Chapter 3: Transport Layer

ATNLP (layers) (PLaNeT-A)
Application
Transport
Network
Link
Physical

- *Fundamental problems in computer networking:*
    - How communication can take place over a medium that may lose/corrupt data
    - How to control transmission rate within Transport Layer to avoid/reduce congestion
- Transport Layer provides **logical communication** between **processes** on different hosts/end systems
    - ***Logical communication*** → from application's perspective, 2 communicating end systems are directly connected → Transmission journey is abstracted
- Transport-layer protocols are implemented in end systems, not in routers
- Routers only look at network-layer fields of datagrams
- On the sending side:
    - transport-layer converts application-layer messages into transport-layer segments/packets
        - (Application messages broken into chunks and each chunk given a transport-layer header → = *segments*)
    - Segments passed to network-layer at sending end-system
    - Segments encapsulated in network-layer packets (datagrams) and sent to destination
- On the receiving side:
    - Network layer extracts transport-layer segment from datagram
    - Passes segment up to transport-layer
    - Transport-layer process + passes segment to application-layer

3.1.2 Relationship between Transport-Layer & Network-Layer

- **Analogy:** Imagine there is *house 1* and *house 2*, each home to multiple people who like to write to each other. *House 1's person A* is responsible for sending/collecting mail from *house 2*, while *house 2's person B* is responsible for sending/collecting mail from *house A.*
    - houses 1 & 2 = **hosts**
    - People in the houses (excluding A & B) = **processes**
    - Persons A & B = **transport-layer protocol**
    - Postal service = **network-layer protocol**
-

Transport Layer (uses postal service)	**Network Layer (postal service)**
Provides logical communication between *processes* (people) running on hosts (houses)	Provides logical communication between *hosts* (houses)

- Even when network layer is potentially unreliable, transport layer can still make certain guarantees (ie: encryption of messages)

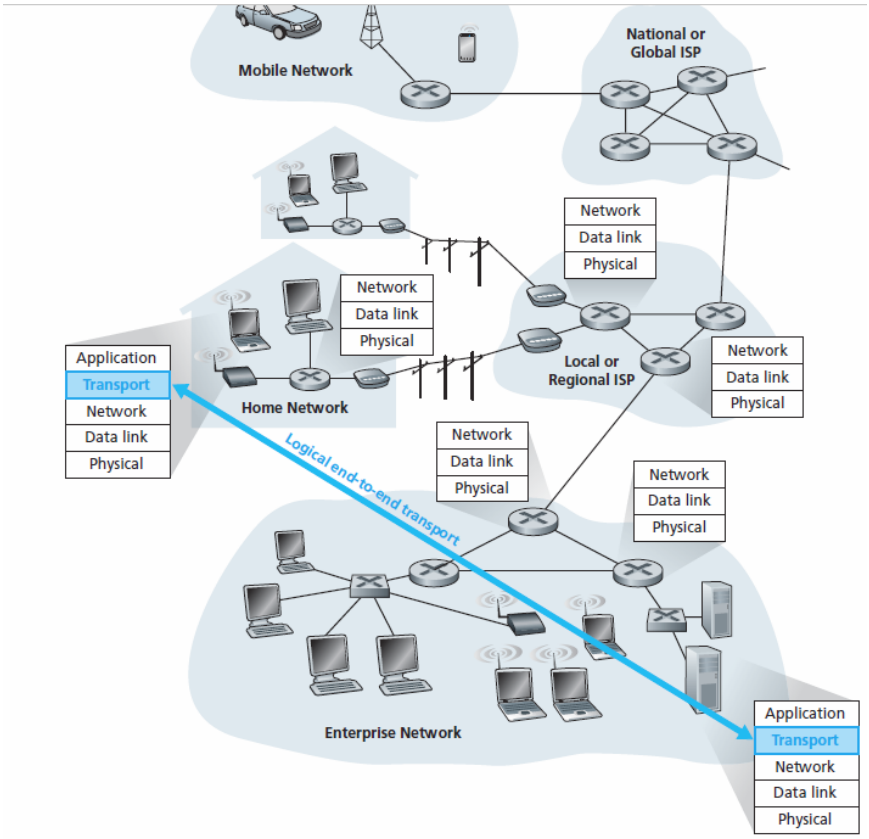


Figure 3.1 ♦ The transport layer provides logical rather than physical communication between application processes

3.1.2 Overview of the Transport Layer in the Internet

- Recall: The internet (or TCP/IP networks) offer 2 different transport-layer protocols:
  - UDP (User Datagram Protocol) → connectionless, unsafe
  - TCP (Transmission Control Protocol) → connection-oriented, safe
  - App dev chooses between UDP and TCP when creating sockets
- *Segment:* transport-layer packet (TCP and UDP)
- *Datagram:* network-layer packet
- IP (Internet Protocol) → logical ('abstracted') communication between hosts
  - = **best-effort delivery service** → makes no guarantees
  - Thus IP = **unreliable**
- *While IP = host-host communication, UDP/TCP = process-process communication*
- *Most fundamental responsibility of TCP and UDP:* **= transport-layer multi/demultiplexing**
  - = extending host-host delivery to process-process delivery

        - AKA extend IP's delivery service between two hosts to a delivery system between two processes, each running on a different host
- UDP and TCP also provide integrity checking → error-detection fields in segment headers
- ***Process-process delivery*** *&* ***integrity checking*** *<u>are the only services offered by UDP</u>*
    - Also recall UDP doesn't provide reliable data transfer (data may not arrive safely)
- TCP, on other hand, offers more services:
    - **Reliable data transfer** using *flow control*
        - *"Converts IP's unreliable service to reliable"*
    - **Congestion control** prevents individual TCP connections from being traffic hogs at routers/links
        - Tries to give each connection equal share of link/router bandwidth by *regulating transmission rates coming into links/routers*
            - UDP does no such regulating
        - *"Good for internet as a whole, not to any particular app"*

TCP services:	**UDP services:**
Process-process delivery	Process-process delivery
Integrity checking	Error checking
Reliable data transfer	*(nothing more)*
Congestion control	*(nothing more)*

3.2 Multiplexing and Demultiplexing:

- *Extending IP's (network-layer) host-host communication to TCP/UDP's (transport-layer) process-process communication*
- Every network needs multi/demultiplexing (not just internet)
- *At destination host, transport-layer receives segments from network-layer and must deliver these segments to the correct processes in the host*
    - ie: you have 2 HTTP processes and 1 FTP process running → transport-layer must send appropriate segments to each of these application-layer processes
    - How it works: multiple sockets at destination host, each with unique identifier
        - Format of identifier tells us whether TCP or UDP socket
        - Fields in data segments specify which is correct socket
            - **Demultiplexing:** HAPPENS AT DESTINATION HOST
                - delivering transport-layer segment to correct socket, based on segment header info
            - **Multiplexing:** HAPPENS AT SOURCE HOST
                - Creating segments (data chunks + header info) and passing into network layer

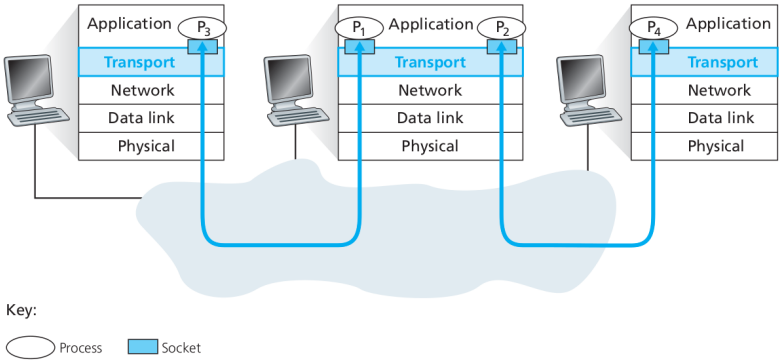


Figure 3.2 ♦ Transport-layer multiplexing and demultiplexing

- Recall house-house mail analogy:
  - When house 1's Person A puts letters in envelopes and gives letters to post office, he is **multiplexing**
  - When house 2's Person B gets mail from post office and gives letters to siblings based on who letters are addressed to (*'header info'*), he is **demultiplexing**

How is multiplexing actually done in a host?

- Requires sockets with unique identifiers
- Requires segments to have header info pointing to correct sockets
  - **→ Source port number & destination port number**
- **Port number**: 16-bit number ranging from 0 to 65535
- **Well-known port numbers:** port numbers ranging from 0 to 1023
  - = restricted → reserved for use by well-known app-layer protocols

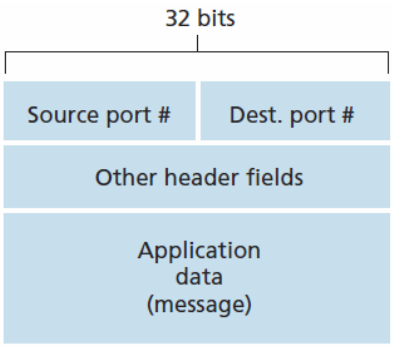


Figure 3.3 ♦ Source and destination port-number fields in a transport-layer segment

(like HTTP which uses port 80)

- When segment arrives at dest host, check which socket appears in segment's header info and send the segment to the socket at the dest host with that port number, to be passed to the destination process

Connectionless (UDP) Multiplexing and Demultiplexing

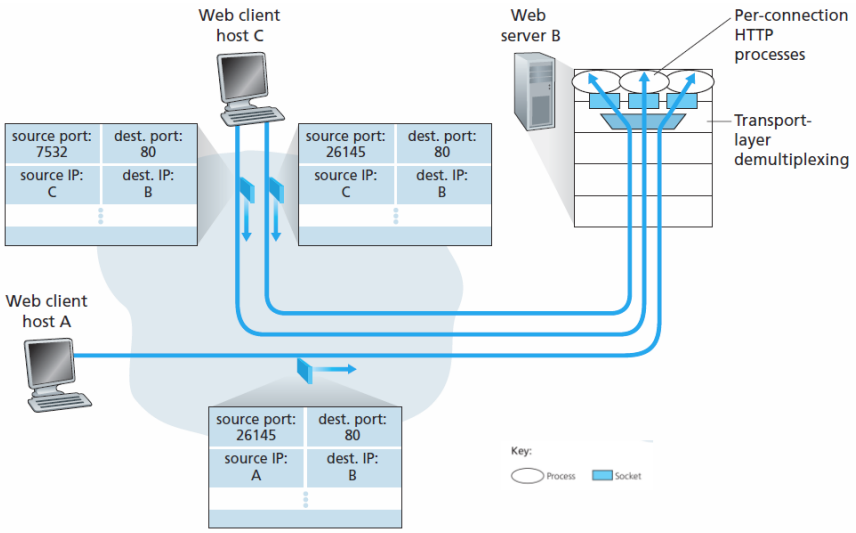
With port numbers assigned to UDP sockets, we can now precisely describe UDP multiplexing/demultiplexing. Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B. The transport layer in Host A creates a transport-layer segment 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.

- A UDP socket is identified by a 2-tuple: [dest IP, dest port #]
- 2 arriving UDP segments with different source IP addresses or source port numbers will be sent to the **same socket**
- The reason the *source port number* is sent to the destination in the segment header info is to allow the destination to send messages back to the host in response

Connection-Oriented (TCP) Multiplexing and Demultiplexing

- A TCP socket is identified by a 4-tuple: *[src IP, src port #, dest IP, dest port #]*
- 2 arriving TCP segments with different source IP addresses or source port numbers will be sent to **2 different sockets**
- When a client *"knocks on the welcoming door socket" of the server*, the connection socket created for said client is identified by 4 pieces of information from the client's connection request: **source port, source IP, destination port,** and **destination IP**



← Host C initiates 2 HTTP sessions to server B, Host A initiates 1 HTTP session to B.
← Hosts A and C, as well as Server B, each have their own IP address

Figure 3.5 ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

← Host C assigns 2 different port numbers (7532 & 26145) to its HTTP sessions. Host A also happens to assign port 26145 to its HTTP connection
← this is not a problem since hosts A and C have different IPs
← so server B can still differentiate between their HTTP sessions and demultiplex correctly

Web servers and TCP:

- Consider a webserver running on port 80
- When clients (browsers) send data segments to the server, *all* segments have destination port 80
    - Both the TCP handshake and subsequent HTTP requests take place on port 80
    - Server distinguishes between different clients on different hosts based on source IP addresses and source port numbers → doesn't matter that they're all entering via port 80

3.3 Connectionless Transport: UDP

- Basic → pretty much fulfills the minimum requirements of a transport protocol
- Provides multi/demultiplexing and light error checking, but nothing else
- If app dev chooses UDP instead of TCP, app is almost directly talking with IP
    - UDP adds nothing to IP protocol
- How UDP works:
    - Takes messages from application process
    - Attaches source/dest port numbers for multi/demultiplexing
    - Adds 2 other small fields
    - Passes resulting segment to network layer
    - Network layer puts segment in *IP Datagram* and makes best-effort attempt to deliver safely
    - If segment arrives at dest host, UDP uses dest port number to drop the segment into the correct socket for transmission to the destination process
- No handshaking involved → *connectionless*
- DNS (application-layer) uses UDP
    - When DNS client-side wants to send query, constructs DNS message
    - Passes message to UDP
    - UDP adds header fields (source/dest port nums) to message → segment created
    - Network layer puts segment in datagram, attempts to send to DNS name server
    - DNS client-side now waits for reply to its query from DNS name-server
        - If no reply (ie because message lost): either tries sending to another name-server, or tells invoking app that it can't get a reply

- If everyone started streaming HD videos over UDP (ie with no congestion control), there would be extreme packet overflow at routers and thus very few UDP packets would successfully reach their destinations
- It is technically possible to have reliable data transfer when using UDP, but will require complex application code

