

CS232 Final - Transport layer



lets discuss transport layer from kurose, I will give topics and book text, you need to prepare study materials

Perfect — that's a great way to study Transport Layer from **Kurose & Ross**

Here's how we'll do it:

What you'll send:

For each round, you can send:

- **Topic name(s)** (e.g., UDP, TCP congestion control, Flow control, etc.)
- **Relevant book text / screenshots / pasted paragraphs**

What I'll prepare for you:

For each topic, I'll generate:

- **Clean, exam-ready study notes**
- **Core intuition (in simple words)**
- **Key formulas (if any)**
- **TCP/UDP mechanisms explained step-by-step**
- **Common traps & confusions**
- **Quick revision bullets**
- And if you want:
 - **Practice questions (exam-style / engineering-focused)**
 - **USMLE-style analogies** (if helpful for memory)

You can go:

"Topic 1: UDP and multiplexing"
[paste Kurose text here]

And I'll convert it into **high-yield transport-layer notes**.

Send me your **first topic + book text** and we'll start building your Transport Layer master notes.



3.1: interaction between transport and network layers, overview

3.1 Introduction and Transport-Layer Services

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

A transport-layer protocol provides for logical communication between application processes running on different hosts. By logical communication, we mean that from an application's perspective, it is as if the hosts running the processes were directly connected; in reality, the hosts may be on opposite sides of the planet, connected via numerous routers and a wide range of link types. Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages. Figure 3.1 illustrates the notion of logical communication. As shown in Figure 3.1, transport-layer protocols are implemented in the end systems but not in network routers. On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer segments in Internet terminology. This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment. The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination. It's important to note that network routers act only on the network-layer fields of the datagram; that is, they do not examine the fields of the transport-layer segment encapsulated with the datagram. On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer. The transport layer then processes the received segment, making the data in the segment available to the receiving application. More than one transport-layer protocol may be 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.

3.1.1 Relationship Between Transport and Network Layers

Recall that the transport layer lies just above the network layer in the protocol stack. Whereas a transport-layer protocol provides logical communication between processes running on different hosts, a network-layer protocol provides logical

186 CHAPTER 3 • TRANSPORT LAYER

3.1 • INTRODUCTION AND TRANSPORT-LAYER SERVICES 187

Mobile Network

National or

Global ISP

Local or

Regional ISP

Enterprise Network

Network Home Network

Data link

Physical

Application

Transport

Network

Data link

Physical
Application
Transport
Network
Data link
Physical
Network
Data link
Physical
Network
Data link
Physical
Network
Data link
Physical
Network
Data link

Physical Logical end-to-end transport

Figure 3.1 [STX] The transport layer provides logical rather than physical communication between application processes

communication between hosts. This distinction is subtle but important. Let's examine this distinction with the aid of a household analogy.

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

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

application messages = letters in envelopes

processes = cousins

hosts (also called end systems) = houses

transport-layer protocol = Ann and Bill

network-layer protocol = postal service (including mail carriers)

Continuing with this analogy, note that Ann and Bill do all their work within their respective homes; they are not involved, for example, in sorting mail in any intermediate mail center or in moving mail from one mail center to another. Similarly, transport-layer protocols live in the end systems. Within an end system, a

transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn't have any say about how the messages are moved within the network core. In fact, as illustrated in Figure 3.1, intermediate routers neither act on, nor recognize, any information that the transport layer may have added to the application messages.

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

The possible services that Ann and Bill can provide are clearly constrained by the possible services that the postal service provides. For example, if the postal service doesn't provide a maximum bound on how long it can take to deliver mail between the two houses (for example, three days), then there is no way that Ann and Bill can guarantee a maximum delay for mail delivery between any of the cousin pairs. In a similar manner, the services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees for transport-layer segments sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for application 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, or duplicates packets. As another example (which we'll explore in Chapter 8 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 confidentiality of transport-layer segments.

3.1.2 Overview of the Transport Layer in the Internet

Recall that the Internet, and more generally a TCP/IP network, makes two distinct transport-layer protocols available 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 Section 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 transport-layer packet as a segment. We mention, however, that the Internet literature (for example, the RFCs) also refers to the transport-layer packet for TCP as a segment but often refers to the packet for UDP as a datagram. But this same Internet literature also uses the term datagram for the network-layer packet! For an introductory book on computer networking such as this, we believe that it is less confusing to refer to both TCP and UDP packets as segments, and reserve the term datagram for the network-layer packet. Before proceeding with our brief introduction of UDP and TCP, it will be useful to say a few words

about the Internet's network layer. (We'll learn about the network layer in detail in Chapter 4.)

The Internet's network-layer protocol has a

3.1 • INTRODUCTION AND TRANSPORT-LAYER SERVICES 189

name—IP, for Internet Protocol. IP provides logical communication between hosts.

The IP service model is a best-effort delivery service. This means that IP makes its "best effort" to deliver segments between communicating hosts, but it makes no guarantees.

In particular, it does not guarantee segment delivery, it does not guarantee orderly delivery of segments, and it does not guarantee the integrity of the data in the segments. For these reasons, IP is said to be an unreliable service. We also mention here that every host has at least one network-layer address, a so-called IP address. We'll examine IP addressing in detail in Chapter 4; for this chapter we need only keep in mind that each host has an IP address.

Having taken a glimpse at the IP service model, let's now summarize the service models provided by 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 transport-layer multiplexing and demultiplexing. We'll discuss transport-layer multiplexing and demultiplexing in the next section. UDP and TCP also provide integrity checking by including error detection fields in their segments' headers. These two minimal transport-layer services—process-to-process data delivery and error checking—are the only two services that UDP provides! In particular, like IP, UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact (or at all!) to the destination process. UDP is discussed in detail in Section 3.3.

TCP, on the other hand, offers several additional services to applications. First and foremost, it provides reliable data transfer. Using flow control, sequence numbers, acknowledgments, and timers (techniques we'll explore in detail in this chapter), TCP ensures that data is delivered from sending process to receiving process, correctly and in order. TCP thus converts IP's unreliable service between end systems into a reliable data transport service between processes. TCP also provides 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. Loosely speaking, TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic. TCP strives to give each connection traversing a congested link an equal share of the link bandwidth. This is done by regulating the rate at which the sending sides of TCP connections can send traffic into the network.

UDP traffic, on the other hand, is unregulated. An application using UDP transport can send at any rate it pleases, for as long as it pleases.

A protocol that provides reliable data transfer and congestion control is necessarily complex.

We'll need several sections to cover the principles of reliable data

transfer and congestion control, and additional sections to cover the TCP protocol

itself. These topics are investigated in Sections 3.4 through 3.8. The approach taken

in this chapter is to alternate between basic principles and the TCP protocol. For

example, we'll first discuss reliable data transfer in a general setting and then discuss how TCP specifically provides reliable data transfer. Similarly, we'll first

190 CHAPTER 3 • TRANSPORT LAYER

discuss congestion control in a general setting and then discuss how TCP performs congestion control. But before getting into all this good stuff, let's first look at transport-layer multiplexing and demultiplexing.

Excellent—this is a **core conceptual section**. I'll convert **3.1 (Interaction between Transport & Network Layers + Overview)** into **exam-ready study notes** with:

- Clean structure
 - Intuition
 - Key distinctions
 - TCP vs UDP summary
 - Common traps
 - Quick revision bullets
-

3.1 Transport Layer: Interaction with Network Layer & Overview

(Kurose & Ross — Exam-Ready Notes)

1 What Does the Transport Layer Do? (Big Picture)

Definition (Exam Line)

The **transport layer provides logical end-to-end communication between application processes running on different hosts**.

Meaning of “Logical Communication”

- Applications behave **as if they are directly connected**
- In reality:
 - Data passes through **routers, links, and networks**
- Applications **do NOT worry about:**
 - Physical links
 - Routers
 - Network congestion
 - Packet loss

Key Idea:

Transport hides all network complexity from applications.

2 Where Does Transport Layer Exist?

Transport Layer Lives Only In:

- End systems (hosts)
- NOT in routers

What Routers Do:

- Routers only look at:
 - IP headers (network layer)
- Routers do NOT:
 - Read TCP headers
 - Read UDP headers
 - Care about ports or processes

Exam favorite:

"Transport layer intelligence exists only at end hosts."

3 Data Flow: Application → Transport → Network

On the Sender:

1. Application sends message
2. Transport:
 - Breaks message into chunks
 - Adds **transport header** → creates **segment**
3. Network layer:
 - Wraps segment into **IP datagram**
4. Datagram sent into network

On the Receiver:

1. IP extracts **segment**
2. Transport processes segment
3. Delivers data to correct **application process**

4 Transport vs Network Layer (Household Analogy)

Networking Concept	House Analogy
Application processes	Cousins
Hosts	Houses
Transport layer	Ann & Bill (internal mail delivery)
Network layer	Postal service

Networking Concept	House Analogy
Message	Letters
Packet	Envelope

✓ Key Distinction:

- **Network Layer:**
→ Moves data **host-to-host**
- **Transport Layer:**
→ Moves data **process-to-process**

✗ Exam Trap:

Many students incorrectly say IP delivers data between **processes**.

✓ Correct: IP delivers between **hosts only**.

5 Relationship Between Transport & Network Layers

✓ Important Principle:

The transport layer is constrained by what the network layer can offer.

Network Can Do	Transport Can Do
No delay guarantee	✗ No delay guarantee
No bandwidth guarantee	✗ No bandwidth guarantee
Unreliable delivery	Transport can still make it reliable ✓
No security	Transport can encrypt ✓

✓ Critical Insight:

- Even though **IP is unreliable**, **TCP makes communication reliable**
- Even though **IP isn't secure**, **TLS/TCP can encrypt**

6 Internet Transport Protocols Overview

The Internet has **two transport protocols**:

Feature	UDP	TCP
Connection	✗ Connectionless	✓ Connection-oriented
Reliability	✗ Unreliable	✓ Reliable
Ordering	✗ Not guaranteed	✓ Guaranteed
Flow control	✗ No	✓ Yes
Congestion control	✗ No	✓ Yes

Feature	UDP	TCP
Speed	✓ Fast	✗ Slower
Header size	8 bytes	20+ bytes

7 IP Layer (Network Layer) Service Model

✓ IP Provides:

- Host-to-host delivery
- Best-effort service

✗ IP Does NOT Guarantee:

- Packet delivery
- Ordering
- No corruption
- No duplication
- No delay bound

✓ This is why:

IP is called an unreliable protocol

8 What Services Do UDP & TCP Add on Top of IP?

✓ Both UDP & TCP Provide:

1. Process-to-process delivery
2. Error detection

✓ UDP Services (Minimalist)

- Process delivery
- Error checking
- ✗ No reliability
- ✗ No ordering
- ✗ No congestion control

↗ Used by:

- DNS

- VoIP
 - Streaming
 - Online gaming
-

✓ TCP Services (Full-Featured)

TCP adds everything UDP doesn't:

1. ✓ Reliable delivery
2. ✓ In-order delivery
3. ✓ Flow control
4. ✓ Congestion control
5. ✓ Connection setup (3-way handshake)

📌 Used by:

- HTTP/HTTPS
 - FTP
 - SMTP
 - SSH
-

9 Transport Multiplexing & Demultiplexing

✓ Meaning:

- Multiple applications share the network simultaneously
- Transport layer:
 - **Multiplexing:** many sockets → one IP
 - **Demultiplexing:** one IP → correct process

This is done using:

- ✓ Port numbers
-

🔴 Common Exam Traps

Trap	Correct Concept
IP gives reliability	✗ TCP gives reliability
Routers run TCP	✗ Only hosts run TCP
UDP has congestion control	✗ Only TCP does
IP delivers between processes	✗ Transport does

Trap	Correct Concept
UDP guarantees delivery	✗ No guarantee

✓ Super-High-Yield Revision Bullets

- Transport = **process-to-process**
- Network = **host-to-host**
- Transport lives only in **hosts**
- IP is **best-effort & unreliable**
- UDP = fast, no guarantees
- TCP = reliable, ordered, congestion-controlled
- Congestion control protects the **network**, not just the app
- Multiplexing uses **port numbers**

✓ One-Line Exam Summary (Gold Line)

IP moves packets between hosts unreliably, while TCP/UDP extend this service to reliable or unreliable communication between application processes.

If you want, next I can convert this into:

- ✓ Quick exam cheat-sheet
- ✓ MCQ practice
- ✓ TCP vs UDP decision questions
- ✓ Transport-layer numericals

✓ Send your next Transport Layer topic (3.2 Multiplexing/Demultiplexing?) and I'll continue the notes in this same clean style.



3.2: connection-oriented and connectionless multiplexing and demultiplexing

3.2 Multiplexing and Demultiplexing

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

At the destination host, the transport layer receives segments from the network layer just below. The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host. Let's take a look at an example. Suppose you are sitting in front of your computer, and you are downloading Web pages while running one FTP session and two Telnet sessions. You therefore have four network application processes running—two Telnet processes, one FTP process, and one HTTP process. When the transport layer in your computer receives data from the network layer below, it needs to direct the received data to one of these four processes. Let's now examine how this is done. First recall from Section 2.7 that a process (as part of a network application) can have one or more sockets, doors through which data passes from the network to the process and through which data passes from the process to the network. Thus, as shown in Figure 3.2, the transport layer in the receiving host does not actually deliver data directly to a process, but instead to an intermediary socket. Because at any given time there can be more than one socket in the receiving host, each socket has a unique identifier. The format of the identifier depends on whether the socket is a UDP or a TCP socket, as we'll discuss shortly.

Now let's consider how a receiving host directs an incoming transport-layer segment to the appropriate socket. Each transport-layer segment has a set of fields in the segment for this purpose. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called demultiplexing. The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called multiplexing. Note that the transport layer in the middle host in Figure 3.2 must demultiplex segments arriving from the network layer below to either process P1 or P2 above; this is done by directing the arriving segment's data to the corresponding process's socket. The transport layer in the middle host must also

3.2 • MULTIPLEXING AND DEMULTIPLEXING 191

gather outgoing data from these sockets, form transport-layer segments, and pass these segments down to the network layer. Although we have introduced multiplexing and demultiplexing in the context of the Internet transport protocols, it's important to realize that they are concerns whenever a single protocol at one layer (at the transport layer or elsewhere) is used by multiple protocols at the next higher layer.

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

Now that we understand the roles of transport-layer multiplexing and demultiplexing, let us examine how it is actually done in a host. From the discussion above, we know that transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields, illustrated in Figure 3.3, are the source port number field and the destination port number field. (The UDP and TCP segments have other fields as well, as discussed in the subsequent sections of this chapter.) Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known

application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21). The list of well-known port numbers is given in RFC 1700 and is updated at <http://www.iana.org> [RFC 3232]. When we develop a new

192 CHAPTER 3 • TRANSPORT LAYER

Network

Key:

Process Socket

Data link

Physical

Transport

Application

Network

Application

Data link

Physical

Transport

Network

Data link

Physical

Transport

Network

Data link

Physical

Transport

P3 P1 P2 P4 Application

Figure 3.2 Transport-layer multiplexing and demultiplexing application (such as the simple application developed in Section 2.7), we must assign the application a port number.

It should now be clear how the transport layer could implement the demultiplexing service: Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket. The segment's data then passes through the socket into the attached process. As we'll see, this is basically how UDP does it. However, we'll also see that multiplexing/demultiplexing in TCP is yet more subtle.

Connectionless Multiplexing and Demultiplexing

Recall from Section 2.7.1 that the Python program running in a host can create a UDP socket with the line

```
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
```

When a UDP socket is created in this manner, the transport layer automatically assigns a port number to the socket. In particular, the transport layer assigns a port number in the range 1024 to 65535 that is currently not being used by any other UDP port in the host. Alternatively, we can add a line into our Python program after we create the socket to associate a specific port number (say, 19157) to this UDP socket via the socket bind() method:

```
clientSocket.bind("", 19157)
```

If the application developer writing the code were implementing the server side of a "well-known protocol," then the developer would have to assign the corresponding

3.2 • MULTIPLEXING AND DEMULTIPLEXING 193

Source port #

32 bits

Dest. port #

Other header fields

Application

data

(message)

Figure 3.3 [STX] Source and destination port-number fields in a transport-layer segment

well-known port number. Typically, the client side of the application lets the transport layer automatically (and transparently) assign the port number, whereas the server side of the application assigns a specific port number.

With port numbers assigned to UDP sockets, we can now precisely describe UDP multiplexing/demultiplexing. Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B. The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428), and two other values (which will be discussed later, but are unimportant for the current discussion). The transport layer then passes the resulting segment to the network layer. The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host.

If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428. Note that Host B could be running multiple processes, each with its own UDP socket and associated port number. As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination port number.

It is important to note that a UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number. As a consequence, if two UDP segments have different source IP addresses and/or source port numbers, but have the same destination IP address and destination port number, then the two segments will be directed to the same destination process via the same destination socket.

