Chapter 3
Transport Layer

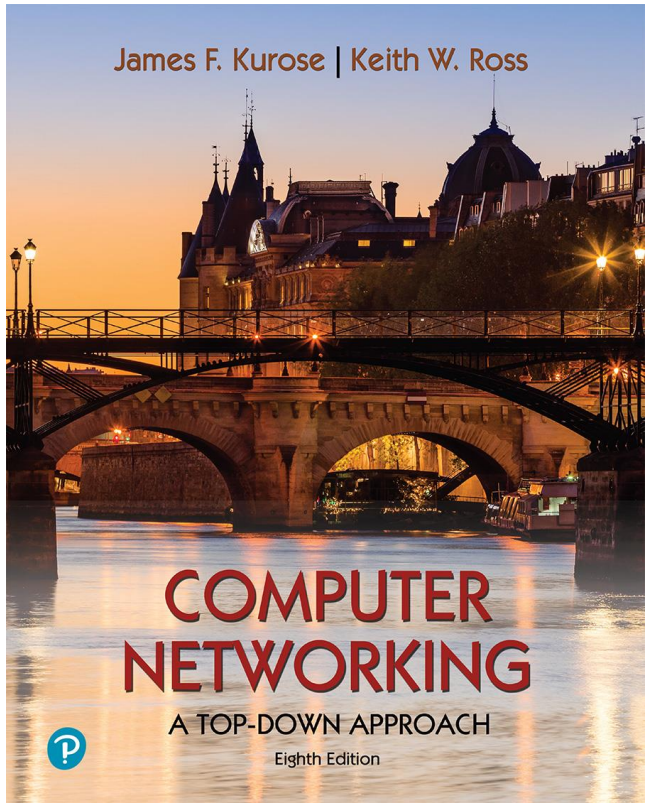
A note on the use of these PowerPoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top-Down Approach

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer:

Multiplexing and Demultiplexing,
Connectionless Transport-UDP: UDP Segment Structure, UDP Checksum,
Go-Back-N, Selective Repeat,
Connection-Oriented Transport-TCP:
The TCP Connection,
TCP Segment Structure,
Round-Trip Time Estimation and Timeout,
Reliable Data Transfer,
Flow Control,
TCP Connection Management,
TCP congestion control.

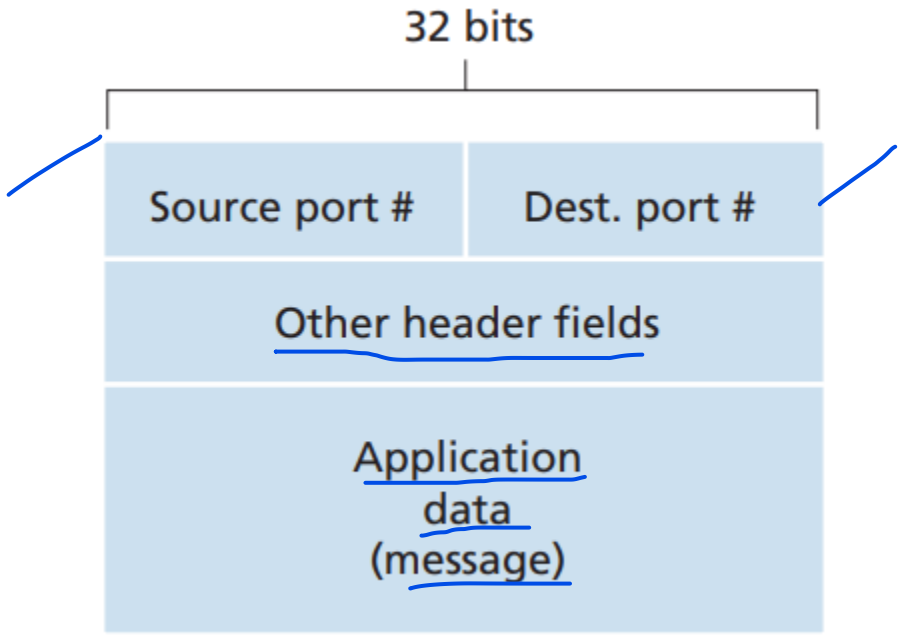
Transport layer: roadmap

- Multiplexing and demultiplexing (3.2)
- Connectionless transport: UDP (3.3)
  - UDP segment structure (3.3.1)
  - UDP checksum (3.3.2)
- Go back N (3.4.3)
- Selective Repeat (3.4.4)
- Connection-oriented transport: TCP (3.5)
  - TCP connection (3.5.1)
  - TCP Segment structure (3.5.2)
  - Rotund trip time estimation and timeout (3.5.3)
  - Reliable data transfer (3.5.4)
  - Flow control (3.5.5)
  - TCP connection management (3.5.6)
- TCP congestion control (3.7)

Chapter 3: roadmap

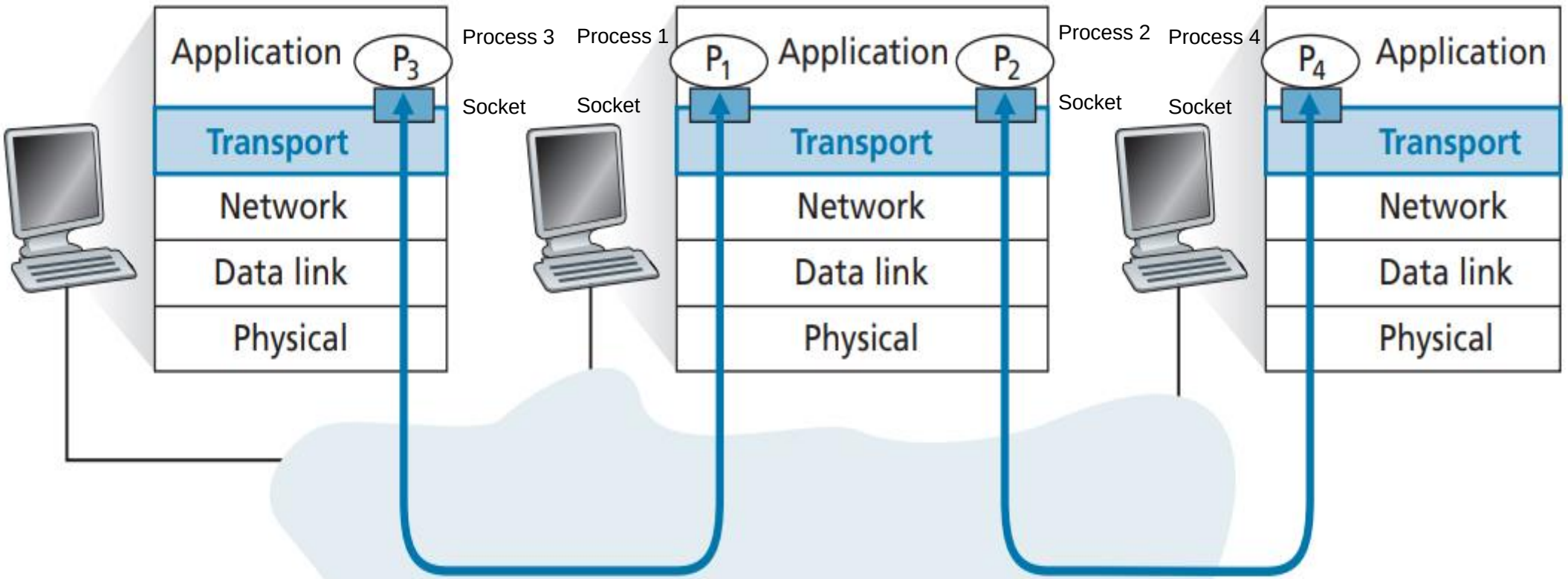
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer (Go-back-N and Selective Repeat)
- Connection-oriented transport: TCP
- TCP congestion control

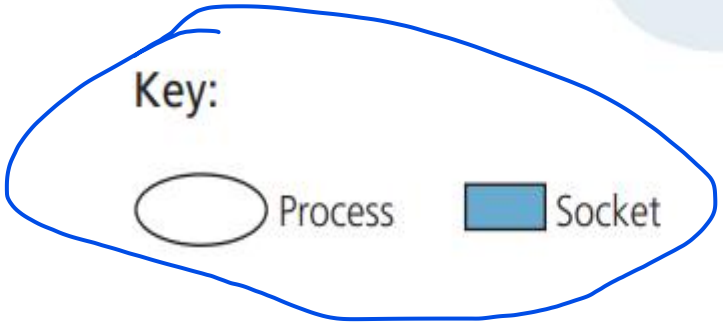
Multiplexing and Demultiplexing



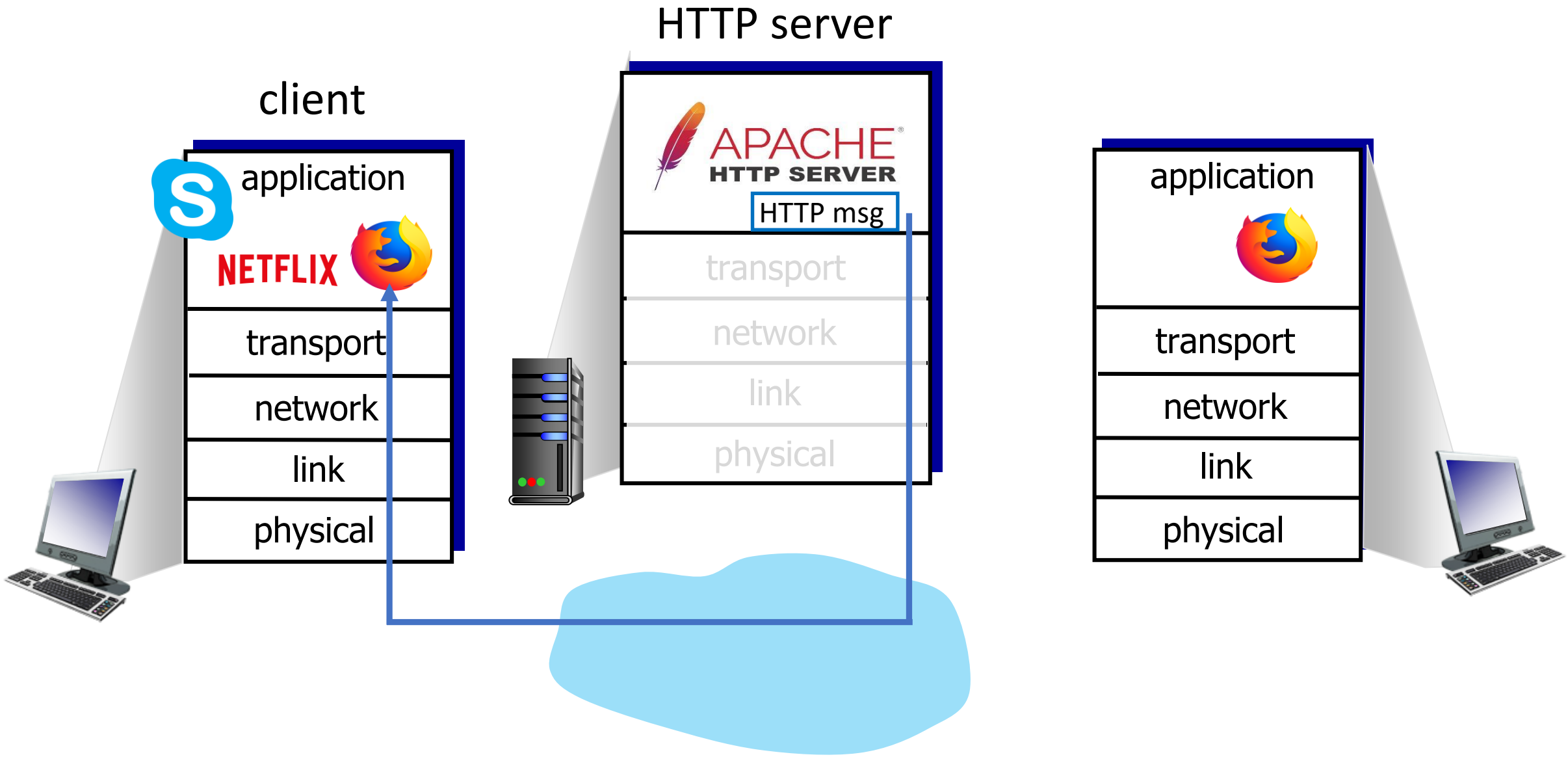
Transport Layer Segment

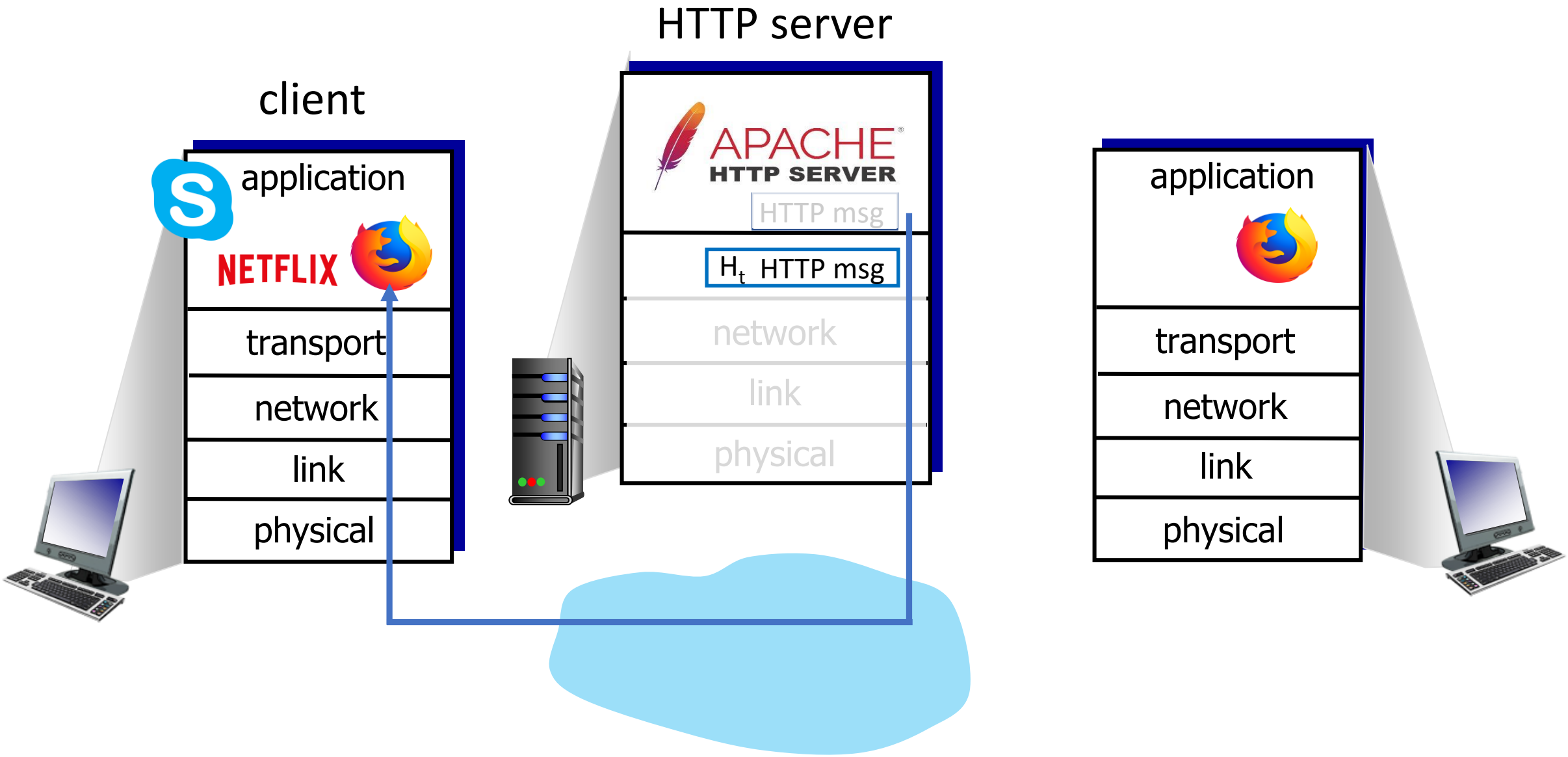
TCP/ UDP Segement

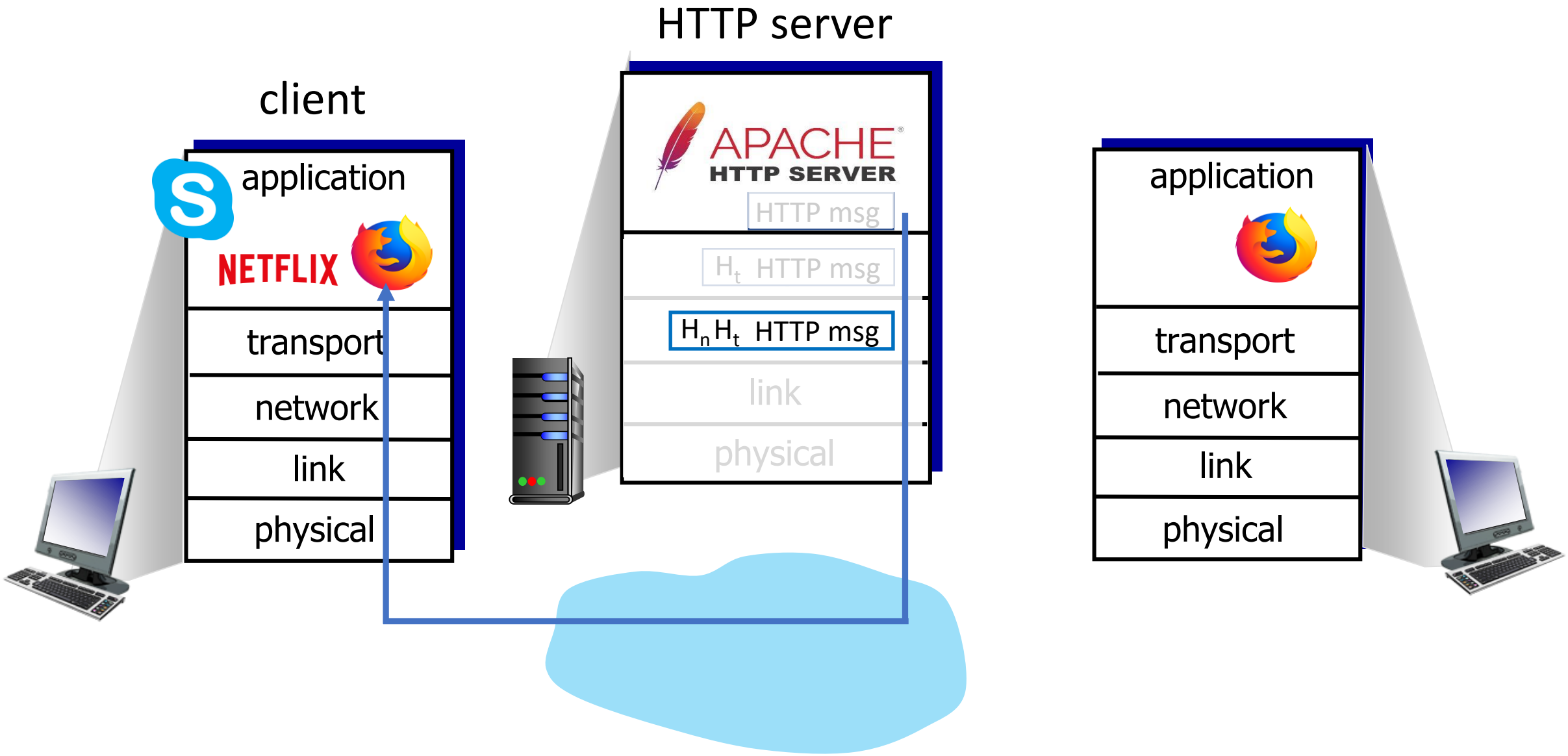


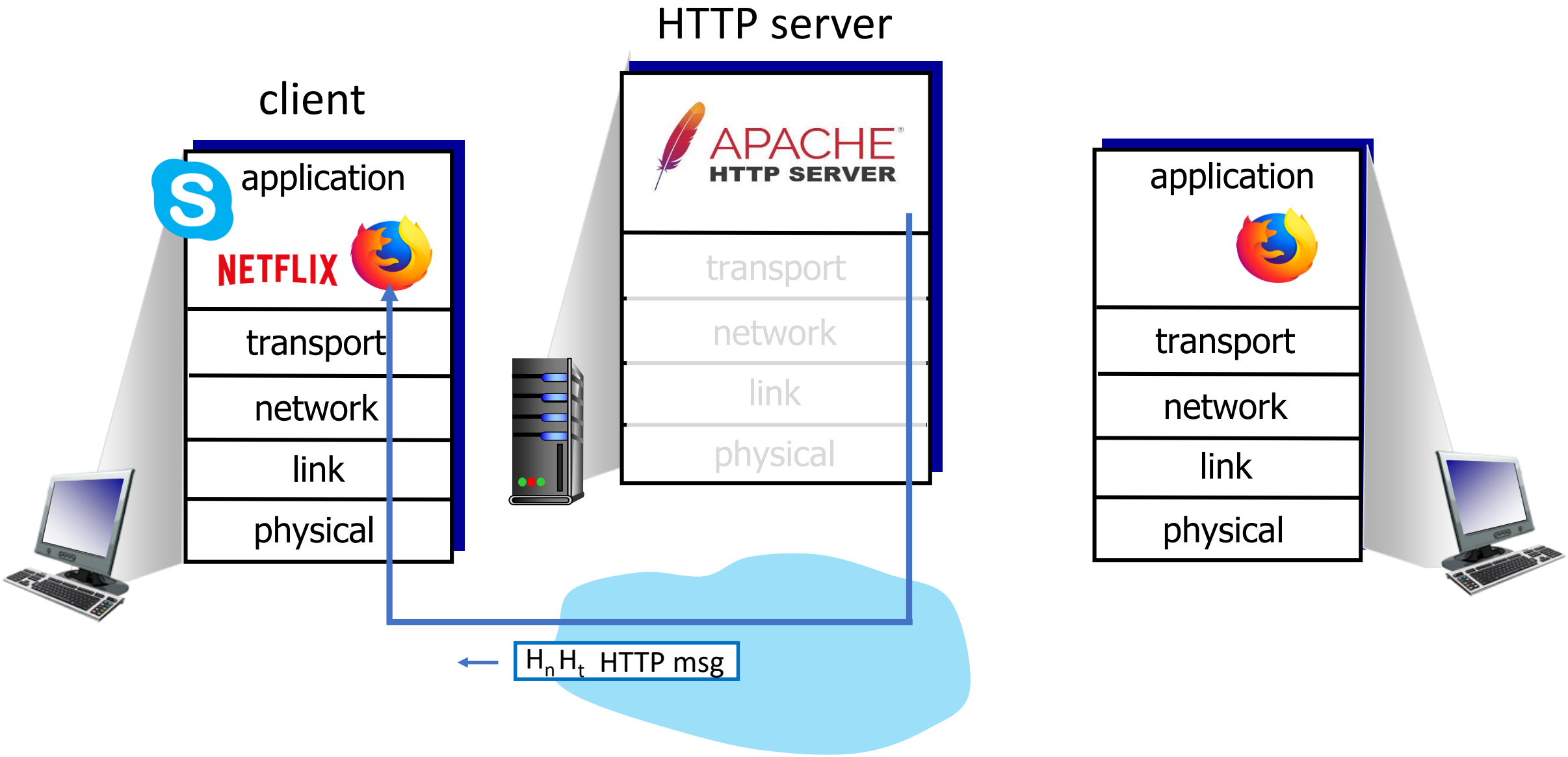


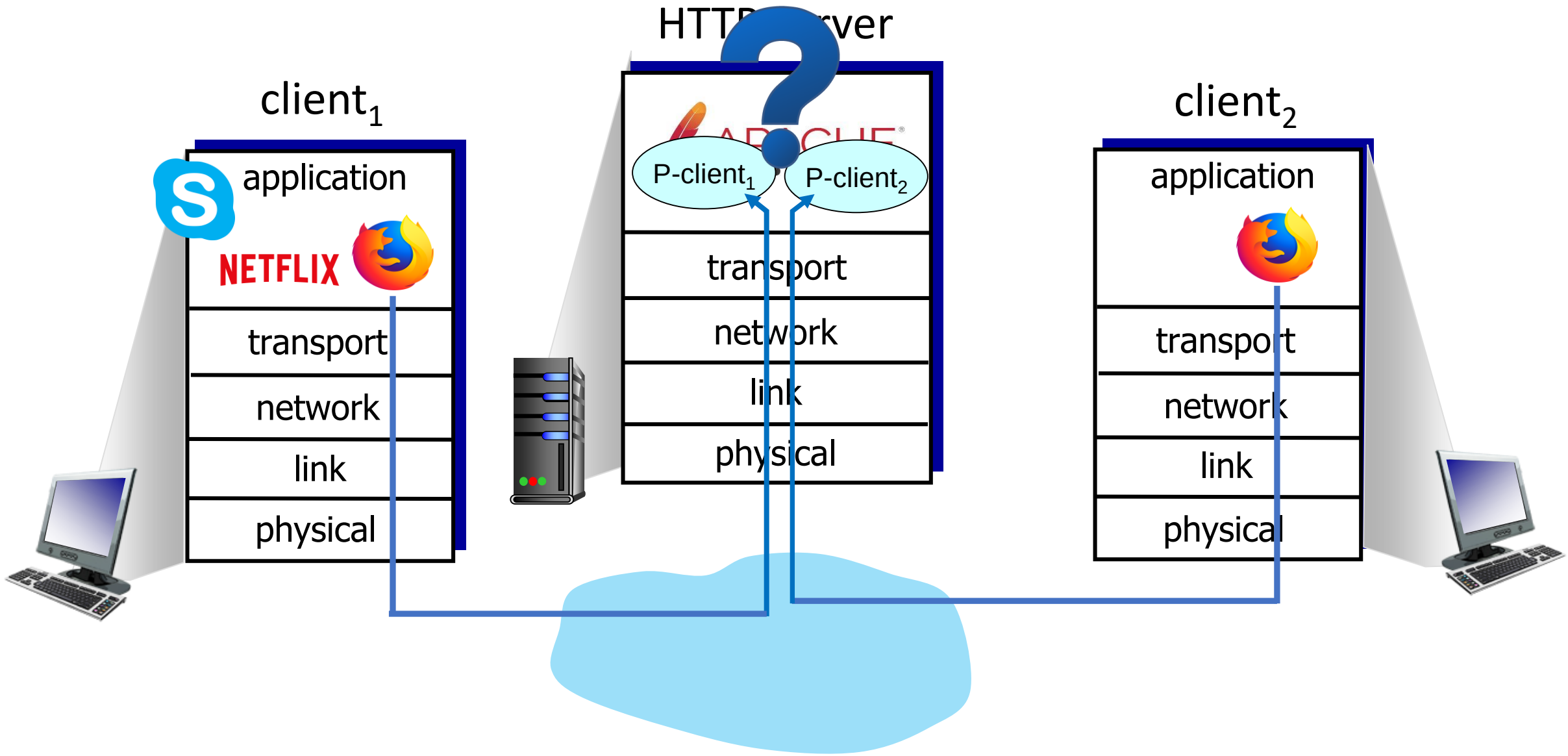
1. Logical communication Channel between Processes- ***Host to Host Delivery service***.
2. Multiplexing and Demultiplexing
3. Congestion Control.
4. Flow Control
5. Error Control


