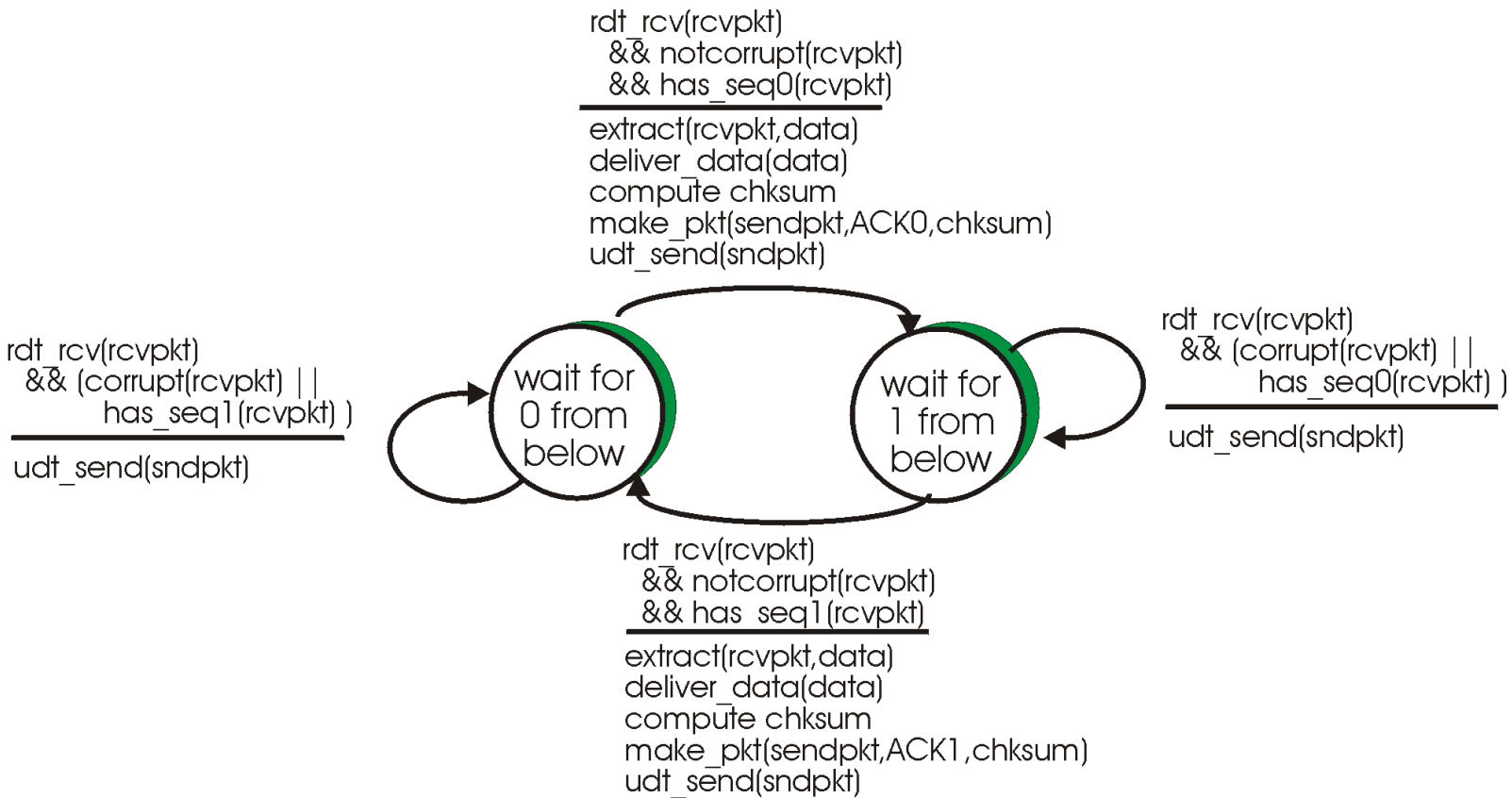


Figure 3.4-7: rdt2.2 receiver

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 this occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions - the techniques already developed in rdt 2.2 - will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism.

There are many possible approaches towards 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? It 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 (i.e., 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 (i) start the timer each time a packet (either a first time packet, or a retransmission) is sent, (ii) respond to a timer interrupt (taking appropriate actions), and (iii) 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 **acknowledgement field**. When the receiver generates an ACK, it will copy the sequence number of the data packet being ACK'ed into this acknowledgement field. By examining the contents of the acknowledgment field, the sender can determine the sequence number of the packet being positively acknowledged.