Reasons UDP can be more appropriate than TCP:

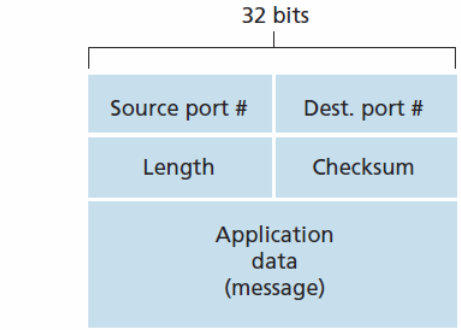
- **Finer app-level control over data**
  - Control what is sent and when → no things like congestion control getting in the way
  - Real-time applications like VoIP can't afford delays due to things like congestion control, so UDP can be better
- **No connection established**
  - Data transmission can begin instantaneously with UDP → which is why DNS uses TCP
  - On other hand, eg, HTTP values reliability over everything, so TCP (with its 3-way handshake) is more appropriate
- **Doesn't maintain connection state**
  - As a result → UDP server can have many more active clients than TCP server
  - (TCP maintains connection state → has to keep track of send/receive buffers and other parameters)
- **Small packet header overload**
  - UDP: 8 bytes of overhead
  - TCP: 20 bytes of overhead

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	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

UDP Segment Structure:

- Application data occupies data field
  - ie: for DNS, data field contains queries
  - ie: for VoIP, data field contains audio
- UDP header:
  - *4 fields, each 2 bytes:*
  - Source port field:
    - Which socket/process source host is sending data from
  - Dest port field:
    - Which socket/process in dest host to send data to
  - Length field:
    - Number of bytes in UDP segment
  - Checksum field:
    - Used by dest host to check if errors introduced into segment



UDP segment structure

- Port numbers (stored in fields) allow destination host to pass application data to correct process (via correct socket) → demultiplexing

3.3.2 UDP Checksum

- Provides error detection
- Used to determine whether bits within UDP segment have been altered (ie: by noise in links)
- UDP at source-side performs 1's complement of sum of all 16-bit words in segment → any overflow encountered during sum is wrapped around
    - Result put in checksum of UDP segment
- Since link-layer protocols like Ethernet also provide error-checking, why should UDP as well?
    - Answer: no guarantee that all links between src & dest will have error checking
    - Could also encounter bit errors when segment stored in router memory

~~data in [Stone 1998; Stone 2000].~~ As an example, suppose that we have the following three 16-bit words:

0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

1011101110110101
1000111100001100
0100101011000010

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum. At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

- So, UDP neither guarantees link-by-link reliability nor router-memory reliability, so it must provide error detection at transport-layer
    - *On an end-end basis*
- **End-end principle:** put features at 'ends' of network
    - **Clarify this**

3.4 Principles of Reliable Data Transfer (in general)

- **Reliable data transfer protocol:** during transmission of data, no bits corrupted and order preserved
- Task is difficult since *layer below* might be unreliable
    - ie you can TCP is reliable, but it runs on top of IP which is unreliable (also physical mediums can be unreliable)

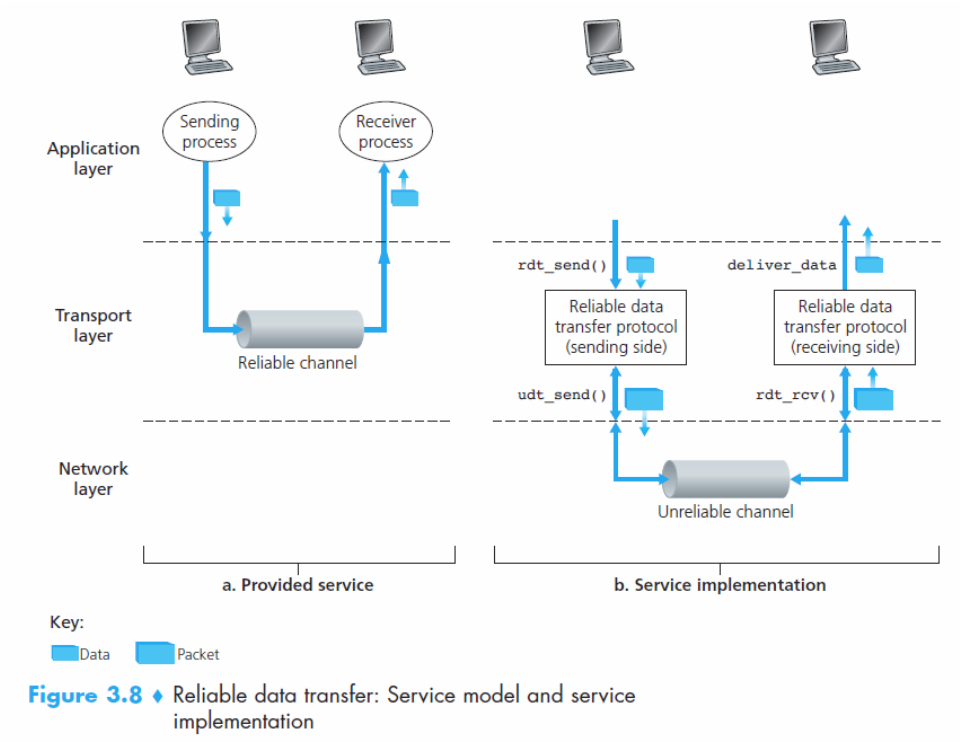


Figure 3.8 ♦ Reliable data transfer: Service model and service implementation

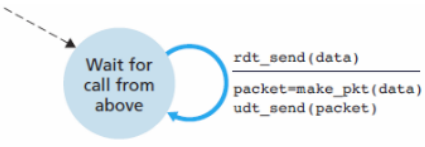
- **Unidirectional data transfer:** sending to receiving side
- **Bidirectional data transfer:** both sides sending and receiving

3.4.1 Building a Reliable Data Transfer Protocol

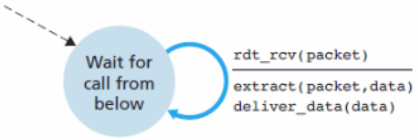
→ going to illustrate process of building reliable data transfer protocol with increasingly complex Finite State Machines
→ in each FSM, time moves forward from top to bottom of diagram

Reliable Data Transfer over perfectly reliable channel: rdt1.0

- Simplest case: underlying channel is completely reliable.
- Protocol = rdt1.0
- Separate Finite State Machines (FSM) for sender and receiver
- Arrows on FSM indicate transition of protocol from one state to another (can be back to itself)
- **Sending side:**



a. rdt1.0: sending side



b. rdt1.0: receiving side

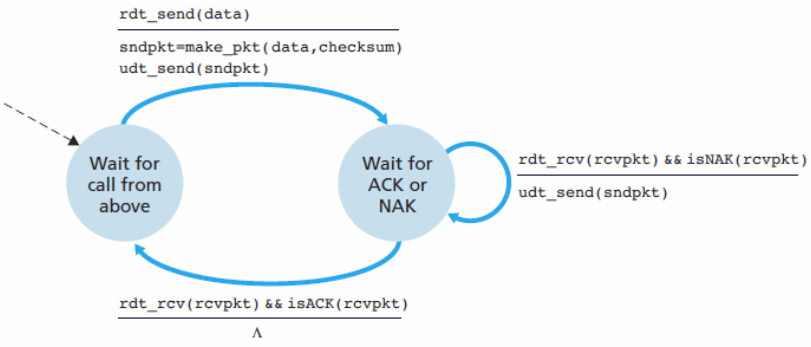
rdt1.0 – A protocol for a completely reliable channel

    - Sending side of *rdt* accepts data from upper level via *rdt_send(data)*
    - Creates packet containing data with *make_pkt(data)*
    - Sends packet into channel with *udt_send(packet)*
- **Receiving side:**
    - Receives packet from underlying channel via *rdt_rcv(packet)*
    - Extracts data from packet with *extract(packet,data)*
    - Passes data to upper layer with *deliver_data(data)*

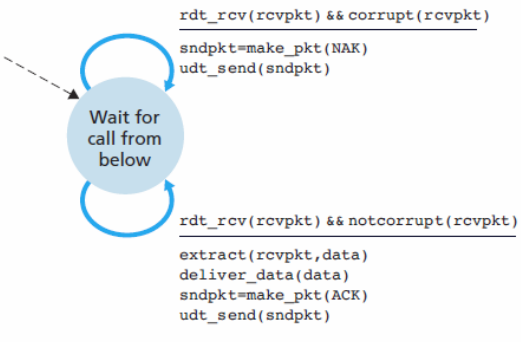
Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

- To deal with errors → ARQs required
- **ARQ (Automatic Repeat reQuest) protocols:** allows dest to let src know what has been received correctly and what hasn't
    - **Positive acknowledgement (ACK) =** "OK" → could be a 1
    - **Negative acknowledgement (NAK) =** "Please repeat that" → could be a 0
- **ARQ requirements:**
    - **Error detection**
        - Must be able to detect when errors have occurred → ie UDP uses checksum field
        - Receiver can tell if data arrived corrupted based on the checksum field
    - **Receiver feedback**
        - *ACK* ("OK") and *NAK* ("Please repeat that")
    - **Retransmission**
        - If dest receives a corrupt packet, src will resend that packet
- rdt2.0 will employ a **stop-and-wait ARQ protocol** → ensures data is not lost due to dropped packets and that data arrives in correct order
    - Using requirements specified above

- the ***stop and wait*** ARQ protocol (below - figure 3.10)
    - employs error detection, positive acknowledgements, negative acknowledgements



a. rdt2.0: sending side



b. rdt2.0: receiving side

Sending side: 2 states (left & right)
left:
→ wait for data to arrive from upper level via *rdt_send(data)*.
→ when data arrives, store it in the packet *sndpkt* along with a checksum.
→ then send *sndpkt* to the next layer with *udt_send(sndpkt)*
right:
→ waiting for ACK/NAK packet from receiver
→ if ACK ("ok") received (*rdt_rcv(rcvpkt && isNAK(rcvpkt)* checks for this) then sender knows that most recent packet was transmitted correctly, so goes back to waiting for data from upper layer
→ if NAK ("pls resend") received as a result of data arriving corrupted,

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

protocol retransmits last packet with *udt_send(sndpkt)* and enters a loop of waiting for an ACK/NAK from receiver, until an ACK is received
→ while '*Wait for ACK or NAK'* loops around, protocol can't get more data from upper level
→ ie: sender doesn't send new data to receiver until it's sure the receiver got the most recent data
→ this is why we call it a stop-and-wait protocol

Receiving side: single state
→ receives packet with *rdt_rcv(rcvpkt)* and ensures *rcvpkt* isn't corrupt with *notcurrupt(rcvpkt)*
→ if one of above checks fail, sends NAK ("pls resend") to sender with *sndpkt=make_pkt(NAK)* and *udt_send(sndpkt)*
→ if both checks pass (ie data arrived and wasn't corrupt), extracts data with *extract(rcvpkt,data)*, delivers the data with *deliver_data(data),* and finally sends an ACK ("ok") with *sndpkt=make_pkt(ACK)* and *udt_send(sndpkt)*

- **But what happens if ACK/NAK becomes corrupted under the stop-and-wait ARQ rdt2.0??:**
  1) Sender can simply re-send data packet if ACK/NAK gets garbled
     → this introduces **duplicate packets** though
     → dest doesn't remember corrupted ACK/NAKs → can't know if an arriving packet is new data, or a retransmission as a result of corrupted ACK/NAK
  2) (Better solution): add new **sequence number** field to data packet
     → dest can check this '*sequence number'* field to ascertain whether a packet is new or a retransmission (from a NAK)
     → **sequence number = 1-bit** → alternates between 0 and 1
     → moves up in increments of modulo 2
     → when a packet with sequence number X has to be retransmitted, the retransmitted packet has sequence number X as well
     → Two consecutive (successfully transmitted) packets will have different sequence numbers
- https://www.youtube.com/watch?v=XLEBBBYnE8U
  - Just watch this video (animation) → illustrates use of sequence numbers amazingly

Rdt2.1: Reliable Data Transfer over a Channel with Bit Errors fixed to account for lost ACK/NAKs

Sender:

Receiver:

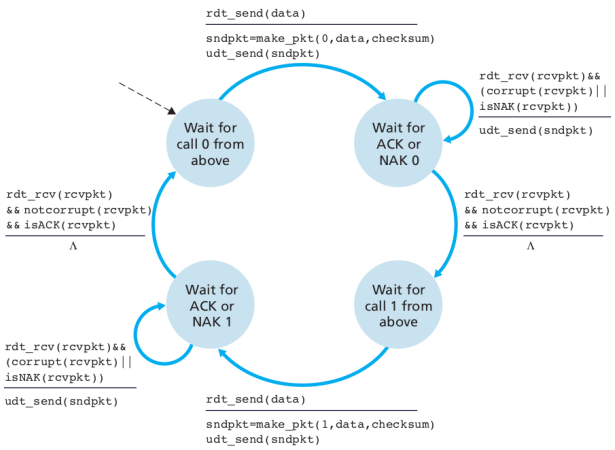


Figure 3.11 ♦ rdt2.1 sender

- Sender and Receiver now have 2 additional states each
    - to reflect whether packet currently being sent (by sender) or received (by receiver) should have a sequence number of 1 or 0
- Receiver now includes sequence number of packet being acknowledged by an ACK message
    - Include *ACK 0* or *ACK 1* in *make_pkt()* in the receiver FSM
    - Include 0 or 1 argument in *isACK()* in the sender FSM

