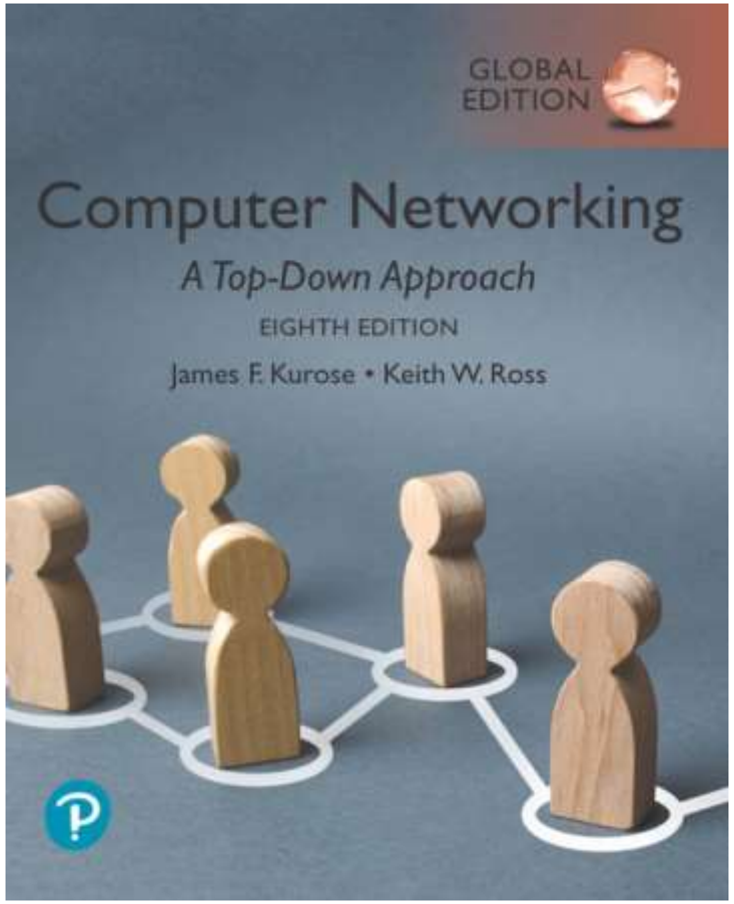
Chapter 3
Transport Layer

Ren Ping Liu
renping.liu@uts.edu.au
adapted from textbook slides by JFK/KWR

25 March 2024



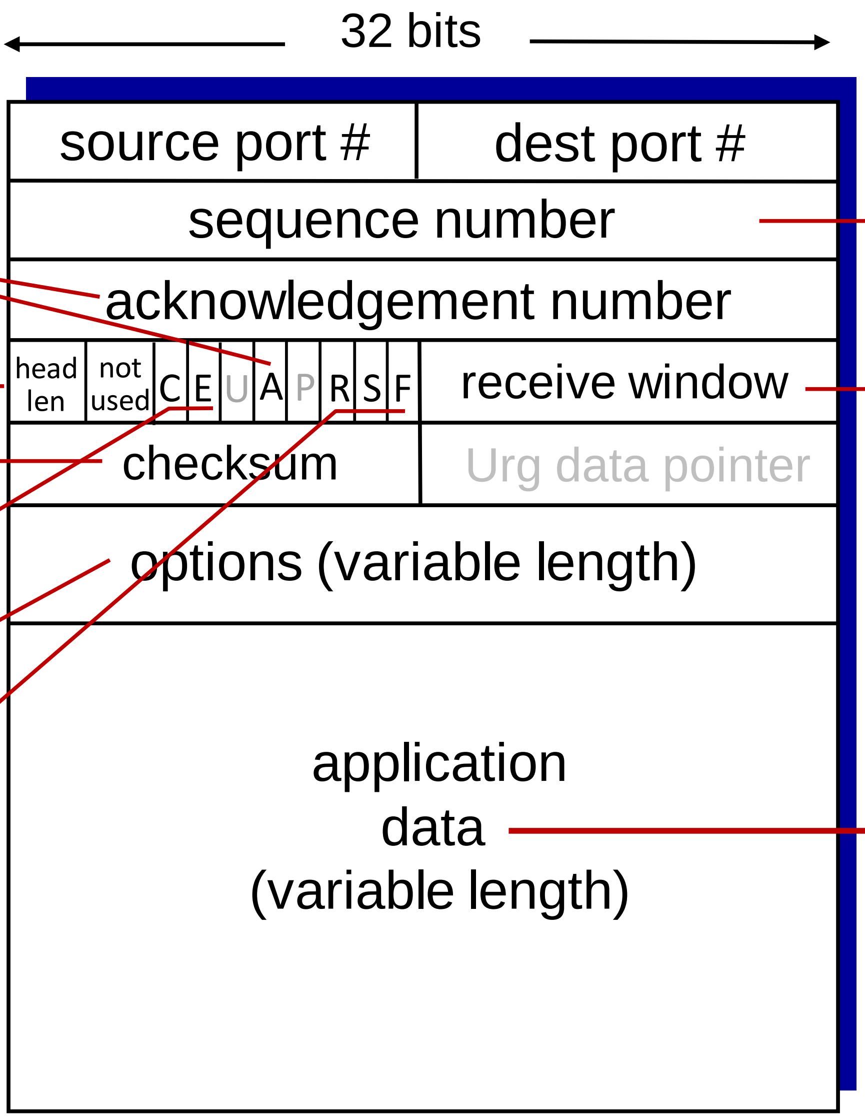
Computer Networking: A Top-Down Approach
8th Edition, Global Edition
Jim Kurose, Keith Ross

Transport layer: roadmap

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

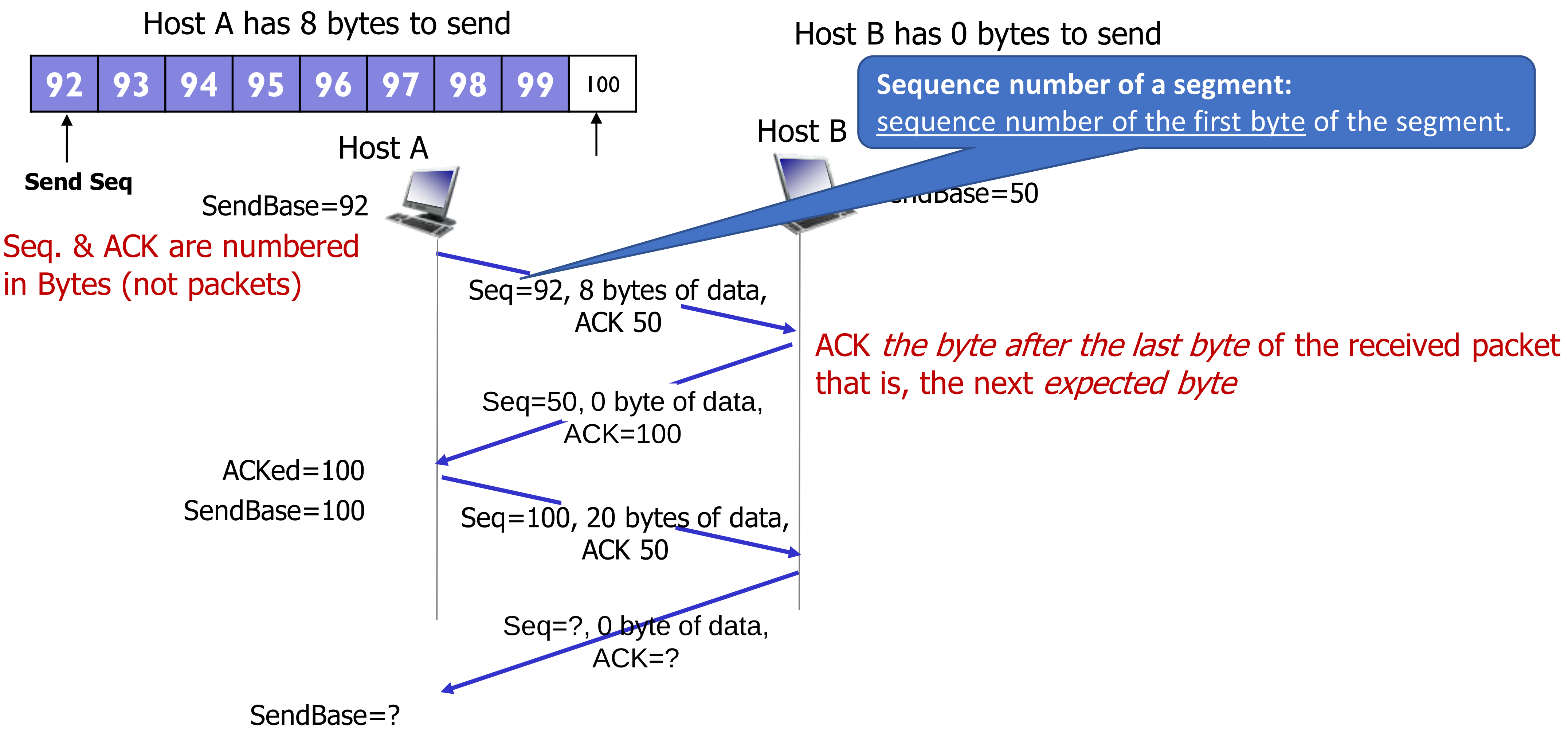
RST, SYN, FIN: connection management

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: Seq. ACKs - one way: A→B

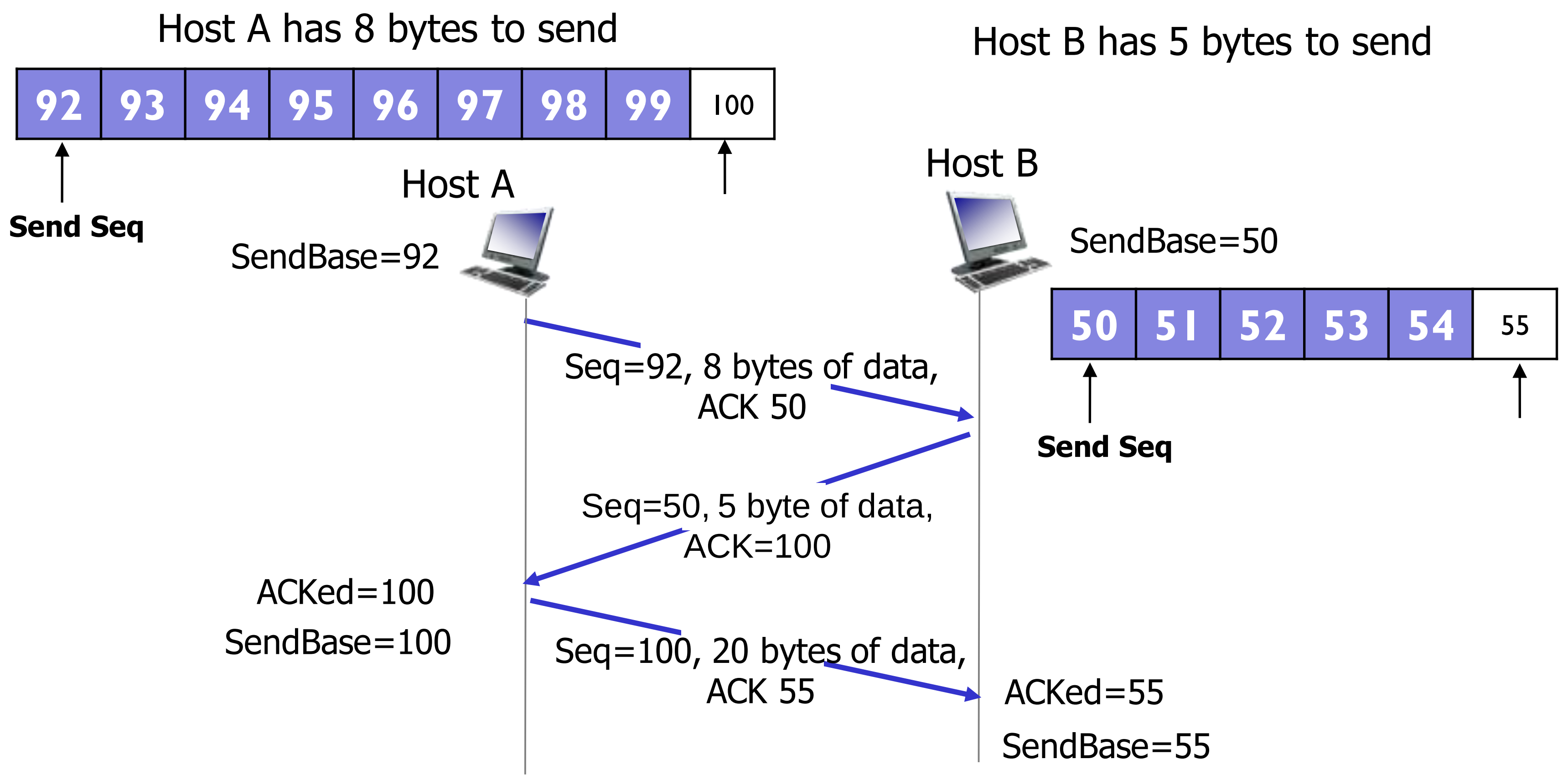


Sequence number of a segment:
sequence number of the first byte of the segment.

