Chapter 3
Transport Layer

Teacher: Xu Yang

Chapter 3: Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

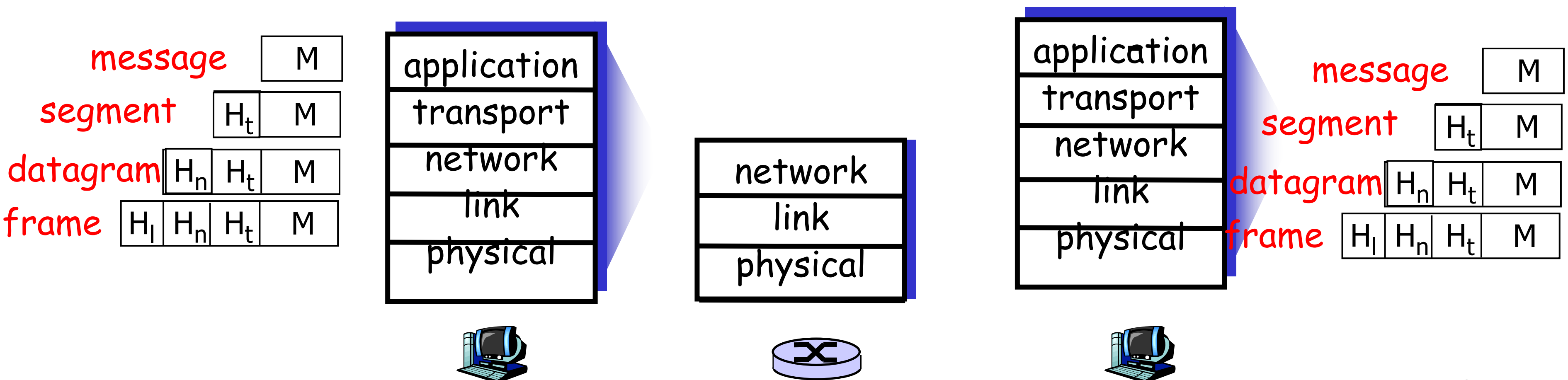
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

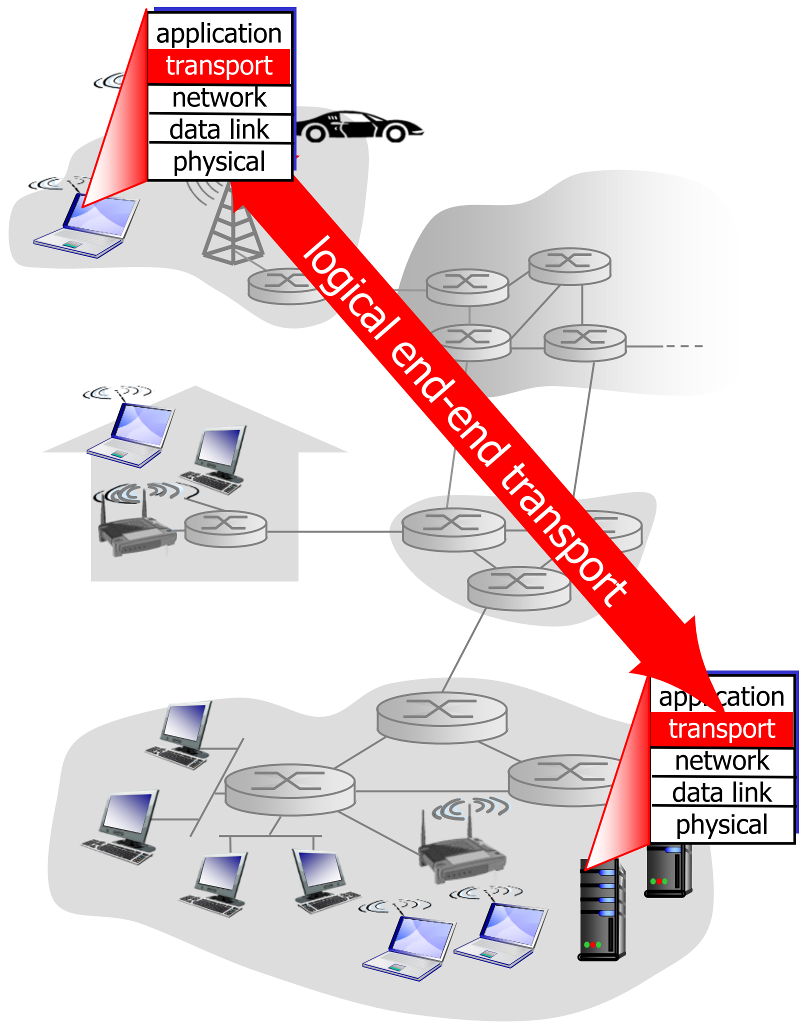
Transport Layer

- Physical communication/connection: connection through physical medium/link
- Logical communication/connection: the connection between two end system running the same protocol at the same layer



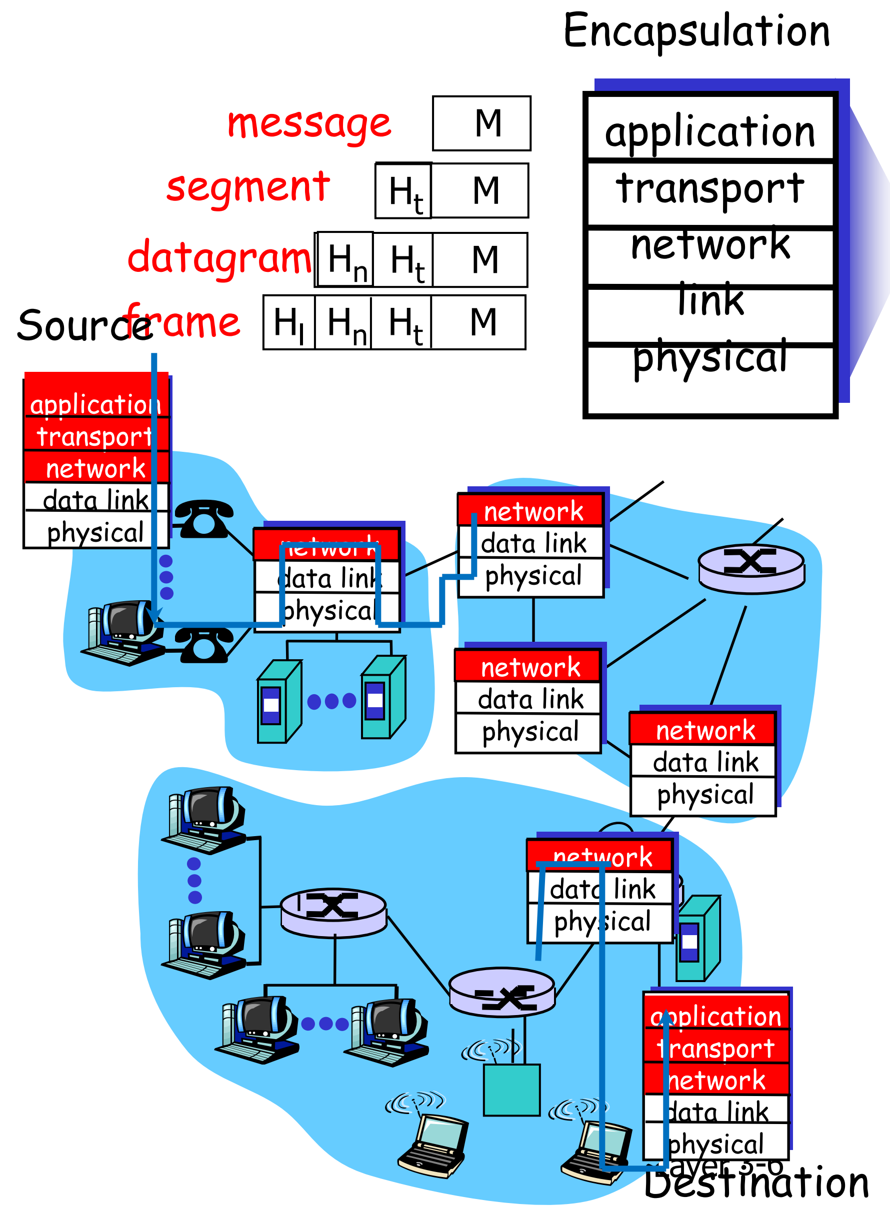
Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems, not at network core (e.g. network routers)
  - send side: breaks app messages into smaller chunks and adds a transport-layer header to each chunk to create the transport-layer segments. The transport layer then passes the segment to the network layer, where the segment is encapsulated within a datagram (i.e., the network layer packet),
  - receiving side: extract the segment from the datagram, reassembles segments into messages, and passes to app layer
- Datagrams are routed through intermediate nodes (routers)



Transport vs. network layer

- *transport layer:* provide logical communication between two processes running on different hosts
  - Extends "host-to-host" communication to "process-to-process" communication
  - Relies on, enhances, network layer services
  - Segment
- *network layer:* provide logical communication between two hosts
  - Datagram
  - Unreliable, best-effort delivery: Datagram's may be lost, duplicated, reordered in the Internet – "best effort" service

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport vs. Network Layer service

- Network layer: IP (Internet protocol) is the name of Internet's network-layer protocol.
  - IP provides unreliable, best-effort delivery
    - No guarantees for delay, bandwidth, successful delivery, orderly delivery, or the integrity of the data.
- Transport layer: Two transport layer protocols provide different services (not available: delay guarantees, bandwidth guarantees)
  - TCP (Transmission Control Protocol) provides reliable, in-order delivery
    - connection setup
    - flow control, sequence numbers, acknowledgements, timers, error checking
    - congestion control
  - UDP (User Datagram Protocol) provides unreliable, unordered delivery
    - error checking: providing integrity checking by including error-detection field in the segments' headers

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Multiplexing/demultiplexing

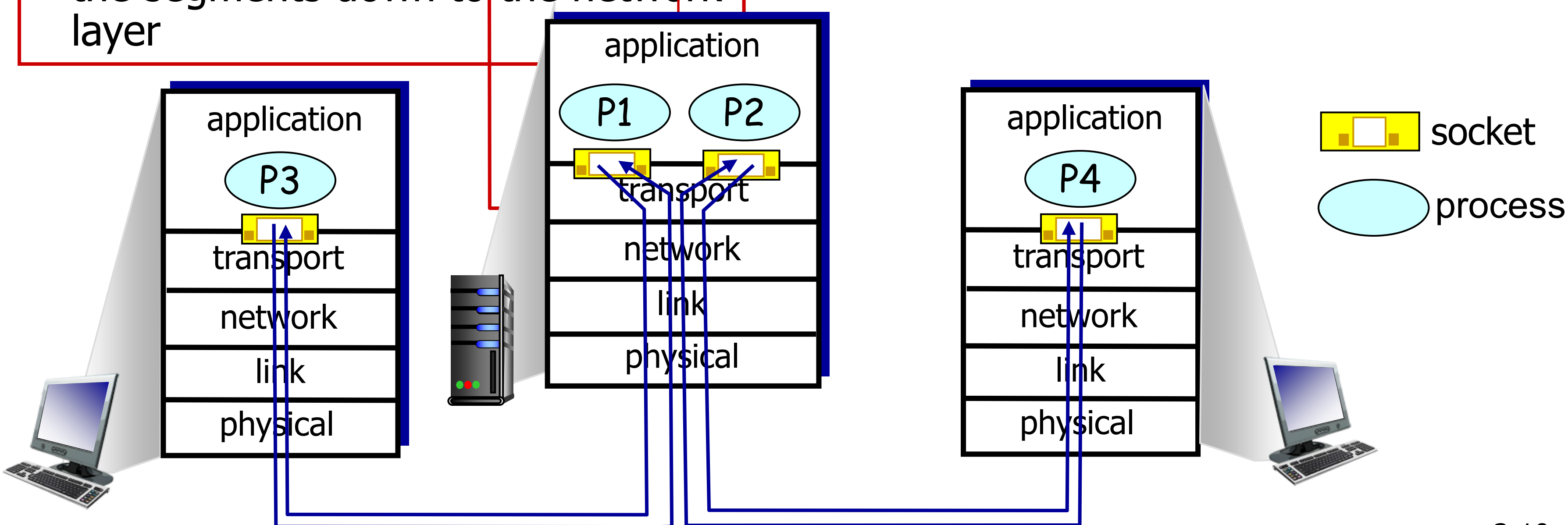
Extending host-to-host delivery to process-to-process delivery is called **transport-layer multiplexing** and **demultiplexing**.

multiplexing at sender:

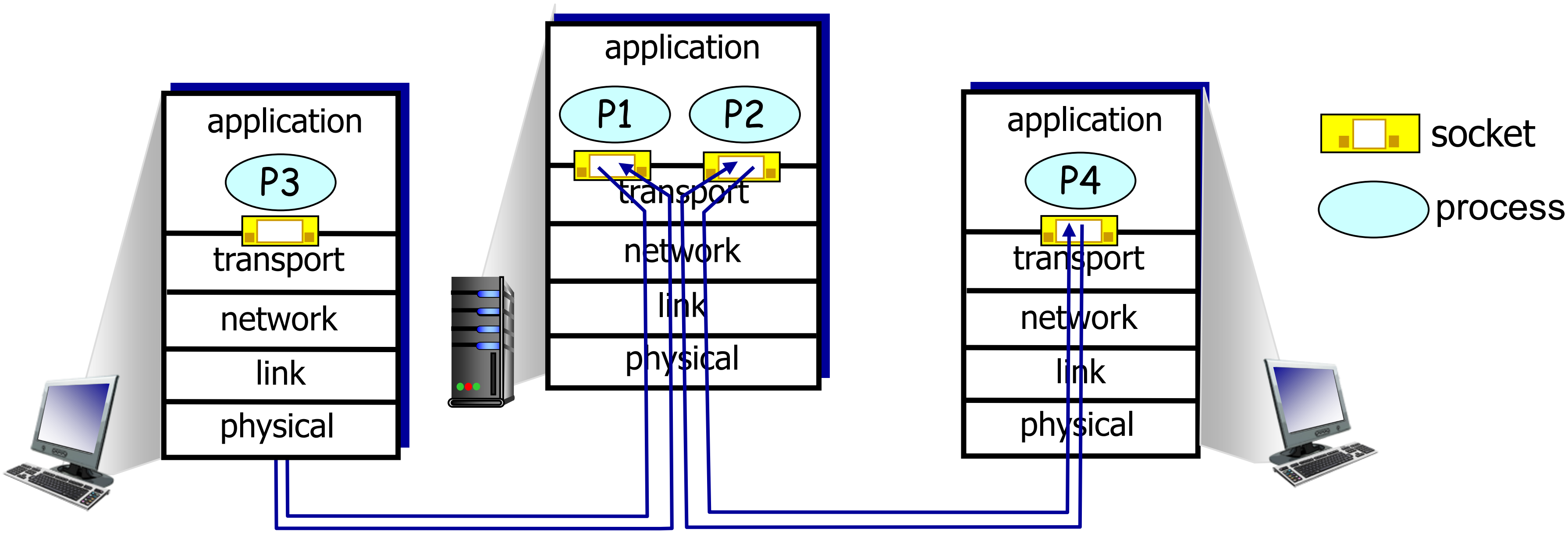
the job of gathering outgoing data chunk from different sockets at the source side, encapsulating each data chunk with header information to create segments, and passing the segments down to the network layer

demultiplexing at receiver:

use header info to deliver received transport layer segments to correct socket



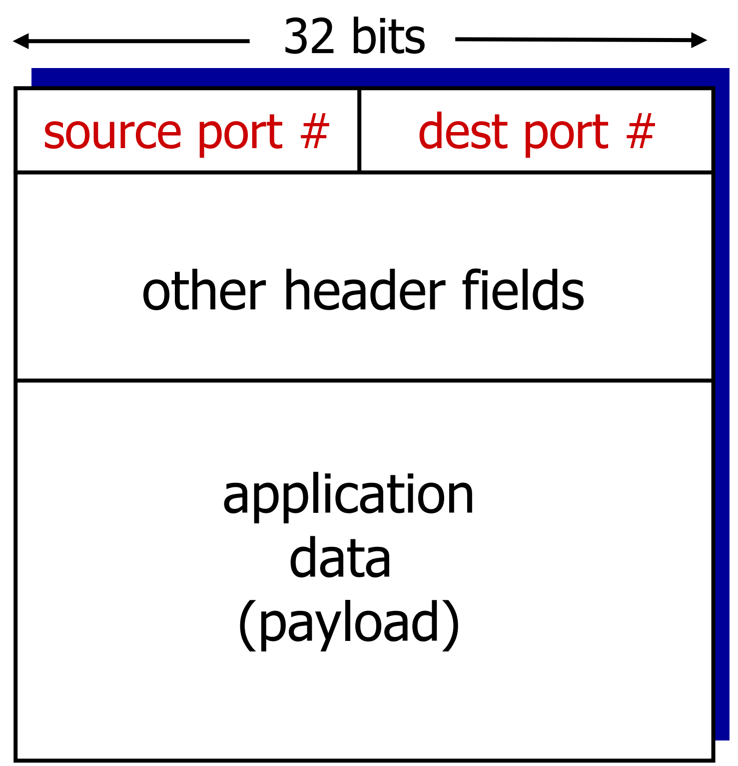
Multiplexing/demultiplexing



- Socket
  - Act as door, through which data passes from the network to the process and from process to the network. A host can have one or more sockets.
  - A host has a unique IP address (32-bit for IPV4)
  - Each socket has a port number (16-bit) at a host , ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which are reserved for use by well-known application protocols such as HTTP (port number 80) and FTP (port number 21).