You may be wondering now, what is the purpose of the source port number? As shown in Figure 3.4, in the A-to-B segment the source port number serves as part of a "return address"—when B wants to send a segment back to A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment. (The complete return address is A's IP address and the source port number.) As an example, recall the UDP server program studied in Section 2.7. In `UDPServer.py`, the server uses the `recvfrom()` method to extract the client-side (source) port number from the segment it receives from the client; it then sends a new segment to the client, with the extracted source port number serving as the destination port number in this new segment.

Connection-Oriented Multiplexing and Demultiplexing

In order to understand TCP demultiplexing, we have to take a close look at TCP sockets and TCP connection establishment. One subtle difference between a TCP socket and a UDP socket is that a TCP socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket. In particular, and in contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will (with the exception of a TCP segment carrying the original connection-establishment request) be directed to two different sockets. To gain further insight, let's reconsider the TCP client-server programming example in Section 2.7.2:

- The TCP server application has a "welcoming socket," that waits for connection establishment requests from TCP clients (see Figure 2.29) on port number 12000.
- The TCP client creates a socket and sends a connection establishment request

segment with the lines:

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, 12000))
```

- A connection-establishment request is nothing more than a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header (discussed in Section 3.5). The segment also includes a source port number that was chosen by the client.
- When the host operating system of the computer running the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000. The server process then creates a new socket:

```
connectionSocket, addr = serverSocket.accept()
```

3.2 • MULTIPLEXING AND DEMULTIPLEXING 195

Host A

Client process

Socket

Server B

source port:

19157

dest. port:

46428

source port:

46428

dest. port:

19157

Figure 3.4 The inversion of source and destination port numbers

- Also, the transport layer at the server notes the following four values in the connection-request segment: (1) the source port number in the segment, (2) the IP address of the source host, (3) the destination port number in the segment, and (4) its own IP address. The newly created connection socket is identified by these four values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket. With the TCP connection now in place, the client and server can now send data to each other.

The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four-tuple. When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (demultiplex) the segment to the appropriate socket.

196 CHAPTER 3 • TRANSPORT LAYER

PORt SCANNING

We've seen that a server process waits patiently on an open port for contact by a remote client. Some ports are reserved for well-known applications (e.g., Web, FTP, DNS, and SMTP servers); other ports are used by convention by popular applications (e.g., the Microsoft 2000 SQL server listens for requests on UDP port 1434). Thus, if we determine that a port is open on a host, we may be able to map that port to a specific application running on the host. This is very useful for system administrators, who are often interested in knowing which network applications are running on the hosts in their networks. But attackers, in order to "case the joint," also want to know which ports are open on target hosts. If a host is found to be running an application with a known security flaw (e.g., a SQL server listening on port 1434 was subject to

a buffer overflow, allowing a remote user to execute arbitrary code on the vulnerable host, a flaw exploited by the Slammer worm [CERT 2003–04]), then that host is ripe for attack.

Determining which applications are listening on which ports is a relatively easy task. Indeed there are a number of public domain programs, called port scanners, that do just that. Perhaps the most widely used of these is nmap, freely available at <http://nmap.org> and included in most Linux distributions. For TCP, nmap sequentially scans ports, looking for ports that are accepting TCP connections. For UDP, nmap again sequentially scans ports, looking for UDP ports that respond to transmitted UDP segments. In both cases, nmap returns a list of open, closed, or unreachable ports. A host running nmap can attempt to scan any target host anywhere in the Internet. We'll revisit nmap in Section 3.5.6, when we discuss TCP connection management.

FOCUS ON SECURITY

The situation is illustrated in Figure 3.5, in which Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B. Hosts A and C and server B each have their own unique IP address—A, C, and B, respectively. Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections. Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection. But this is not a problem—server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses.

Web Servers and TCP

Before closing this discussion, it's instructive to say a few additional words about Web servers and how they use port numbers. Consider a host running a Web server, such as an Apache Web server, on port 80. When clients (for example, browsers) send segments to the server, all segments will have destination port 80. In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80. As we have just described,

3.2 • MULTIPLEXING AND DEMULTIPLEXING 197

source port:

7532

dest. port:

80

source IP:

C

dest. IP:

B

source port:

26145

dest. port:

80

source IP:

C

dest. IP:

B

source port:

26145

dest. port:

80

source IP:

A
 dest. IP:
 B
 Per-connection
 HTTP
 processes
 Transport^{STX}layer
 demultiplexing
 Web
 server B
 Web client
 host C
 Web client
 host A

Figure 3.5  Two clients, using the same destination port number (80) to communicate with the same Web server application the server distinguishes the segments from the different clients using source IP addresses and source port numbers.

Figure 3.5 shows a Web server that spawns a new process for each connection. As shown in Figure 3.5, each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent. We mention, however, that there is not always a one-to-one correspondence between connection sockets and processes. In fact, today's high-performing Web servers often use only one process, and create a new thread with a new connection socket for each new client connection. (A thread can be viewed as a lightweight subprocess.) If you did the first programming assignment in Chapter 2, you built a Web server that does just this. For such a server, at any given time there may be many connection sockets (with different identifiers) attached to the same process.

If the client and server are using persistent HTTP, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket. However, if the client and server use non-persistent HTTP, then a new TCP connection is created and closed for every request/response, and hence a new socket is created and later closed for every request/response. This frequent creating and closing of sockets can severely impact the performance of a busy Web server (although a number of operating system tricks can be used to mitigate the problem). Readers interested in the operating system issues surrounding persistent and non-persistent HTTP are encouraged to see [Nielsen 1997; Nahum 2002].

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

Nice, this is one of the most testable parts of the transport layer. Let's turn all of **3.2** into clean notes focused on:

-  What *multiplexing/demultiplexing* actually mean
-  How **UDP (connectionless)** does it

-  How **TCP (connection-oriented)** does it
 -  How sockets & port numbers fit into all of this
-

3.2 Multiplexing and Demultiplexing

(Connectionless vs Connection-Oriented)

1 What Are Multiplexing & Demultiplexing?

- ◆ **Demultiplexing (receiver side)**

Deliver each **incoming segment** to the **correct socket / application**.

- Segment comes from **network layer** → **transport layer**
- Transport layer looks at **header fields** (port numbers + sometimes IPs)
- Uses them to decide “*Which socket should get this data?*”

- ◆ **Multiplexing (sender side)**

Collect data from many **sockets/applications**, add transport headers, and pass segments down to the network.

- Many applications → transport layer → one IP address
- Transport adds:
 - **Source port**
 - **Destination port**
 - Other header fields

 **Big idea:**

Network layer only knows **hosts** (via IP addresses).

Transport layer extends this to **processes** (via sockets & port numbers).

2 Sockets and Port Numbers

- ◆ **Sockets**

- A **socket** = the endpoint for sending/receiving data between app and transport layer.
- Process never deals with IP packets directly; it reads/writes through **sockets**.

- ◆ **Port Numbers**

- Each socket is identified by a **port number** at the host.
- Port number = 16-bit → **0-65535**

- **0-1023 → well-known ports**
 - HTTP: 80
 - HTTPS: 443
 - FTP: 21
 - DNS: 53
- **1024-65535** → ephemeral / dynamically assigned

How demux works in general

- Incoming segment has:
 - **Destination port number**
- Transport layer:
 1. Looks at destination port
 2. Finds matching socket
 3. Hands data to that socket

👉 That's the basic idea. Now the difference between **UDP** and **TCP** is *what tuple identifies the socket*.

3 Connectionless Multiplexing/Demultiplexing (UDP)

♦ How a UDP socket is identified

A UDP socket is identified by the **pair**:
(Destination IP address, Destination port number)

- That's it. **2-tuple**.

♦ Creating UDP sockets

- In code (Python example from Kurose):

```
python
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

- OS will:
 - Automatically pick an **ephemeral port** (1024-65535),
 - Or you can bind one manually:

```
python
clientSocket.bind(('', 19157))
```

♦ Example of UDP demux

- Host A, process with UDP port **19157**
- Host B, process with UDP port **46428**

When process on A sends to B:

- UDP segment header:
 - **Source port:** 19157
 - **Destination port:** 46428
- At Host B:
 - Transport checks **dest port = 46428**
 - Hands data to the socket bound to port 46428

♦ Important detail for UDP

Different senders ⇒ same destination port ⇒ same socket.

- Suppose:
 - Client 1 → B: dest port 5000
 - Client 2 → B: dest port 5000
- At B:
 - Both segments go to the **same UDP socket (IP_B, port 5000)**
- Why store the source port?
 - So B can **reply**:
 - It takes **source IP + source port** from incoming segment
 - Uses them as **destination IP + destination port** in the reply
 - So source port is part of the **return address**, but **not** part of how UDP *identifies* the socket.

📌 Key point to memorize:

- **UDP socket ID = (dest IP, dest port)**
- **UDP does not separate flows by source info** (multiple clients can share same server UDP socket).

4 Connection-Oriented Multiplexing/Demultiplexing (TCP)

Here's where it becomes more subtle.

♦ How a TCP socket (i.e., connection) is identified

A TCP connection is identified by a **4-tuple**:
(Source IP, Source port, Destination IP, Destination port)

So:

- Two segments with same destination port but different **source IP/port** → treated as **different connections**.

◆ Two kinds of TCP sockets on a server

1. Welcoming (listening) socket

- Bound to a **well-known port** (e.g., port 80 for HTTP, 12000 in Kurose's examples)
- Waits for **incoming connection requests**.

2. Connection socket

- Created after a TCP connection is established
- Identified by the **4-tuple**
- Used for **actual data transfer**

◆ TCP demux procedure (simplified)

1. Client creates a TCP socket and does:

```
python
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, 12000))
```

2. Client sends a segment with:

- Src IP = client's IP
- Src port = chosen by client
- Dest IP = server's IP
- Dest port = 12000
- Plus a special "**connection request**" flag (SYN) in TCP header.

3. Server OS:

- Sees segment with **dest port 12000**
- Hands it to the **welcoming socket**
- **Creates a new connection socket** identified by:
(Src IP, Src port, Dest IP, Dest port)

4. All subsequent segments of that connection:

- With **matching 4-tuple**
- Get demuxed to that connection socket.

◆ Multiple clients connecting to same TCP port

Kurose's example:

- Host C → B: HTTP connection 1
 - Src IP: C, Src port: 26145, Dest IP: B, Dest port: 80

- Host C → B: HTTP connection 2
 - Src IP: C, Src port: 7532, Dest IP: B, Dest port: 80
- Host A → B: HTTP connection
 - Src IP: A, Src port: 26145, Dest IP: B, Dest port: 80

At server B, it sees **three distinct 4-tuples**, so it knows:

- These are **three different TCP connections**, even though:
 - They all go to **the same dest port (80)**
 - Two of them share the same **source port number** (26145) but IPs differ.

📌 Key takeaway:

- **TCP does per-connection demux** using **all four fields**.
- That's why multiple clients can talk to the same port 80 at the same time.

5 Quick UDP vs TCP Multiplexing Summary

Feature	UDP (Connectionless)	TCP (Connection-Oriented)
Socket / connection ID	(Destination IP, Destination port)	(Source IP, Source port, Destination IP, Destination port)
Uses source port to demux?	✗ No (only dest IP+port used to pick socket)	✓ Yes (full 4-tuple used)
Multiple clients same server port	All share the same socket on server	Each connection gets its own 4-tuple and socket
Typical use	DNS, streaming, simple request-response	HTTP, HTTPS, FTP, SSH, email
Connection state	None	Maintains per-connection state

6 One-liner “golden” memory lines

- **UDP**: connectionless, demux only by **destination port (and IP)** → many clients can share same server socket.
- **TCP**: connection-oriented, demux by **full 4-tuple** → each connection is unique.

If you want, next I can:

- Make **2-3 exam-style conceptual questions** just on this part
- Or move on to **3.3 UDP** and I'll turn that into another tight note set.



3.3: UDP services, pro and cons of connectionless transport

3.3 Connectionless Transport: UDP

In this section, we'll take a close look at UDP, how it works, and what it does.

We encourage you to refer back to Section 2.1, which includes an overview of the UDP service model, and to Section 2.7.1, which discusses socket programming using UDP. To motivate our discussion about UDP, suppose you were interested in designing a no-frills, bare-bones transport protocol. How might you go about doing this?

You might first consider using a vacuous transport protocol. In particular, on the sending side, you might consider taking the messages from the application process and passing them directly to the network layer; and on the receiving side, you might consider taking the messages arriving from the network layer and passing them directly to the application process. But as we learned in the previous section, we have to do a little more than nothing! At the very least, the transport layer has to

198 CHAPTER 3 • TRANSPORT LAYER

provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application-level process.

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

DNS is an example of an application-layer protocol that typically uses UDP.

When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to UDP. Without performing any handshaking with the UDP entity running on the destination end system, the host-side UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply (possibly because the underlying network lost the query or the reply), either it tries sending the query to another name server, or it informs the invoking application that it can't get a reply.

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, since TCP provides a reliable data transfer service, while UDP does not? The answer is no, as many applications are better suited for UDP for the following reasons:

- Finer application-level control over what data is sent, and when. Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested. TCP will also continue to resend a

segment until the receipt of the segment has been acknowledged by the destination, regardless of how long reliable delivery takes. Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs. As discussed below, these applications can use UDP and implement, as part of the application, any additional functionality that is needed beyond UDP's no-frills segment-delivery service.

3.3 • CONNECTIONLESS TRANSPORT: UDP 199

- No connection establishment. As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text. But, as we briefly discussed in Section 2.2, the TCP connection-establishment delay in HTTP is an important contributor to the delays associated with downloading Web documents.
- No connection state. TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. We will see in Section 3.5 that this state information is needed to implement TCP's reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.
- Small packet header overhead. The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

Figure 3.6 lists popular Internet applications and the transport protocols that they use. As we expect, e-mail, remote terminal access, the Web, and file transfer run over TCP—all these applications need the reliable data transfer service of TCP. Nevertheless, many important applications run over UDP rather than TCP. UDP is used for RIP routing table updates (see Section 4.6.1). Since RIP updates are sent periodically (typically every five minutes), lost updates will be replaced by more recent updates, thus making the lost, out-of-date update useless. UDP is also used to carry network management (SNMP; see Chapter 9) data. UDP is preferred to TCP in this case, since network management applications must often run when the network is in a stressed state—precisely when reliable, congestion-controlled data transfer is difficult to achieve. Also, as we mentioned earlier, DNS runs over UDP, thereby avoiding TCP's connection-establishment delays.

As shown in Figure 3.6, both UDP and TCP are used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video. We'll take a close look at these applications in Chapter 7. We just mention now that all of these applications can tolerate a small amount of packet loss, so that reliable data transfer is not absolutely critical for the application's success. Furthermore, real-time applications, like Internet phone and video conferencing, react very poorly to TCP's congestion control. For these reasons, developers of multimedia applications may choose to run their applications over UDP instead of TCP. However, TCP is increasingly being used for streaming media transport. For example, [Sripanidkulchai 2004] found that nearly 75% of on-demand and live streaming used TCP. When packet loss rates are low, and with some organizations

200 CHAPTER 3 • TRANSPORT LAYER

blocking UDP traffic for security reasons (see Chapter 8), TCP becomes an increasingly attractive

protocol for streaming media transport.

Although commonly done today, running multimedia applications over UDP is controversial. As we mentioned above, UDP has no 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-bitrate video without using any congestion control, there would be so much packet overflow at routers that very few UDP packets would successfully traverse the source-to-destination path. Moreover, the high loss rates induced by the uncontrolled UDP senders would cause the TCP senders (which, as we'll see, do decrease their sending rates in the face of congestion) to dramatically decrease their rates. Thus, the lack of congestion control in UDP can result in high loss rates between a UDP sender and receiver, and the crowding out of TCP sessions—a potentially serious problem [Floyd 1999]. Many researchers have proposed new mechanisms to force all sources, including UDP sources, to perform adaptive congestion control [Mahdavi 1997; Floyd 2000; Kohler 2006: RFC 4340].

Before discussing the UDP segment structure, we mention that it is possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms, such as those we'll study in the next section). But this is a nontrivial task that would keep an application developer busy debugging for

3.3 • CONNECTIONLESS TRANSPORT: UDP 201

Application-Layer Underlying Transport

Application Protocol 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 typically proprietary UDP or TCP

Internet telephony typically proprietary UDP or TCP

Network management SNMP Typically UDP

Routing protocol RIP Typically UDP

Name translation DNS Typically UDP

Figure 3.6 Popular Internet applications and their underlying transport protocols

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 subjected to the transmission-rate constraints imposed by TCP’s congestion-control mechanism

Nice, this is a super “definition + reasoning” style section. I’ll turn **3.3 (UDP)** into:

- What UDP actually does (its service model)
- Where it is used
- Advantages (why choose UDP over TCP)
- Disadvantages / dangers
- A few gold “exam lines”

3.3 Connectionless Transport: UDP

Services, and Pros & Cons of Connectionless Transport

1 What Does UDP Actually Do?

- ♦ Minimal Transport Layer

UDP = IP + ports + checksum (light error check).