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg


client1
application
NETFLIX
transport
network
link
physical
HTTP server
P-client1
P-client2
transport
network
link
physical
client2
application
transport
network
link
physical

Multiplexing/demultiplexing

The job of delivering the data in a transport-layer segment to the correct socket is called demultiplexing

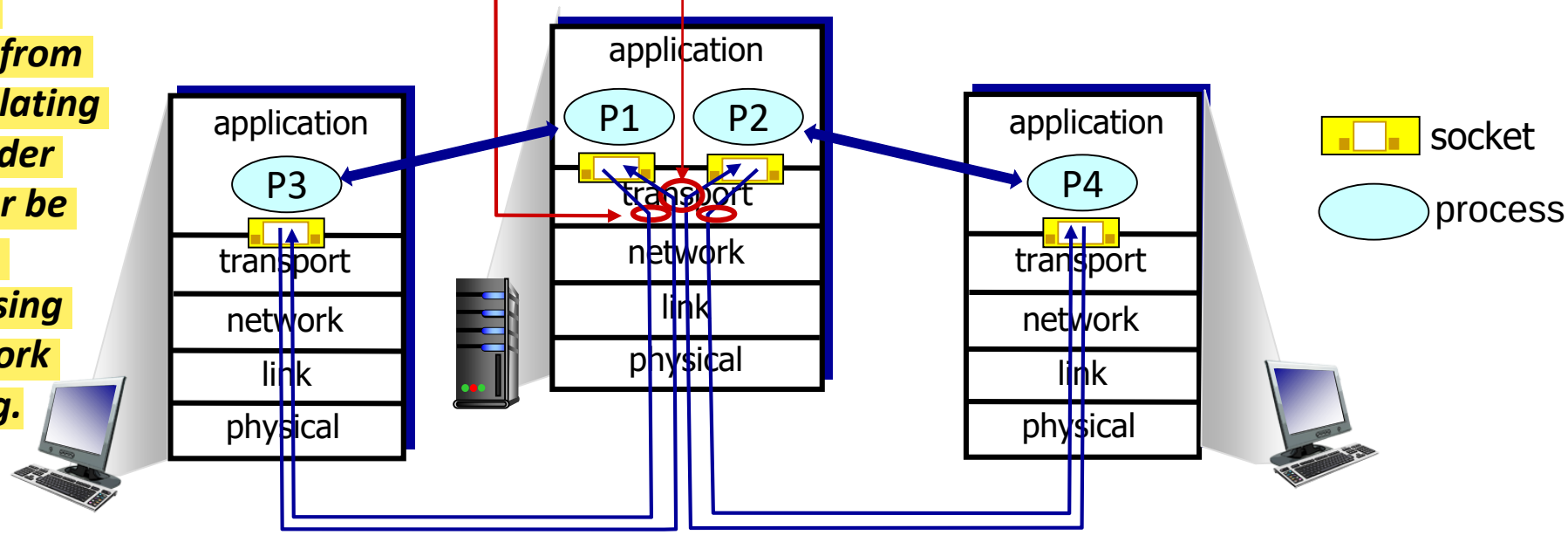
multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

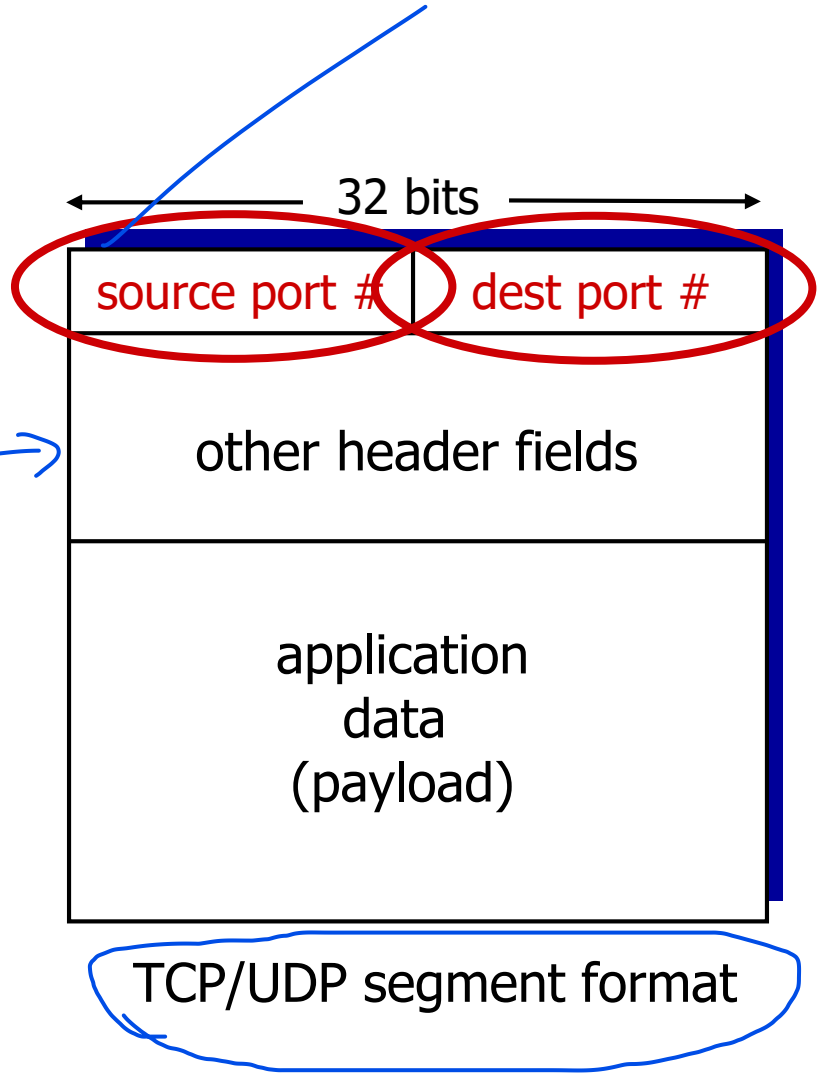
use header info to deliver received segments to correct socket

The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called multiplexing.



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



Connectionless demultiplexing UDP

Recall:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1
 = new DatagramSocket(12534);
```

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

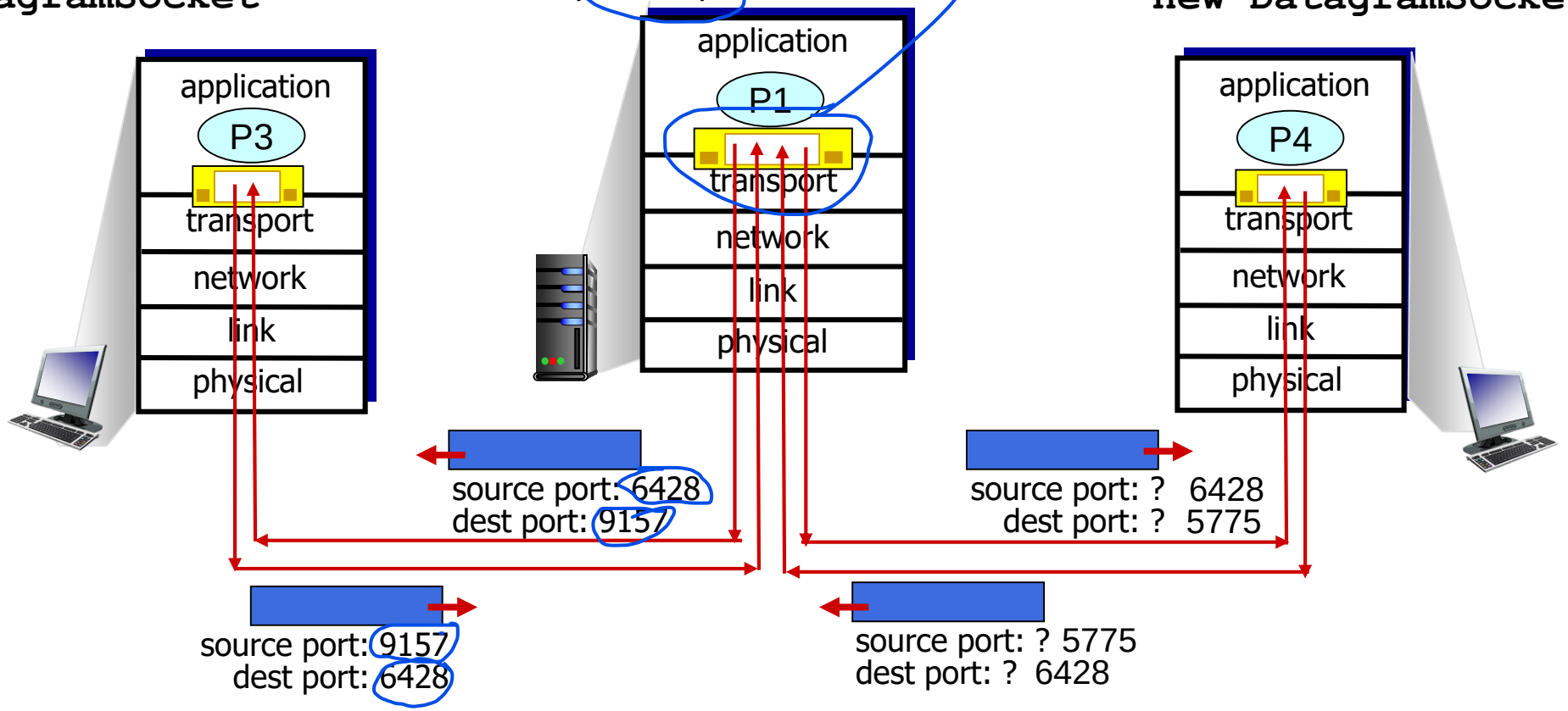
Connectionless demultiplexing: an example

```
DatagramSocket mySocket2 =
 new DatagramSocket
 (9157);
```

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