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket

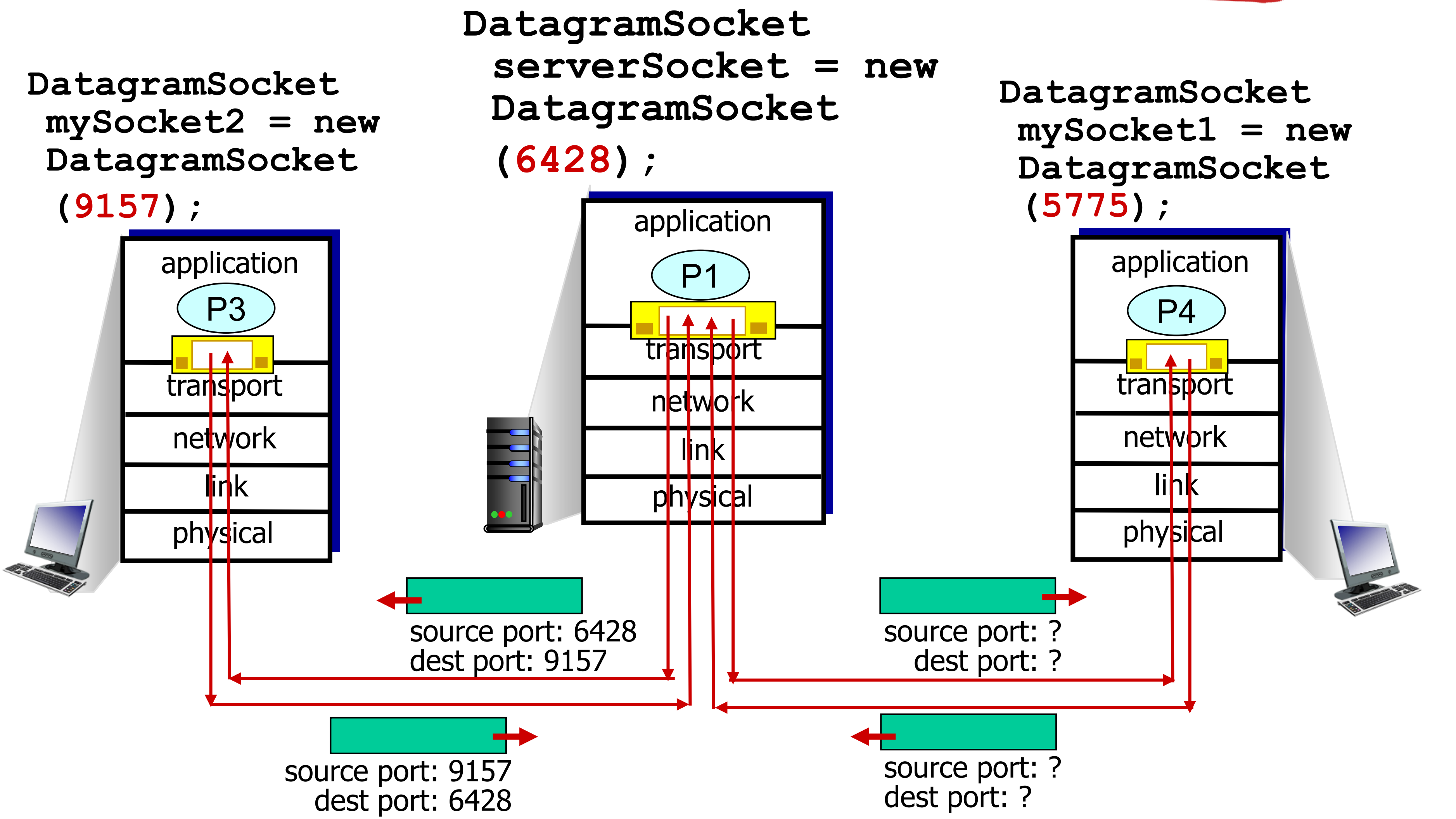


TCP/UDP segment format

Connectionless Mux/Demux (UDP)

- An application program (i.e., Skype) creates a UDP socket in Host A
- The transport layer in Host A does the UDP multiplexing (create segments, pass them to the network layer)
- The network layer encapsulates the segment in an IP datagram and makes a best-effort delivery to the receiving host.
- When receiving host gets UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- Host uses both the dest. IP address and dest. Port number to direct segments to appropriate socket.
- UDP socket is identified by two-tuple
  - destination IP address
  - destination port number

Connectionless demux: example



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.

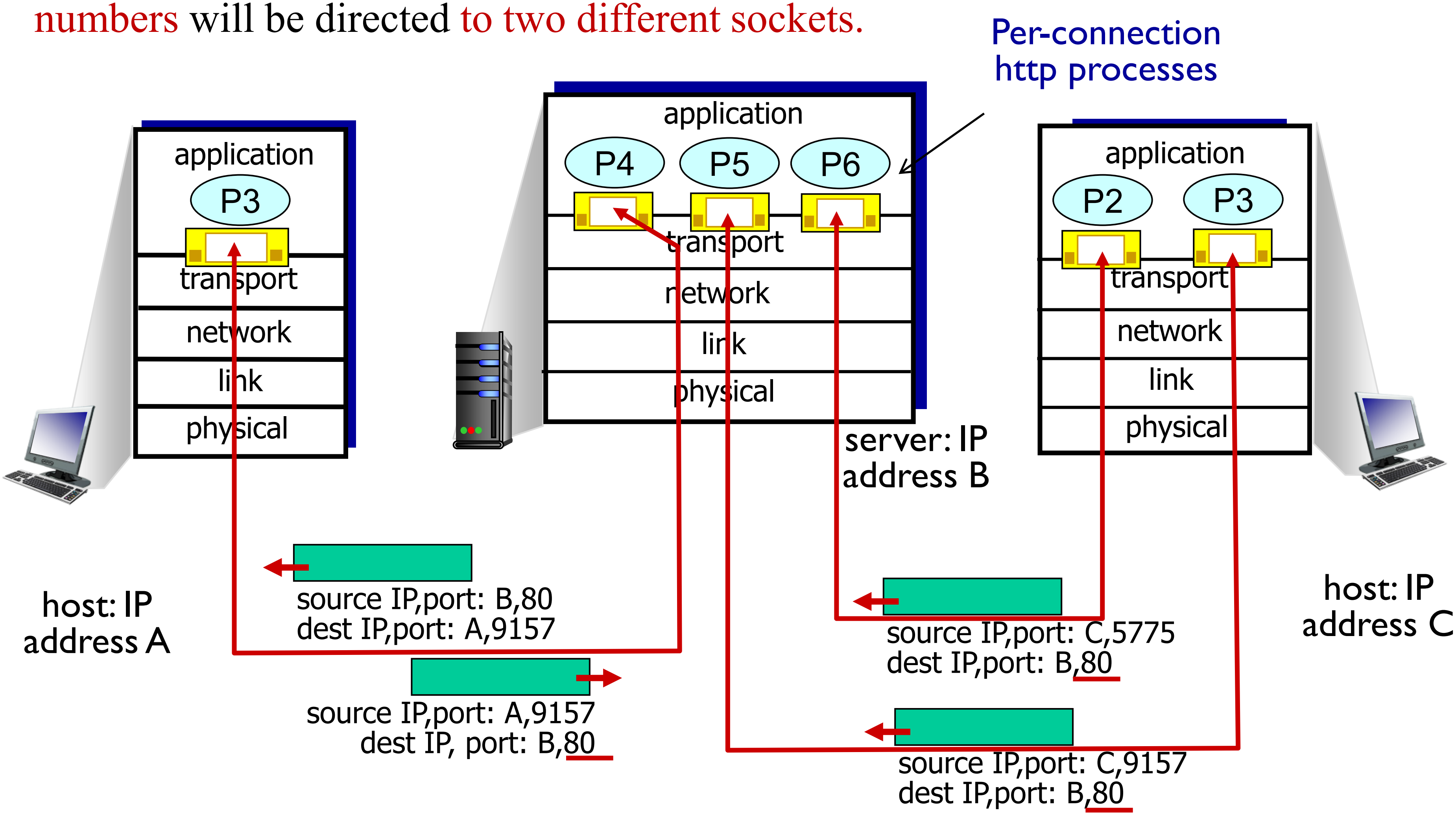
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
  - Persistent HTTP will have the same server socket for exchanging messages

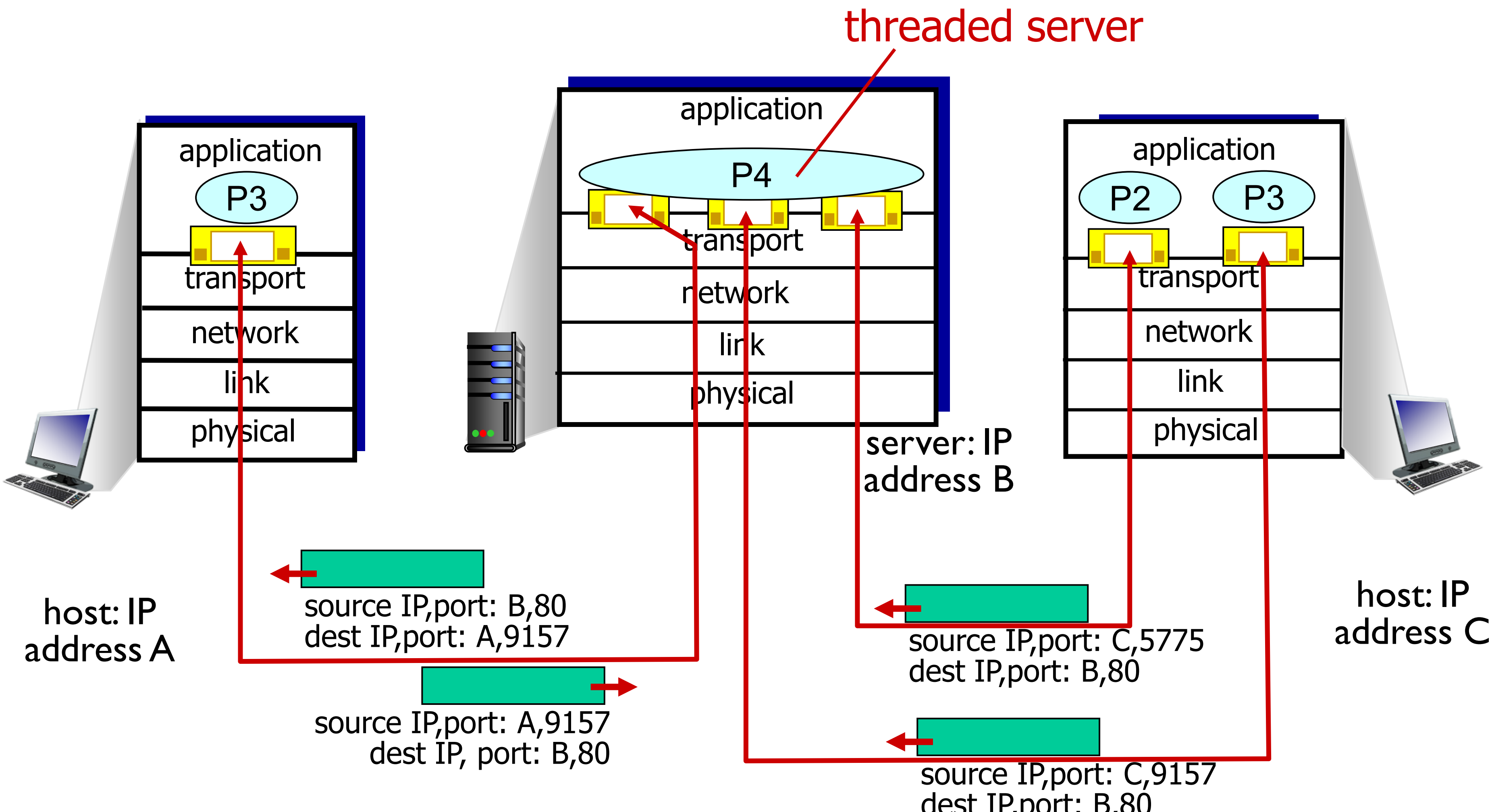
Connection-oriented demux: example

Two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets.



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

Connection-oriented demux: example



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.

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

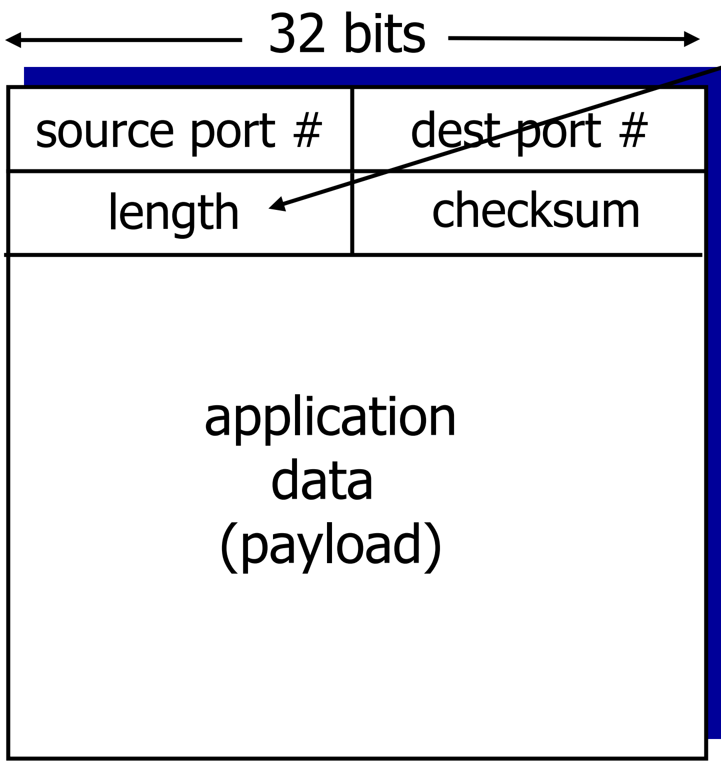
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- UDP only provides essential functions that transport protocol can do (i.e. multiplexing/demultiplexing and some error check).
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

8 bytes header

length, in bytes of UDP segment, including header

Checksum: used by the receiving host to check whether error have been introduced into the segment during the transmission

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

1's complement sum

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

1. Add the 16-bit values up. Each time a carry-out (17th bit) is produced, swing that bit around and add it back into the LSb (one's digit). This is somewhat erroneously referred to as "one's complement addition."
2. Once all the values are added in this manner, invert all the bits in the result. A binary value that has all the bits of another binary value inverted is called its "one's complement," or simply its "complement."

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

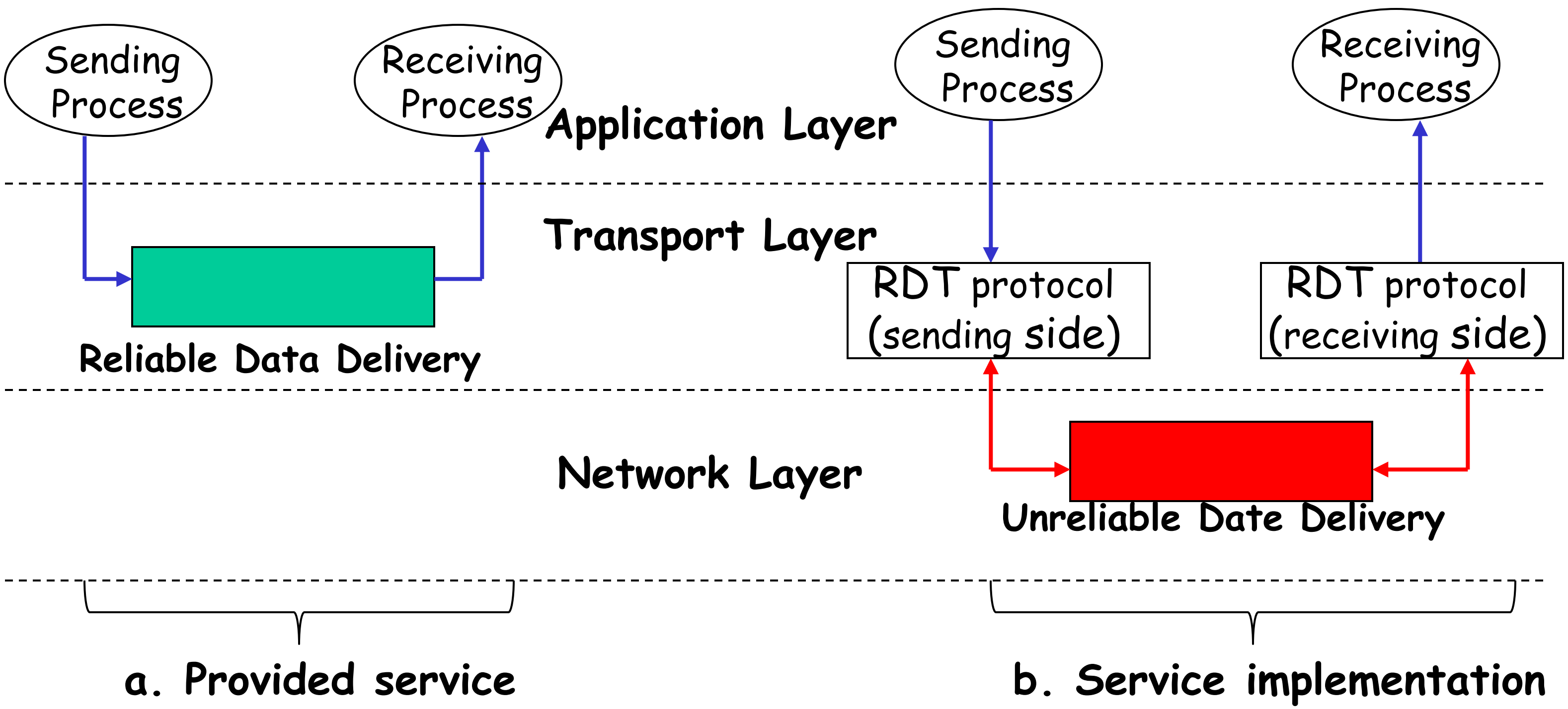
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of Reliable Data Transfer

- important in application, transport, and link layers



- How to provide the reliable data transfer on the top of an unreliable end-to-end network layer
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)

Three Types of Channels

we' ll incrementally develop sender, receiver sides of reliable data transfer protocol **(rdt)**

- Perfect channel: underlying channel is perfectly reliable
  - no bit errors
  - no loss of packets
- Channel with bit errors
  - All packets are received in correct order
  - Packets may be corrupted (i.e., bits may be flipped)
- "Loosy" channel: underlying channel not only corrupts bits in packets but also loses packets
  - Packets may be lost
  - Packets may be corrupted

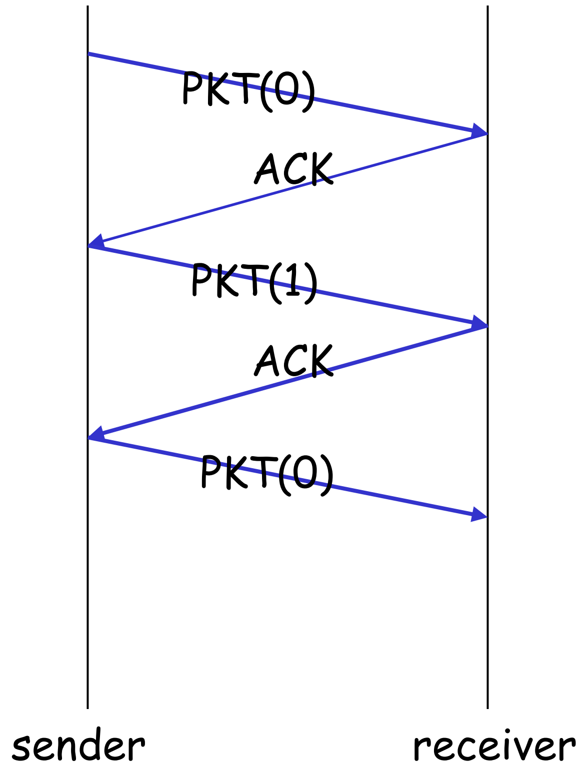
Data Transfer over a Perfect Channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- consider only unidirectional data transfer, but control info will flow on both directions!
  - sender sends data into underlying channel
  - receiver reads data from underlying channel
- there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong!