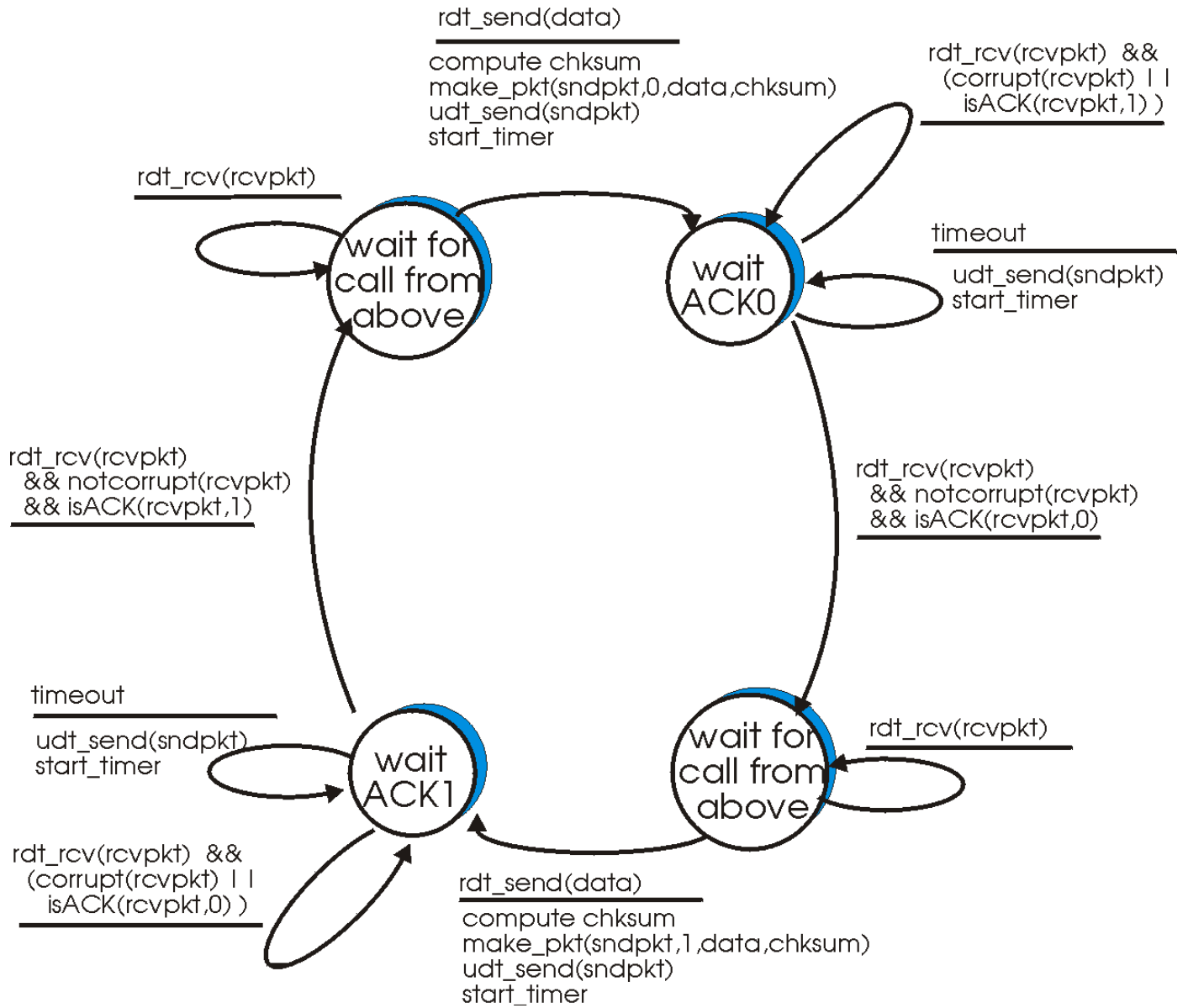


Figure 3.4-8: rdt 3.0 sender FSM

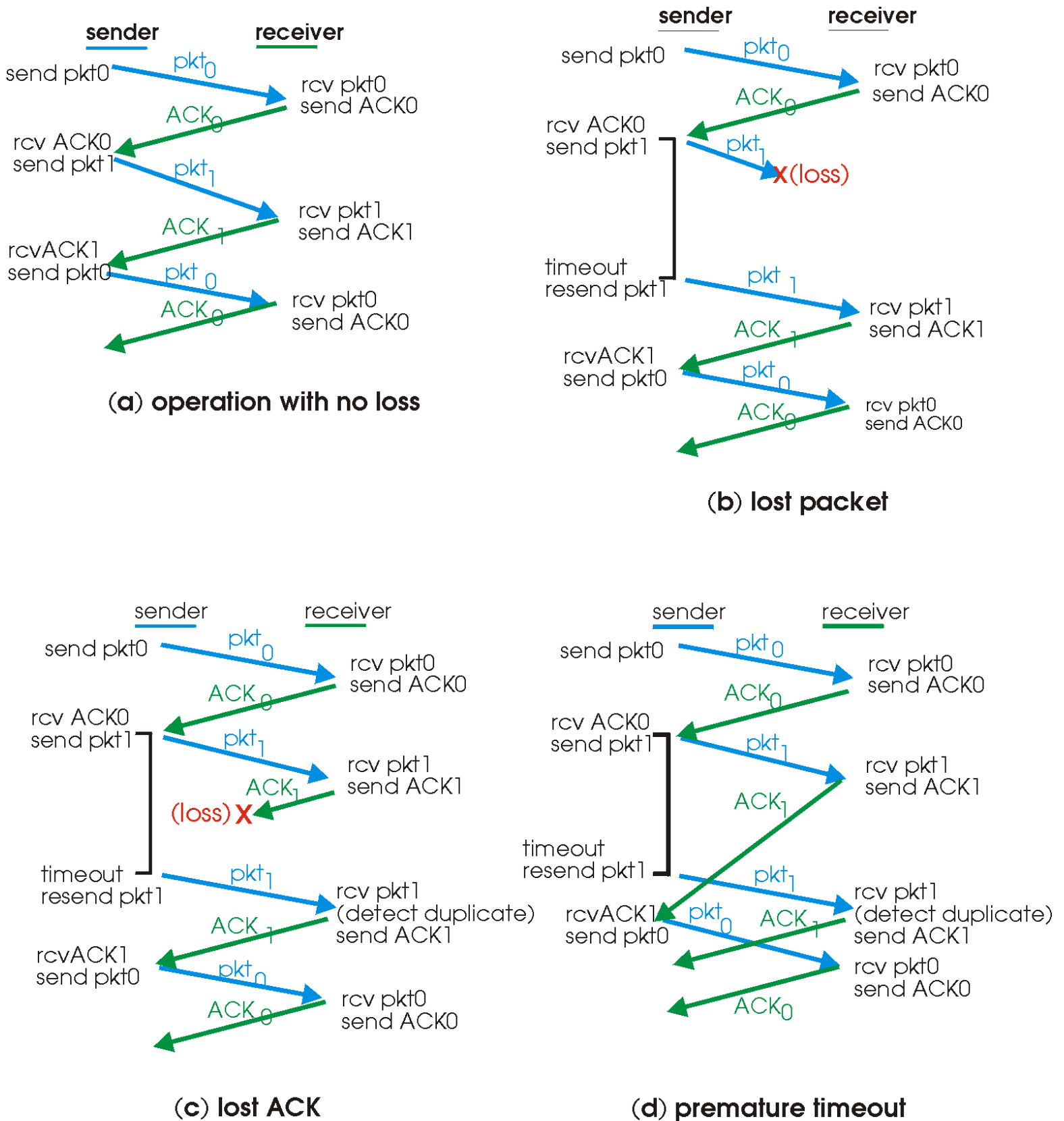


Figure 3.4-9: Operation of rdt 3.0, the alternating bit protocol

Figure 3.4-8 shows the sender FSM for rdt 3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets. Figure 3.4-9 shows how the protocol operates with no lost or delayed packets, and how it handles lost data packets. In

Figure 3.4-9, time moves forward from the top of the diagram towards 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 3.4-9(b)-(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 acknowledgement 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 1K bytes per packet including both header fields and data, the time needed to actually transmit the packet into the 1Gbps link is

$$T_{trans} = SP/C = (8 \text{ Kbits/packet}) / (10^9 \text{ bits/sec}) = 8 \text{ microseconds}$$

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 3.4-10a, with the last bit of the packet emerging at the receiver at $t = 15.008$ msec. 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 only busy (sending or receiving) for .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 into the channel, we have a rather dismal sender utilization, U_{sender} of

$$U_{sender} = (.008 / 30.016) = 0.00015$$

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 1K bytes in 30.016 milliseconds, an effective throughput of only 33KB/sec - even though a 1Gigabit 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 33KB! 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 queueing 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.