UDP provides:

1. Multiplexing / demultiplexing via:
 - Source port number
 - Destination port number
2. Light error checking via checksum
3. No handshake before sending ⇒ connectionless

How it works:

- Sender side
 - Takes application message
 - Adds:
 - Source port
 - Destination port
 - Length
 - Checksum
 - → forms a **UDP segment**
 - Hands it to IP for **best-effort delivery**
- Receiver side
 - IP delivers UDP segment to host
 - UDP checks destination port
 - Hands data to **the correct socket / application**
 - If checksum fails, segment may be discarded

No reliability, no retransmission, no ordering, no congestion control.

🧠 Key phrase:

"UDP adds just enough to IP to support multiple applications: ports + checksum. Nothing more."

2 Why Is UDP Called *Connectionless*?

- There is **no connection establishment** before data is sent
- No **3-way handshake** (unlike TCP)
- Sender “just blasts” segments
- Receiver does not maintain **per-connection state**

Example: **DNS over UDP**

- Client sends a DNS query in a UDP segment
- No prior handshake with server
- If reply is lost:
 - DNS client may retry or try another server
 - Reliability is implemented **at the application layer**, not transport

3 When Would a Developer Choose UDP Over TCP?

At first it feels like “TCP is always better because it’s reliable,” but many apps prefer UDP.

✓ Advantages of UDP (Pros)

1. Fine-grained control over *what is sent and when*

- As soon as app gives data to UDP → UDP sends it immediately
- **No delay** due to:
 - Congestion control slowdown
 - Waiting for ACKs
 - Buffering / Nagle’s algorithm, etc.
- Great for:
 - **Real-time audio/video**, gaming, VoIP
 - Apps that **tolerate some loss but hate delay**

2. No connection establishment (no handshake delay)

- No 3-way handshake like TCP
- Lower latency for **short interactions / single queries**
- Major reason why **DNS uses UDP**:
 - Using TCP would add connection setup + teardown for every small query

3. No connection state at sender/receiver

- TCP must maintain per-connection:

- Send buffer / receive buffer
- Sequence numbers
- ACKs, timers
- Congestion-control variables
- UDP keeps **no per-connection state**:
 - Easier to implement
 - **Servers can handle more concurrent clients** using UDP (lighter per-client overhead)

4. Smaller header (lower overhead)

- **UDP header = 8 bytes**
- **TCP header = 20 bytes (minimum)**
→ Less overhead per packet, especially good for small messages (e.g., DNS)

5. Can build custom reliability on top (if needed)

- App can:
 - Add its own sequence numbers
 - ACKs
 - Retransmission logic
- Allows:

"Have reliability *and* avoid TCP's congestion-control behavior."

4 Disadvantages of UDP (Cons / Risks)

1. ✗ No reliable data transfer

- Packets may be:
 - Lost
 - Duplicated
 - Arrive corrupted (if checksum fails and is ignored/discard)
- No automatic retransmission
- App must handle loss itself if reliability is needed

2. ✗ No ordering or reassembly

- Packets can arrive **out of order**
- App must reorder if needed

3. ✗ No congestion control

- UDP sends at whatever rate the app wants:
 - Can **flood the network**

- Can cause **router buffer overflow** → massive packet loss
 - TCP flows sharing the path **back off** when they see loss
→ UDP flows can “bully” TCP flows and starve them
 - This is why **uncontrolled UDP streaming at high bitrates** is considered bad behavior.
4. **✗ If you need strong reliability, it's harder for the developer**

- Must **reimplement reliability logic** (ACKs, retransmissions, etc.) at application level
 - Extra complexity and more chances to get it wrong
-

5 Typical Applications Using UDP vs TCP

From Figure 3.6 (high yield to memorize):

Over TCP (need reliability)

- Email → **SMTP**
- Remote terminal → **Telnet / SSH**
- Web → **HTTP**
- File transfer → **FTP**

Typically over UDP

- DNS
- RIP (routing)
- SNMP (network management)
- Some streaming / multimedia protocols
- Some Internet telephony protocols

Can use either UDP or TCP (often proprietary)

- Streaming media (video/audio)
- Internet telephony (VoIP)

Note: In practice, a lot of **streaming** is done over **TCP** now, especially because:

- Firewalls often block UDP
 - TCP works reasonably well when loss is low
-

6 Connectionless Transport: Pros & Cons Summary Table

Aspect	UDP (Connectionless)	TCP (Connection-Oriented)
Setup handshake	✗ None (no delay)	✓ 3-way handshake (delay & state setup)
Reliability	✗ None	✓ Yes (retransmission, ACK, seq #s)

Aspect	UDP (Connectionless)	TCP (Connection-Oriented)
Ordering	✗ None	✓ In-order delivery
Congestion control	✗ None	✓ Yes (fairness, avoids meltdown)
Header size	8 bytes	≥ 20 bytes
State at endpoints	✗ No connection state	✓ Per-connection state
Latency	Very low for small messages	Higher (handshake + congestion control)
Good for	DNS, VoIP, live video, gaming, custom protocols	Web, email, file transfer, anything needing 100% reliability

7 Golden “Exam Sentences”

- **UDP:**

“UDP provides connectionless, best-effort, process-to-process delivery with minimal overhead and no reliability or congestion control.”

- **Why use UDP instead of TCP?**

“To avoid connection setup delay, maintain no per-connection state, have fine-grained app control over timing, and reduce header overhead—usually in applications that tolerate some loss.”

- **Risk of UDP without congestion control:**

“Uncontrolled UDP flows can cause congestion collapse and starve TCP flows, because UDP does not reduce its sending rate in response to loss.”

If you'd like, next I can:

- Make **5–6 conceptual MCQs** just on **UDP vs TCP choice & pros/cons**, or move on to **3.4: Principles of Reliable Data Transfer** and we'll start building that block next.



3.4: principles (retransmission, feedback, error detection, numbering, timeouts), stop-and-wait, Go-Back-N, selective repeat

3.4.1 Building a Reliable Data Transfer Protocol

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

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

We first consider the simplest case, in which the underlying channel is completely reliable. The protocol itself, which we'll call rdt1.0, is trivial. The finite-state machine (FSM) definitions for the rdt1.0 sender and receiver are shown in

Figure 3.9. The FSM in Figure 3.9(a) defines the operation of the sender, while the FSM in Figure 3.9(b) defines the operation of the receiver. It is important to note that there are separate FSMs for the sender and for the receiver. The sender and receiver FSMs in Figure 3.9 each have just one state. The arrows in the FSM description indicate the transition of the protocol from one state to another. (Since each FSM in Figure 3.9 has just one state, a transition is necessarily from the one state back to itself; we'll see more complicated state diagrams shortly.) The event causing the transition is shown above the horizontal line labeling the transition, and

206 CHAPTER 3 • TRANSPORT LAYER

```

Wait for
call from
above
a. rdt1.0: sending side
rdt_send(data)
packet=make_pkt(data)
udt_send(packet)
Wait for
call from
below
b. rdt1.0: receiving side
rdt_rcv(packet)
extract(packet,data)
deliver_data(data)

```

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

the actions taken when the event occurs are shown below the horizontal line. When no action is taken on an event, or no event occurs and an action is taken, we'll use the symbol below or above the horizontal, respectively, to explicitly denote the lack of an action or event. The initial state of the FSM is indicated by the dashed arrow. Although the FSMs in Figure 3.9 have but one state, the FSMs we will see shortly have multiple states, so it will be important to identify the initial state of each FSM.

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

On the receiving side, rdt receives a packet from the underlying channel via the `rdt_rcv(packet)` event, removes the data from the packet (via the action `extract(packet, data)`) and passes the data up to the upper layer (via the action `deliver_data(data)`). In practice, the `rdt_rcv(packet)` event

would result from a procedure call (for example, to `rdt_rcv()`) from the lower layer protocol.

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

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

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted. Such bit errors typically occur in the physical components of a network as a packet is transmitted, propagates, or is buffered. We'll continue to

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

3.4 • PRINCIPLES OF RELIABLE DATA TRANSFER 207

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

- Error detection. First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. Recall from the previous section that UDP uses the Internet checksum field for exactly this purpose. In Chapter 5 we'll examine error-detection and -correction techniques in greater detail; these techniques allow the receiver to detect and possibly correct packet bit errors. For now, we need only know that these techniques require that extra bits (beyond the bits of original data to be transferred) be sent from the sender to the receiver; these bits will be gathered into the packet checksum field of the rdt2.0 data packet.
- Receiver feedback. Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world (in this case, whether or not a packet was received correctly) is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback. Our rdt2.0 protocol will similarly send ACK and NAK packets back from the receiver to the sender. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.
- Retransmission. A packet that is received in error at the receiver will be retransmitted by the sender.

Figure 3.10 shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments. The send side of rdt2.0 has two states. In the leftmost state, the send-side protocol is waiting for data to be passed down from the upper layer. When the rdt_send(data) event occurs, the sender will create a packet (sndpkt) containing the data to be sent, along with a packet checksum (for example, as discussed in Section 3.3.2 for the case of a UDP segment), and then send the packet via the udt_send(sndpkt) operation. In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation rdt_rcv(rcvpkt) && isACK(rcvpkt) in Figure 3.10 corresponds to this event), the sender knows that 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 sender is in the wait-for-ACK-or-NAK

state, it cannot get more data from the upper layer; that is, the rdt_send() event can not occur; that will happen only after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as rdt2.0 are known as stop-and-wait protocols.

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

Protocol rdt2.0 may look as if it works but, unfortunately, it has a fatal flaw. In particular, we haven't accounted for the possibility that the ACK or NAK packet could be corrupted! (Before proceeding on, you should think about how this

3.4 • PRINCIPLES OF RELIABLE DATA TRANSFER 209

Wait for

call from

above

a. rdt2.0: sending side

b. rdt2.0: receiving side

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt=make_pkt(NAK)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_send(data)

sndpkt=make_pkt(data,checksum)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)

deliver_data(data)

sndpkt=make_pkt(ACK)

udt_send(sndpkt)

Wait for

call from

below

Wait for

ACK or

NAK

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

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 receiver 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 not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.
- A third approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, however, introduces duplicate packets into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know a priori whether an arriving packet contains new data or is a retransmission! A simple solution to this new problem (and one adopted in almost all existing data transfer protocols, including TCP) is to add a new field to the data packet and have the sender number its data packets by putting a sequence number into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission. For this simple case of a stop-and-wait protocol, a 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, moving "forward" in modulo-2 arithmetic). Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging. The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

210 CHAPTER 3 • TRANSPORT LAYER

Figures 3.11 and 3.12 show the FSM description for rdt2.1, our fixed version of rdt2.0. The rdt2.1 sender and receiver FSMs 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 acknowledgments from the receiver to the sender. When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received. When a corrupted packet is received, the receiver sends a negative acknowledgment. We can accomplish the same effect as a NAK if, instead of sending a NAK, we send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet (that is, receives duplicate ACKs) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Our NAK-free reliable data

3.4 • PRINCIPLES OF RELIABLE DATA TRANSFER 211

```
Wait for
call 0 from
above
rdt_rcv(rcvpkt)&&
```

```

(corrupt(rcvpkt) || 
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt)&&
(corrupt(rcvpkt) || 
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
&&
rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)
Wait for
ACK or
NAK 0
Wait for
ACK or
NAK 1
Wait for
call 1 from
above

```

Figure 3.11  rdt2.1 sender

transfer protocol for a channel with bit errors is rdt2.2, shown in Figures 3.13 and 3.14. One subtle change between rdt2.1 and rdt2.2 is that the receiver must now include the sequence number of the packet being acknowledged by an ACK message (this is done by including the ACK,0 or ACK,1 argument in make_pkt() in the receiver FSM), and the sender must now check the sequence number of the packet being acknowledged by a received ACK message (this is done by including the 0 or 1 argument in isACK() in the sender FSM).

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

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

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

packet has been lost, it can simply retransmit the data packet. You should convince yourself that this protocol does indeed work.

But how long must the sender wait to be certain that something has been lost?

The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers) 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 even to 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

Wait for

call 0 from

above

```
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || 
isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || 
isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)
```

Wait for

ACK 0

Wait for

ACK 1

ΛΛ

Wait for

call 1 from

above

Figure 3.13 rdt2.2 sender

the sender-to-receiver channel. Happily, protocol rdt2.2 already has enough functionality (that is, sequence numbers) to handle the case of duplicate packets. From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was

simply overly delayed. In all cases, the action is the same: retransmit. Implementing a time-based retransmission mechanism requires a countdown timer that can interrupt the sender after a given amount of time has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

Figure 3.15 shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets; in the homework problems, you'll be asked to provide the receiver FSM for rdt3.0. Figure 3.16 shows how the protocol operates with no lost or delayed packets and how it handles lost data packets. In Figure 3.16, time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. In Figures 3.16(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 acknowledgment packets each play a crucial and necessary role in the operation of the protocol. We now

have a working reliable data transfer protocol!

3.4.2 Pipelined Reliable Data Transfer Protocols

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

To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two hosts, one located on the West Coast of the United States and the other located on the East Coast, as shown in Figure 3.17. The speed-of-light round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a transmission rate, R , of 1 Gbps (109 bits per second). With a packet size, L , of 1,000 byte (8,000 bits) per packet, including both header fields and data, the time needed to actually transmit the packet into the 1 Gbps link is

d_{trans}

Figure 3.18(a) shows that with our stop-and-wait protocol, if the sender begins sending the packet at $t = 0$, then at $t = L/R = 8$ microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at $t = RTT/2 + L/R = 15.008$ msec. Assuming for simplicity that ACK packets are extremely small (so that we can ignore their transmission time) and that the receiver can send an ACK as soon as the last bit of a data packet is received, the ACK emerges back at the sender at $t = RTT + L/R = 30.008$ msec. At this point, the sender can now transmit the next message. Thus, in 30.008 msec, the sender was sending for only 0.008 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, the analysis in Figure 3.18(a) shows that the stop-and-wait protocol has a rather dismal sender utilization, U_{sender} , of

That is, the sender was busy only 2.7 hundredths of one percent of the time!

Viewed another way, the sender was able to send only 1,000 bytes in 30.008 milliseconds, an effective throughput of only 267 kbps—even though a 1 Gbps 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 267 kilobits per second!

This is a graphic example of how network protocols can limit the capabilities

Usender = L>R

RTT + L>R = .008

30.008 = 0.00027

= L

R = 8000 bits>packet

109

bits/sec = 8 microseconds

3.4 • PRINCIPLES OF RELIABLE DATA TRANSFER 217

Data packet Data packets

ACK packets

- a. A stop-and-wait protocol in operation
- b. A pipelined protocol in operation

Figure 3.17 Stop-and-wait versus pipelined protocol

provided by the underlying network hardware. Also, we have neglected lower-layer protocol-processing times at the sender and receiver, as well as the processing and queuing delays that would occur at any intermediate routers between the sender and receiver. Including these effects would serve only to further increase the delay and further accentuate the poor performance.

The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in Figure 3.17(b). Figure 3.18(b)

shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled. Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as pipelining. Pipelining has the following consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.
- The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.
- The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: Go-Back-N and selective repeat.

3.4.3 Go-Back-N (GBN)

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N, of unacknowledged packets in the pipeline. We describe the GBN protocol in some detail in this section. But before reading on, you are encouraged to play with the GBN applet (an awesome applet!) at the companion Web site.

Figure 3.19 shows the sender's view of the range of sequence numbers in a GBN protocol. If we define base to be the sequence number of the oldest unacknowledged packet and nextseqnum to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval [0,base-1] correspond to packets that have already been transmitted and acknowledged. The interval [base,nextseqnum-1] corresponds to packets that have been sent but not yet

acknowledged. Sequence numbers in the interval [nextseqnum,base+N-1] can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged.

As suggested by Figure 3.19, the range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size N over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, N is often referred to as the window size and the GBN protocol itself as a sliding-window protocol. You might be wondering why we would even limit the number of outstanding, unacknowledged packets to a value of N in the first place. Why not allow an unlimited number of such packets? We'll see in Section 3.5 that flow control is one reason to impose a limit on the sender. We'll examine another reason to do so in Section 3.7, when we study TCP congestion control.

In practice, a packet's sequence number is carried in a fixed-length field in the packet header. If k is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0, 2^k - 1]$. With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2^k arithmetic. (That is, the sequence number space can be thought of as a ring of size 2^k , where

sequence number 2^k

- 1 is immediately followed by sequence number 0.) Recall that

rdt3.0 had a 1-bit sequence number and a range of sequence numbers of $[0, 1]$. Several of the problems at the end of this chapter explore the consequences of a finite range of sequence numbers. We will see in Section 3.5 that TCP has a 32-bit sequence number field, where TCP sequence numbers count bytes in the byte stream rather than packets.