Seq. & ACK are numbered in Bytes (not packets)

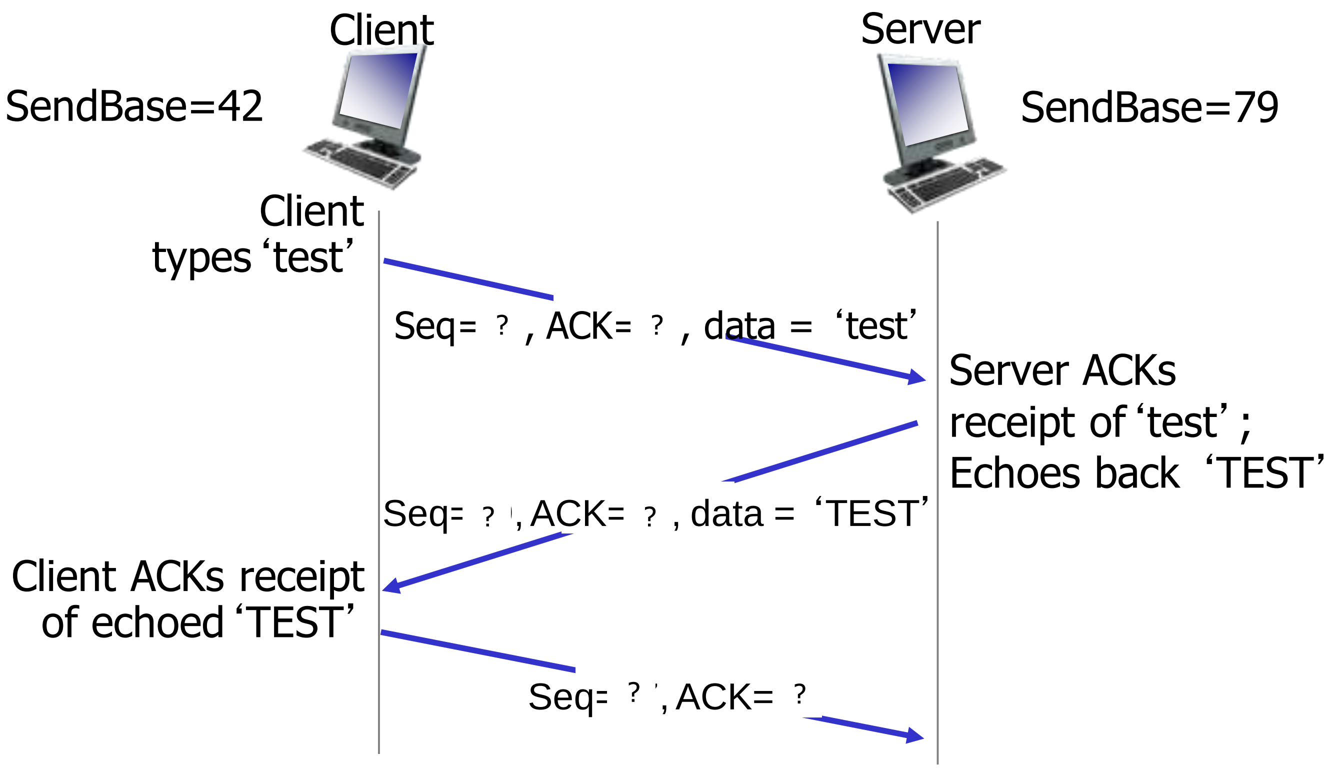
ACK *the byte after the last byte* of the received packet that is, the next *expected byte*

TCP: Seq. ACK - bidirectional


Host A has 8 bytes to send
92
93
94
95
96
97
98
99
100
Send Seq
Host A
SendBase=92
Host B has 5 bytes to send
Host B
SendBase=50
50
51
52
53
54
55
Send Seq
Seq=92, 8 bytes of data, ACK 50
Seq=50, 5 byte of data, ACK=100
ACKed=100
SendBase=100
Seq=100, 20 bytes of data, ACK 55
ACKed=55
SendBase=55

TCP Seq. ACKs - socket prog example

Client send 'test' (5Bytes) to Server, Server echo 'TEST' back


Client
Server
SendBase=42
SendBase=79
Client types 'test'
Seq= ? , ACK= ? , data = 'test'
Server ACKs receipt of 'test' ; Echoes back 'TEST'
Seq= ? , ACK= ? , data = 'TEST'
Client ACKs receipt of echoed 'TEST'
Seq= ? , ACK= ?

simple TCP client / server Python Socket Programming

Transport layer: roadmap

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

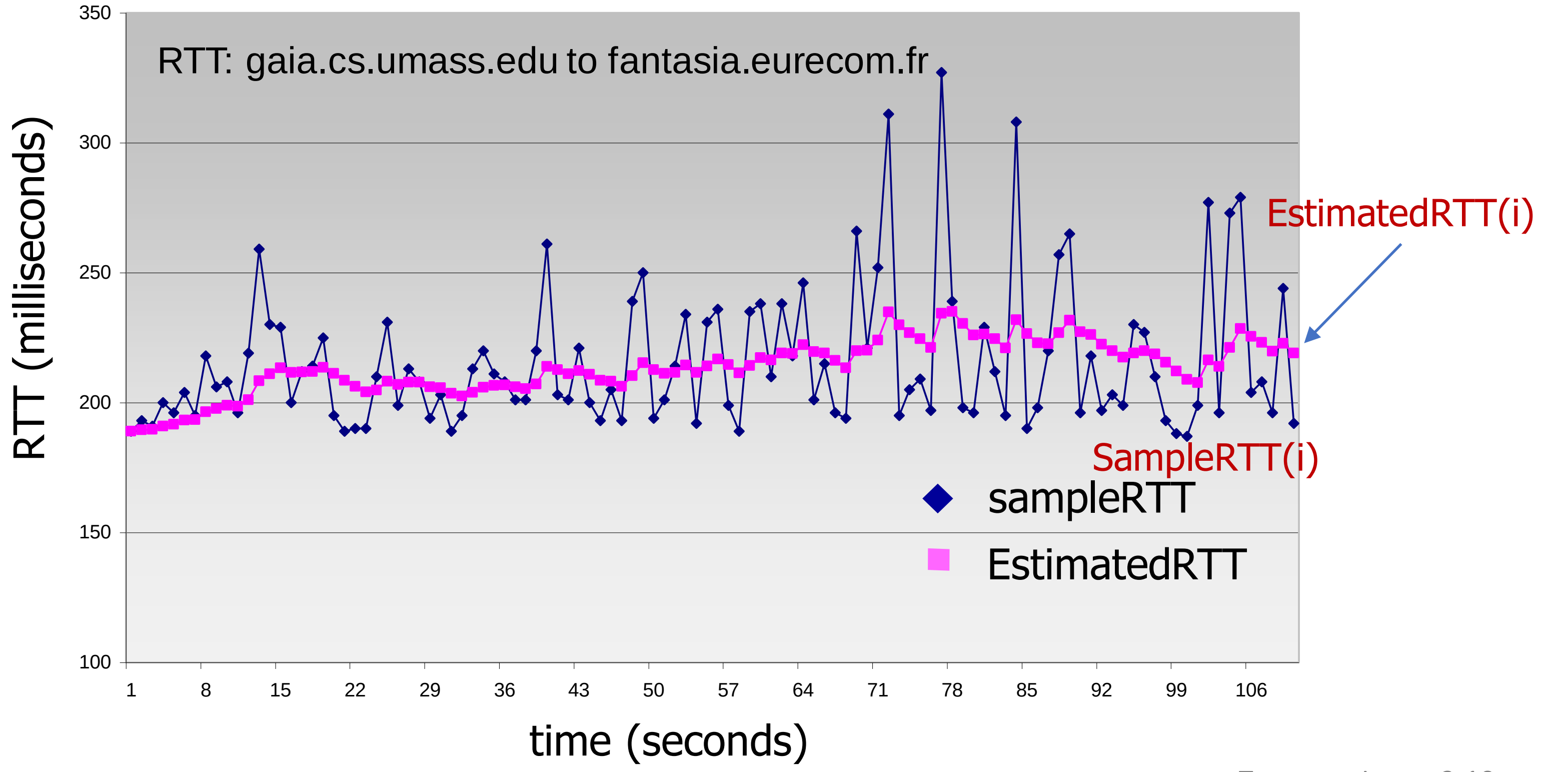
Q: how to estimate RTT?

- `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

$$\texttt{EstimatedRTT}_{(i)} = (1-\alpha)*\texttt{EstimatedRTT}_{(i-1)} + \alpha*\texttt{SampleRTT}_{(i)}$$

- exponential weighted moving average (EWMA)
- 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:`** 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/

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
  - expiration interval: **`TimeOutInterval`**
- transmit the segment