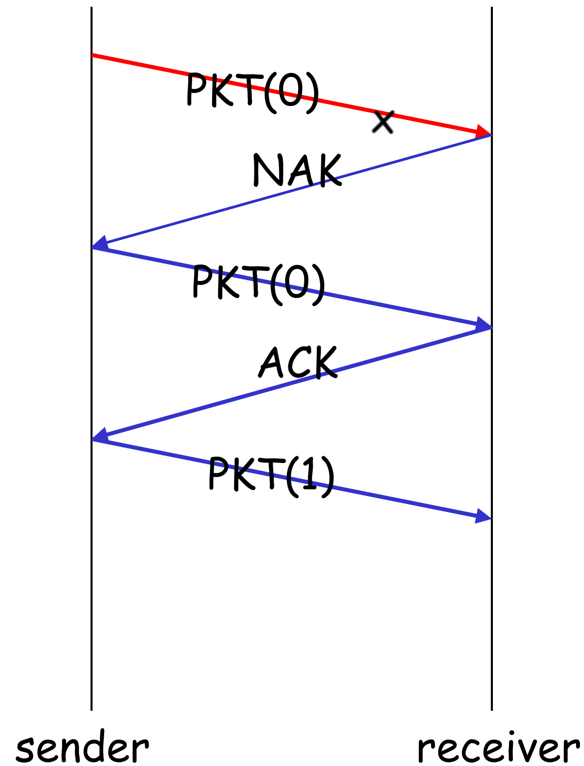
Data Transfer over Channel with Bit Errors

- Assumptions
  - All packets are received in correct order
  - Packets may be corrupted (i.e., bits may be flipped)
  - Checksum to detect bit errors
- How to recover from bit errors? Use ARQ (automatic repeat request) mechanism
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received correctly
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet is received with errors
  - sender retransmits packet on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - Feedback : control message from receivers to senders
  - Retransmission

ARQ (automatic repeat request)



Everything is good

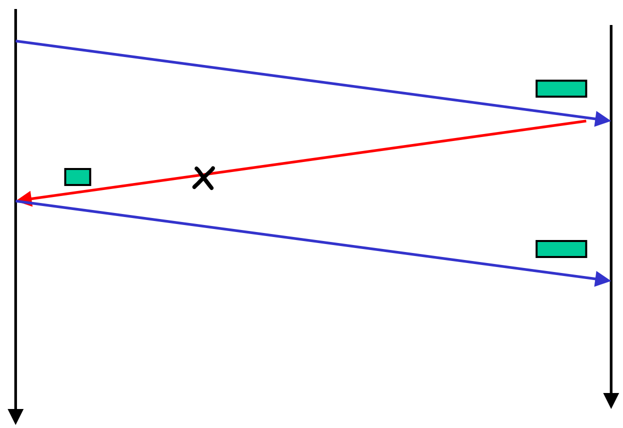


Received NAK

a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate



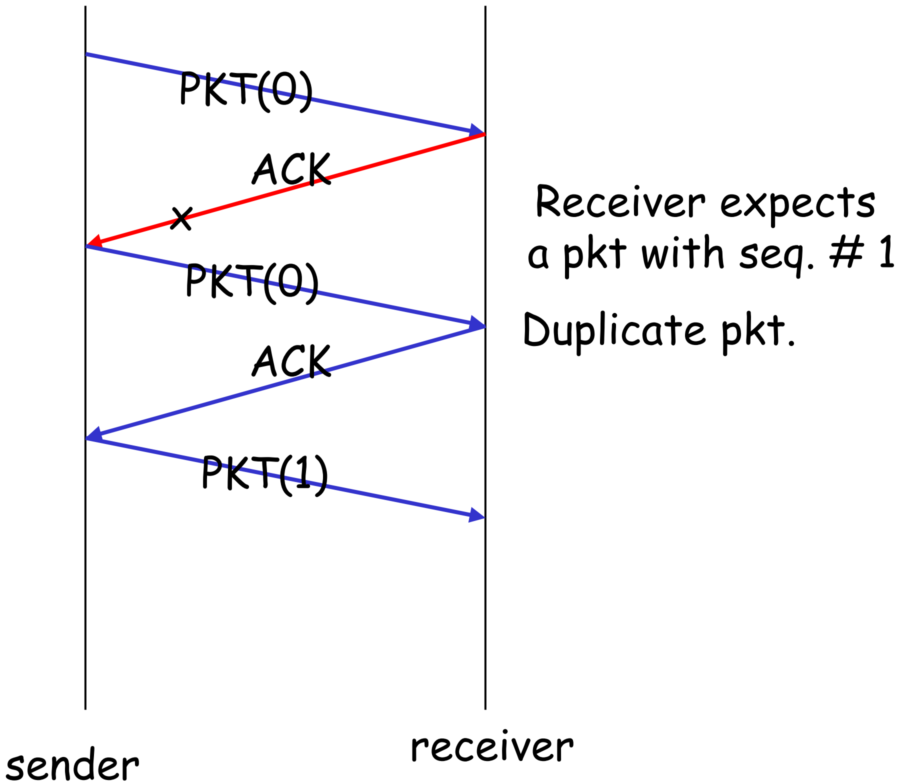
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

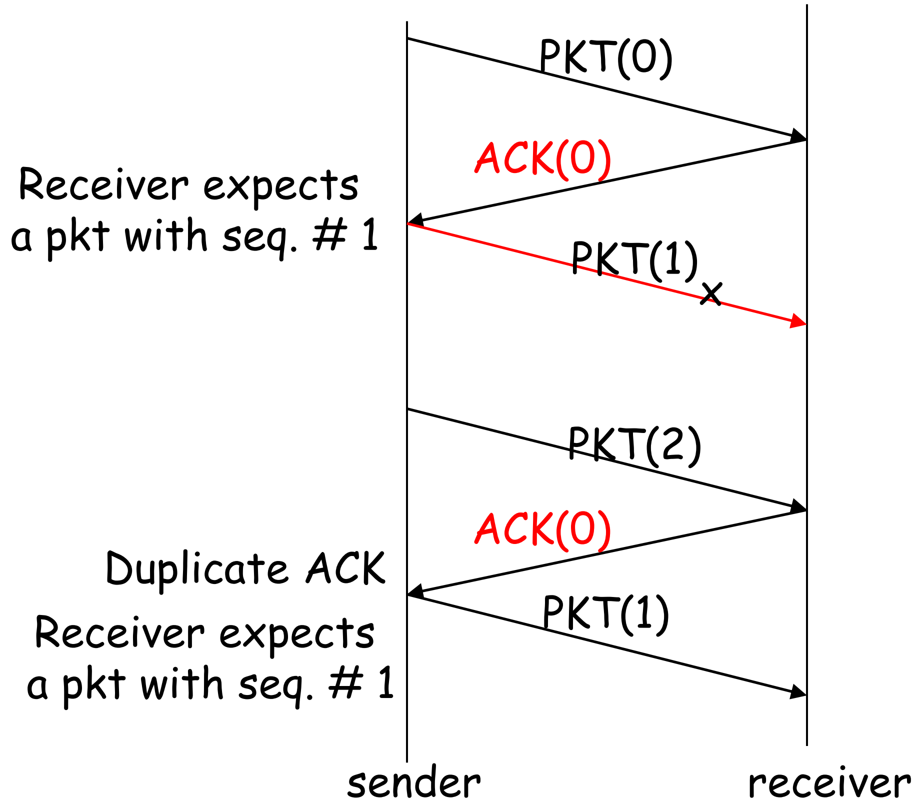
Handling Duplicate Packets



- 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 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 need only check this sequence number to determine whether or not the received packet is a retransmission.

A NAK-free Method

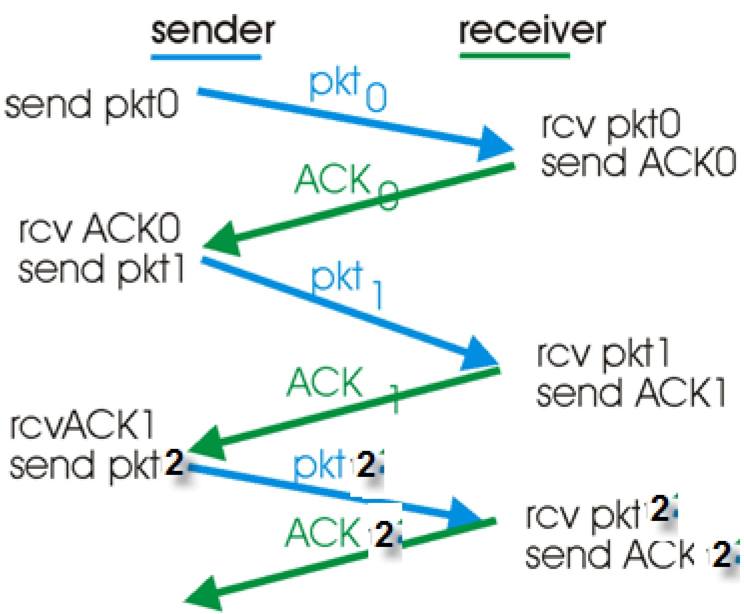
- using ACKs only
- instead of NAK, receiver sends ACK for correctly received packet with the highest in-order sequence number
  - receiver must *explicitly* include seq # of pkt being ACKed into ACK
- The sender that receives two ACKs for the same packets (that is, the sender receives duplicate ACKs) knows that the receive did not correctly receive the packet following the packet that is being ACKed twice, then retransmit this packet.



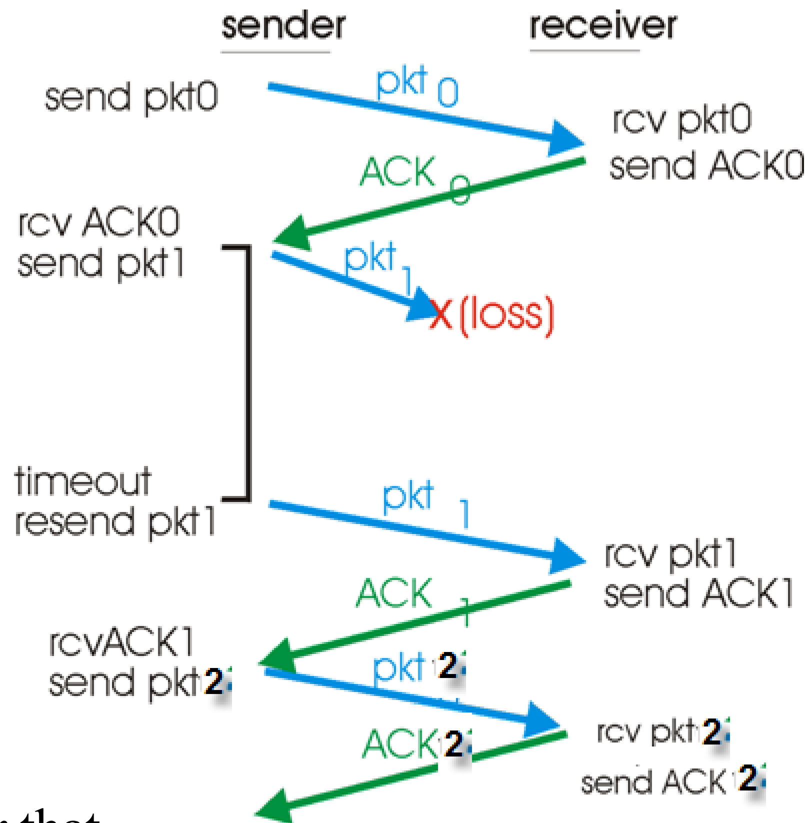
The case of "Lossy" Channels

- Assumption: underlying channel not only corrupt bits in packets but also lose packets (data or ACKs)
  - Packets may be corrupted
  - Packets may be lost
- How to identify the packet lost?
- Method:
  - Set a timer: sender waits "reasonable" amount of time for ACK (a Time-Out).

Examples



(a) operation with no loss



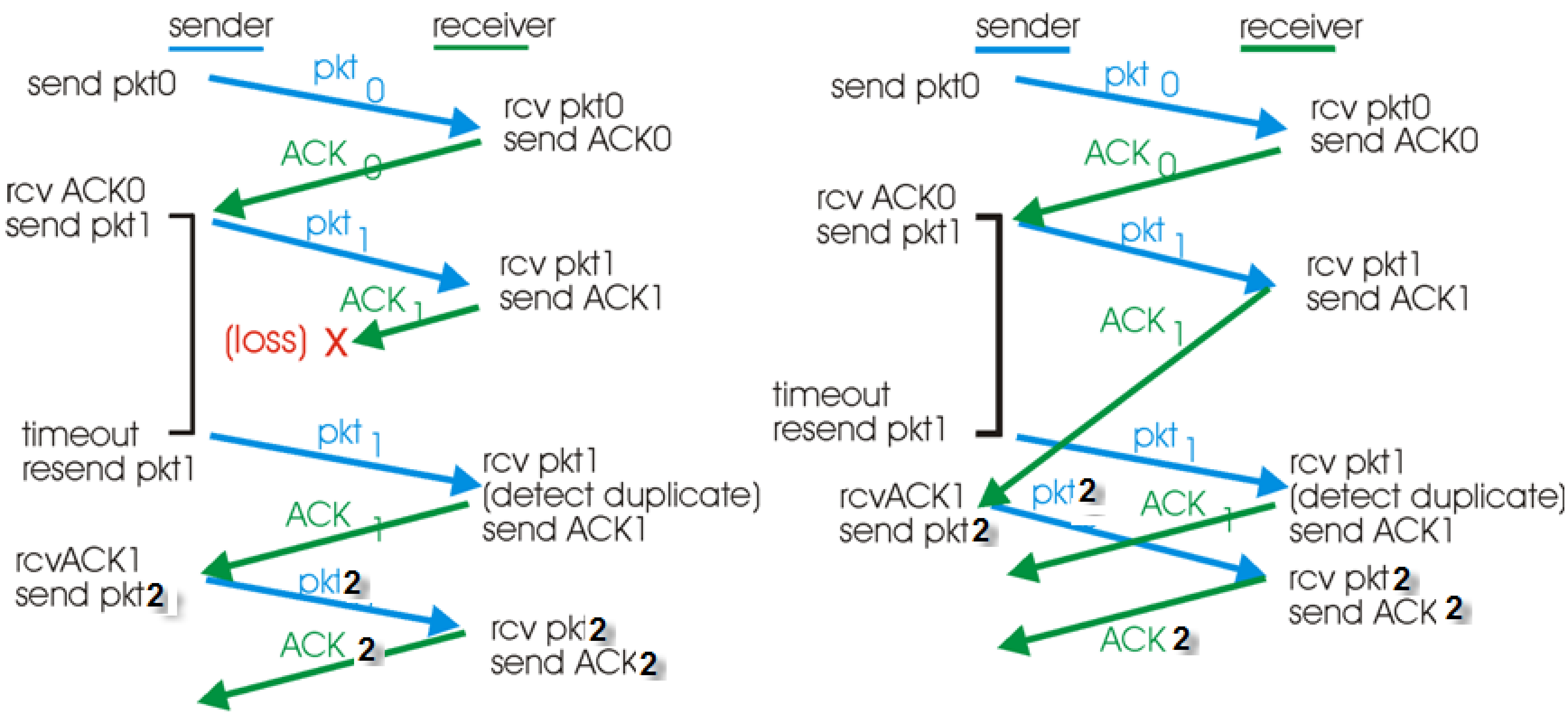
(b) lost packet

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.

The case of “Lossy” Channels

- Approach used in the case of “lossy” channel
  - Set a timer: sender waits “reasonable” amount of time for ACK (a Time-Out).
  - Sender adds sequence number to each packet.
  - Receiver must specify sequence # of packet being ACKed.
  - Sender retransmits current packet if ACK/NAK is corrupted or lost.
  - Receiver discards (doesn’t deliver up) duplicate packet.

Examples



(c) lost ACK

(d) premature timeout

How to set the value of time out is a key issue!