Figures 3.20 and 3.21 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 because we have added variables (similar to programming-language variables) for base and nextseqnum, and added operations on these variables and conditional actions involving these variables. Note that the extended FSM specification is now beginning to look somewhat like a programming-language specification. [Bochman 1984] provides an excellent survey of additional extensions to FSM techniques as well as other programming-language-based techniques for specifying protocol. The GBN sender must respond to three types of events:

- Invocation from above. When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are N outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call rdt_send() only when the window is not full.
- Receipt of an ACK. In our GBN protocol, an acknowledgment for a packet with sequence number n will be taken to be a cumulative acknowledgment, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver. We'll come back to this issue shortly when we examine the receiver side of GBN.

• A timeout event. The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends all packets that have been previously sent but that have not yet been acknowledged. Our sender in Figure 3.20 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 not yet acknowledged packets, the timer is restarted. If there are no outstanding, unacknowledged packets, the timer is stopped.

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

In our GBN protocol, the receiver discards out-of-order packets. Although it may seem silly and wasteful to discard a correctly received (but out-of-order) packet, there is some justification for doing so. Recall that the receiver must deliver data in order to the upper layer. Suppose now that packet n is expected, but packet $n + 1$ arrives. Because 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

222 CHAPTER 3 • TRANSPORT LAYER

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 3.21. 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 3.22 shows the operation of the GBN protocol for the case of a window size of four packets. Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding. As each successive ACK (for example, ACK0 and ACK1)

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

Before closing our discussion of GBN, it is worth noting that an implementation of this protocol in a protocol stack would likely have a structure similar to that of the extended FSM in Figure 3.20. The implementation would also likely be in the form of various procedures that implement the actions to be taken in response to the various events that can occur. In such event-based programming, the various procedures are called (invoked) either by other procedures in the protocol stack, or as the result of an interrupt. In the sender, these events would be (1) a call from the upper-layer entity to invoke rdt_send(), (2) a timer interrupt, and (3) a call from

the lower layer to invoke `rdt_rcv()` when a packet arrives. The programming exercises at the end of this chapter will give you a chance to actually implement these routines in a simulated, but realistic, network setting.

We note here that the GBN protocol incorporates almost all of the techniques that we will encounter when we study the reliable data transfer components of TCP in Section 3.5. These techniques include the use of sequence numbers, cumulative acknowledgments, checksums, and a timeout/retransmit operation.

3.4.4 Selective Repeat (SR)

The GBN protocol allows the sender to potentially “fill the pipeline” in Figure 3.17 with packets, thus avoiding the channel utilization problems we noted with stop-and-wait protocols. There are, however, scenarios in which GBN itself suffers 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 unnecessarily. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions. Imagine, in our message-dictation scenario, that if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated. The dictation would be slowed by all of the reiterated words.

As the name suggests, selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as needed, retransmission will require that the receiver individually acknowledge correctly received packets. A window size of N will again be used to limit the number

224 CHAPTER 3 • TRANSPORT LAYER

Sender Receiver

```

send pkt0
send pkt1
send pkt2
send pkt3
(wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
send pkt2
send pkt3
send pkt4
send pkt5
pkt2 timeout
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver

```

```

send ACK2
rcv pkt3, deliver
send ACK3
X
(loss)

```

Figure 3.22 Go-Back-N in operation

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 3.23 shows the SR sender's view of the sequence number space. Figure 3.24 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 (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer. Figure 3.25 itemizes the various actions taken by the SR receiver. Figure 3.26 shows an example of SR operation in the presence of lost packets. Note that in Figure 3.26, the receiver initially buffers packets 3, 4, and 5, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

It is important to note that in Step 2 in Figure 3.25, the receiver reacknowledges (rather than ignores) already received packets with certain sequence numbers below the current window base. You should convince yourself that this reacknowledgment is indeed needed. Given the sender and receiver sequence number spaces in Figure 3.23, for example, if there is no ACK for packet `send_base` propagating from the receiver to the sender, the sender will eventually retransmit packet `send_base`, even though it is clear (to us, not the sender!) that the receiver has already received that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward! This example illustrates an important aspect of SR protocols (and many other protocols as well). The sender and receiver will not always have an identical view of what has been received correctly and what has not.

For SR protocols, this means that the sender and receiver windows will not always coincide. The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers. Consider what could happen, for example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively. Now consider two scenarios. In the first scenario, shown in Figure 3.27(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 3.27(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, and 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 3.27, which has a figurative curtain between the sender and the receiver, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the sequence of messages it receives from the channel and sends into the channel. As far as it is concerned, the two scenarios in Figure 3.27 are identical. There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet.

Clearly, a window size that is 1 less than the size of the sequence number space won't work. But how small must the window size be? A problem at the end of the chapter asks you to show that the window size must be less than or equal to half the size of the sequence number space for SR protocols.

At the companion Web site, you will find an applet that animates the operation of the SR protocol. Try performing the same experiments that you did with the GBN applet. Do the results agree with what you expect?

This completes our discussion of reliable data transfer protocols. We've covered a lot of ground and introduced numerous mechanisms that together provide for reliable data transfer. Table 3.1 summarizes these mechanisms. Now that we have seen all of these mechanisms in operation and can see the "big picture," we encourage you to review this section again to see how these mechanisms were incrementally added to cover increasingly complex (and realistic) models of the channel connecting the sender and receiver, or to improve the performance of the protocols.

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

3.4 • PRINCIPLES OF RELIABLE DATA TRANSFER 229

230 CHAPTER 3 • TRANSPORT LAYER

Table 3.1 Summary of reliable data transfer mechanisms and their use

Mechanism Use, Comments

Checksum Used to detect bit errors in a transmitted packet.

Timer Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.

Sequence number Used for sequential numbering of packets of data flowing from sender to receiver.

Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.

Acknowledgment Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.

Negative acknowledgment Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the

packet that was not received correctly.

Window, pipelining The sender may be restricted to sending only packets with sequence numbers that

fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

for high-speed networks [RFC 1323]. [Sunshine 1978] describes a method for using sequence numbers such that reordering problems can be completely avoided

Awesome, this is the heart of transport layer theory. Let's turn **3.4** into a tight, exam-ready packet.

3.4 Principles of Reliable Data Transfer

We want to send data **reliably** over an **unreliable** channel (loss, corruption, reordering, delay).

To do that, almost all real protocols (including TCP) use the same building blocks:

1 Core Principles / Mechanisms

1. Error detection

- Use **checksum** bits to detect if a packet was corrupted in transit.
- If checksum fails → treat packet as **bad**.

2. Feedback

- Receiver sends control messages back to sender:
 - **ACK** (acknowledgment) = "I got it correctly."
 - **NAK** (negative acknowledgment) = "That packet was bad / missing."
- This creates a **closed-loop** system: sender learns what happened at receiver.

3. Retransmission

- If receiver says "bad" (NAK) **or** sender hears nothing for too long:
 - Sender **retransmits** the packet.
- This is the core idea behind **ARQ (Automatic Repeat reQuest)** protocols.

4. Sequence numbers

- Each packet gets a **sequence number**.
- Allows receiver to:

- Detect **duplicates**.
- Detect **missing** packets (gap in sequence).
- Deliver data **in order** to application.

5. Timers & timeouts

- Used to detect **loss** (data or ACK).
- If no ACK in time → **timeout** → **retransmit**.
- Timer must be wisely chosen (based on RTT estimate).

📌 **Golden line:**

Reliable data transfer = checksum + ACK/NAK + retransmission + sequence numbers + timers.

2 Stop-and-Wait Protocols (rdt1.0 → rdt3.0)

✓ rdt1.0 – Perfect channel

Assumptions:

- No errors, no loss, no reordering.

Protocol:

- Sender:
 - Gets data from above → makes packet → sends via unreliable channel (udt_send).
- Receiver:
 - Gets packet → extracts data → delivers to app.

No ACKs, no sequence numbers, nothing fancy.

✓ rdt2.0 – Channel with bit errors, but no loss

New problem:

- Bits may be corrupted → packet content or ACK/NAK could be wrong.

Solutions added:

1. **Checksum** to detect corruption.

2. **ACK / NAK** feedback:

- Receiver sends ACK if packet correct.
- Sends NAK if corrupted.

3. **Retransmission:**

- If NAK received → sender resends packet.

- Sender waits in “wait-for-ACK-or-NAK” state → this is **stop-and-wait**.

Problem:

- What if **ACK/NAK itself is corrupted?**
 - Sender can't tell if receiver got the packet or not.

✓ rdt2.1 / rdt2.2 – Fixing corrupted ACK/NAK with sequence numbers

New idea:

- **Add sequence numbers** (0 or 1) to data packets.
- Receiver:
 - Remembers last correctly received seq number.
 - If it gets:
 - New seq number → accept, deliver, send ACK(new).
 - Old seq number (duplicate) → discard, but ACK last correct one again.
- rdt2.2 removes explicit NAK:
 - Uses **duplicate ACKs** instead to signal a problem.

Now:

- Sender can safely retransmit when needed.
- Receiver can detect duplicates using seq numbers.

✓ rdt3.0 – Channel with bit errors + packet loss

Now packets (data or ACK) can be **lost**.

New mechanism:

- **Timer + timeout** on sender side:
 - Start timer whenever data packet is sent.
 - If ACK not received before timeout → retransmit packet.
- Still **stop-and-wait**, but with:
 - checksum + seq numbers + ACKs + timer.

Because seq numbers alternate between 0 and 1, rdt3.0 is also called the **alternating bit protocol**.

3 Performance Problem of Stop-and-Wait

Stop-and-wait is **correct** but potentially **super slow**.

Sender sends:

1 packet → waits for ACK → next packet → wait again → ...

Utilization:

$$U_{\text{sender}} = \frac{\text{transmission time}}{\text{RTT} + \text{transmission time}} = \frac{L/R}{RTT + L/R}$$

For:

- RTT = 30 ms (U.S. coast-to-coast)
- Rate R = 1 Gbps
- L = 1000 bytes = 8000 bits → L/R = 8 μs

$$U \approx \frac{0.008 \text{ ms}}{30.008 \text{ ms}} \approx 0.00027 \text{ (0.027%)}$$

So we're using **0.027%** of link capacity. Ouch.

 Fix: **allow multiple packets in flight** → **pipelining**.

4 Pipelining: Idea

Instead of stop-and-wait:

- Sender can send **multiple packets before waiting** for ACKs.
- These packets are “in flight” in the network → like a **pipeline**.

Consequences:

- Need **more sequence numbers** (one per in-flight packet).
 - Sender must **buffer** unACKed packets.
 - Receiver may need to **buffer out-of-order** packets (depends on protocol).
 - Leads to two main designs:
 1. **Go-Back-N (GBN)**
 2. **Selective Repeat (SR)**
-

5 Go-Back-N (GBN)

Sender View: Sliding Window

Variables:

- **base** = seq number of **oldest unACKed** packet.
- **nextseqnum** = seq number of **next packet to send**.
- Window size = **N**.

Four regions of sequence space:

1. $[0, \text{base}-1]$ → already sent & ACKed.
2. $[\text{base}, \text{nextseqnum}-1]$ → sent but not yet ACKed.
3. $[\text{nextseqnum}, \text{base}+\text{N}-1]$ → allowed to send immediately (window).
4. $\geq \text{base}+\text{N}$ → **not allowed** yet (outside window).

Window slides forward when ACKs arrive → hence **sliding window protocol**.

Sender Actions

1. `rdt_send(data):`

- If window not full ($\text{nextseqnum} < \text{base} + \text{N}$):
 - Make packet with $\text{seq} = \text{nextseqnum}$, send it.
 - If $\text{base} == \text{nextseqnum}$ → start timer.
 - Increment nextseqnum .
- Else: window full → either buffer data or make app retry later.

2. On $\text{ACK}(n)$ arrival:

- Cumulative ACK:
 - Means all packets $\leq n$ received.
- Set $\text{base} = n+1$.
- If $\text{base} == \text{nextseqnum}$ → no unACKed packets → stop timer.
- Else → restart timer for oldest unACKed packet.

3. On timeout:

- **Go back N:**
 - Retransmit **all packets** with seq in $[\text{base}, \text{nextseqnum}-1]$.
 - Restart timer.

Receiver Actions

- Maintains `expectedseqnum`.
- On packet with $\text{seq} = \text{expectedseqnum}$ and uncorrupted:
 - Deliver data up.
 - Send $\text{ACK}(\text{expectedseqnum})$.
 - $\text{expectedseqnum}++$.
- On packet out-of-order or corrupted:
 - Discard.
 - Re-send ACK for **last in-order packet**.

No buffering of out-of-order packets → simple receiver.

📌 Key characteristics of GBN:

- Single timer (for oldest unACKed packet).
 - **Cumulative ACKs.**
 - Receiver discards out-of-order packets.
 - On timeout, **retransmit everything from base onward.**
-

6 Selective Repeat (SR)

Problem with GBN:

- A **single packet loss** can force retransmission of **many** packets.
- Wasteful, especially if window and RTT are large.

Main Idea

Only retransmit the **specific packets** that were lost or corrupted.

So:

- Sender **keeps a window** of up to N unACKed packets.
- **Each packet** has its **own timer**.
- Receiver **buffers out-of-order packets**.

Receiver Behavior

- Accepts **correct** packets within its window, even if out of order.
- Buffers them.
- Sends **individual ACKs** for each correctly received packet.
- Once missing lower-numbered packets arrive, it can deliver data in order to app.

Sender Behavior

- Maintains window [`send_base`, `send_base + N - 1`].
- On data:
 - If seq in window → send, start individual timer.
- On ACK(k):
 - Mark k as received.
 - Slide window base forward if k == `send_base` and subsequent ones already ACKed.
- On timeout for packet k:
 - Retransmit only **packet k**.

Sequence Number Space Issue

Because sender and receiver windows can be **out of sync**, and sequence numbers are finite (modulo arithmetic):

- To avoid confusion between:

- A **new packet** with seq number x
- An **old duplicate** of x

- It can be shown:

Window size $\leq \frac{1}{2}$ of sequence number space is required for SR to work correctly.

7 Summary: GBN vs Selective Repeat

Feature	Go-Back-N (GBN)	Selective Repeat (SR)
ACK type	Cumulative ACKs	Individual ACKs per packet
Receiver out-of-order	✗ Discard	✓ Buffer
Timer per packet	✗ One timer for oldest unACKed	✓ One timer per unACKed packet
On timeout	Retransmit all unACKed packets	Retransmit only that specific packet
Receiver complexity	Simple	More complex (buffer, window)
Sender retransmission	Potentially many unnecessary	Minimal / targeted
Window limit vs seq space	No strict $\frac{1}{2}$ rule	Window size $\leq \frac{1}{2}$ of sequence space

8 One-Line Gist for Each Concept

- **Reliable data transfer** = checksum + sequence numbers + ACK/NAK + retransmission + timers.
 - **Stop-and-wait (rdt3.0)** = simple & reliable, but horribly underutilizes fast links.
 - **Pipelining** = multiple unACKed packets in flight to improve utilization.
 - **GBN** = sliding window with cumulative ACK; on loss, resend everything from the missing one onward.
 - **Selective Repeat** = sliding window with per-packet ACK + buffering; resend only what's missing.
-

If you want, next we can:

- Do a **small set of conceptual Qs** on GBN vs SR (like exam-style “which statement is true?”), or
- Move on to **3.5 (TCP)** and I'll map these abstract ideas directly onto how TCP actually implements them.



3.5: timeout and RTT, timeout and fast retransmit, flow control

3.5.3 Round-Trip Time Estimation and Timeout

TCP, like our rdt protocol in Section 3.4, uses a timeout/retransmit mechanism to recover from lost segments. Although this is conceptually simple, many subtle issues arise when we implement a timeout/retransmit mechanism in an actual protocol such as TCP. Perhaps the most obvious question is the length of the timeout.

Time Time

Host A Host B

User types

'C' Seq=42, ACK=79, data='C'

Seq=79, ACK=43, data='C'

Seq=43, ACK=80

Host ACKs

receipt of 'C',

echoes back 'C'

Host ACKs

receipt of

echoed 'C'

Figure 3.31 Sequence and acknowledgment numbers for a simple

Telnet application over TCP

3.5 • CONNECTION-ORIENTED TRANSPORT: TCP 239

intervals. Clearly, the timeout should be larger than the connection's round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent. But how much larger? How should the RTT be estimated in the first place? Should a timer be associated with each and every unacknowledged segment? So many questions! Our discussion in this section is based on the TCP work in [Jacobson 1988] and the current IETF recommendations for managing TCP timers [RFC 6298].

Estimating the Round-Trip Time

Let's begin our study of TCP timer management by considering how TCP estimates the round-trip time between sender and receiver. This is accomplished as follows.

The sample RTT, denoted SampleRTT, for a segment is the amount of time

between when the segment is sent (that is, passed to IP) and when an acknowledgement for the segment is received. Instead of measuring a SampleRTT for every

transmitted segment, most TCP implementations take only one SampleRTT measurement at a time. That is, at any point in time, the SampleRTT is being estimated

for only one of the transmitted but currently unacknowledged segments, leading to a new value of SampleRTT approximately once every RTT. Also, TCP never computes a SampleRTT for a segment that has been retransmitted; it only measures

SampleRTT for segments that have been transmitted once [Karn 1987]. (A problem at the end of the chapter asks you to consider why.)

Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given SampleRTT value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values. TCP maintains an average, called EstimatedRTT, of the SampleRTT values. Upon obtaining a new SampleRTT, TCP updates

EstimatedRTT according to the following formula:

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

The formula above is written in the form of a programming-language statement—the new value of EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT. The recommended value

of α is $\alpha = 0.125$ (that is, $1/8$) [RFC 6298], in which case the formula above becomes:

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

Note that EstimatedRTT is a weighted average of the SampleRTT values.

As discussed in a homework problem at the end of this chapter, this weighted average puts more weight on recent samples than on old samples. This is natural, as more recent samples better reflect the current congestion in the network. In statistics, such an average is called an exponential weighted moving average (EWMA).

The word “exponential” appears in EWMA because the weight of a given SampleRTT decays exponentially fast as the updates proceed. In the homework problems you will be asked to derive the exponential term in EstimatedRTT.

Figure 3.32 shows the SampleRTT values and EstimatedRTT for a value of $\alpha = 1/8$ for a TCP connection between gaia.cs.umass.edu (in Amherst, Massachusetts) to fantasia.eurecom.fr (in the south of France). Clearly, the variations in the SampleRTT are smoothed out in the computation of the EstimatedRTT.

In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. [RFC 6298] defines the RTT variation, DevRTT, as an estimate of how much SampleRTT typically deviates from EstimatedRTT:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Note that DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT. If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of β is 0.25 Setting and Managing the Retransmission Timeout Interval

Given values of EstimatedRTT and DevRTT, what value should be used for TCP’s timeout interval? Clearly, the interval should be greater than or equal to EstimatedRTT, or unnecessary retransmissions would be sent. But the timeout interval should not be too much larger than EstimatedRTT; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays. It is therefore desirable to set the timeout equal to the EstimatedRTT plus some margin. The margin should be large when there is a lot of fluctuation in the SampleRTT values; it should be small when there is little fluctuation. The value of DevRTT should thus come into play here. All of these considerations are taken into account in TCP’s method for determining the retransmission timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

An initial TimeoutInterval value of 1 second is recommended [RFC 6298].

Also, when a timeout occurs, the value of TimeoutInterval is doubled to avoid a premature timeout occurring for a subsequent segment that will soon be acknowledged. However, as soon as a segment is received and EstimatedRTT is updated, the TimeoutInterval is again computed using the formula above

Fast Retransmit

One of the problems with timeout-triggered retransmissions is that the timeout period can be relatively long. When a segment is lost, this long timeout period forces the sender to delay resending the lost packet, thereby increasing the end-to-end delay. Fortunately, the sender can often detect packet loss well before the timeout event occurs by noting so-called duplicate ACKs. A duplicate ACK is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment. 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 3.2 summarizes the TCP receiver’s ACK generation policy [RFC 5681]. When a TCP receiver

receives a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream—that is, a missing segment. This gap could be the result of lost or reordered segments within the network.

3.5.5 Flow Control

Recall that the hosts on each side of a TCP connection 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 provides a flow-control service to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading. As noted earlier, a TCP sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as congestion control, a topic we will explore in detail in Sections 3.6 and 3.7. Even though the actions taken by flow and congestion control are similar (the throttling of the sender), they are obviously taken for very different reasons. Unfortunately, many authors use the terms interchangeably, and the savvy reader would be wise to distinguish between them. Let's now discuss how TCP provides its flow-control service. In order to see the forest for the trees, we suppose throughout this section that the TCP implementation is such that the TCP receiver discards out-of-order segments.

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 of how much free buffer space is available at the receiver. Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window. 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

250 CHAPTER 3 • TRANSPORT LAYER

3.5 • CONNECTION-ORIENTED TRANSPORT: TCP 251

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

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

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

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

Because the spare room changes with time, $rwnd$ is dynamic. The variable $rwnd$ is illustrated in Figure 3.38.

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

Host A in turn keeps track of two variables, $LastByteSent$ and $LastByteAcked$, which have obvious meanings. Note that the difference between these

two variables, LastByteSent – LastByteAcked, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of rwnd, Host A is assured that it is not overflowing the receive buffer at Host B. Thus, Host A makes sure throughout the connection's life that

LastByteSent – LastByteAcked \leq rwnd

Application

process

Data

from IP/TCP data

in buffer

rwnd

RcvBuffer

Spare room

Figure 3.38  The receive window (rwnd) and the receive buffer (RcvBuffer)

(RcvBuffer)

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

The online site at <http://www.awl.com/kurose-ross> for this book provides an interactive Java applet that illustrates the operation of the TCP receive window. Having described TCP's flow-control service, we briefly mention here that UDP does not provide flow control. To understand the issue, consider sending a series of UDP segments from a process on Host A to a process on Host B. For a typical UDP implementation, UDP will append the segments in a finite-sized buffer that "precedes" the corresponding socket (that is, the door to the process). The process reads one entire segment at a time from the buffer. If the process does not read the segments fast enough from the buffer, the buffer will overflow and segments will get dropped

Nice chunk—this is exactly the part where TCP starts to feel “alive”: it measures the network, adapts, and avoids drowning the receiver.

I'll split this into:

1.  RTT estimation & timeout interval
2.  Fast retransmit using duplicate ACKs
3.  Flow control (receive window)

1 RTT Estimation & Timeout Interval

TCP uses **timeout + retransmission** (like rdt3.0) but needs a **good timeout value**:

- Too small → **spurious retransmissions**
- Too large → **slow recovery** from real loss

So TCP **continuously estimates RTT** and its **variability**.

(a) Measuring SampleRTT

- **SampleRTT** = time from **sending a segment** (to IP) until its **ACK is received**.
 - TCP usually:
 - Measures **only one segment at a time** (not every segment).
 - **Does NOT** take SampleRTT for segments that were **retransmitted** (Karn's algorithm) – because you can't know which transmission the ACK corresponds to.
-

(b) EstimatedRTT: Smoothed RTT

TCP keeps a **smoothed average** of SampleRTT:

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

- Recommended: $\alpha = 0.125 (= 1/8)$
- This is an **Exponential Weighted Moving Average (EWMA)**:
 - Recent samples get **more weight**.
 - Old samples decay **exponentially**.

Intuition:

- If network slows down → SampleRTT increases → EstimatedRTT slowly moves upward.
 - If network speeds up → it moves downward.
-

(c) DevRTT: RTT Variability

TCP also tracks **how much RTT fluctuates**:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

- Recommended: $\beta = 0.25$
 - If SampleRTT is very noisy → DevRTT big
 - If SampleRTT is stable → DevRTT small
-

(d) TimeoutInterval

Finally, TCP sets the **retransmission timeout** as:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

Key behavior:

- Start with **initial** TimeoutInterval \approx 1 second (per RFC 6298).
- If a **timeout occurs**:
 - TCP does **exponential backoff**:
 $\text{TimeoutInterval} \leftarrow 2 \times \text{TimeoutInterval}$
 - This avoids repeatedly timing out too aggressively when network is congested.
- When a **new, successful ACK** arrives:
 - Recompute EstimatedRTT, DevRTT
 - Recompute TimeoutInterval with that formula.

Golden sentence:

TCP sets $\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$, and doubles it after timeouts, to balance quick recovery with stability.

2 Fast Retransmit (Using Duplicate ACKs)

Problem with timeout-only retransmission:

- Even if segment is clearly lost, TCP might wait **until timeout expires**, which can be relatively long
 \rightarrow bad delay.

Solution: **Fast retransmit** using **duplicate ACKs**.

(a) Why do duplicate ACKs occur?

Receiver sends an ACK for **the highest in-order byte** it has received.

Example:

- Sender sends segments with seq: 100, 200, 300, 400
- Segment **200 is lost**, but 300 and 400 arrive.

Receiver behavior:

- Receives **100** \rightarrow sends ACK=200 (next expected)
- **200 is lost**
- Receives 300:
 - It sees a **gap** (missing 200), so it **cannot advance** in-order ACK
 - It **re-ACKs 200** (duplicate ACK)
- Receives 400:

- Still missing 200 → again ACK=200
→ Another **duplicate ACK**

So sender starts receiving **multiple ACKs with the same ACK number = duplicate ACKs**.

(b) Fast Retransmit Rule

TCP sender logic:

- When it receives **3 duplicate ACKs** (i.e., 4 ACKs with same ACK number total):
 - It **assumes** the corresponding segment is **lost (not just delayed)**.
 - It **retransmits that segment immediately, before the timer expires.**

This is **fast retransmit**.

Key points:

- It speeds up loss recovery.
- It relies on the fact that:
 - Multiple segments got through (we see their ACKs as duplicates).
 - So the network path still has capacity; the one missing packet is very likely lost.

(There's also **fast recovery** tied to congestion control, but that's Section 3.7; here we just focus on the retransmit logic.)

Golden sentence:

If TCP sees 3 duplicate ACKs for the same byte, it performs fast retransmit of the missing segment without waiting for the timeout.

3 Flow Control (Not the Same as Congestion Control!)

Flow control = don't **overrun the receiver's buffer**.

Congestion control = don't **overwhelm the network**.

Here we only care about **flow control**.

(a) Receive Buffer & Variables

At Host B (receiver):

- It has a finite **receive buffer** of size **RcvBuffer**.
- As data arrives in-order and uncorrupted:
 - TCP puts it into the receive buffer.
 - The **application** (e.g., `read()` on socket) pulls data out when it wants → may be slow.

We track:

- **LastByteRead:** last byte **already read** by B's application.
- **LastByteRcvd:** last byte **received from network** and placed in buffer.

Buffer occupancy = $\text{LastByteRcvd} - \text{LastByteRead}$

Cannot exceed buffer size:

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$

(b) Receive Window (rwnd)

Define:

$$\text{rwnd} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$$

This is how much **spare room** is left in buffer.

- **rwnd** changes dynamically as:
 - More data arrives → rwnd **decreases**
 - App reads data → rwnd **increases**

Host B **advertises rwnd** to Host A in every TCP header (receive window field).

(c) Sender Side: Using rwnd

At Host A (sender), it tracks:

- **LastByteSent:** last byte sent into network.
- **LastByteAcked:** last byte that has been **ACKed** by B.

So the amount of **in-flight unACKed data** is:

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

To avoid overflowing B's buffer, TCP enforces:

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

So A **never injects more unACKed data** than B has advertised as free space.

This is TCP **flow control**: sender's sending rate is matched to **receiver's read rate**, not to network capacity.

(d) Zero Window Problem & Keep-Alive Probes

Tricky case:

1. B's buffer becomes **full** → **rwnd = 0**.
2. B sends a segment to A with **rwnd=0**.
3. B's app slowly starts reading data → buffer **frees up**, but:

- B has no new data/ACKs to send,
- So it doesn't automatically send an updated rwnd.

Result:

A thinks **rwnd = 0** forever and stops sending. Deadlock.

Solution (TCP spec):

- When A sees **rwnd = 0**:
 - It **periodically sends 1-byte "probe" segments**.
- B must ACK these probes and:
 - Include the **current rwnd** in ACK.
- Once rwnd > 0, A resumes normal sending.

Golden sentence:

TCP flow control uses the advertised receive window (rwnd) so the sender's unACKed data never exceeds the receiver's free buffer space.

If you want, next step we can:

- Put these together in a **single timeline diagram** (RTT estimation + timeout + fast retransmit + rwnd), **or**
- Move to **3.6 / 3.7: Congestion control** and connect this with cwnd and TCP's "two windows" (rwnd vs cwnd).



3.6: congestion control principles and approaches

3.6 Principles of Congestion Control

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

In this section, we consider the problem of congestion control in a general context, seeking to understand why congestion is a bad thing, how network congestion is manifested in the performance received by upper-layer applications, and various approaches that can be taken to avoid, or react to, network congestion. This more general study of congestion control is appropriate since, as with reliable data transfer, it is high on our "top-ten" list of fundamentally important problems in networking. We conclude this section with a discussion of congestion control in the

available bit-rate (ABR) service in asynchronous transfer mode (ATM) networks. The following section contains a detailed study of TCP's congestion control algorithm.

3.6.1 The Causes and the Costs of Congestion

Let's begin our general study of congestion control by examining three increasingly complex scenarios in which congestion occurs. In each case, we'll look at why congestion occurs in the first place and at the cost of congestion (in terms of resources not fully utilized and poor performance received by the end systems). We'll not (yet) focus on how to react to, or avoid, congestion but rather focus on the simpler issue of understanding what happens as hosts increase their transmission rate and the network becomes congested.

Scenario 1: Two Senders, a Router with Infinite Buffers

We begin by considering perhaps the simplest congestion scenario possible: Two hosts (A and B) each have a connection that shares a single hop between source and destination, as shown in Figure 3.43.

Let's assume that the application in Host A is sending data into the connection (for example, passing data to the transport-level protocol via a socket) at an average rate of λ_{in} bytes/sec. These data are original in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a

260 CHAPTER 3 • TRANSPORT LAYER

simple one. Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed. Ignoring the additional overhead due to adding transport- and lower-layer header information, the rate at which Host A offers traffic to the router in this first scenario is thus λ_{out} in bytes/sec. Host B operates in a similar manner, and we assume for simplicity that it too is sending at a rate of λ_{out} in bytes/sec. Packets from Hosts A and B pass through a router and over a shared outgoing link of capacity R. The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity. In this first scenario, we assume that the router has an infinite amount of buffer space.

Figure 3.44 plots the performance of Host A's connection under this first scenario. The left graph plots the per-connection throughput (number of bytes per second at the receiver) as a function of the connection-sending rate. For a sending rate between 0 and $R/2$, the throughput at the receiver equals the sender's sending rate—everything sent by the sender is received at the receiver with a finite delay. When the sending rate is above $R/2$, however, the throughput is only $R/2$. This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds $R/2$. No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than $R/2$.

Achieving a per-connection throughput of $R/2$ might actually appear to be a good thing, because the link is fully utilized in delivering packets to their destinations. The right-hand graph in Figure 3.44, however, shows the consequence of operating near link capacity. As the sending rate approaches $R/2$ (from the left), the average delay becomes larger and larger. When the sending rate exceeds $R/2$, the Host B

Unlimited shared output link buffers

λ_{in} : original data

Host A Host C Host D

λ_{out}

Figure 3.43 Congestion scenario 1: Two connections sharing a single

hop with infinite buffers

3.6 • PRINCIPLES OF CONGESTION CONTROL 261

average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite (assuming that the connections operate at these sending rates for an infinite period of time and there is an infinite amount of buffering available). Thus, while operating at an aggregate throughput of near R may be ideal from a throughput standpoint, it is far from ideal from a delay standpoint. Even in this (extremely) idealized scenario, we've already found one cost of a congested network—large queuing delays are experienced as the packet arrival rate nears the link capacity.

Scenario 2: Two Senders and a Router with Finite Buffers

Let us now slightly modify scenario 1 in the following two ways (see Figure 3.45).

First, the amount of router buffering is assumed to be finite. A consequence of this real-world assumption is that packets will be dropped when arriving to an already full buffer.

Second, we assume that each connection is reliable. If a packet containing a transport-level segment is dropped at the router, the sender will eventually retransmit it. Because packets can be retransmitted, we must now be more careful with our use of the term sending rate. Specifically, let us again denote the rate at which the application sends original data into the socket by λ_{in} in bytes/sec. The rate at which the transport layer sends segments (containing original data and retransmitted data) into the network will be denoted λ_{out} in bytes/sec. λ_{out} is sometimes referred to as the offered load to the network.

The performance realized under scenario 2 will now depend strongly on how retransmission is performed. First, consider the unrealistic case that Host A is able to somehow (magically!) determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free. In this case, no loss would occur,

$R/2$

$R/2$

Delay

$R/2 \lambda_{in} \lambda_{in}$

λ_{out}

a. b.

Figure 3.44 Congestion scenario 1: Throughput and delay as a function of host sending rate

262 CHAPTER 3 • TRANSPORT LAYER

λ_{in} would be equal to λ_{out} , and the throughput of the connection would be equal to λ_{in} . This case is shown in Figure 3.46(a). From a throughput standpoint, performance is ideal—everything that is sent is received. Note that the average host sending rate cannot exceed $R/2$ under this scenario, since packet loss is assumed never to occur.

Consider next the slightly more realistic case that the sender retransmits only when a packet is known for certain to be lost. (Again, this assumption is a bit of a stretch. However, it is possible that the sending host might set its timeout large enough to be virtually assured that a packet that has not been acknowledged has been lost.) In this case, the performance might look something like that shown in Figure 3.46(b). To appreciate what is happening here, consider the case that the offered load, λ_{out} (the rate of original data transmission plus retransmissions), equals $R/2$. According to Figure 3.46(b), at this value of the offered load, the rate at which data are delivered to the receiver application is $R/3$. Thus, out of the $0.5R$ units of data transmitted, $0.333R$ bytes/sec (on average) are original data and $0.166R$ bytes/sec (on average) are retransmitted data. We see here another cost of a congested network—the

sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.