```
DatagramSocket mySocket1 =
 new DatagramSocket (5775);
```

Only 1 socket at server



Connection-oriented demultiplexing TCP

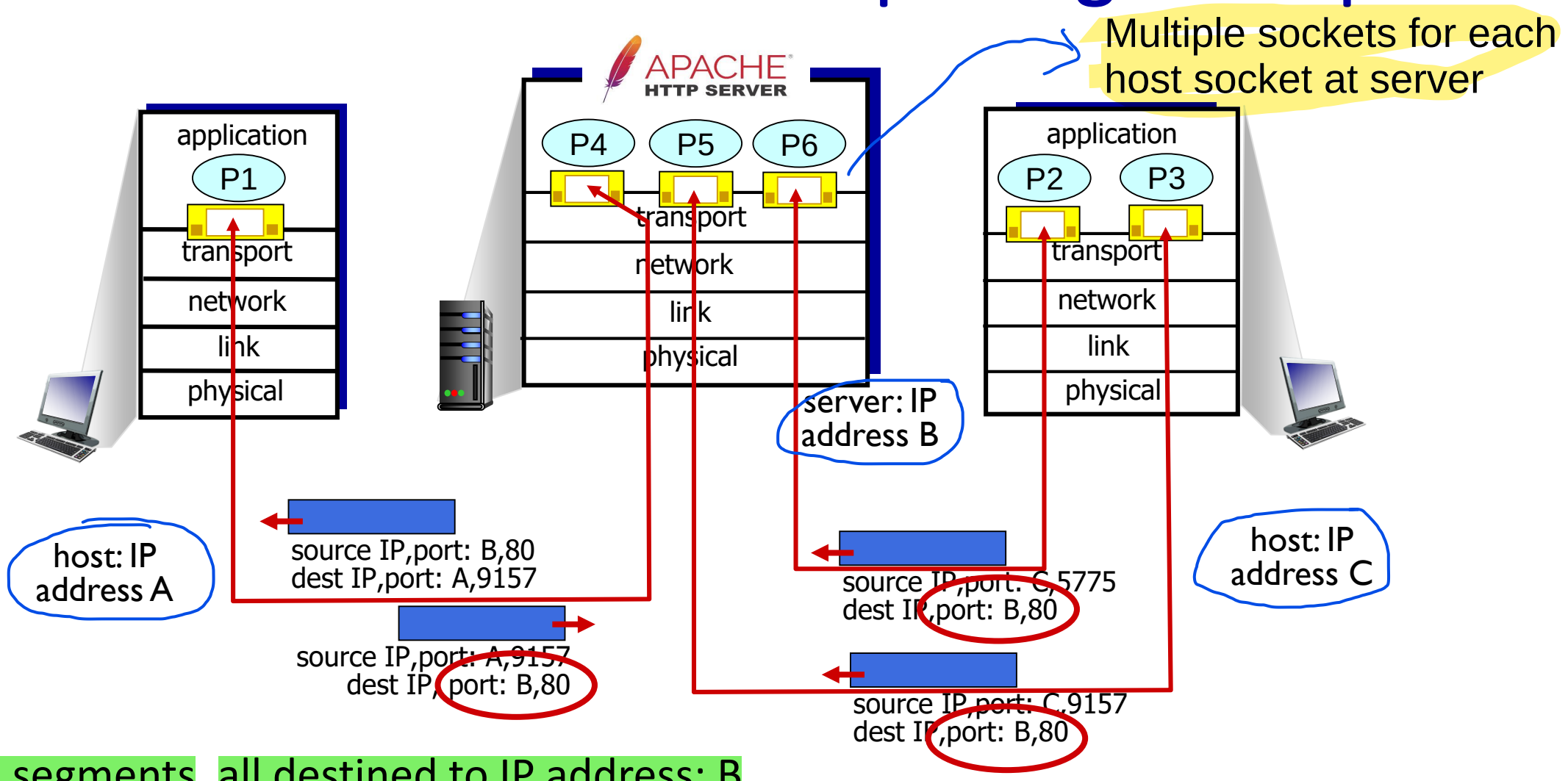
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

4- Tuple

(Src IP, Src Port #, Dest Ip, Dest Port #)

Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

Chapter 3: roadmap

- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer (Go-back-N and Selective Repeat)
- Connection-oriented transport: TCP
- TCP congestion control

UDP: User Datagram Protocol

Faster, transaction-oriented, non reliable, un-secure connection-less, stateless

(Not additional... Apt)

- "no frills," "bare bones" Internet transport protocol
- "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

Why is there a UDP?

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

Application over UDP rather than over TCP

- ***Finer application-level control over what data is sent, and when.***
- ***No connection establishment***
- ***No connection state***
- ***Small packet header overhead:*** The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of

Simple
Faster
Transaction-oriented,
Non reliable
Un-secure
Connection-less
Stateless,
Less space

Application	Application-Layer Protocol	Underlying Transport Protocol
[redacted]	SMTP	TCP
Remote terminal access	Telnet	TCP
Secure remote terminal access	SSH	TCP
Web	HTTP, HTTP/3	TCP (for HTTP), UDP (for HTTP/3)
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	DASH	TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

Figure 3.6 ♦ Popular Internet applications and their underlying transport protocols

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP — Simple network management protocol
  - HTTP/3 — Remote file server
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

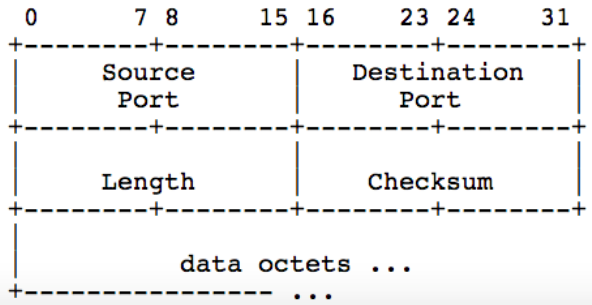
User Datagram Protocol

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

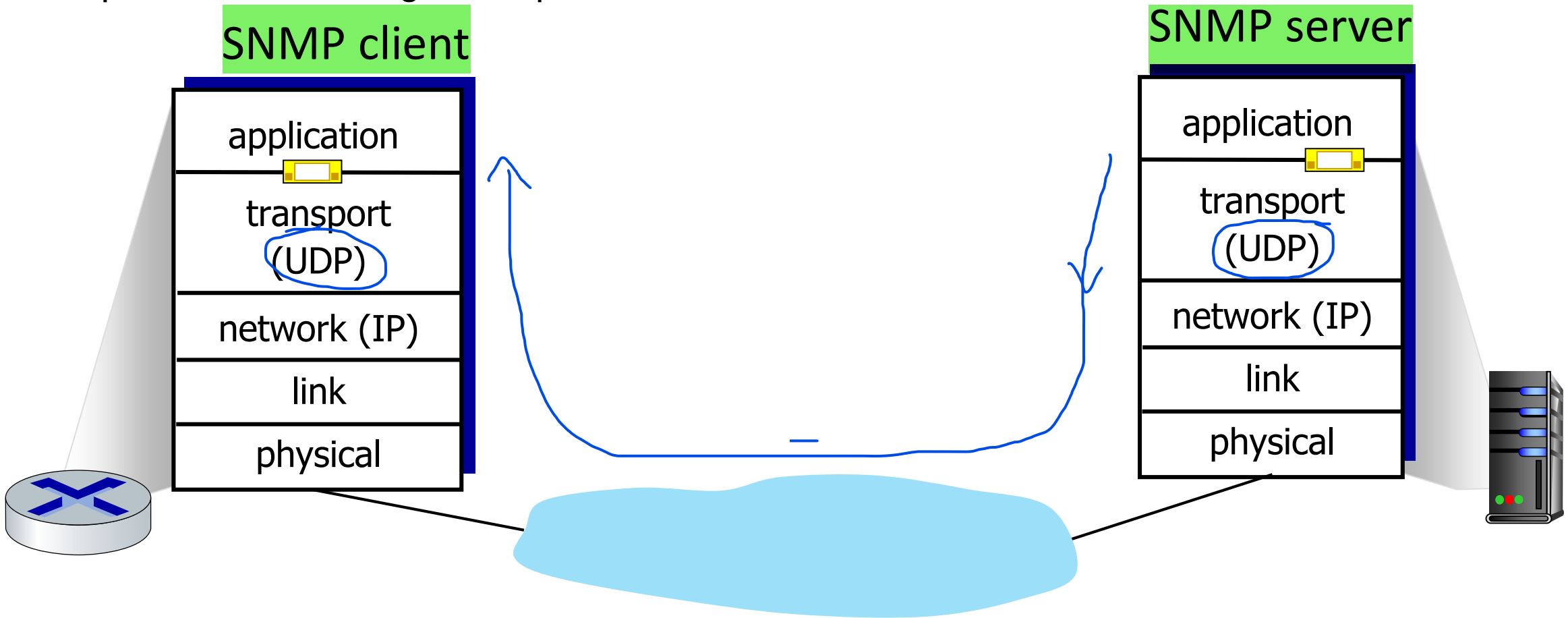
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format

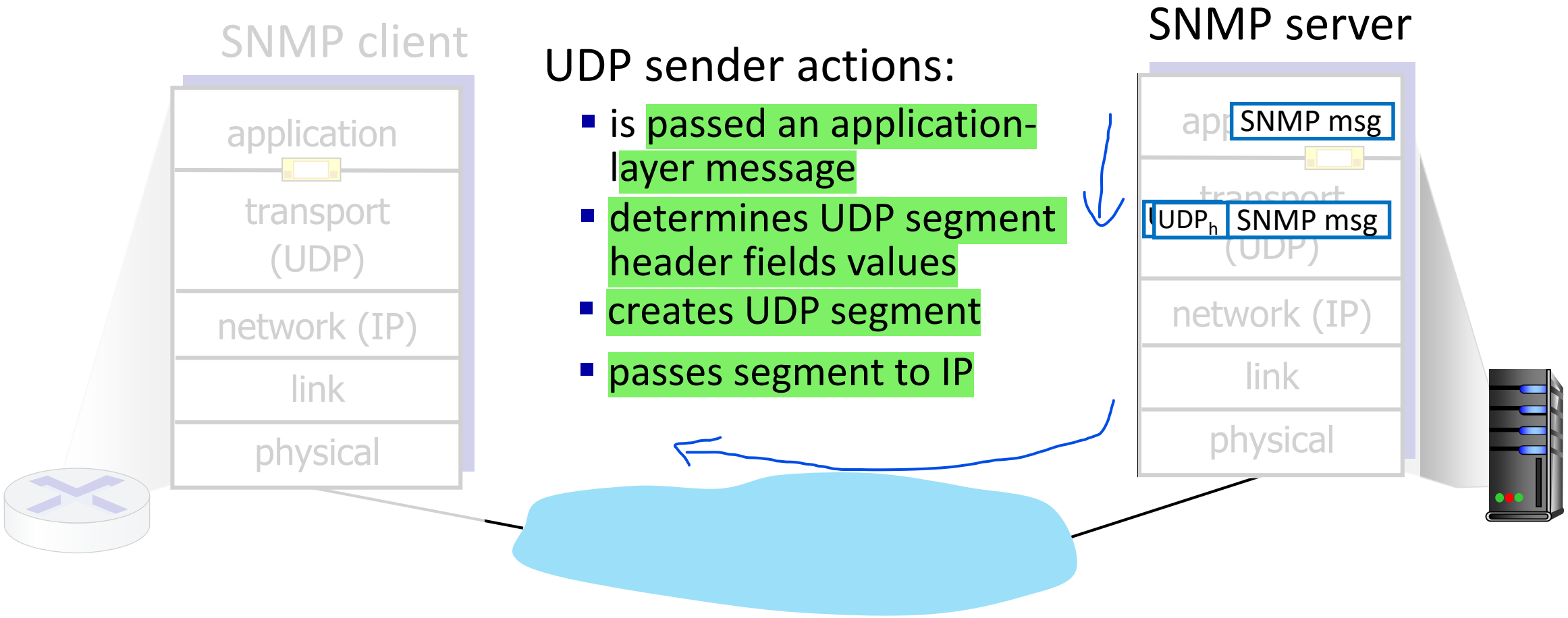


UDP: Transport Layer Actions

simple network management protocol



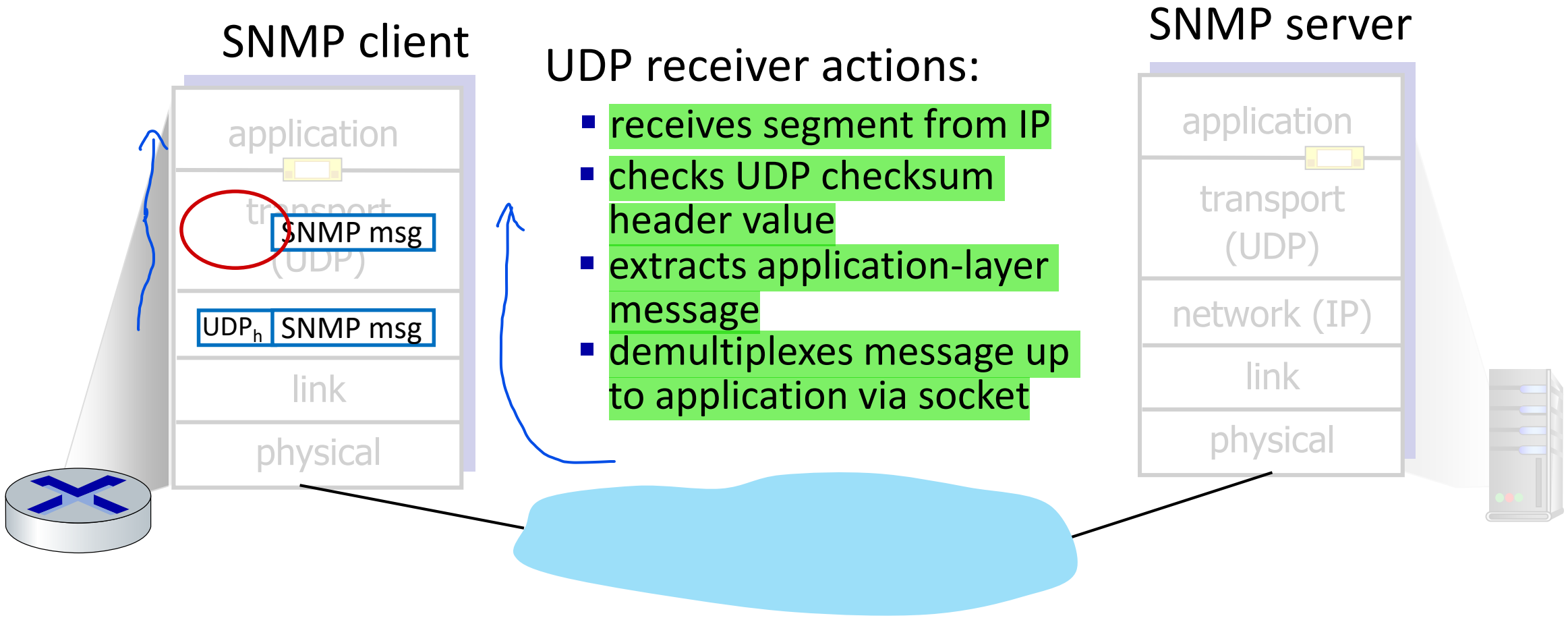
UDP: Transport Layer Actions



UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

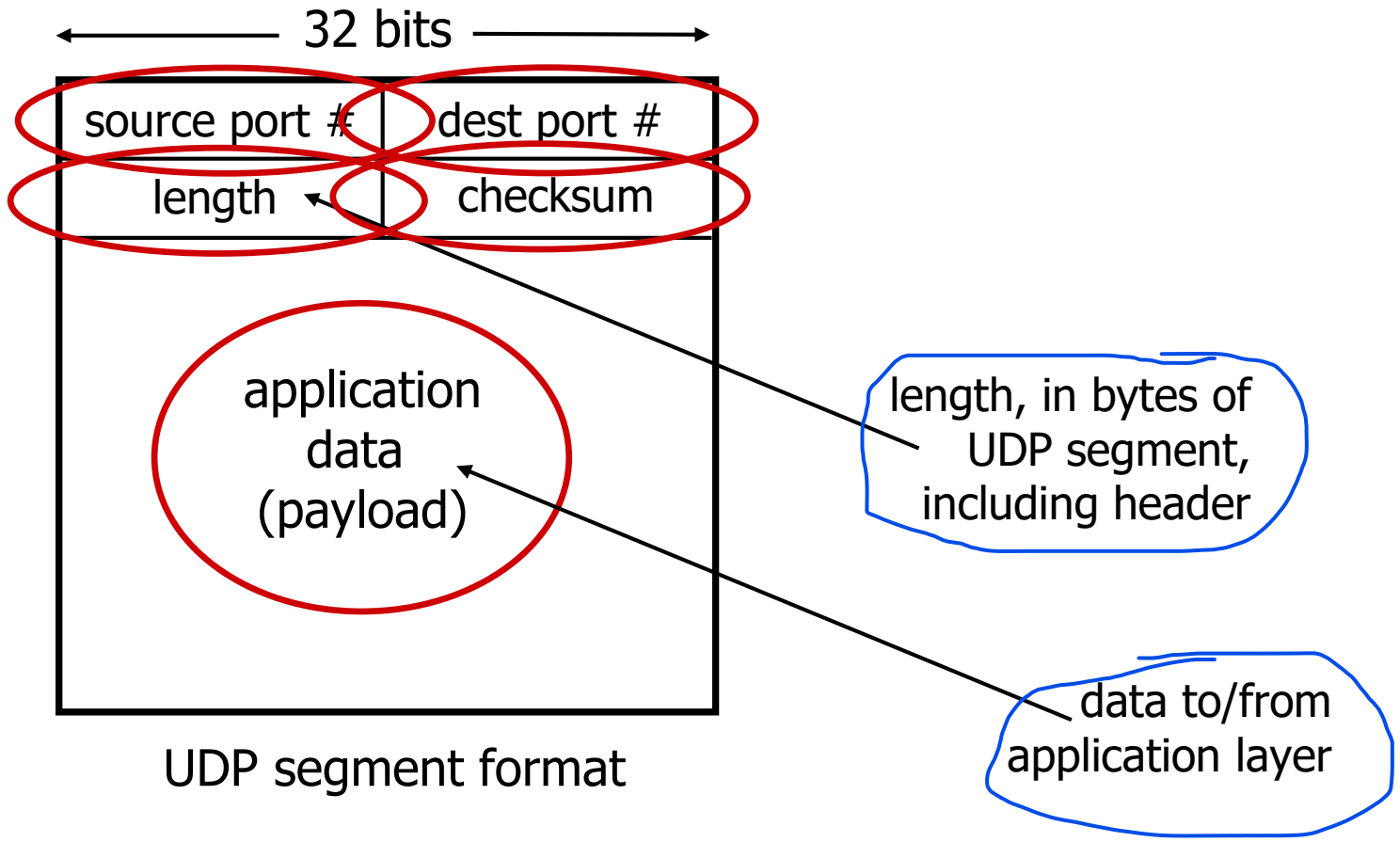
UDP: Transport Layer Actions



UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

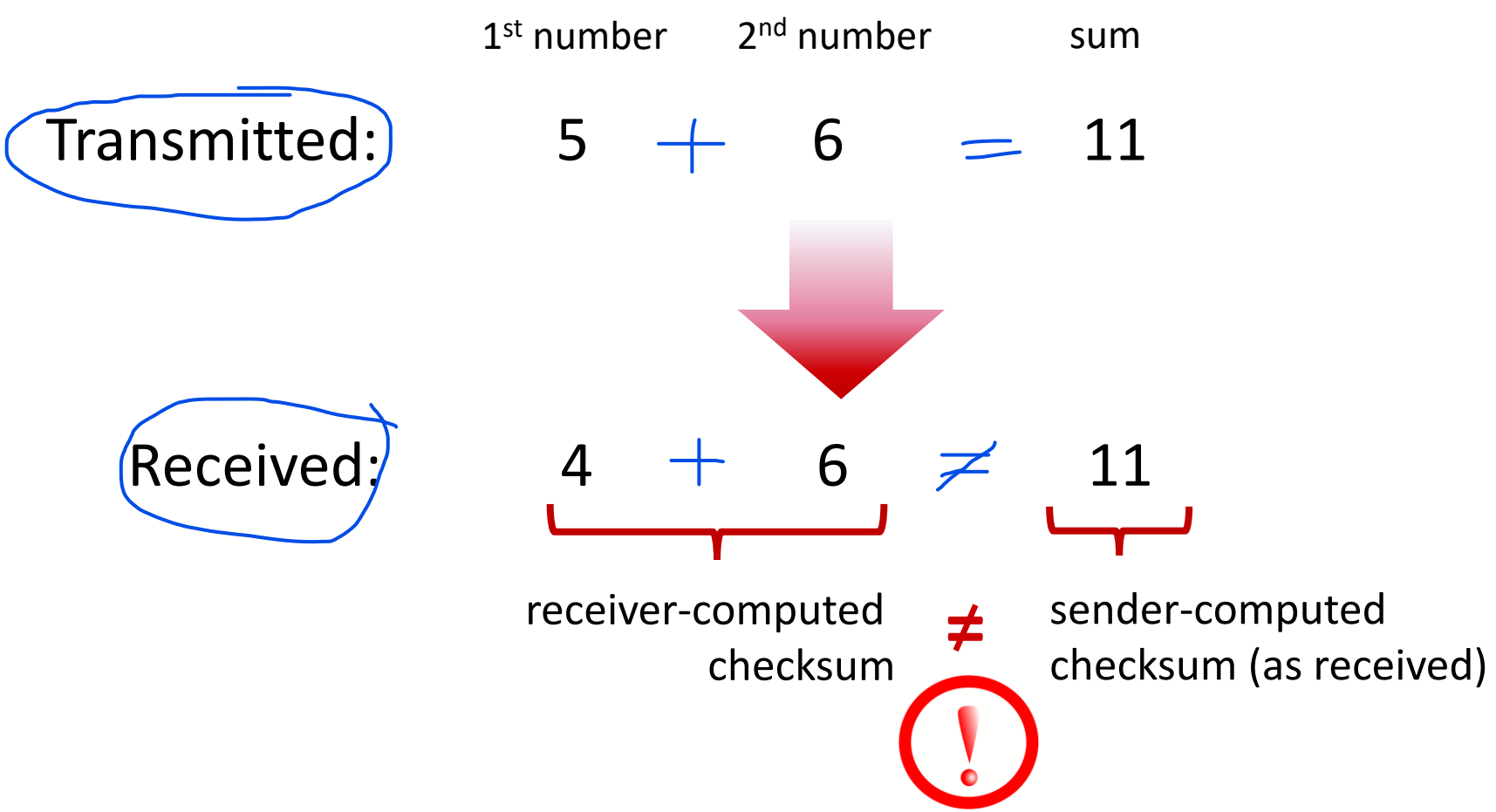
UDP segment header



UDP segment format

UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment



UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

sender:

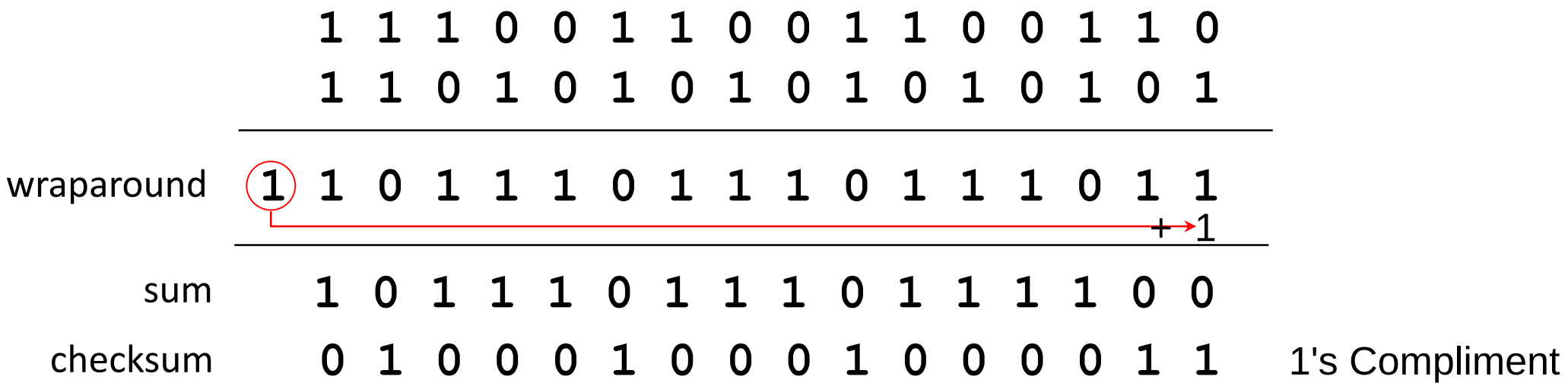
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

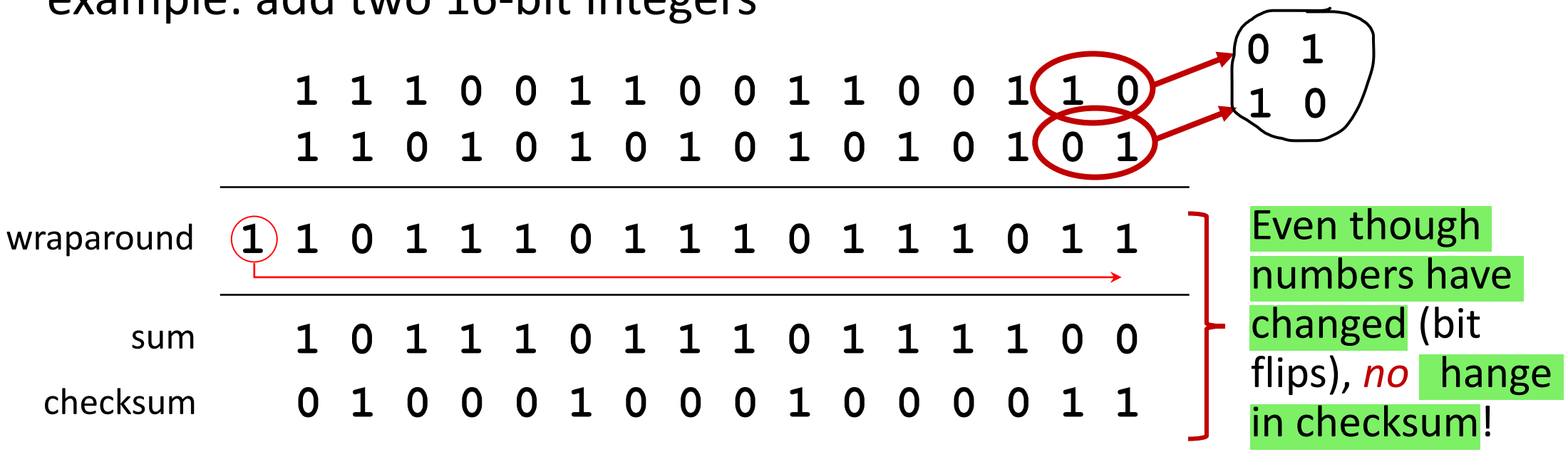


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

example: add two 16-bit integers



Even though numbers have changed (bit flips), *no* change in checksum!

Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Go-Back-N (GBN)

Sliding Window Protocol

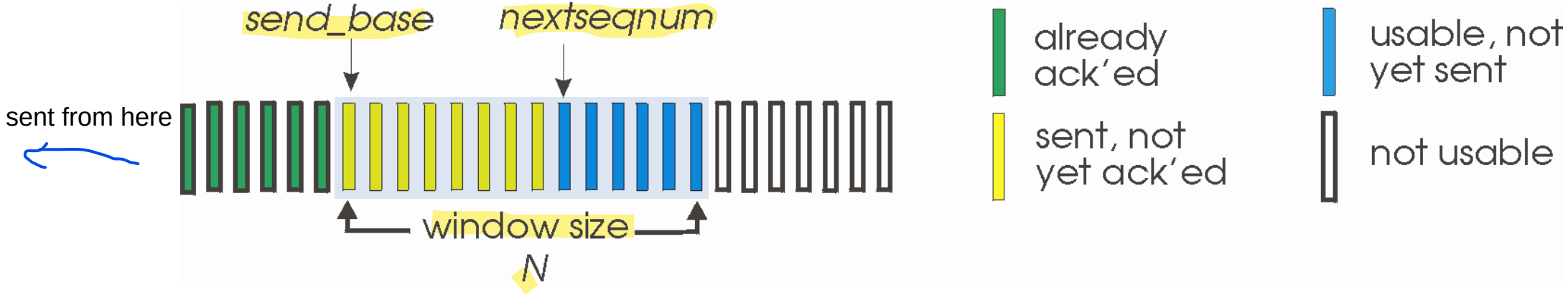