Performance of "stop and wait"

- performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

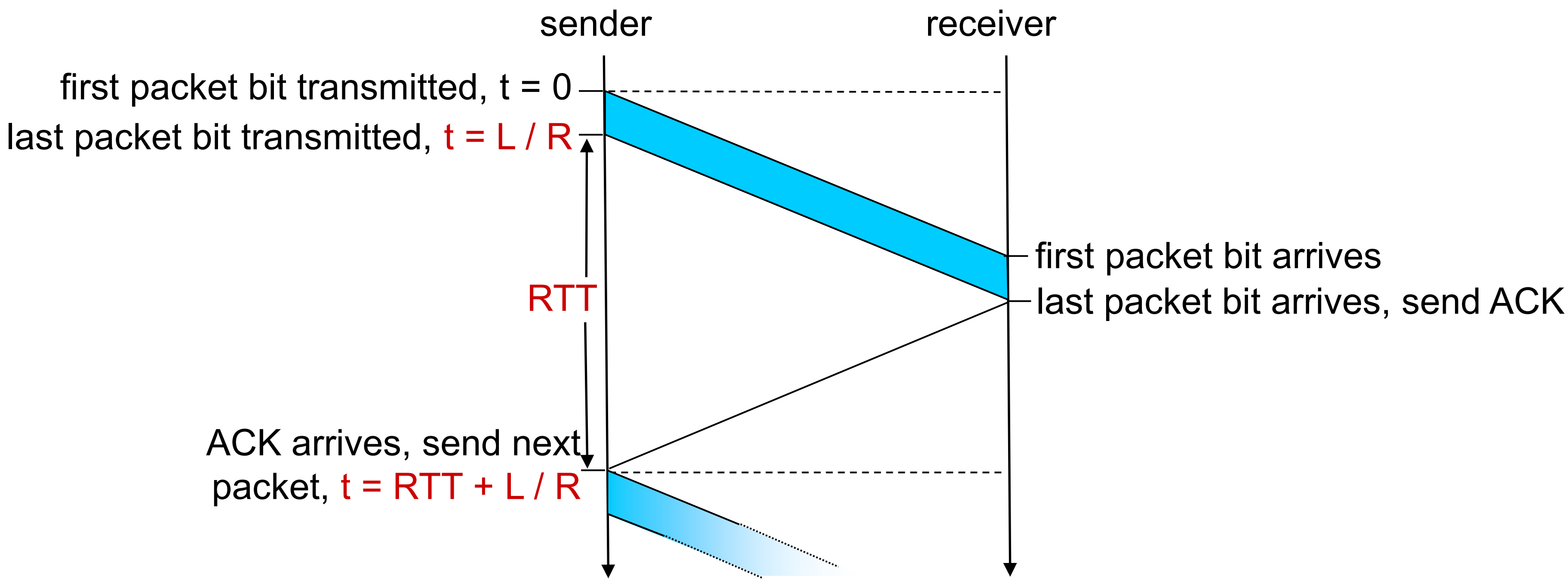
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: *utilization* – fraction of time sender is busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

  - if RTT=30 msec, 1KByte pkt every 30 msec: 267kbps throuput over 1 Gbps link
- network protocol limits use of physical resources!

stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Stop and Wait: sender sends one packet, then waits for receiver response

Low efficiency, low utilization

Stop-and-Wait Protocol

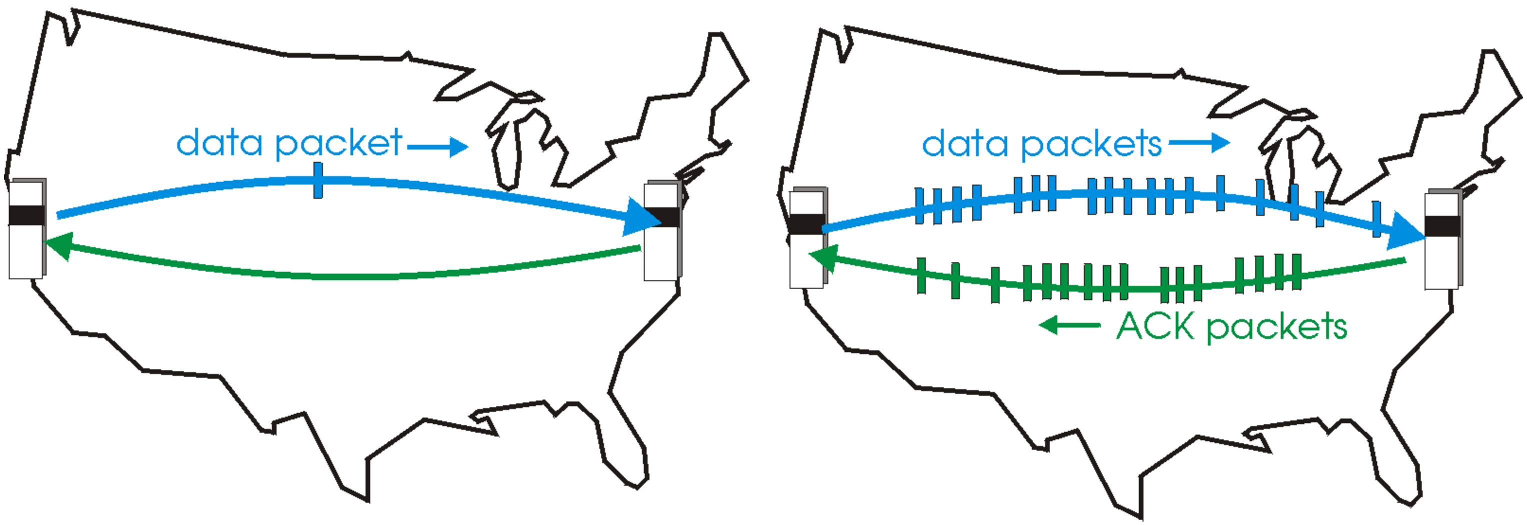
- Features
  - It has a timer implementation
  - It has bit error detection mechanism
  - Timer should be set for each individual packet
  - Only 1 packet is sent at a time
  - No pipelining
  - Sender window size is 1
  - Receiver window size is 1
- Advantage
  - Simple
- Disadvantage
  - Efficiency and utilization are very low

How to Address the Low Efficiency of Stop-and-Wait Protocol

- Solution: the sender is allowed to send multiple packets without waiting for acknowledgements
- Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets. Since these in-transit packets can be visualized as filling a pipeline, this technique is known as pipelining.
- Two features of pipelined protocol
  - The range of sequence numbers must be increased.
  - The sender and the receiver may have to buffer more than one packet.

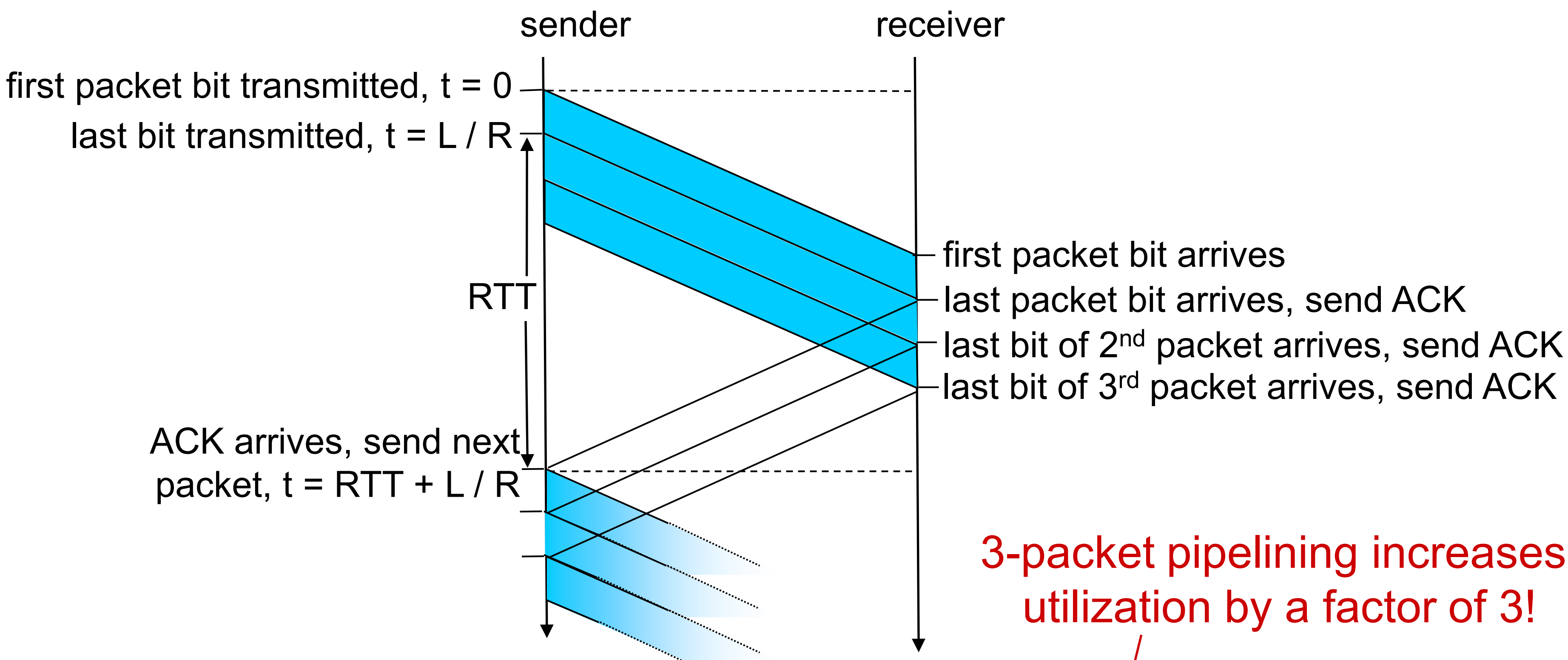
Pipelined protocols

- Two generic forms of pipelined protocols
  - *go-Back-N*
  - *selective repeat*



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

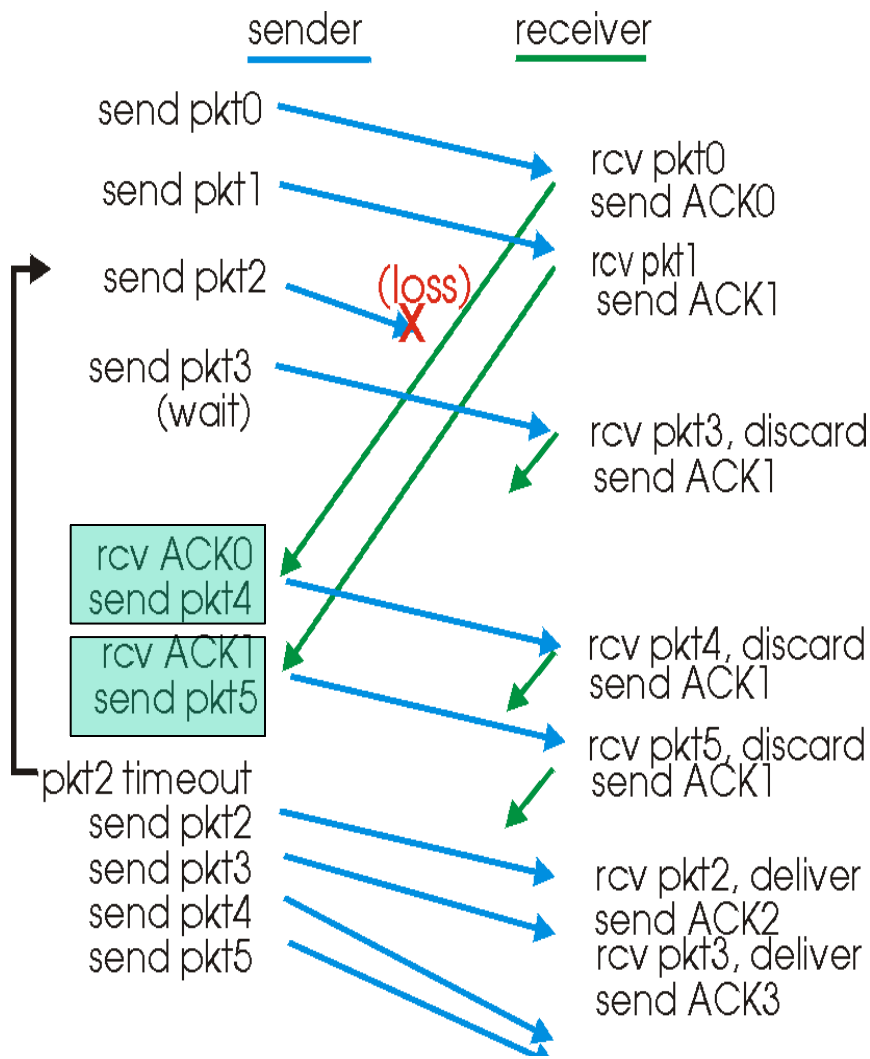
Go-Back-N

- A pipelined protocol
- It improves the transmission efficiency
- It introduces a window of size N: N is called as window size.
- It allows up to *N* unACKed packets in the network
  - Sender can transmit N packets into the network before receiving an ACK
- Concepts
  - Window size
  - Sequence number
  - Cumulative acknowledgement
  - How to deal with out-of-order packets
  - Timer and timeout

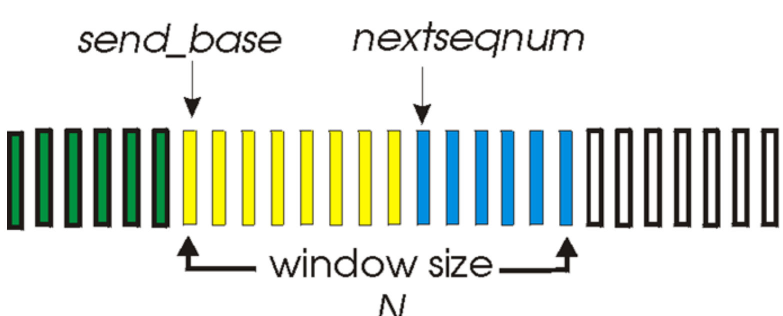
http://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_6/video_applets/GBNindex.html

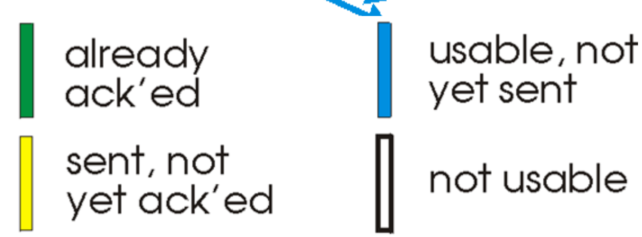
GBN in action

- Window size N (here N=4):
  - Label each packet with a sequence number.
  - A window is a collection of adjacent sequence numbers.
  - The size of the collection is the sender's window size.
  - If window not full, transmit.



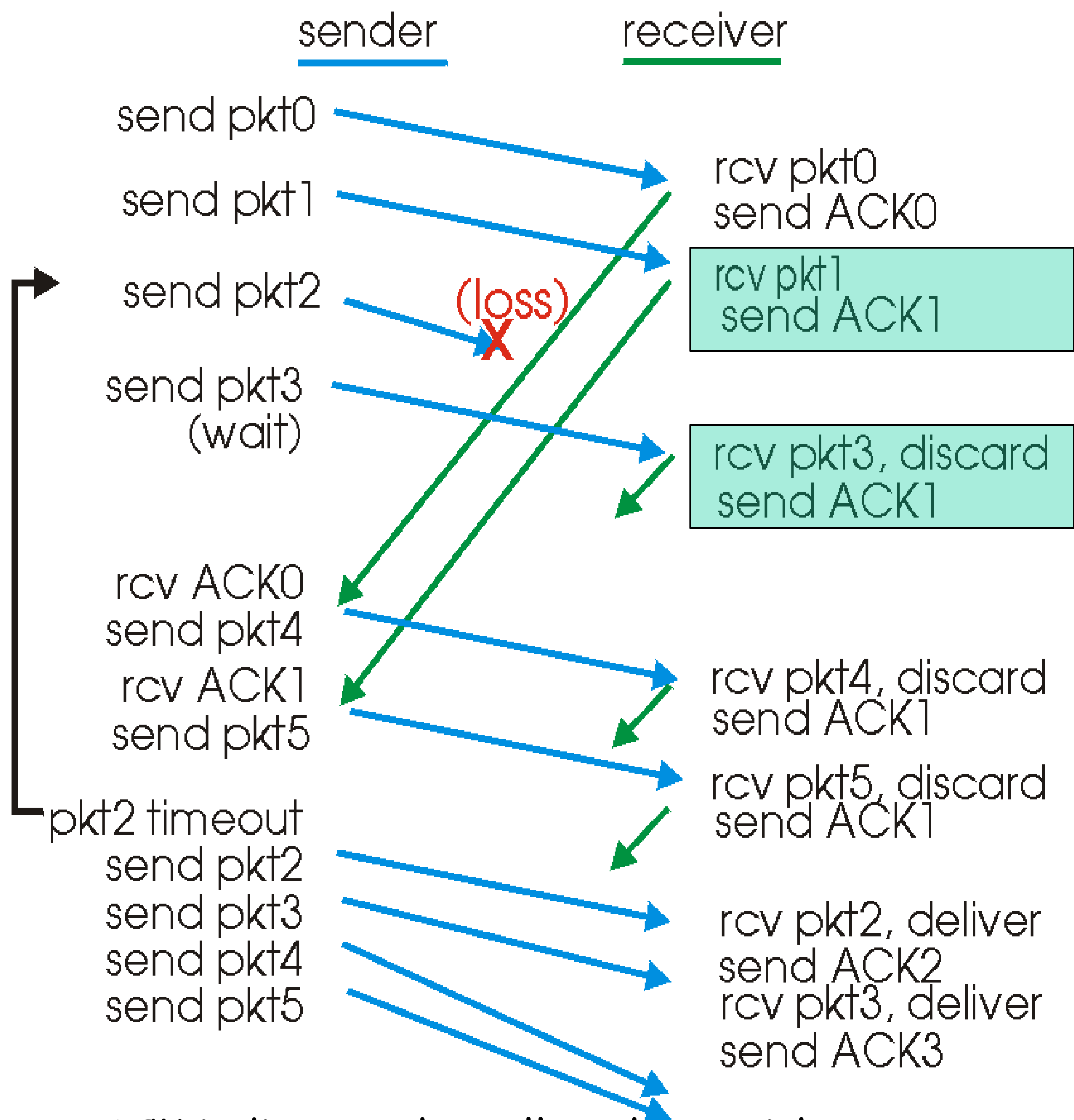
0 1 2 3 4 5 6 7 8 9





GBN in action

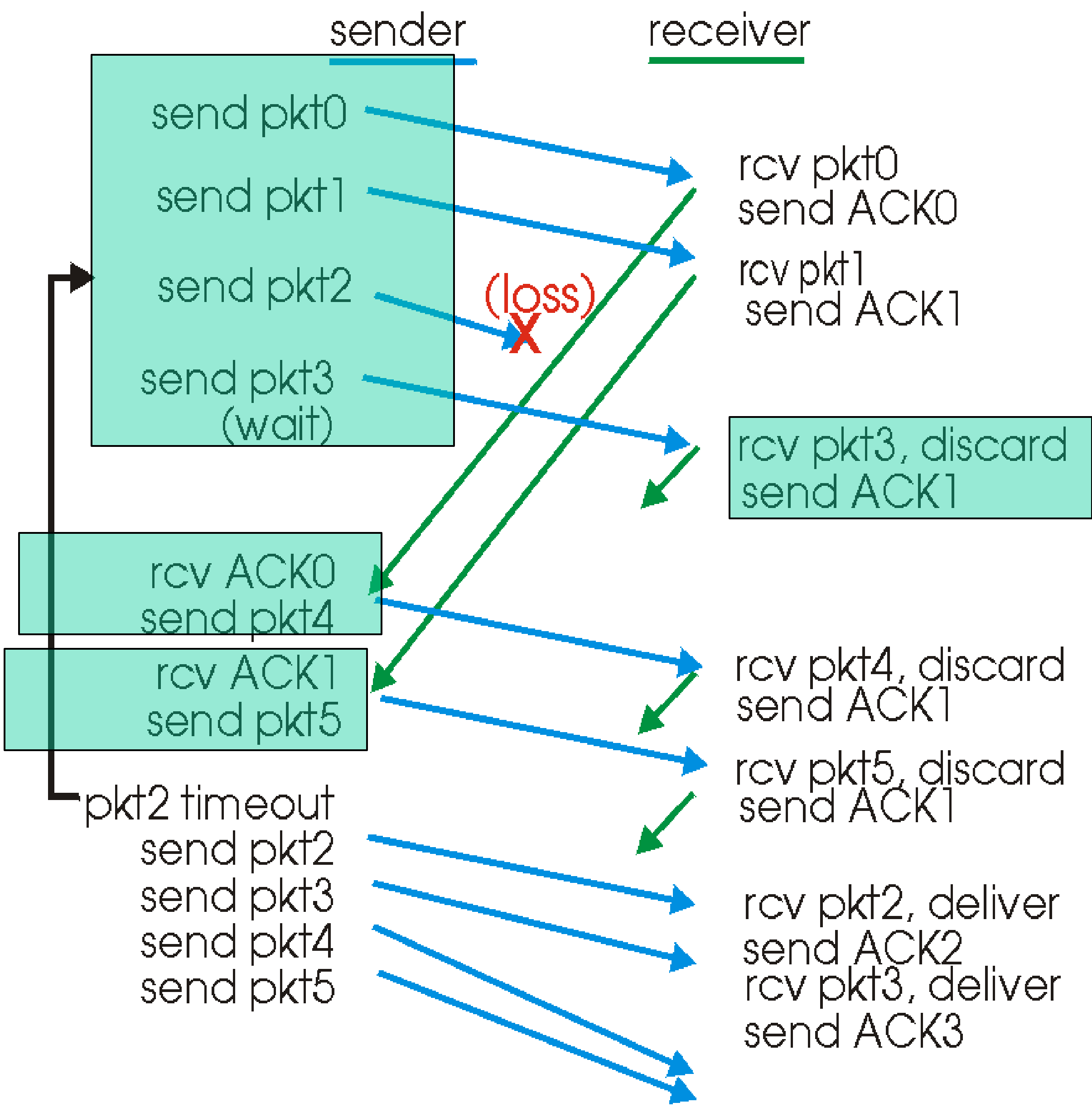
- Cumulative acknowledgement: always send ACK for correctly-received packet with highest *in-order seq #*
  - need only remember **`expected sequence number`**
  - may generate duplicate ACKs



With cumulative acknowledge, an ACK indicates that all packets with a sequence number up to and including the number specified in the ACK have been correctly received.