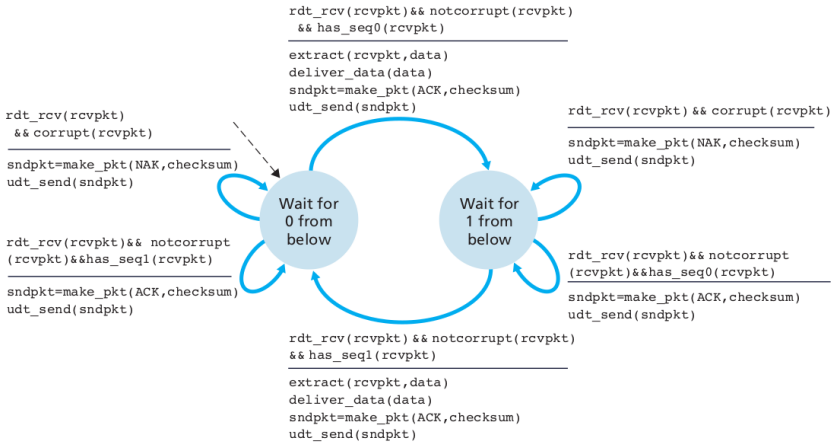


Figure 3.12 ♦ rdt2.1 receiver

Rdt2.2:

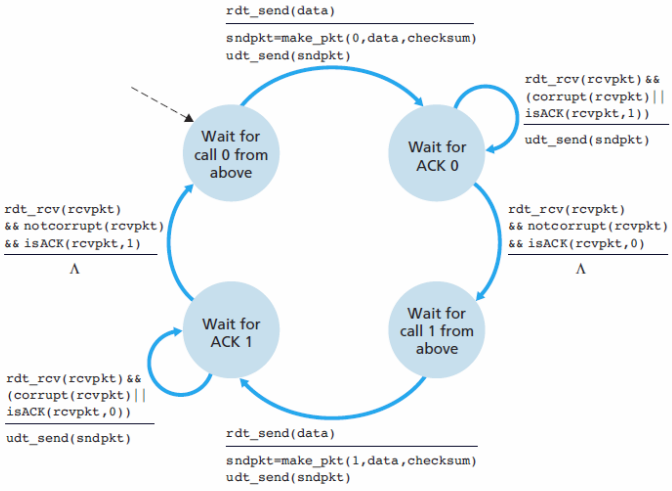


Figure 3.13 ♦ rdt2.2 sender

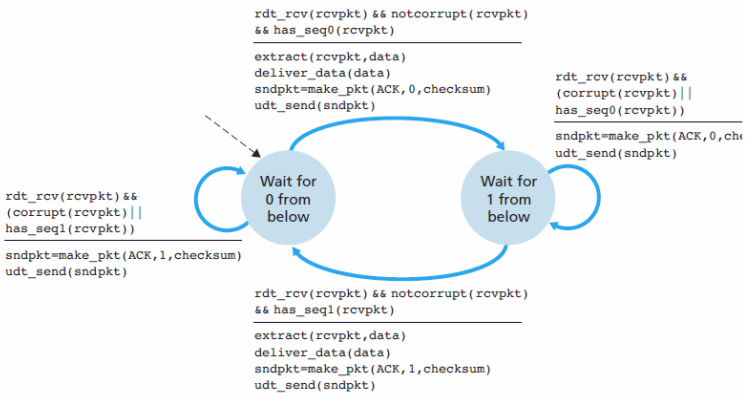


Figure 3.14 ♦ rdt2.2 receiver

Reliable data transfer over Lossy Channel with Bit Errors: rdt3.0

- Now, in addition to corrupting bits, underlying channel can *lose packets* as well
- To handle lost packets, the use of checksum, sequence numbers, ACK/NAKs and retransmission can be used (like in rdt2.2)
- To ***detect*** lost packets, must add a new protocol mechanism
    - = **if a certain amount of time passes without the sender receiving an ACK reply, the sender retransmits the data**
    - Waiting period must be at least as long as round-trip delay between sender and receiver
- **Duplicate packets** can be sent if a packet has a really long delay (ie the packet took so long to send that the sender assumed it got lost, and thus resends a copy of it while the original is still on its way to the receiver)
    - → once again, this can be solved using sequence numbers
    - **https://www.youtube.com/watch?v=XLEBBBYnE8U**
        - The animation shows how sequence numbers can prevent duplicate packets when using the time-based technique
- All sender ever has to do is resend packets when it doesn't receive an ACK reply → protocol handles the rest
    - Sender doesn't know if packet was lost, ACK was lost, if there were delays etc. → all it knows is that it must retransmit if it doesn't receive an ACK when it expects to
- The time-based technique we are discussing is a **countdown timer.**
    - Sender starts the timer each time a (new or duplicate) packet is sent
    - When timer runs out for sender to receive ACK, a *timer interrupt* tells the sender to stop what it's doing and retransmit the packet
    - The sender must be able to stop the timer
- Because packet sequence numbers alternate between 0 and 1, **rdt3.0** is known as the **alternating protocol**

rdt3.0 sender

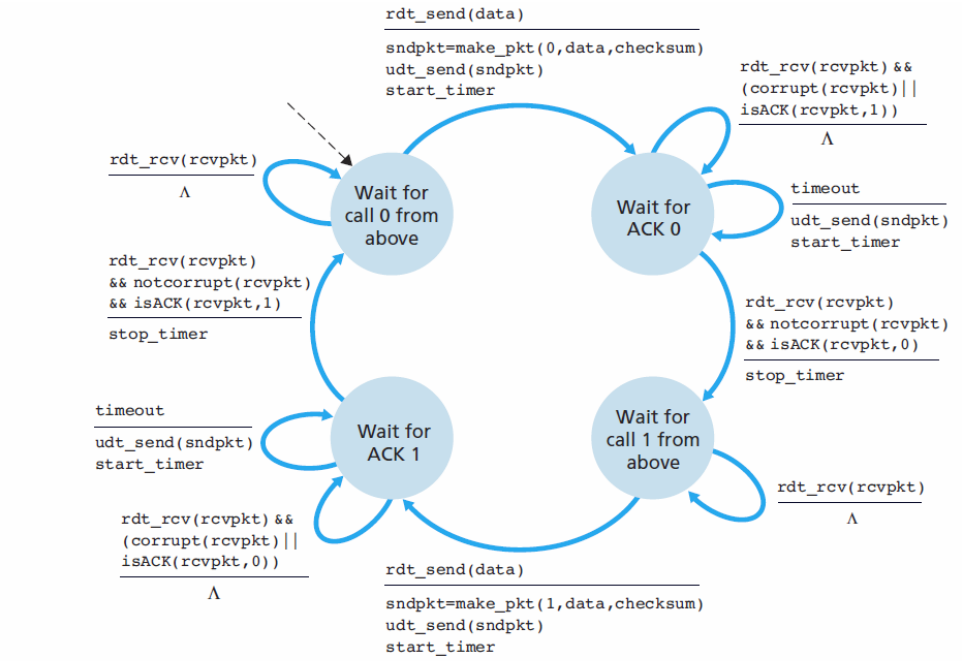
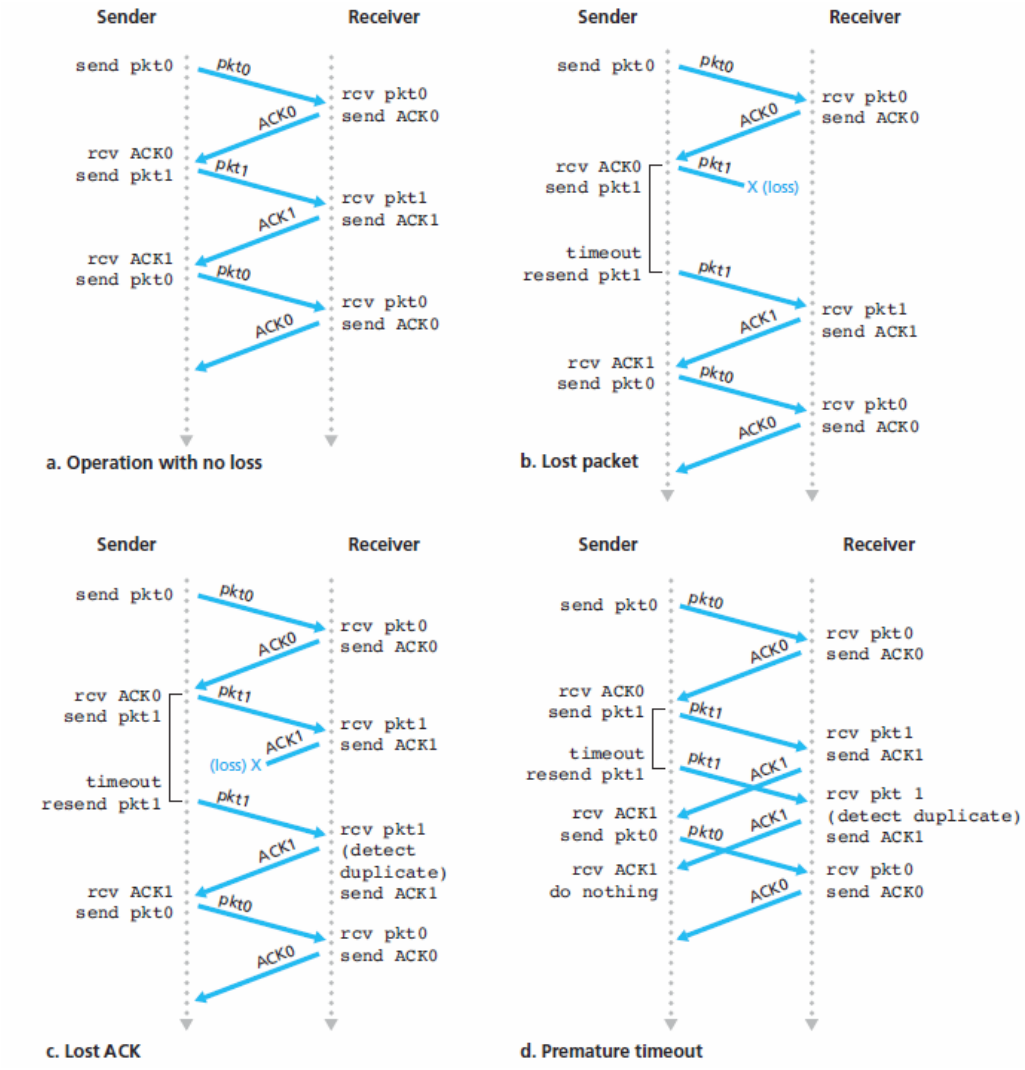


Figure 3.15 ♦ rdt3.0 sender

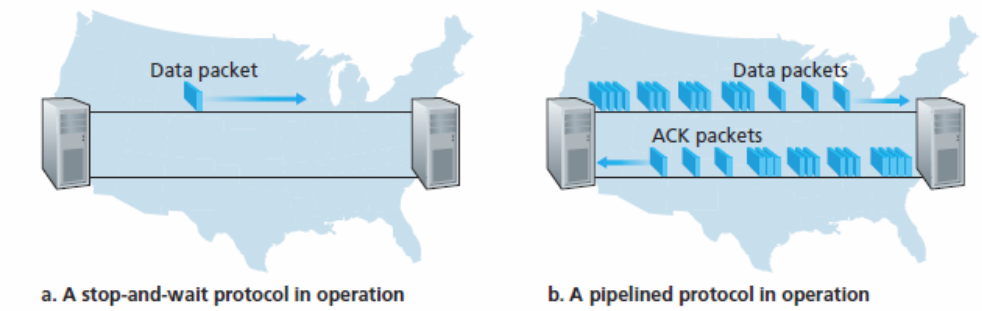
- **rdt3.0 (alternating protocol)** is a working reliable data transfer protocol
  - Checksums
    - Detect whether data arrived corrupted
  - Positive and negative acknowledgements (ACKs and NAKs)
    - Receiver can tell sender whether it received the (in-tact) data or not
  - Sequence numbers
    - Recover from lost/corrupt ACKs/NAKs and from packet loss → allow data to be transmitted until it arrives in-tact, without receiver having to worry about having duplicate data or anything like that
  - Timers
    - Sender uses countdown timer to decide when a (seemingly) lost packet should be retransmitted. If a packet isn't lost but takes longer than the timer period to arrive, causing the sender to retransmit it, the receiver can observe sequence numbers to ignore the retransmitted packet

3.4.2 Pipelined Reliable Data Transfer Protocols

- rdt3.0 (alternating protocol) is reliable but inefficient (since it's a *stop-and-wait* protocol) **Different rdt3.0 protocol scenarios (below):**



- **Pipelining for rdt:**
  - Instead of operating in a stop-and-wait manner, sender is allowed to send multiple packets out without waiting for consecutive acknowledgements



Stop-and-wait protocol versus pipelined protocol (below):

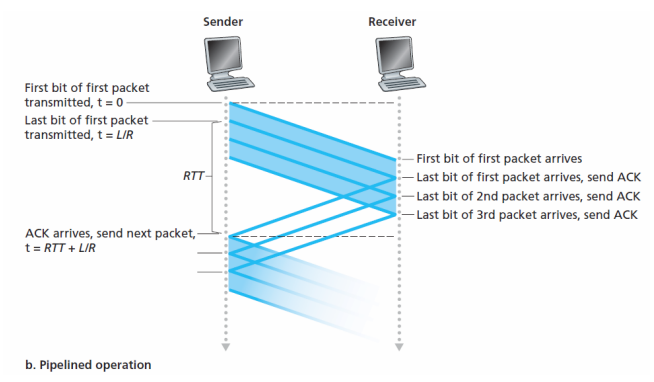
- Pipelining increases throughput a lot when (segment_size / RTD) is small

