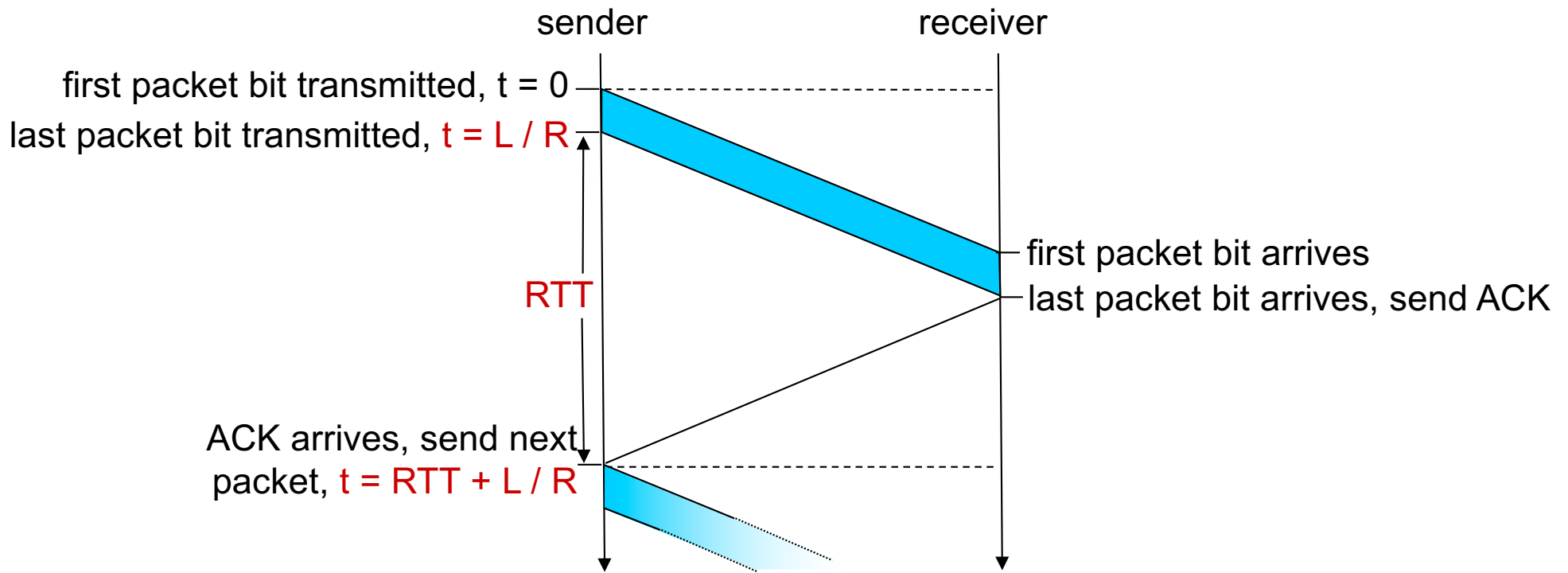# Logistics

* HW3 (Reading/questions) extended
  * My fault: various deadlines & personal events
  * HW3 Due Sun 2/26, 10pm, online after class

* UDP Programming assignment online
  * Due 3/2

* Today:
  * Finish transport layer

Review: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R

RTT

first packet bit arrives
last packet bit arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R
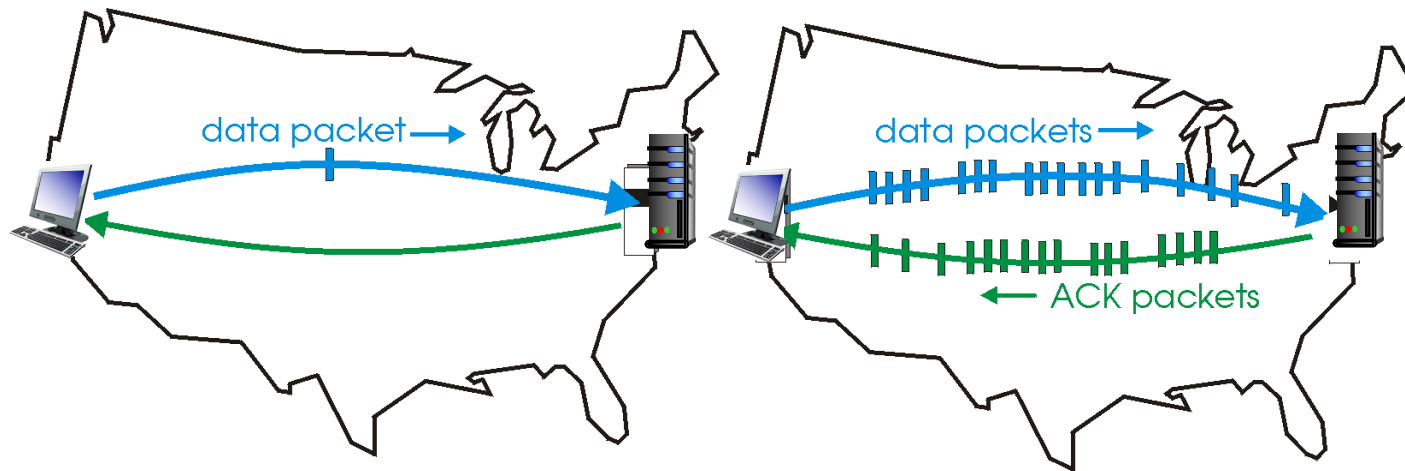
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Review: Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

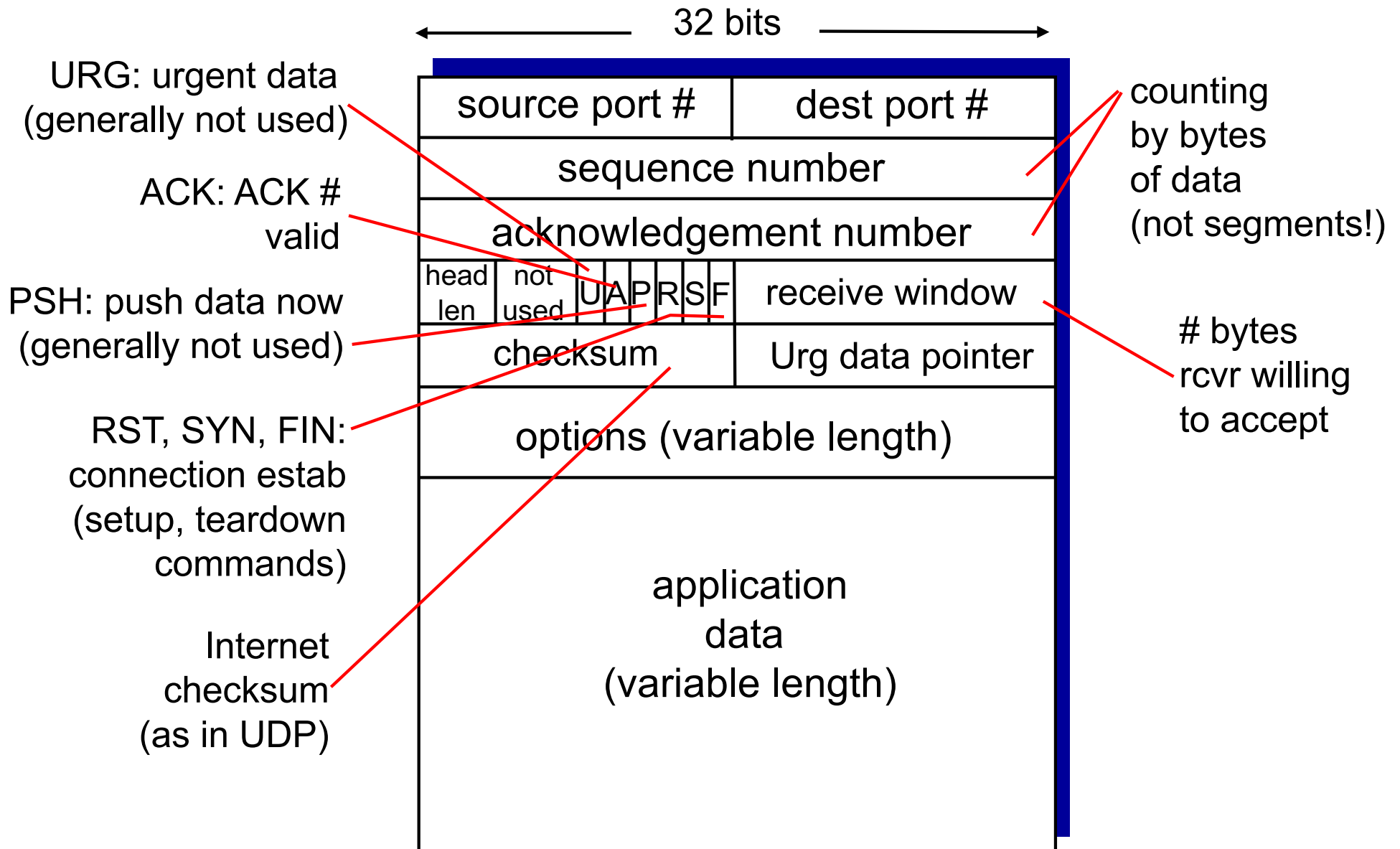(a) a stop-and-wait protocol in operation   (b) a pipelined protocol in operation

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

- ❖ **point-to-point:**
  - ▪ one sender, one receiver
- ❖ **reliable, in-order** *byte steam:*
  - ▪ no "message boundaries"
- ❖ **pipelined:**
  - ▪ TCP congestion and flow control set window size

- ❖ **full duplex data:**
  - ▪ bi-directional data flow in same connection
  - ▪ MSS: maximum segment size
- ❖ **connection-oriented:**
  - ▪ handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- ❖ **flow controlled:**
  - ▪ sender will not overwhelm receiver

Review: TCP header

32 bits

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)

source port #	dest port #
sequence number	
acknowledgement number	

head len	not used	U	A	P	R	S	F	receive window
checksum								Urg data pointer

options (variable length)

application
data
(variable length)

counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept

Review: seq. numbers

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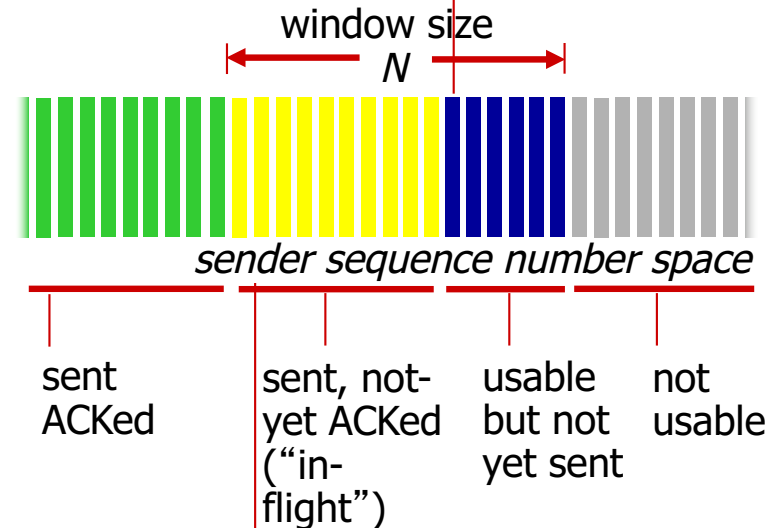- byte stream "number" of first byte in segment's data