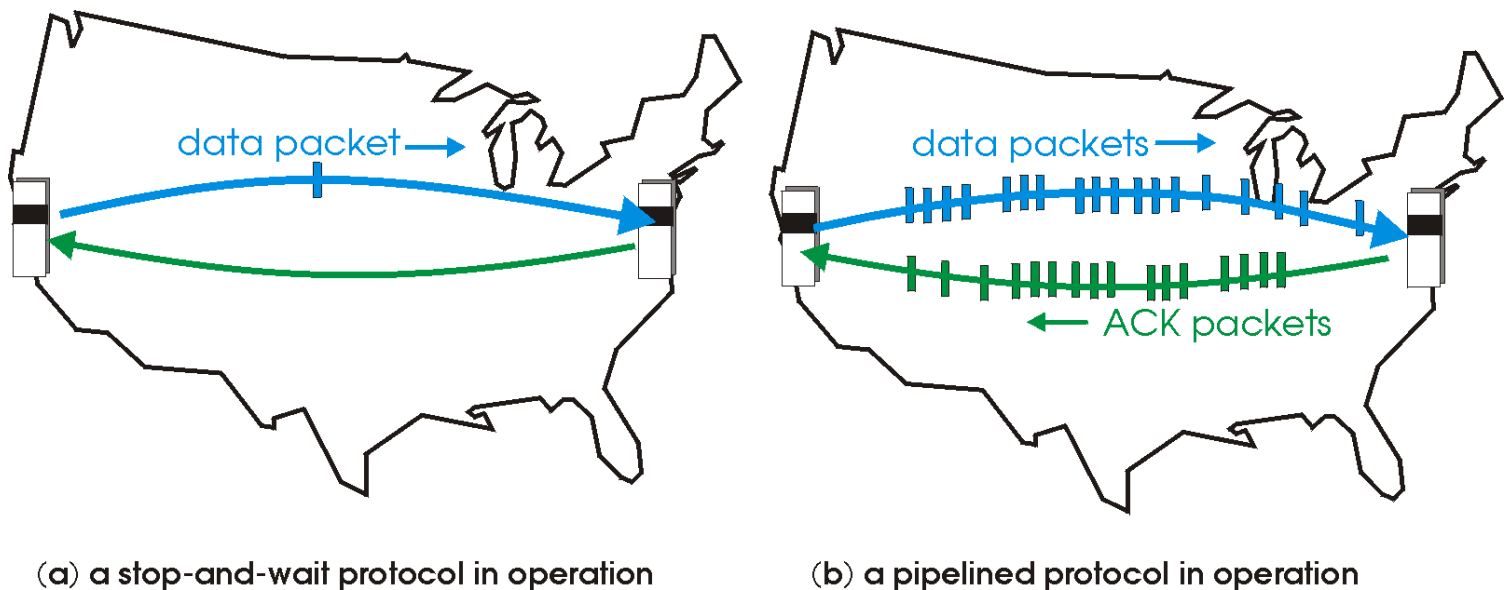


Figure 3.4-10: Stop-and-wait versus pipelined protocols

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 acknowledgements, as shown in Figure 3.4-10(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 towards pipelined error recovery can be identified: **Go-Back-N** and **selective repeat**.

3.4.3 Go-Back-N (GBN)

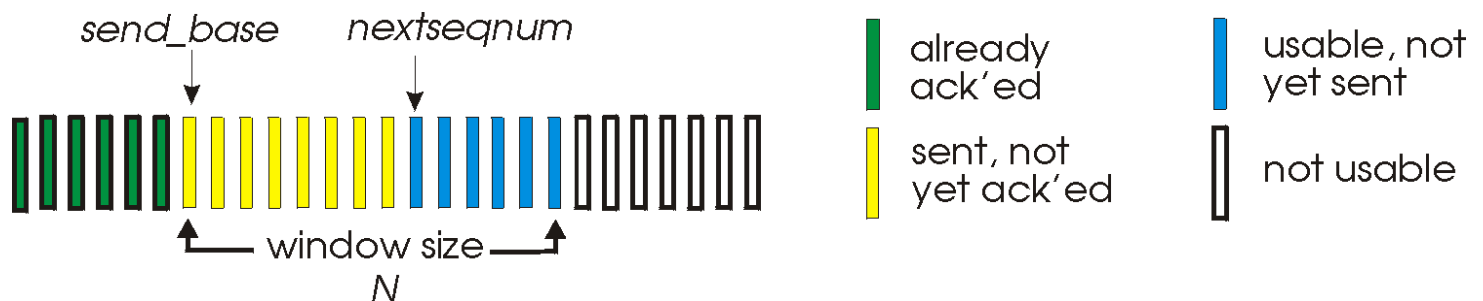


Figure 3.4-11: Sender's view of sequence numbers in Go-Back-N

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 3.4-11 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 (i.e., 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 $base+N$ can not be used until an unacknowledged packet currently in the pipeline has been acknowledged.

As suggested by Figure 3.4-11, 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 even limit the number of outstanding, unacknowledged packet 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 rtd3.0 had a 1-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.

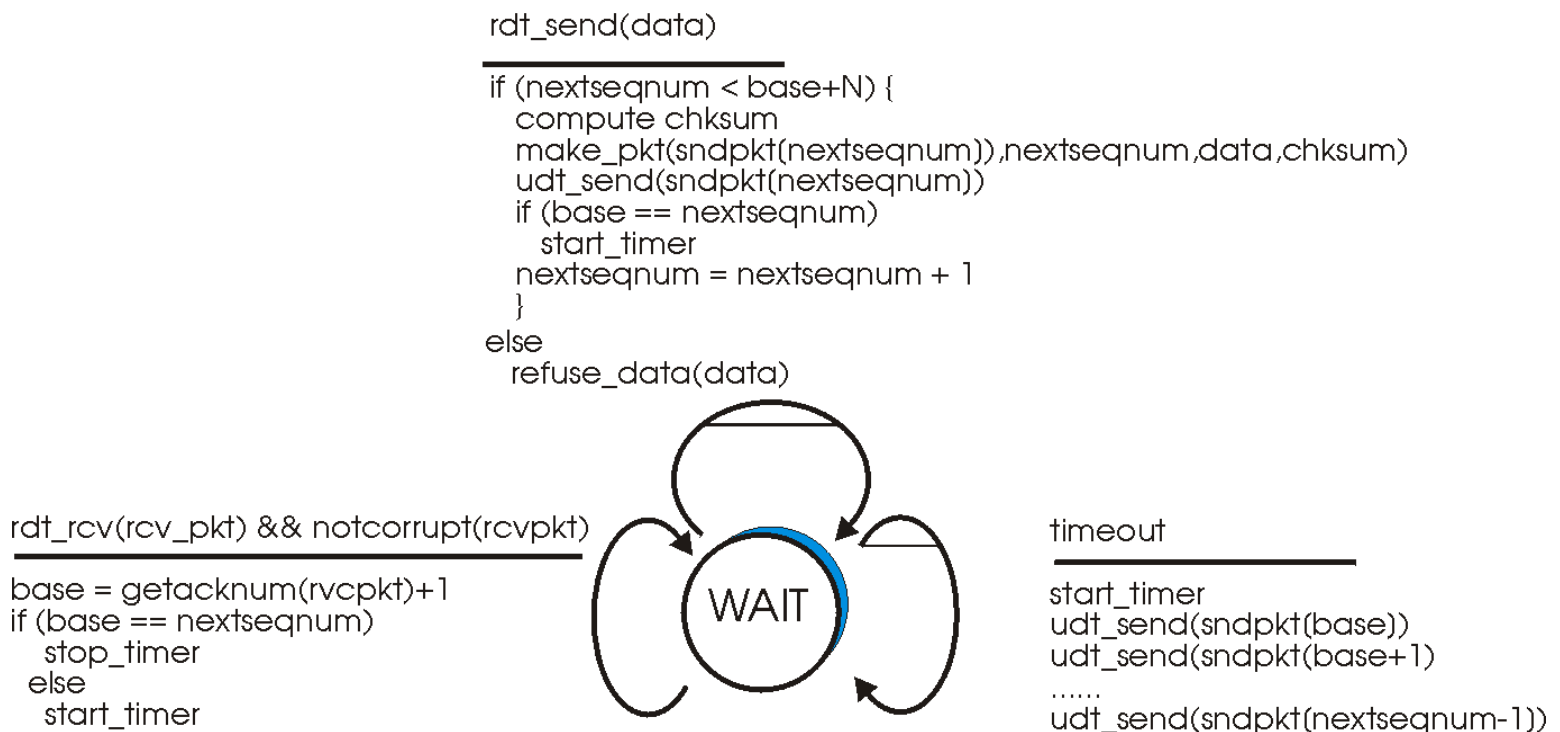


Figure 3.4-12 Extended FSM description of GBN sender.