- In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N, of unacknowledged packets in the pipeline.

- https://computerscience.unicam.it/marcantoni/reti/applet/GoBackProtocol/goback.html

- https://www.youtube.com/watch?v=gaocB7unrqs

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

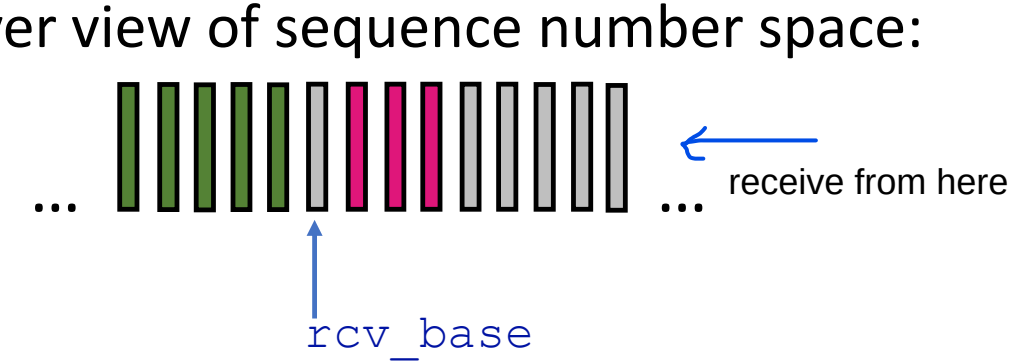


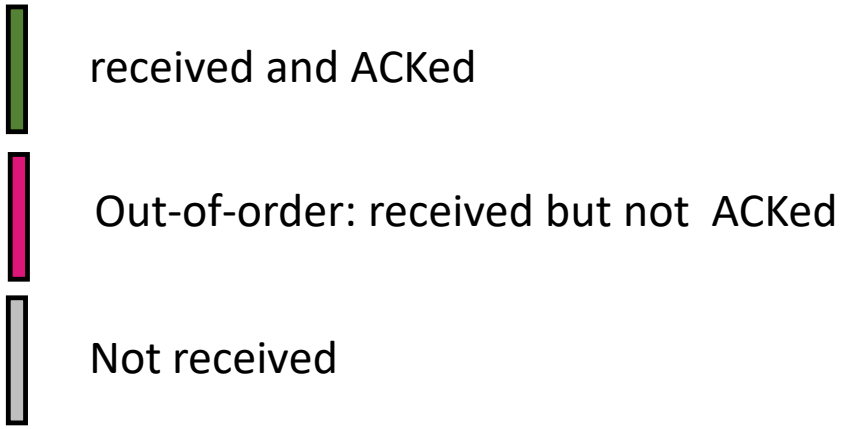
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

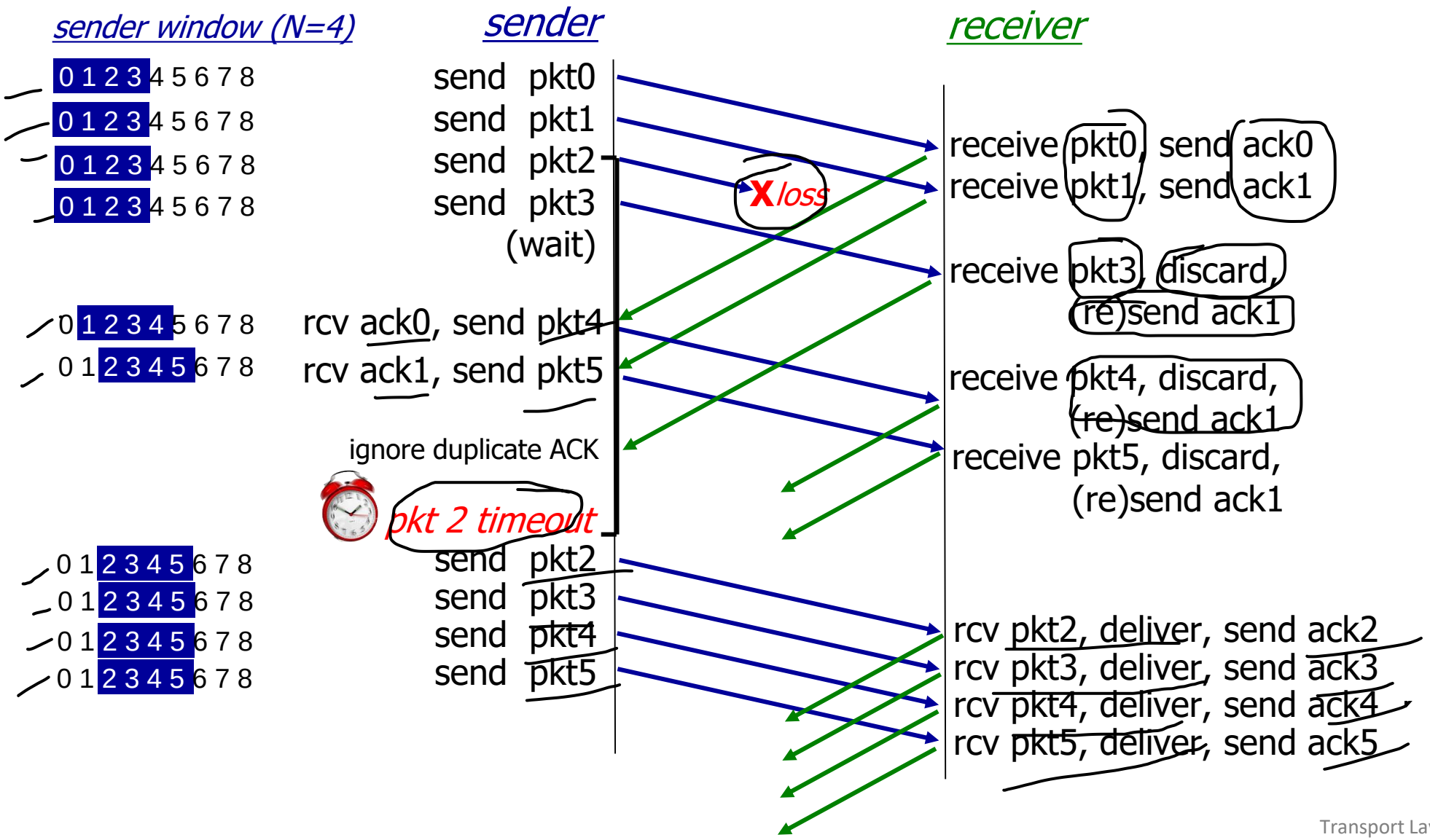
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:





Go-Back-N in action



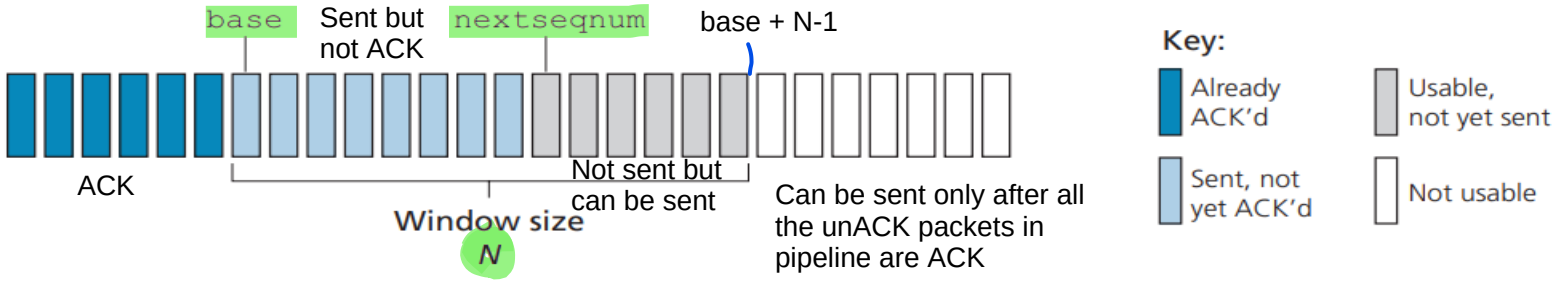
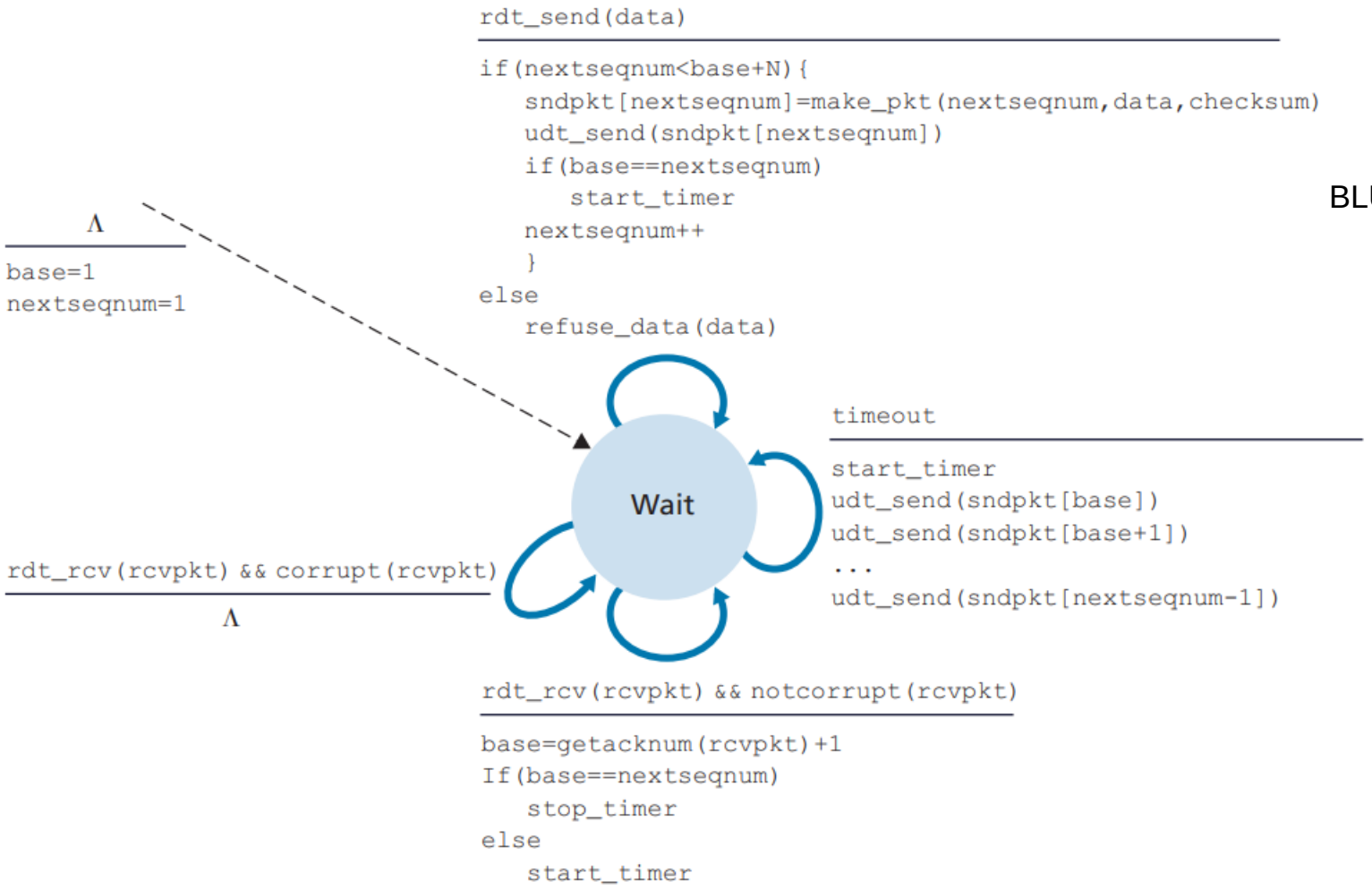


Figure 3.19 ♦ Sender's view of sequence numbers in Go-Back-N

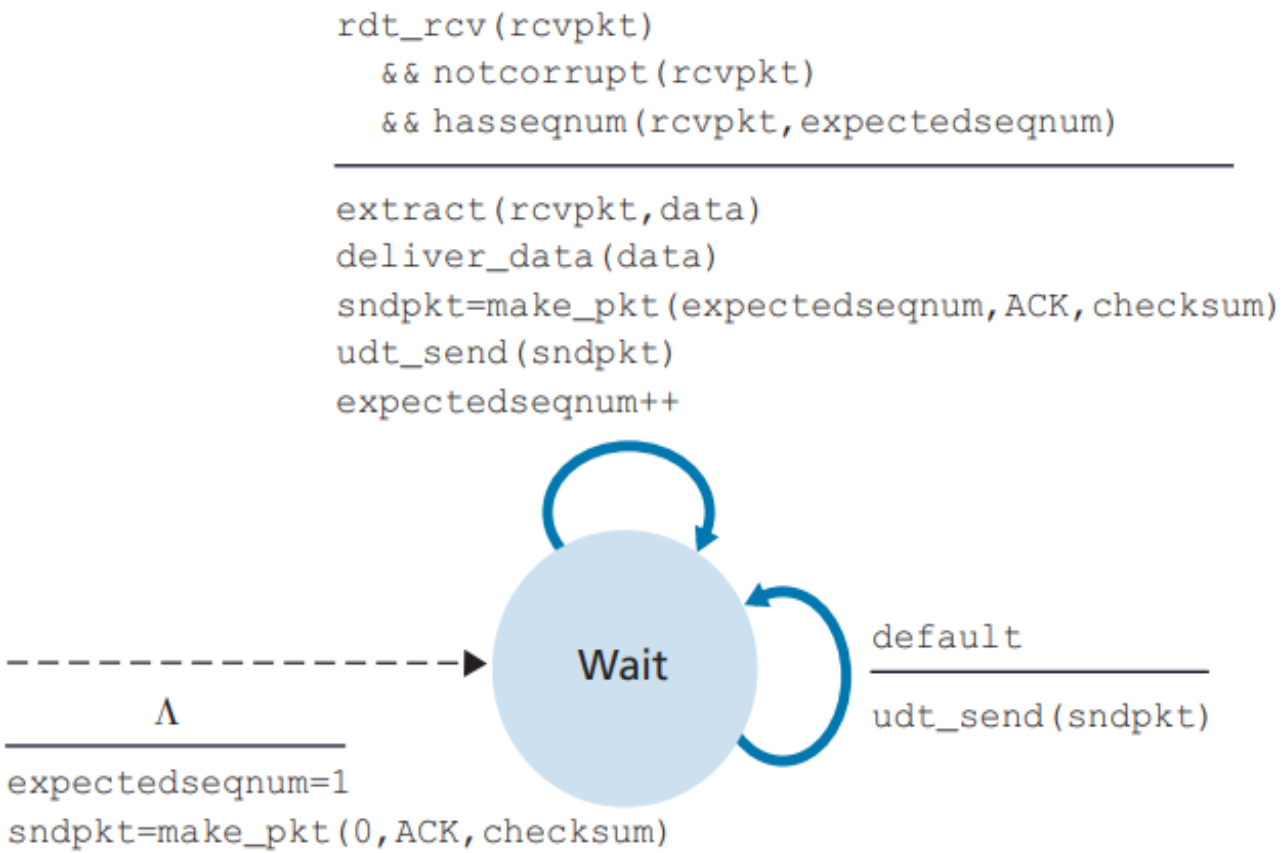
- **Base-** to be the sequence number of the oldest unacknowledged packet
- **nextseqnum** - to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent).
- Sequence numbers in the interval **[0,base-1]** correspond to packets that have already been transmitted and acknowledged.
- The interval **[base,nextseqnum-1]** corresponds to packets that have been sent but not yet acknowledged.
- Sequence numbers in the interval **[nextseqnum,base+N-1]** can be used for packets that can be sent immediately, should data arrive from the upper layer.
- sequence numbers **greater than or equal to base+N** cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged

FSM for sender



BLUFF AND MANIPULATE YOUR ANSWER

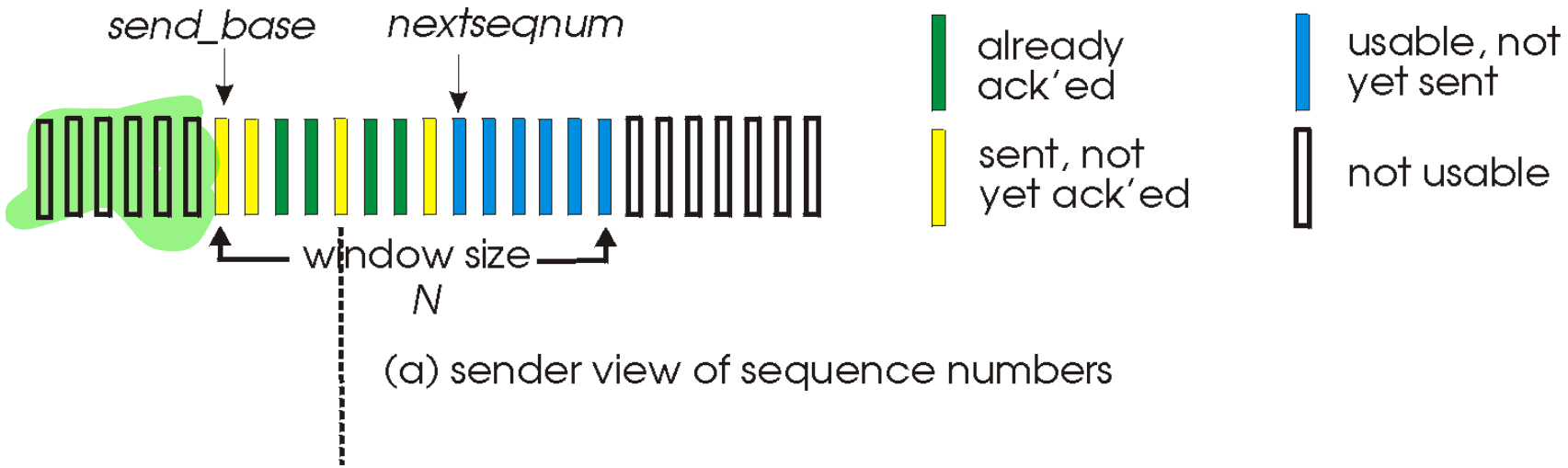
FSM for receiver



Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

packet *n* in [rcvbase, rcvbase+N-1]

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

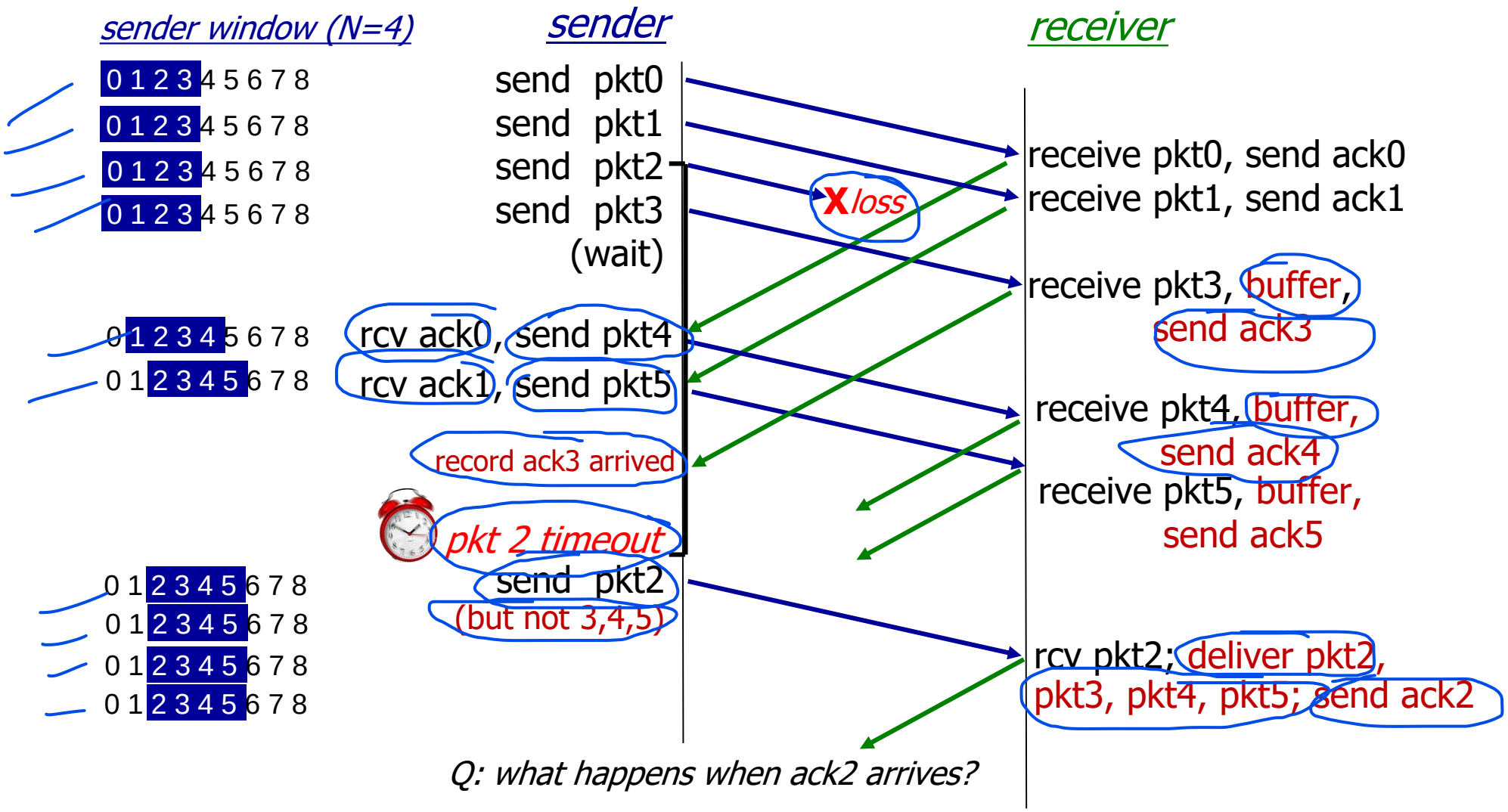
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

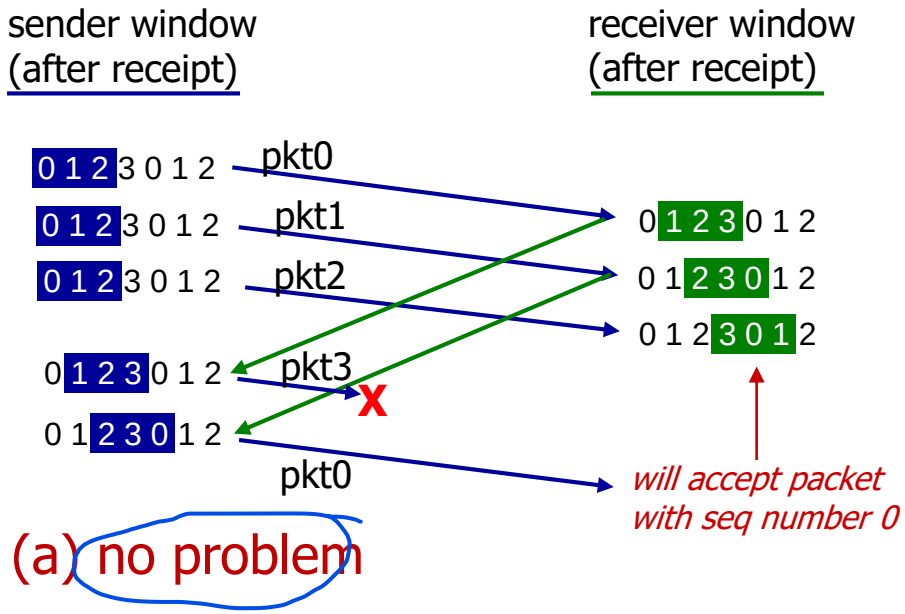
Selective Repeat in action

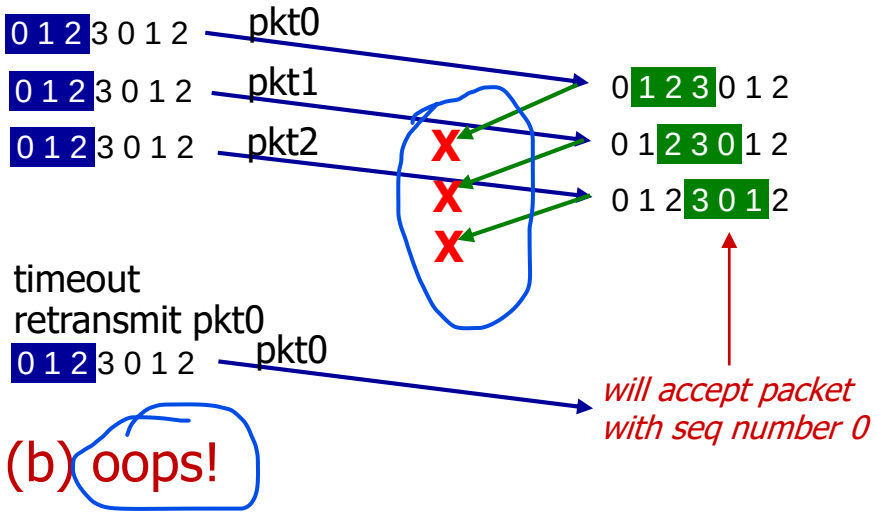


Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



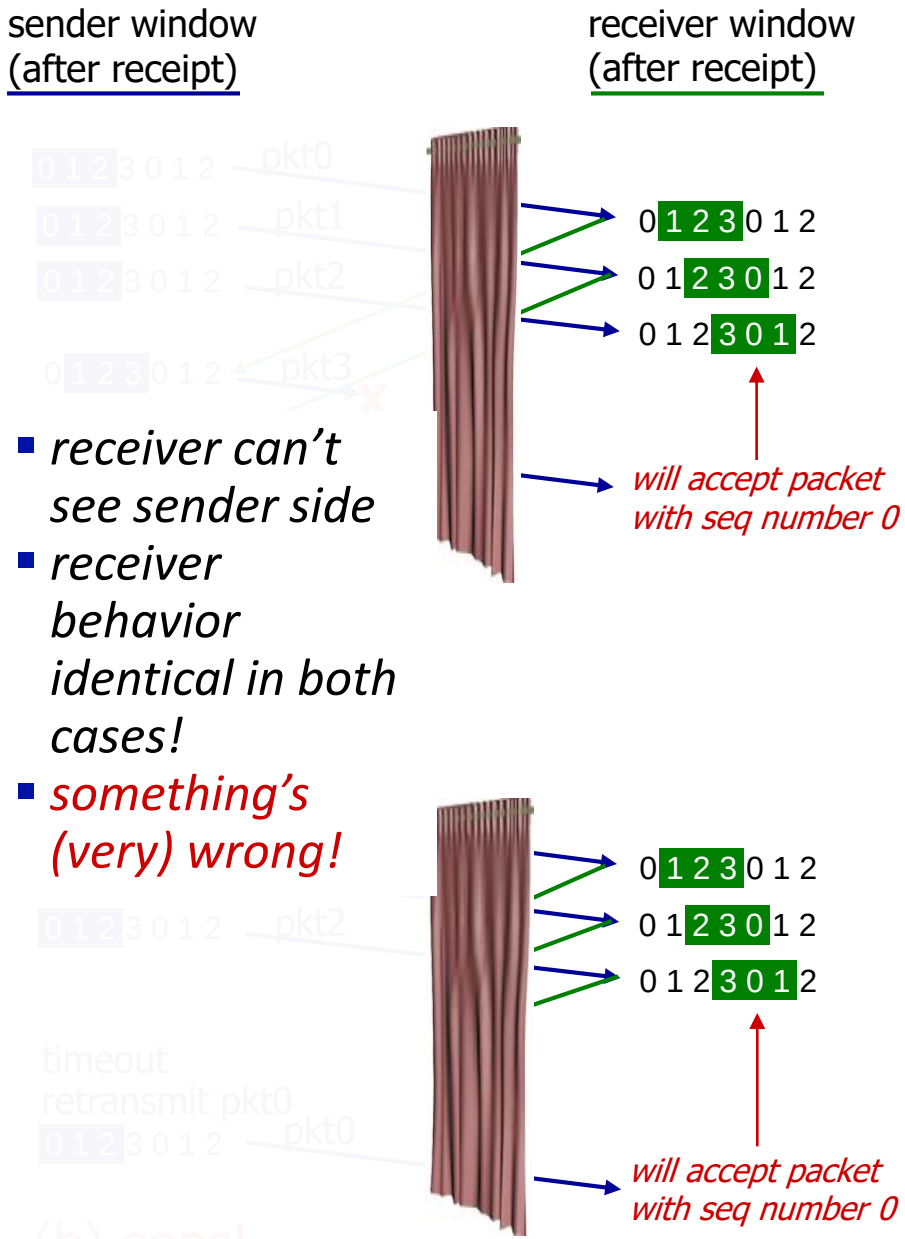


Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?



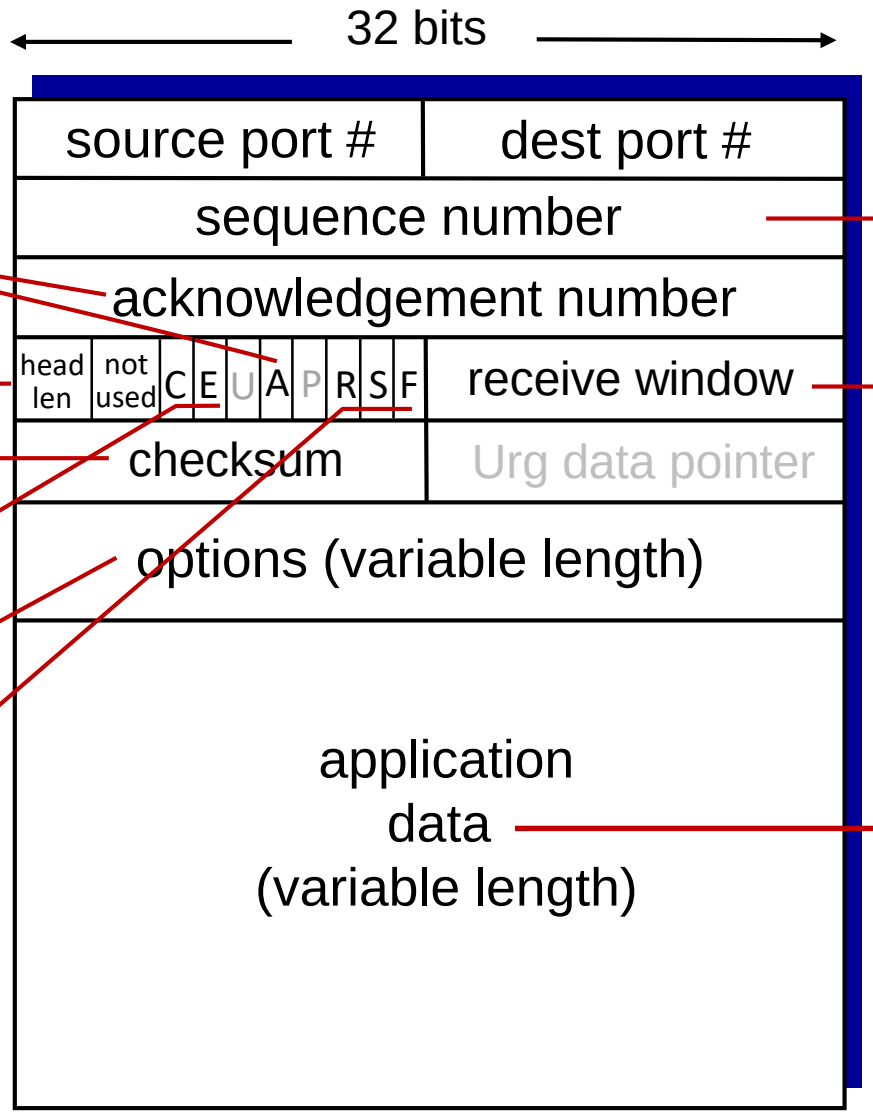
Chapter 3: roadmap

- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer (Go-back-N and Selective Repeat)
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet **checksum**

C, E: congestion notification

TCP **options**

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP Segment

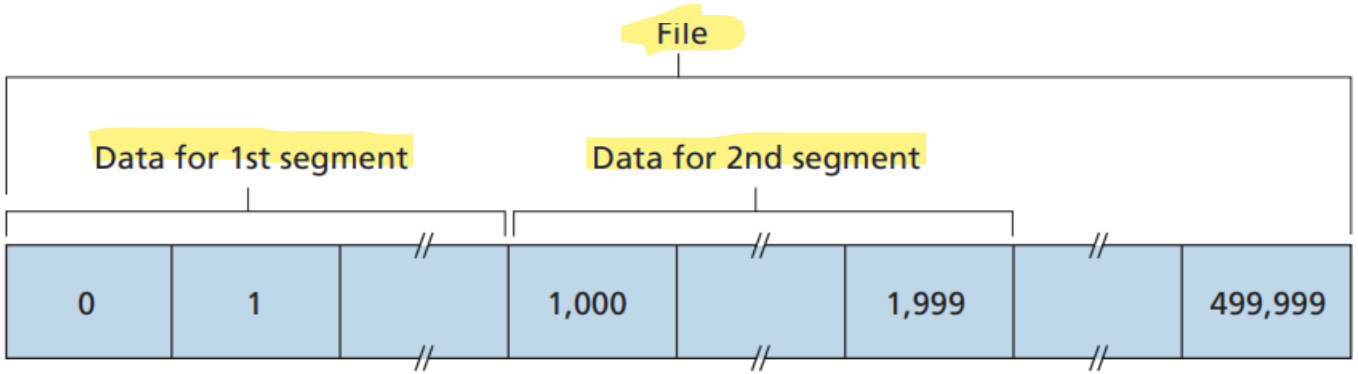
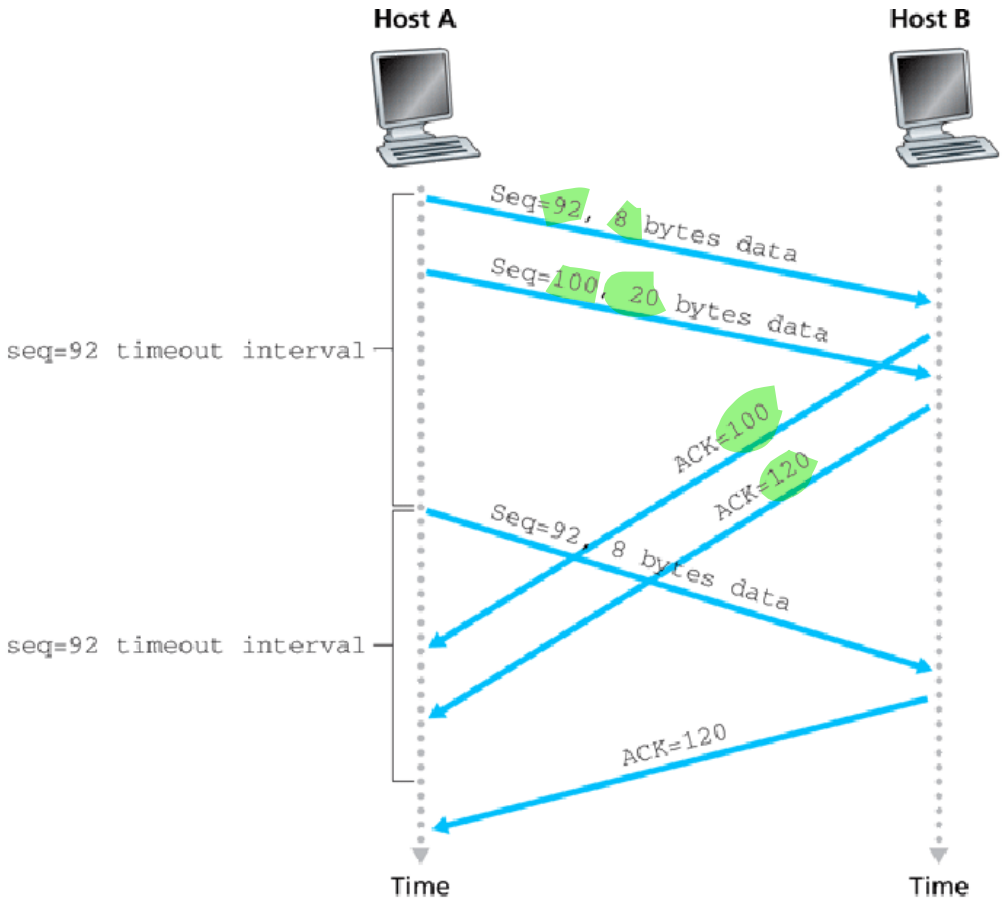


Figure 3.30 ♦ Dividing file data into TCP segments

Cumulative Acknowledgement



TCP is said to provide cumulative acknowledgments

Next Seq # = ACK # =
Sent Seq # + No of Bytes sent

TCP sequence numbers, ACKs

Sequence numbers:

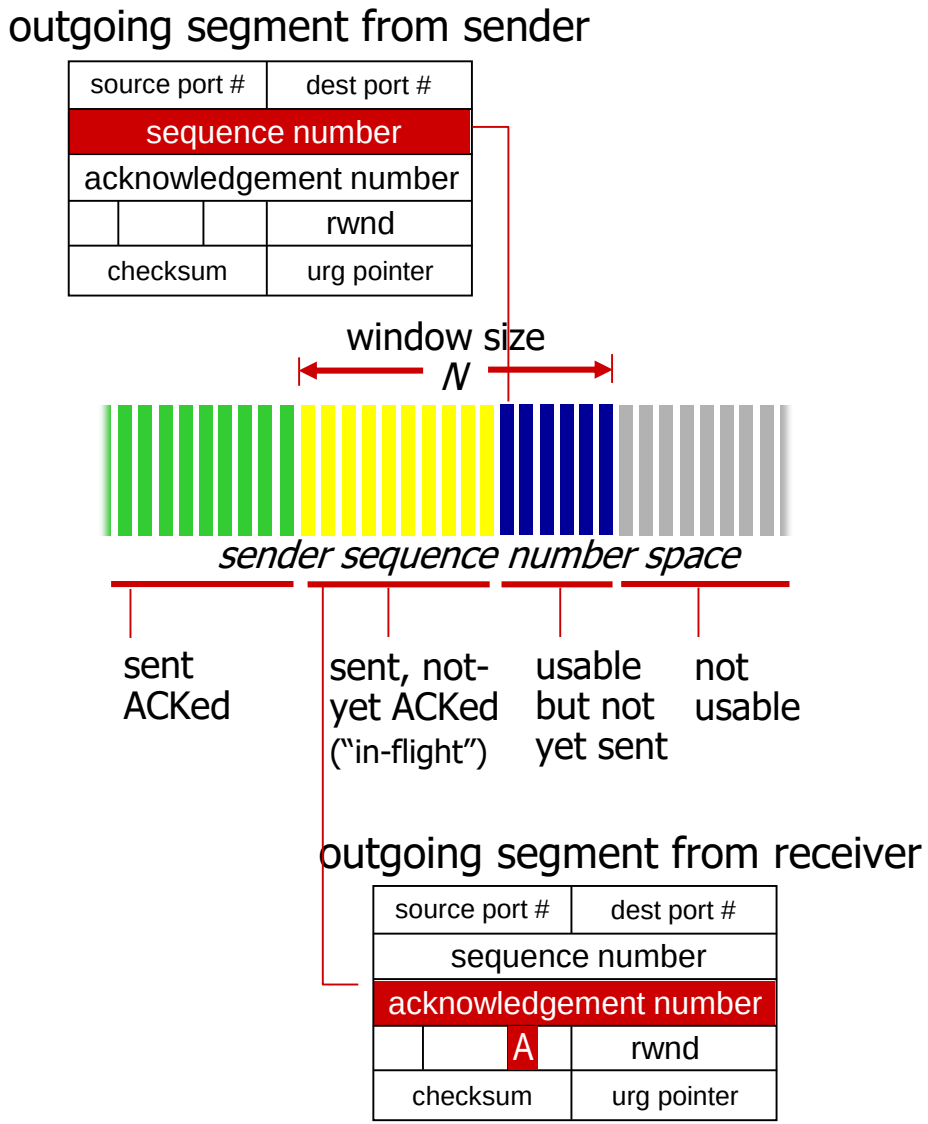
- byte stream "number" of first byte in segment's data

Acknowledgements:

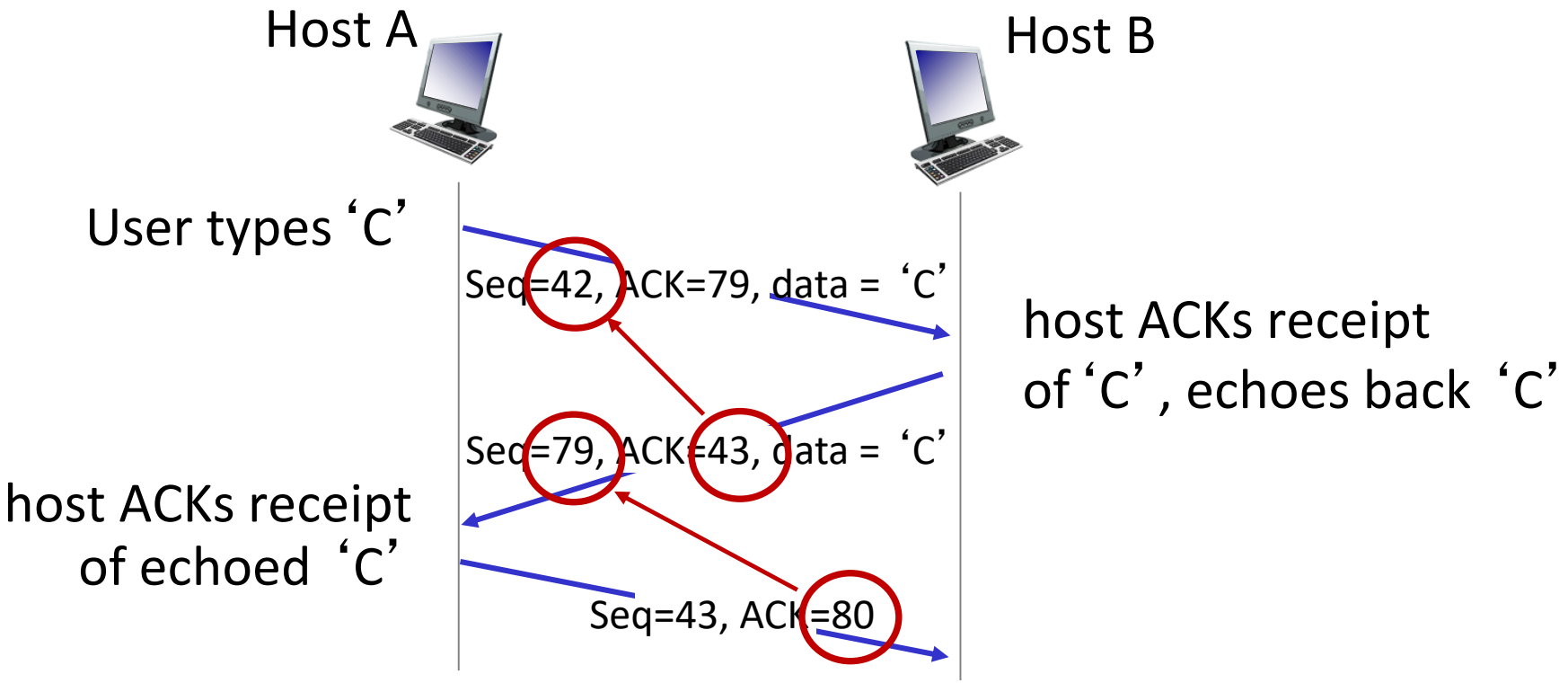
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor



TCP sequence numbers, ACKs



simple telnet scenario

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

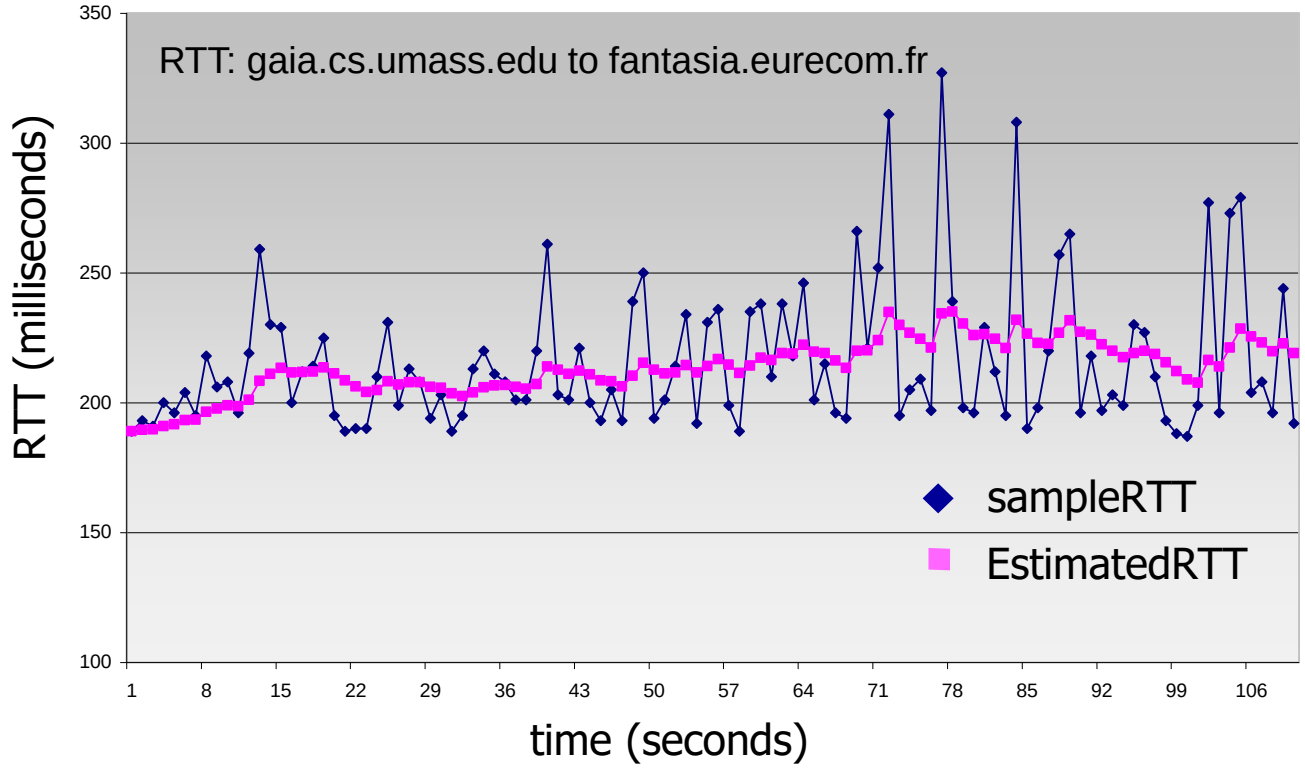
Q: how to estimate RTT?