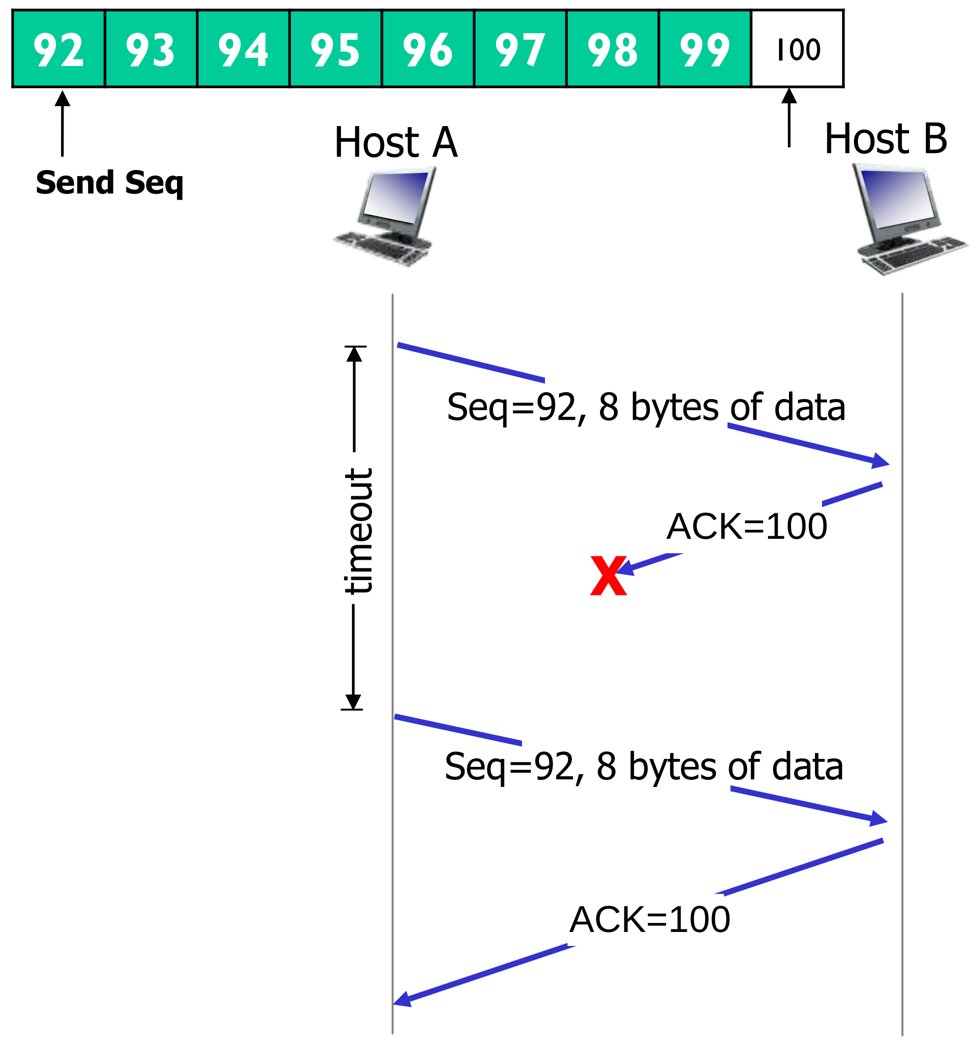
event: ACK received

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

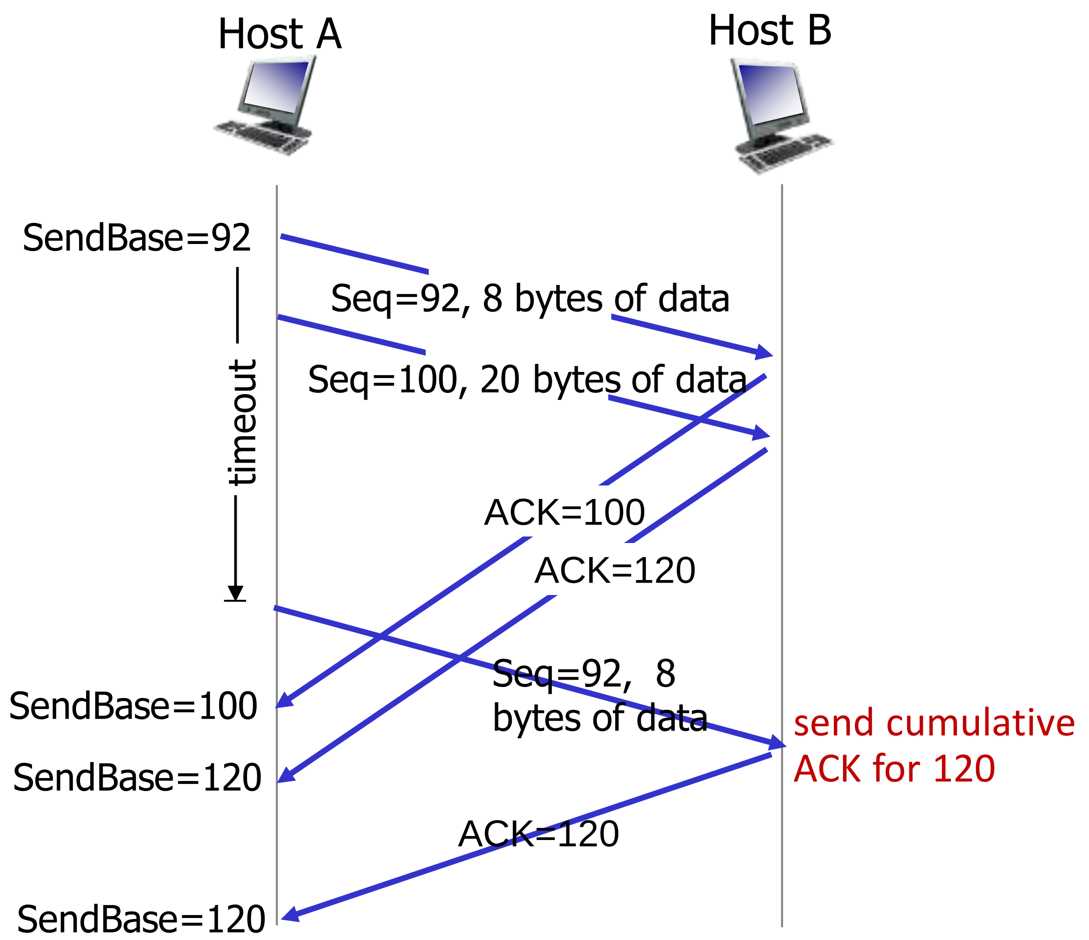
event: timeout

- retransmit segment that caused timeout
- restart timer

TCP retransmission - Timeout

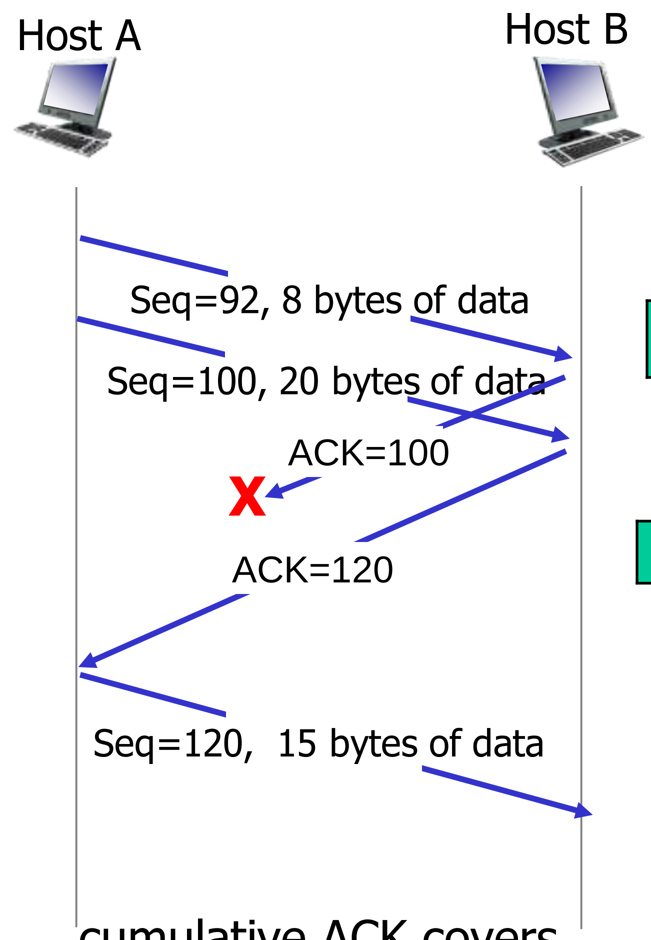


lost ACK scenario

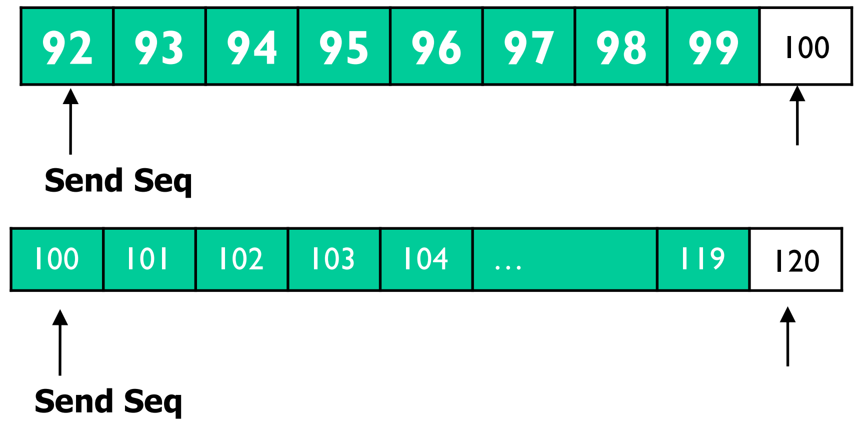


premature timeout

TCP retransmission - Cumulative ACK



cumulative ACK covers for earlier lost ACK



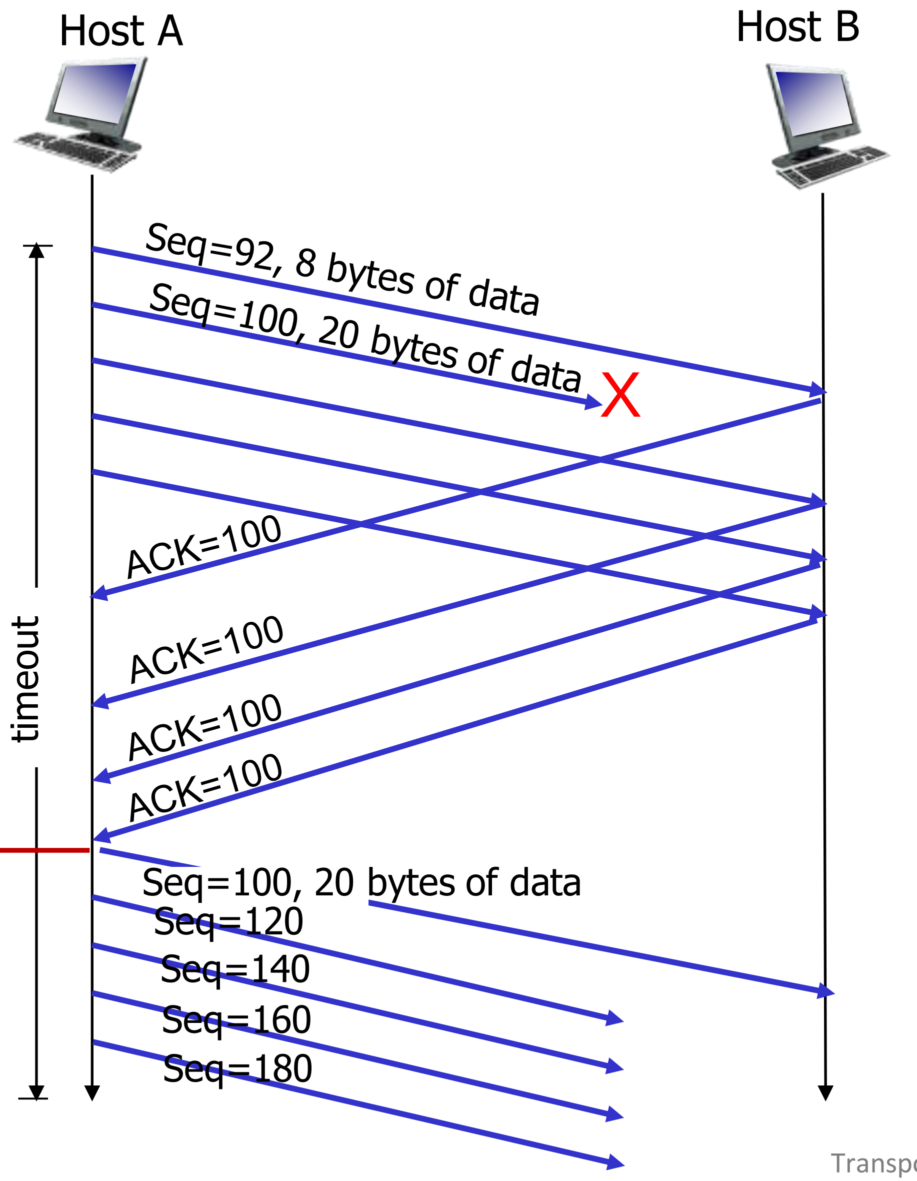
TCP retransmission - Fast retransmit

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

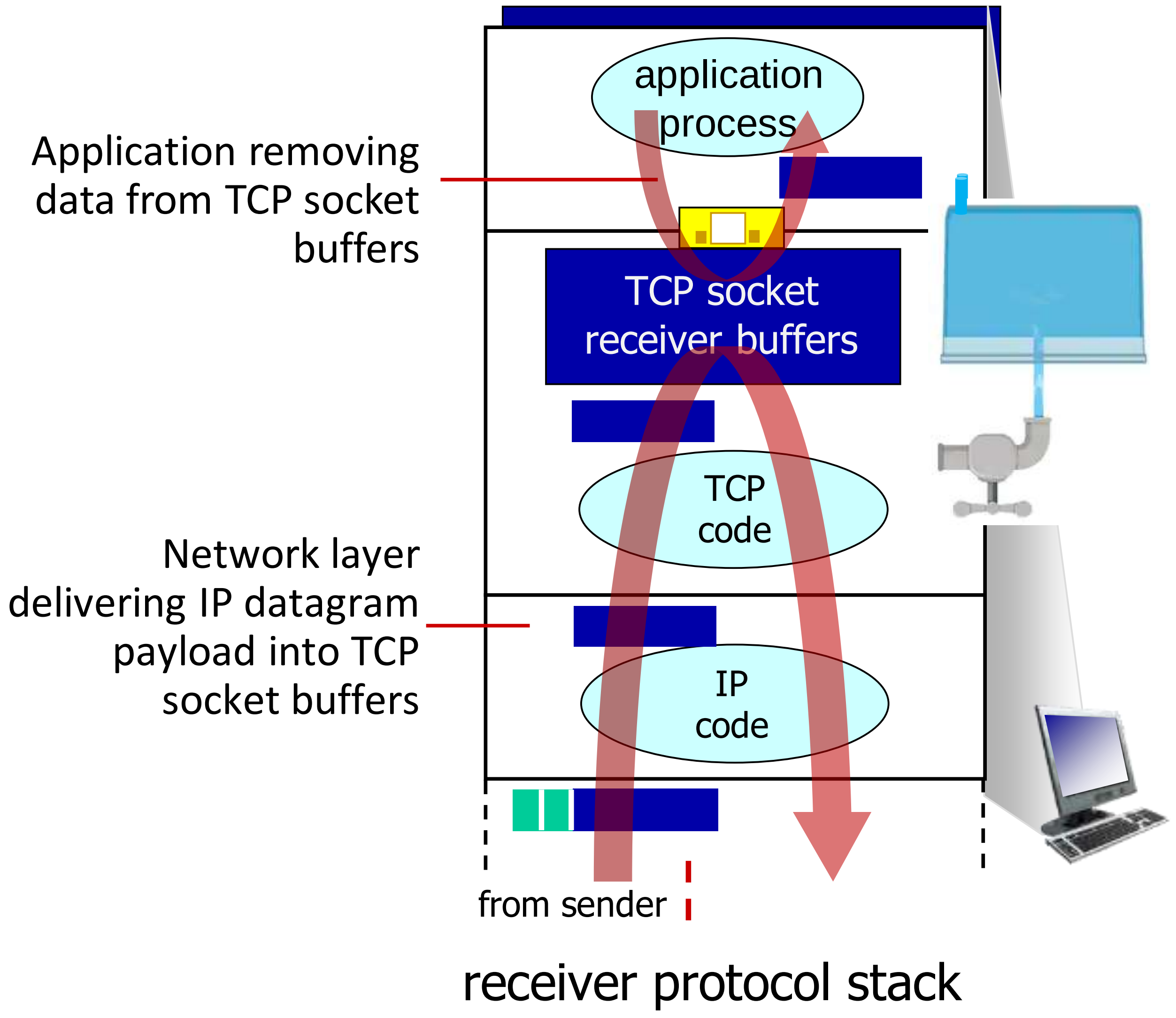
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



Transport layer: roadmap

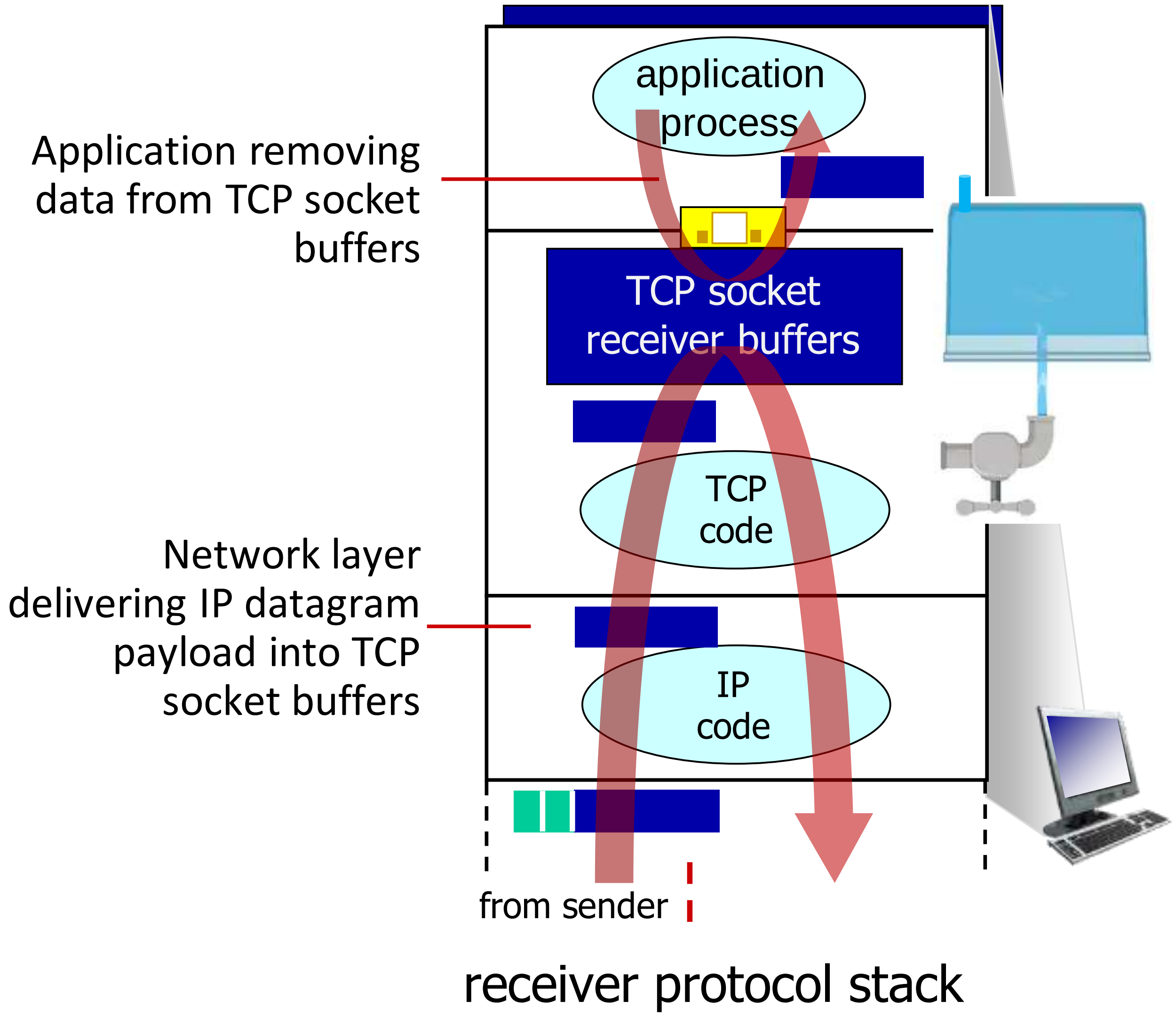
TCP flow control

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


Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
Network layer delivering IP datagram payload into TCP socket buffers
IP code
from sender
receiver protocol stack