Finally, let us consider the case that the sender may time out prematurely and retransmit a packet that has been delayed in the queue but not yet lost. In this case, both the original data packet and the retransmission may reach the receiver. Of course, the receiver needs but one copy of this packet and will discard the retransmission. In this case, the work done by the router in forwarding the retransmitted

copy of the original packet was wasted, as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead. Here then is yet another cost of a congested network—unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet. Figure 3.46 (c) shows the throughput versus offered load when each packet is assumed to be forwarded (on average) twice by the router. Since each packet is forwarded twice, the throughput will have an asymptotic value of $R/4$ as the offered load approaches $R/2$.

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

In our final congestion scenario, four hosts transmit packets, each over overlapping two-hop paths, as shown in Figure 3.47. We again assume that each host uses a timeout/retransmission mechanism to implement a reliable data transfer service, that all hosts have the same value of λ_{in} , and that all router links have capacity R bytes/sec.

Let's consider the connection from Host A to Host C, passing through routers R1 and R2. The A-C connection shares router R1 with the D-B connection and shares router R2 with the B-D connection. For extremely small values of λ_{in} , buffer overflows are rare (as in congestion scenarios 1 and 2), and the throughput approximately equals the offered load. For slightly larger values of λ_{in} , the corresponding throughput is also larger, since more original data is being transmitted into the

$R/2$

$R/2$ $R/2$

λ_{out}

a. b.

$R/2$

λ_{out}

$R/3$

$R/2$

$R/2$

λ_{out}

$R/4$

c.

λ'

λ'_{in}

λ'_{in}

λ'_{in}

Figure 3.46 Scenario 2 performance with finite buffers

264 CHAPTER 3 • TRANSPORT LAYER

network and delivered to the destination, and overflows are still rare. Thus, for small values of λ_{in} , an increase in λ_{in} results in an increase in λ_{out} .

Having considered the case of extremely low traffic, let's next examine the case that λ_{in} (and hence λ_{out}) is extremely large. Consider router R2. The A-C

traffic arriving to router R2 (which arrives at R2 after being forwarded from R1) can have an arrival rate at R2 that is at most R , the capacity of the link from R1 to R2, regardless of the value of λ_{in} . If λ_{in} is extremely large for all connections (including the B-D connection), then the arrival rate of B-D traffic at R2 can be much larger than that of the A-C traffic. Because the A-C and B-D traffic must compete at router R2 for the limited amount of buffer space, the amount of A-C traffic that successfully gets through R2 (that is, is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B-D gets larger and larger. In the limit, as the offered load approaches infinity, an empty buffer at R2

Host A Host B

R1

R4 R2

R3

Host D Host C Finite shared output

link buffers

λ_{in} : original data

λ'

in : original
data, plus
retransmitted

data

λ_{out}

Figure 3.47 Four senders, routers with finite buffers, and multihop paths

3.6 • PRINCIPLES OF CONGESTION CONTROL 265

is immediately filled by a B-D packet, and the throughput of the A-C connection at R2 goes to zero. This, in turn, implies that the A-C end-to-end throughput goes to zero in the limit of heavy traffic. These considerations give rise to the offered load versus throughput tradeoff shown in Figure 3.48.

The reason for the eventual decrease in throughput with increasing offered load is evident when one considers the amount of wasted work done by the network. In the high-traffic scenario outlined above, whenever a packet is dropped at a second-hop router, the work done by the first-hop router in forwarding a packet to the second-hop router ends up being “wasted.” The network would have been equally well off (more accurately, equally bad off) if the first router had simply discarded that packet and remained idle. More to the point, the transmission capacity used at the first router to forward the packet to the second router could have been much more profitably used to transmit a different packet. (For example, when selecting a packet for transmission, it might be better for a router to give priority to packets that have already traversed some number of upstream routers.) So here we see yet another cost of dropping a packet due to congestion—when a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.

3.6.2 Approaches to Congestion Control

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

R/2

λ_{out}

λ'

in

Figure 3.48 Scenario 3 performance with finite buffers and multihop paths

266 CHAPTER 3 • TRANSPORT LAYER

At the broadest level, we can distinguish among congestion-control approaches by whether the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

- End-to-end congestion control. In an end-to-end approach to congestion control, the network layer provides no explicit support to the transport layer for congestion control purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). We will see in Section 3.7 that TCP must necessarily take this end-to-end approach toward congestion control, since the IP layer provides no feedback to the end systems regarding network congestion. TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly. We will also see a more recent proposal for TCP congestion control that uses increasing round-trip delay values as indicators of increased network congestion.
- Network-assisted congestion control. With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. This approach was taken in the early IBM SNA [Schwartz 1982] and DEC DECnet [Jain 1989; Ramakrishnan 1990] architectures, was recently proposed for TCP/IP networks [Floyd TCP 1994; RFC 3168], and is used in ATM available bit-rate (ABR) congestion control as well, as discussed below. More sophisticated network feedback is also possible. For example, one form of ATM ABR congestion control that we will study shortly allows a router to inform the sender explicitly of the transmission rate it (the router) can support on an outgoing link. The XCP protocol [Katabi 2002] provides router-computed feedback to each source, carried in the packet header, regarding how that source should increase or decrease its transmission rate. For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in Figure 3.49. Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a choke packet (essentially saying, "I'm congested!"). The second form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. Note that this latter form of notification takes at least a full round-trip time.

3.6.3 Network-Assisted Congestion-Control Example:

ATM ABR Congestion Control

We conclude this section with a brief case study of the congestion-control algorithm in ATM ABR—a protocol that takes a network-assisted approach toward congestion control. We stress that our goal here is not to describe aspects of the ATM architecture in great detail, but rather to illustrate a protocol that takes a markedly different approach toward congestion control from that of the Internet's TCP protocol. Indeed, we only present below those few aspects of the ATM architecture that are needed to understand ABR congestion control.

Fundamentally ATM takes a virtual-circuit (VC) oriented approach toward packet switching. Recall from our discussion in Chapter 1, this means that each switch on the source-to-destination path will maintain state about the source-to-destination VC.

This per-VC state allows a switch to track the behavior of individual senders (e.g., tracking their average transmission rate) and to take source-specific congestion-control actions (such as explicitly signaling to the sender to reduce its rate when the switch becomes congested). This per-VC state at network switches makes ATM ideally suited to perform network-assisted congestion control. ABR has been designed as an elastic data transfer service in a manner reminiscent of TCP. When the network is underloaded, ABR service should be able to take advantage of the spare available bandwidth; when the network is congested, ABR service should throttle its transmission rate to some predetermined minimum transmission rate. A detailed tutorial on ATM ABR congestion control and traffic management is provided in [Jain 1996].

Figure 3.50 shows the framework for ATM ABR congestion control. In our discussion we adopt ATM terminology (for example, using the term switch rather than router, and the term cell rather than packet). With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches. Interspersed with the data cells are resource-management cells

3.6 • PRINCIPLES OF CONGESTION CONTROL 267

Host A

Network feedback via receiver

Direct network
feedback

Host B

Figure 3.49 Two feedback pathways for network-indicated congestion information

268 CHAPTER 3 • TRANSPORT LAYER

(RM cells); these RM cells can be used to convey congestion-related information among the hosts and switches. When an RM cell arrives at a destination, it will be turned around and sent back to the sender (possibly after the destination has modified the contents of the RM cell). It is also possible for a switch to generate an RM cell itself and send this RM cell directly to a source. RM cells can thus be used to provide both direct network feedback and network feedback via the receiver, as shown in Figure 3.50.

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

- EFCI bit. Each data cell contains an explicit forward congestion indication (EFCI) bit. A congested network switch can set the EFCI bit in a data cell to 1 to signal congestion to the destination host. The destination must check the EFCI bit in all received data cells. When an RM cell arrives at the destination, if the most recently received data cell had the EFCI bit set to 1, then the destination sets the congestion indication bit (the CI bit) of the RM cell to 1 and sends the RM cell back to the sender. Using the EFCI in data cells and the CI bit in RM cells, a sender can thus be notified about congestion at a network switch.
- CI and NI bits. As noted above, sender-to-receiver RM cells are interspersed with data cells. The rate of RM cell interspersion is a tunable parameter, with the default value being one RM cell every 32 data cells. These RM cells have a congestion indication (CI) bit and a no increase (NI) bit that can be set by a Source Destination Switch

Key:

RM cells Data cells

Figure 3.50 Congestion-control framework for ATM ABR service
 congested network switch. Specifically, a switch can set the NI bit in a passing RM cell to 1 under mild congestion and can set the CI bit to 1 under severe congestion conditions. When a destination host receives an RM cell, it will send the RM cell back to the sender with its CI and NI bits intact (except that CI may be set to 1 by the destination as a result of the EFCI mechanism described above).

- ER setting. Each RM cell also contains a 2-byte explicit rate (ER) field. A congested switch may lower the value contained in the ER field in a passing RM cell. In this manner, the ER field will be set to the minimum supportable rate of all switches on the source-to-destination path.

An ATM ABR source adjusts the rate at which it can send cells as a function of the CI, NI, and ER values in a returned RM cell. The rules for making this rate adjustment are rather complicated and a bit tedious. The interested reader is referred to [Jain 1996] for details

Nice, this is the “why is TCP even throttling me?” section. Let’s turn 3.6 into something you can recall under exam pressure.

I’ll break it into:

1. What congestion actually *is* and why it’s costly
 2. The three scenarios (1–3) and what each teaches
 3. Two big approaches: end-to-end vs network-assisted
 4. Example: ATM ABR (just enough to recognize it)
-

1 What is congestion control really about?

- **Reliability mechanisms** (retransmissions) fix the *symptom* (lost segments).
- **Congestion control** tries to fix the *cause*:

Too many senders injecting data too fast → router buffers overflow → delay & loss explode.

So the central job of congestion control:

Make senders back off when the network is overloaded.

2 Three congestion scenarios – causes & costs

Scenario 1: 2 senders, 1 router, infinite buffers

- Two hosts A and B send to C and D through a single shared link of rate R .

- Both send at rate λ_{in} (original data only, no retransmissions, no reliability).
- Router has **infinite buffer** → never drops packets, just queues them.

Throughput per connection:

- If $0 \leq \lambda_{in} \leq R/2$:
 - Each gets throughput = λ_{in} (what you send, you receive).
- If $\lambda_{in} > R/2$:
 - Each gets at most **R/2** (since total cannot exceed R).

Delay:

- As $\lambda_{in} \rightarrow R/2$ (from below), the **average queue grows** → **delay** $\rightarrow \infty$ (theoretically).
- So near full utilization, delay blows up.

👉 Cost #1 of congestion:

Even with no loss, operating near capacity ⇒ **massive queuing delay**.

Scenario 2: 2 senders, 1 router, finite buffer, reliable transport

Now modify:

1. Router buffer is **finite** → packets **can be dropped**.
2. Transport layer has **reliability** (timeout + retransmission).

We distinguish:

- λ_{in} : rate of *original* data from app into socket.
- λ'_{in} : rate of segments injected into network
(original + **retransmissions**) = **offered load**.

Now performance depends on **how retransmission is done**.

Case (a): “Magical” sender – sends only if buffer is free

- Sender somehow knows if router has space and sends only then.
- No loss, no retransmissions → $\lambda_{in} = \lambda'_{in} = \lambda_{out}$.
- Throughput increases until $R/2$ (like earlier), but **no drops**.

This is ideal but unrealistic.

Case (b): Retransmit only when a packet is truly lost

- Sender has a **very conservative timeout**:
only retransmits when sure the packet is lost.
- Example: at offered load $\lambda'_{in} = R/2$:

- Maybe throughput is $\approx R/3$:
 - $R/3$ is original delivered,
 - $R/6$ is retransmissions.

👉 Cost #2 of congestion:

We start wasting capacity on **retransmissions due to losses** from full buffers.

Case (c): Premature timeouts (retransmit even when not lost)

- Packets are just delayed in the queue, but timeout fires **too early**.
- Sender retransmits; both **original and retransmission** may reach receiver.
- Receiver discards the duplicate → router forwarded it **for nothing**.

Result: router is using bandwidth to send **unneeded copies**, so effective throughput drops even more (curve asymptotes to $R/4$ in the book's example).

👉 Cost #3 of congestion:

Unneeded retransmissions due to long delays + timeouts waste bandwidth.

Scenario 3: 4 senders, multihop paths, finite buffers

Setup:

- Four hosts (A, B, C, D) with two-hop paths that **overlap** ($A \rightarrow C$, $B \rightarrow D$, etc.).
- Links all have rate **R**, router buffers finite, reliability via retransmission.

For small λ_{in} :

- Similar story: low load → **throughput ≈ offered load**.

For very large λ_{in} (everyone blasting):

- Consider $A \rightarrow C$ traffic going via $R_1 \rightarrow R_2$.
- The link $R_1 \rightarrow R_2$ has max rate R ; $B \rightarrow D$ may hog most of the buffer at R_2 .
- As load $\rightarrow \infty$, **A → C throughput can go to 0**, because every free buffer slot at R_2 is grabbed by $B \rightarrow D$.

Think about a packet dropped at some **second-hop** router:

- Work done by the **first-hop** router to move that packet was **wasted**.
- If the packet is dropped after n hops, all previous hops' transmission effort is wasted.

👉 Cost #4 of congestion in multihop networks:

When packets are dropped **downstream**, all **upstream transmission work is wasted**, and at very high load some flows' throughput can actually go **down to zero**.

Overall throughput vs offered load curve in Scenario 3:

- Starts low, rises, peaks, then **falls** when the network is "melting down".

3 Congestion control approaches (big picture)

Two main philosophies:

(a) End-to-end congestion control

- **Network gives no explicit signal.**
No congestion bit from routers, no built-in feedback.
- Sender must **infer congestion** from what it sees:
 - **Packet loss** (timeout, triple duplicate ACKs).
 - **Increasing RTT** (longer delays).

TCP is the classic example:

- IP gives **no congestion feedback**, so TCP:
 - Treats **loss** as congestion,
 - Reduces its sending window when it sees loss,
 - Grows its window cautiously when no loss is seen.

This is why TCP congestion control is called **end-to-end**.

(b) Network-assisted congestion control

- Routers **explicitly participate** in signaling congestion.
- Two main ways to get feedback back to sender (see Fig. 3.49):
 1. **Direct feedback:**
router sends a special **choke packet** directly to the sender ("I'm congested!").
 2. **Marking packets:**
router sets a congestion bit in packets going forward;
receiver sees this and later tells the sender in its ACKs.

Examples:

- Old IBM SNA, DECnet
 - **Explicit Congestion Notification (ECN)** in IP/TCP (RFC 3168)
 - **ATM ABR** (next subsection)
 - Research protocols like **XCP**: routers compute how each sender should adjust its rate and write it into packet headers.
-

4 Example of network-assisted control: ATM ABR

You don't need ATM details deeply; just know roughly:

- ATM uses **virtual circuits (VCs)**:
each switch keeps **per-VC state** → can track each source's behavior.
- ABR = **Available Bit Rate** service: an *elastic* data service like TCP.

Goals:

- If network is **lightly loaded** → let ABR sources go faster (use spare bandwidth).
- If **congested** → force ABR sources down to at least some minimum rate.

Mechanism:

- Data sent in fixed-size **cells**.
- Periodically interspersed **RM (Resource Management) cells** carry congestion info.

Key bits/fields:

1. **EFCI bit** (Explicit Forward Congestion Indication) in *data cells*:

- Congested switch sets EFCI=1.
- Receiver sees EFCI=1 and, when sending back an RM cell, sets congestion indication bit **CI=1** in RM cell.

2. **CI and NI bits** in *RM cells*:

- **CI** = Congestion Indication (serious congestion).
- **NI** = No Increase (mild congestion: "don't increase your rate").
- Switches modify CI/NI in passing RM cells.

3. **ER field (Explicit Rate)** in RM cells:

- 2-byte field specifying an explicit allowed rate.
- Each switch can **lower ER** to the maximum rate it can support.
- By the time RM cell returns to the source, ER reflects the **minimum allowed rate along the path**.

Sender behavior:

- Uses CI, NI, and ER values from returned RM cells to **adjust its sending rate**.
- This is a **rate-based** scheme (not window-based like TCP).

What to remember:

- ABR is **network-assisted, rate-based**, and uses **RM cells + EFCI + CI/NI + ER** to signal congestion and allowable sending rates.

Quick mental summary for 3.6

- **Congestion costs:**
 1. Huge delays near capacity

2. Drops → retransmissions
3. Premature timeouts → *unnecessary* retransmissions
4. Multihop: wasted work on upstream links & throughput can **drop** at very high load

- **Approaches:**

- **End-to-end** (TCP): infer from loss/delay.
- **Network-assisted**: routers explicitly signal (choke packets, ECN, RM cells).
- **ATM ABR**: per-VC, uses RM cells and explicit rate feedback.