- `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 (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT**: want a larger safety margin

TimeoutInterval = EstimatedRTT + 4*DevRTT

estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Problems

1. Suppose the measured RTT values are 106 ms and 120 ms.
   a. Compute estimated RTT after each of these sample RTT values is obtained using the value of α = 0.125 and assuming that the value of estimated RTT was 100 ms just before the first of these two samples was obtained.

- Compute also DevRTT after each sample is obtained assuming the value of β=0.25 and assuming the value of DevRTT = 5 ms just before the first of these two samples was obtained.

Calculate the EstimatedRTT after obtaining the first sample RTT=106ms,

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

EstimatedRTT =0.125 * 106 + (1-0.125) * 100

=0.125* 106 + 0.875 * 100

=13.25 + 87.5

=100.75ms

Calculate the DevRTT after obtaining the first sample RTT:

DevRTT = β * | SampleRTT- EstimatedRTT|+(1- β)* DevRTT

=0.25 * |106-100.75| + (1-0.25) *5

=0.25 *5.25 + 0.75 * 5

=1.3125 + 3.75

=5.0625ms

TCP Sender (simplified)

event: data received from application

- 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`**

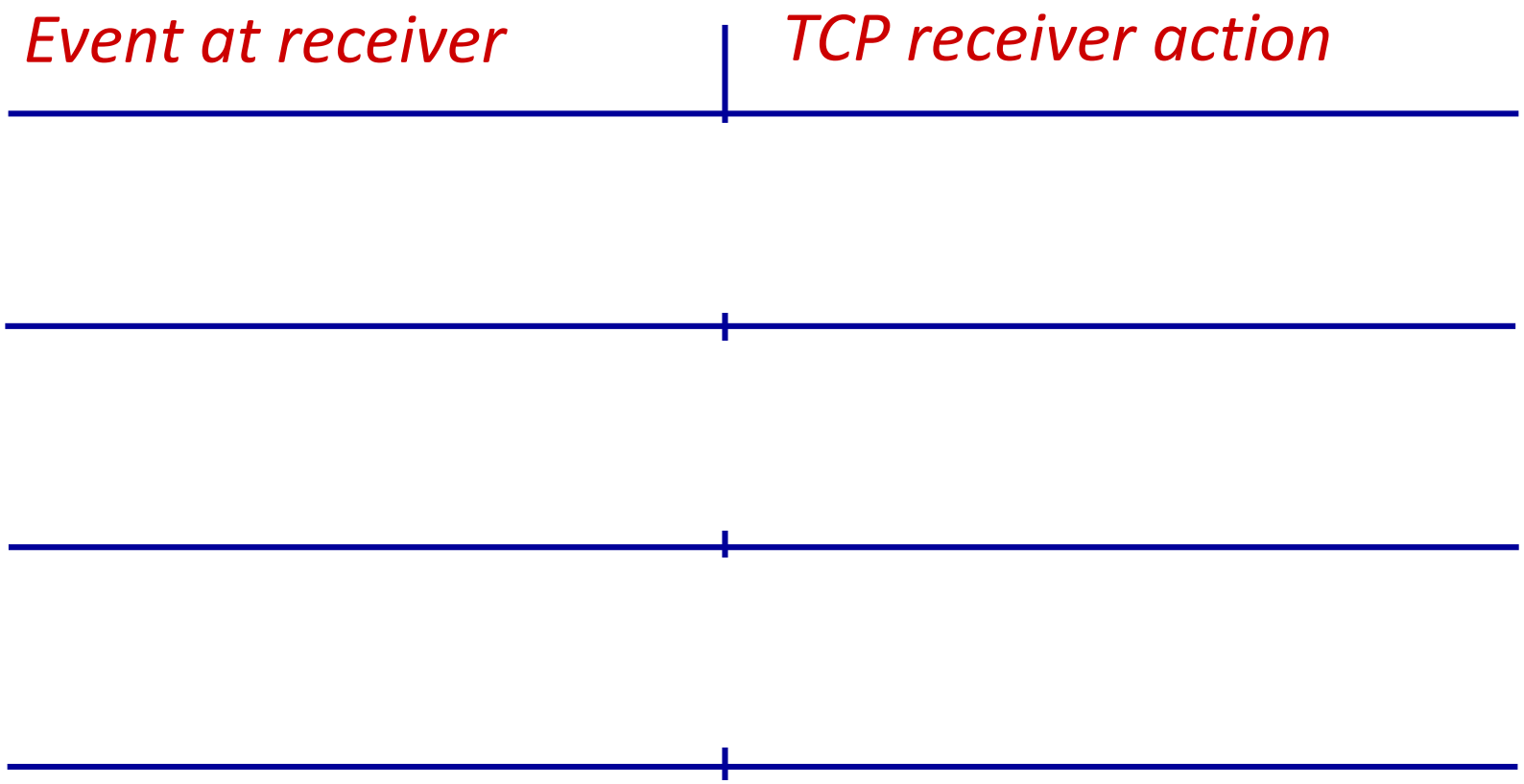
event: timeout

- retransmit segment that caused timeout
- restart timer

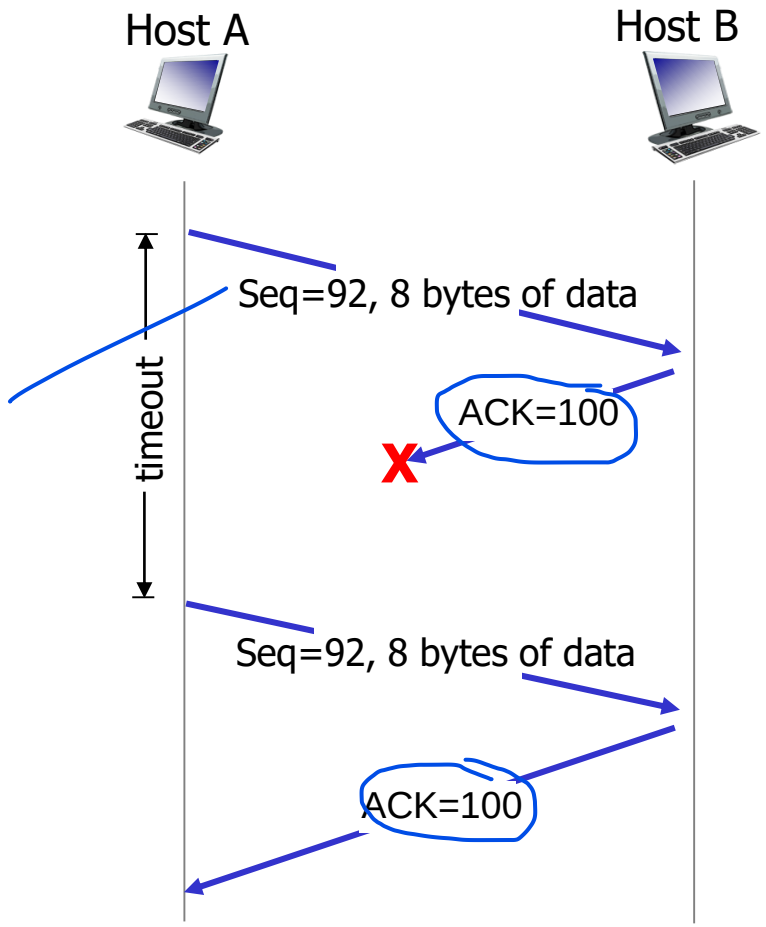
event: ACK received

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

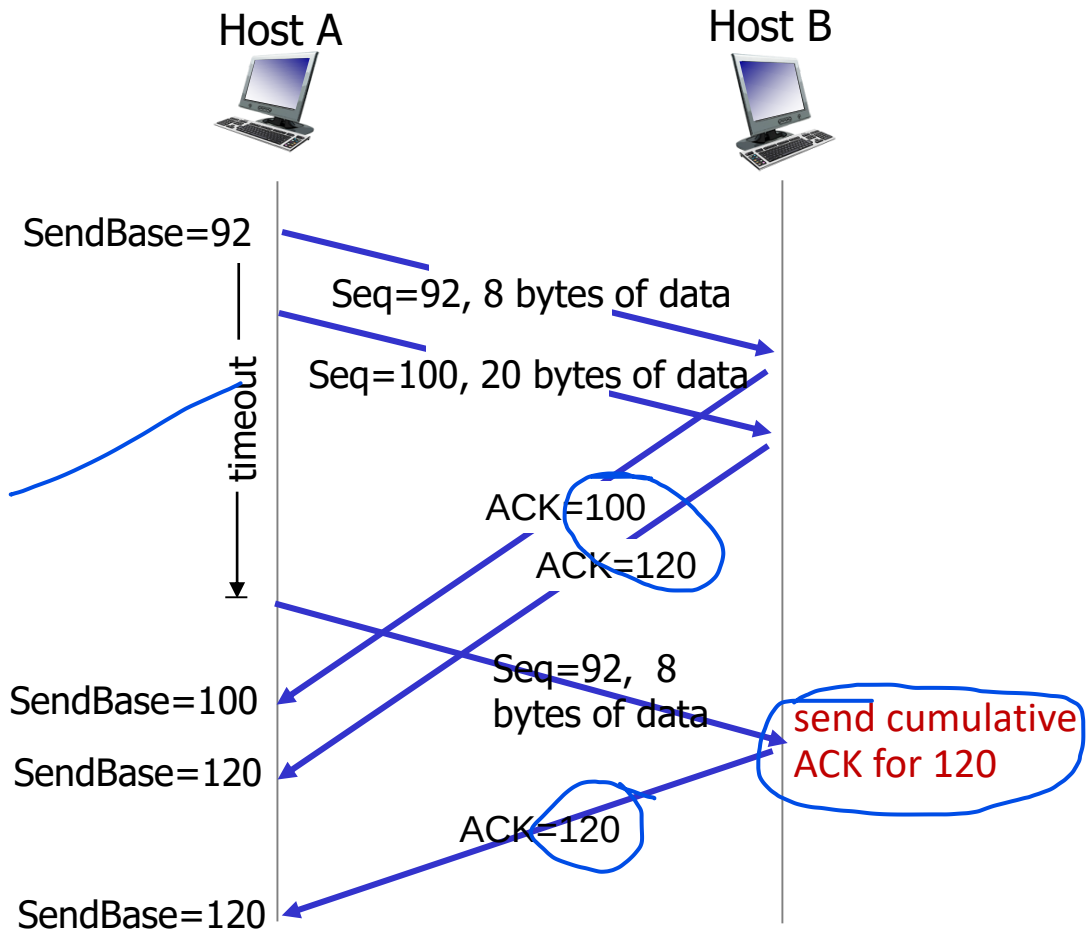
TCP Receiver: ACK generation [RFC 5681]



TCP: retransmission scenarios

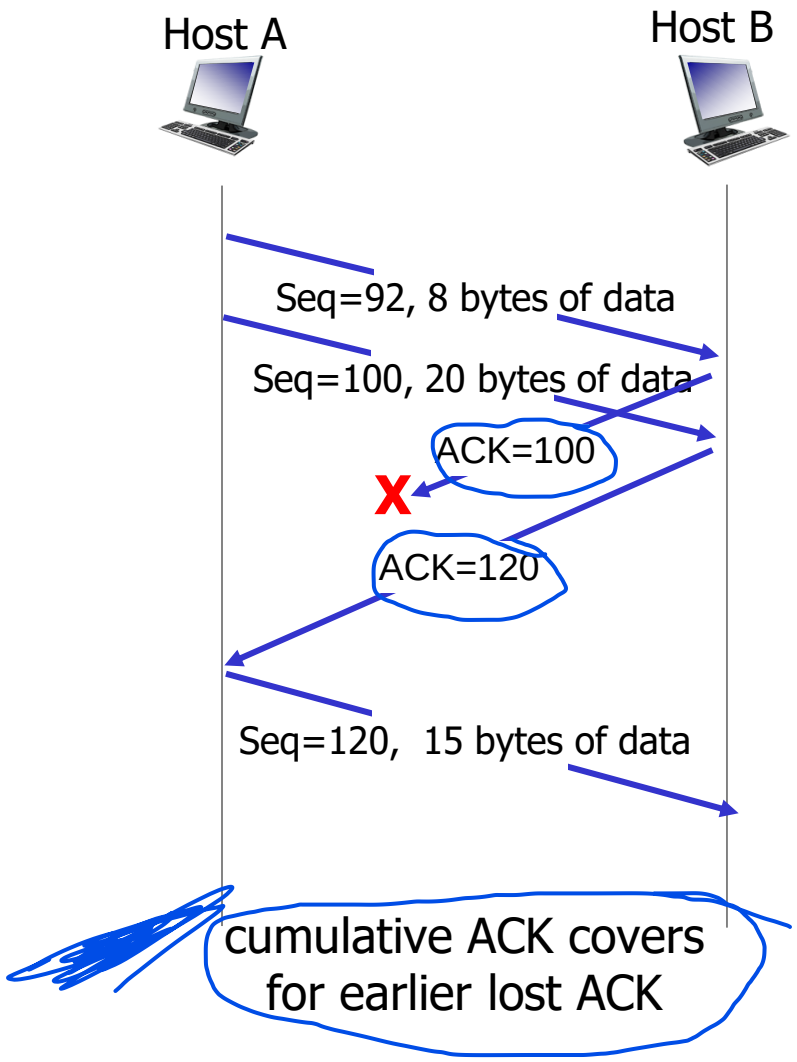


lost ACK scenario



premature timeout

TCP: retransmission scenarios



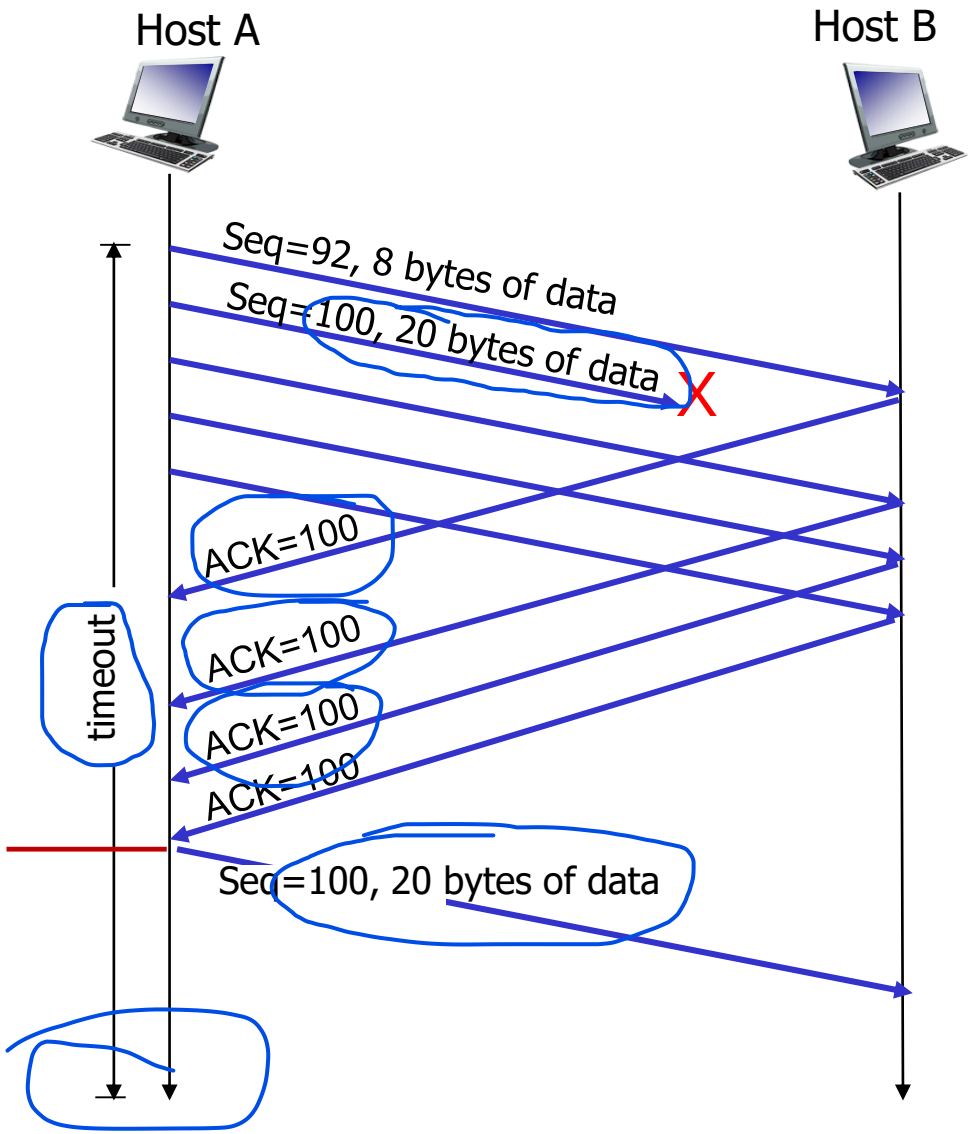
TCP fast retransmit

TCP fast retransmit

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

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

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



Chapter 3: roadmap

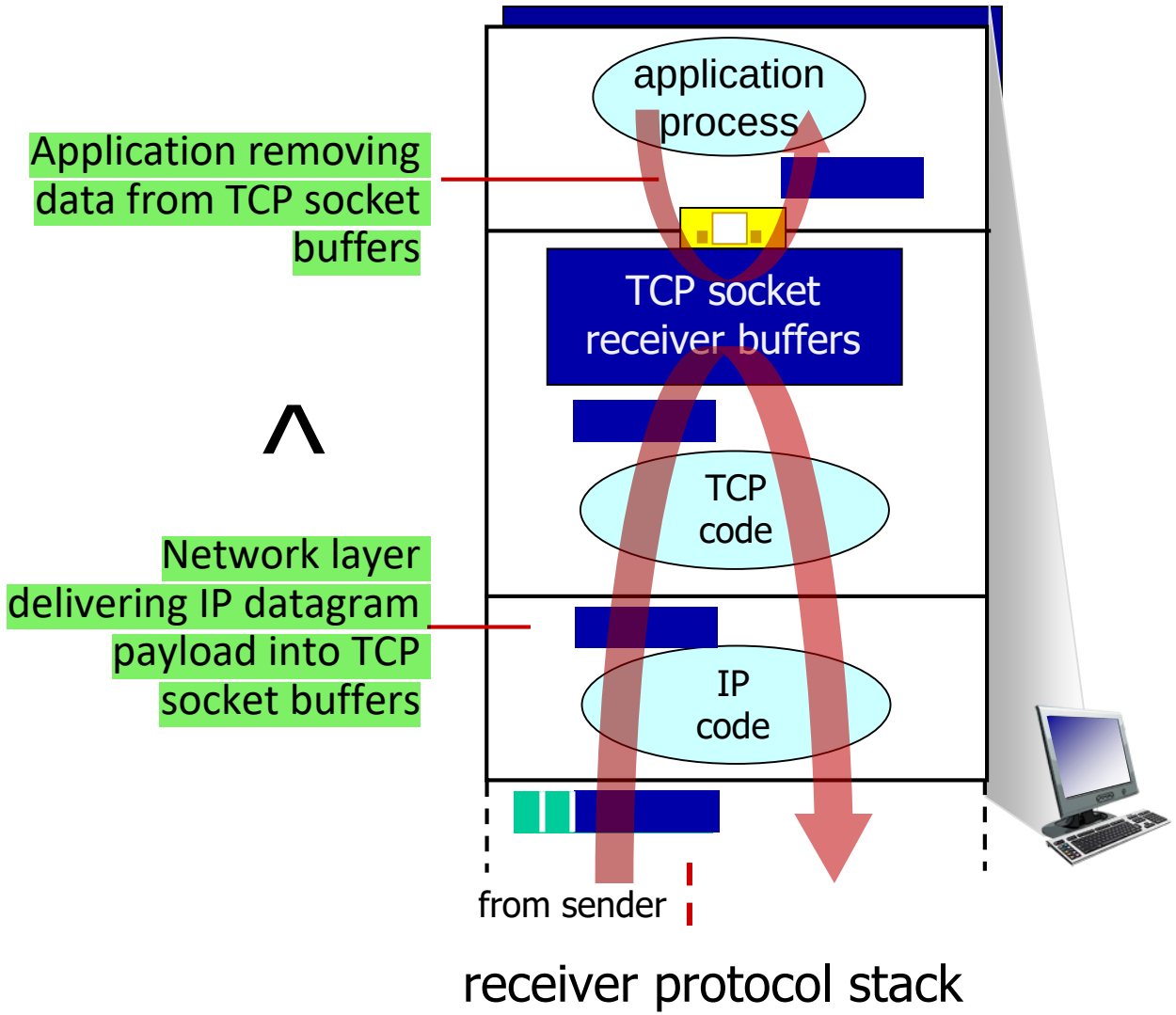
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- TCP congestion control

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

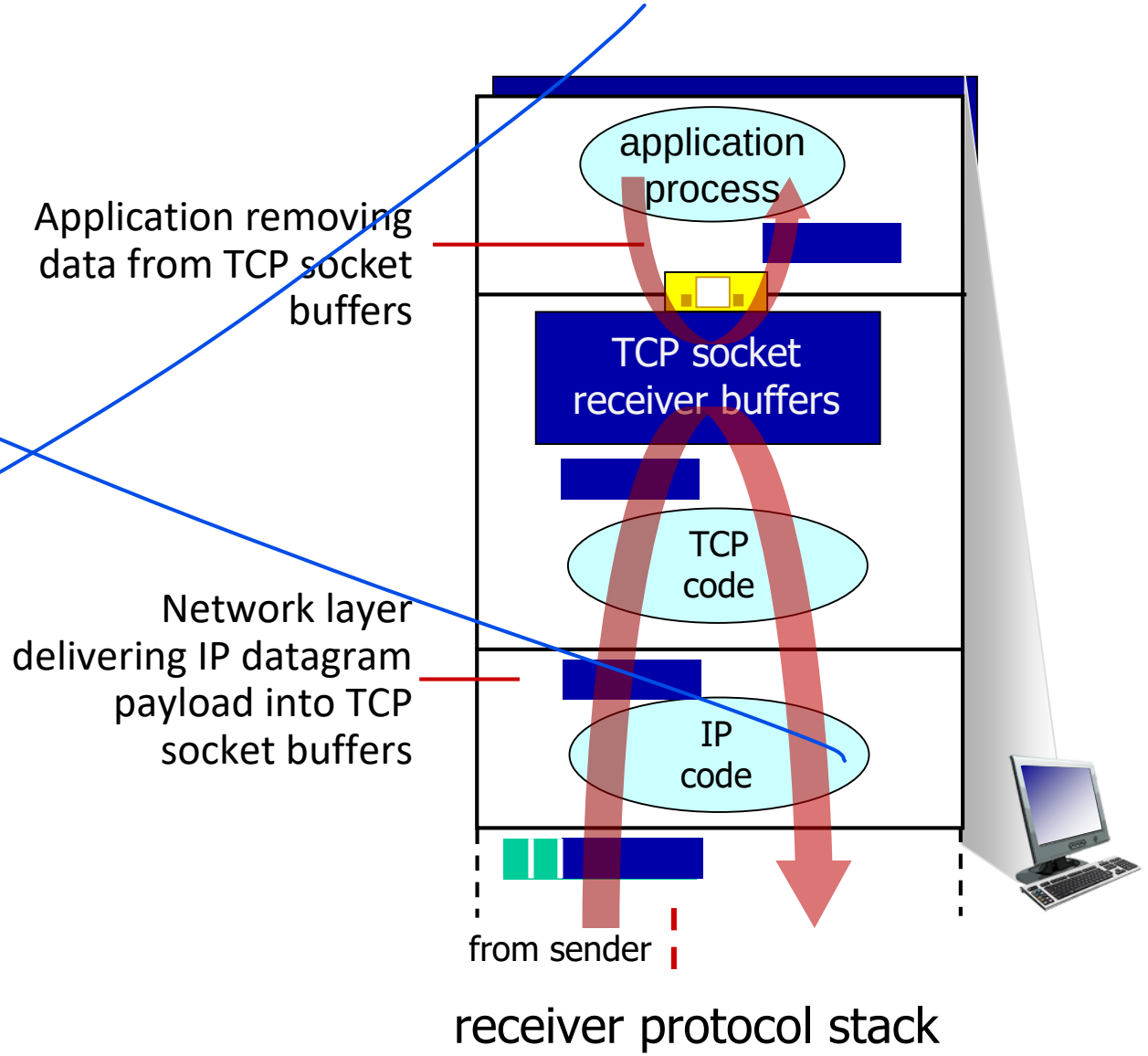
Ans: In this case the same unremoved data is sent again through Network Layer

But Flow Control Mechanism prevents it



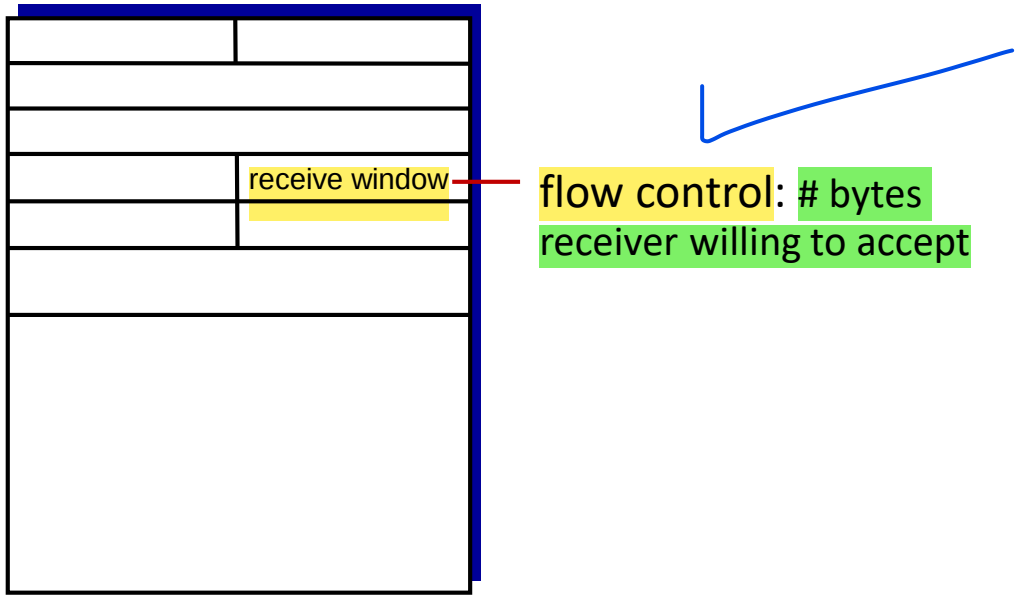
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



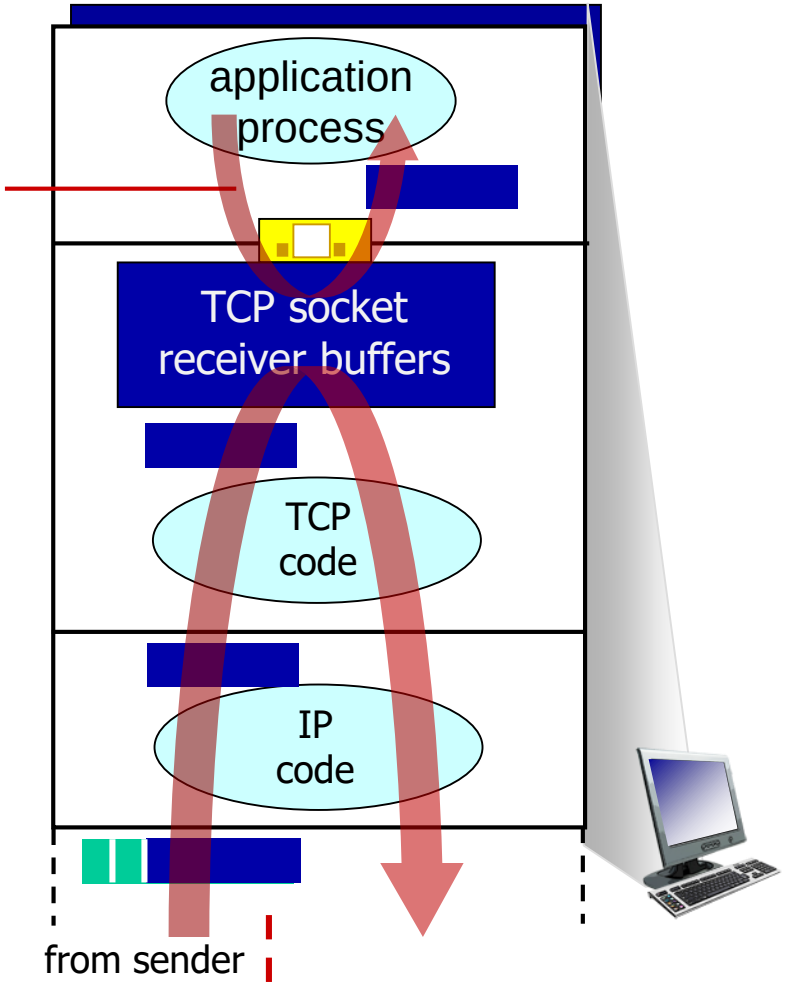
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



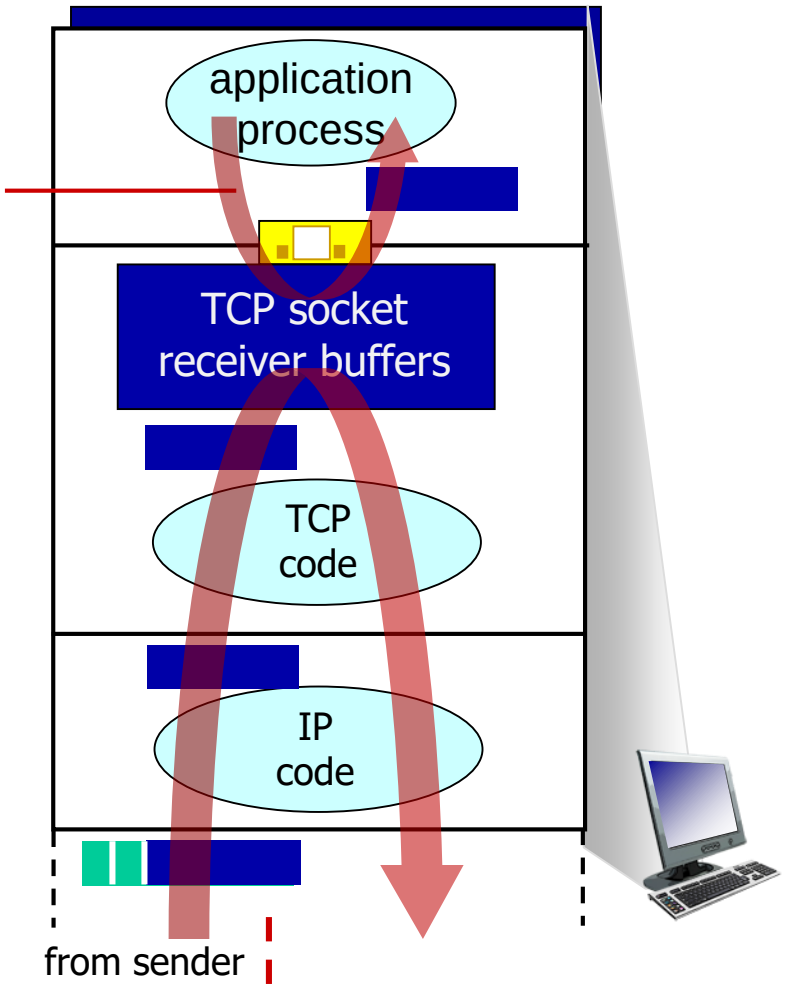
receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

flow control

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



receiver protocol stack

TCP Flow control

- Flow control is the process of managing the rate of data transmission between 2 nodes to prevent a fast sender from overwhelming the slow receiver.

- It is a ***speed- matching service :***

Matching the rate at which the sender is sending against the rate at which receiver application is reading.

TCP Flow Control – reciever Side

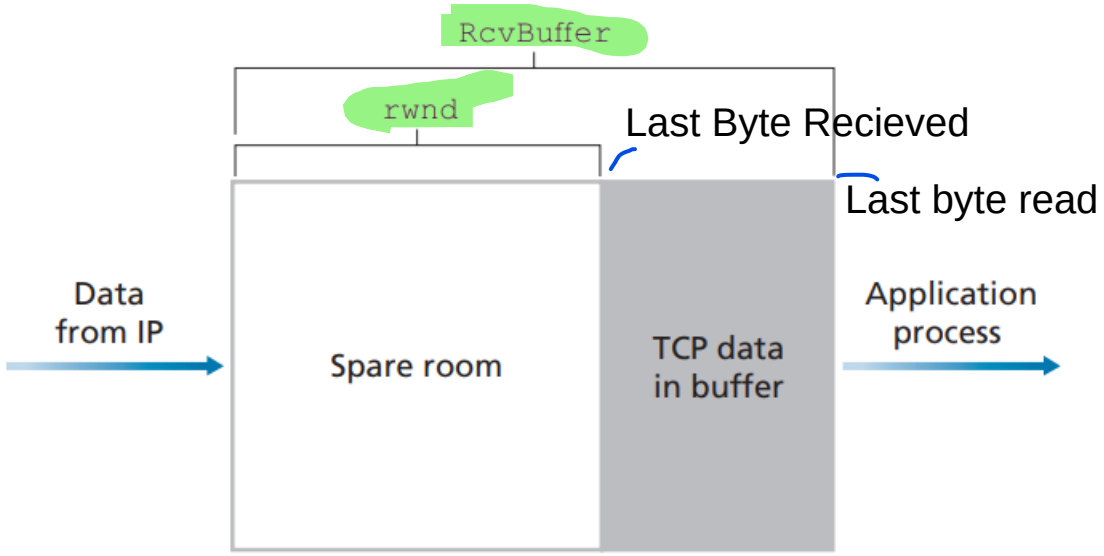