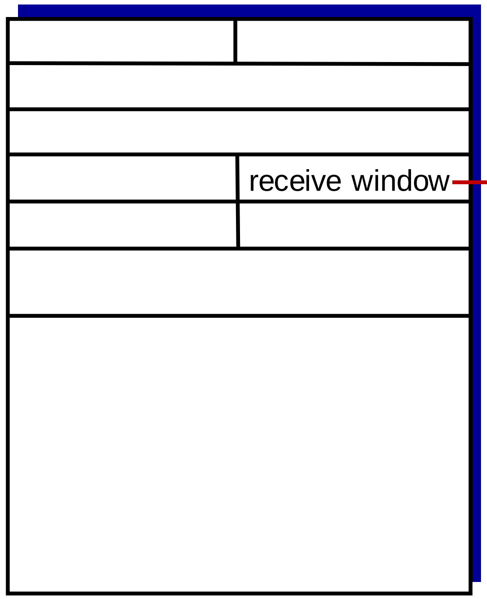
TCP flow control

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

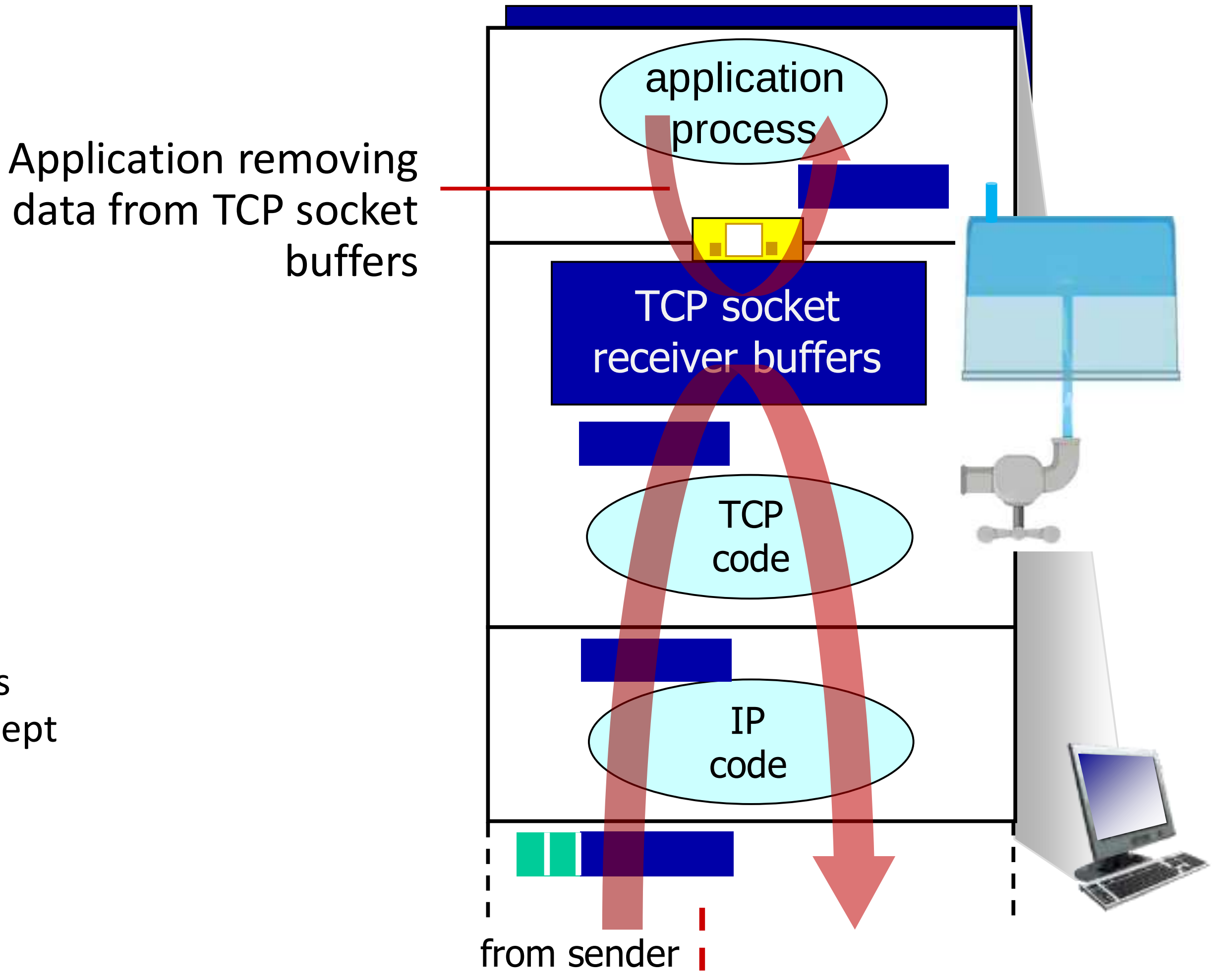

application process
Application removing data from TCP socket buffers
TCP socket receiver buffers
TCP code
Network layer delivering IP datagram payload into TCP socket buffers
IP code
from sender
receiver protocol stack

TCP flow control

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



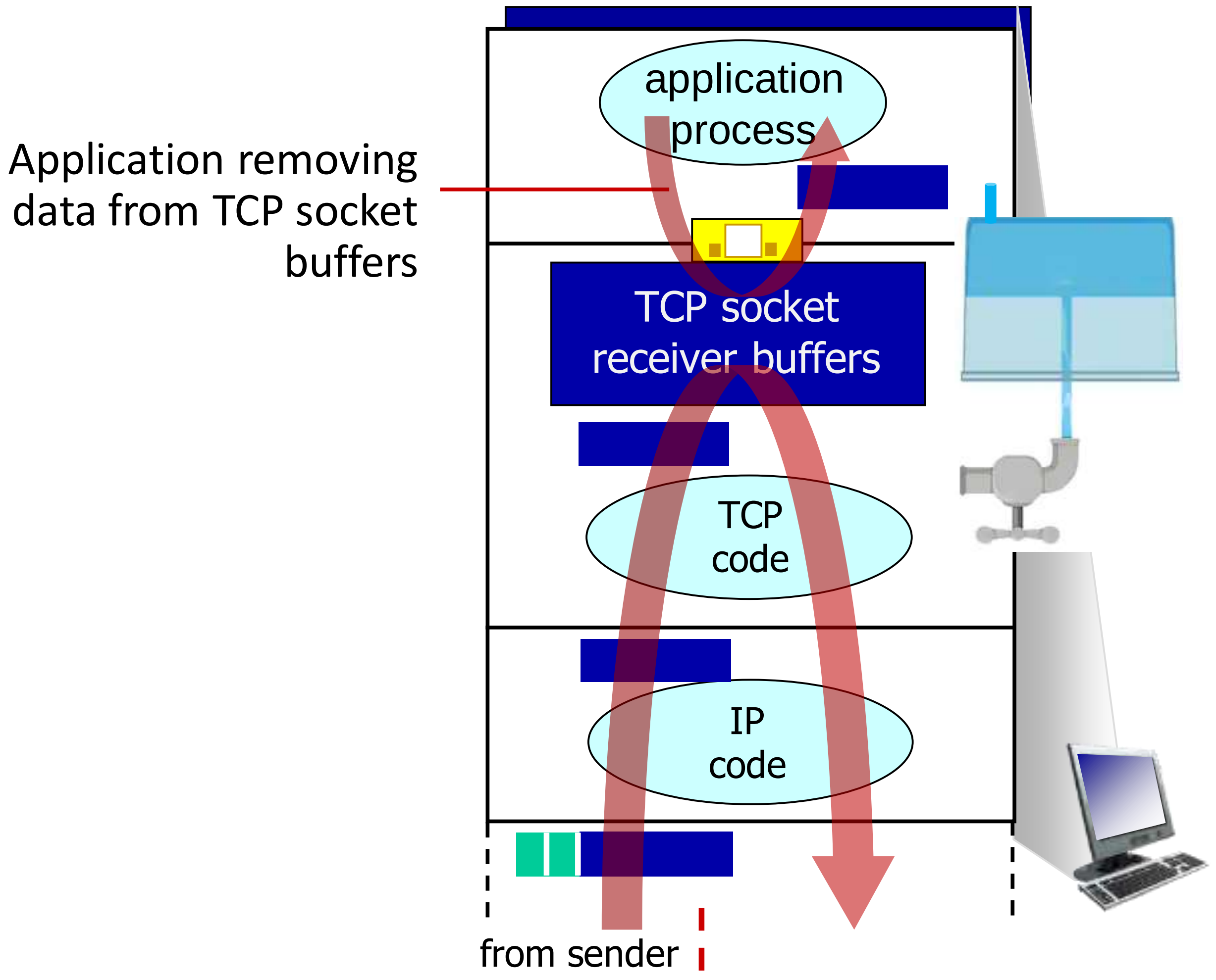
flow control: # bytes receiver willing to accept


Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
IP code
from sender

receiver protocol stack

TCP flow control

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


Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
IP code
from sender

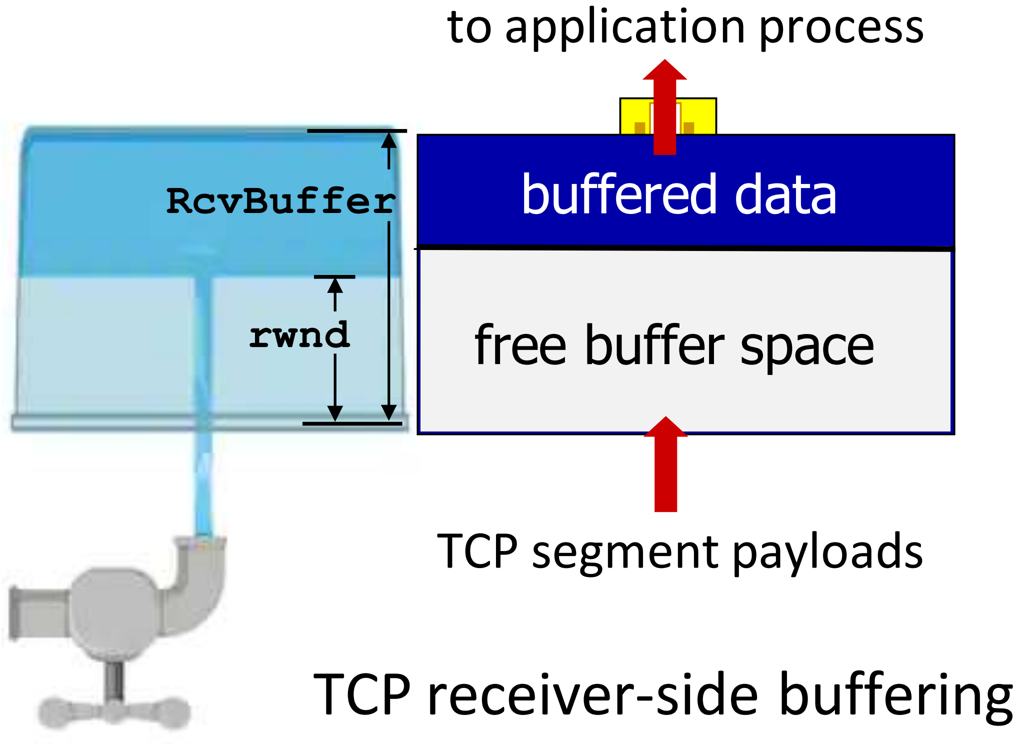
receiver protocol stack

flow control

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

TCP flow control

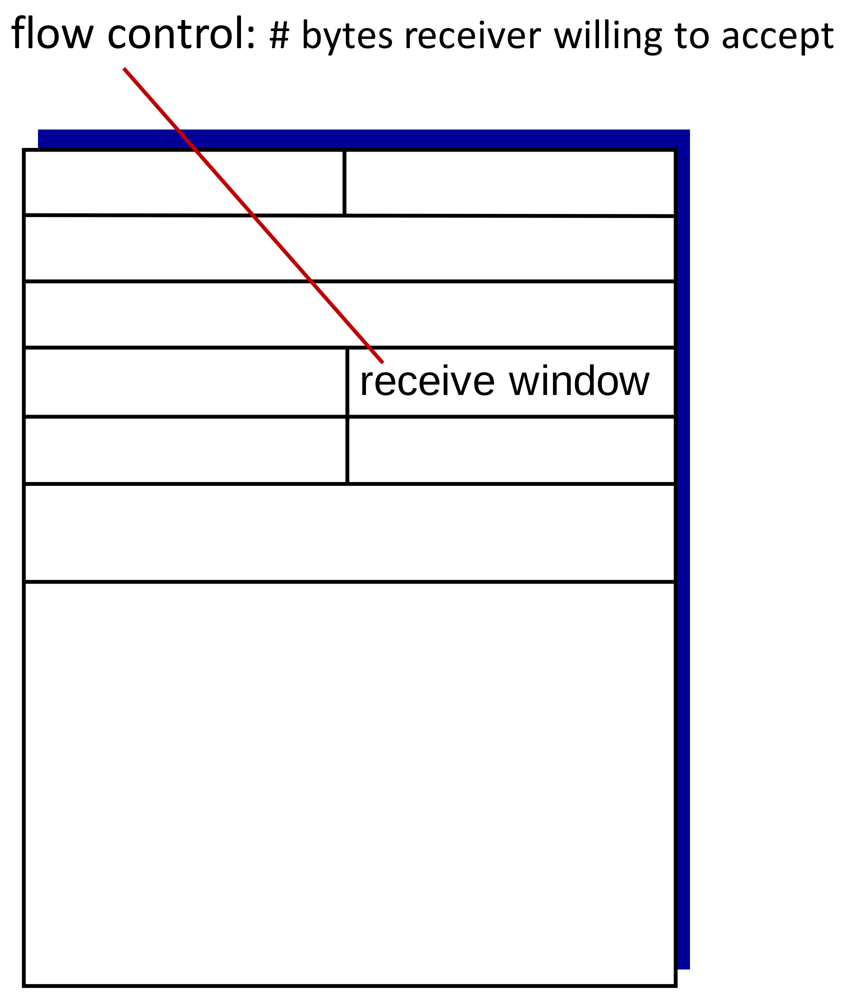
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees 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 sequence numbers, ACKs