If you want, next we can jump to **3.7 TCP congestion control** and tie these principles directly to **cwnd**, AIMD, slow start, etc.



3.7: TCP congestion control: principles of congestion estimation (lost segment, acknowledgment, probing),

slow start, congestion avoidance, fast recovery. TCP Reno

3.7 TCP Congestion Control

In this section we return to our study of TCP. As we learned in Section 3.5, TCP provides a reliable transport service between two processes running on different hosts.

Another key component of TCP is its congestion-control mechanism. As indicated in the previous section, TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion.

The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate. But this approach raises three questions. First, how does a TCP sender limit the rate at which it sends traffic into its connection? Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination? And third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Let's first examine how a TCP sender limits the rate at which it sends traffic into its connection. In Section 3.5 we saw that each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, rwnd, and so on). The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the congestion window. The congestion window, denoted cwnd, imposes a constraint on the rate at which a TCP sender can send traffic

3.7 • TCP CONGESTION CONTROL 269

270 CHAPTER 3 • TRANSPORT LAYER

into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$$

In order to focus on congestion control (as opposed to flow control), let us henceforth assume that the TCP receive buffer is so large that the receive-window constraint can be ignored; thus,

the amount of unacknowledged data at the sender is solely limited by cwnd. We will also assume that the sender always has data to send, i.e., that all segments in the congestion window are sent.

The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate. To see this, consider a connection for which loss and packet transmission delays are negligible. Then, roughly, at the beginning of every RTT, the constraint permits the sender to send cwnd bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data. Thus the sender's send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

Let's next consider how a TCP sender perceives that there is congestion on the path between itself and the destination. Let us define a "loss event" at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. (Recall our discussion in Section 3.5.4 of the timeout event in Figure 3.33 and the subsequent modification to include fast retransmit on receipt of three duplicate ACKs.) When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped. The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

Having considered how congestion is detected, let's next consider the more optimistic case when the network is congestion-free, that is, when a loss event doesn't occur. In this case, acknowledgments for previously unacknowledged segments will be received at the TCP sender. As we'll see, TCP will take the arrival of these acknowledgments as an indication that all is well—that segments being transmitted into the network are being successfully delivered to the destination—and will use acknowledgments to increase its congestion window size (and hence its transmission rate). Note that if acknowledgments arrive at a relatively slow rate (e.g., if the end-end path has high delay or contains a low-bandwidth link), then the congestion window will be increased at a relatively slow rate. On the other hand, if acknowledgments arrive at a high rate, then the congestion window will be increased more quickly. Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be self-clocking.

Given the mechanism of adjusting the value of cwnd to control the sending rate, the critical question remains: How should a TCP sender determine the rate at which it should send? If TCP senders collectively send too fast, they can congest the network, leading to the type of congestion collapse that we saw in Figure 3.48. Indeed, the version of TCP that we'll study shortly was developed in response to observed Internet congestion collapse [Jacobson 1988] under earlier versions of TCP. However, if TCP senders are too cautious and send too slowly, they could under utilize the bandwidth in the network; that is, the TCP senders could send at a higher rate without congesting the network. How then do the TCP senders determine their sending rates such that they don't congest the network but at the same time make use of all the available bandwidth? Are TCP senders explicitly coordinated, or is there a distributed approach in which the TCP senders can set their sending rates based only on local information? TCP answers these questions using the following guiding principles:

- A lost segment implies congestion, and hence, the TCP sender's rate should be

decreased when a segment is lost. Recall from our discussion in Section 3.5.4, that a timeout event or the receipt of four acknowledgments for a given segment (one original ACK and then three duplicate ACKs) is interpreted as an implicit “loss event” indication of the segment following the quadruply ACKed segment, triggering a retransmission of the lost segment. From a congestion control standpoint, the question is how the TCP sender should decrease its congestion window size, and hence its sending rate, in response to this inferred loss event.

- An acknowledged segment indicates that the network is delivering the sender’s segments to the receiver, and hence, the sender’s rate can be increased when an ACK arrives for a previously unacknowledged segment. The arrival of acknowledgments is taken as an implicit indication that all is well—segments are being successfully delivered from sender to receiver, and the network is thus not congested. The congestion window size can thus be increased.
- Bandwidth probing. Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP’s strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then begins probing again to see if the congestion onset rate has changed. The TCP sender’s behavior is perhaps analogous to the child who requests (and gets) more and more goodies until finally he/she is finally told “No!”, backs off a bit, but then begins making requests

3.7 • TCP CONGESTION CONTROL 271

272 CHAPTER 3 • TRANSPORT LAYER

again shortly afterwards. Note that there is no explicit signaling of congestion state by the network—ACKs and loss events serve as implicit signals—and that each TCP sender acts on local information asynchronously from other TCP senders.

Given this overview of TCP congestion control, we’re now in a position to consider the details of the celebrated TCP congestion-control algorithm, which was first described in [Jacobson 1988] and is standardized in [RFC 5681]. The algorithm has three major components: (1) slow start, (2) congestion avoidance, and (3) fast recovery. Slow start and congestion avoidance are mandatory components of TCP, differing in how they increase the size of cwnd in response to received ACKs. We’ll see shortly that slow start increases the size of cwnd more rapidly (despite its name!) than congestion avoidance. Fast recovery is recommended, but not required, for TCP senders.

Slow Start

When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS [RFC 3390], resulting in an initial sending rate of roughly MSS/RTT. For example, if MSS = 500 bytes and RTT = 200 msec, the resulting initial sending rate is only about 20 kbps. Since the available bandwidth to the TCP sender may be much larger than MSS/RTT, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the slow-start state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged. In the example of Figure 3.51, TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgement arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of

the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase. But when should this exponential growth end? Slow start provides several answers to this question. First, if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew. It also sets the value of a second state variable, ssthresh (shorthand for “slow start threshold”) to $cwnd/2$ —half of the value of the $cwnd$ when congestion was detected. The second way in which slow start may end is directly tied to the value of ssthresh. Since ssthresh is half the value of cwnd when congestion was last detected, it might be a bit reckless to keep doubling cwnd when it reaches or surpasses the value of ssthresh. Thus, when the value of cwnd equals ssthresh, slow start ends and TCP transitions into congestion avoidance mode. As we’ll see, TCP increases cwnd more cautiously when in congestion-avoidance mode. The final way in which slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit (see Section 3.5.4) and enters the fast recovery state, as discussed below. TCP’s behavior in slow start is summarized in the FSM

274 CHAPTER 3 • TRANSPORT LAYER

description of TCP congestion control in Figure 3.52. The slow-start algorithm traces its roots to [Jacobson 1988]; an approach similar to slow start was also proposed independently in [Jain 1986].

Congestion Avoidance

On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT [RFC 5681]. This can be accomplished in several ways. A common approach is for the TCP sender to increase cwnd by MSS bytes ($MSS/cwnd$) whenever a new acknowledgment arrives. For example, if MSS is 1,460 bytes and cwnd is 14,600 bytes, then 10 segments are being sent within an RTT. Each arriving ACK (assuming one ACK per segment) increases the congestion window size by 1/10

Host A Host B

one segment

two segments

four segments

RTT

Time Time

Figure 3.51 TCP slow start

3.7 • TCP CONGESTION CONTROL 275

MSS, and thus, the value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received.

But when should congestion avoidance’s linear increase (of 1 MSS per RTT) end? TCP’s congestion-avoidance algorithm behaves the same when a timeout occurs. As in the case of slow start: The value of cwnd is set to 1 MSS, and the value of ssthresh is updated to half the value of cwnd when the loss event occurred. Recall, however, that a loss event also can be triggered by a triple duplicate ACK event. In this case, the network is continuing to deliver segments from sender to receiver (as indicated by the receipt of duplicate ACKs). So TCP’s behavior to this type of loss event should be less drastic than with a timeout-indicated loss: TCP halves the value of cwnd (adding in 3 MSS for good measure to account for

Slow
start
duplicate ACK
dupACKcount++
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
cwnd ≥ ssthresh
Congestion
avoidance
Fast
recovery
new ACK
cwnd=cwnd+MSS •(MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
retransmit missing segment
retransmit missing segment dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
Λ

Λ

Figure 3.52 FSM description of TCP congestion control

276 CHAPTER 3 • TRANSPORT LAYER

the triple duplicate ACKs received) and records the value of ssthresh to be half the value of cwnd when the triple duplicate ACKs were received. The fast-recovery state is then entered.

Fast Recovery

In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd. If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

Fast recovery is a recommended, but not required, component of TCP [RFC 5681]. It is interesting that an early version of TCP, known as TCP Tahoe, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The newer version of TCP, TCP Reno, incorporated fast recovery.

Figure 3.53 illustrates the evolution of TCP's congestion window for both Reno and Tahoe. In this figure, the threshold is initially equal to 8 MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicateACK event occurs, just after transmission round 8. Note that the congestion window is $12 \cdot \text{MSS}$ when this loss event occurs. The value of ssthresh is then set to

0

0 2345678 1

Transmission round

TCP Tahoe

ssthresh

ssthresh

Congestion window

(in segments)

9 10 11 12 13 14 15

2

4

6

8

10

12

14

16

TCP Reno

Figure 3.53 Evolution of TCP's congestion window (Tahoe and Reno)

VideoNote

Examining the behavior of TCP

3.7 • TCP CONGESTION CONTROL 277

$0.5 \cdot \text{cwnd} = 6 \cdot \text{MSS}$. Under TCP Reno, the congestion window is set to $\text{cwnd} = 6 \cdot \text{MSS}$ and then grows linearly. Under TCP Tahoe, the congestion window is set to

1 MSS and grows exponentially until it reaches the value of ssthresh, at which point it grows linearly.

Figure 3.52 presents the complete FSM description of TCP's congestion control algorithms—slow start, congestion avoidance, and fast recovery. The figure also indicates where transmission of new segments or retransmitted segments can occur. Although it is important to distinguish between TCP error control/retransmission and TCP congestion control, it's also important to appreciate how these two aspects of TCP are inextricably linked.

TCP Congestion Control: Retrospective

Having delved into the details of slow start, congestion avoidance, and fast recovery, it's worthwhile to now step back and view the forest from the trees. Ignoring the initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCP's congestion control consists of linear (additive) increase in cwnd of 1 MSS per RTT and then a halving (multiplicative decrease) of cwnd on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an additive-increase, multiplicative-decrease (AIMD) form of congestion control. AIMD congestion control gives rise to the "saw tooth" behavior shown in Figure 3.54, which also nicely illustrates our earlier intuition of TCP "probing" for bandwidth—TCP linearly increases its congestion window size (and hence its transmission rate) until a triple duplicate-ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth.

24 K

16 K

8 K

Time

Congestion window

Figure 3.54 Additive-increase, multiplicative-decrease congestion control

278 CHAPTER 3 • TRANSPORT LAYER

As noted previously, many TCP implementations use the Reno algorithm [Padhye 2001]. Many variations of the Reno algorithm have been proposed [RFC 3782; RFC 2018]. The TCP Vegas algorithm [Brakmo 1995; Ahn 1995] attempts to avoid congestion while maintaining good throughput. The basic idea of Vegas is to (1) detect congestion in the routers between source and destination before packet loss occurs, and (2) lower the rate linearly when this imminent packet loss is detected. Imminent packet loss is predicted by observing the RTT. The longer the RTT of the packets, the greater the congestion in the routers. Linux supports a number of congestion-control algorithms (including TCP Reno and TCP Vegas) and allows a system administrator to configure which version of TCP will be used. The default version of TCP in Linux version 2.6.18 was set to CUBIC [Ha 2008], a version of TCP developed for high-bandwidth applications. For a recent survey of the many flavors of TCP, see [Afanasyev 2010].

TCP's AIMD algorithm was developed based on a tremendous amount of engineering insight and experimentation with congestion control in operational networks. Ten years after TCP's development, theoretical analyses showed that TCP's congestion-control algorithm serves as a distributed asynchronous-optimization algorithm that results in several important aspects of user and network performance being simultaneously optimized [Kelly 1998]. A rich theory of congestion control has since been developed [Srikant 2004].

Macroscopic Description of TCP Throughput

Given the saw-toothed behavior of TCP, it's natural to consider what the average

throughput (that is, the average rate) of a long-lived TCP connection might be. In this analysis we'll ignore the slow-start phases that occur after timeout events. (These phases are typically very short, since the sender grows out of the phase exponentially fast.) During a particular round-trip interval, the rate at which TCP sends data is a function of the congestion window and the current RTT. When the window size is w bytes and the current round-trip time is RTT seconds, then TCP's transmission rate is roughly w/RTT . TCP then probes for additional bandwidth by increasing w by 1 MSS each RTT until a loss event occurs. Denote by W the value of w when a loss event occurs. Assuming that RTT and W are approximately constant over the duration of the connection, the TCP transmission rate ranges from $W/(2 \cdot RTT)$ to W/RTT .

These assumptions lead to a highly simplified macroscopic model for the steady-state behavior of TCP. The network drops a packet from the connection when the rate increases to W/RTT ; the rate is then cut in half and then increases by MSS/RTT every RTT until it again reaches W/RTT . This process repeats itself over and over again. Because TCP's throughput (that is, rate) increases linearly between the two extreme values, we have

$$\text{average throughput of a connection} = 0.75 \frac{W}{RTT}$$

3.7 • TCP CONGESTION CONTROL 279

Using this highly idealized model for the steady-state dynamics of TCP, we can also derive an interesting expression that relates a connection's loss rate to its available bandwidth [Mahdavi 1997]. This derivation is outlined in the homework problems. A more sophisticated model that has been found empirically to agree with measured data is [Padhye 2000].

TCP Over High-Bandwidth Paths

It is important to realize that TCP congestion control has evolved over the years and indeed continues to evolve. For a summary of current TCP variants and discussion of TCP evolution, see [Floyd 2001, RFC 5681, Afanasyev 2010]. What was good for the Internet when the bulk of the TCP connections carried SMTP, FTP, and Telnet traffic is not necessarily good for today's HTTP-dominated Internet or for a future Internet with services that are still undreamed of.

The need for continued evolution of TCP can be illustrated by considering the high-speed TCP connections that are needed for grid- and cloud-computing applications. For example, consider a TCP connection with 1,500-byte segments and a 100 ms RTT, and suppose we want to send data through this connection at 10 Gbps. Following [RFC 3649], we note that using the TCP throughput formula above, in order to achieve a 10 Gbps throughput, the average congestion window size would need to be 83,333 segments. That's a lot of segments, leading us to be rather concerned that one of these 83,333 in-flight segments might be lost. What would happen in the case of a loss? Or, put another way, what fraction of the transmitted segments could be lost that would allow the TCP congestion-control algorithm specified in Figure 3.52 still to achieve the desired 10 Gbps rate? In the homework questions for this chapter, you are led through the derivation of a formula relating the throughput of a TCP connection as a function of the loss rate (L), the round-trip time (RTT), and the maximum segment size (MSS):

Using this formula, we can see that in order to achieve a throughput of 10 Gbps, today's TCP congestion-control algorithm can only tolerate a segment loss probability of $2 \cdot 10^{-10}$ (or equivalently, one loss event for every 5,000,000,000 segments)—a very low rate. This observation has led a number of researchers to investigate new versions of TCP that are specifically designed for such high-speed environments; see [Jin 2004; RFC 3649; Kelly 2003; Ha 2008] for discussions of these efforts.

3.7.1 Fairness

Consider K TCP connections, each with a different end-to-end path, but all passing through a bottleneck link with transmission rate R bps. (By bottleneck link, we mean average throughput of a connection = $1.22 \frac{\text{bps}}{\text{MSS}}$

RTT $2L$

280 CHAPTER 3 • TRANSPORT LAYER

that for each connection, all the other links along the connection's path are not congested and have abundant transmission capacity as compared with the transmission capacity of the bottleneck link.) Suppose each connection is transferring a large file and there is no UDP traffic passing through the bottleneck link. A congestion-control mechanism is said to be fair if the average transmission rate of each connection is approximately R/K ; that is, each connection gets an equal share of the link bandwidth. Is TCP's AIMD algorithm fair, particularly given that different TCP connections may start at different times and thus may have different window sizes at a given point in time? [Chiu 1989] provides an elegant and intuitive explanation of why TCP congestion control converges to provide an equal share of a bottleneck link's bandwidth among competing TCP connections.

Let's consider the simple case of two TCP connections sharing a single link with transmission rate R , as shown in Figure 3.55. Assume that the two connections have the same MSS and RTT (so that if they have the same congestion window size, then they have the same throughput), that they have a large amount of data to send, and that no other TCP connections or UDP datagrams traverse this shared link. Also, ignore the slow-start phase of TCP and assume the TCP connections are operating in CA mode (AIMD) at all times.

Figure 3.56 plots the throughput realized by the two TCP connections. If TCP is to share the link bandwidth equally between the two connections, then the realized throughput should fall along the 45-degree arrow (equal bandwidth share) emanating from the origin. Ideally, the sum of the two throughputs should equal R . (Certainly, each connection receiving an equal, but zero, share of the link capacity is not a desirable situation!) So the goal should be to have the achieved throughputs fall somewhere near the intersection of the equal bandwidth share line and the full bandwidth utilization line in Figure 3.56.