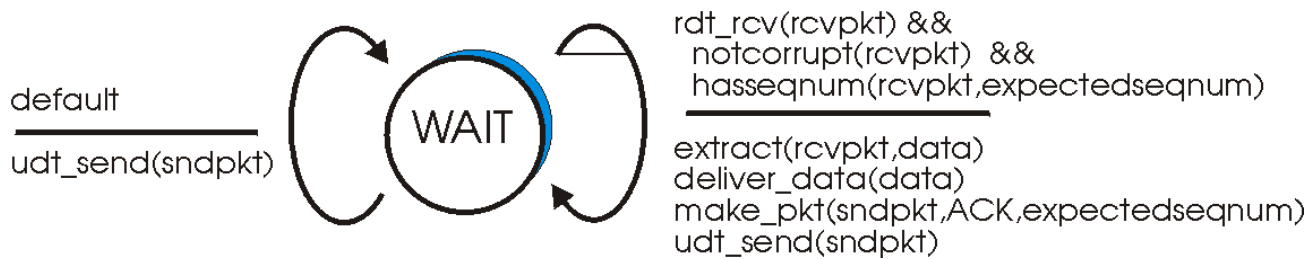


Figure 3.4-13 Extended FSM description of GBN receiver.

Figures 3.4-12 and 3.4-13 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 84] provides an excellent survey of additional extensions to FSM techniques as well as other programming language-based techniques for specifying protocols.

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, i.e., 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 (e.g., 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 acknowledgement for packet with sequence number n will be taken to be a **cumulative acknowledgement**, 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 acknowledgement 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 3.4-12 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 (i.e., 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 acknowledgements is a natural choice for GBN.

In our GBN protocol, the receiver discards out-of-order packets. While 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 3.4-13. 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.

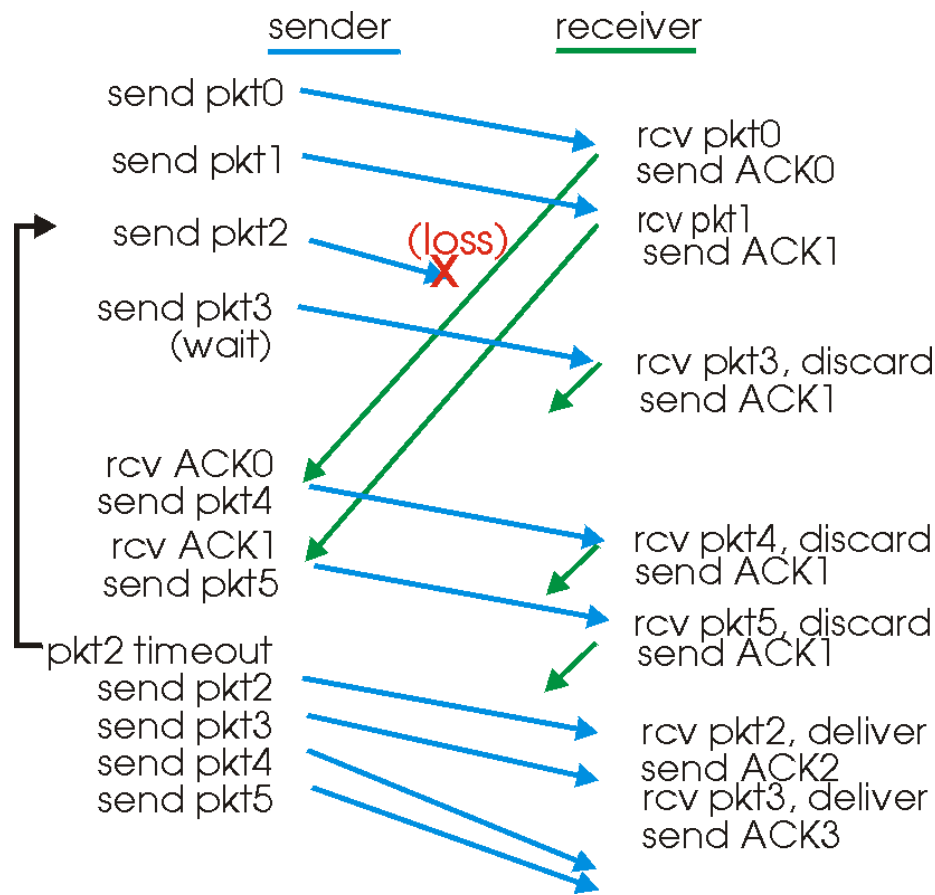


Figure 3.4-14: Go-Back-N in operation

Figure 3.4-14 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 (e.g., ACK0 and ACK1) is received, the window slides forwards 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.

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 3.4-12. 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 (i) a call from the upper layer entity to invoke `rdt_send()`, (ii) a timer interrupt, and (iii) 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: the use of sequence numbers, cumulative acknowledgements, checksums, and a timeout/retransmit operation. Indeed, TCP is often referred to as a GBN style of protocol. There are, however, some differences. 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 2018], 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.

3.4.4 Selective Repeat (SR)

The GBN protocol allows the sender to potentially "fill the pipeline" in Figure 3.4-10 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 1000 words (e.g., a window size of 1000 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 (i.e., 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 3.4-15 shows the SR sender's view of the sequence number space. Figure 3.4-16 details the various actions taken by the SR sender.

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 (i.e., packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in-order to the upper layer. Figure 3.4-17 itemizes the various actions taken by the SR receiver. Figure 3.4-18 shows an example of SR operation in the presence of lost packets. Note that in Figure 3.4-18, 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.