Sequence numbers:

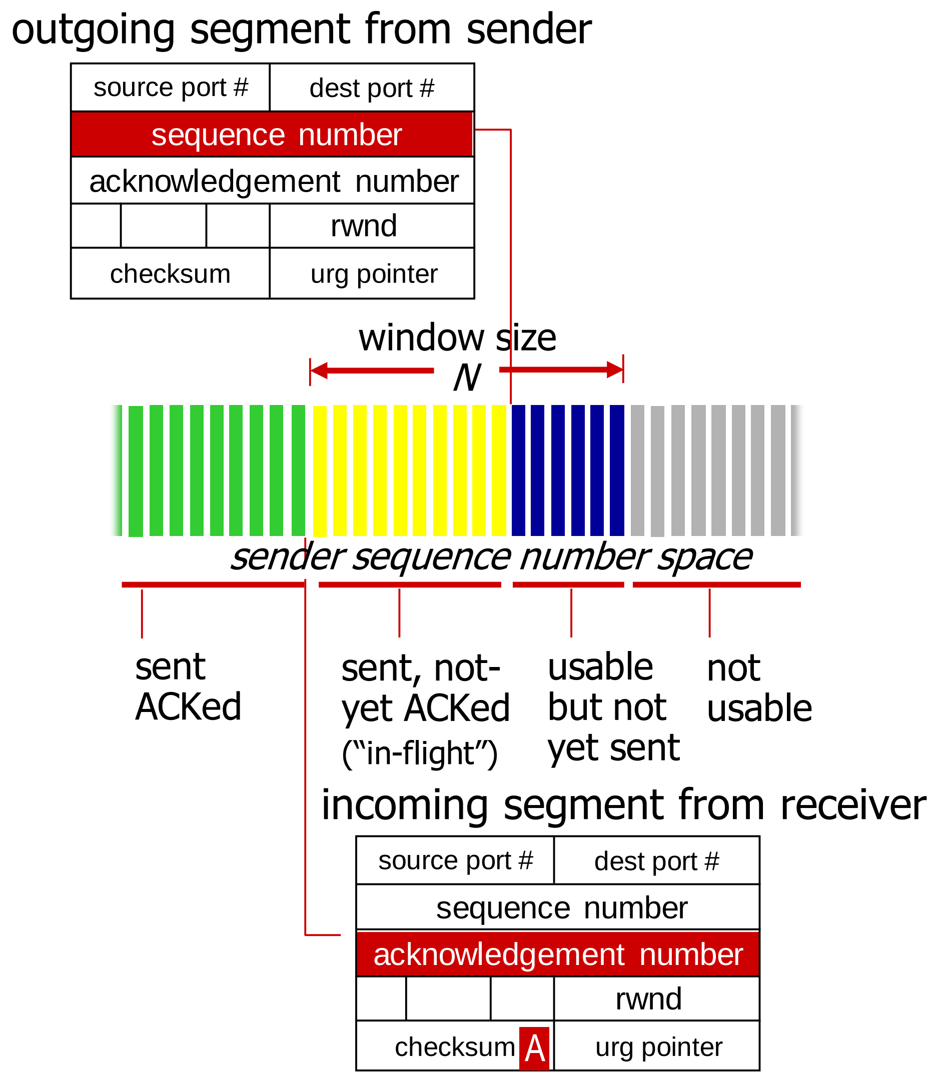
- byte stream "number" of first byte in segment's data

Acknowledgements:

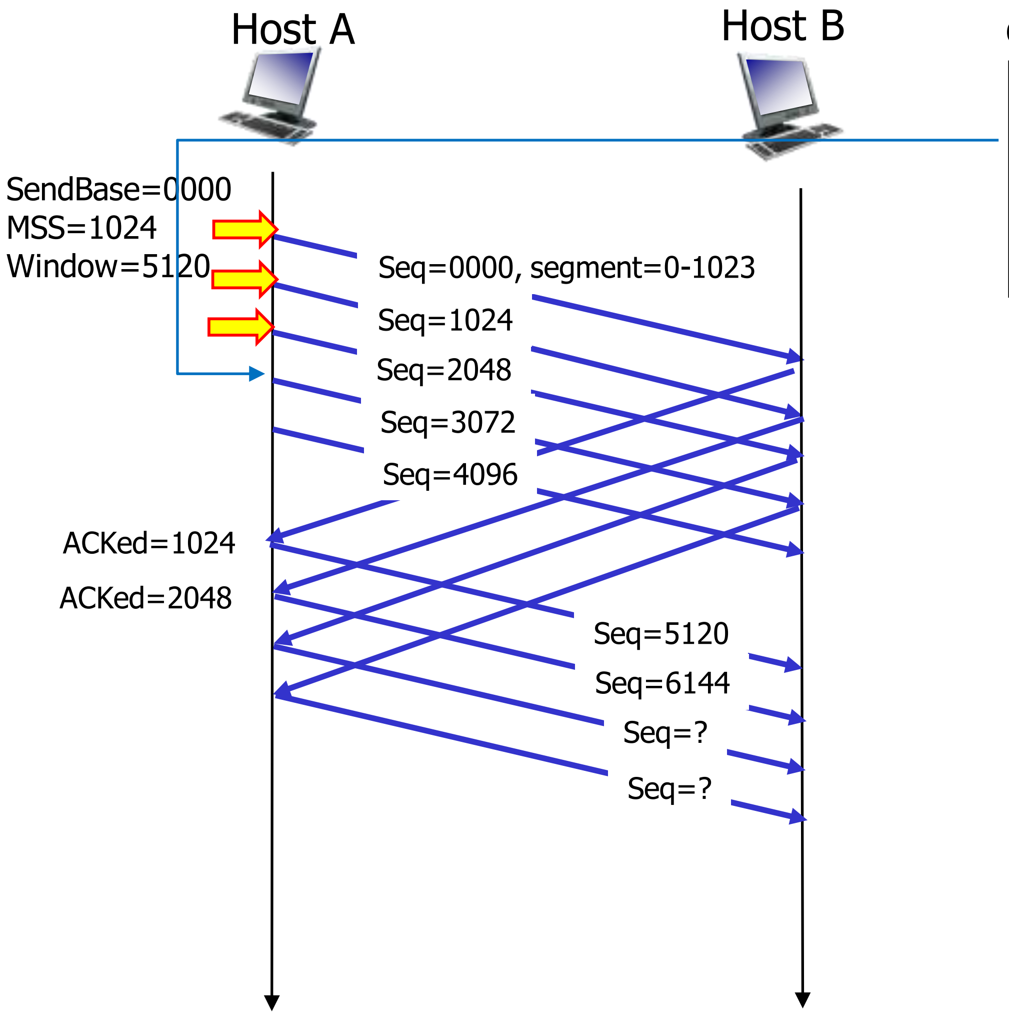
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

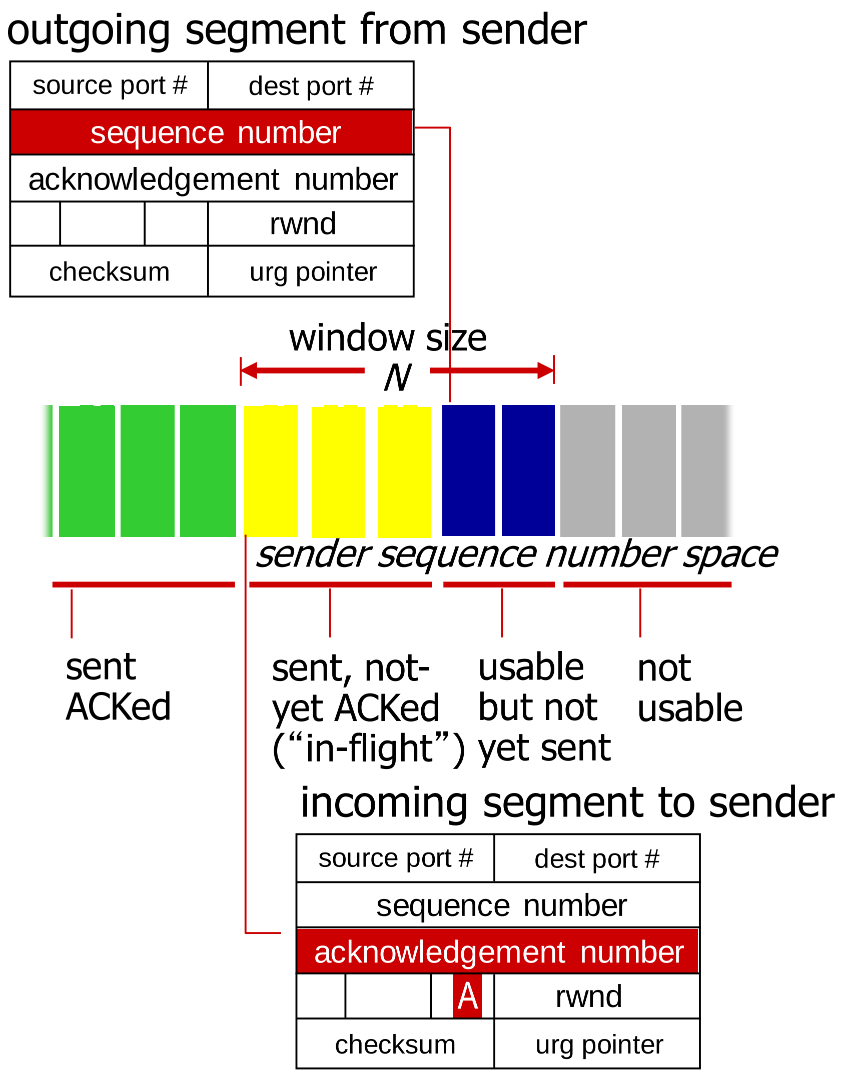
- *A:* TCP spec doesn't say, - up to implementor