GBN in action

- out-of-order packet:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #



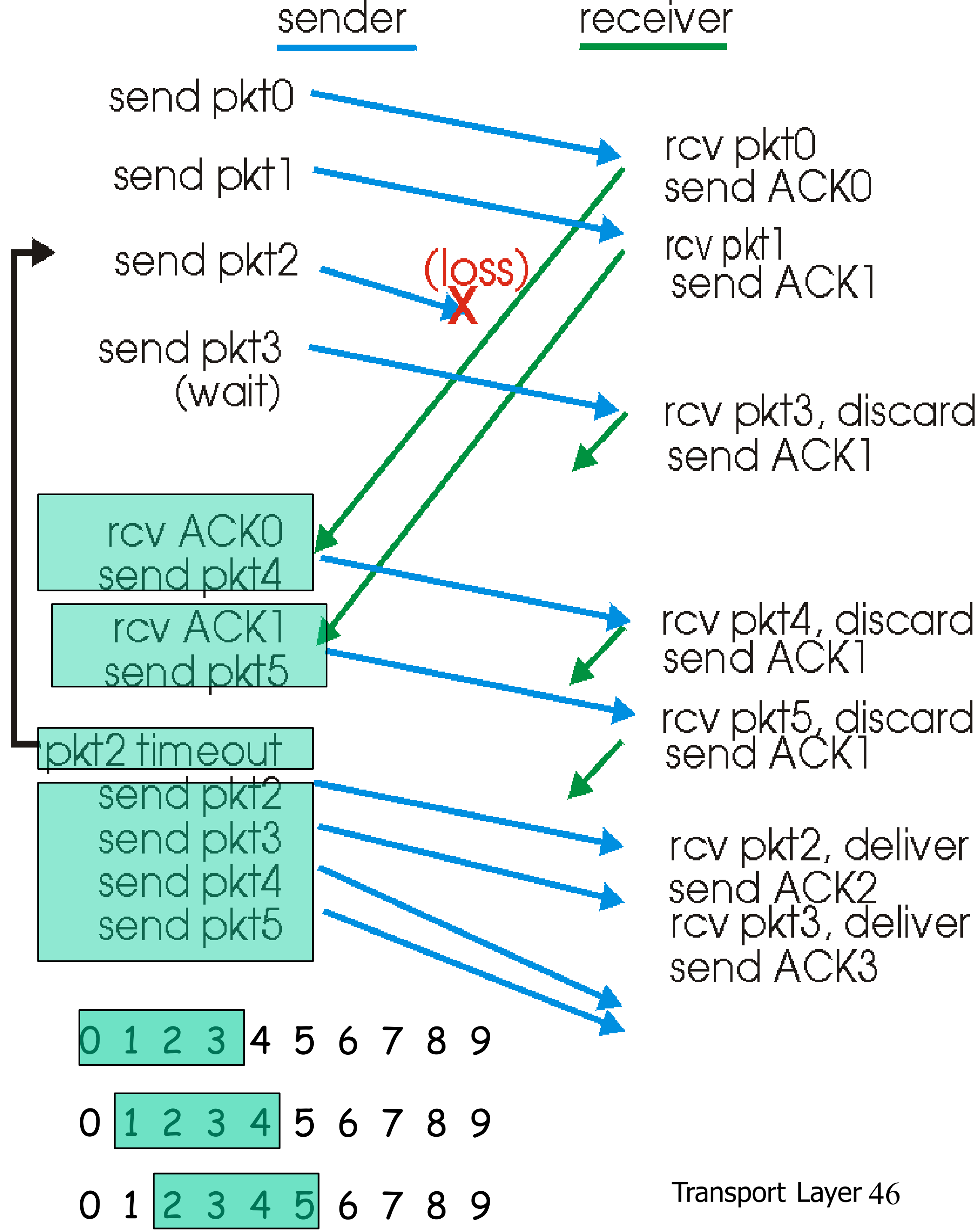
0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

GBN in action

- Timer and timeout:
  - Uses a single timer – represents the oldest transmitted, but not yet ACKed pkt
  - If an ACK is received but there are still other transmitted but not yet ACKed packets, the timer is restarted.
  - On timeout, send all packets previously sent but not yet ACKed.



Go-Back-N

- Sender Operation:
  - Label each packet with a sequence number
  - A window is a collection of adjacent sequence numbers
  - The size of the collection is the sender's window size
  - If window not full, sender transmits packets
  - ACKs are cumulative
  - Timer and timeout
    - On timeout, sender sends all packets previously sent but not yet ACKed.
    - Uses a single timer – represents the oldest transmitted, but not yet ACKed packet.

Go-Back-N

- Receiver Operation:
  - ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
    - may generate duplicate ACKs
    - need only remember **`expectedseqnum`**
  - out-of-order pkt:
    - discard (don't buffer): *no receiver buffering!* deal the out-of-order by simply discarding the out-of-order packets.
    - re-ACK pkt with highest in-order seq #

Go-Back-N

- Advantages
  - Sender can send multiple packets at a time
  - Efficiency is high compared with stop-and-wait protocol
  - We can configure the window size
- Disadvantage
  - Sender needs to store the last unAcked N packets
  - Retransmission of many error-free packets following an erroneous packet.
  - It is inefficient when round-trip delay large and data transmission rate is high.

Selective repeat

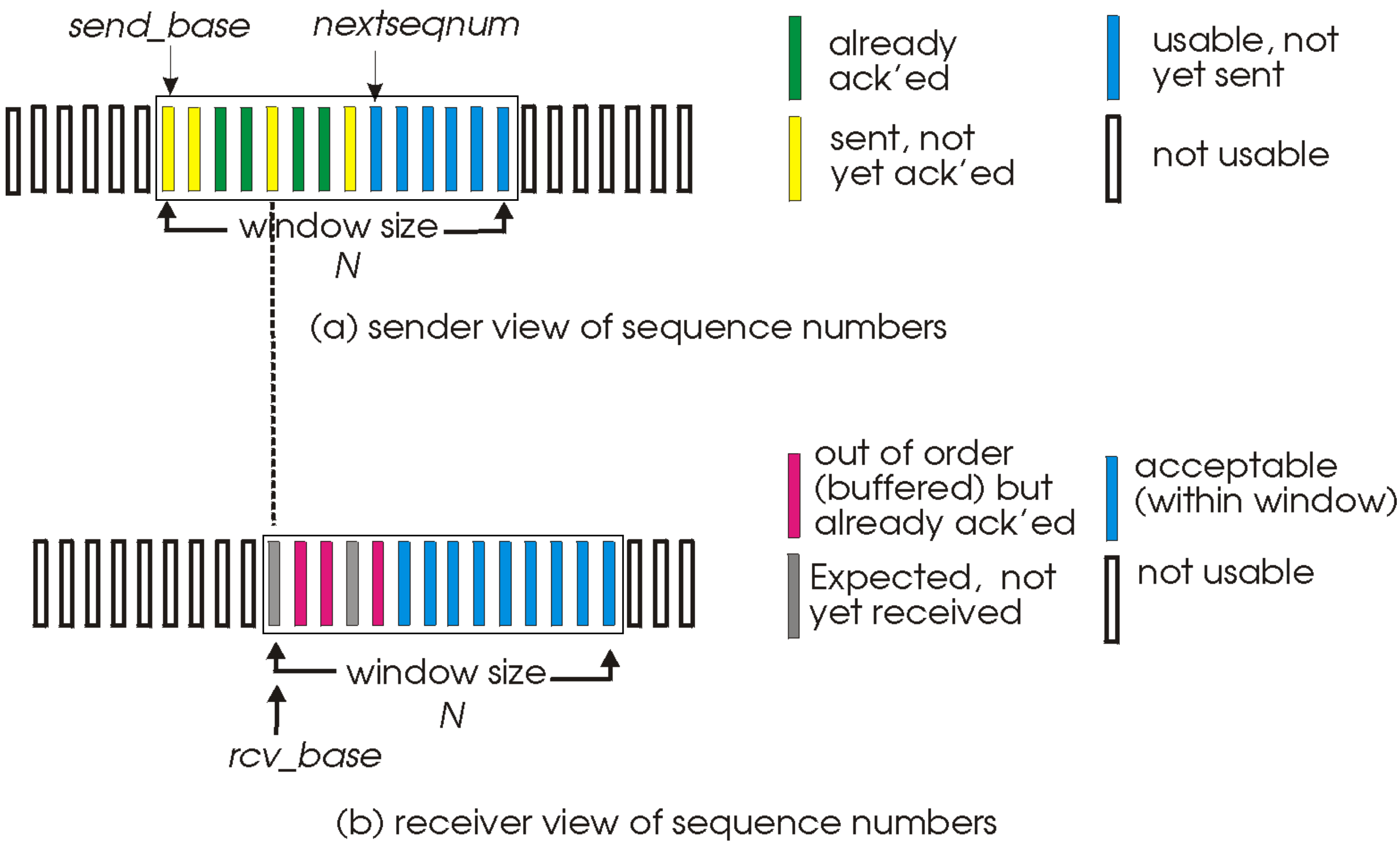
- It is proposed to overcome the unnecessarily retransmitting the error-free packets.
  - having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver.
- Concepts:
  - Sender window size
  - Sequence number
  - Not cumulative acknowledgement
  - How to deal with out-of-order packets
  - Timer and timeout

http://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_6/video_applets/SRindex.html

Selective repeat

- Window size N
  - Data is received from above layer. Label each packet with a sequence number.
  - if next available seq # in window, send pkt
- ACK is not cumulative
  - Receiver *individually* acknowledges all correctly received pkts (ACK is not cumulative)
  - Mark packet n as received
  - If n is the smallest unACKed pkt, advance window base to next unACKed packet with the smallest seq #. After moving window, if there are untransmitted packets with sequence numbers within the window, these packets are transmitted
- How to deal with the out-of-order packets:
  - Buffer is needed for eventual in-order delivery packets to upper layer
- Timer and timeout
  - Sender sets timer for each unACKed pkt (multiple timers)
  - If packet n is timeout, resend pkt n and restart timer

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

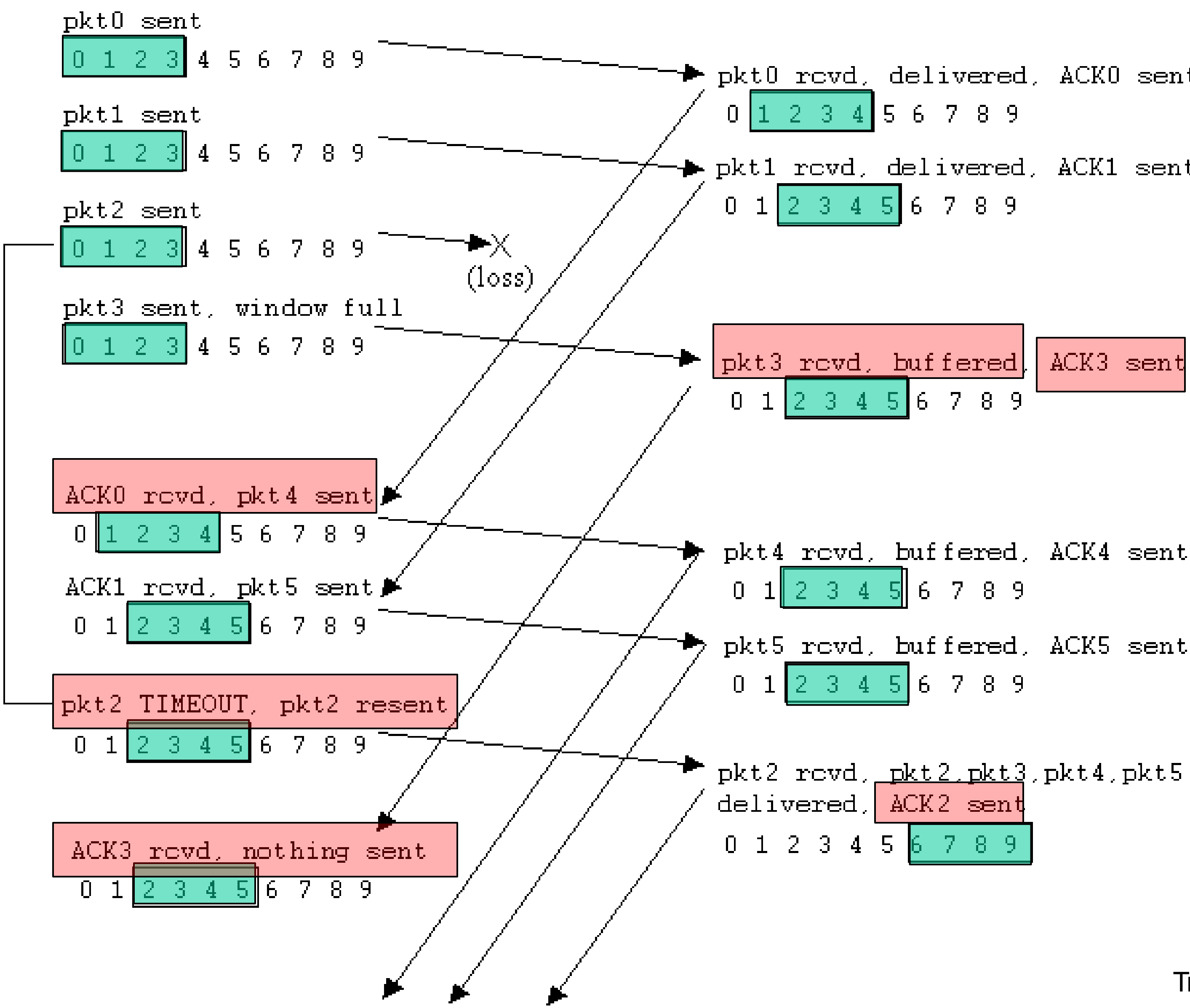
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

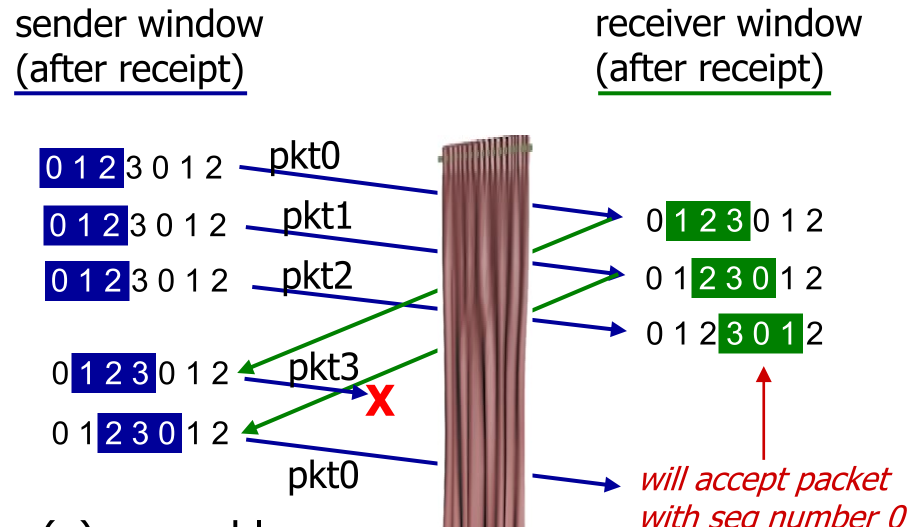
Selective Repeat Example: Window Size N=4



Selective repeat: dilemma

example:

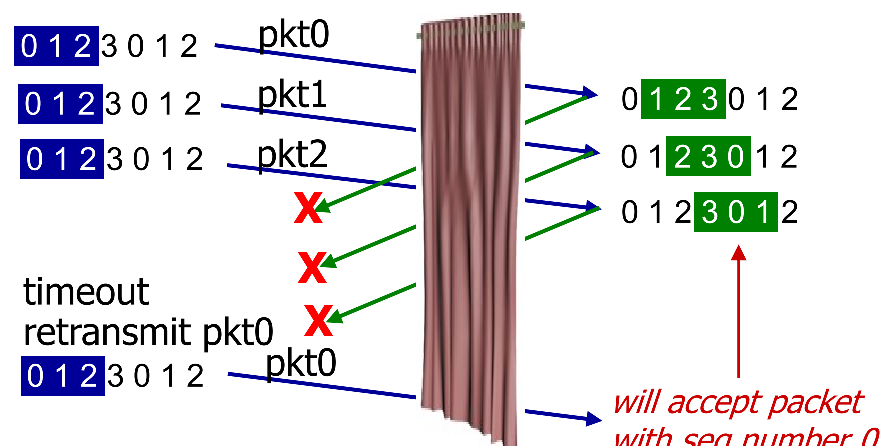
- seq #'s: 0, 1, 2, 3
- window size=3



(a) no problem

- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



(b) oops!