Figure 3.38 ♦ The receive window (rwnd) and the receive buffer (RcvBuffer)

Rwnd = RcvBuffer – [**Last Byte Rcvd - Last Byte Read]**

Initially ***rwnd=RcvBuffer***

- Tcp provides flow control by having sender maintain a variable called receive window**(rwnd).**
  - **Last Byte Read-** The number of last byte in data stream read from the buffer by the receiver application process.
  - **Last Byte Rcvd**: The number of last byte in data stream that has been arrived from the network and has been placed in the buffer.
- To avoid Over flow:

Last Byte Rcvd- Last Byte Read < = RcvBuffer

Flow control-sender side

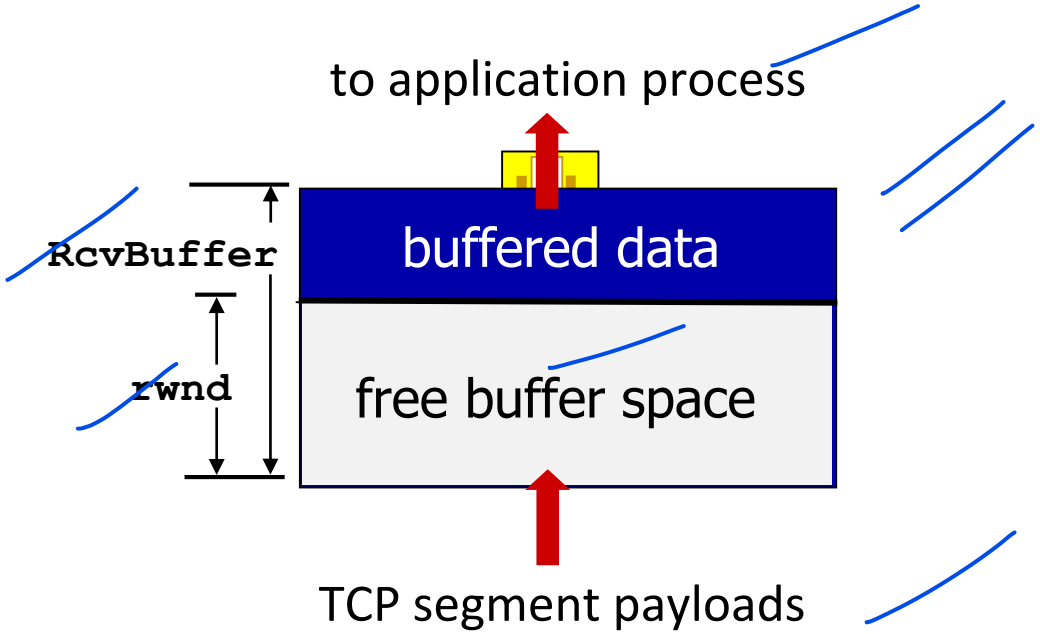
- Sender has to keep track of LastByte_sent and LastByte_Acked, which gives the value of unACK Packets.

- For each packet to be sent , sender checks for rwnd condition:

LastByte_sent - LastByte_Acked <=rwnd

TCP flow control

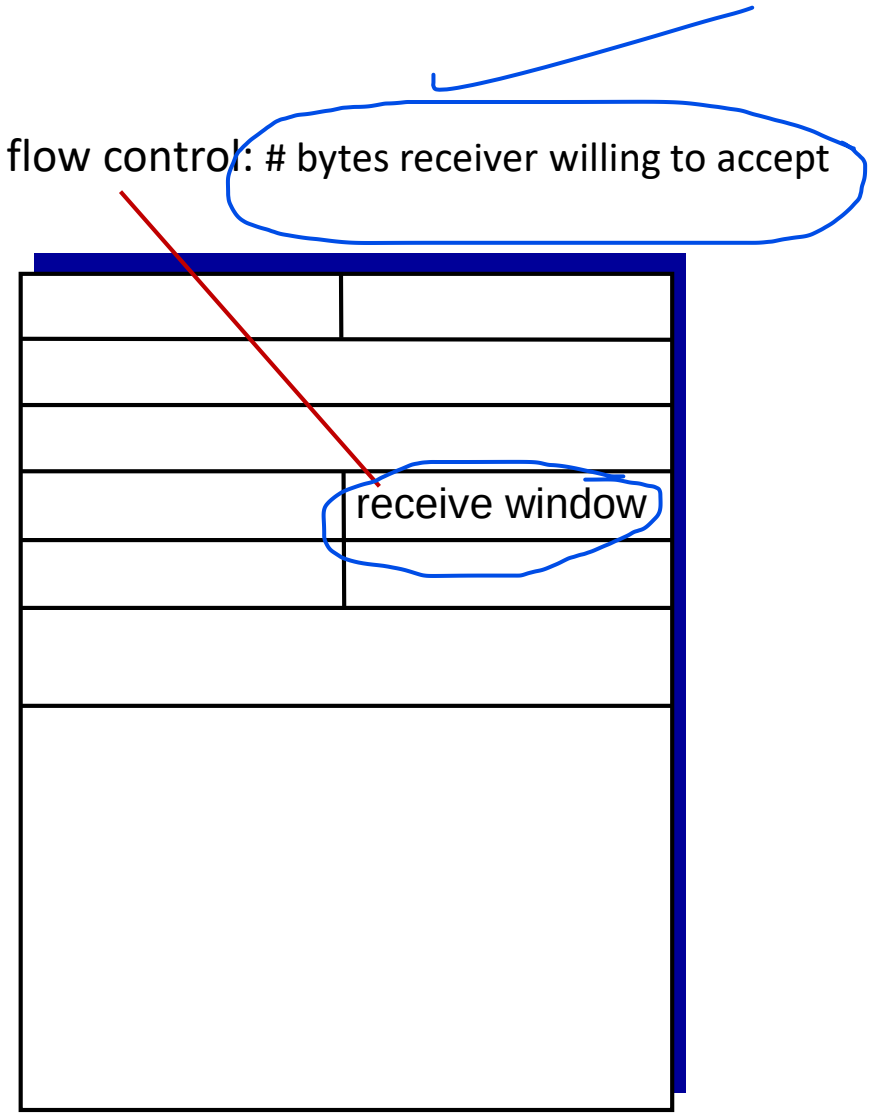
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`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 received **`rwnd`**
- guarantees receive buffer will not overflow

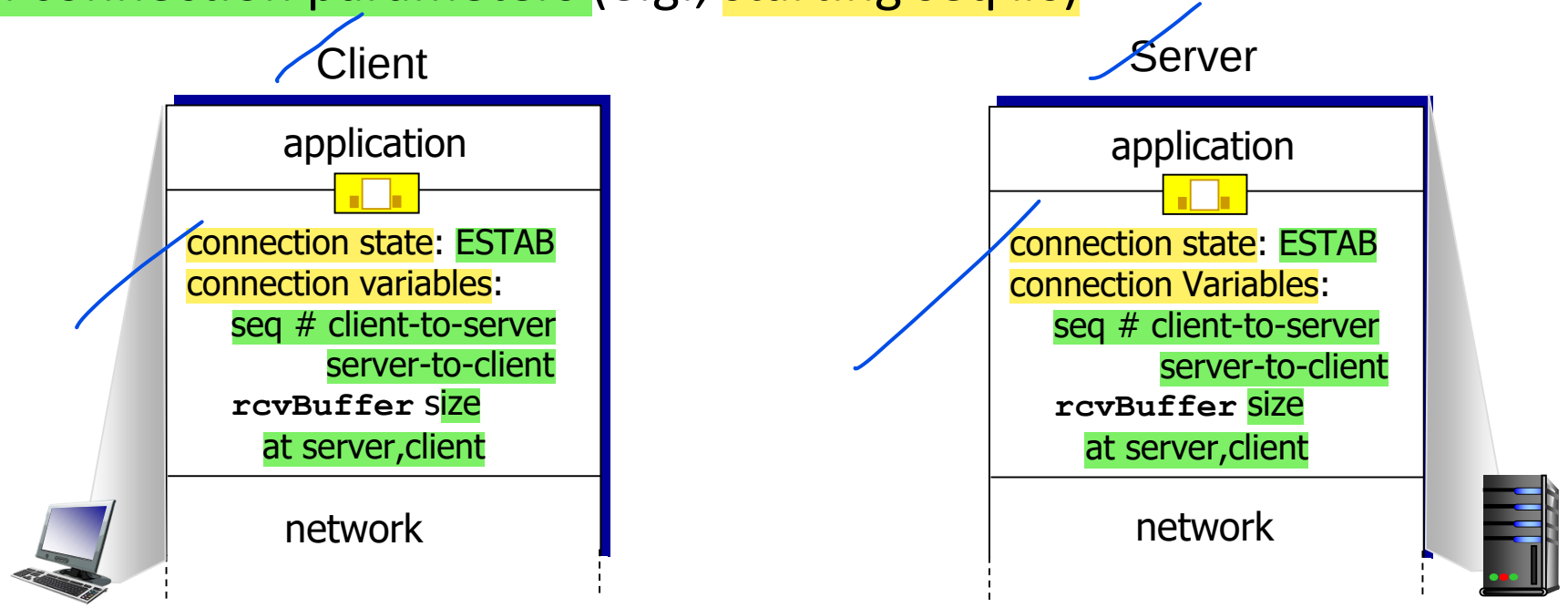


TCP segment format

TCP connection management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

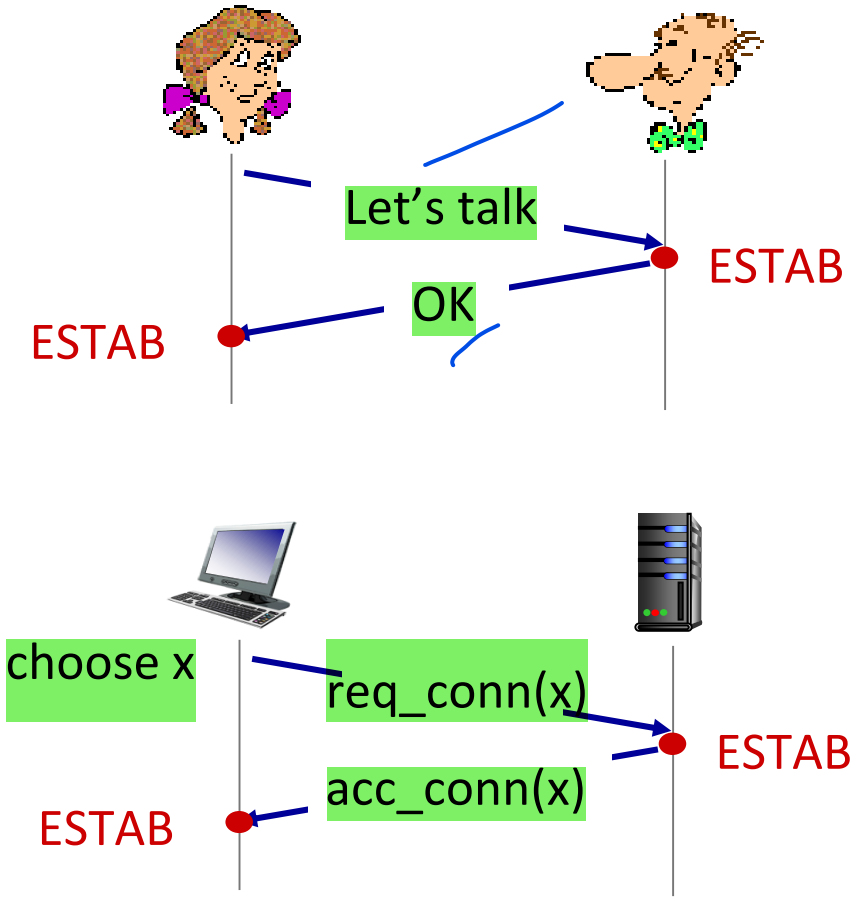



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Agreeing to establish a connection

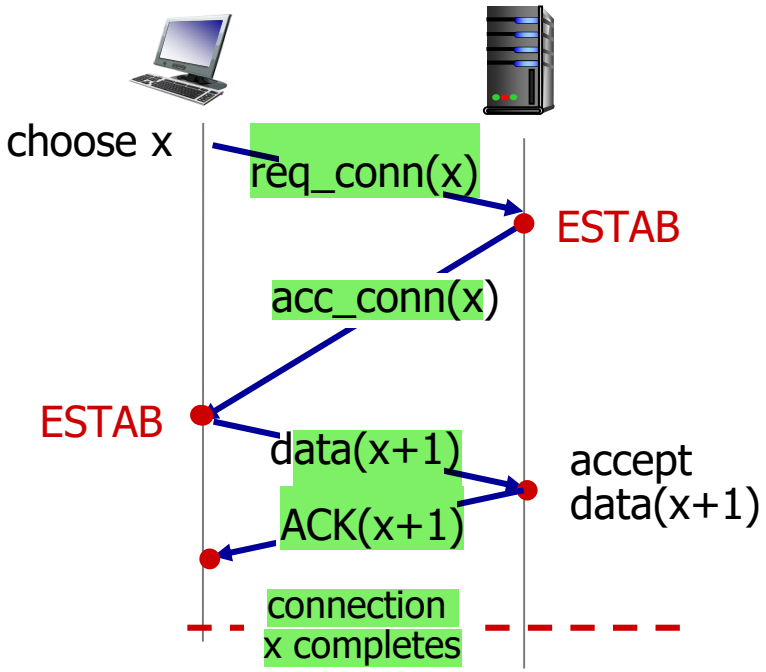
2-way handshake:



Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



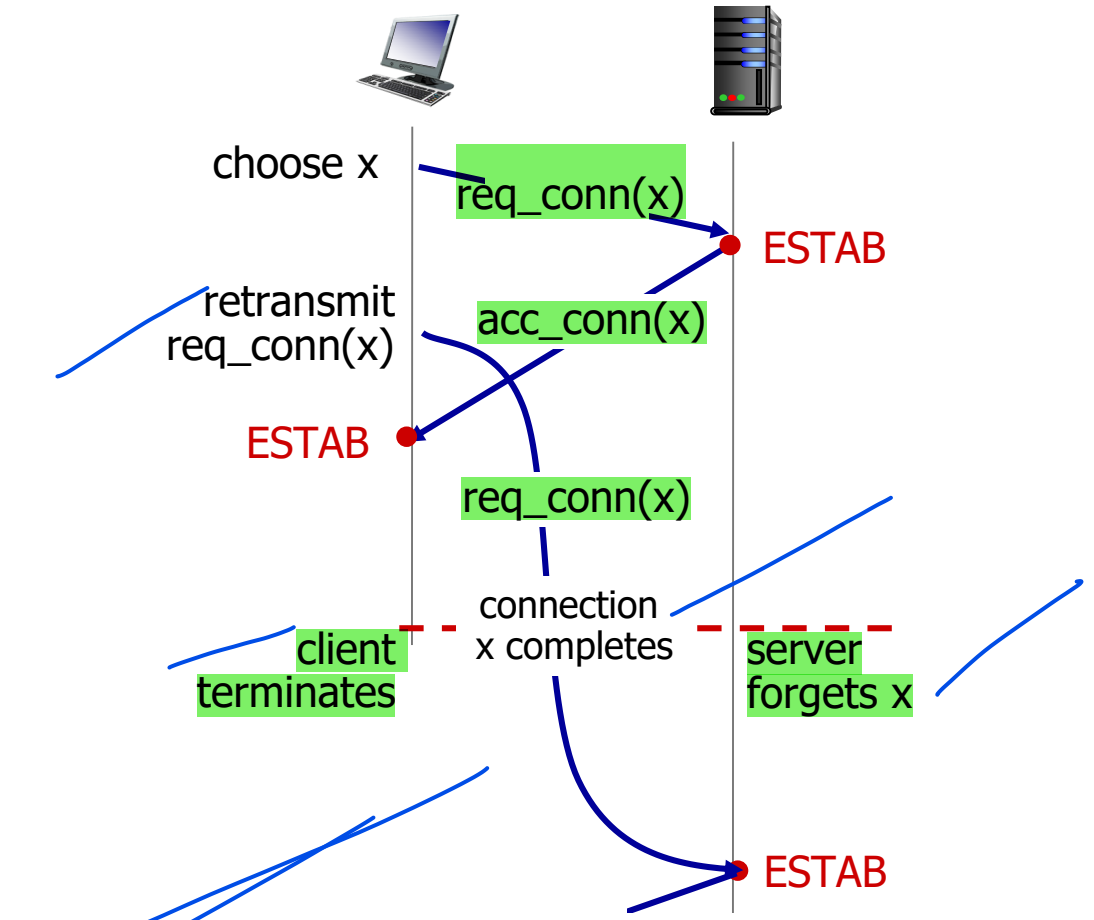
No problem!

1. C: Connect?
2. S: Okay!

Starts DATA TRANSMISSION

2-way handshake scenarios

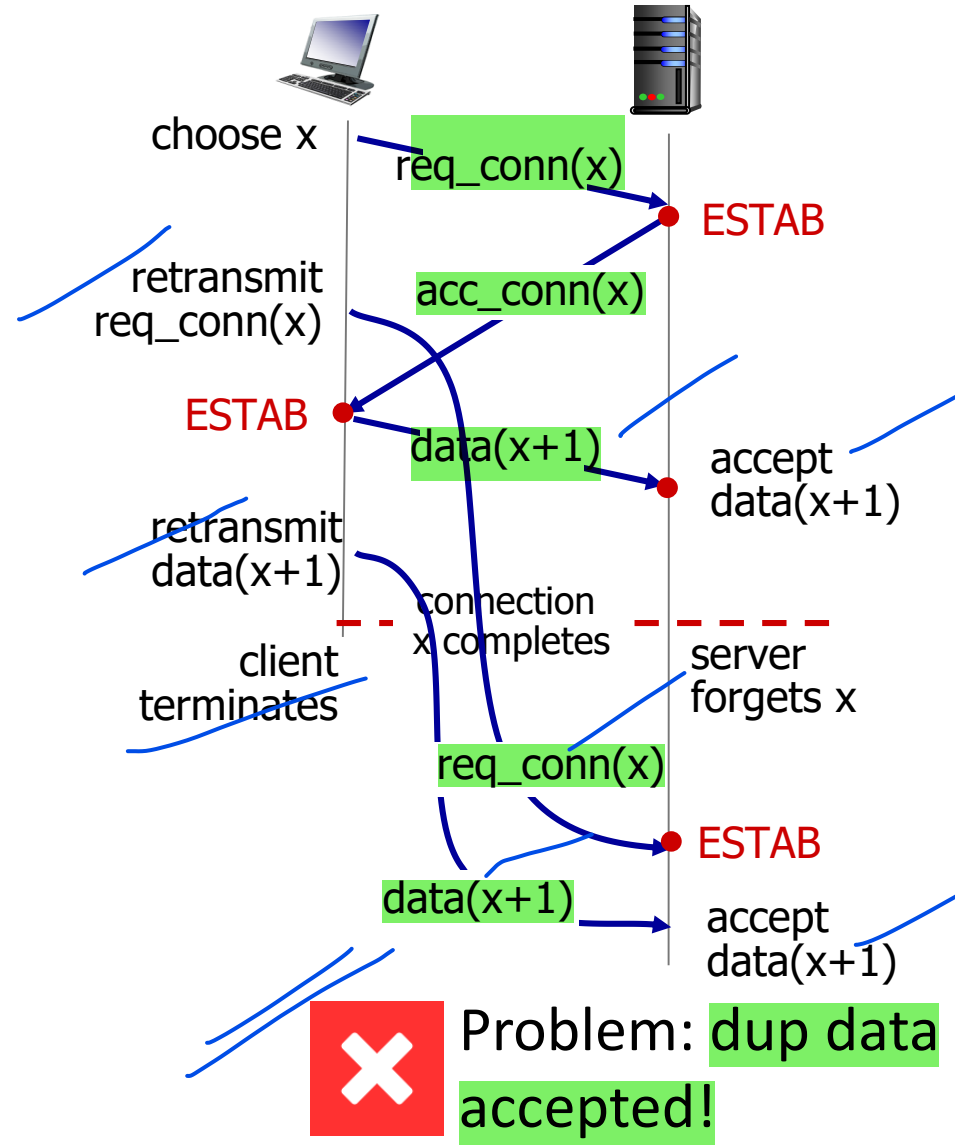
Failure



Problem: half open connection! (no client)

2-way handshake scenarios

Failure



Problem: dup data accepted!

TCP 3-way handshake

1. C: Accept?
2. S: Accepted!
3. C: Transmitting!

Starting a connection: SYN is used
Acknowledgement: ACK is used
Closing a connection: FIN is used

Client state

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

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

A human 3-way handshake protocol

1. C: Accept?
2. S: Accepted!
3. C: Transmitting!

Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

Starting a connection: SYN is used
Acknowledgement: ACK is used
Closing a connection: FIN is used

Chapter 3: roadmap

- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer (Go-back-N and Selective Repeat)
- Connection-oriented transport: TCP
- TCP congestion control
- TCP congestion control

Fast Transmit

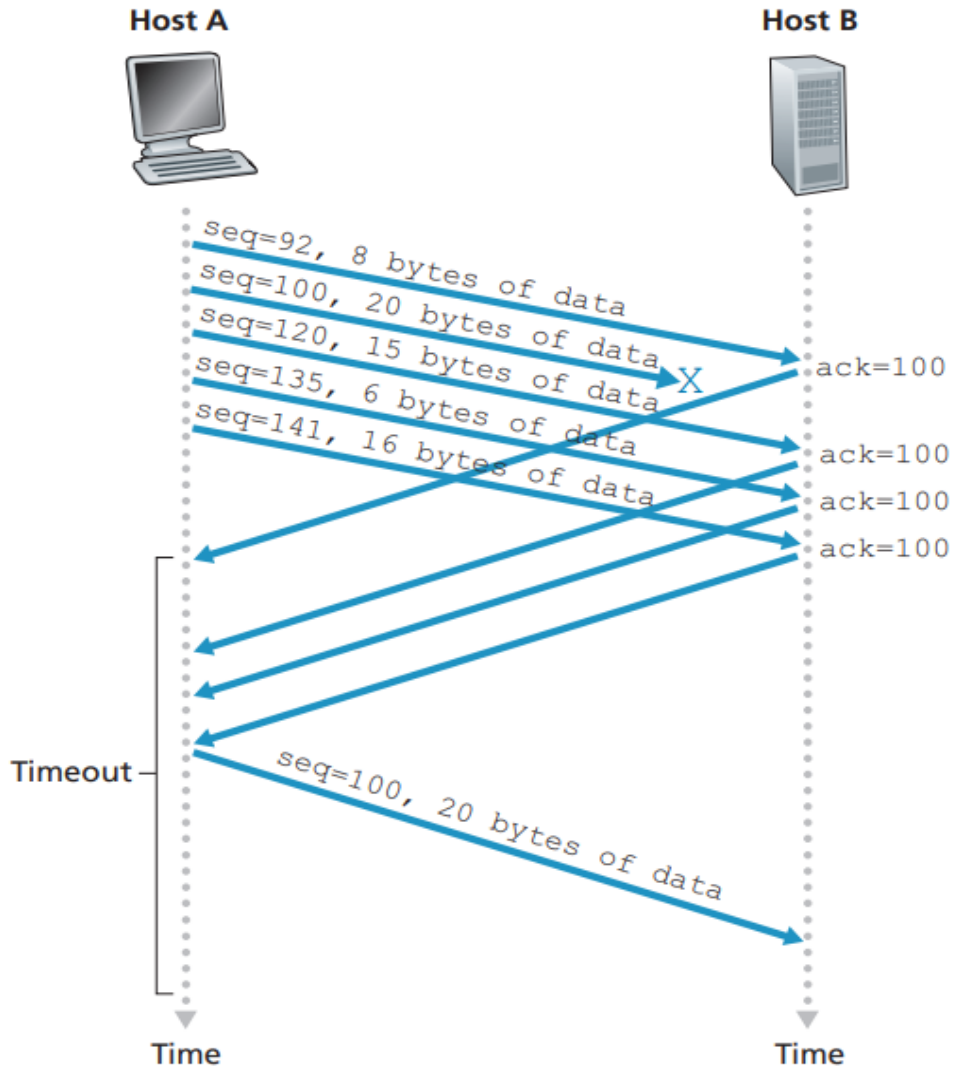


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
            SendBase=y
            if (there are currently any not yet
                        acknowledged segments)
               start timer
               }

  else {/* a duplicate ACK for already ACKed
          segment */
      increment number of duplicate ACKs
          received for y
      if (number of duplicate ACKS received
          for y==3)
           /* TCP fast retransmit */
           resend segment with sequence number y
      }
 break;
```