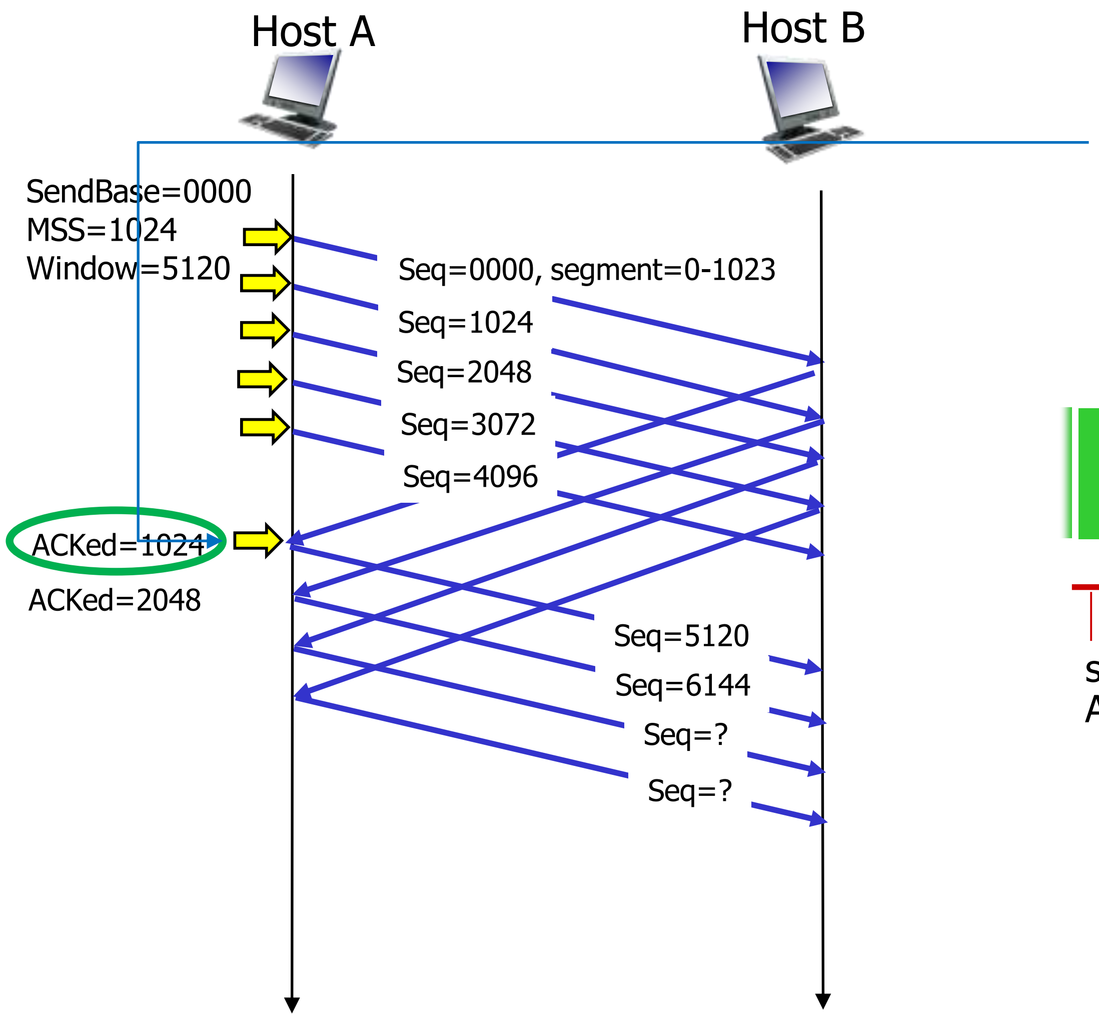
TCP window flow control



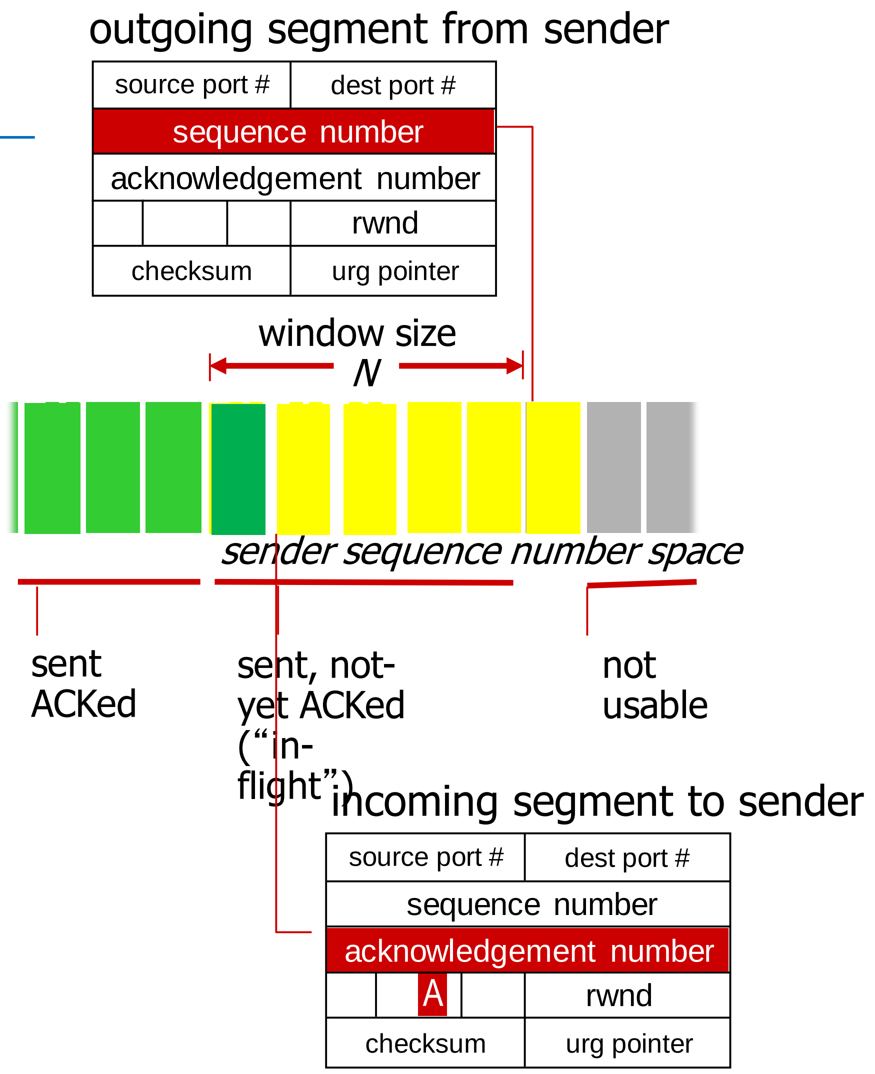
fast retransmit after sender receipt of triple duplicate ACK



TCP window flow control



fast retransmit after sender receipt of triple duplicate ACK



Mid-break

☕□

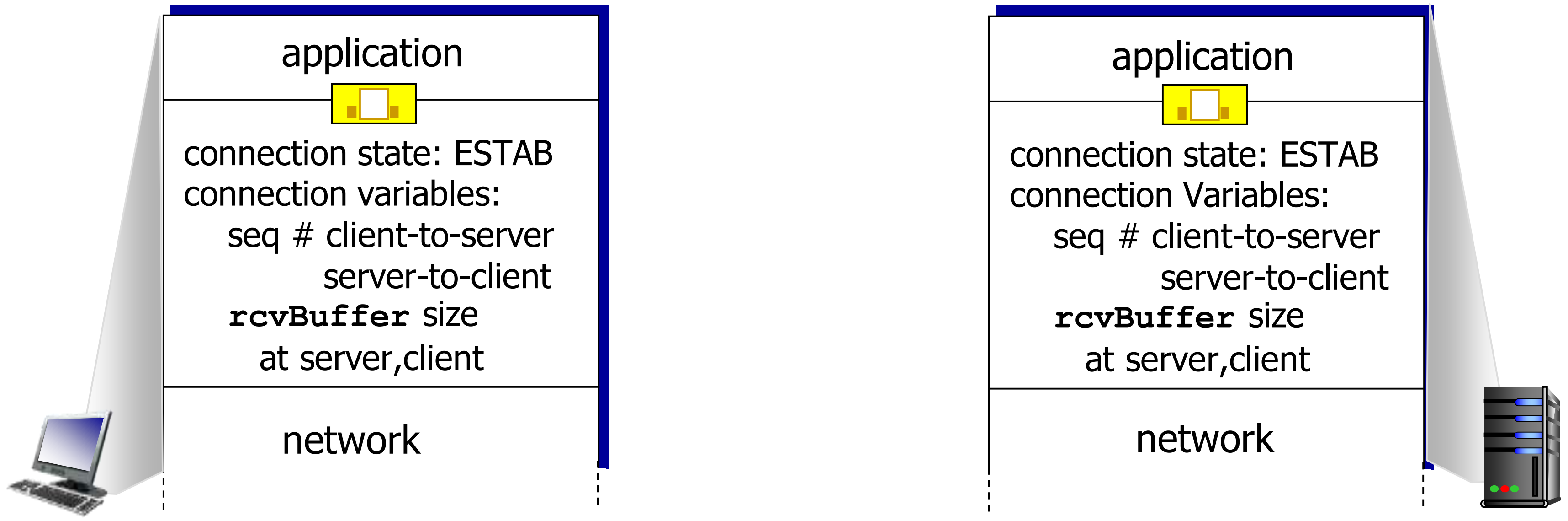
- Q & A

Transport layer: roadmap

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)

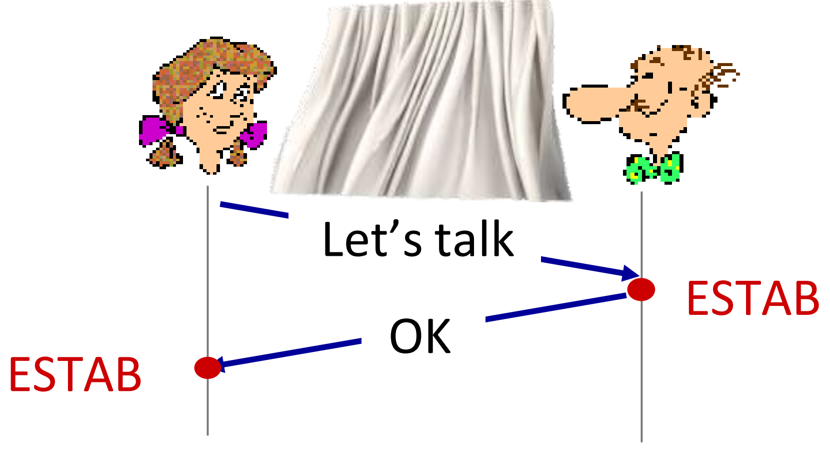


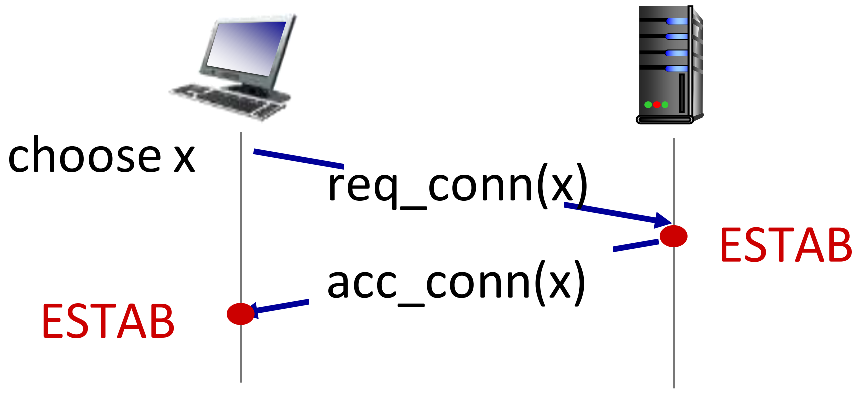
```
clientSocket.connect("server","port");
...
clientSocket.send(…)
```

```
connSocket,addr = serverSocket.accept();
...
connSocket.recv(…)
```

Agreeing to establish a connection

2-way handshake:

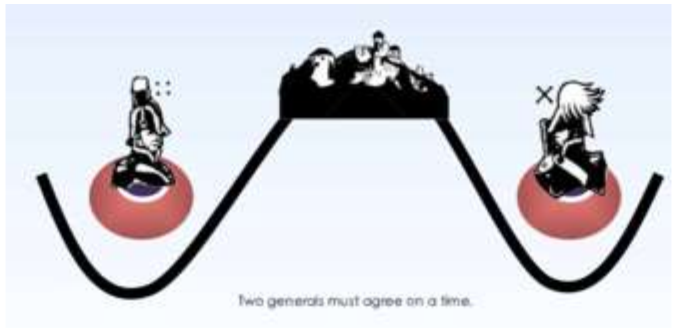




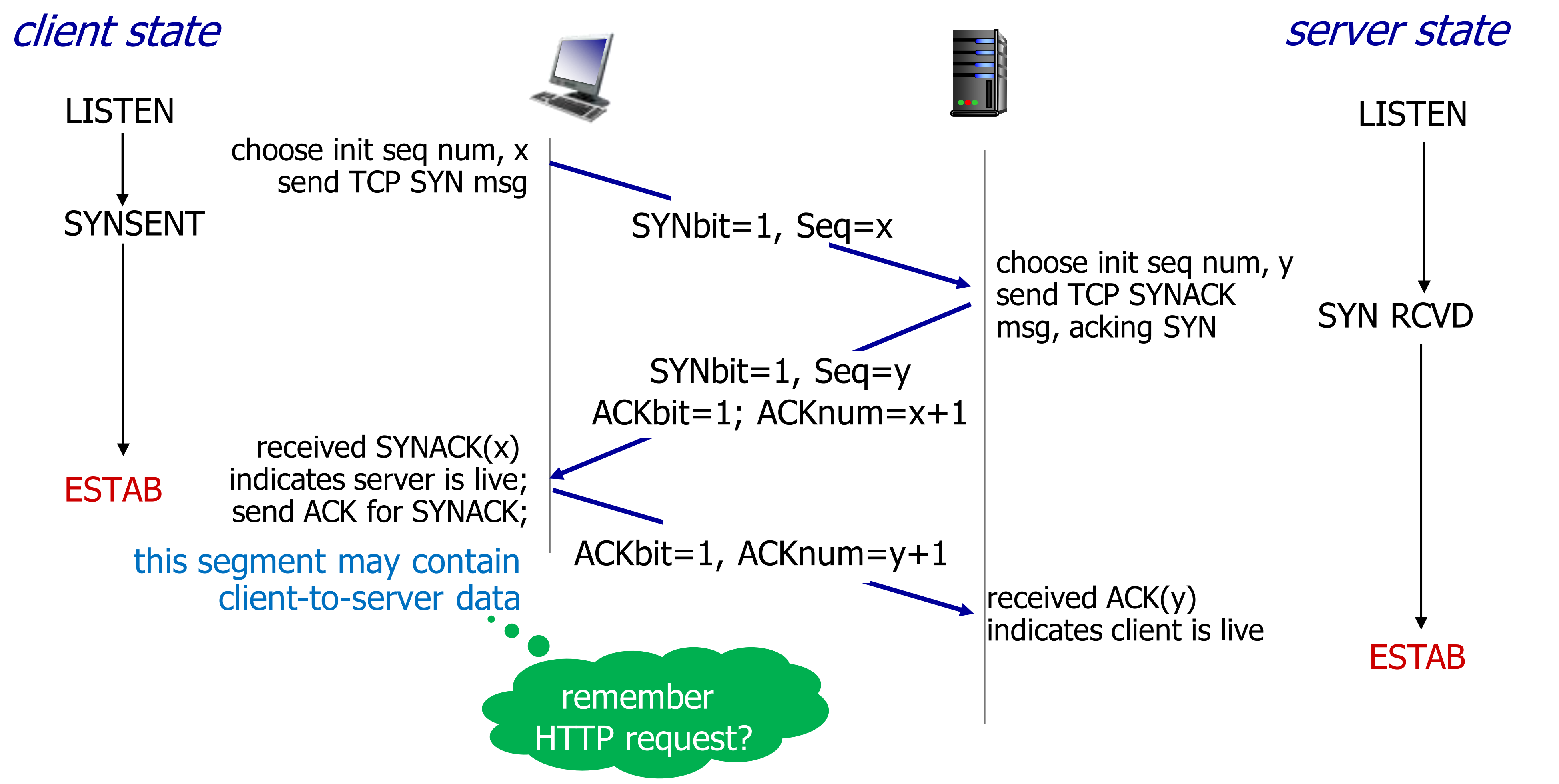
Q: will 2-way handshake always work in network?