Q: what relationship between seq # size and window size to avoid problem in (b)?

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

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Comparison

Protocol	Stop and Wait	Go-Back-N	Selective Repeat
Bandwidth Utilization	Low	Medium	High
Maximum sender window size	1	the size of sequence number space	half of the size of the sequence number space
Maximum receiver window size	1	1	half of the size of the sequence number space
Pipelining	Not implemented	Implemented	Implemented
Out-of-order packets	Discarded	Discarded	Buffered
Cumulative ACK	N.A	Yes	No

Mechanism	
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.
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.

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

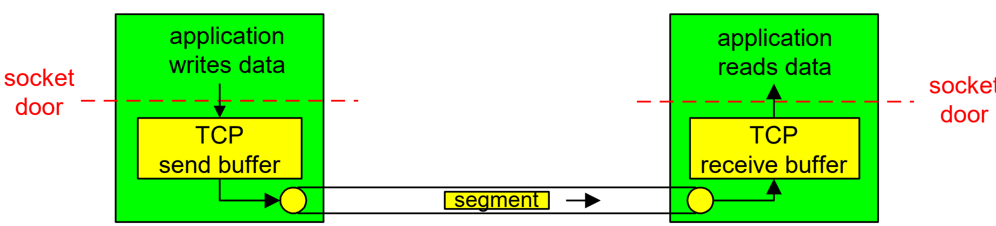
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
- pipelined:
  - TCP congestion control
  - TCP flow control
- Send & receive bufferers



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (application layer data)
- connection-oriented:
  - TCP setup: three-way handshaking (exchange of control msgs) inits sender, receiver state before data exchange
  - TCP teardown: close a TCP connection

TCP segment structure

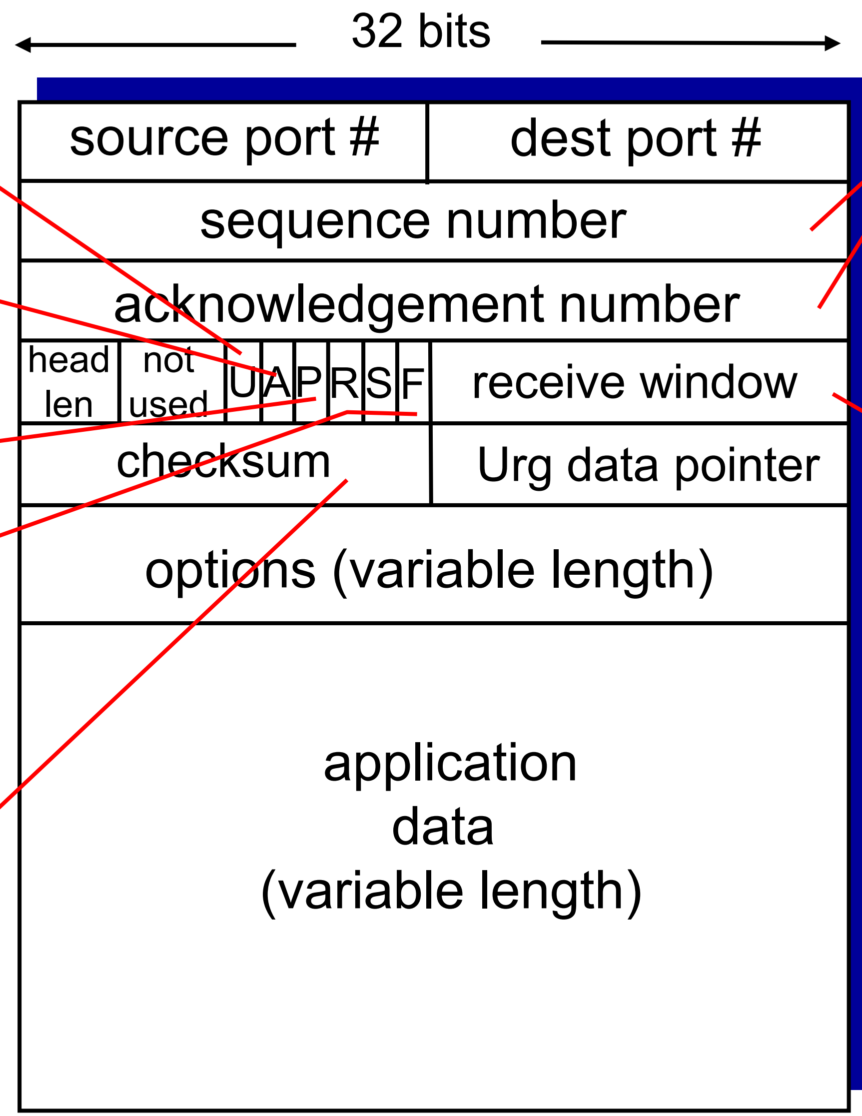
URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet
checksum
(as in UDP)



counting
by bytes
of data
(not segments!)

the
number of
bytes
rcvr willing
to accept

MSS limits the maximum size of a segment's application layer **data field**

the length of the typical TCP header is 20 bytes.)

TCP Reliable Data Transfer

- After a TCP connection is established, the two application processes can send data to each other.
- Client app process passes a stream of data through the socket(the door of the process), then TCP running in the client handles these data (direct data to send buffer, break these data into chunks, pair each chunk of data with a TCP header to form a TCP segment, then pass them down to the network layer).

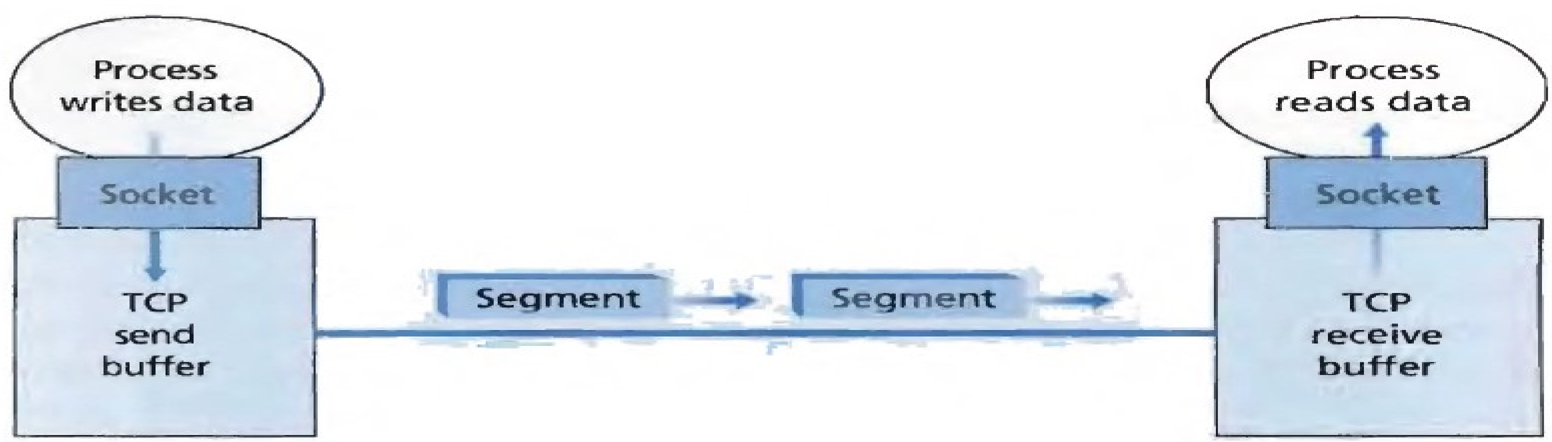


Figure 3.28 ♦ TCP send and receive buffers

TCP Reliable Data Transfer

- Both the client and server allocate buffers to hold incoming and outgoing data to be transmitted or received throughput TCP protocol.
- Sending buffer: buffer the outgoing data.
- Receiving buffer: buffer the incoming data.
  - The TCP does not know when the application will ask for any received data. TCP buffers incoming data so it is ready when the applications require them.

How to Decide the Sequence No.

- TCP views data as unstructured, but ordered stream of bytes.
- We label these bytes with integer numbers.
- Sequence number is the number of the first data in the segment in unit of bytes.
- Sequence numbers are over bytes, not segments.
- Example:
  - The data file consisting of 500,000 bytes, MSS is 1000bytes, the initial sequence number is 0.
  - TCP constructs 500 segments; the sequence number set in the first, second, third segments is 0, 1000, 2000, respectively.

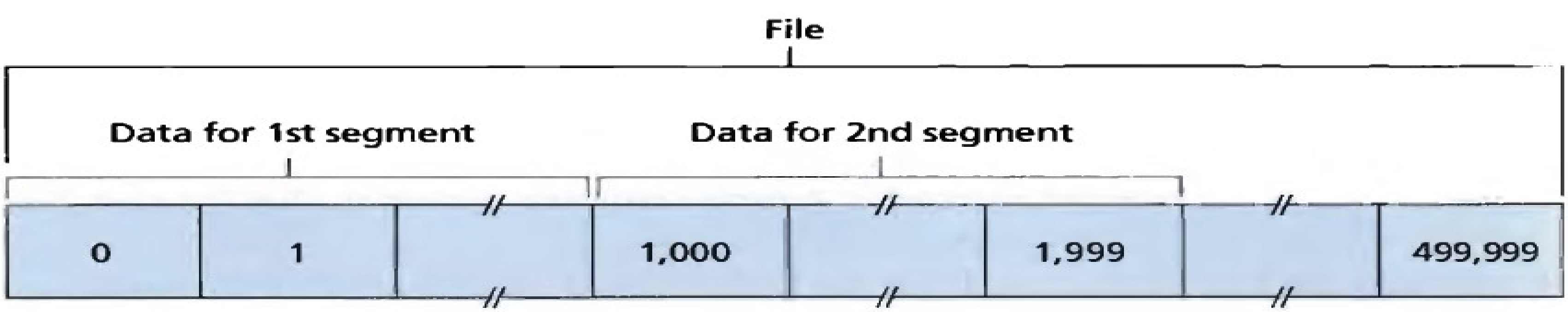


Figure 3.30 ♦ Dividing file data into TCP segments

How to Decide the ACK No.

- Acknowledgement number – At host B(A), ACK number is the sequence number of the next byte that Host B(A) is expecting from host A(B).
- Example:
  - Host B received the 1st segment (i.e., all bytes numbered 0 through 999) from Host A and is waiting for all the subsequent segments.
  - In this case, Host B puts 1000 in the ACK number field of the segment it sends to Host A.

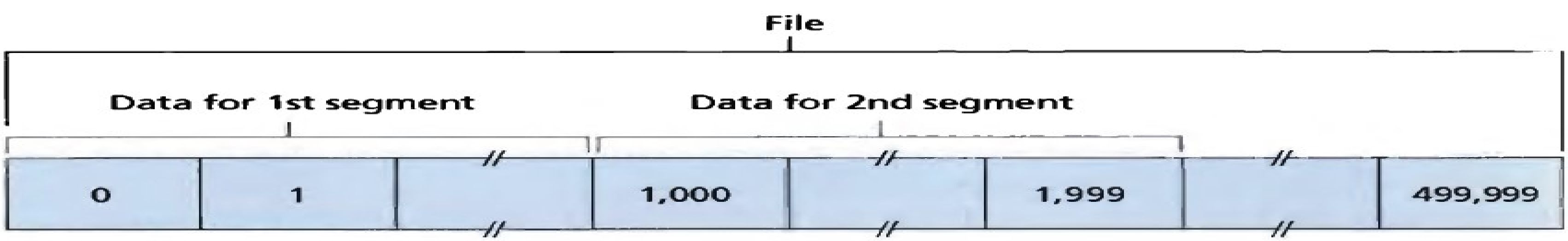


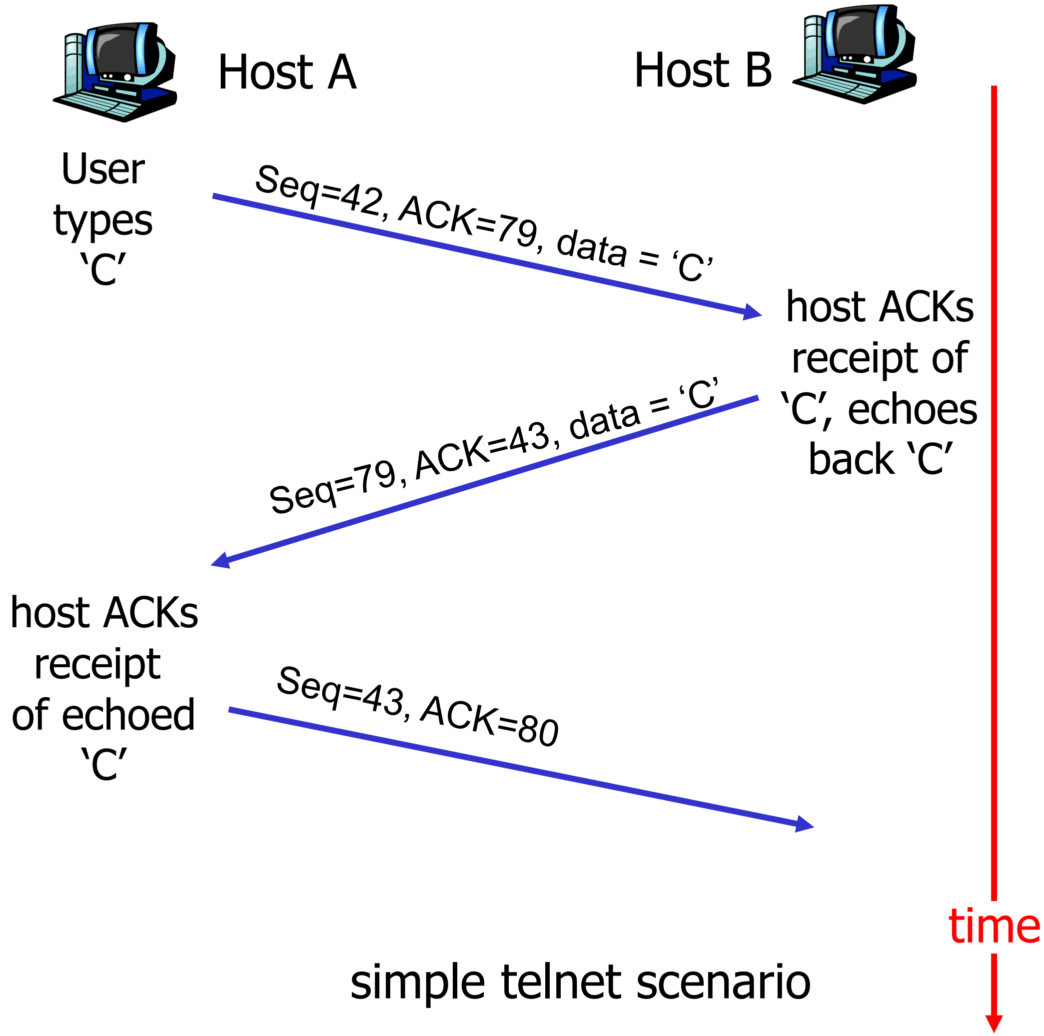
Figure 3.30 • Dividing file data into TCP segments

TCP seq. #'s and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementer



simple telnet scenario

Timer and Timeout

- TCP uses single timer
- Restart timer is triggered when
  - A segment is sent and the timer is not running for any other segment.
  - Timeout event happens.
  - ACK is received.

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss, lead to large data transfer delay

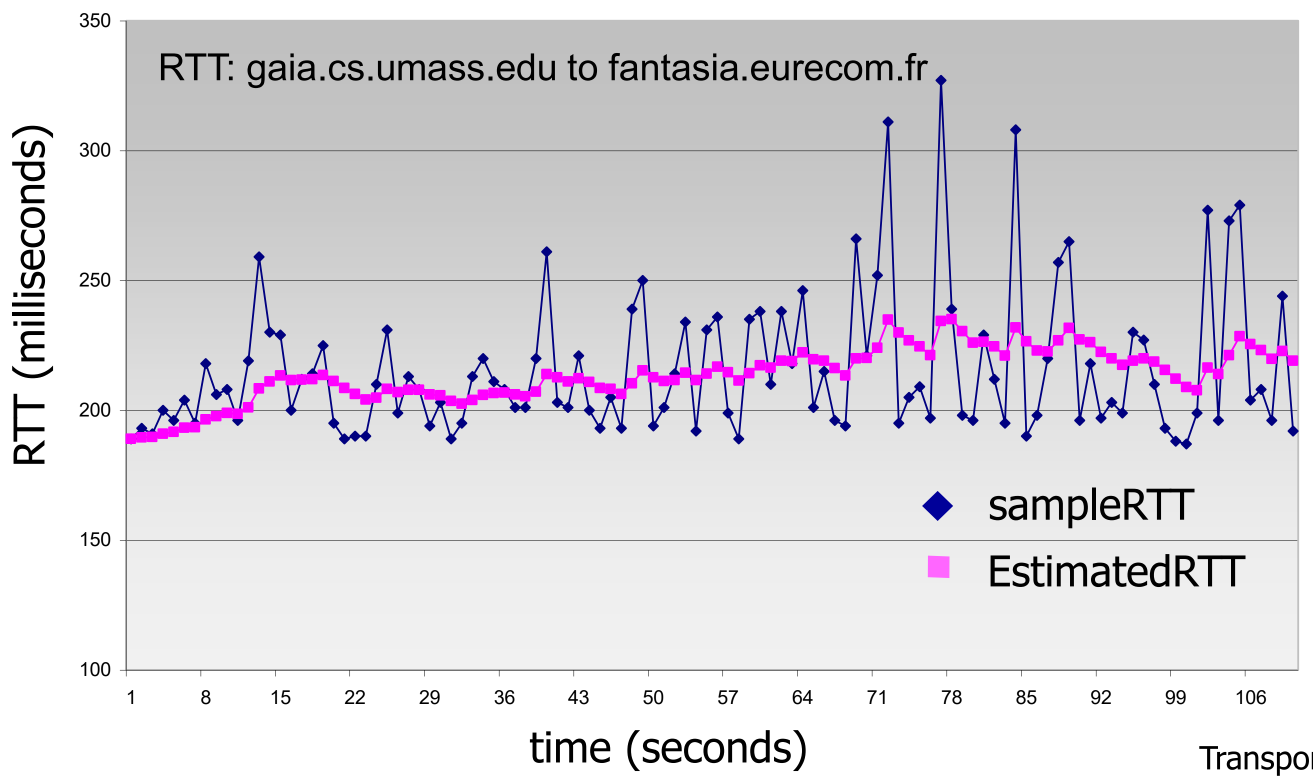
Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|
         (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

TCP sender events:

Three major events related to data transmission and retransmission in the TCP sender: data received from application above; timer timeout; and ACK receipt.