TCP Congestion Control

- TCP use end to end congestion control as IP layer does not provide explicit feedback to the end system regarding network congestion.
- The approach taken by TCP is to have each sender limit the rate at which it sends the traffic into its connection as a function of perceived network congestion.

If a TCP sender perceives that there is little or no congestion on the path between itself and destination, then the TCP sender increases its sender rate increases its send rate, if there is congestion, then the sender reduces the send rate.

How does a TCP sender limit the rate at which it sends the traffic into its connection?

Sender keep tracks of additional variable – cwnd (Congestion Window)

TCP Congestion Control

Congestion Window (cwnd) is a TCP state variable that limits the amount of data the TCP can send into the network before receiving an ACK.

- The **Receiver Window (rwnd)** is a variable that advertises the amount of data that the destination side can receive.

Together, the two variables are used to regulate data flow in TCP connections, minimize congestion, and improve network performance.

The amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:

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

same eqns

LastByteRecived - LastByteRead <= Rwnd/RcvBuffer

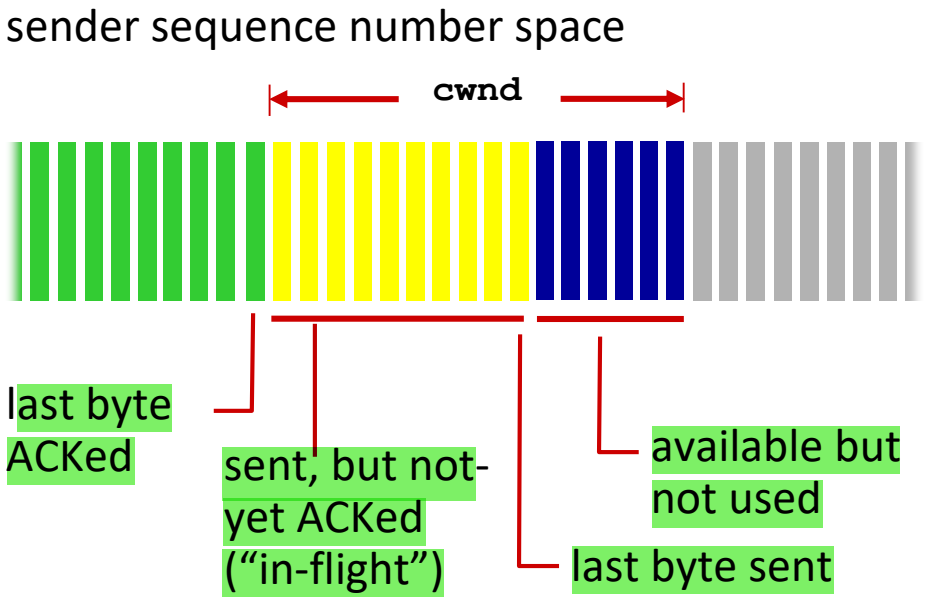
TCP maintains the sending rate by following the principles namely

- 1. Lost segment implies congestion and hence the sender's rate should be decreased when a segment is lost.

- 3. Bandwidth Probing: ***sstrsh(slow start threshold)***

- 2. An acknowledged segment indicates that the network is delivering the sender's segment to the receiver and hence the **sender rate can be increased** when an **ACK arrives** for the previously **unacknowledged frame.**

TCP congestion control: details



TCP sending behavior:

- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

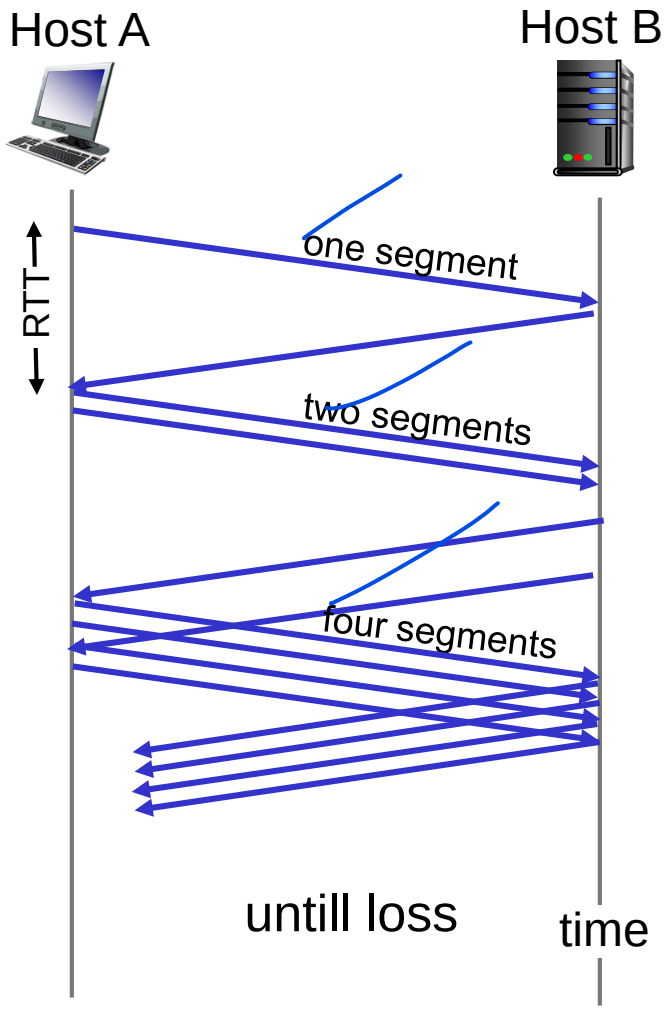
- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP Congestion control includes:

- 1. Slow start.
- 2. Congestion Avoidance.
- 3. Fast recovery

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



When should this exponential growth ends?