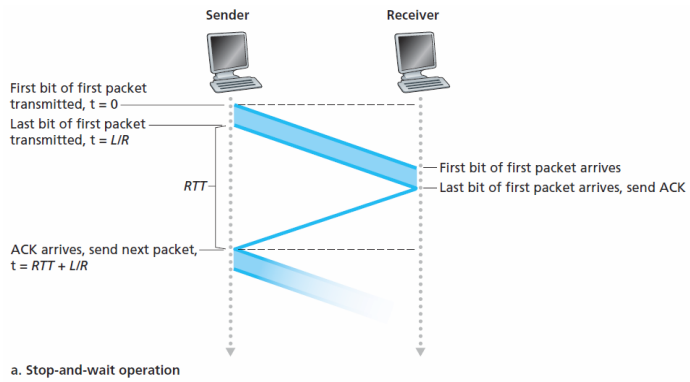
Consequences of pipelining for rdt:

- **Must increase range of sequence numbers**
  - Since each new packet (not counting retransmissions) must have a unique sequence number, and there will often be multiple packets in-transit that need to be uniquely identified to solve any ACK/NAK or packet loss issues they run into
- **Sender and Receiver have to buffer packets**
  - dunno what that is
- **Range of sequence numbers required will depend on mechanisms the protocol uses to resolve lost, corrupted and over-delayed packets**
  - 2 basic approaches to Pipelined error recovery:
    - **Go-Back-N**
    - **Selective Repeat**

3.4.3 Go-Back-N (sliding window protocol)

- Pipelined error recovery approach for rdt
- **Mechanism:**
  - Sender is allowed to transmit multiple packets without waiting for an acknowledgement, but must stop sending packets when there are *N*





unacknowledged packets in the pipiline

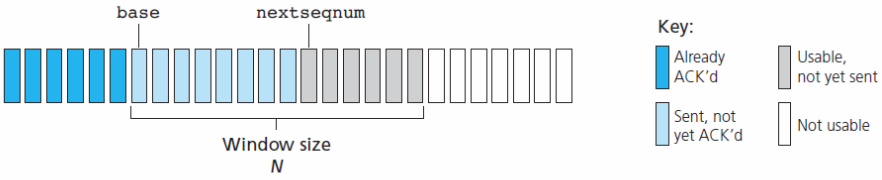


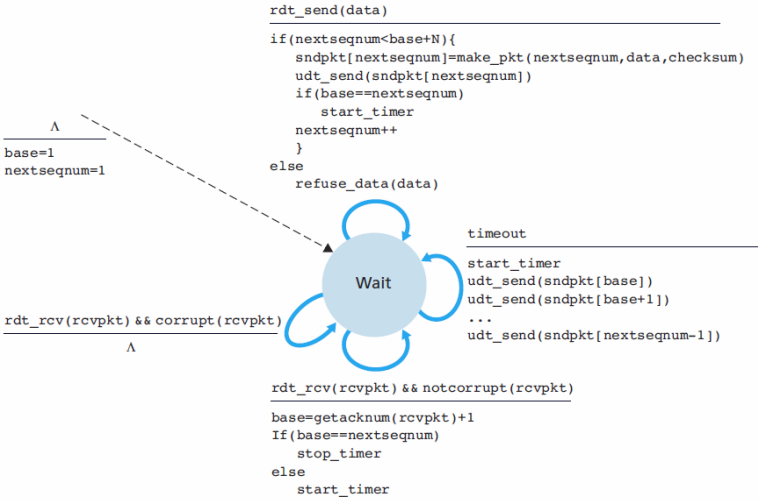
Figure 3.19 ♦ Sender's view of sequence numbers in Go-Back-N

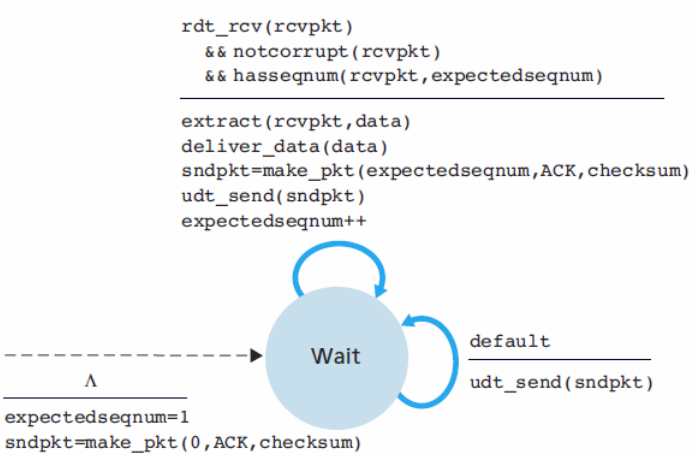
base: sequence number of oldest unacknowledged packet
nextseqnum: smallest unused sequence number (which will be for next packet in-line for sending)

[0, base-1]: transmitted and acknowledged packets
[base, nextseqnum-1]: sent but unacknowledged packets
[nextseqnum, base+N-1]: used for packets that can be sent immediately
Anything >= (base+N): cannot be used until an unacknowledged packet in the pipeline becomes acknowledged (specifically, the packet with sequence number *base*)

- Range of permissible sequence numbers for transmitted but unacknowledged can be viewed as a window of size *N* over the range of sequence numbers (refer to Figure 3.19)
  - As the protocol operates, this window 'slides forward' over the sequence number space
  - Thus, ***N* = window size**
  - **GBN (Go-Back-N) = sliding-window protocol**
- Why limit the number of in-transit but unacknowledged packets?
  - Answer = congestion control (further discussed later)
- In practice, a packet's sequence number is contained in a fixed-length header field
  - If *k* is number of bits sequence number field, the range of sequence numbers for that packet is [0, (2^k) - 1]
    - All arithmetic must thus be done using modulo 2^k arithmetic
- TCP has a 32-bit sequence number field

GBN sender:





GBN receiver:

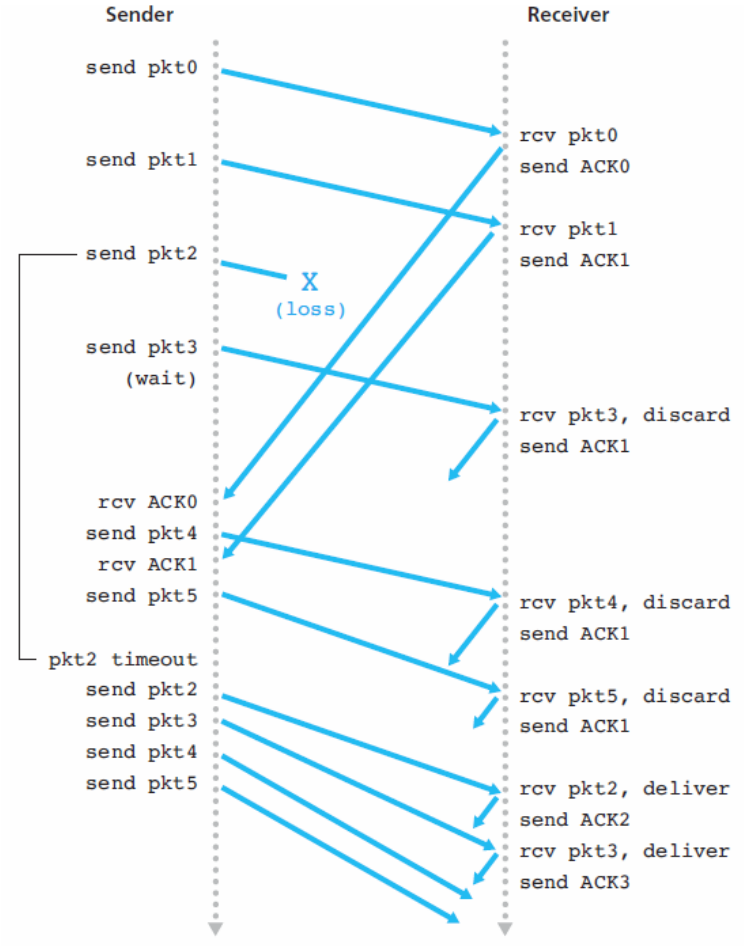
GBN sender must do the following:

- **Check if window is full before sending packets**
  - *Window* = packets in-transit which are unacknowledged
  - If window is full, returns data to upper layer
  - If window is not full, packet is created and sent (goes into the window)
- **Cumulative acknowledgment of ACKs**
  - Acknowledging a packet with sequence number *n* will also acknowledge all packets with sequence numbers up to and including *n*
  - *Cumulative Acknowledgment: the receiver acknowledges that it correctly received a packet, message, or segment in a stream which implicitly informs the sender that the previous packets were received correctly. TCP uses cumulative acknowledgment with its TCP sliding window.*
- **Timeout event to recover lost packets**
  - "Go-Back-N" implies we have to "go back N (window) packets if we lose a packet"
  - If timeout occurs for one packet, sender resends **all** packets that have been sent out but are unacknowledged
  - If an ACK is received but there are still packets unacknowledged packets in-transit, the timer is reset
  - If an ACK is received and there are no unacknowledged packets in-transit, timer is stopped

GBN receiver must do the following:

- 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.

Go-Back-N in action (below - 3.22)



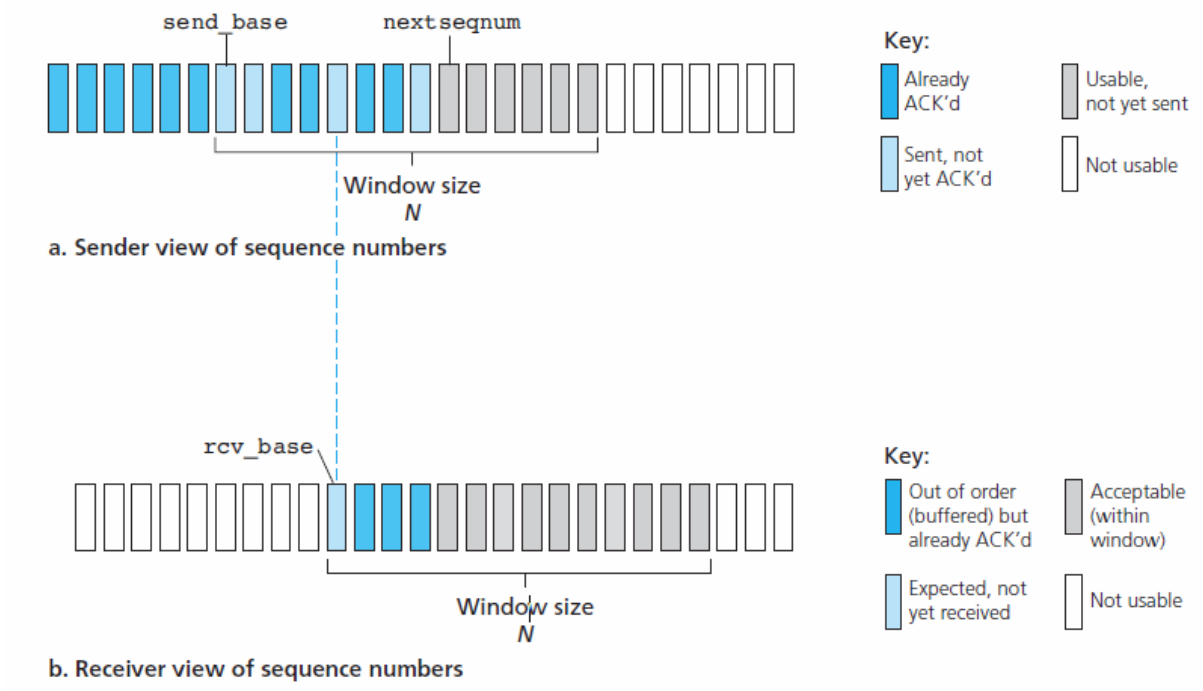
← Notice how, since pk2 was lost, subsequent packets are discarded and re-sent to preserve the correct order of packets being sent

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.