Suppose that the TCP window sizes are such that at a given point in time, connections 1 and 2 realize throughputs indicated by point A in Figure 3.56. Because the amount of link bandwidth jointly consumed by the two connections is less than R , no loss will occur, and both connections will increase their window by 1 MSS per RTT as a result of TCP's congestion-avoidance algorithm. Thus, the joint throughput of the two connections proceeds along a 45-degree line (equal increase for both connections) starting from point A. Eventually, the link bandwidth jointly consumed by the two connections will be greater than R , and eventually packet loss will occur. Suppose that connections 1 and 2 experience packet loss when they realize throughputs indicated by point B. Connections 1 and 2 then decrease their windows by a factor of two. The resulting throughputs realized are thus at point C, halfway along a vector starting at B and ending at the origin. Because the joint bandwidth use is less than R at point C, the two connections again increase their throughputs along a 45-degree line starting from C. Eventually, loss will again occur, for example, at point D, and the two connections again decrease their window sizes by a factor of two, and so on. You should convince yourself that the bandwidth realized by the two connections eventually fluctuates along the equal bandwidth share line. You should also convince yourself that the two connections will converge to this behavior regardless of where they are in the two-dimensional

space! Although a number of idealized assumptions lie behind this scenario, it still provides an intuitive feel for why TCP results in an equal sharing of bandwidth among connections.

In our idealized scenario, we assumed that only TCP connections traverse the bottleneck link, that the connections have the same RTT value, and that only a

R

R

Equal

bandwidth

share

Connection 1 throughput

Connection 2 throughput

D

B

C

A

Full bandwidth

utilization line

Figure 3.56 Throughput realized by TCP connections 1 and 2

282 CHAPTER 3 • TRANSPORT LAYER

single TCP connection is associated with a host-destination pair. In practice, these conditions are typically not met, and client-server applications can thus obtain very unequal portions of link bandwidth. In particular, it has been shown that when multiple connections share a common bottleneck, those sessions with a smaller RTT are able to grab the available bandwidth at that link more quickly as it becomes free (that is, open their congestion windows faster) and thus will enjoy higher throughput than those connections with larger RTTs [Lakshman 1997].

Fairness and UDP

We have just seen how TCP congestion control regulates an application's transmission rate via the congestion window mechanism. Many multimedia applications, such as Internet phone and video conferencing, often do not run over TCP for this very reason—they do not want their transmission rate throttled, even if the network is very congested. Instead, these applications prefer to run over UDP, which does not have built-in congestion control. When running over UDP, applications can pump their audio and video into the network at a constant rate and occasionally lose packets, rather than reduce their rates to "fair" levels at times of congestion and not lose any packets. From the perspective of TCP, the multimedia applications running over UDP are not being fair—they do not cooperate with the other connections nor adjust their transmission rates appropriately. Because TCP congestion control will decrease its transmission rate in the face of increasing congestion (loss), while UDP sources need not, it is possible for UDP sources to crowd out TCP traffic. An area of research today is thus the development of congestion-control mechanisms for the Internet that prevent UDP traffic from bringing the Internet's throughput to a grinding halt [Floyd 1999; Floyd 2000; Kohler 2006].

Fairness and Parallel TCP Connections

But even if we could force UDP traffic to behave fairly, the fairness problem would still not be completely solved. This is because there is nothing to stop a TCP-based application from using multiple parallel connections. For example, Web browsers often use multiple parallel TCP connections to transfer the multiple objects within a Web page. (The exact number of multiple connections is configurable in most browsers.) When an application uses multiple parallel connections, it gets a larger

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

Let's turn 3.7 into a clean mental picture you can carry into the exam.

1 How TCP *estimates* congestion (no router help!)

TCP **must do end-to-end congestion control** (IP gives no feedback), so it infers congestion from what it sees:

a) Lost segment \Rightarrow congestion signal

A “**loss event**” at the sender is:

- **Timeout** for an unacked segment, or
- **3 duplicate ACKs** for the same sequence number (\rightarrow fast retransmit)

Interpretation:

- Router buffers probably overflowed \rightarrow **network is congested**.
 - Reaction: **reduce sending rate** by shrinking **cwnd**.
-

b) Acknowledgment arrival \Rightarrow things are okay

Every **new ACK** tells the sender:

- “Data you sent *did* reach the receiver.”
- So, the path currently can handle at least this rate.

TCP uses ACKs to **clock** its sending:

- More frequent ACKs \rightarrow increase **cwnd** faster.
- Fewer ACKs \rightarrow increase **cwnd** more slowly.

This is called **self-clocking**.

c) Bandwidth probing idea

TCP’s strategy:

1. **Increase** sending rate while no congestion is seen (ACKs coming normally).
2. **Decrease** sending rate when a loss event happens.
3. Repeat.

So TCP repeatedly:

- **Probes up** (increases `cwnd`)
- Hits congestion (loss)
- **Backs off**, then probes again

This is the classic “**probe until you hear NO**” behavior.

2 The basic control knob: cwnd

- `cwnd` = **congestion window** (bytes)
- Assume receiver `rwnd` is huge (ignore flow control), and sender always has data.
- TCP constraint:

$$\text{Unacked data} = \text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- Rough **sending rate** $\approx \text{cwnd} / \text{RTT}$ bytes/sec.

So, **changing cwnd = changing sending rate**.

3 Slow start

Despite the name, growth is **fast (exponential)**.

Initial state

- When connection starts or after timeout:
 - `cwnd` = 1 MSS
 - `ssthresh` = some large initial value (e.g., 64 KB) or half of previous `cwnd` after loss.

Behavior

- For **each new ACK** that acknowledges previously unsent data:
 - `cwnd += 1 MSS`
- Over 1 RTT, if `cwnd = N MSS`, you send N segments and (roughly) get N ACKs → `cwnd` doubles each RTT.

So:

- RTT 1: 1 segment
- RTT 2: 2 segments

- RTT 3: 4 segments
- RTT 4: 8 segments
- ...

Stop conditions for slow start:

1. **cwnd reaches ssthresh** → switch to **congestion avoidance**.

2. Timeout (loss):

- **ssthresh = cwnd / 2**
- **cwnd = 1 MSS**
- Enter **slow start** again.

3. 3 duplicate ACKs:

- Trigger **fast retransmit & fast recovery**, not a full restart.

Key idea: quickly ramp up from tiny rate to some plausible region before being cautious.

4 Congestion avoidance (AIMD “additive increase” part)

Once **cwnd ≥ ssthresh**, TCP assumes it's near the congestion point and grows **slowly**.

Behavior

- Target: **increase cwnd by 1 MSS per RTT** (linear).
- Implementation: for each new ACK:

$$\text{cwnd} += \frac{\text{MSS}^2}{\text{cwnd}} \approx \text{MSS} \cdot \frac{\text{MSS}}{\text{cwnd}}$$

(Common shorthand: **cwnd += MSS * (MSS/cwnd)**).

If there are N segments in flight during an RTT, you get N ACKs, each increases **cwnd** by $\approx \text{MSS}/N$, so over one RTT:

$$\Delta\text{cwnd} \approx 1 \text{ MSS}.$$

When loss happens in congestion avoidance:

There are **two cases**:

1. Timeout:

- Strong signal of heavy congestion.
- **ssthresh = cwnd / 2**
- **cwnd = 1 MSS**
- Enter **slow start**.

2. 3 duplicate ACKs:

- Less severe: network is still delivering later packets.
 - Trigger **fast retransmit + fast recovery**.
-

5 Fast retransmit & fast recovery (the “Reno” part)

Fast retransmit (detect loss without waiting for timeout)

If sender gets **3 duplicate ACKs** for the same sequence number:

- Assume the **next segment is lost**.
- **Immediately retransmit** that missing segment (no timeout wait).

This is **fast retransmit**.

Fast recovery (what to do with cwnd around this loss)

This is what **TCP Reno** adds compared to old TCP Tahoe.

When **3 duplicate ACKs** arrive:

1. **Set**:

- $\text{ssthresh} = \text{cwnd} / 2$
- $\text{cwnd} = \text{ssthresh} + 3 * \text{MSS}$
(3 MSS for the 3 duplicate ACKs already in flight)

2. **Retransmit** the missing segment (fast retransmit).

3. For **each additional duplicate ACK** for that same missing segment:

- $\text{cwnd} += 1 \text{ MSS}$
- This allows sending **one new segment per dupACK**, keeping the pipe full.

When a **new ACK** (for the retransmitted segment) finally arrives:

- This means the loss is recovered and data is now in order again.
- **Deflate** cwnd back to ssthresh .
- Enter **congestion avoidance** (no slow start).

If instead a **timeout** occurs during fast recovery:

- Treat it like a serious loss:
 - $\text{ssthresh} = \text{cwnd} / 2$
 - $\text{cwnd} = 1 \text{ MSS}$
 - Enter **slow start**.
-

6 Tahoe vs Reno (high-level difference)

- **TCP Tahoe:**

- On **any loss event** (timeout or 3 dupACKs):
 - $ssthresh = cwnd / 2$
 - $cwnd = 1 \text{ MSS}$
 - Enter **slow start**.

- **TCP Reno:**

- On **timeout**:
 - Same as Tahoe (full reset → slow start).
- On **3 dupACKs**:
 - Use **fast retransmit + fast recovery**:
 - Halve $cwnd$, but do **not** go all the way to 1 MSS.
 - Stay in “high throughput but slightly reduced” mode, then go to congestion avoidance.

Graphically:

- Both do exponential **slow start** until **ssthresh**, then linear growth.
- On loss:
 - Tahoe: $cwnd$ drops to 1 MSS (big sawtooth).
 - Reno: $cwnd$ drops to about half (flatter, smoother sawtooth).

7 Big-picture: AIMD, sawtooth, probing

Ignoring the initial slow start and timeouts, steady-state TCP Reno does:

- **Additive Increase:**
 - $+1 \text{ MSS}$ per RTT while no losses (congestion avoidance).
- **Multiplicative Decrease:**
 - $cwnd = cwnd / 2$ on 3 dupACK loss event.

This gives the classic **sawtooth** graph:

- $cwnd$ ramps up linearly → hits loss → drops by half → ramps up again → ...

That is the “probing” process: try more bandwidth until loss; back off; try again.

If you’d like, next we can do a **compact exam-style summary** (“If you see $cwnd$ graph like this, which phase is it?” + quick Q&A prompts), or some **short numerical examples** (e.g., what happens to $cwnd$ across a timeline of ACKs and a triple dupACK).



Q16 Describe the update rules of the transmission window W in slow start and congestion control states of TCP (Tahoe).

exam question attached, answer it

Here's a compact exam-style answer:

For **TCP Tahoe**, let W be the congestion window in units of MSS and let **ssthresh** be the slow-start threshold.

1. Slow start state

- Initial: $W = 1$.
- **On each ACK for new data:**

$$W \leftarrow W + 1$$

(so W roughly **doubles every RTT**).

- **When a loss is detected (timeout or triple dup ACK):**

$$\text{ssthresh} \leftarrow W/2, \quad W \leftarrow 1$$

and remain/return to slow start (if $W < \text{ssthresh}$).

2. Congestion avoidance state

- Enter when $W \geq \text{ssthresh}$.
- **On each ACK for new data:**

$$W \leftarrow W + \frac{1}{W}$$

so that W increases by **about 1 MSS per RTT** (linear growth).

- **On loss (timeout or triple dup ACK):**

$$\text{ssthresh} \leftarrow W/2, \quad W \leftarrow 1$$

and go back to **slow start**.



why $w=w+1/w$ means W increases by about 1 MSS per RTT (linear growth)?

Great question — this line is **very commonly tested**, and the intuition is simple once you see it.

We are in **Congestion Avoidance**, and the rule is:

$$W \leftarrow W + \frac{1}{W} \quad (\text{on each ACK})$$

Here's **why this leads to linear growth of ~1 MSS per RTT:**

✓ Key Idea

- **There are about W ACKs per RTT**
 - Because the sender can have **at most W outstanding segments**
 - Each of those segments gets ACKed → about **W ACKs per RTT**
- **Each ACK increases W by $1/W$**

So over **one RTT**:

$$\text{Total increase in } W = W \times \frac{1}{W} = 1$$

✓ So the congestion window increases by exactly ~1 MSS per RTT

That's why we say:

Congestion avoidance increases the window linearly: +1 MSS per RTT

✓ Compare with Slow Start

Phase	Update per ACK	Growth per RTT	Type
Slow Start	$W \leftarrow W + 1$	$+W$	Exponential
Congestion Avoidance	$W \leftarrow W + \frac{1}{W}$	$+1$	Linear ✓

✓ Intuitive Interpretation

Think of it like this:

- In **slow start**, TCP is **aggressive** → "Let me grow as fast as possible."
- In **congestion avoidance**, TCP is **careful** → "Let me probe slowly for extra bandwidth."

So instead of doubling,
it **adds only 1 segment per RTT.**

✓ One-Line Exam Answer

Since about W ACKs are received per RTT and each ACK increases the window by $1/W$, the total increase per RTT is $W \cdot (1/W) = 1$, hence linear growth of 1 MSS per RTT.

If you want, I can also explain:

- Why Tahoe resets to $W = 1$
- Why Reno reduces to $W/2$
- Or how this appears as the **sawtooth curve** in AIMD.



Q4 TCP is implemented at:

- a) Switches
- b) Routers
- c) host devices
- d) all the above

Correct answer: (c) host devices

Why the answer is host devices

TCP is implemented only at the end systems (hosts) — that is:

- Your **computer**
- The **server**
- Any device that actually **runs applications**

TCP is part of the **Transport Layer**, whose job is to provide:

- Process-to-process communication
- Reliability
- Flow control
- Congestion control

All of this logic exists in the **operating system of end hosts**, not inside the network.

Why the other options are wrong

(a) **Switches** → Wrong

- Switches operate at:

- Layer 2 (Data Link)
 - Sometimes Layer 1
 - They **forward frames**, they do NOT:
 - Track TCP sequence numbers
 - Do congestion control
 - Do retransmissions
-

(b) Routers → Wrong

- Routers operate at:
 - Layer 3 (Network Layer)
 - They only:
 - Forward IP packets
 - Use routing tables
 - They **do not implement TCP logic**
 - No sliding windows
 - No ACK tracking
 - No congestion window
-

(d) All the above → Wrong

- Because switches and routers **do not run TCP**
-

One-line exam answer

TCP is implemented only at host devices because it is a transport-layer protocol providing end-to-end reliability and congestion control.

If you want, I can also explain:

- Where IP is implemented
- Where UDP is implemented
- Where congestion actually *physically occurs vs is controlled* (great exam concept)



Q11 Mark the (one) correct statement

- a) UDP implements packet retransmissions and packet reordering
- b) UDP implements packet retransmissions, but does not implement packet reordering
- c) UDP does not implement packet retransmissions and packet reordering

Q12 In TCP, if we increase the timeout time with respect to the RTT

- a) The number of retransmitted packets will increase
- b) The number of retransmitted packets will decrease
- c) The number of retransmitted packets will be the same

Here are the correct answers with clear reasoning (exam-oriented):

✓ Q11: Mark the (one) correct statement

Options:

- a) UDP implements packet retransmissions and packet reordering ✗
- b) UDP implements packet retransmissions, but does not implement packet reordering ✗
- c) **UDP does not implement packet retransmissions and packet reordering ✓**

✓ Correct answer: (c)

🔍 Explanation:

UDP is a **connectionless, best-effort protocol**. It provides:

- ✗ No retransmissions
- ✗ No packet reordering
- ✗ No reliability
- ✗ No congestion control
- ✓ Only multiplexing + checksum

If packets are:

- Lost → UDP does **nothing**
- Reordered → UDP does **nothing**

Any retransmission or reordering (e.g., in video streaming) must be handled by the **application**, not by UDP.

✓ Q12: In TCP, if we increase the timeout time with respect to the RTT

Options:

- a) The number of retransmitted packets will increase ✗
- b) **The number of retransmitted packets will decrease ✓**
- c) The number of retransmitted packets will be the same ✗

 **Correct answer: (b)**

Explanation:

Retransmissions due to **timeout happen when:**

ACK not received before Timeout

If we **increase the timeout value:**

- TCP waits **longer before assuming loss**
-  **Fewer premature timeouts**
-  **Fewer unnecessary retransmissions**
-  **But at the cost of slower recovery from real losses**

So:

- **Larger timeout → fewer retransmissions**
 - **Smaller timeout → more spurious retransmissions**
-

Final Exam Answers (One-liners)

- **Q11:** (c) UDP does not implement packet retransmissions and packet reordering
 - **Q12:** (b) The number of retransmitted packets will decrease
-

If you want, I can also give you:

- A **one-page TCP vs UDP reliability summary**
- Or a **timeout vs RTT numerical example** for intuition.