- 1. Lost segment (Time out):

Ssthresh = cwnd/2;

Slow Start THRESHold

- 2. when **cwnd >=ssthresh**
  - Then TCP enters Congestion Avoidance mode.

3. When three DUPACK are received:

Enter Fast recovery state :

Cwnd= sstresh+3MSS

Sstresh= cwnd/2

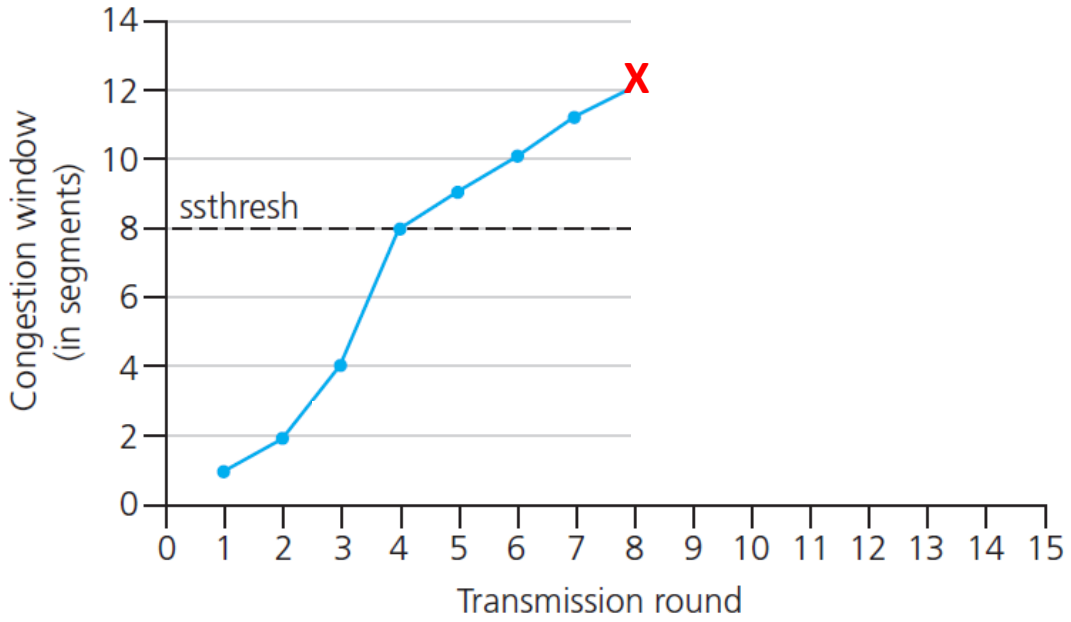
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

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

Implementation:

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



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Congestion Avoidance

- On entry on congestion state, value of cwnd will be half.
- TCP now adopts a more conservative approach I,e:

Cwnd=cwnd + MSS[MSS/cwnd]

This increases cwnd by 1/10 MSS for each iteration

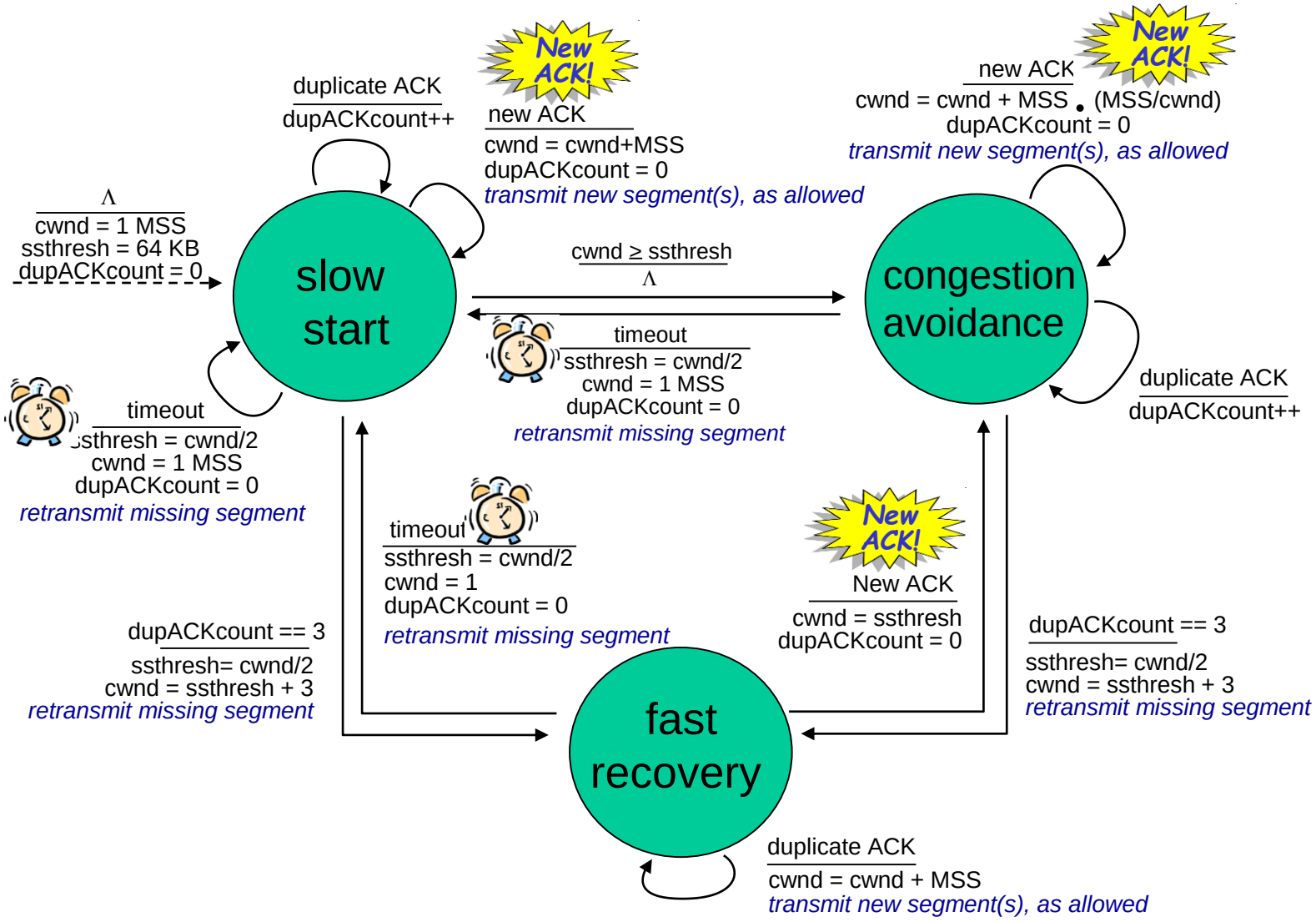
- Congestion avoidance linear increase ends when timeout occurs or if 3 ACK duplicative is received.

Fast recovery

- In fast recovery , the value of cwnd is increased by 1MSS for every duplicate ACK.

- TCP enters back :
  - **To Congestion Avoidance**: on new ACK
  - **To Slow start**: if Time out occurs.

Summary: TCP congestion control



TCP congestion control: AIMD

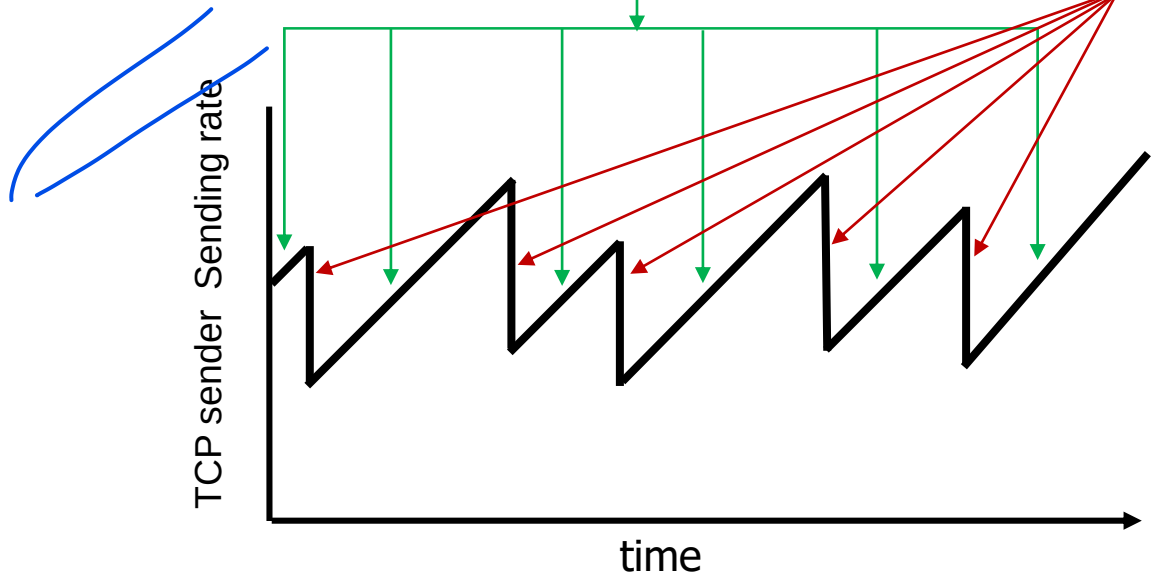
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

Multiplicative decrease detail: sending rate is

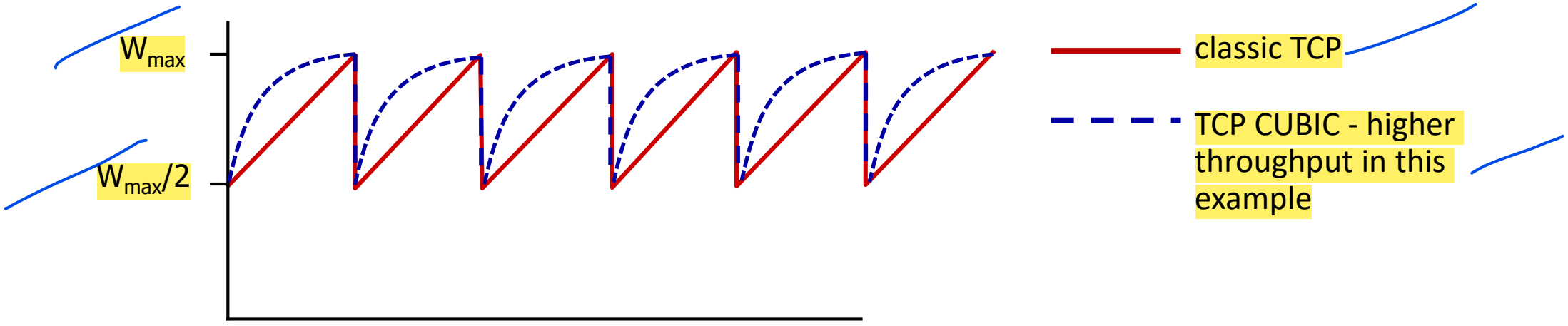
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

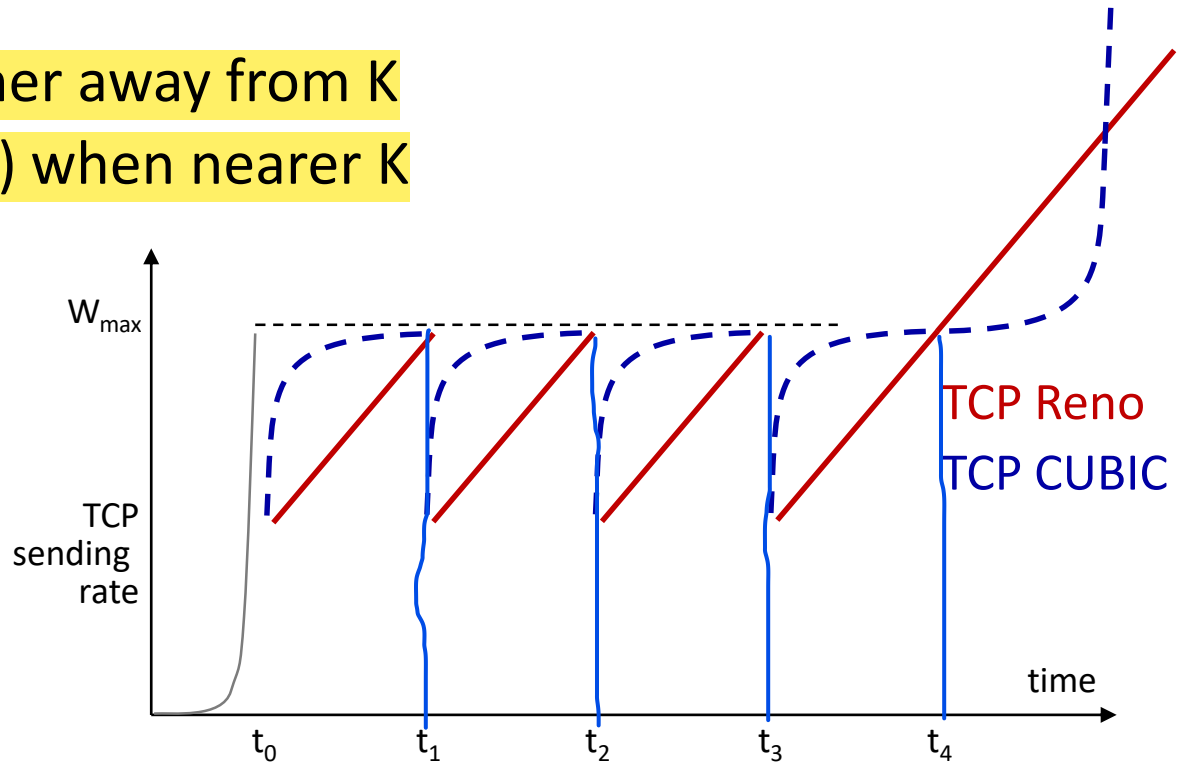
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



Chapter 3: summary

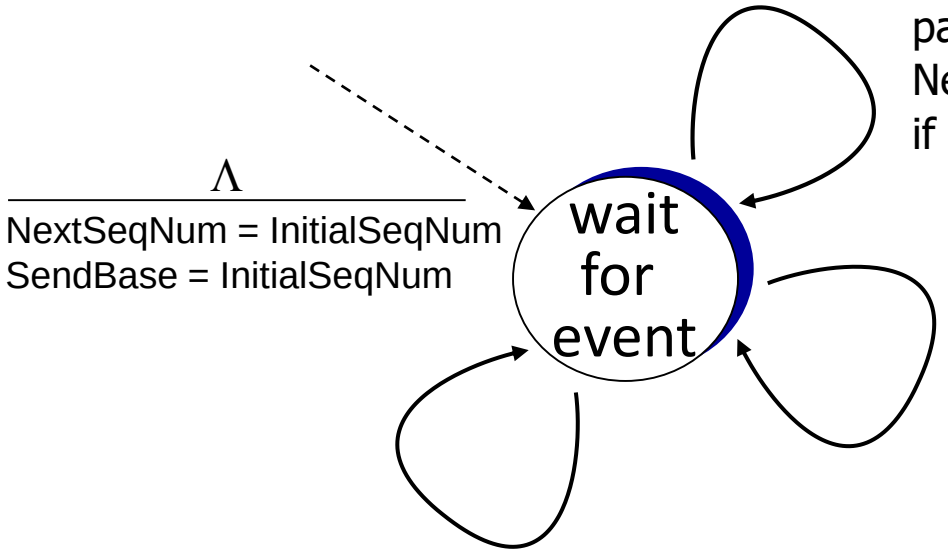
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

Additional Chapter 3 slides

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

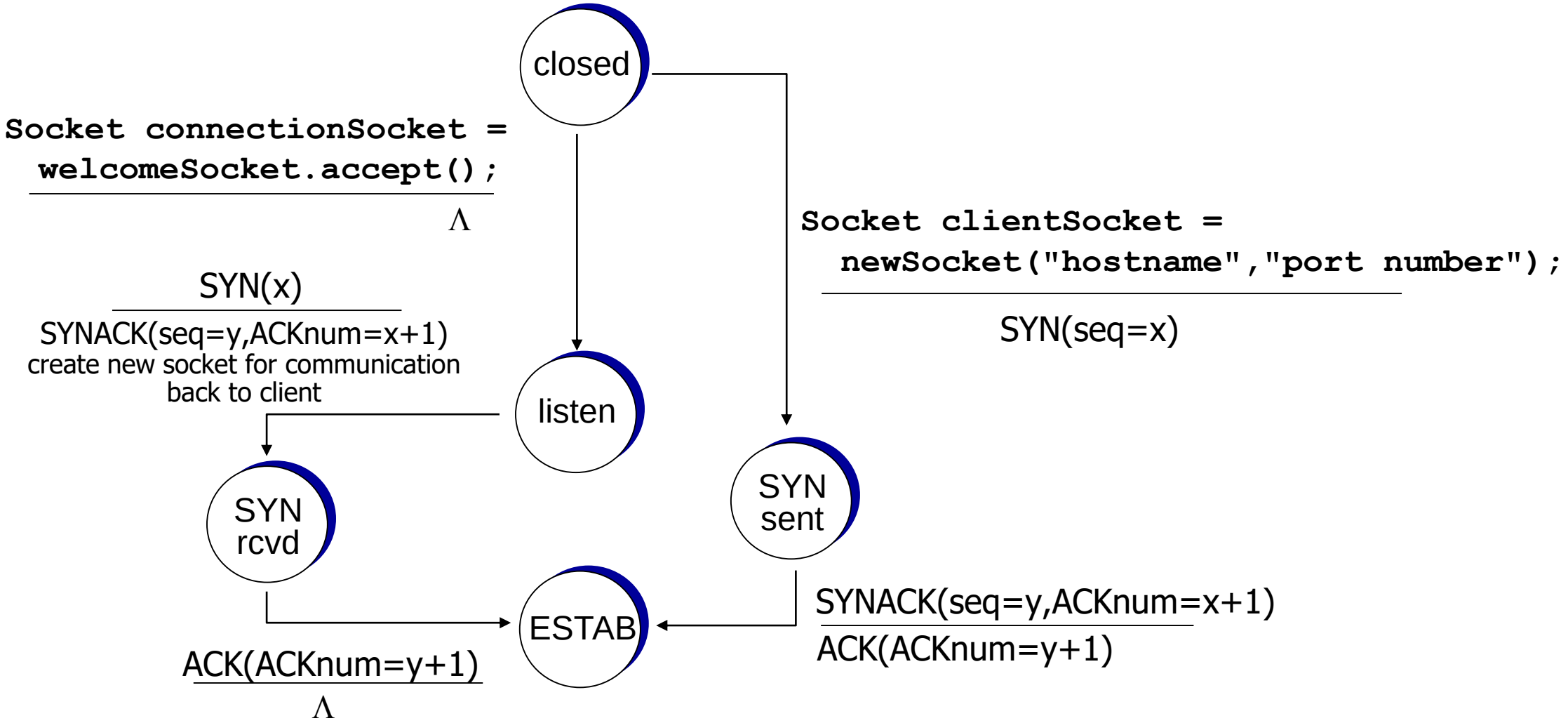
timeout

retransmit not-yet-acked segment
        with smallest seq. #
start timer

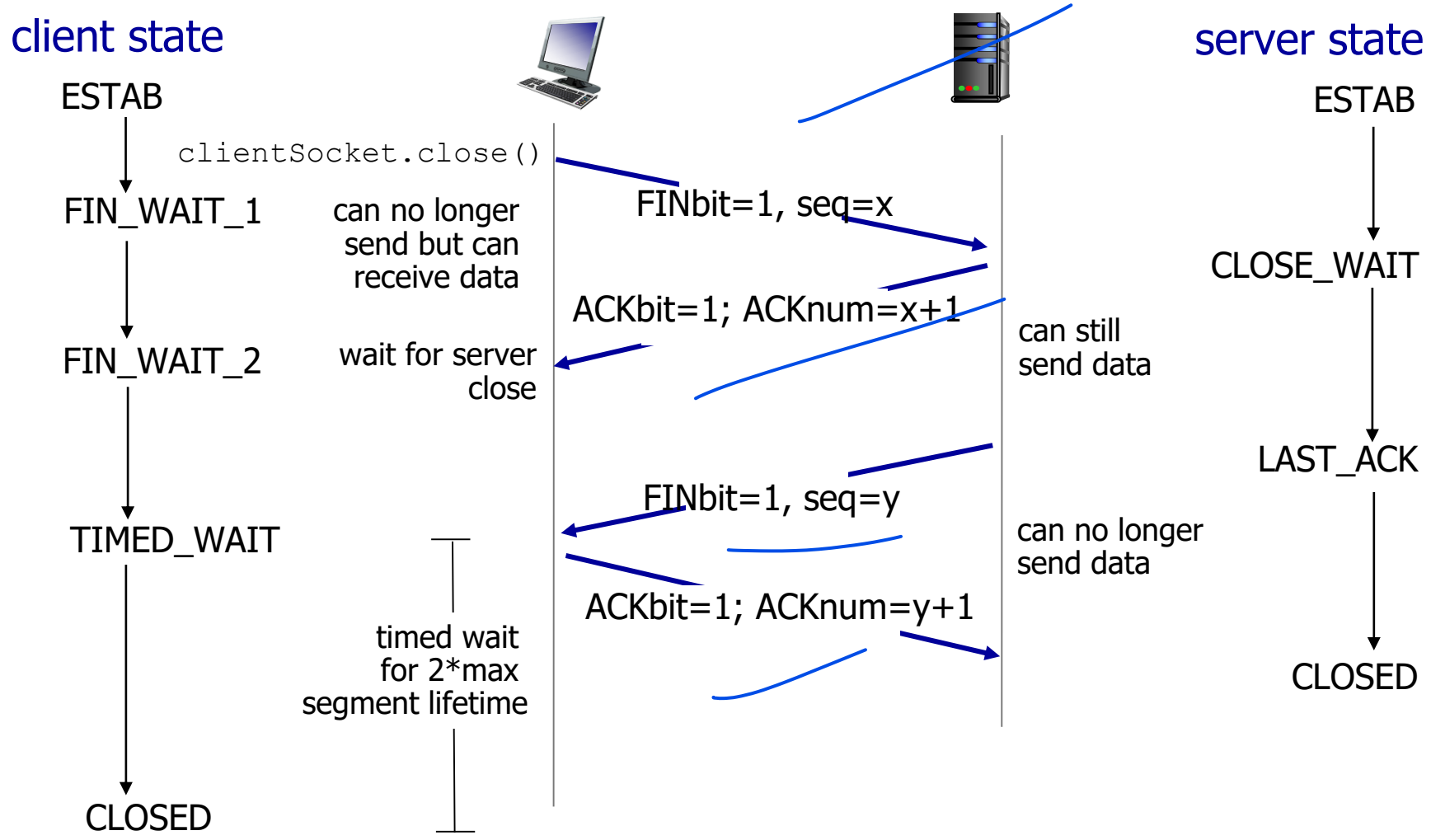
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
     }
```

TCP 3-way handshake FSM



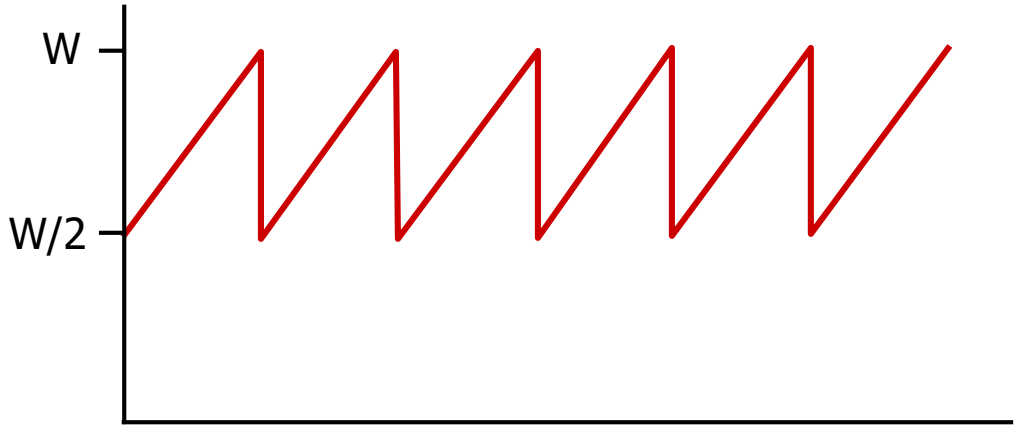
Closing a TCP connection



TCP throughput

- avg. TCP througput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ – *a very small loss rate!*

- versions of TCP for long, high-speed scenarios