- variable delays
- unreliable channel,
- retransmitted messages due to message delay, loss

Byzantine Generals Problem



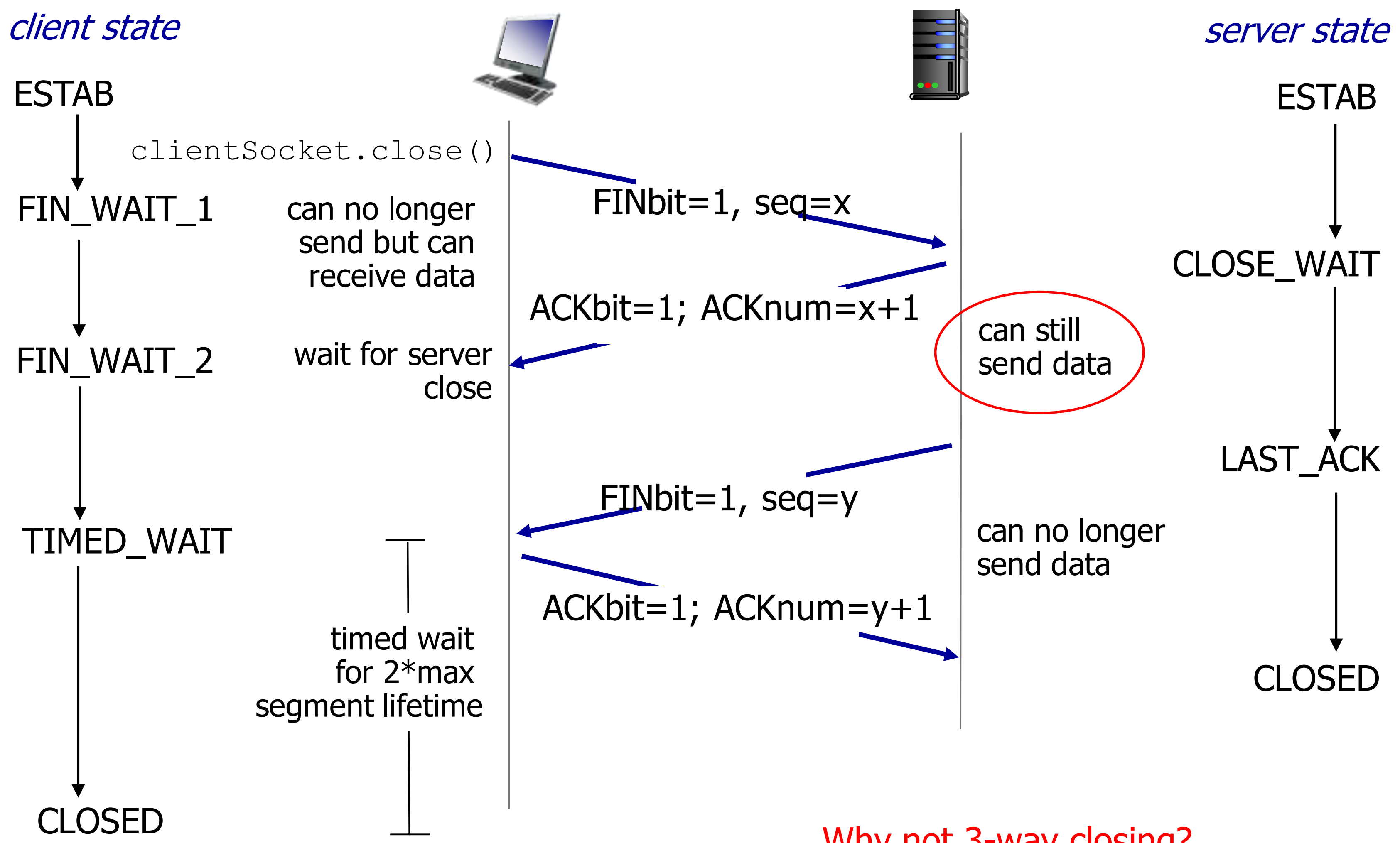
TCP 3-way handshake



Closing a TCP connection

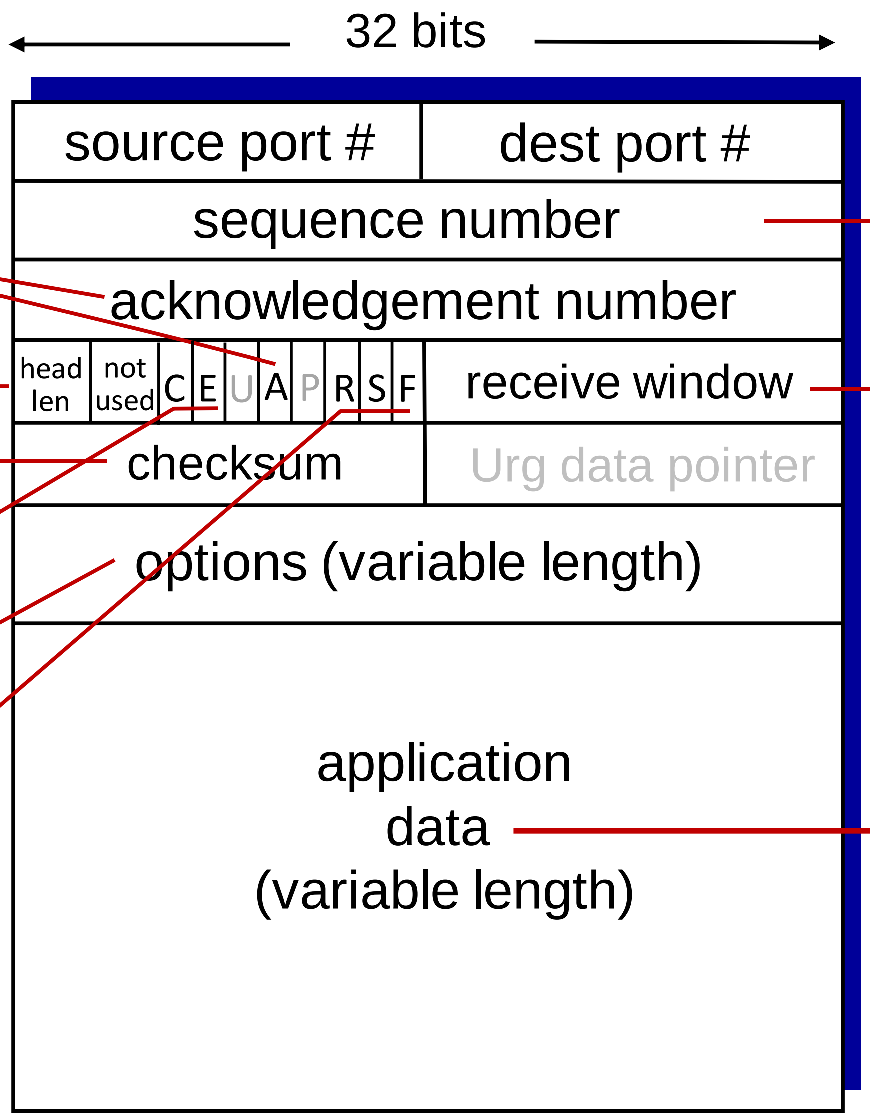
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

Closing a TCP connection



Why not 3-way closing?

TCP segment structure - review



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

Transport layer: roadmap

Principles of congestion control

Congestion:

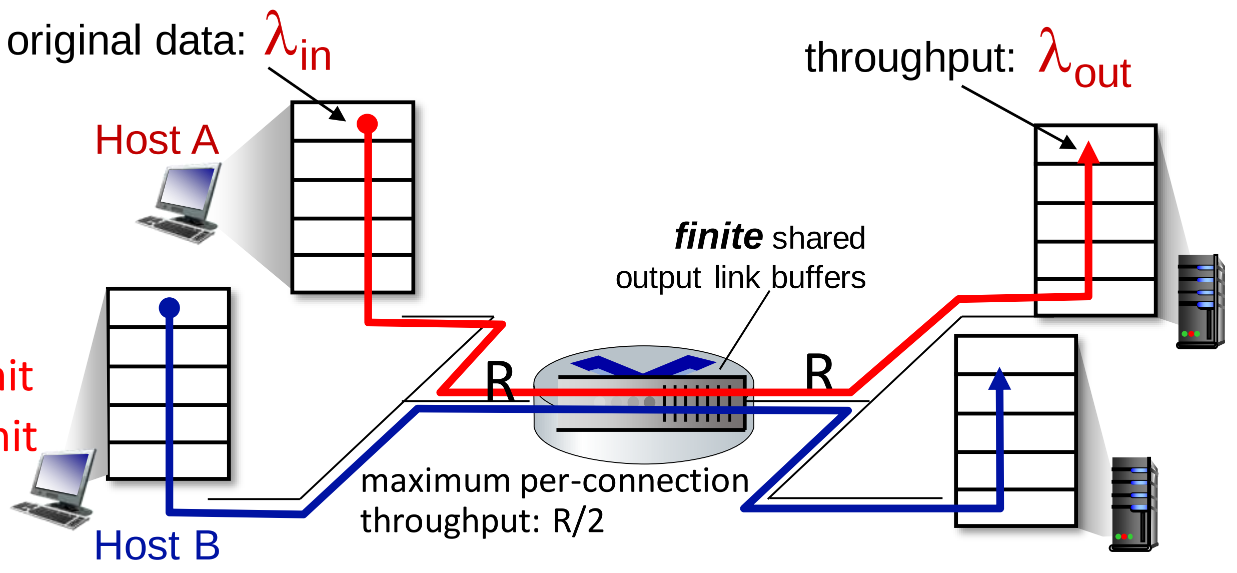
- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control (one-to-one)!
- a top-10 problem!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

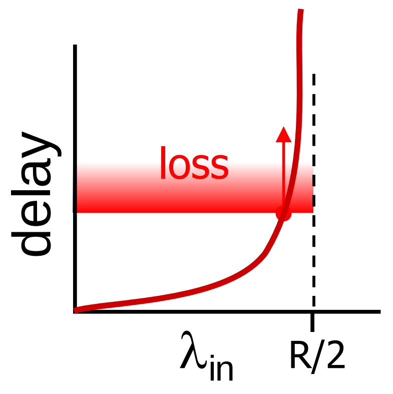
Causes/costs of congestion: scenario 1

- two senser/receiver pairs
- one router, two flows,
- link capacity: R
- As traffic increases → R/2
  - ✗ Delay increase - timeout → retransmit
  - ✗ Packet loss → retransmit

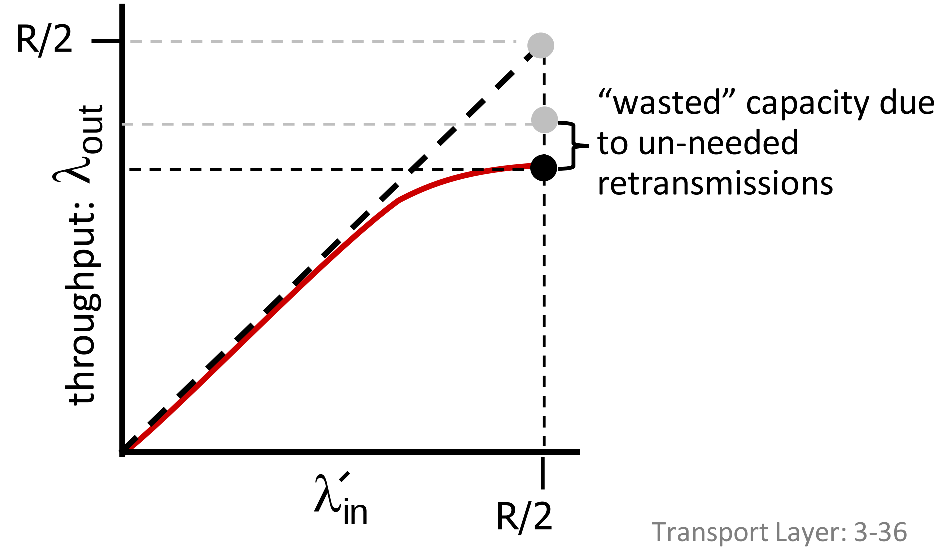


original data: λ_{in}
throughput: λ_{out}
Host A
Host B
finite shared output link buffers
R
R
maximum per-connection throughput: R/2