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

outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
		rwnd
checksum	urg pointer

window size
N

sender sequence number space

sent ACKed

sent, not-yet ACKed ("in-flight")

usable but not yet sent

not usable

incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	A	rwnd
checksum	urg pointer

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

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

- ❖ exponential weighted moving average
- ❖ influence of past sample decreases exponentially fast
- ❖ typical value: α = 0.125

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

◆ sampleRTT

■ EstimatedRTT

time (seconds)
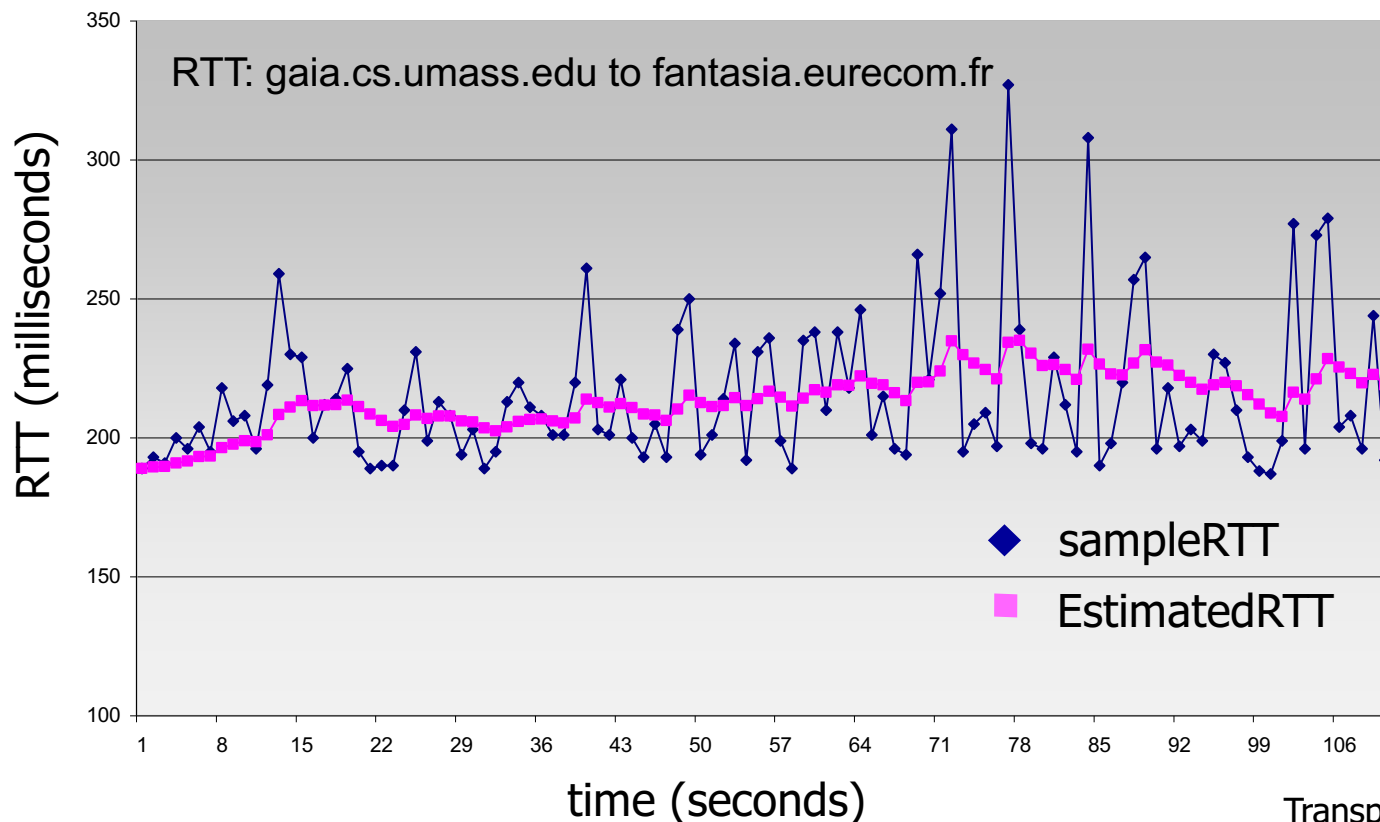
TCP round trip time, timeout

❖ 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

let's initially consider simplified TCP sender:
* ignore duplicate acks
* ignore flow control, congestion control

TCP sender events:

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
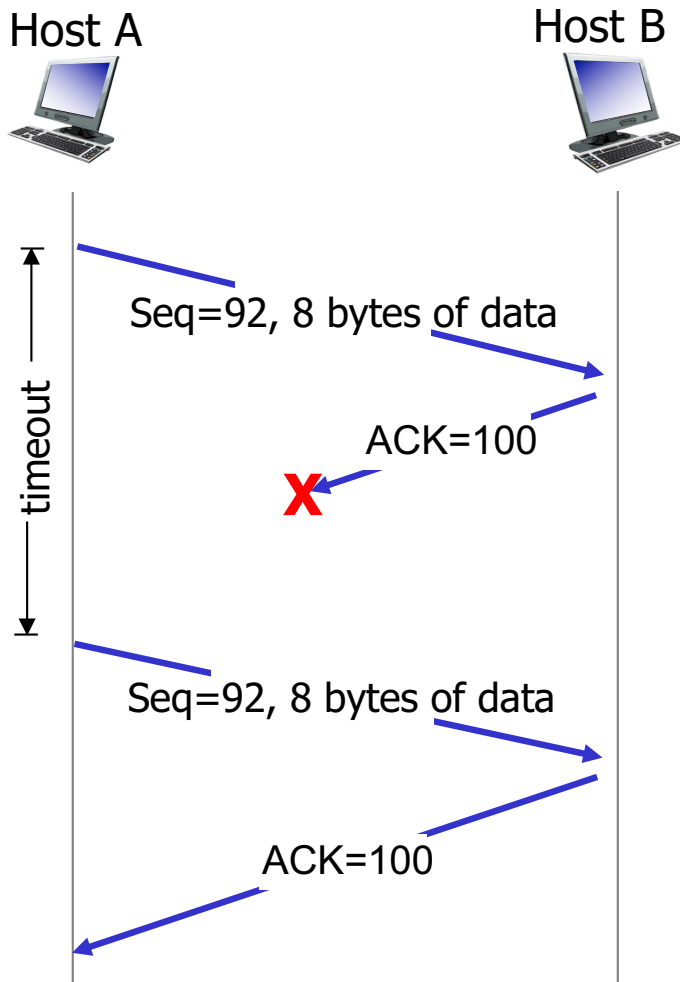
lost ACK scenario
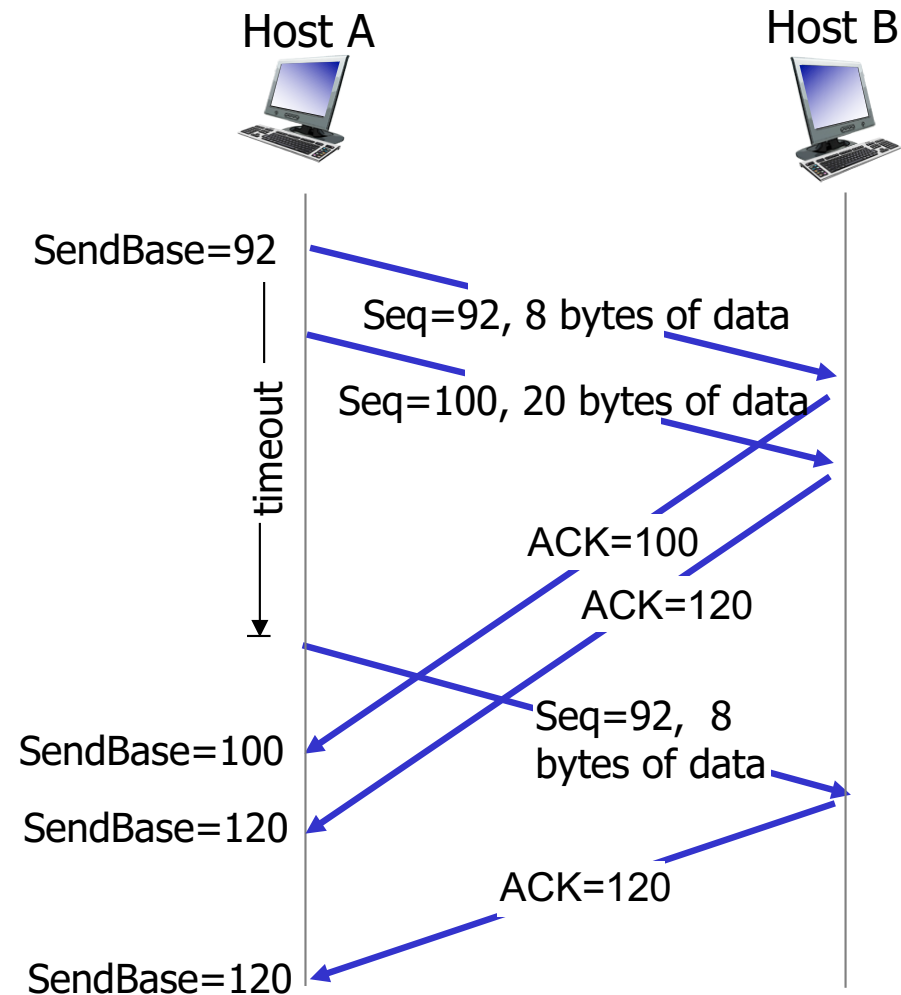
premature timeout