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

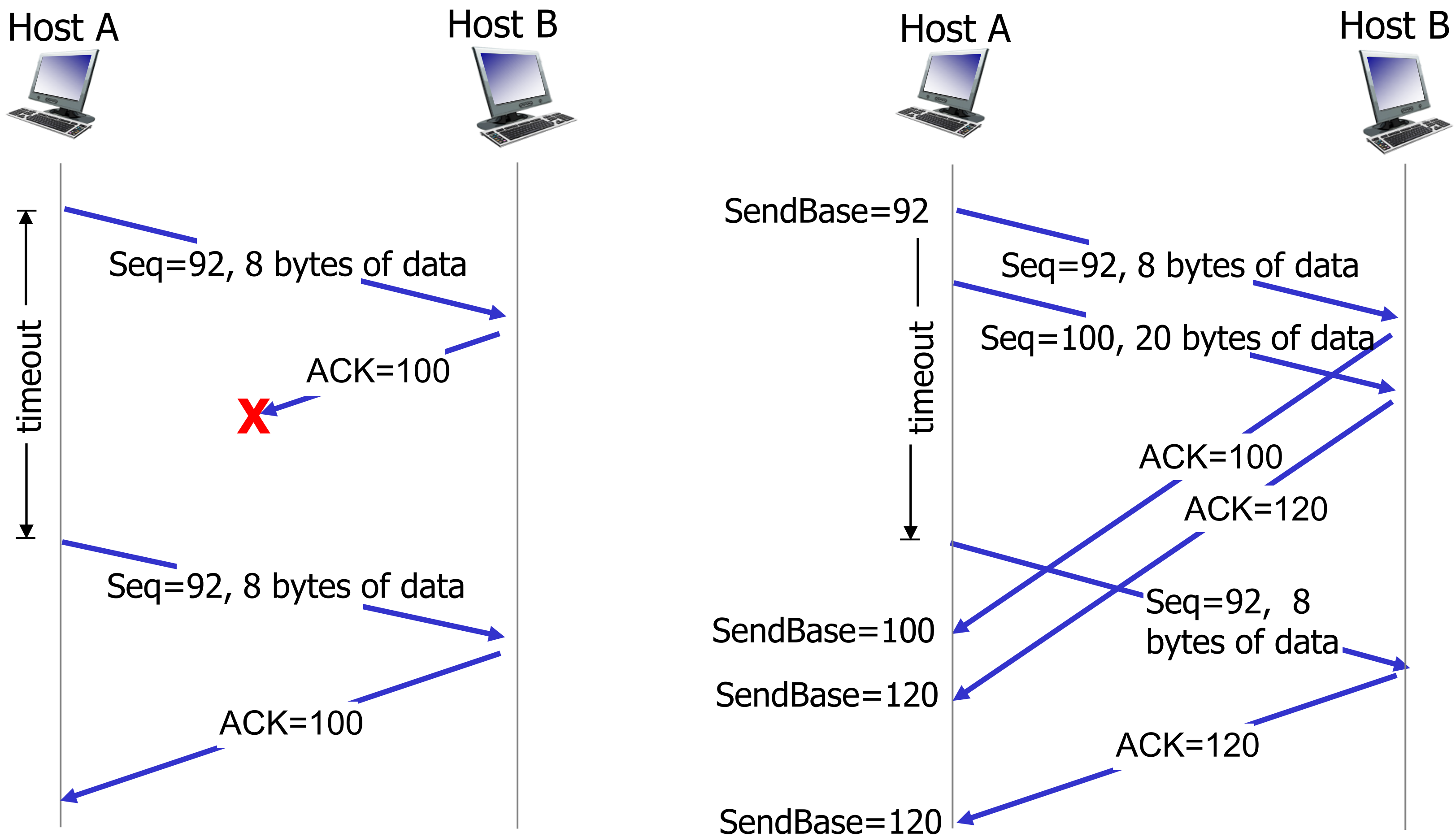
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

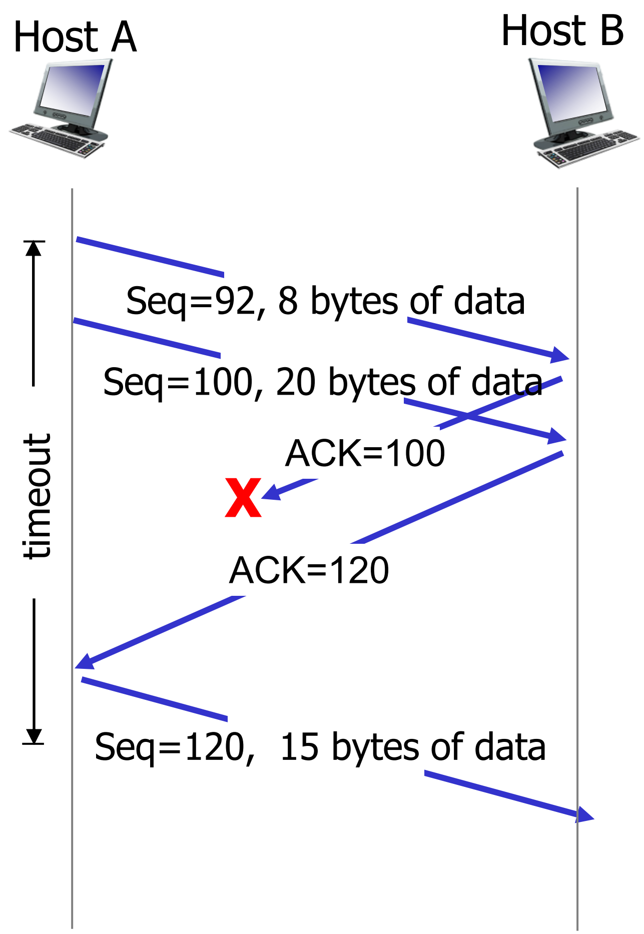
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK

Host A therefore knows that Host B has received *everything* up through byte 119; so Host A does not resend either of the two segments.

TCP fast retransmit

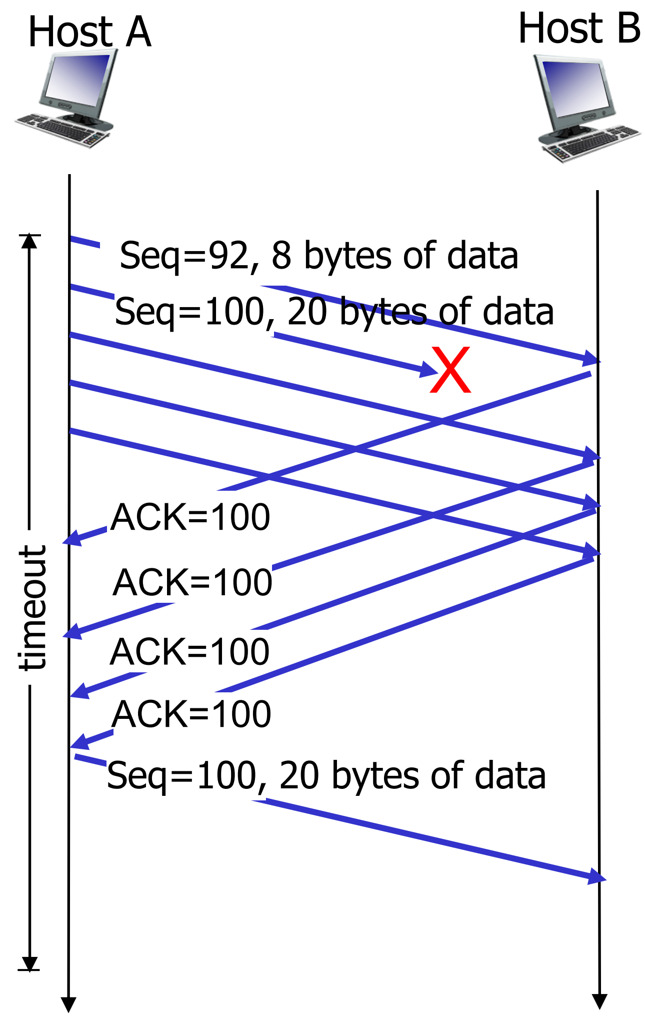
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Buffers and Buffer Overflow

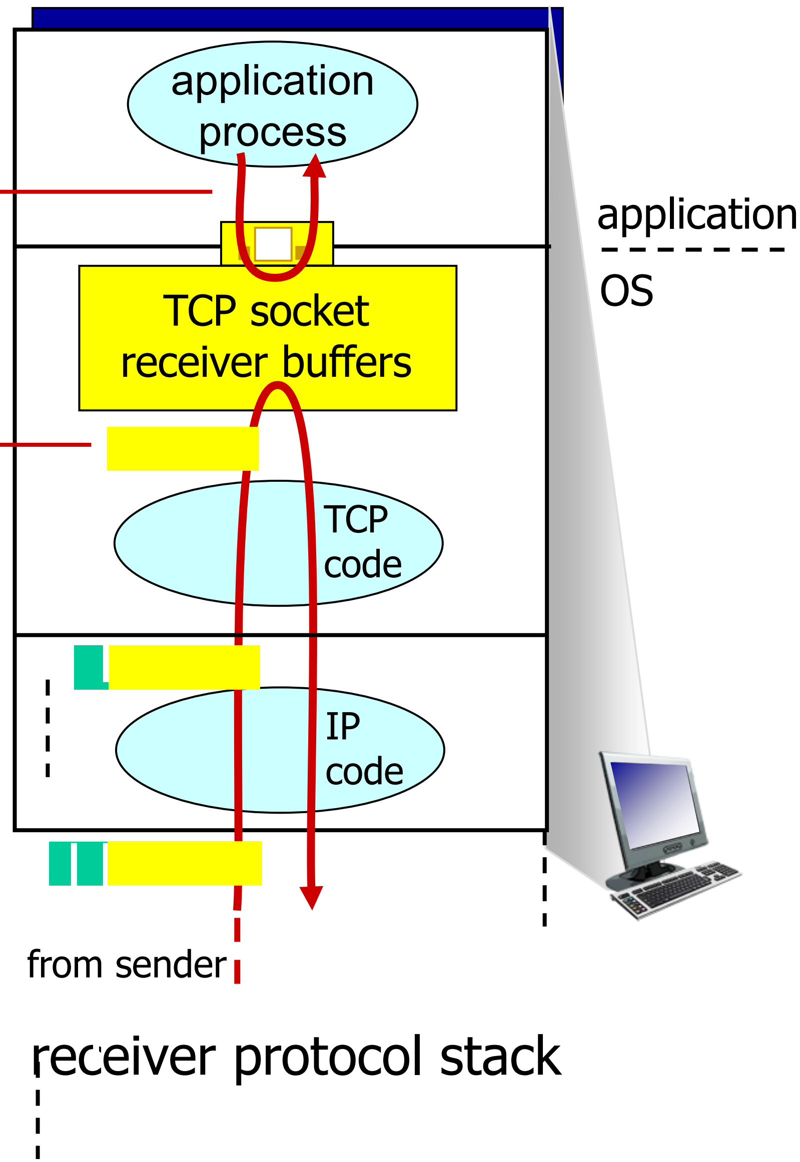
- Both the client and server allocate buffers to hold incoming and outgoing data
- Sending buffer:
  - The application gives the TCP some data to send.
  - The data is put in a sending buffer.
  - The TCP won't accept data from the application unless

  there is buffer space.
- Receiving buffer
  - When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer.
  - The TCP does not know when the application will ask for the received data. TCP buffers incoming data so it is ready when the application ask for it.
- Buffer overflow: the sender may easily overflow the receive buffer if the sender sends too much data too quickly and the application at the receiver is relatively slow at reading the data.

TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



flow control

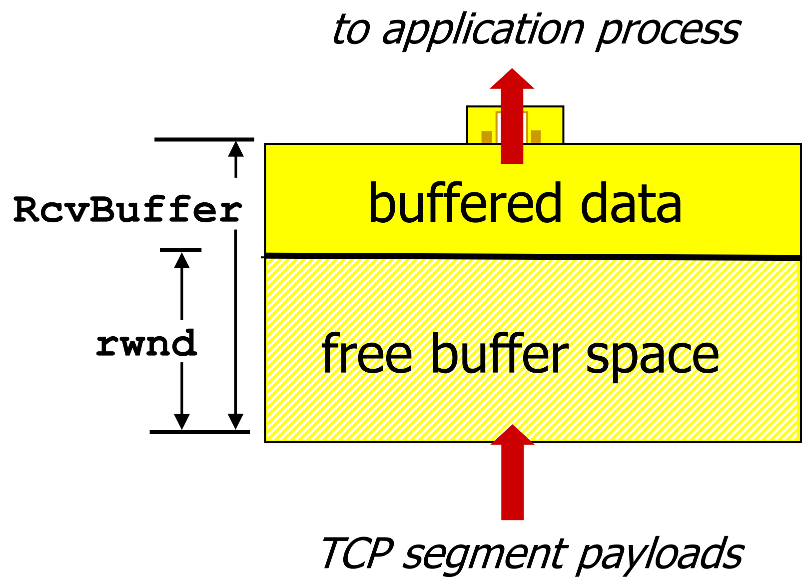
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

Speed Matching

matching send rate to receiving application's drain rate

TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

unused buffer space:
= rwnd
= RcvBuffer-[LastByteRcvd - LastByteRead]

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

How is a TCP Connection Established

- An application process running in host A wants to initiate a connection with another application process running in host B
  - The application process that initiates the connection is called client process, and the other application process is called server process.
  - The client application process first informs the client transport layer that it wants to establish a connection to a process in the host B using a program command.

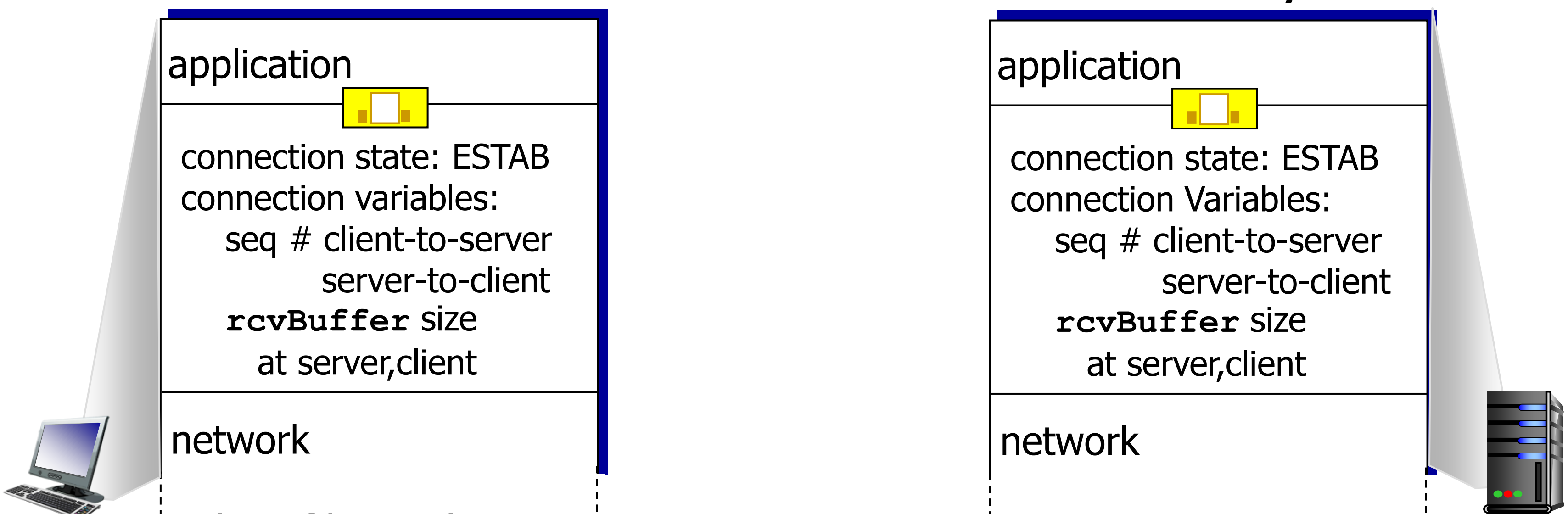
```
Socket clientSocket
= new Socket ("hostname", "portNumber")
```

  - Then, client transport layer proceeds to establish a TCP connection through three-way handshaking.

Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters : seq.#s, buffers, flow control info (e.g. RcwWindow)
- Client: connection initiator; Server: contacted by client.