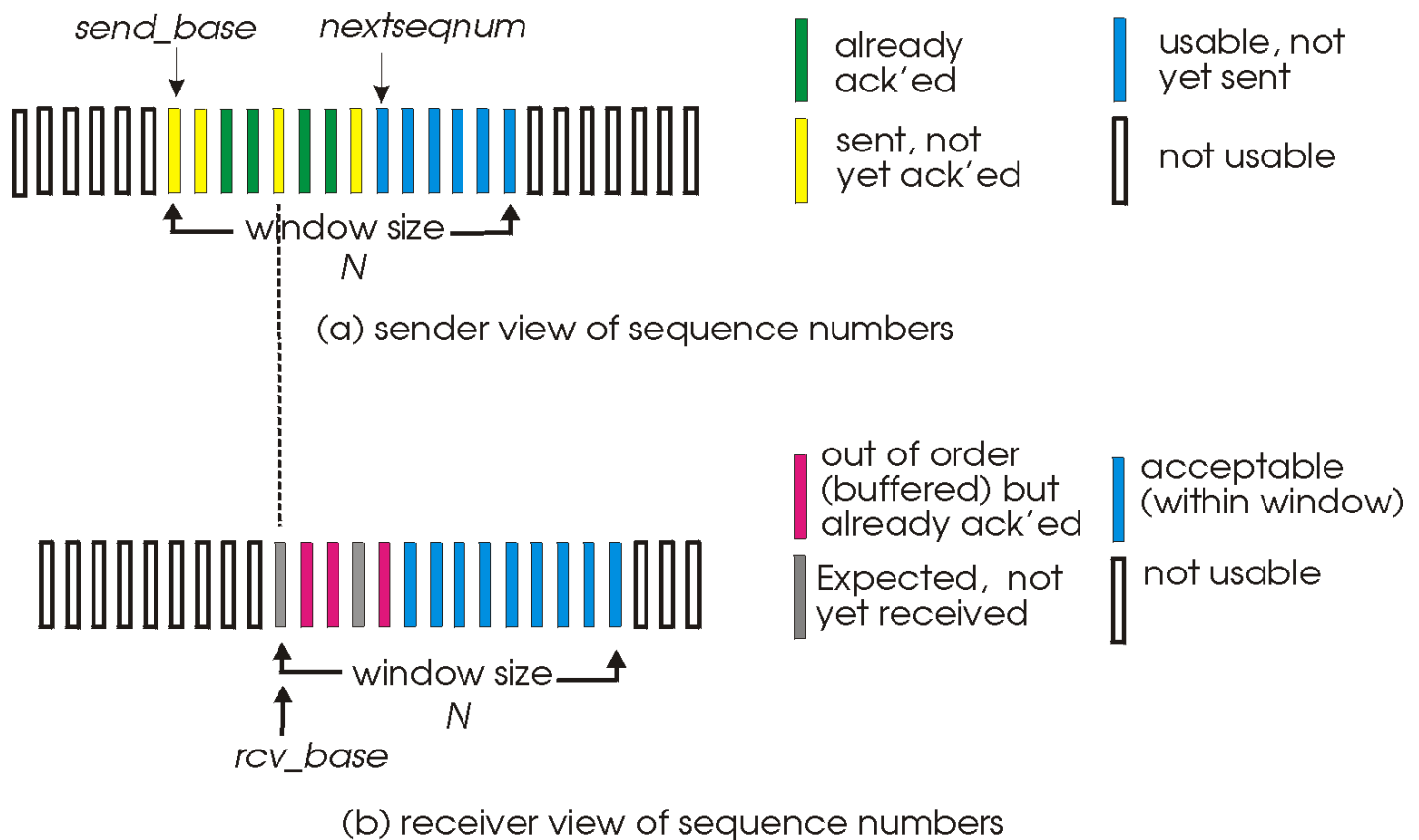


Figure 3.4-15: SR sender and receiver views of sequence number space

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.
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 `sendbase`, 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 3.4-16: Selective Repeat sender actions

1. **Packet with sequence number in $[rcvbase, rcvbase+N-1]$ 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 (`rcvbase` in Figure 3.4-15), then this packet, and any previously buffered and consecutively numbered (beginning with `rcvbase`) 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 3.4-18. When a packet with a sequence number of `rcvbase=2` is received, it and packets `rcvbase+1` and `rcvbase+2` can be delivered to the upper layer.
2. **Packet with sequence number in $[rcvbase-N, rcvbase-1]$ 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 3.4-17: Selective Repeat Receiver Actions

It is important to note that in step 2 in Figure 3.4-17, 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-acknowledgement is indeed needed. Given the sender and receiver sequence number spaces in Figure 3.4-15 for example, if there is no ACK for packet `sendbase` propagating from the receiver to the sender, the sender will eventually retransmit packet `sendbase`, even though it is clear (to us, not the sender!) that the receiver has already received that packet. If the receiver were not to ACK 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.