TCP: retransmission scenarios

Host A   Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

Seq=120, 15 bytes of data
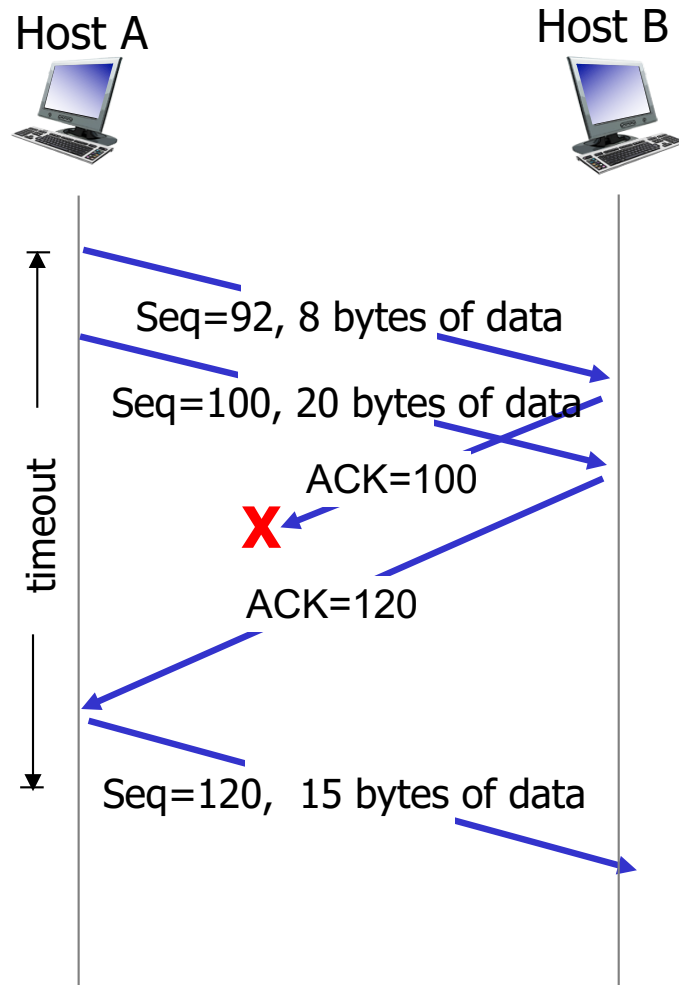
timeout

cumulative ACK

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
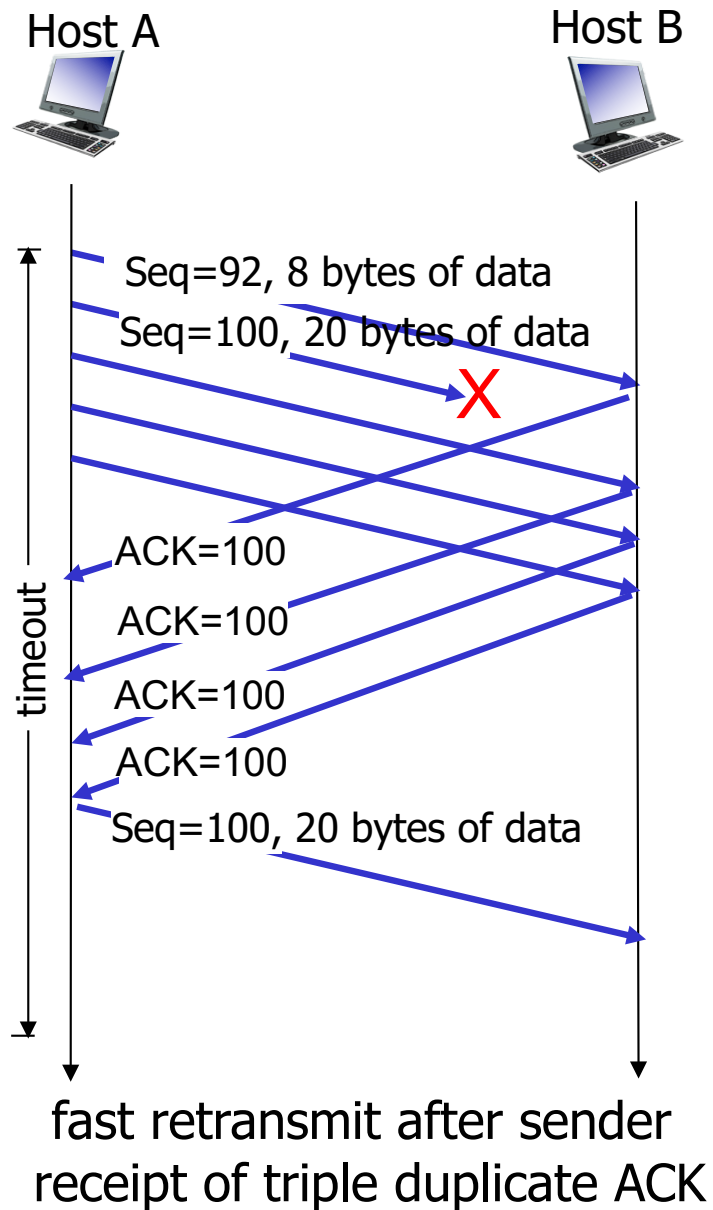
Modem Configuration

28.8 Data Fax Modem (COM1)

Initial speed (bps): **57600**

Hardware Features

- ☑ Enable hardware flow control
- ☑ Enable modem error control
- ☐ Enable modem compression

☐ Disable modem speaker

[OK]   [Cancel]

TCP flow control

application may remove data from TCP socket buffers ....

application process

TCP socket receiver buffers

application
- - - - - - -
OS

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

TCP code

IP code

flow control

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

from sender

receiver protocol stack
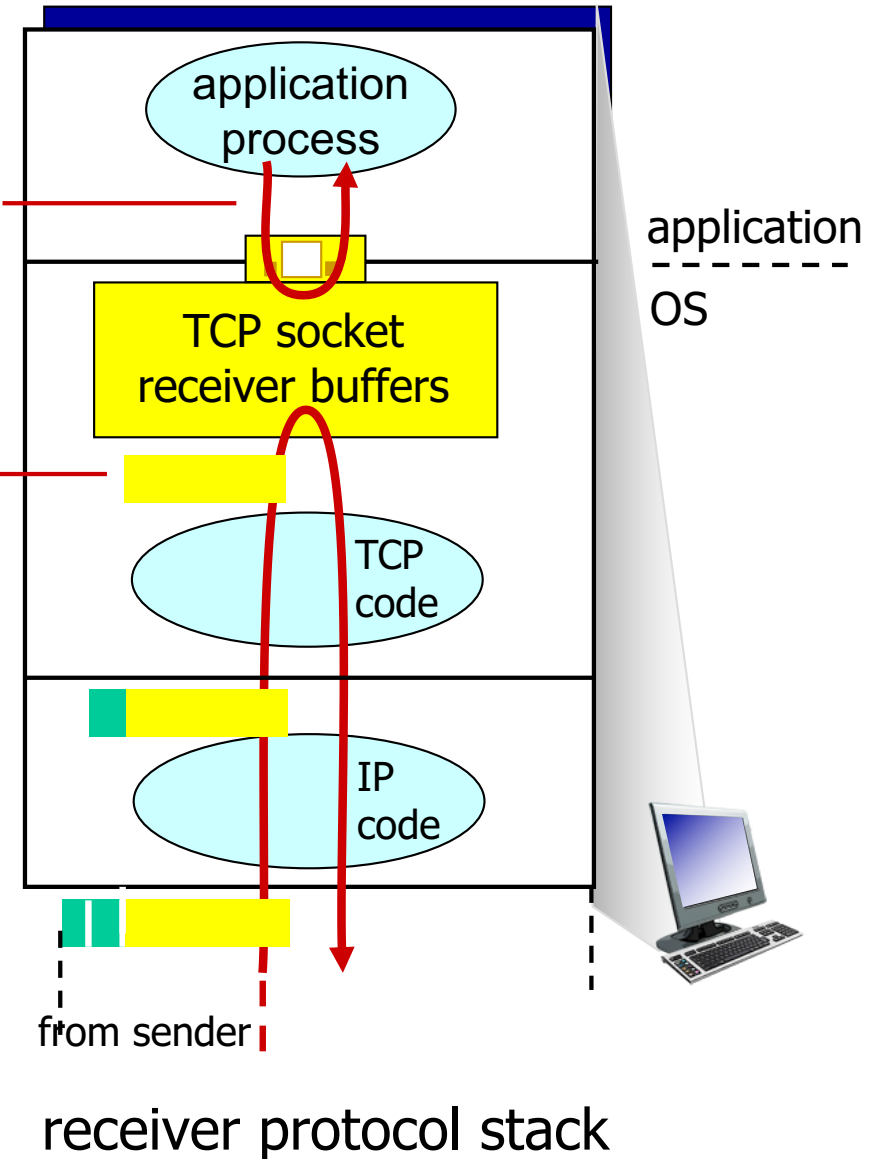
TCP flow control

❖ 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

to application process

RcvBuffer	buffered data
rwnd	free buffer space

TCP segment payloads
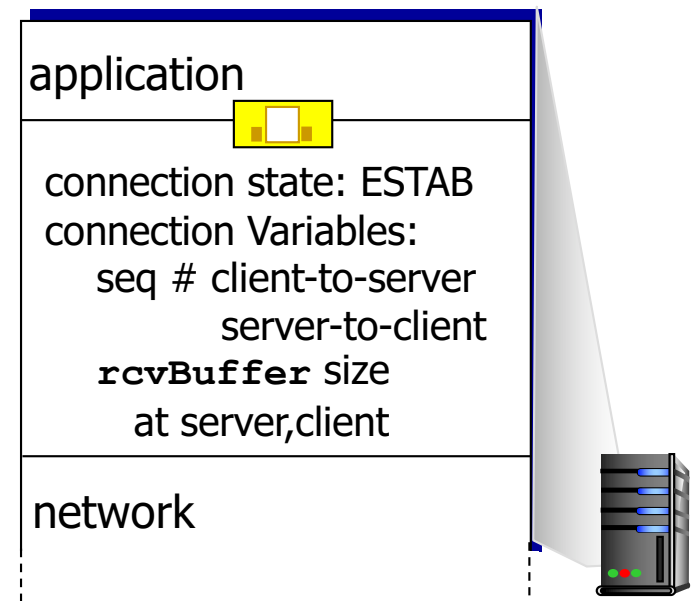
receiver-side buffering

Chapter 3 outline

Connection Management

before exchanging data, sender/receiver "handshake":

❖ agree to establish connection (each knowing the other willing to establish connection)

❖ agree on connection parameters

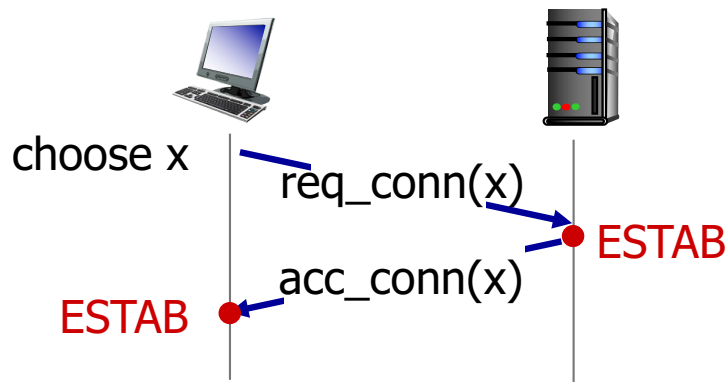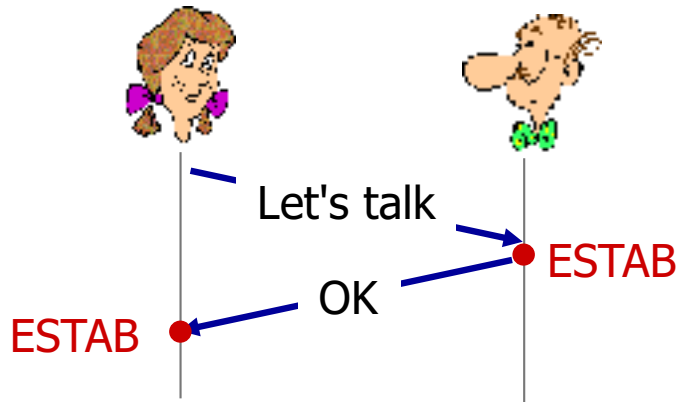application

connection state: ESTAB
connection variables:
    seq # client-to-server
        server-to-client
    rcvBuffer size
    at server,client

network

application

connection state: ESTAB
connection Variables:
    seq # client-to-server
        server-to-client
    rcvBuffer size
    at server,client