3.4.4 Selective Repeat

- GBN has some performance problems:
    - ie: when there's a really large window size and high chance of packets being dropped/lost, then the entire window has to be re-transmitted each time a single packet is lost
- *Selective Repeat* protocols avoid unnecessary retransmissions by having the sender retransmit only packets that suspects were lost or corrupted
- Since this requires individual retransmission of packets, the receiver must individually acknowledge correctly received packets

- Just like in GBN, a window size of N is used to limit the number of outstanding unacknowledged packets in the pipeline, but **unlike for GBN, the sender will have already received ACKs for some of the packets in the window**



Selective-Repeat (SR) sender and receiver views of sequence-number space (figure 3.23 - below):

- SR receiver will acknowledge a correctly received packet regardless of whether it's in order or not

- Out-of-order packets are **buffered (saved)** until any missing packets (ie: packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer.
- Watch https://www.youtube.com/watch?v=Cs8tR8A9jm8 for great animation of selective-repeat protocol in action
- Basically, with selective-repeat, the sender can continue sending packets if a packet is lost. But it can only send packets that are in the current window. ie: If our window size N is 4 and we have 0, 1, [2, 3, 4, 5], 6, 7, 8
    - 2, 3, 4, 5 are in the window
    - Packet 3 was lost, but 4 and 5 could still be sent to the receiver (and buffered)
    - **(note that the sender doesn't have to send in order, but the receiver must send back ACKs in order)**
    - When the sender receives ACK2, the window changes to 0, 1, 2, [3, 4, 5, 6], 7, 8
        - If 4 instead of 3 had been lost earlier, we could have continued sending packets and moving the window forward. But 3 (the lost packet) is now at the edge of the window, so we have to re-transmit it
        - Once we have re-sent 3 and it arrives safely at the receiver, the receiver can send 3, 4, 5 and 6 from its buffer to the upper layer
- **For selective-repeat, the window size must be less than or equal to half the size of the sequence number space**

Selective-Repeat protocol in action (below):

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

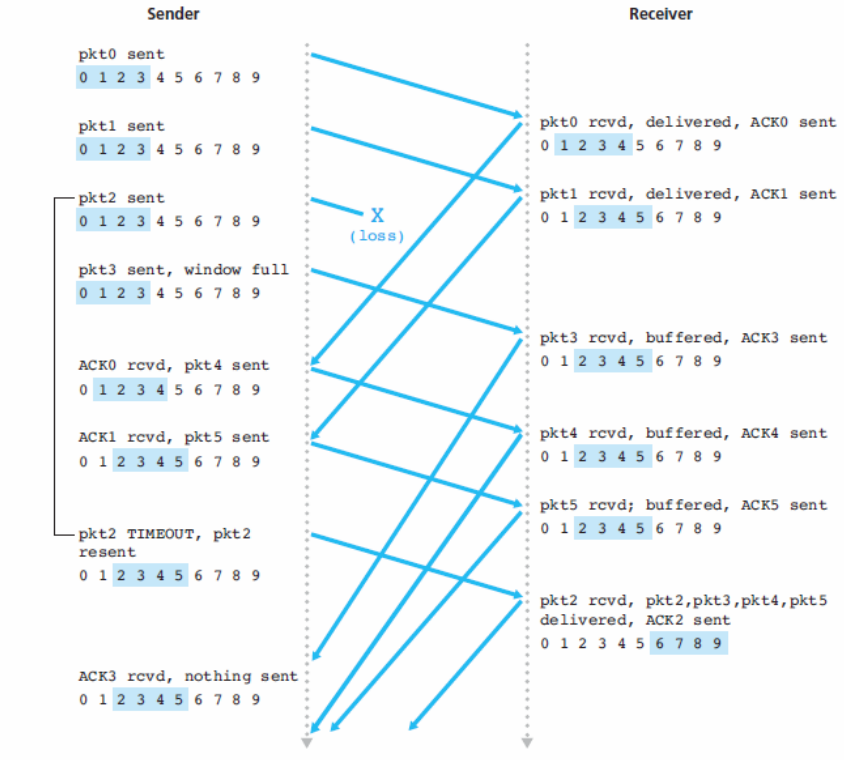
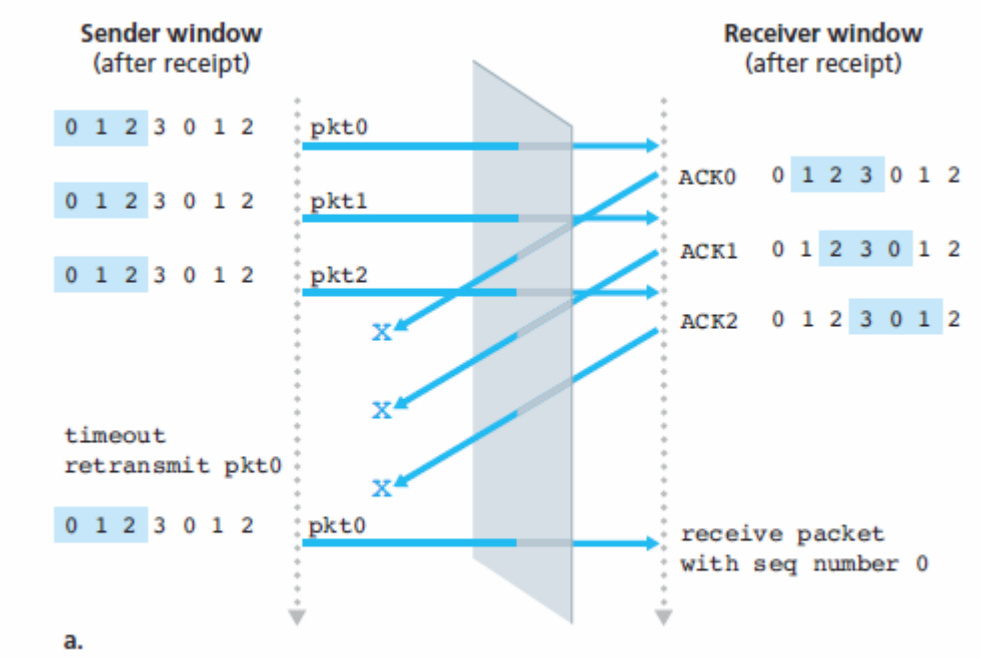


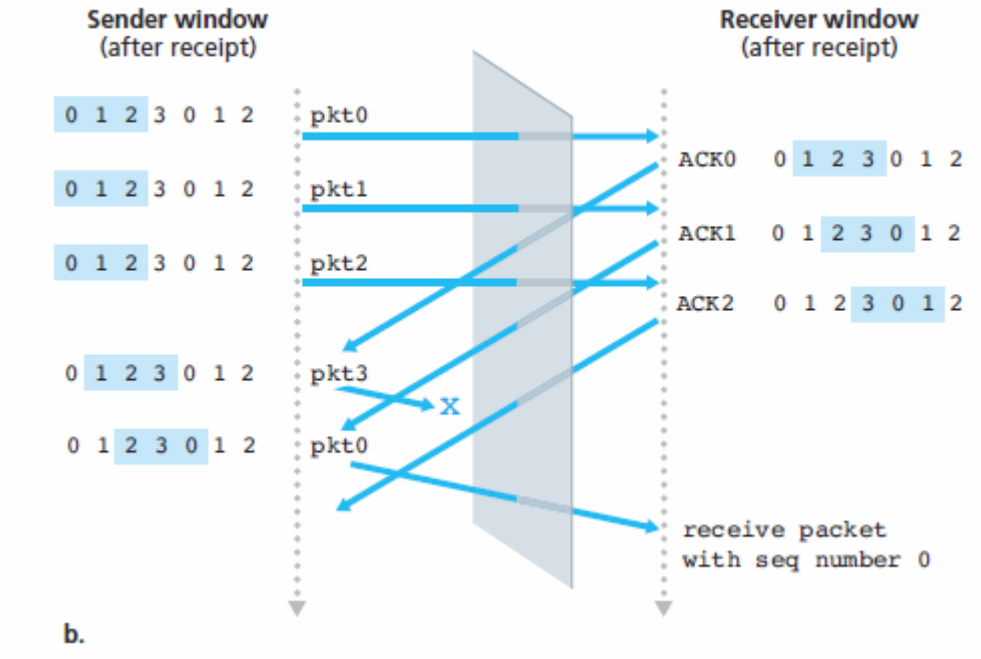
Figure 3.26 ♦ SR operation

Summary of rtd mechanisms and their use (below):

Selective-Repeat Dilemma with too-large windows: New packet or a retransmission?:



a.

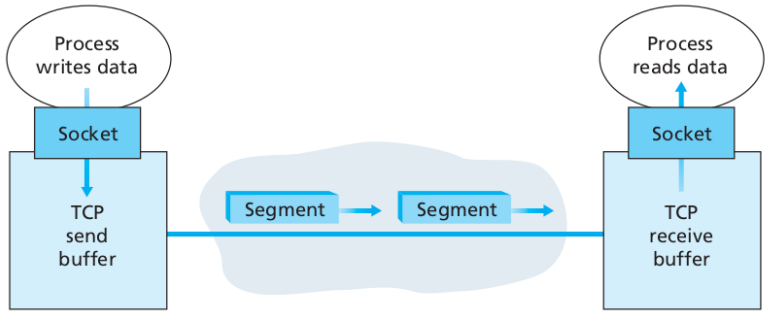


b.

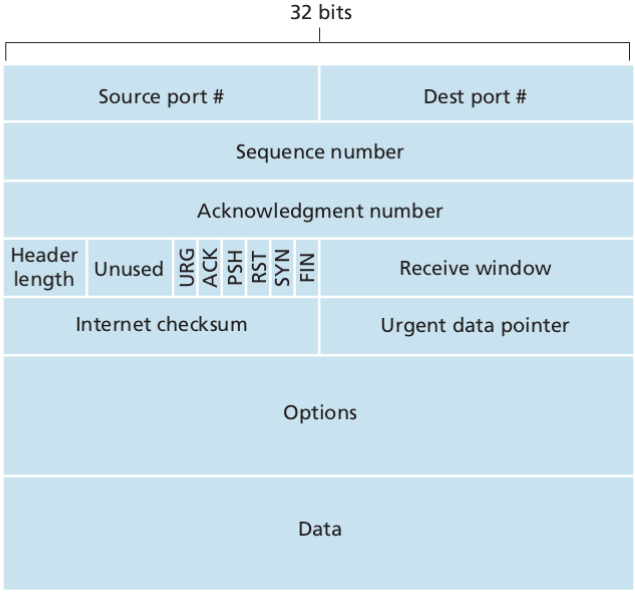
3.5 Connection-Oriented Transport: TCP

- TCP uses many of the underlying principles we have discussed
- TCP = **connection-oriented**
    - *Handshake* must take place before processes can communicate with each other
- TCP only runs in end systems → not in routers/switches
    - routers/switches are oblivious to TCP connections → only see datagrams, not connections
- TCP connection provides **full-duplex** service
    - → if there is a connection between Process A on one host and Process B on another host, then application-layer data can flow from Process A to Process B at the same time as application-layer data flows from Process B to Process A
- TCP connection = **point-to-point**
    - → = between a single sender and a single receiver
    - TCP does not support **multicasting** (transfer of data from one sender to many receivers in a single *send* operation
- **How is a TCP connection established?:**
    - 3-way handshake:
        - Client sends special TCP segment to server
        - Server responds with a second special TCP segment
        - Client responds again with a third TCP segment
    - Processes can now send data to each other
- **How is data sent from client to server?:**
    - Client process passes stream of data through its clientSocket
    - Data now in hands of TCP client-side
    - TCP directs data to connection's **send buffer**
        - (Buffer is setup during 3-way handshake)
    - TCP passes chunks of data from send buffer into network layer
    - Max size of a data chunk that can be placed in a segment for network layer is limited by **Maximum Segment Size (MSS)**
        - **Maximum Segment Size (MSS) =** max amount of application-layer data in the segment
            - Not max size of TCP segment including headers
        - **Maximum Transmission Unit (MTU) =** Largest link-layer frame that can be sent out
    - TCP pairs each chunk of client data with a TCP header to form segments
    - Segments are passed down to network layer → encapsulated within **IP datagrams** → IP datagrams sent into network
- **How is data received?:**
    - TCP receives segment at other end → data placed in connection's **receive buffer**
    - Application reads stream of data from this buffer
- Each side of connection has its own send and receive buffer
- http://www.allwebleads.com/kurose-ross supposedly provides animation of send & receive buffers (try archive.org)

TCP send & receive buffers (below):



3.5.2 TCP Segment Structure

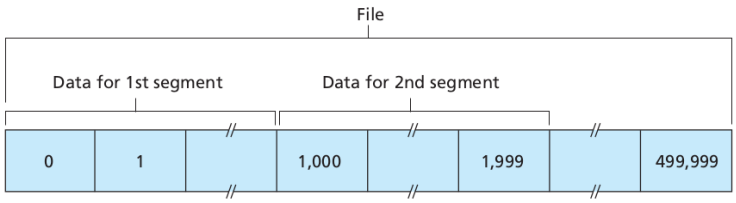


- **Figure 3.29** ♦ TCP segment structure
- Contains header fields & data field
  - Data field contains chunk of application data
  - MSS limits max size of this data field
- When TCP sends a large file (ie: webpage image), typically breaks file into chunks of size MSS
  - Last chunk will often be less than MSS
- Source & Destination port #s used for multi/demultiplexing
- Checksum field for error/corruptness detection
- **32-bit sequence** and **acknowledgement numbers**
  - Discussed further down → for making TCP reliable data transfer service
- **16-bit receive window:** used for flow control → indicates number of bytes
- **4-bit header length field:** specifies length of TCP header in 32-bit words
- **Options field:** used for determining **MSS (Maximum Segment Size)**
- **Flag field:** contains 6 bits
  - ACK bit → value in acknowledgement field is valid
  - RST, SYN, FIN → used for connection setup/closing
  - PSH bit → indicates that receiver should pass data to upper layer immediately

- URG → indicates that sending-side wants this data delivered urgently
  - Red highlighted shit = not used in practice, mentioned for completeness

Sequence Numbers and Acknowledgement Numbers (in TCP)

- Essential for reliable data transfer
- TCP views data as an unstructured but ordered stream of bytes
- **Sequence number for a segment =** byte-stream number of first byte in segment
  - segment. Let's look at an example. Suppose that a process in Host A wants to send a stream of data to a process in Host B over a TCP connection. The TCP in Host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0. As shown in Figure 3.30, TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.