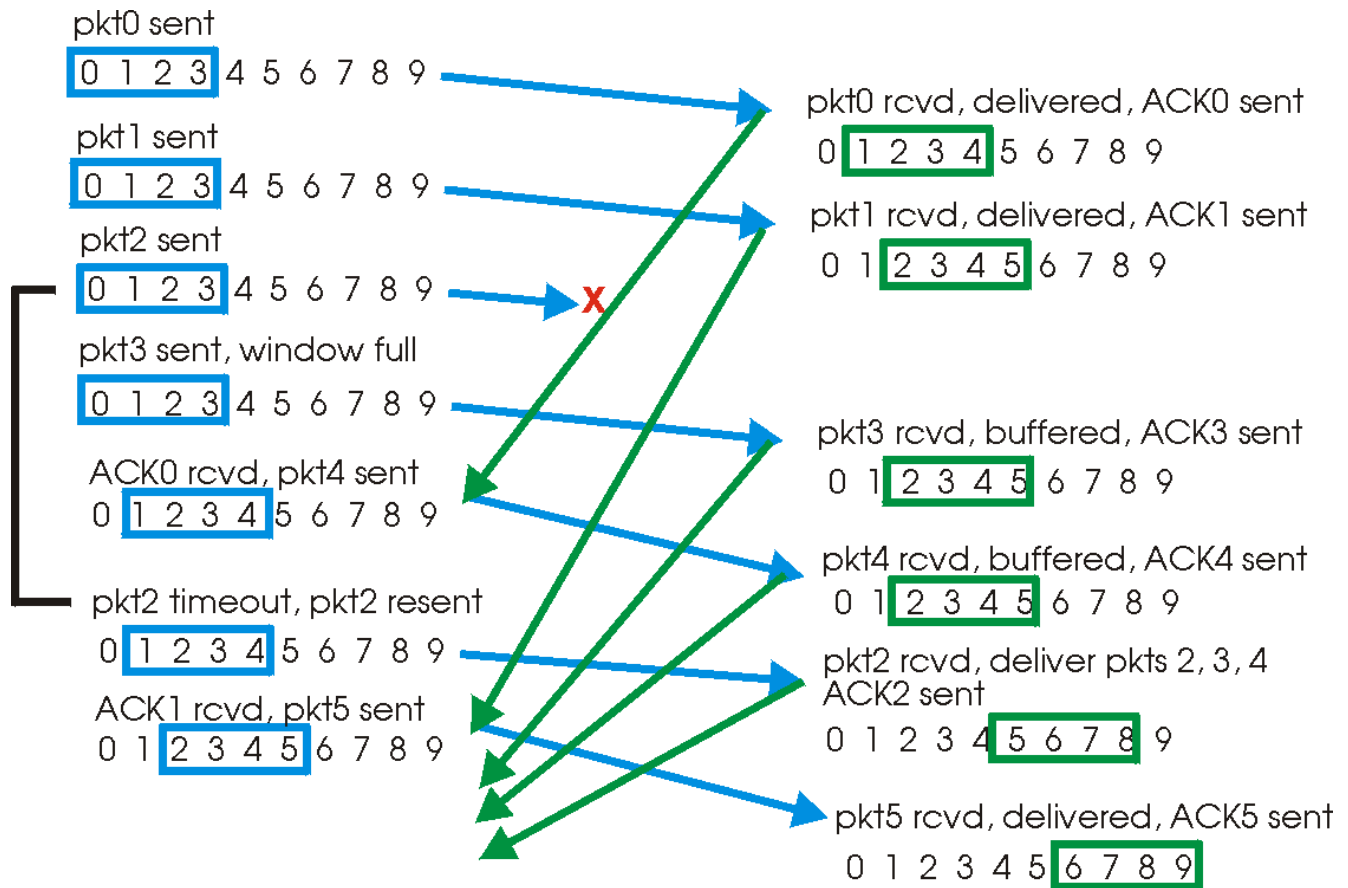


Figure 3.4-18: SR Operation

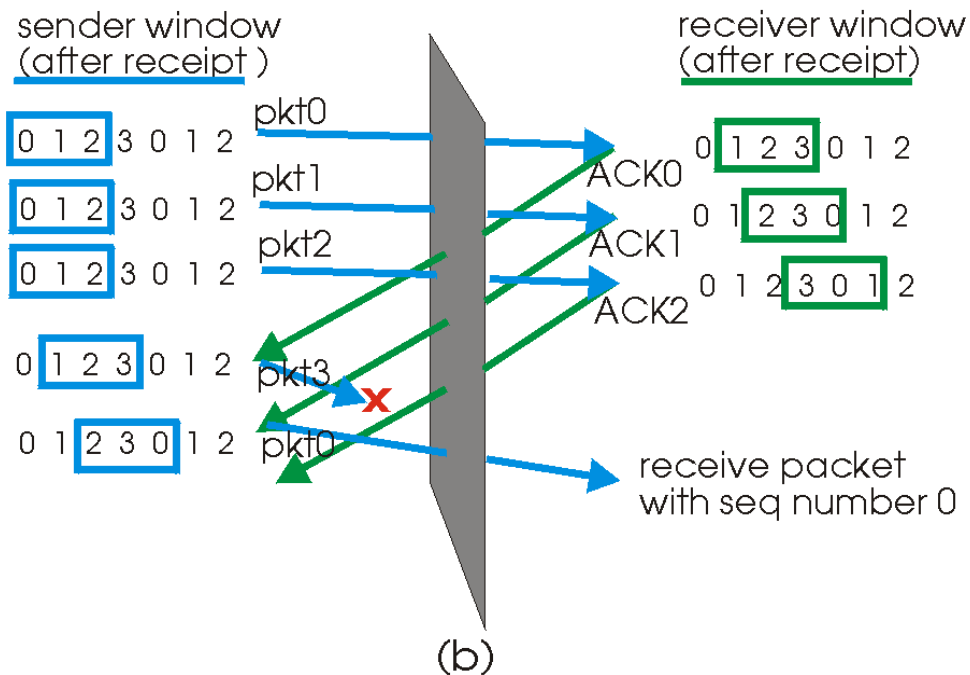
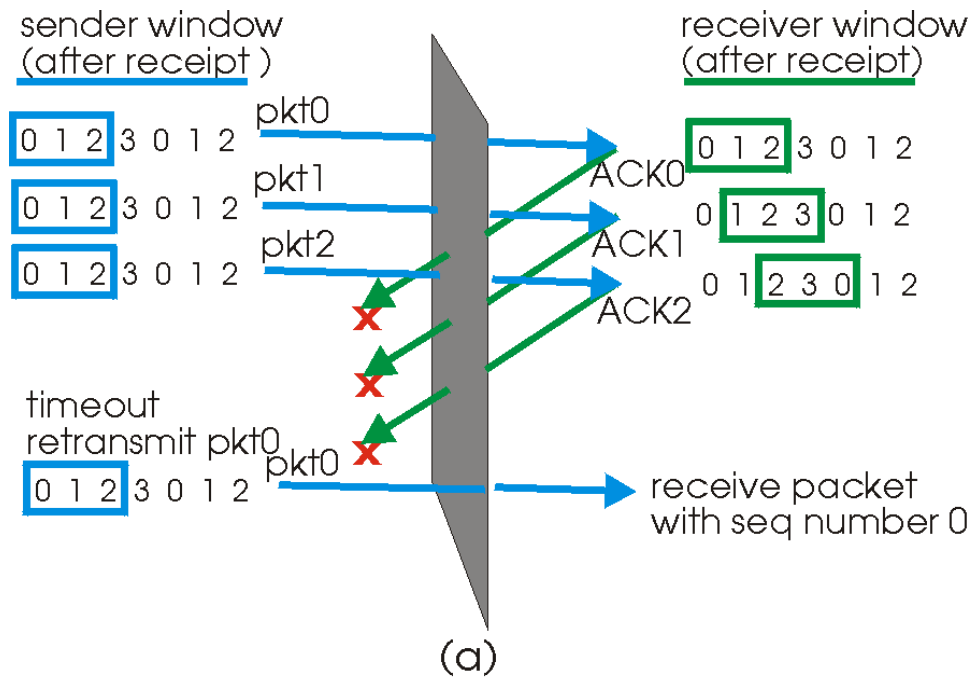


Figure 3.4-19: SR receiver dilemma with too large windows: a new packet or a retransmission?

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 3.4-19(a), 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.

In the second scenario, shown in Figure 3.4-19(b), 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 3.4-19, which has a figurative curtain between the sender and the receiver, since the receiver can not "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 3.4-19 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 one smaller 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.

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 can not be re-ordered 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 acknowledgement 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 insure 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 can not "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 [Sunshine 1978] describes a method for using sequence numbers such that reordering problems can be completely avoided.

References

- [Bochman 84] G.V. Bochmann and C.A. Sunshine, "Formal methods in communication protocol design", *IEEE Transactions on Communicaitons*, Vol. COM-28, No. 4, (April 1980), pp 624-631.
- [RFC 1323] V. Jacobson, S. Braden, D. Borman, "TCP Extensions for High Performance," [RFC 1323](#), May 1992.
- [RFC 2018] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, October 1996
- [Stevens 1994] W.R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, 1994.
- [Sunshine 1978] C. Sunshine and Y.K. Dalal, "Connection Management in Transport Protocols", *Computer Networks*, Amsterdam, The Netherlands: North-Holland", 1978.

TCP Flow Control

NOTES :

1. Host B consumes data in 2Kbyte chunks at random times.
2. When Host A receives an acknowledgment with $WIN=0$, Host A sends a packet with one byte of data. It is assumed for simplicity, that this one byte is not consumed by the receiver.

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 acknowledgements, timers and header fields for sequence and acknowledgement 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 (but not recovery) 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. That is, application-level data can be transferred in both directions between two hosts - 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. TCP connection is also always **point-to-point**, i.e., 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 us 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 host that is initiating the connection is called the **client host**, while the other host is called the **server host**. 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 TCP 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," i.e., 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 the Figure 3.5-1, 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. 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.)

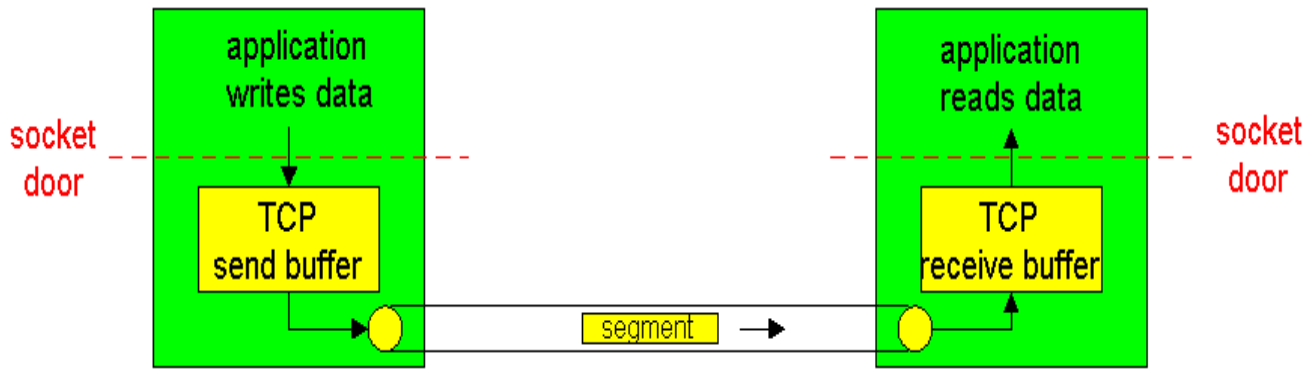


Figure 3.5-1: TCP send and receive buffers

TCP encapsulates each chunk of client data with 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 3.5-1.

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.

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 3.3-2 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**. 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 the purposes of 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], [RFC1323] 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 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.)