network

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

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

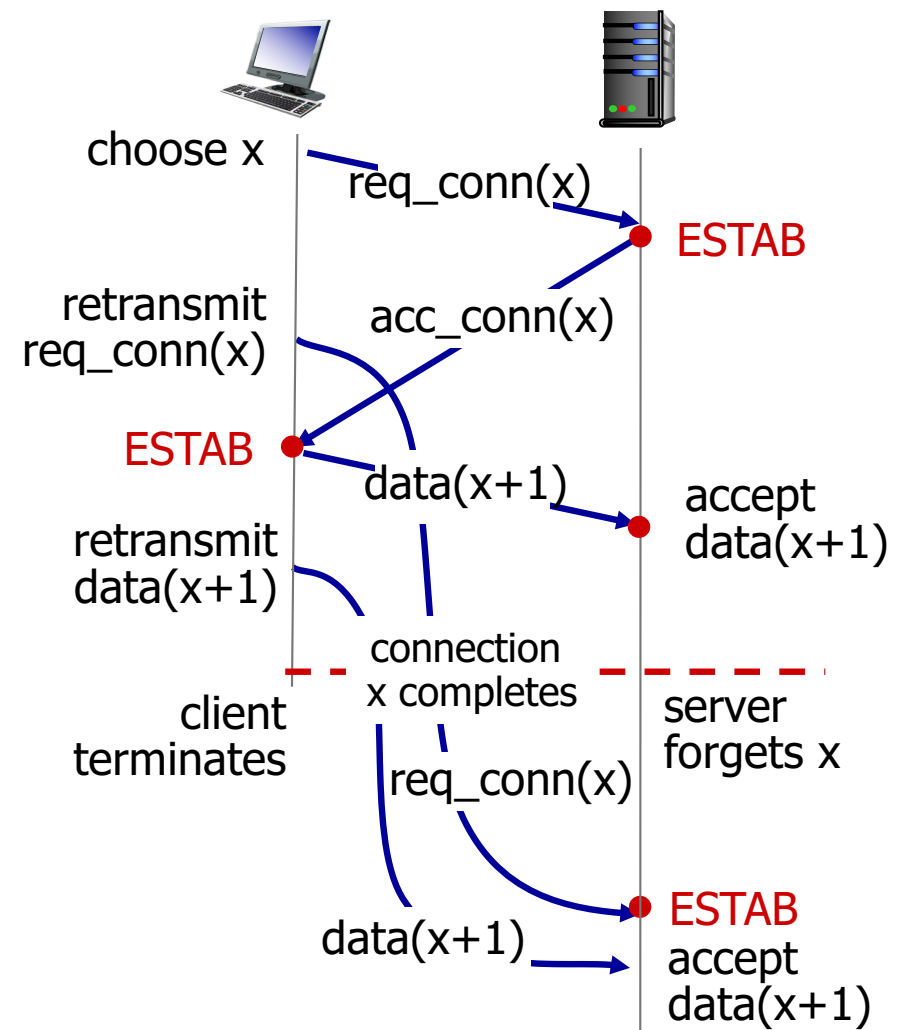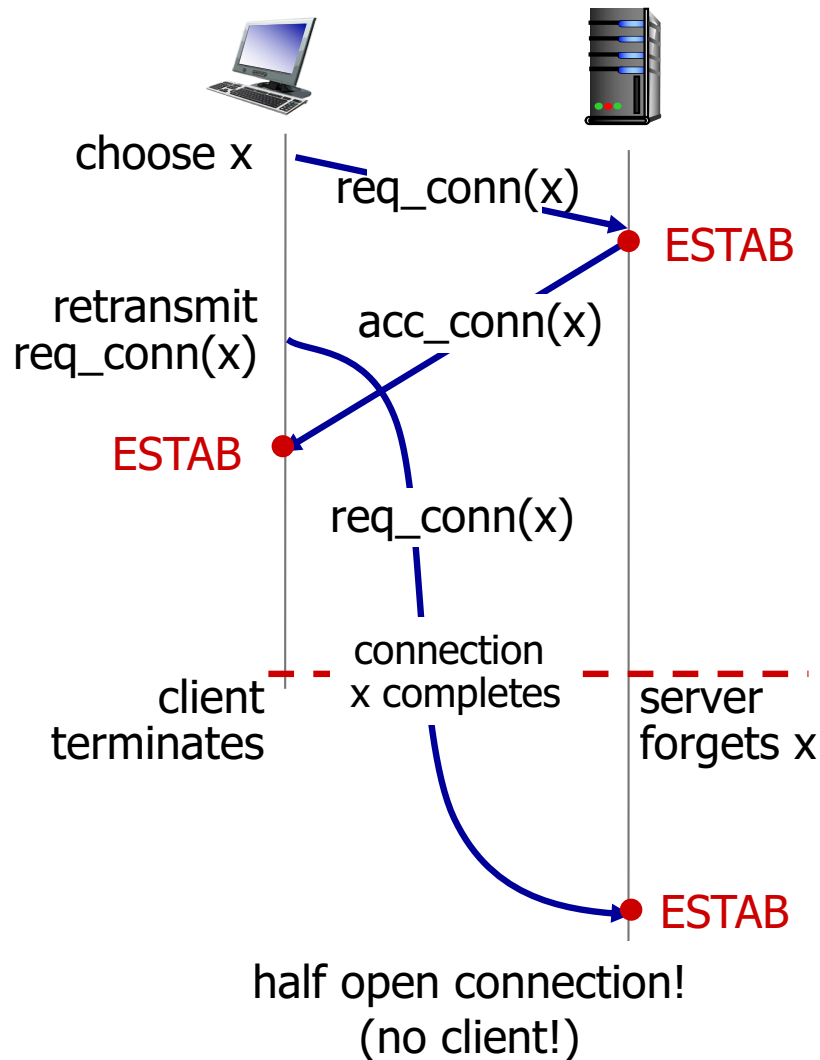# Agreeing to establish a connection

2-way handshake:
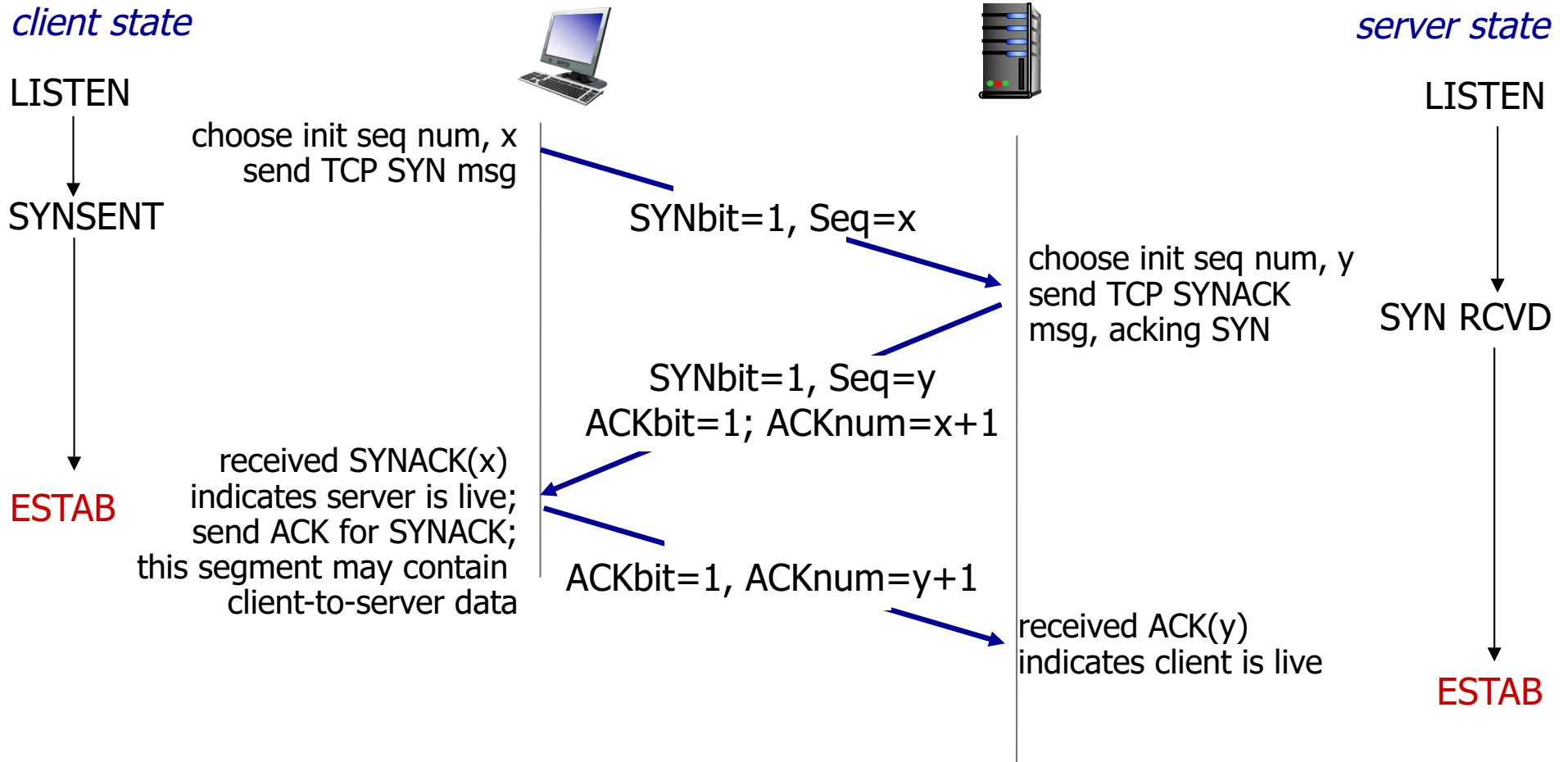

Q: will 2-way handshake always work in network?

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

Agreeing to establish a connection

2-way handshake failure scenarios:

TCP 3-way handshake

client state

LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

ESTAB

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

server state

LISTEN

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

received ACK(y)
indicates client is live

ESTAB

TCP: closing a 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

TCP: closing a connection

client state

ESTAB

clientSocket.close()

FIN_WAIT_1   can no longer
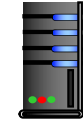   send but can
              receive data

FIN_WAIT_2   wait for server
              close

TIMED_WAIT

              timed wait
              for 2*max
              segment lifetime

CLOSED

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

server state

ESTAB

CLOSE_WAIT

can still
send data

LAST_ACK

can no longer
send data

CLOSED

Principles of congestion control
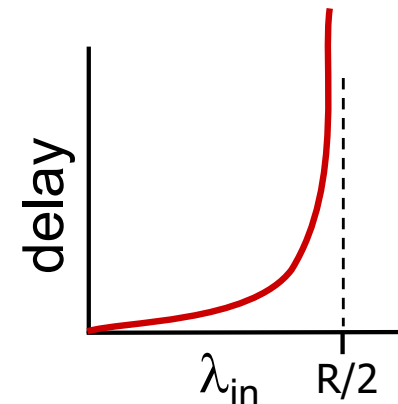
congestion:

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

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission

original data: λ_{in}

throughput: λ_{out}
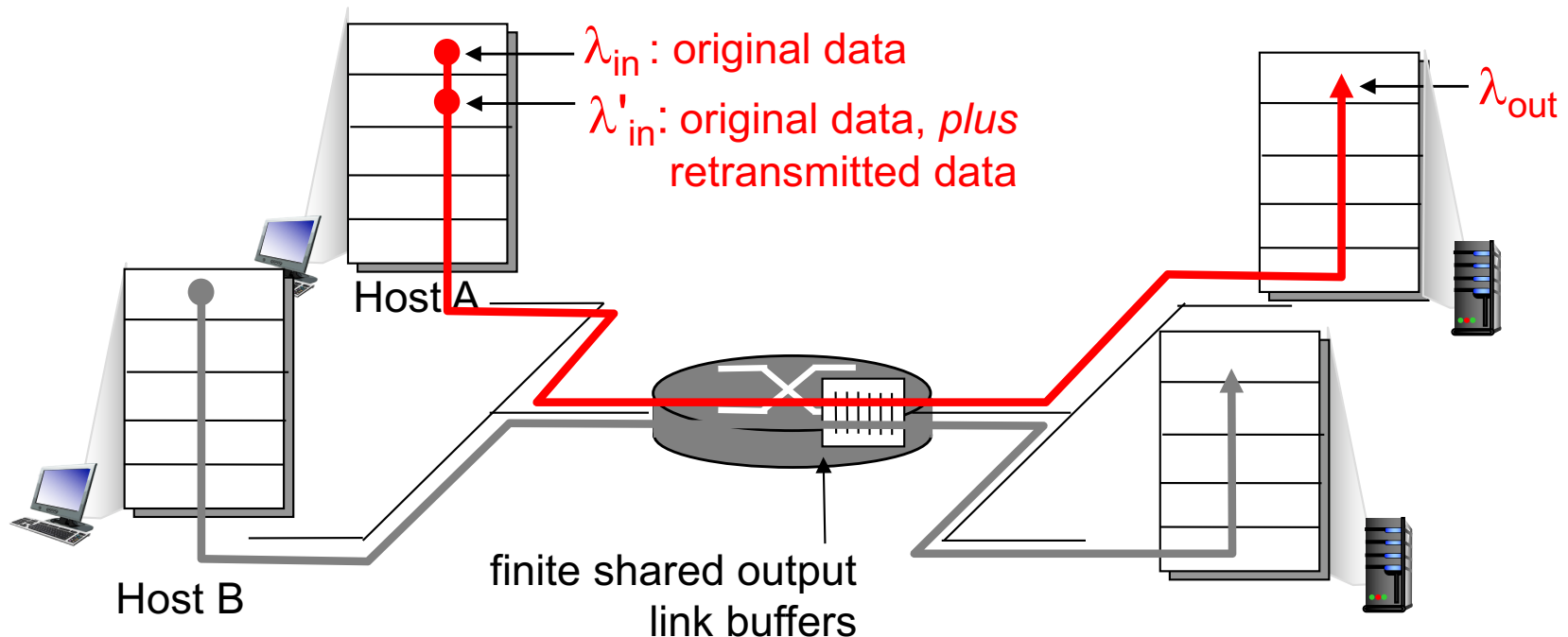
Host A

Host B

unlimited shared output link buffers
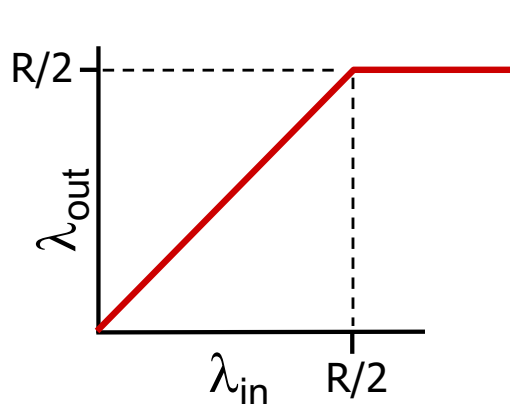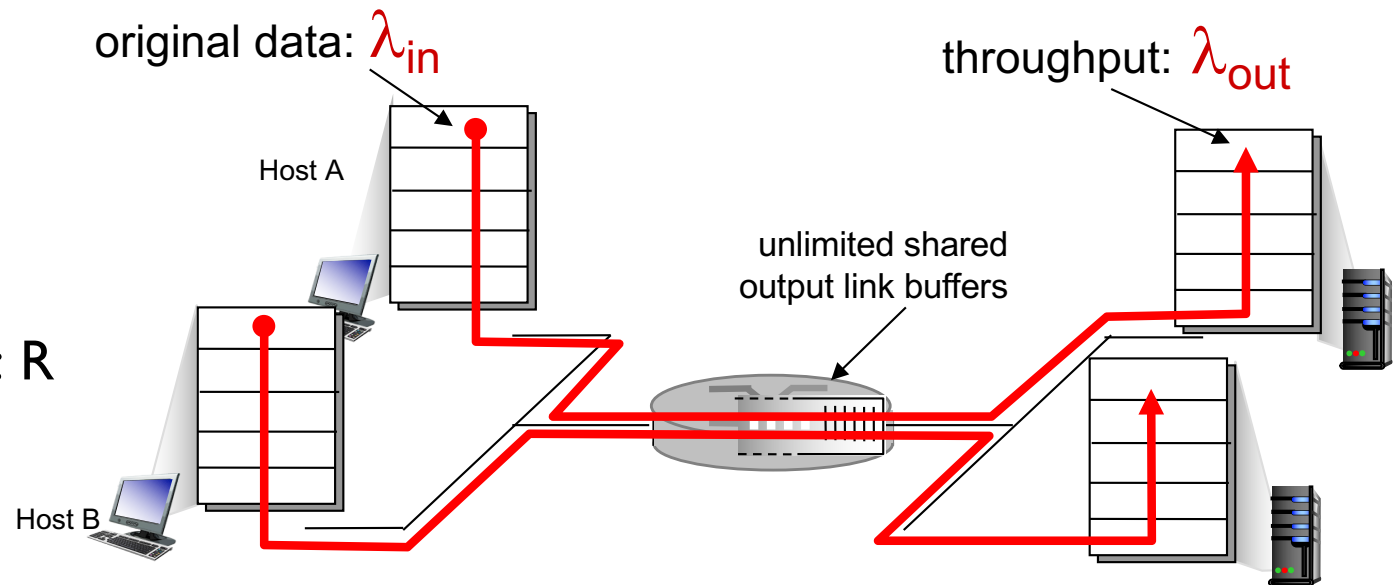
- maximum per-connection throughput: R/2

- large delays as arrival rate, λ_{in}, approaches capacity
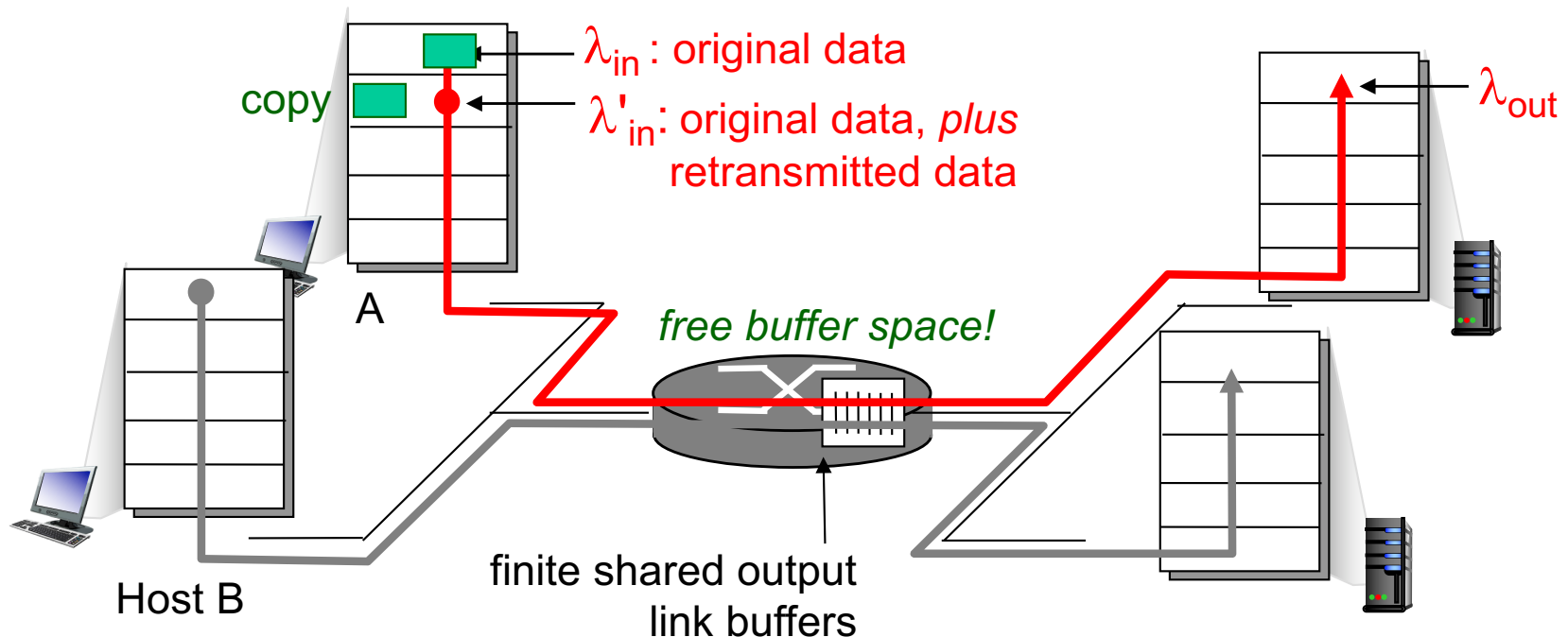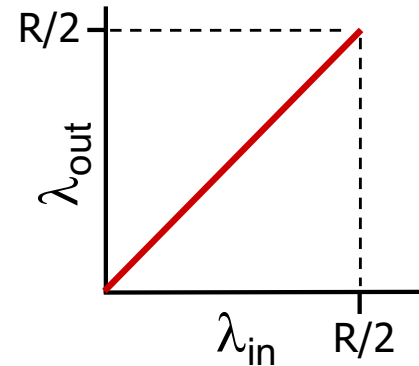
Causes/costs of congestion: scenario 2

❖ one router, *finite* buffers
❖ sender retransmission of timed-out packet
  ▪ application-layer input = app-layer output: $\lambda_{in} = \lambda_{out}$
  ▪ transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