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

```
Socket connectionSocket =
  welcomeSocket.accept();
```

Three way handshaking

Step 1:client host sends TCP SYN segment to server

specifies initial seq #

no data

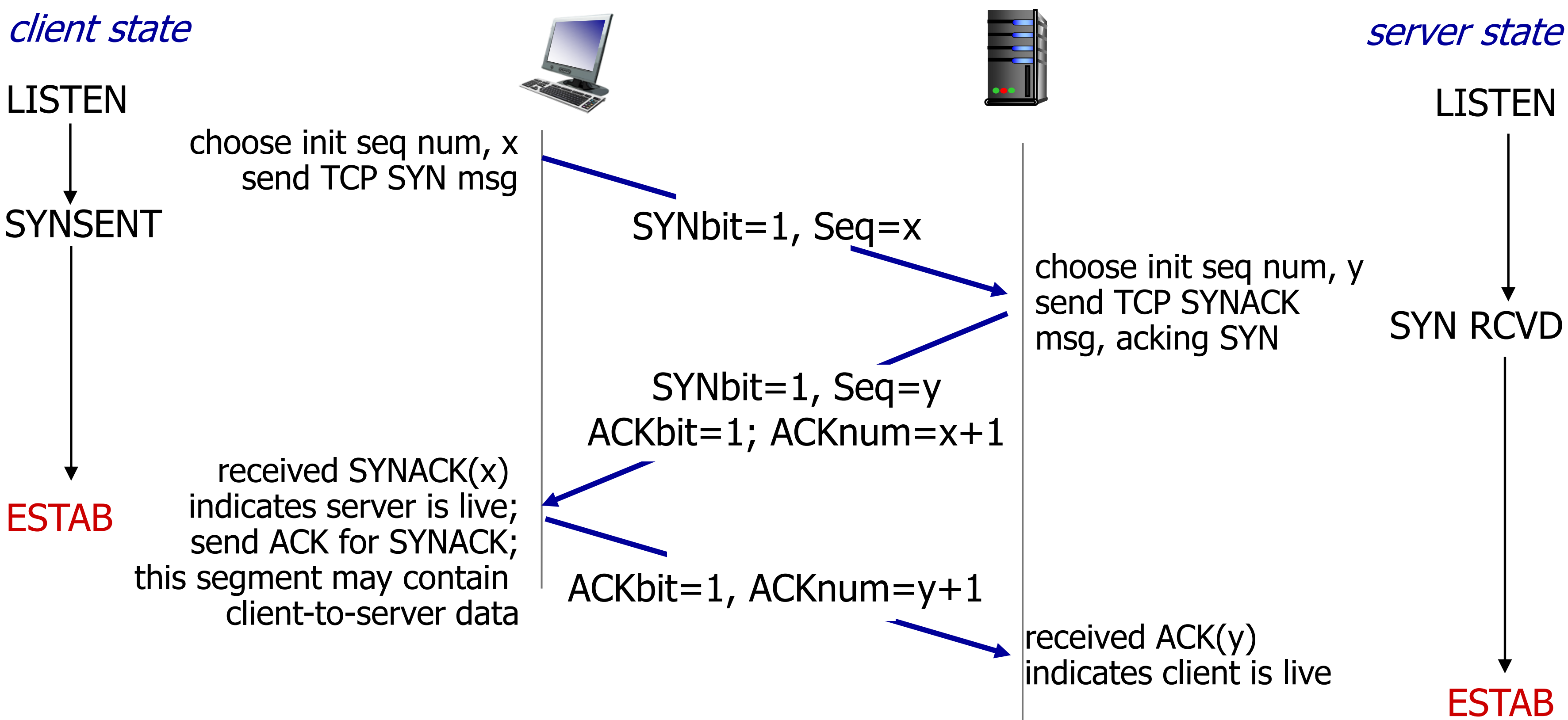
Step 2:server host receives SYN, replies with SYNACK segment

server allocates buffers

specifies server initial seq. #

Step 3:client receives SYNACK, replies with ACK segment, which may contain data

TCP 3-way handshake



The first two segments carry no payload, that is, no application-layer data; the third of these segments may carry a payload.

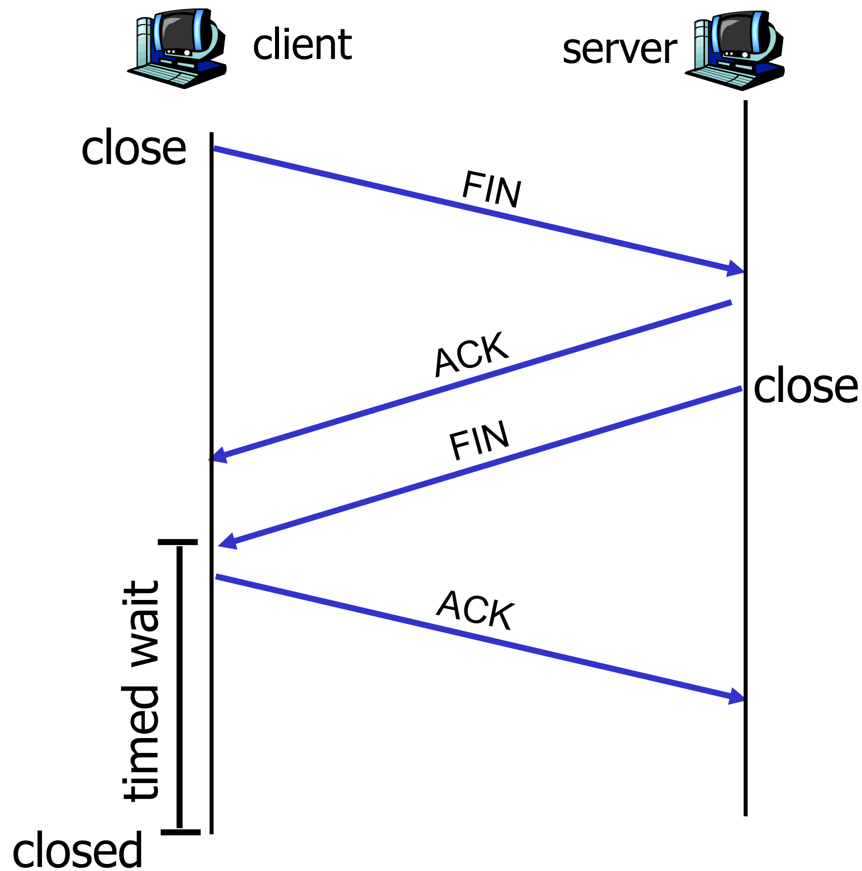
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



Note that the server could also choose to close the connection.

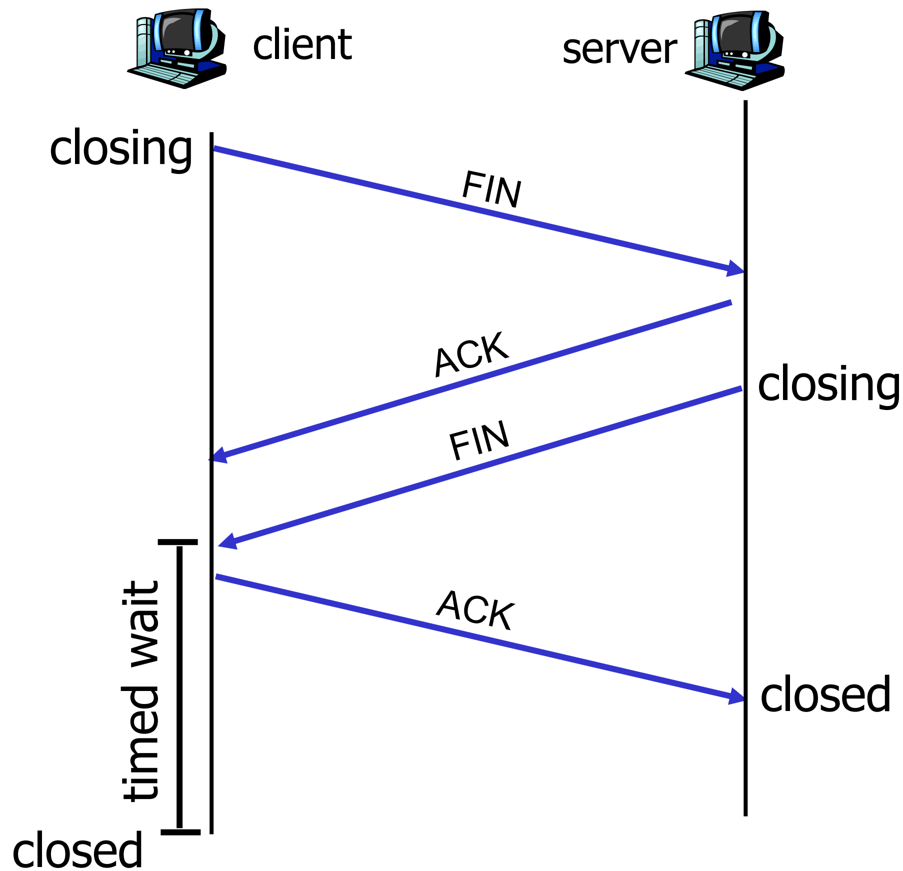
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

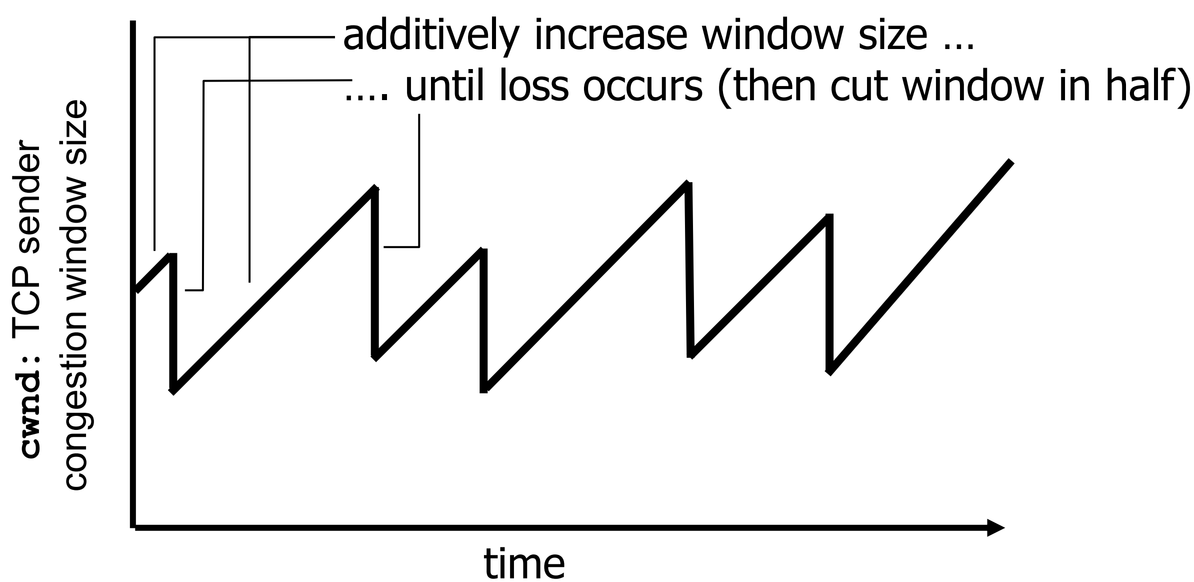
3.6 principles of congestion control

3.7 TCP congestion control

TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Congestion and TCP congestion control

congestion: informally: "too many sources sending too much data too fast for *network* to handle"

- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- Mechanisms are needed to shrink the sending rate in the face of network congestion.
- TCP congestion control mechanism
  - How does a TCP sender limit the sending rate?
  - How does a TCP sender perceive the degree of network congestion?
  - What algorithm should the TCP sender use to adjust the sending rate?

TCP Congestion Control

- Congestion window (cwnd): a parameter to limit the transmission rate in sender
  - Sender limits transmission: the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd

LastByteSent-LastByteAcked ≤ min{CongWin, rwnd}

Roughly, sender's send rate is

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

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.

- `CongWin` is dynamic. It is a function of perceived network congestion.

TCP Congestion Control

- How does sender perceive congestion:
  - Loss event (a timeout or 3 duplicate ACKs)
- TCP sender reduces rate (congestion window) after loss event.
- TCP is self-clocking: TCP uses acknowledgements to trigger (clock) its adjustment of congestion window size
- TCP congestion-control algorithm: three mechanisms
  - Slow start (initial phase or after RTO (retransmission timeout)
  - Additive-increase, multiplicative-decrease (AIMD)
  - Reaction to timeout events: conservative after timeout event (CW=1 MSS (maximum segment size))

TCP congestion control: Slow Start

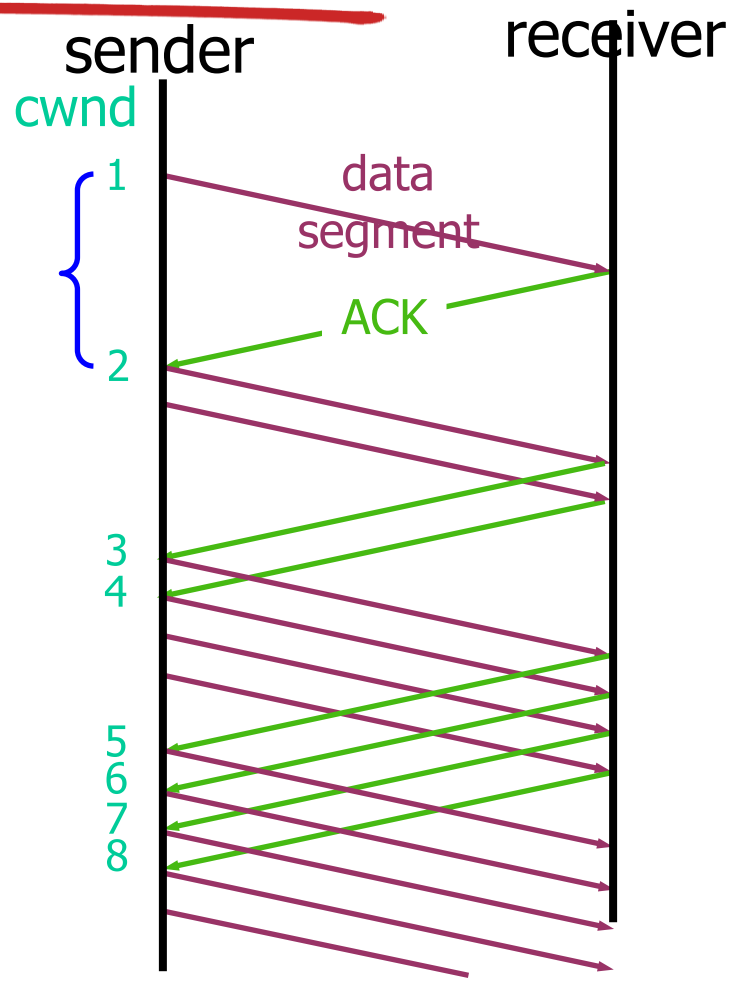
- "Slow Start": when connection begins, increase rate exponentially until first loss event

  Initially: CW = 1 MSS

  example: MSS = 500 bytes & RTT = 200 msec

  initial rate = 20 kbps
- increase rate exponentially until first loss event or when threshold reached
  - double **cwnd** every RTT
  - done by incrementing **cwnd** by 1 for every ACK received

  $CW \leftarrow CW + 1$



double **CongWin** every RTT
done by incrementing **CongWin** for every ACK received

http://histrory.visualland.net/tcp_swnd.html

Transitioning into/out of slowstart

`ssthresh:` **`cwnd`** threshold maintained by TCP

- on loss event: set **`ssthresh`** to **`cwnd/2`**
  - **`Loss can be indicated by time out or 3 duplicated ACKs`**
  - remember (half of) TCP rate when congestion last occurred
- when **`cwnd`** >= **`ssthresh`**: transition from slowstart to congestion avoidance phase

TCP: congestion avoidance

- when **`cwnd > ssthresh`** grow **`cwnd`** linearly
  - increase **`cwnd`** by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: **`cwnd = cwnd + MSS/cwnd`** for each ACK received

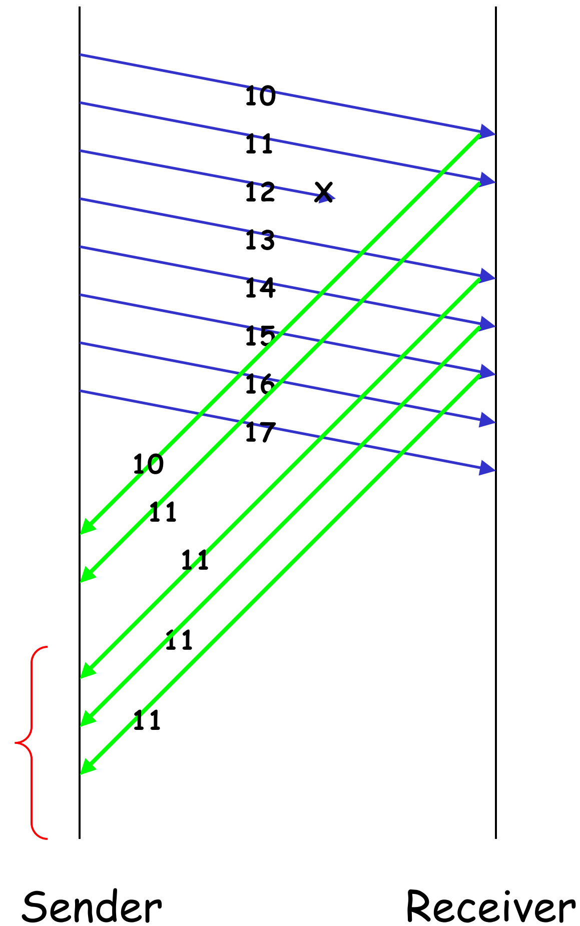
AIMD

- *ACKs:* increase **`cwnd`** by 1 MSS per RTT: additive increase
- *loss:* cut **`cwnd`** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

TCP: detecting, reacting to loss

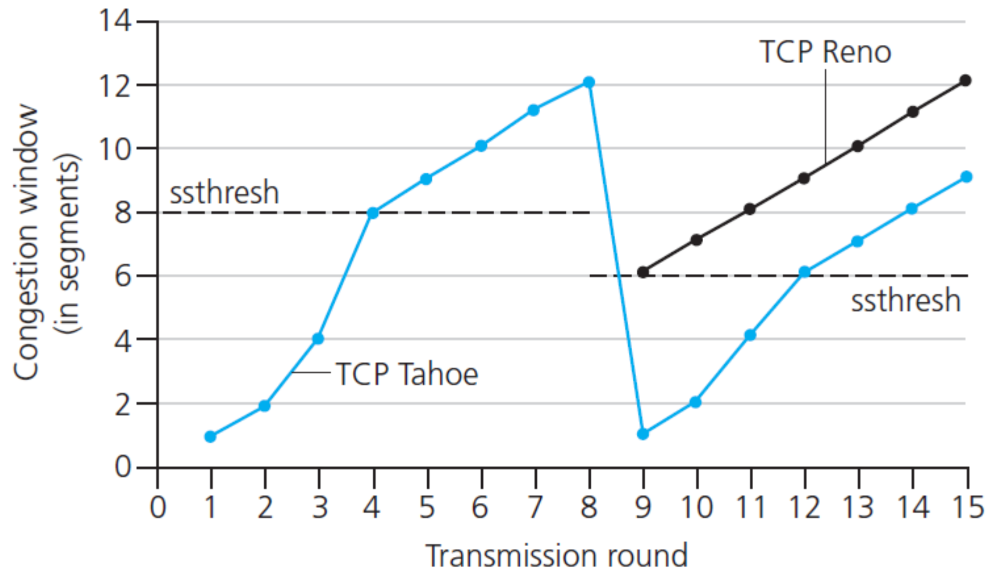
- Option 1: loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold (, then grows linearly
- Option 2: loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly
- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



the threshold is initially equal to 8 MSS.

Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"