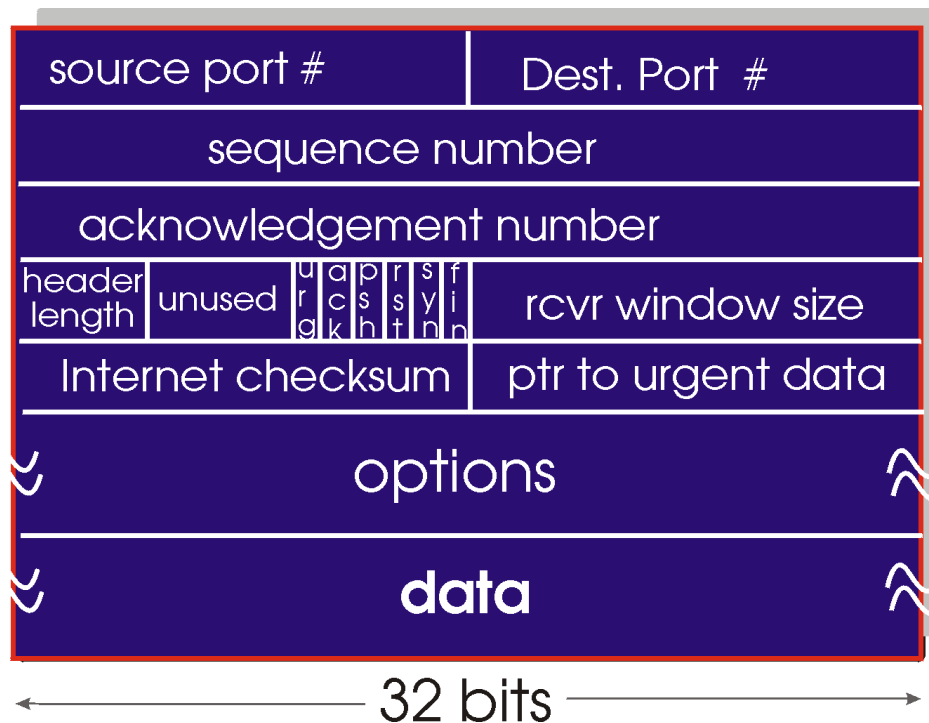


Figure 3.5-2: TCP segment structure

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 3.5-3, 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.

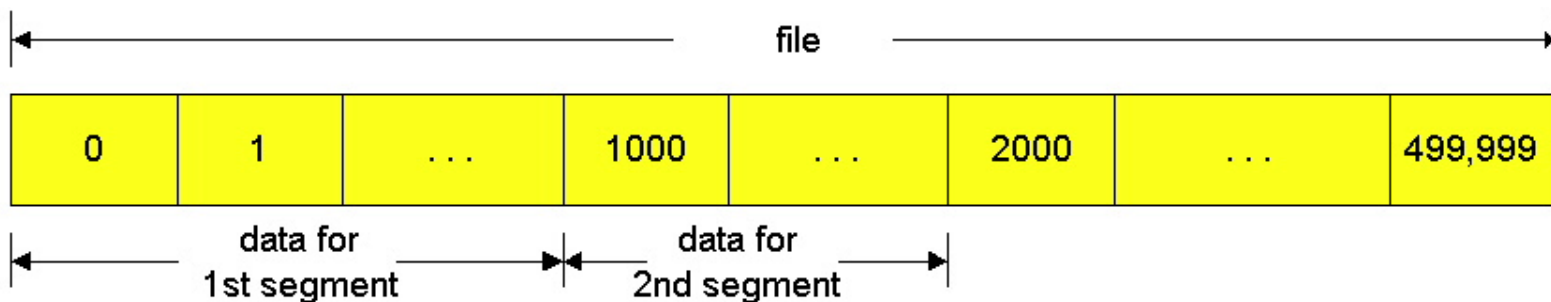


Figure 3.5-3: Dividing file data into TCP segments.

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 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 acknowledgements**.

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 (i) the receiver immediately discards out-of-order bytes; or (ii) 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 significantly 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 3.5.3 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 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 78].

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 one host, 88.88.88.88, initiates a Telnet session with host 99.99.99.99. (Anticipating our discussion on IP addressing in the next chapter, we take the liberty to use IP addresses to identify the hosts.) Because host 88.88.88.88 initiates the session, it is labeled the client and host 99.99.99.99 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 when the user hits the key and when 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 3.5-4, 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 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.

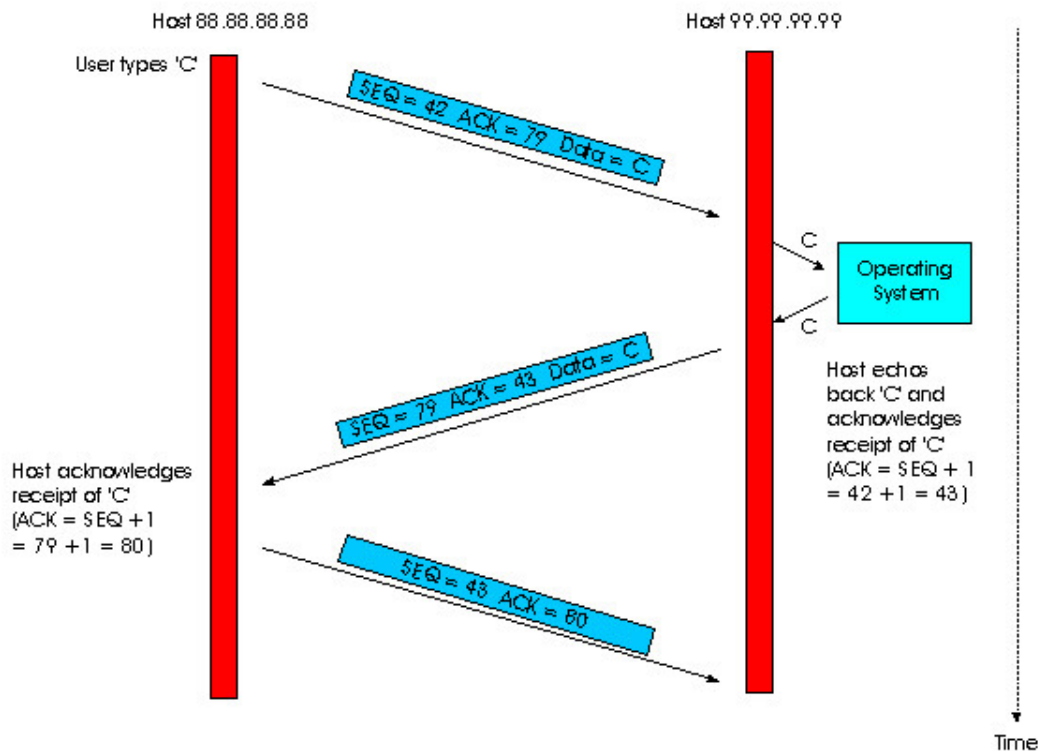


Figure 3.5-4: Sequence and acknowledgment numbers for a simple Telnet application over TCP

As shown in Figure 3.5-4, 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 client 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 acknowledgement for client-to-server data is carried in a segment carrying server-to-client data; this acknowledgement 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 (i.e., 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. Many popular application protocols -- including FTP, SMTP, NNTP, HTTP and Telnet -- use TCP rather than UDP primarily because TCP provides reliable data transfer 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, i.e., 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 informal overview of how TCP provides reliable data transfer. We shall see that the reliable data transfer service of TCP uses many of the principles that we studied in Section 3.4.

Retransmissions

Retransmission of lost and corrupted data is crucial for providing reliable data transfer. TCP provides reliable data transfer by using positive acknowledgments and timers in much the same way as we studied in section 3.4. TCP acknowledges data that has been received correctly, and retransmits segments when segments or their corresponding acknowledgements are thought to be lost or corrupted. Just as in the case of our reliable data transfer protocol, `rdt3.0`, TCP can not itself tell for certain if a segment, or its ACK, is lost, corrupted, or overly delayed. In all cases, 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 in the previous section that pipelining can greatly improve the throughput of a TCP connection 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 sender can have multiple transmitted, but unacknowledged, segments at any given time.

```

/* 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 3.4-11 */
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)

        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

        event: ACK received, with ACK field value of y
            if (y > sendbase) { /* cumulative ACK of all data 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
                }
            }
    } /* end of loop forever */

```

Figure 3.5-5: simplified TCP sender

Figure 3.5-5 shows the three major events related to data transmission/retransmission at a simplified TCP sender. Let us 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.