λ_{in} : original data

λ'_{in}: original data, *plus* retransmitted data

λ_{out}

Host A

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

idealization: perfect knowledge

❖ sender sends only when router buffers available

λ_{in} : original data

λ'_{in} : original data, *plus* retransmitted data

λ_{out}

copy

A

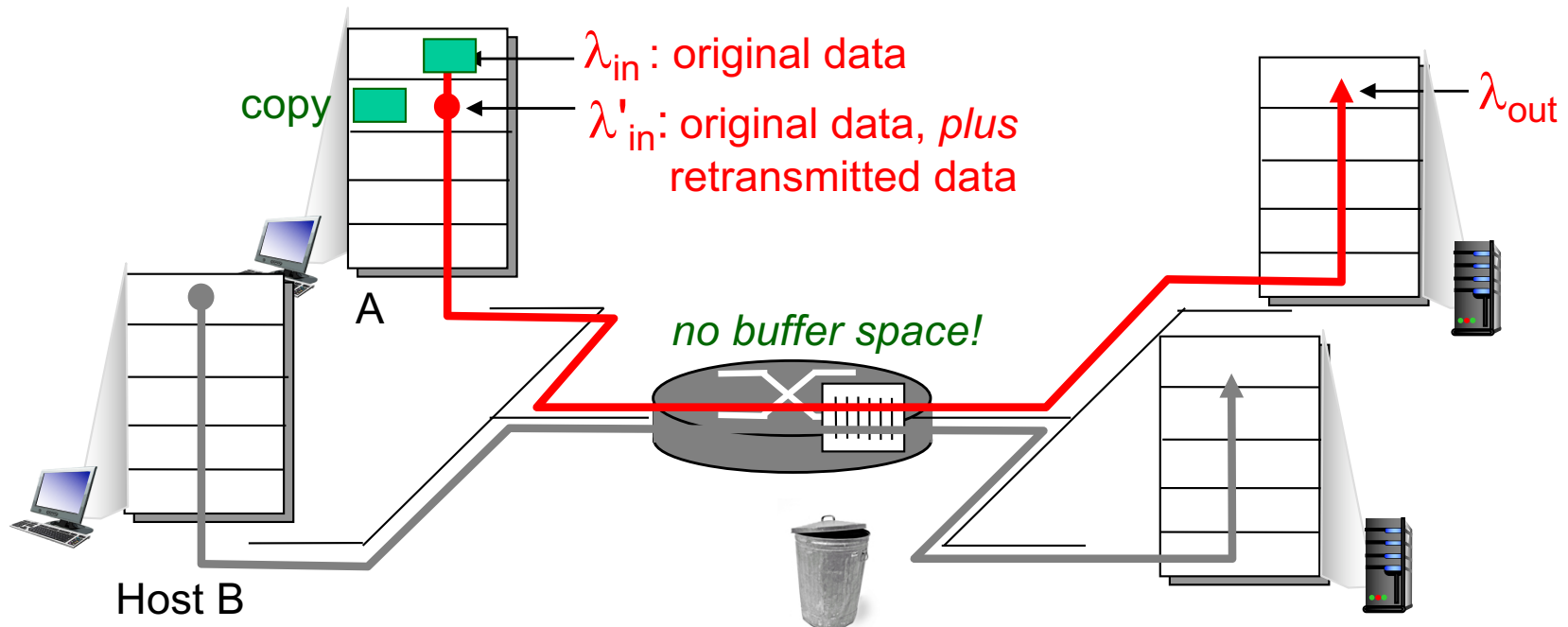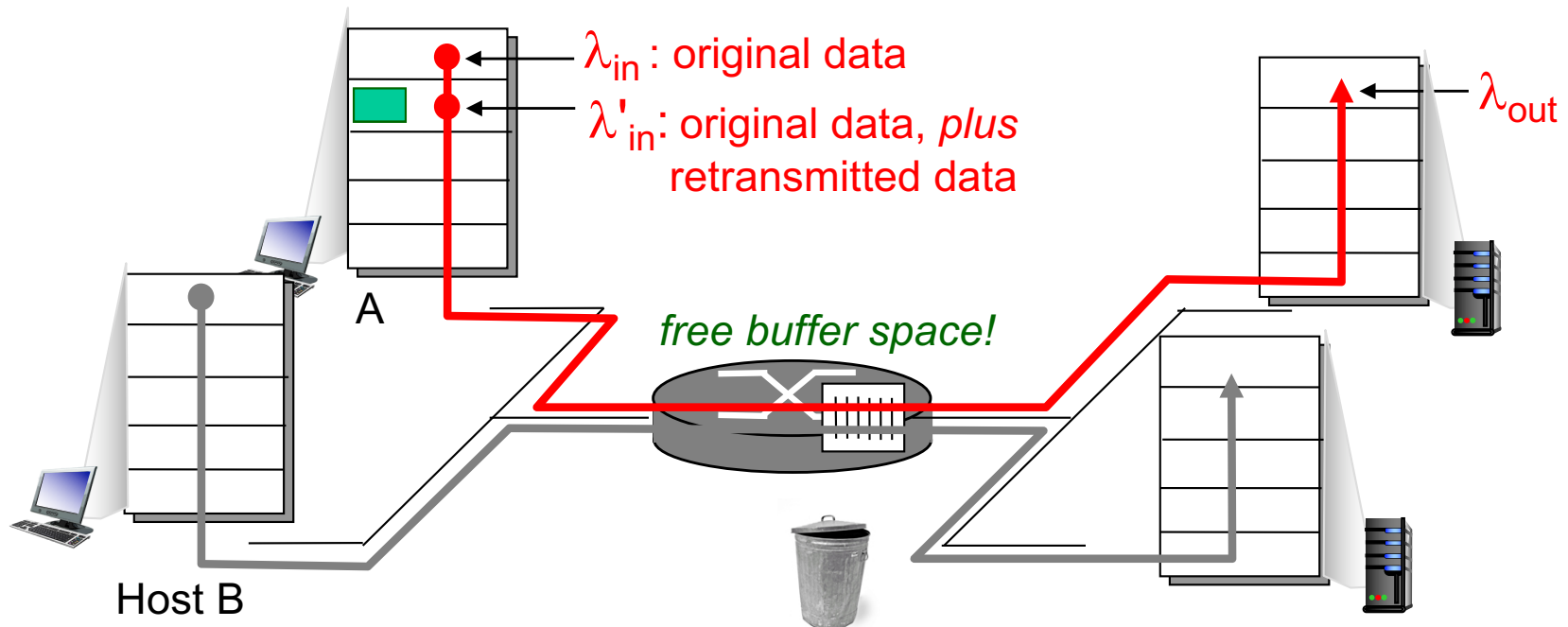free buffer space!

Host B

finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost,
dropped at router due
to full buffers

❖ sender only resends if
packet *known* to be lost

λ_{in} : original data

λ'_{in}: original data, *plus*
retransmitted data
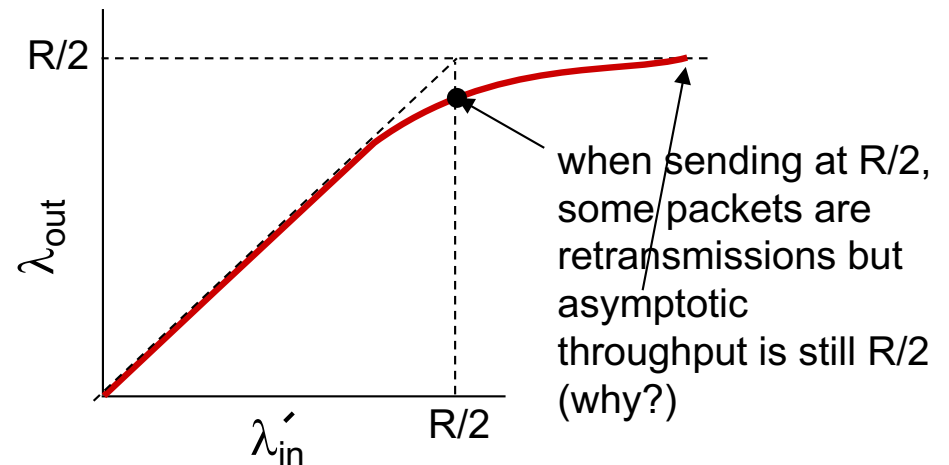
λ_{out}

copy

A

no buffer space!

Host B

Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost,
dropped at router due
to full buffers
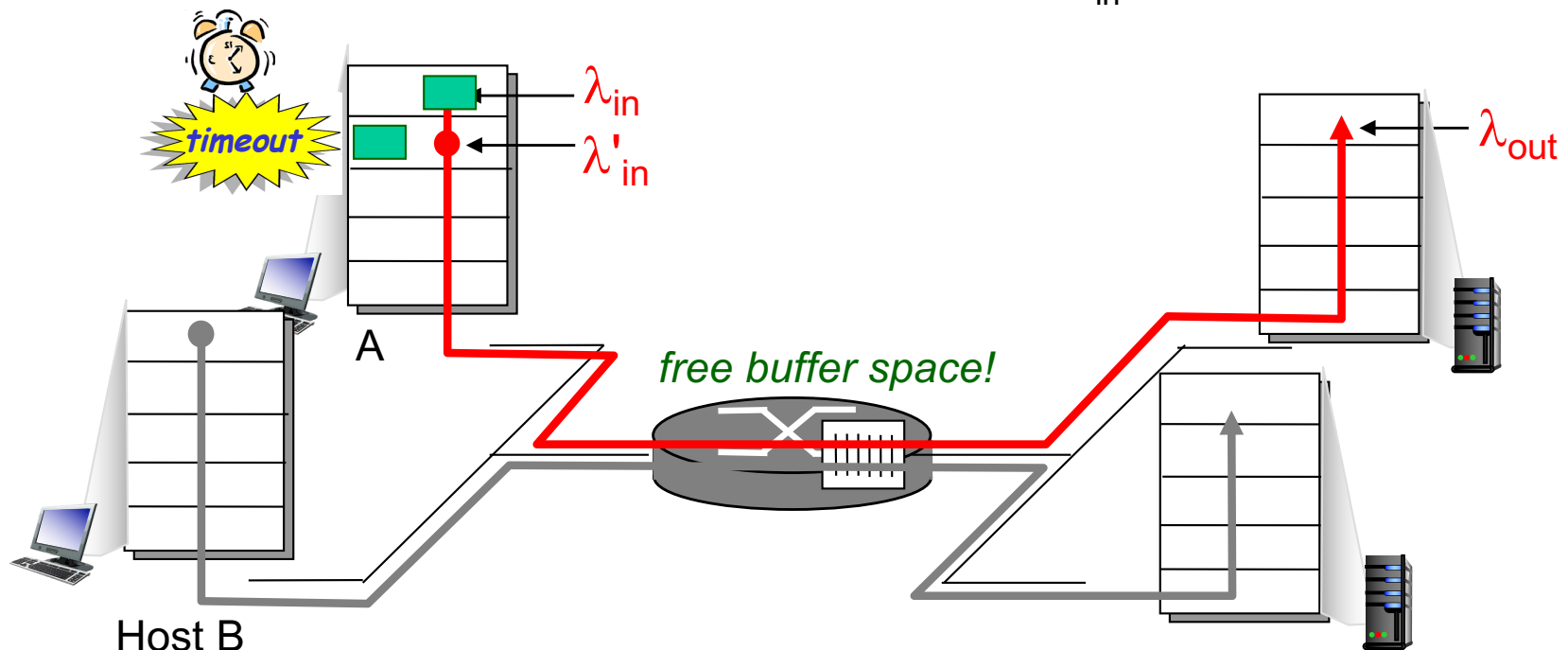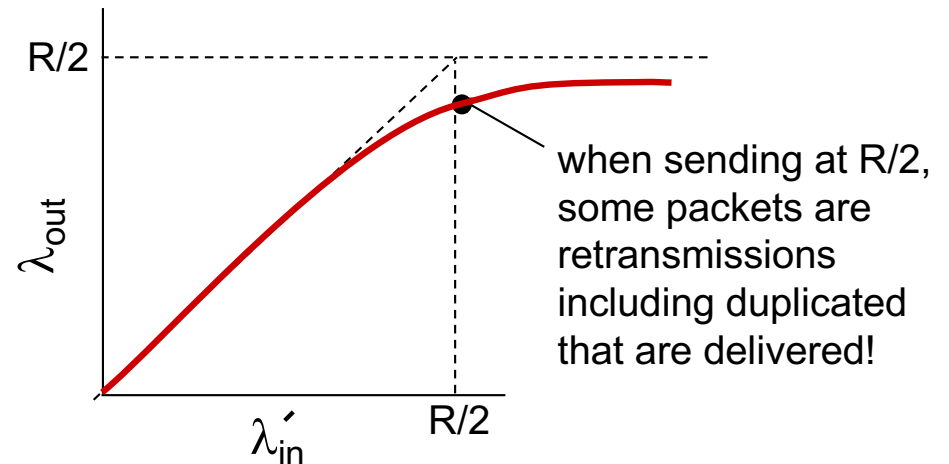
❖ sender only resends if
  packet *known* to be lost

when sending at R/2,
some packets are
retransmissions but
asymptotic
throughput is still R/2
(why?)