  - **Figure 3.30** ♦ Dividing file data into TCP segments
- **Acknowledgement number:**
  - *Recall*: TCP is **full-duplex** → host A can send to and receive from host B simultaneously
  - Each segment that arrives from host B has a sequence number
  - **The Acknowledgment Number that Host A puts in its segment is the sequence number of the next byte expected from Host B**
    - to look at a few examples to understand what is going on here. Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B.
  - **TCP provides *cumulative acknowledgments***
    - → only acknowledges bytes up to first missing byte in stream
      - (ie stops acknowledging new bytes as soon as one goes missing)
    - As another example, suppose that Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899. In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream. Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments**.
    - (above example) brings up NB issue:
      - Host A received the third segment before the second one

        - So the third segment arrived out of order
        - But TCP needs data to be ordered before the receiver can pass it to upper layer. Solution?
            - **Solution (1)**: receiver immediately discards out-of-order segments, sender can't send new data till order is correct
            - **Solution (2)**: receiver keeps out-of-order bytes and waits for missing bytes to fill in gaps
                - This is used in practice. Choice is up to the programmer though.
- **Note**: both sides of a TCP connection randomly choose initial sequence numbers (0 or 1)
    - This minimises possibility that a segment (which is present in the network from a previous connection with same port #) is mistaken for a valid segment in the current connection

Telnet: A case study…

- **= OPTIONAL … skipping it**

3.5.3 Round-trip Time Estimation and Timeout

- TCP, like *rdt* which we built with FSMs, uses a timeout/retransmit mechanism to recover from lost segments,
- *Big question*: what should length of timeout be?
    - Should certainly be larger than connection's **Round-Trip-Time (RTT)**
        - *Recall* RTT = time from when a segment is sent until it is acknowledged
        - ie: we have to at least wait for the expected amount of time it takes for data to arrive before we decide it's taking too long
    - So we need to know how to estimate the RTT before we can define length of timeout

Estimating the Round-Trip Time:

- Let *SampleRTT* be the amount of time between when the segment is sent (passed to IP) and when an acknowledgement is received
- TCP doesn't measure a *SampleRTT* for every transmitted segment
    - → at any point in time, the *SampleRTT* is being estimated for only one of the transmitted but unacknowledged segments
    - → new value of *SampleRTT* once every RTT
- Also, TCP never computes a *SampleRTT* for a segment that has been retransmitted
    - → only measures *SampleRTT* for segments that have been transmitted once

- *SampleRTTs* will fluctuate from segment to segment (due to congestion in routers, etc.) → so TCP maintains an average called *EstimatedRTT*

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

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:

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

- EstimatedRTT = **exponential weighted moving average (EWMA)** of SampleRTT values

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 *EstimatedRTT* and *DevRTT*, what should we set TCP's timeout interval to?
- Timeout interval should be >= *SampleRTT,* but not too much bigger
  - → We want TCP to retransmit the segment soon after *SampleRTT* has elapsed
  - Should set timeout = *EstimatedRTT* + *margin*
  - If *SampleRTT* is fluctuating a lot (ie: if *DevRTT* is high) then *margin* = large
  - If *SampleRTT* is not fluctuating a lot (ie: if DevRTT is low) then margin = small

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

- 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.

Checkpoint: *Let's discuss how TCP provides reliable data transfer:*

- Acknowledges correctly received data
- Retransmits lost/corrupted segments
- Some versions have an *implicit NAK mechanism*
  - → 3 duplicate ACKs for one segment = NAK for that segment
  - The 3 ACKs are sent super fast

- Sequence numbers allow receivers to identify lost /duplicate segments → sender retransmits if so
- Uses pipelining → sender can have multiple transmitted but not-yet-acknowledged segments outstanding at any given time
    - Increases throughput a lot when (segment_size / RTD) is small
    - Flow-control and congestion-control determines how many outstanding segments the sender can have

3.5.5 Flow Control

- Hosts on each side of TCP connection have a **receive buffer**
    - → when TCP connection receives bytes that are correct and in order, places in **receive buffer**
- Application process reads data from receive buffer if it isn't busy with something else
    - → so if too much data is going to receive buffer and app process is busy with other stuff, buffer can overflow
- **Flow-control service:** TCP provides to application process to prevent buffer overflow
    - → = *speed-matching service*
        - Rate at which sending app is sending must be matched with rate at which receiving app is **reading**
- **Congestion control:** TCP throttles sender based on degree of congestion in network
    - → similar to flow control (throttling sender) but used for different reasons

How does TCP provide flow control service?

- *(note: for this discussion, assume that out-of-order segments are discarded)*
- Sender maintains **receive window** variable
    - *Receive window* gives sender an idea of how much free buffer space is available at receiver.
- Since TCP is full-duplex (both sides send & receive), sender at each side of the connection has a *receive window* variable
- Suppose Host A is sending large file to Host B over TCP connection:
    - Host B creates receive buffer *RcvBuffer*
    - App. process in B reads from *RcvBuffer* every so often
    - *LastByteRead* = number of last byte read from the receive buffer by B
    - *LastByteRcvd* = number of last byte that has been placed in receive buffer at B

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

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

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

      $$\texttt{rwnd} = \texttt{RcvBuffer} - [\texttt{LastByteRcvd} - \texttt{LastByteRead}]$$

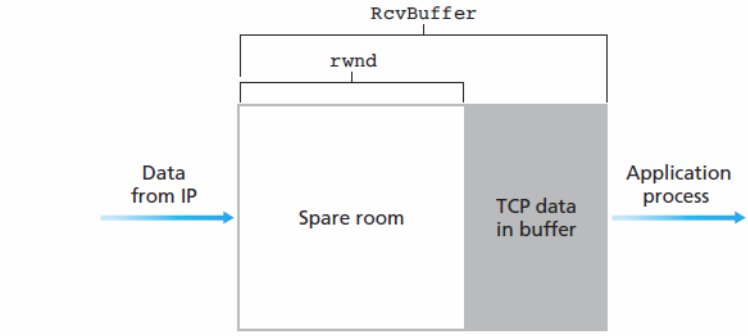


Figure 3.38 ♦ The receive window (`rwnd`) and the receive buffer (`RcvBuffer`)

- So how does *rwnd* (the receive window) provide flow control?
  - → Host B tells A how much space it has in its receive buffer by placing the current value of *rwnd* in in the *receive window* field of each segment it sends to A
  - Initially, *rwnd* = *RcvBuffer* (the window can be the entire buffer space, since no data received yet)
  - **(*LastByteSent - LastByteAcked)* = amount of unacknowledged data that A has sent into connection**
    - Host A must keep LastByteSent - LastByteAcked <= rwnd
      - → amount of unacknowledged data in connection cannot be more than the window of space in A's receive buffer (to avoid overflow)

One problem with TCP's flow-control:

- Suppose Host B has no data to send to Host A
- Also Suppose B's receive buffer becomes full, so rwnd = 0 (no free space in window)
- Since the data arrived safely and in order, B will send an ACK to A
  - The segment containing the ACK will have a header field stating that rwnd = 0
- When B's application process empties/reads the receive buffer (ie: freeing up space in the buffer), it never sends the header field to A which states that some buffer space became available
  - (since B has no data to send, and since it already sent the last necessary ACK)
- So A is blocked from ever knowing that B has space in its receive buffer, and thus never sends more data to B
- **The fix?:**
  - TCP requires A to continue sending 1-byte data segments to B, since this forces B to continuously send back ACKs, and these ACKs contain the header field stating how much space is in B's receive buffer

3.5.6 TCP Connection Management

- *We now discuss how a TCP connection is established and torn down*
- Suppose a client process wants to initiate TCP connection with a server process

- → TCP in client establishes connection with TCP in server as follows (**three-way handshake)**:
    1. **Client sends SYN:** Client-side TCP sends special TCP segment (the SYN segment) to server, with SYN bit in a header field
       → Client randomly chooses initial sequence number *client-isn* and puts *client-isn* in the sequence number field of the initial SYN segment .
       → SYN segment encapsulated in IP datagram and sent to server
    2. **Server sends SYNACK:** IP datagram containing TCP SYN segment arrives at server.
       → server extracts segment from datagram, allocates TCP buffers and variables to connection, sends *connection-granted segment* (SYNACK segment) to client TCP
       → in SYNACK segment:
            SYN bit field = 1
            ACK field = *client_isn + 1*
            Sequence # field = *generate_server_isn()*
       → essentially, SYNACK segment says: I got your SYN packet and your *client_isn*, and I agree to establish this connection. My initial sequence number is *server_isn.*
    3. **Client acknowledges SYNACK (with an ACK):**
       → allocates buffers and variables for connection
       → sends one more segment to server, indicating that it acknowledges the server's granting of the connection request
       → in this segment:
            ACK field = *server_isn + 1*
            SYN bit = 0
       → note: this last segment can carry normal data in its payload

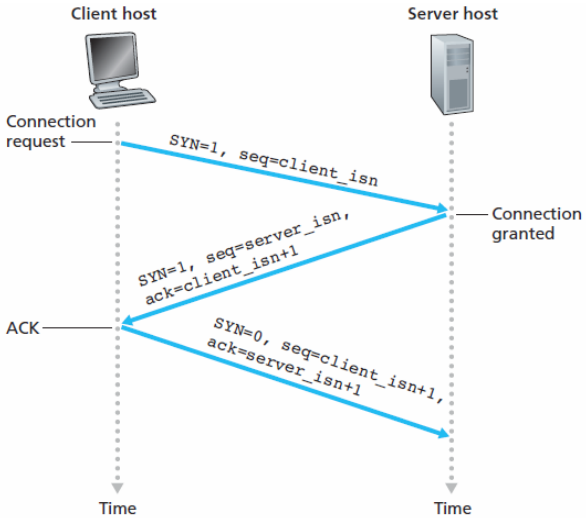


Figure 3.39 ♦ TCP three-way handshake: segment exchange

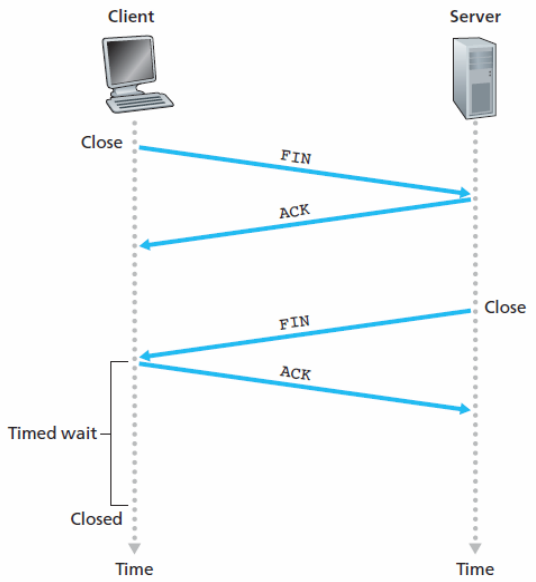


Figure 3.40 ♦ Closing a TCP connection

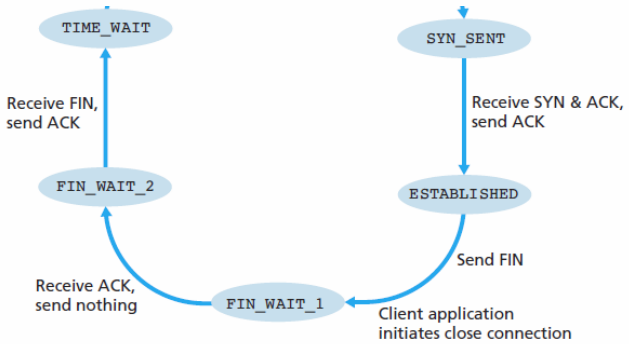


Figure 3.41 ♦ A typical sequence of TCP states visited by a client TCP

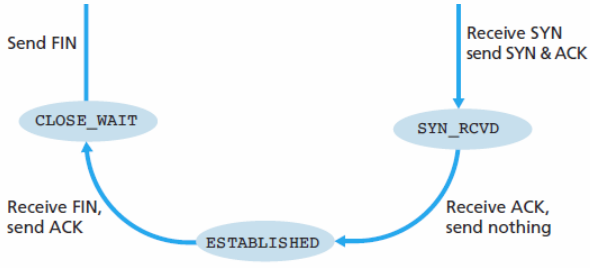


Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

- A legitimate client will return an ACK segment. When the server receives this ACK, it must verify that the ACK corresponds to some SYN sent earlier. But how is this done if the server maintains no memory about SYN segments? As you may have guessed, it is done with the cookie. Recall that for a legitimate ACK, the value in the acknowledgment field is equal to the initial sequence number in the SYNACK (the cookie value in this case) plus one (see Figure 3.39). The server can then run the same hash function using the source and destination IP address and port numbers in the SYNACK (which are the same as in the original SYN) and the secret number. If the result of the function plus one is the same as the acknowledgment (cookie) value in the client's SYNACK, the server concludes that the ACK corresponds to an earlier SYN segment and is hence valid. The server then creates a fully open connection along with a socket.
- On the other hand, if the client does not return an ACK segment, then the original SYN has done no harm at the server, since the server hasn't yet allocated any resources in response to the original bogus SYN.

- **What happens when a host receives a TCP segment with the wrong port number or source address?**
- Consider the following: host receives TCP SYN packet with dest port 80, but it isn't accepting connections on port 80 (ie: isn't running a webserver on port 80)
- → host will send special *reset segment* to the source
- → *reset segment* has *RST flag bit = 1*
    - → this tells source not to resend the segment
- *Similarly, in the case of UDP, the host sends a special ICMP datagram to the source*

3.6 Principles of Congestion Control

- So we know that packet loss typically occurs as a result of router buffer overflows when the network is congested. We also know that packet retransmission can help to ensure the packet arrives.
- But packet retransmission doesn't fix the core issue behind packet loss, which is network congestion.
    - We need mechanisms to throttle senders when there is network congestion.
- **Congestion control is different to flow control -- explain why**

3.6.1 The Causes and the Costs of Congestion:

- We will discuss 3 different scenarios.