The third major event that must be handled by the TCP sender is the arrival of an acknowledgement 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 that the sender has yet to receive an acknowledgement for, or a so-called **duplicate ACK** that re-acknowledges a segment for which the sender has already received an earlier acknowledgement. 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 3.5-1 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 - i.e., a missing segment. Since TCP does not use negative acknowledgements, the receiver can not send an explicit negative acknowledgement back to the sender. Instead, it simply re-acknowledges (i.e., 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.

Event	TCP receiver action
Arrival of in-order segment with expected sequence number. All data up to 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 arrives 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.

Table 3.5-1: TCP ACK generation recommendations [RFC 1122, RFC 2581]

A Few Interesting Scenarios

We end this discussion by looking at a few simple scenarios. Figure 3.5-6 depicts the scenario where 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 that the bytes in the segment duplicate bytes it has already deposited in its receive buffer. Thus TCP in host B will discard the bytes in the retransmitted segment.

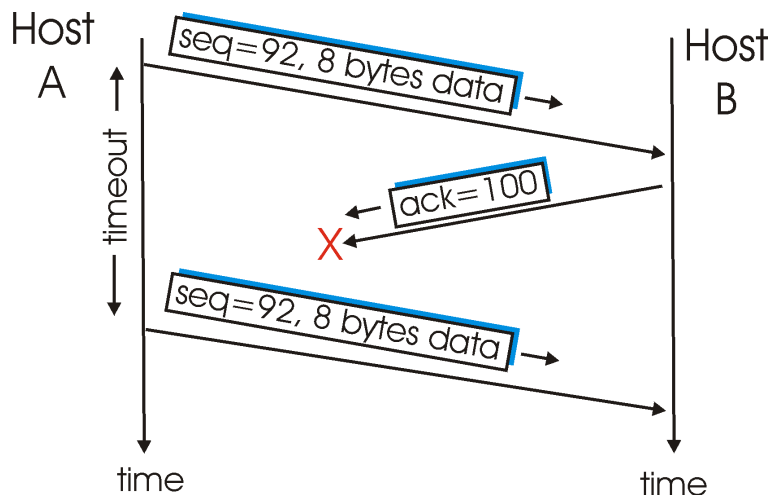


Figure 3.5-6: Retransmission due to a lost acknowledgment

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 acknowledgements for each of these segments. The first of these acknowledgements has acknowledgment number 100; the second has acknowledgment number 120. Suppose now that neither of the acknowledgements 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 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 3.5-7, if the second acknowledgment does not get lost and arrives before the timeout of the second segment, A does not resend the second segment.

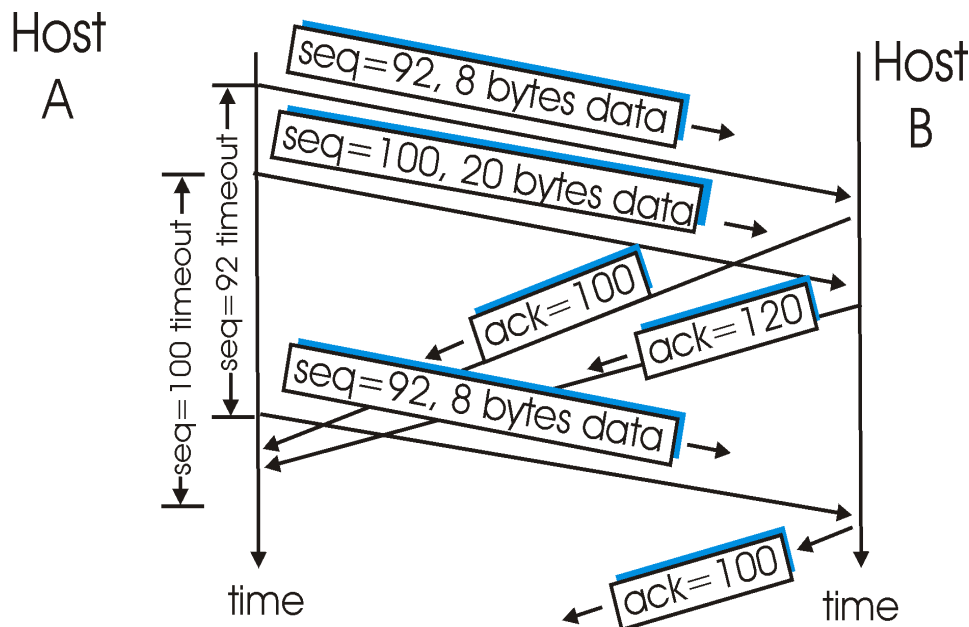


Figure 3.5-7: Segment is not retransmitted because its acknowledgment arrives before the timeout.

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 3.5-8.

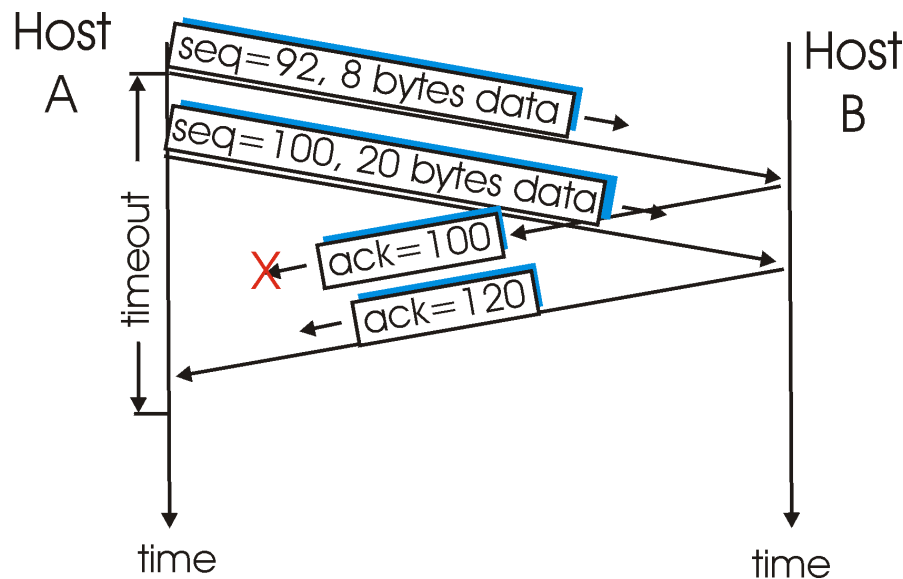


Figure 3.5-8: A cumulative acknowledgment avoids retransmission of first segment

Recall that in the previous section we said that TCP is a Go-Back-N style protocol. This is because acknowledgements are cumulative and correctly-received but out-of-order segments are not individually ACKed by the receiver. Consequently, as shown in Figure 3.5-5 (see also Figure 3.4-11), 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 $1, 2, \dots, N$, 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 the subsequent packets $n+1, n+2, \dots, N$. TCP, on the other hand, would retransmit at most one segment, namely, segment n . Moreover, TCP would not even retransmit segment n if the acknowledgement 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 by eliminating 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. While 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, i.e., 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.