- How can we?
  - ✓ Avoid congestion
  - ✓ Resolve congestion

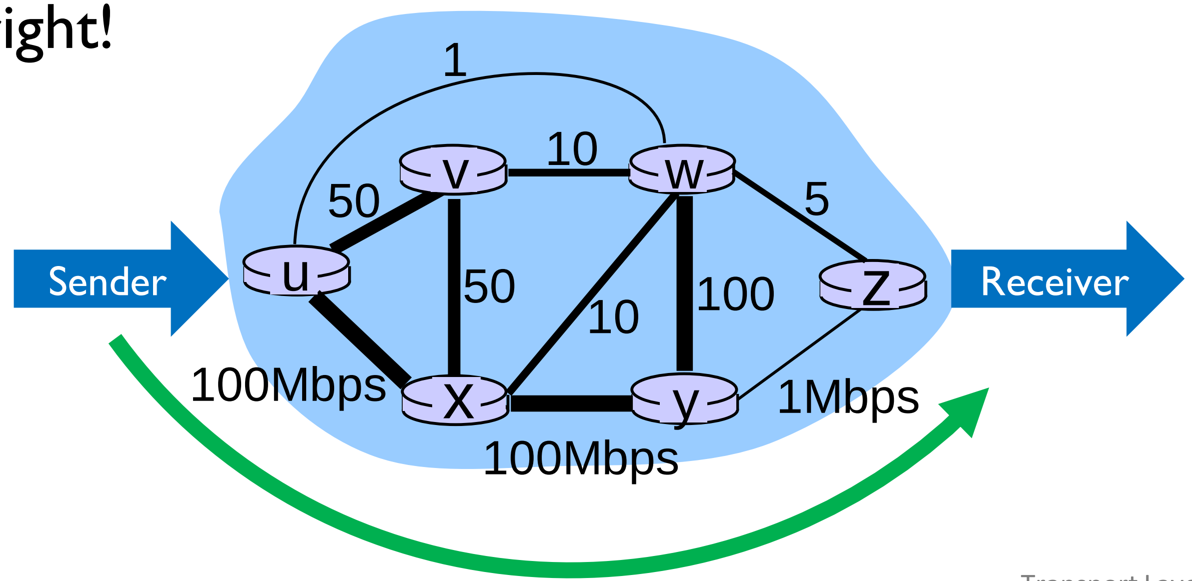


✗ Delay increase
✗ Packet loss



Congestion: multi-hop scenario

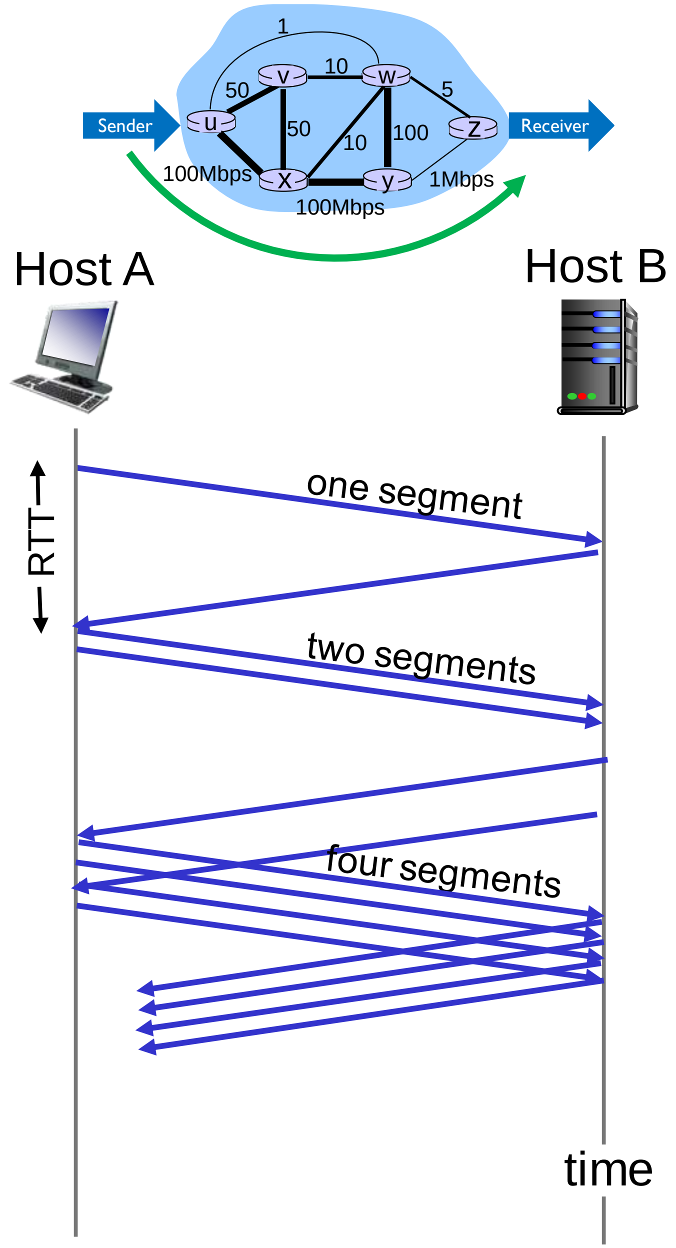
- Sender → Receiver
  - Download100MB file, using FTP/TCP
  - Via multiple hops of various BW (unknown!)
- Question:
  - Decide sending data rate? (wnd/RTT)
  - not too low, not too high – just right!
- Strategy:
  - Start from low rate
  - Slowly ramp up
  - Stabilise at allowed



Transport layer: roadmap

TCP slow start

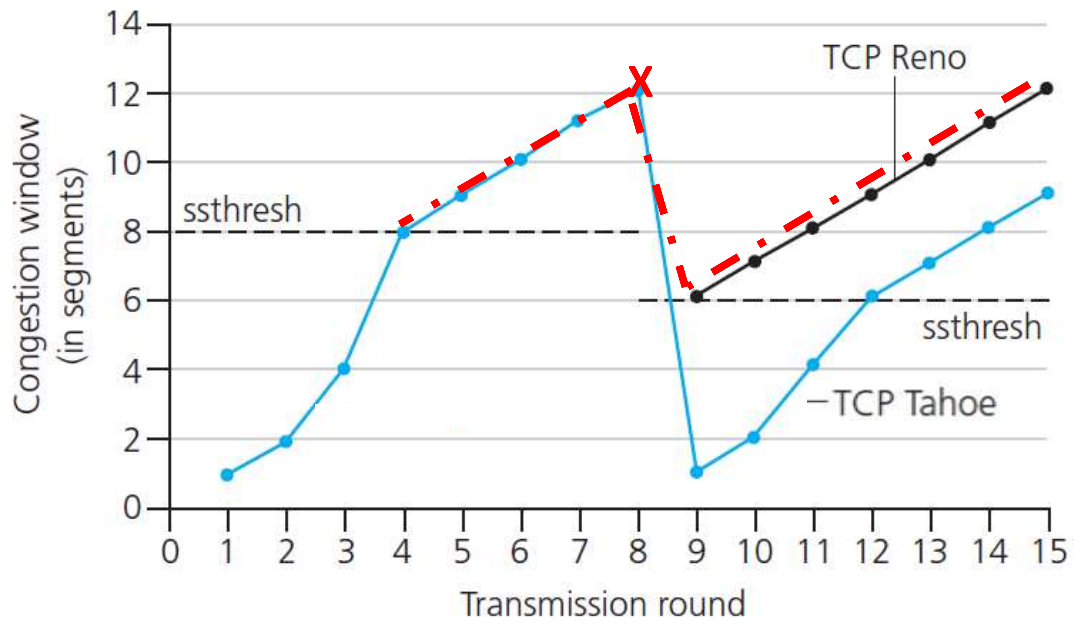
- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **cwnd** ≥ **ssthresh**



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/

TCP congestion control: AIMD

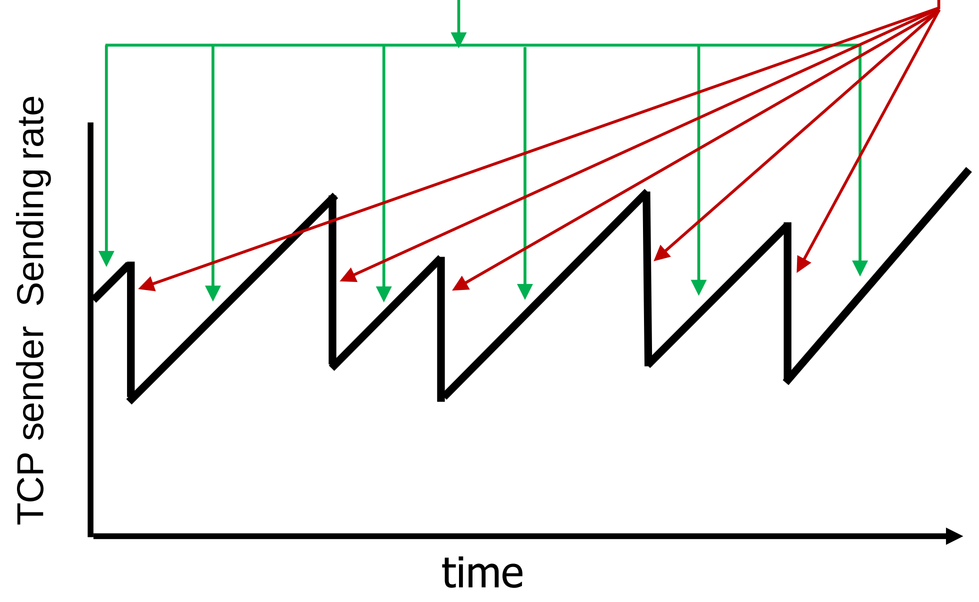
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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

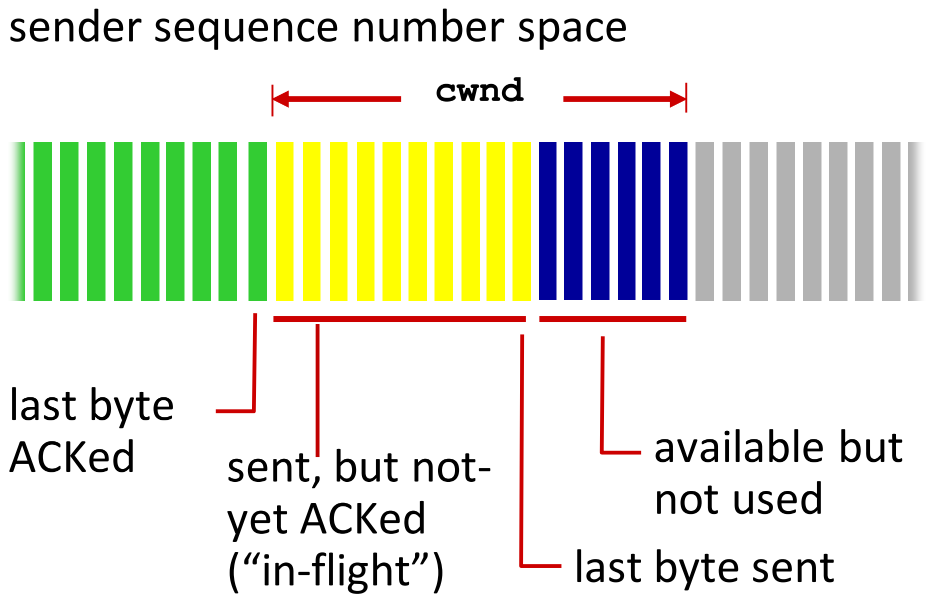
Multiplicative decrease detail: sending rate is

- 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 congestion control: details


sender sequence number space
cwnd
last byte ACKed
sent, but not-yet ACKed ("in-flight")
available but not used
last byte sent

TCP sending behavior:

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

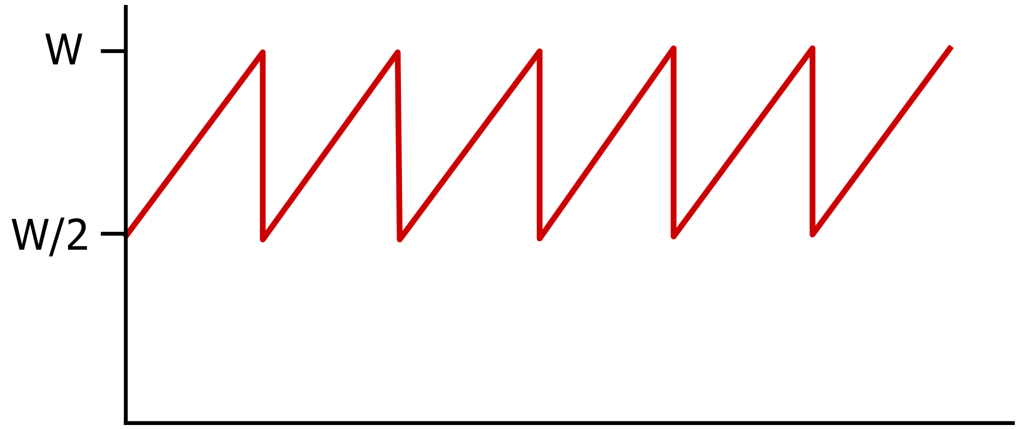
$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

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

TCP throughput

- avg. TCP thruput 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

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



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:

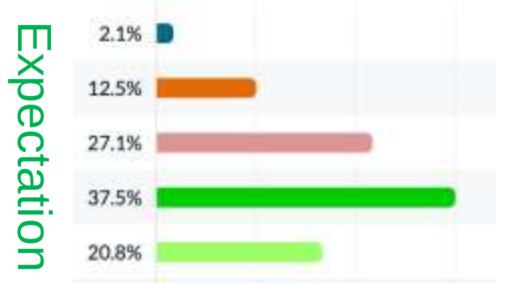
- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

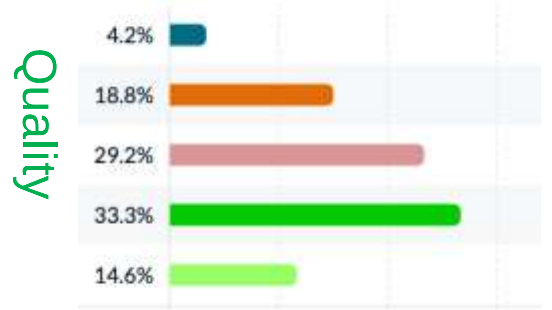
Assignment2 - 5%

- Assignment2 – 5%: Transport Layer
  - Based on weeks 5 & 6 learning materials
  - Answer in required format - **strictly!**
  - Due by Sunday 31st March - No late submission accepted!

Student Feedback Survey (SFS) Results

- Some Positive
  - Detailed explanation – that's the way!
  - Engaging Q&A during lecture – making the most!
  - Tutorial/Lab – generally positive, some not - passed to the tutors
- Some Issues:
  - Content heavy – reducing load, keeping fundamental
  - overwhelmed by IT concepts – recursive teaching in CS





Lecture done ✓

- Q & A