Scenario 1: *Two senders, one router with infinite buffers*

- Hosts A and B each have a connection that shares a single hop/link between source and dest

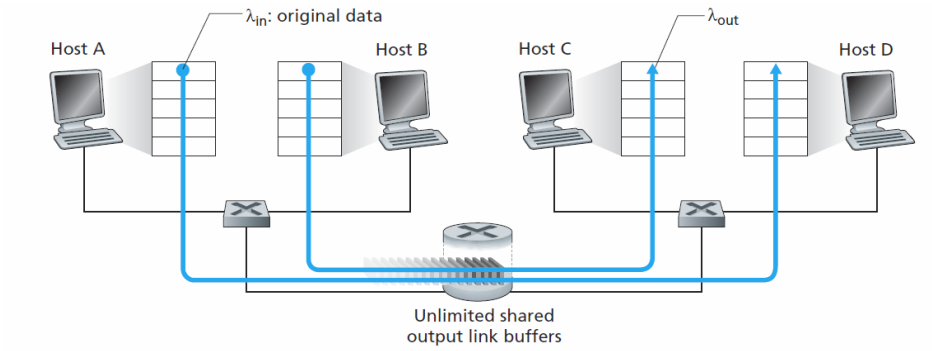


Figure 3.43 ♦ Congestion scenario 1: Two connections sharing a single hop with infinite buffers

- Apps in A and B send data into connection at **λ_{in} bytes/sec.**
- Connection has no error recovery (ie: retransmission), flow control or packet control
- We ignore overheads from header fields, and router has infinite buffer space
- Router uses its buffers to store incoming packets when the packet arrival rate exceeds the outgoing link's capacity (= *R*)

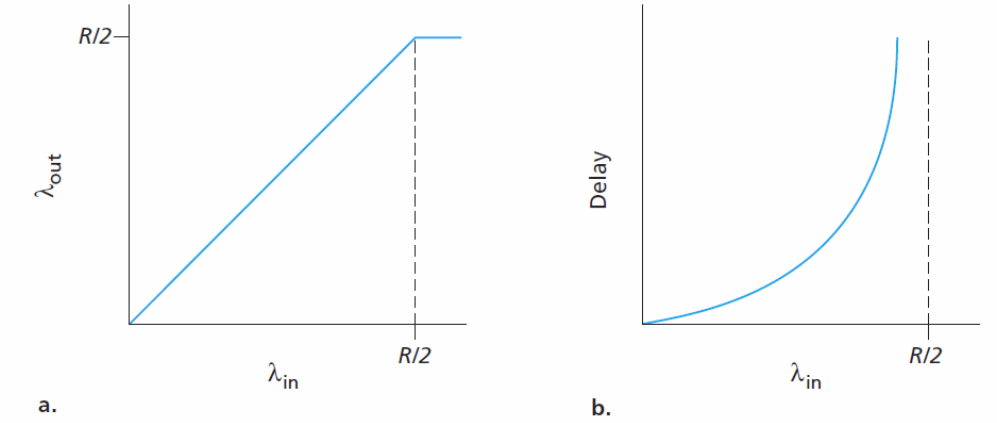
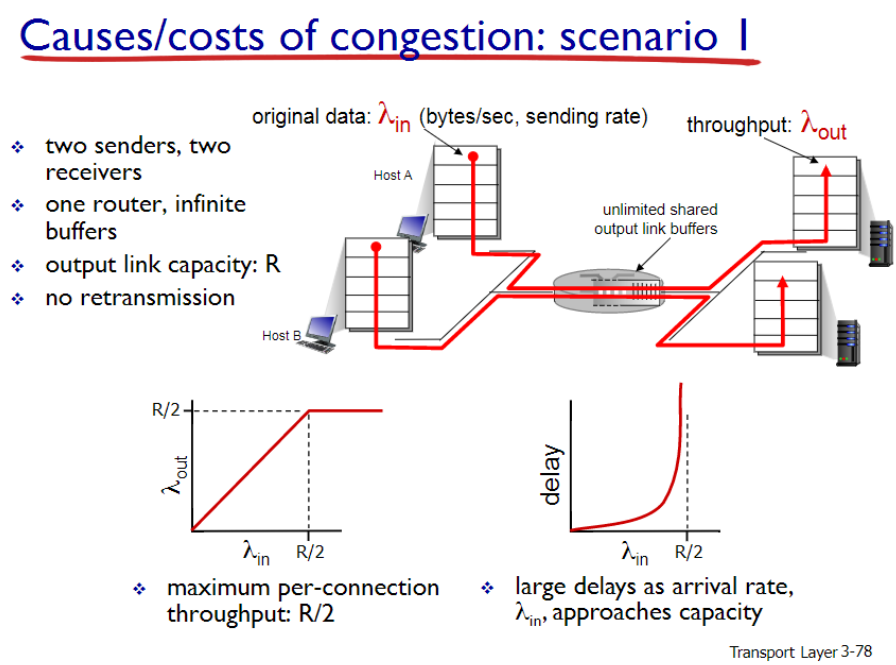


Figure 3.44 ♦ Congestion scenario 1: Throughput and delay as a function of host sending rate

- **(sending rate = λ_{in})** *(= rate at which app sends original data into socket)*
- **Left graph (a):** <u>per-connection throughput</u> (bytes/second at receiver) as a function of connection-sending rate
  - For (*0* < *connection sending rate* < *R/2*), throughput at receiver = sender's sending rate.
  - ie: packets are received with a finite delay
  - But for (*connection sending rate* > *R/2*), throughput = *R/2*
  - No matter how high A and B set their sending rates, their throughput will always max at *R/2*
- **Right graph (b):** <u>delay</u> as a function of connection-sending rate
  - Delay increases gradually as sending rate approaches *R/2*
    - But limit as sending rate tends to *R/2* is infinity
- *As packet-arrival rate approaches link capacity → large queueing delays*



Scenario 2: *Two senders and a Router with Finite Buffers*

- Router no longer has infinite buffer capacity. This will obviously negatively impact throughput.
- If a packet is dropped as a result of the router buffer being full, sender will need to retransmit it
- **λ'in bytes/sec** = rate at which app sends original data into socket
- **Offered load = λ'in bytes/sec** = rate at which app sends segments containing original *and* retransmitted data into socket

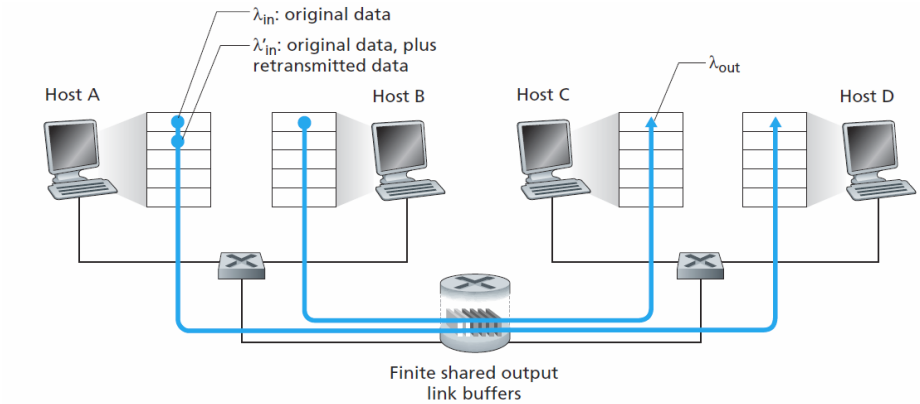
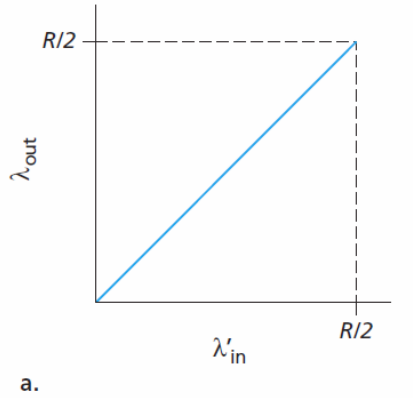


Figure 3.45 ♦ Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

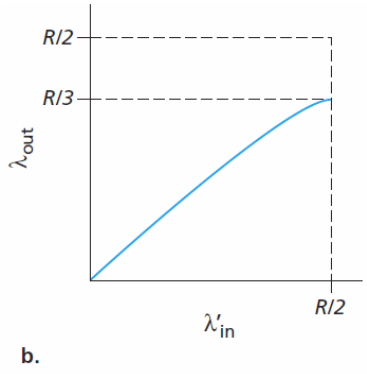
Performance of scenario 2 (finite buffers) strongly depends on how retransmission is handled → Consider three examples of handling retransmission (below):

Scenario 2 Example 1 throughput: perfect knowledge



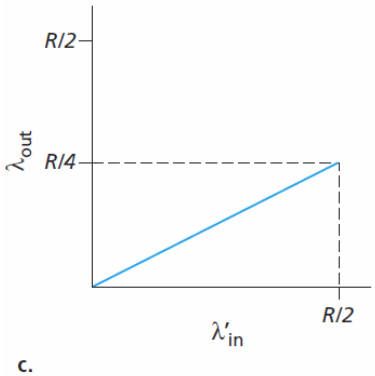
← Host A has perfect knowledge of whether or not router buffer is free, so sends data only when buffer is free
← no loss, since A only sends data when the router buffer has enough space to not have to drop packets
← since no retransmissions necessary, **$\lambda_{in} = \lambda'_{in}$**
← host sending rate (x-axis) cannot exceed *R/2*, since packet loss is assumed never to occur

Scenario 2 Example 2 throughput: retransmit if packet lost



← sender retransmits only when it's 100% sure the packet was lost
← offered load (rate of original + retransmissions) = *R/2*
← at the offered load rate, throughput to B is *R/3*
← *sender must perform retransmissions to compensate for dropped/lost packets due to buffer overflows*

Scenario 2 Example 3 throughput: premature timeout



← sender may timeout prematurely & retransmit a packet that is in the router buffer (and not lost)
← original & retransmission may reach receiver = duplicate packet sent = waste of potential throughput
← graph shows throughput vs offered load when every packet has an extra copy sent
← since each packet forwarded twice, throughput tends to *R/4* as offered load approaches *R/2*
← *Unneeded retransmissions by sender may cause duplicate packet sending, which is a waste of potential throughput*

Scenario 3: *Four senders, Routers with Finite Buffers, and Multihop paths*

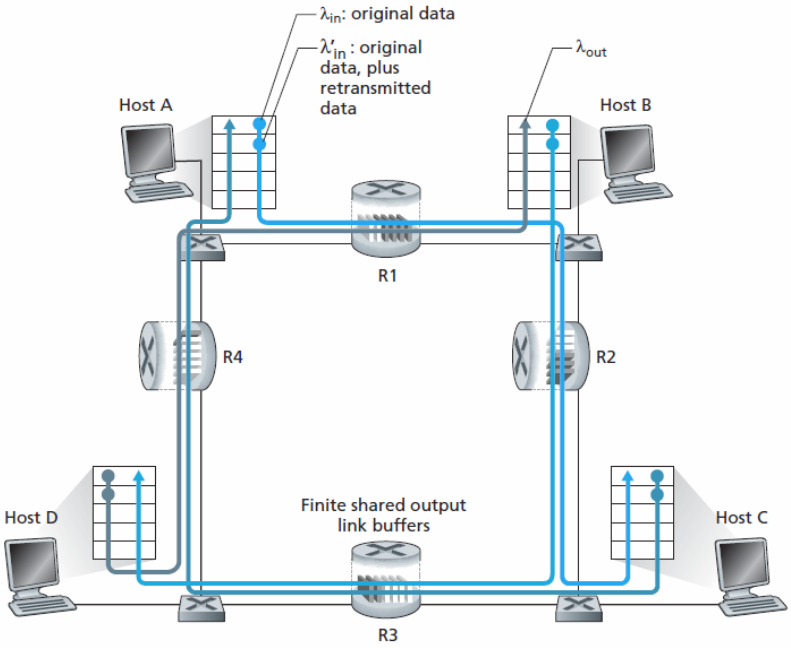


Figure 3.47 ♦ Four senders, routers with finite buffers, and multihop paths

← 4 hosts, each overlapping 2-hop paths
A-C connection shares router R1 with D-B connection and R2 with B-D connection:
Low traffic:
← as always, for tiny values of **λin,** buffer overflows are rare & throughput = offered load
← for small values of **λin,** increase in **λin** = increase in throughput (**λout**), since more original data is being received and few overflows
High traffic:
← A-C's arrival rate at R2 is constrained by the capacity *R* of R1, which it has to pass through first.
← A-C and B-D have to compete for buffer space at R2. Since B-D doesn't have to pass through R1 to get to R2, if **λin** is extremely big then A-C will get held up at R1 and B-D will constantly fill up the buffer at R2, blocking A-C from R2.
← Thus, *A-C end-to-end throughput goes to 0 in heavy traffic*

- **Reason for huge decrease in throughput when offered load increases →**
  - = A lot of wasted work when packets are dropped at routers after having already passed through prior routers which could have served other packets instead.
  - Might be better for routers to give priority to packets that have already traversed some number of routers
- *When a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward the packet to the point at which it was dropped ends up having been wasted*

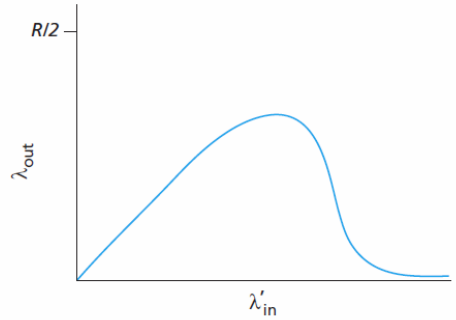


Figure 3.48 ♦ Scenario 3 performance with finite buffers and multihop paths

3.6.2 Approaches to Congestion Control

- ***End-to-end congestion control***
  - Network layer gives provides no information regarding congestion state in network
  - Used in TCP:

        - TCP on each end-system adjusts window size based on packet loss and delay
- ***Network-assisted congestion control***
    - Network layer (routers) provides explicit feedback to senders regarding congestion state in network
    - Can be as simple as a single bit indicating congestion at a link
    - Network provides this feedback to senders in one of two ways:
        - **Method 1:**
            - Direct feedback from router to sender
            - Router sends **choke packet** with message "I'm congested!" back to the sender before forwarding the original segment to the receiver
        - **Method 2:**
            - Router marks/updates 'congestion field' in headers of packets flowing through it
            - Host A → Router R1 → Host B
                - R1 marks the packet sent from A with 'congested'.
                - B receives the packet, sees R's note about congestion, and must now inform A of the congestion by sending a message.
                - Requires at least a full round-trip time (RTT)

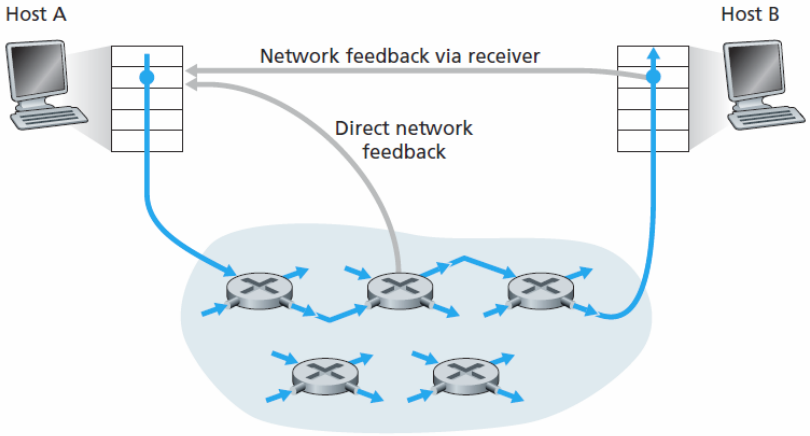


Figure 3.49 ♦ Two feedback pathways for network-indicated congestion information

3.6.3 → OPTIONAL → SKIP

3.7 TCP Congestion Control

- TCP must use end-to-end congestion control, since the IP (network) layer does not give explicit 'congestion feedback' to sender hosts.
- Note: *cwnd* = congestion window
- *TCP approach:*
    - Each sender limits rate at which it sends data through socket, based on perceived network congestion
    - If sender perceives uncongested network → increases send rate
    - If sender perceives congested network → decreases send rate
- Must decide:
    - **(1) How TCP sender limits send rate**
    - **(2) How TCP sender infers/perceives congestion**
    - **(3) How does TCP sender limit send rate *based on congestion*?**

(1) How TCP sender limits send rate

- Sender keeps track of **congestion window** variable (in addition to buffers & other vars)
- **congestion window** constrains send rate by limiting allowed amount of unacknowledged data
    - recall: *unacknowledged data* = *LastByteSent - LastByteAcked*
    - Window requires that *unacknowledged data* <= min{*cwnd, rwnd*}

(2) How TCP sender infers/perceives congestion

- Let's define a packet 'loss event' as:
    - (a) sender's timer waiting for ACK runs out (timeout)
    - (b) sender receives a burst of 3 ACKs from receiver
- If ACKs arrive at slow rate → window - -
- If ACKs arrive at high rate → window + +
- Thus TCP is **self-clocking →** uses acknowledgments to adjust/clock its increase in congestion window size

(3) How does TCP sender choose send rate *based on congestion*?

- So we know how senders can recognise congestion, and how they can adjust their send rate. But how are (1) and (2) combined to determine the optimal send rate taking into account network congestion?
- Answer = the following:
    - *Decrease send rate in the event of a lost segment, since a lost segment implies congestion.*
    - *Increase send rate when an ACK arrives (not a triple ACK!), since this implies the network is not dropping packets due to congestion.*
    - *Bandwidth probing* → incrementally increase send rate, but back off when we notice it's causing congestion.

We now know enough to discuss the **TCP congestion-control algorithm.**

TCP congestion-control algorithm

- 3 big components:
    - (1) slow start
    - (2) congestion avoidance
    - (3) fast recovery (recommended but not required)

Slow start

- Connection begins → *cwnd* = 1 MSS (Max Segment Size)
- Thus initial sending rate = MSS/RTT
    - ie: MSS = 500 bytes, RTT = 200msec → initial send rate ~ 20kpbs
- *cwnd* increases by 1 MSS for each ACK
    - → ie: send rate doubles every RTT
- So TCP send rate starts slow but grows exponentially

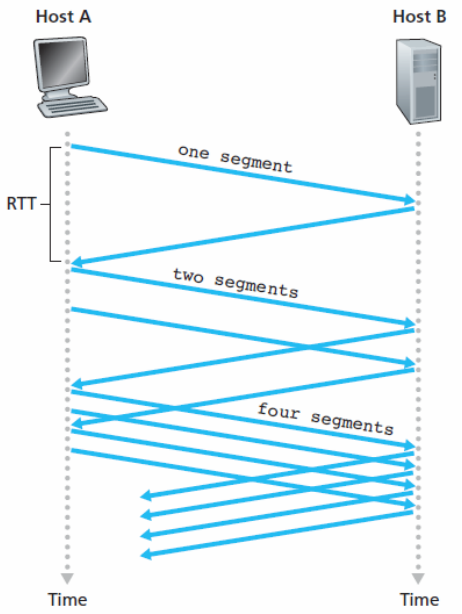


Figure 3.51 ♦ TCP slow start

← **illustration of TCP slow-start**

← TCP sends first segment into network, waits for ACK

← ACK arrives, sender increases *cwnd* by 1 MSS

← *cnwd* now fits 2 segments, so 2 segments sent out

← *cwnd* +=1 for each ACK from previous 2 segments
← *cwnd* now fits 4 segments, so 4 segments sent out

← etc…

- **When does slow-start's exponential growth end?:**
  - (1) if loss event → *cwnd = 1* MSS (slow-start ends and window growth starts from scratch)
    - also, *ssthresh (*slow-start threshold) set to *cwnd/2* (half the value of the congestion window when loss event took place)
  - (2) if *cwnd == ssthresh* → slow start ends
    - **enter congestion avoidance mode** (*cwnd* ++ with caution)
  - (3) if three duplicate ACKs detected (==NAK) → slow-start ends
    - Retransmit performed
    - **enter fast recovery mode**

Congestion Avoidance

- = state of conservatively increasing *cwnd* → activated when *cwnd == ssthresh*
  - ie: when the current *cwnd value* is half of the *cwnd* value which last caused congestion (/ loss event)
  - at *cwnd == ssthresh,* we know we're approaching congestion, so activate **congestion avoidance** mode
- **Congestion avoidance mode** increases *cwnd* by just 1 MSS each RTT
  - Does this by increasing *cwnd* by *MSS bytes (MSS/cwnd)* for each ACK received at sender
    - ie:
      - MSS = 1460 bytes
      - *cwnd* = 14600 bytes
      - So *cwnd* can fit 10 segments at a time
      - The ACK received from each of the 10 segments increases *cwnd* by (*MSS/cwnd* = 1/10), so *cwnd* will have increased by 1 after all 10 segments are ACK'd
    - = linear increase

- **When does congestion avoidance mode's linear growth end?:**
    - (1) if loss event → *cwnd* = *1* MSS (window growth starts from scratch)
        - also, *ssthresh (*slow-start threshold) set to *cwnd/2* (half the value of the congestion window when loss event took place)
    - (2) if three duplicate ACKs detected (== NAK)
        - → *new_cwnd* = *cwnd/2* + 3 MSS
            - '+3 MSS' to account for the 3 ACKs
        - *ssthresh* = *cwnd/2*
        - **Fast recovery state entered**

Slow-start mode = exponential *cwnd* growth
Congestion avoidance mode = linear *cwnd growth*

Fast Recovery mode

- This state is entered when 3 duplicate ACKs are received (during either slow-start or congestion avoidance mode)
    - Recall: 3 duplicate/burst ACKs == a NAK (implying a segment was lost)
- For each of the 3 burst ACKs that activated fast recovery mode, *cwnd* += *1* MSS
- When the missing/lost packet is eventually ACK'd, **congestion avoidance** is activated
- If a timeout occurs, fast recovery transitions to slow start
    - *new_cwnd* set = 1 MSS
    - *ssthresh* = *cwnd/2*
- **Fast recovery is recommended but not required**
- **Pg 276:** read about **TCP Tahoe** and **TCP Reno**

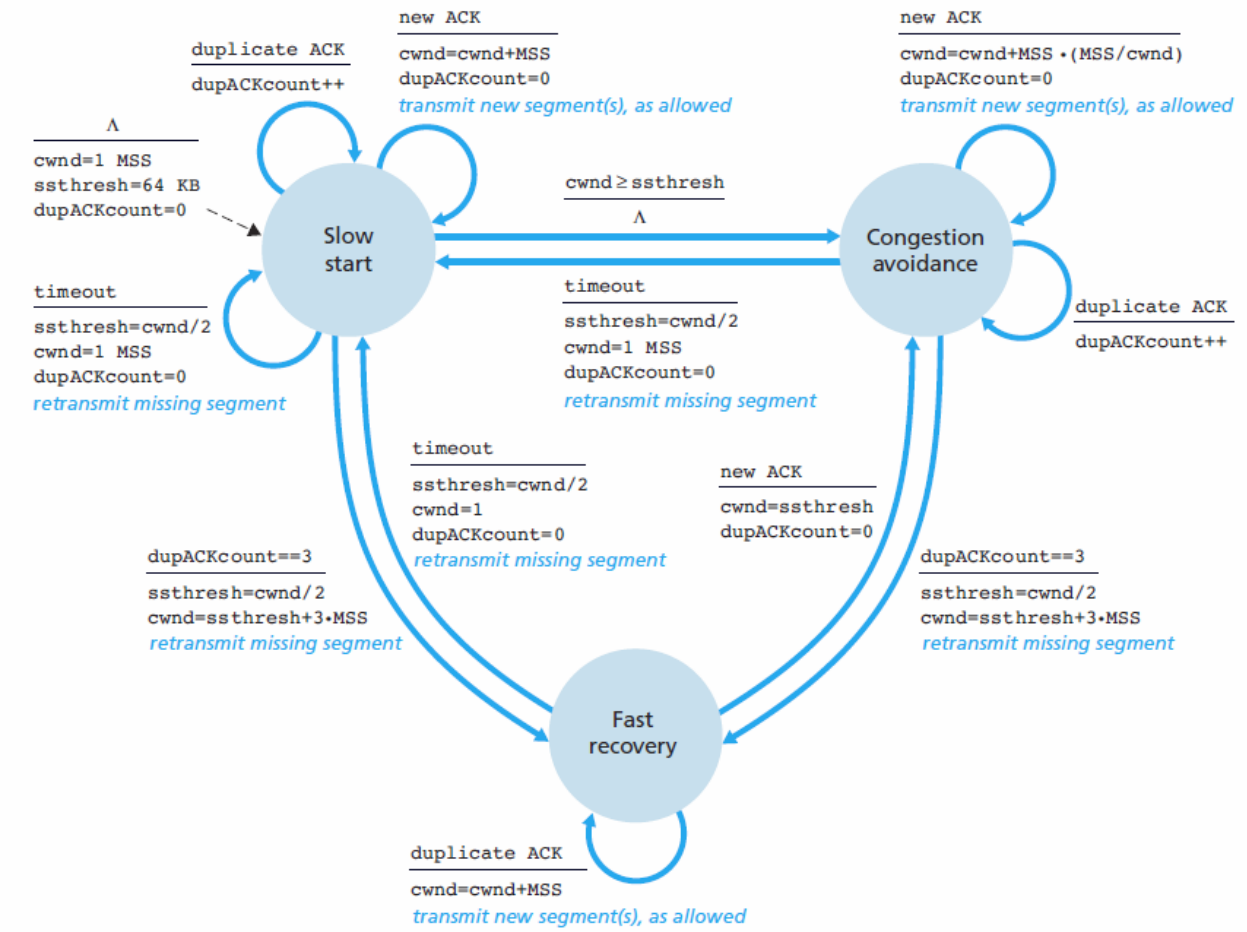
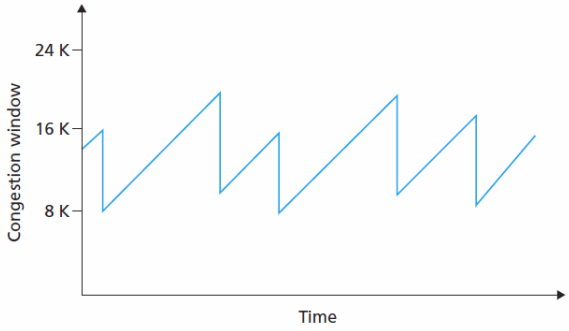


Figure 3.52 ♦ FSM description of TCP congestion control

TCP Congestion Control: Retrospective

- TCP congestion control known as **additive-increase, multiplicative-decrease (AIMD)**
  - Since *cwnd* =+1 MSS for each RTT → *additive-increase*
    - (when not in slow start mode)
  - Since *cwnd *= 0.5* for each triple-burst ACK (loss) → *multiplicative-decrease*
- **AIMD congestion control** = "saw tooth" behaviour



← AIMD TCP 'probes' for bandwidth by linearly increasing congestion window (and thus send rate) until a triple-burst ACK is received. It then halves the congestion window and restarts the linear increasing.

Figure 3.54 ♦ Additive-increase, multiplicative-decrease congestion control

Go to page 279 and read section 3.7.1 on *Fairness.* Also read the summary of chap.3 at the end.