λ_{in} : original data

λ'_{in}: original data, *plus*
          retransmitted data

free buffer space!

λ_{out}

A

Host B

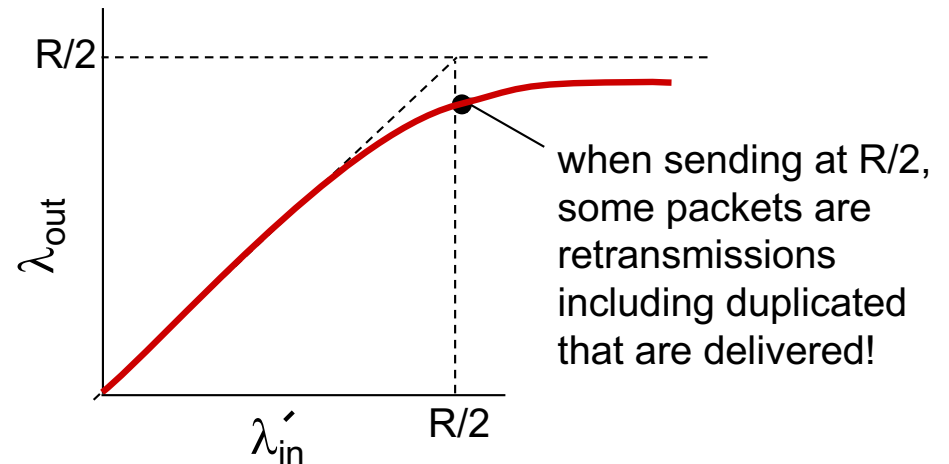# Causes/costs of congestion: scenario 2

Realistic: duplicates

❖ packets can be lost, dropped at router due to full buffers

❖ sender times out prematurely, sending *two* copies, both of which are delivered

when sending at R/2, some packets are retransmissions including duplicated that are delivered!

timeout

free buffer space!

λ_{in}

λ'_{in}

λ_{out}

A

Host B

Causes/costs of congestion: scenario 2

Realistic: duplicates

❖ packets can be lost, dropped at router due to full buffers

❖ sender times out prematurely, sending *two* copies, both of which are delivered

when sending at R/2, some packets are retransmissions including duplicated that are delivered!
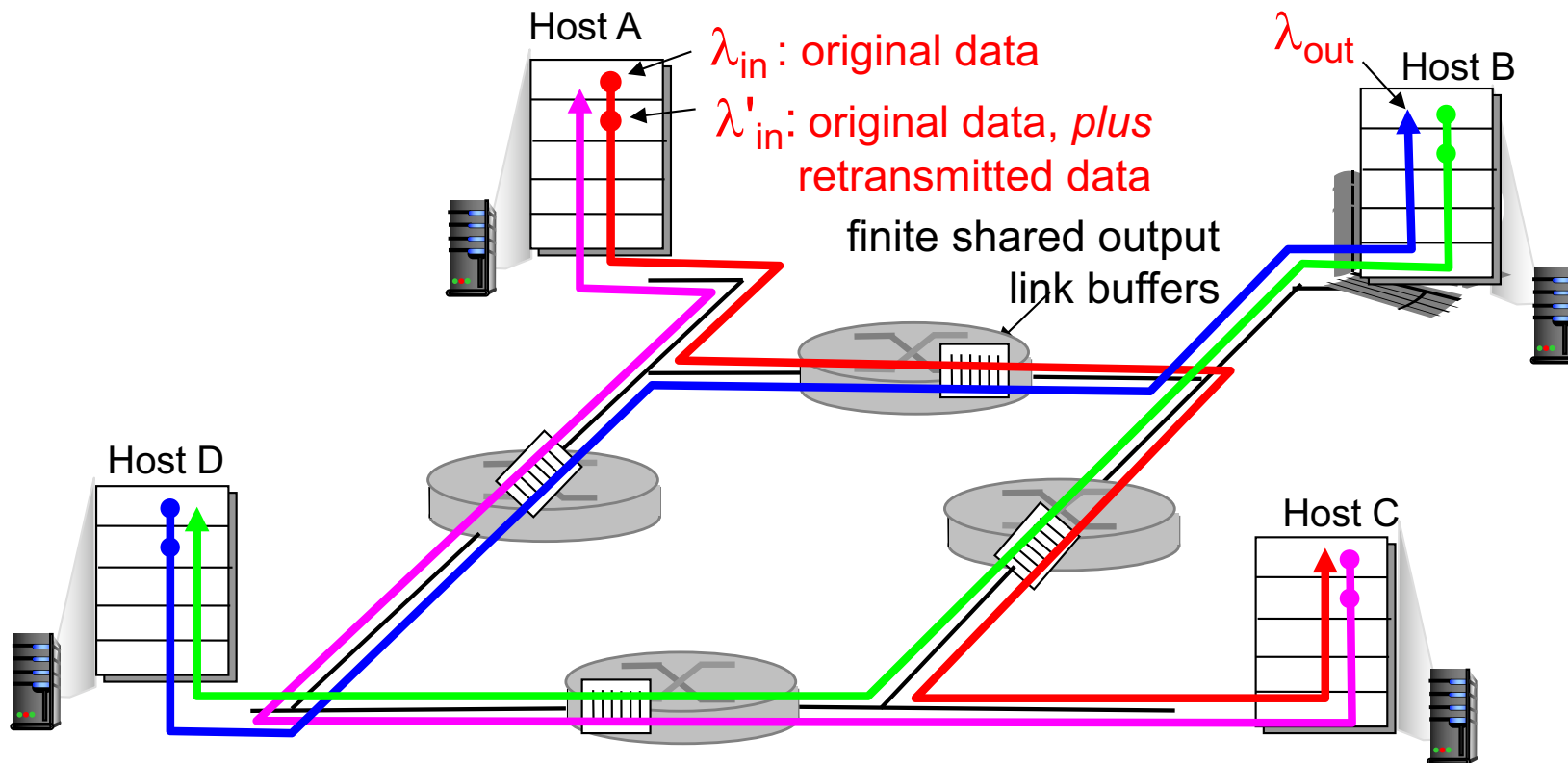
"costs" of congestion:

❖ more work (retrans) for given "throughput"

❖ unneeded retransmissions: link carries multiple copies of pkt

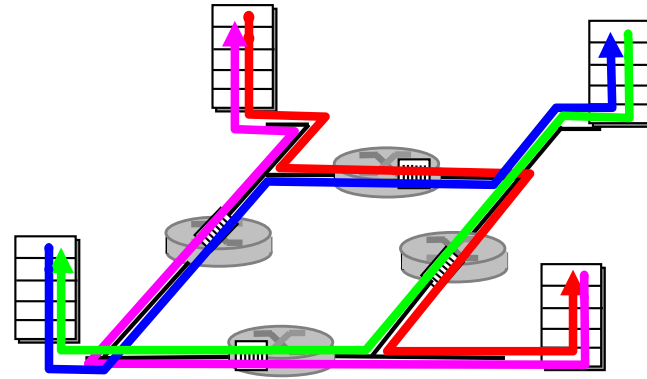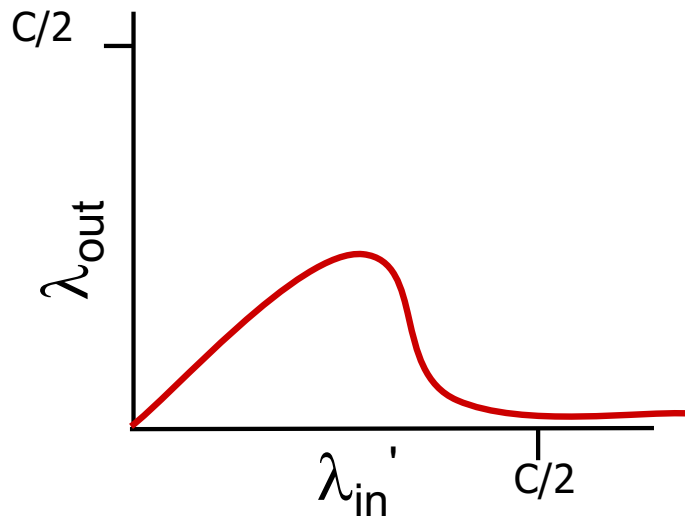▪ decreasing throughput

Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ_{in}' increase?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0

Host A

λ_{in} : original data

λ_{out} Host B

λ_{in}': original data, *plus* retransmitted data

finite shared output link buffers

Host D

Host C

Causes/costs of congestion: scenario 3

another "cost" of congestion:

❖ when packet dropped, any upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

- ❖ no explicit feedback from network
- ❖ congestion inferred from end-system observed loss, delay
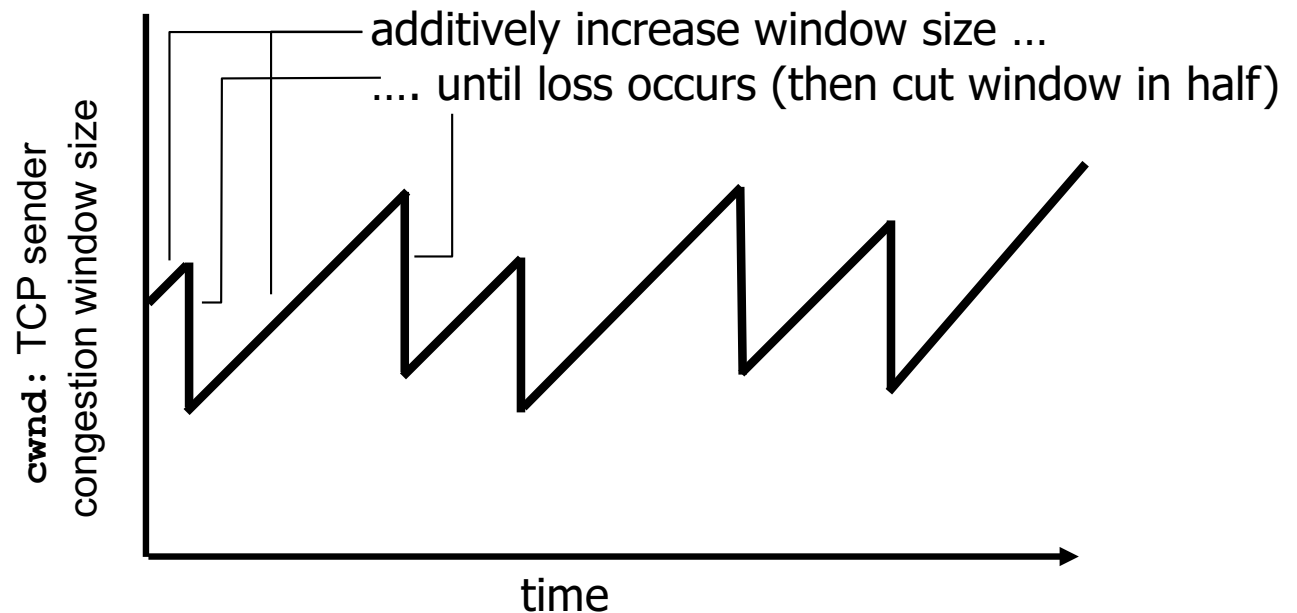- ❖ approach taken by TCP

network-assisted congestion control:

- ❖ routers provide feedback to end systems
  - ▪ single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
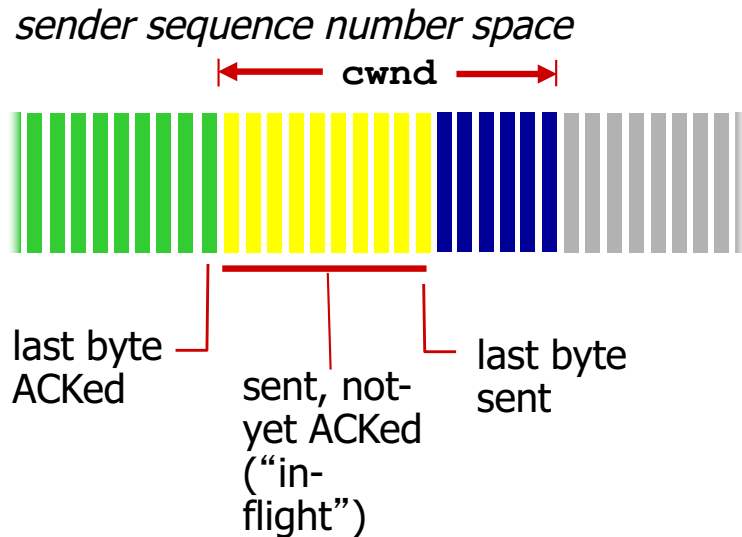 - ▪ explicit rate for sender to send at

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

additively increase window size …

…. until loss occurs (then cut window in half)

cwnd: TCP sender congestion window size

time

TCP Congestion Control: details

sender sequence number space

cwnd

last byte ACKed

sent, not-yet ACKed ("in-flight")

last byte sent

❖ **sender limits transmission:**

LastByteSent-
LastByteAcked $\leq$ cwnd

❖ **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

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

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

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

Host A   Host B

RTT

one segment

two segments

four segments
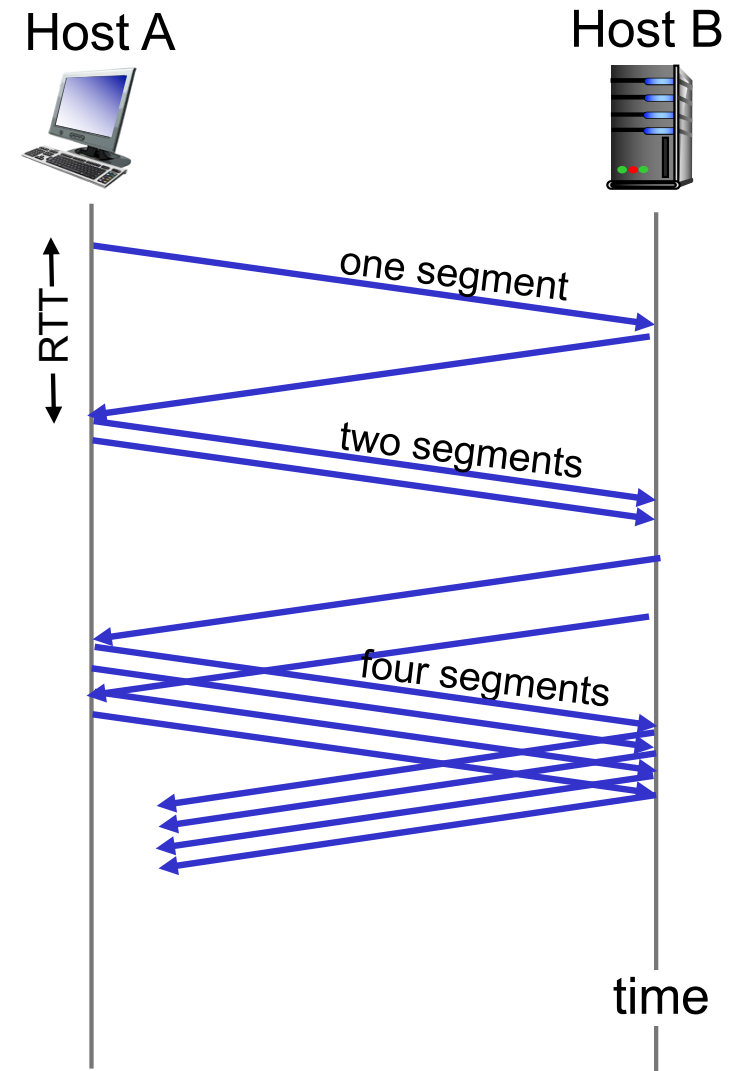
time

TCP: detecting, reacting to loss

- ❖ loss indicated by timeout:
  - ▪ `cwnd` set to 1 MSS;
  - ▪ window then grows exponentially (as in slow start) to threshold, then grows linearly
- ❖ 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
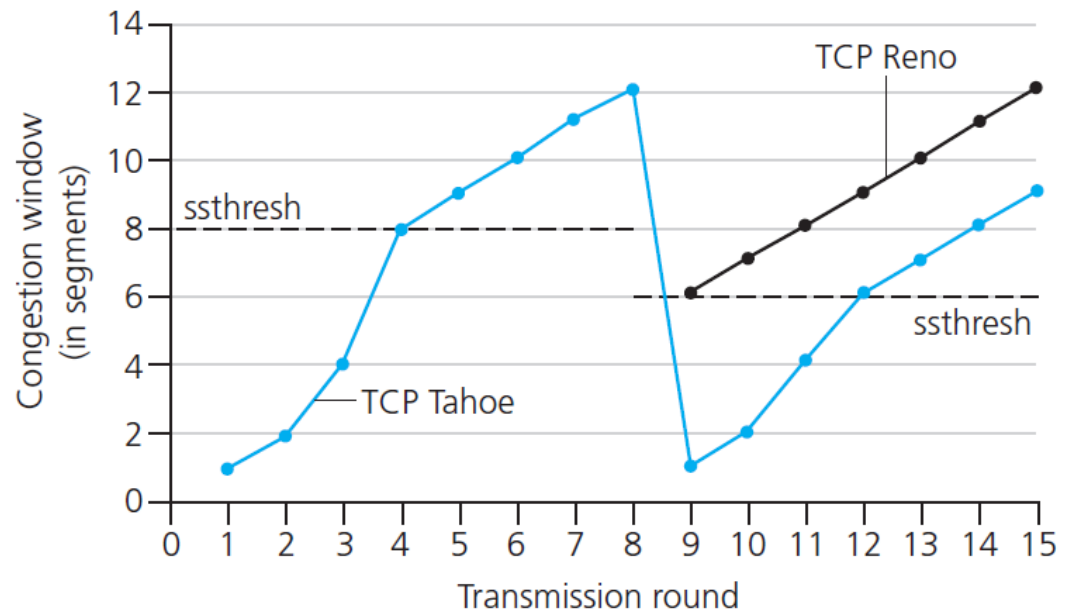- ❖ TCP Tahoe always sets `cwnd` to 1 (timeout or 3 duplicate acks)

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.

Implementation:

❖ variable `ssthresh`

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

TCP throughput

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

$$\text{avg TCP thruput} = \frac{3}{4} \frac{W}{RTT} \text{ bytes/sec}$$
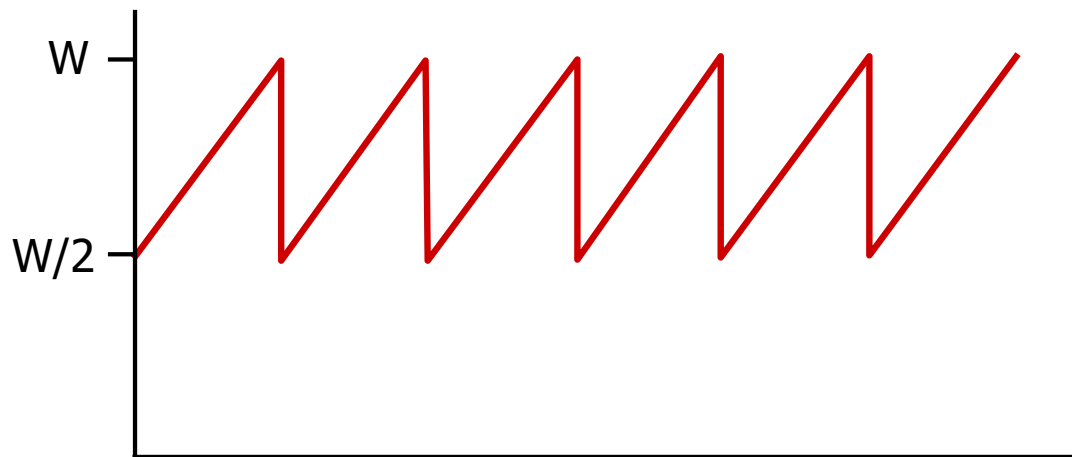
TCP Futures: TCP over "long, fat pipes"

❖ 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!*

❖ new versions of TCP for high-speed

TCP Fairness

fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

TCP connection 1

TCP connection 2

bottleneck router capacity R
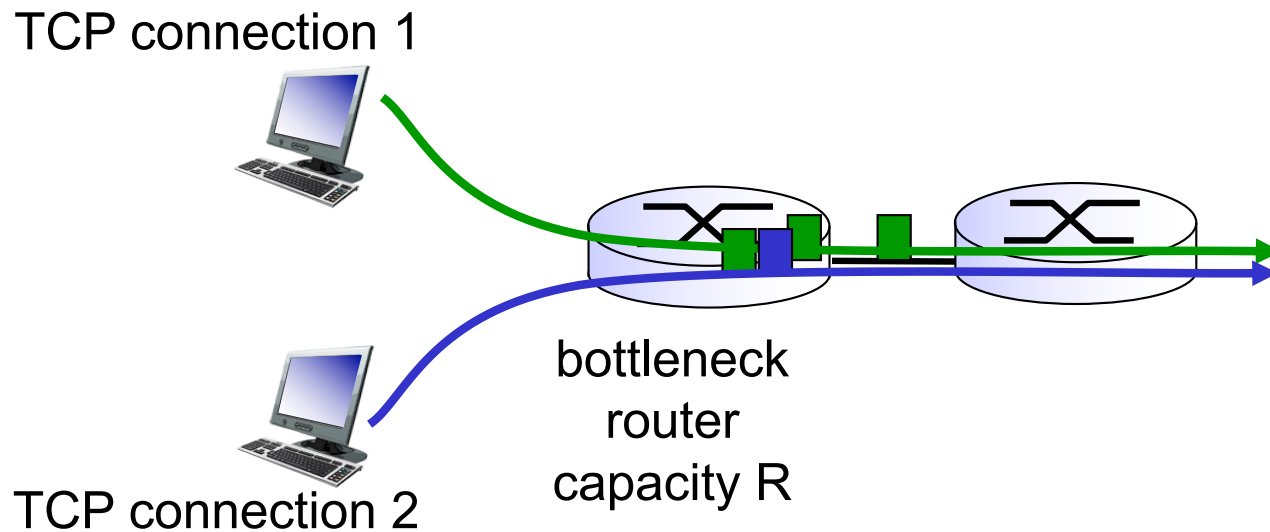
Why is TCP fair?

two competing sessions:

❖ additive increase gives slope of 1, as throughout increases
❖ multiplicative decrease decreases throughput proportionally
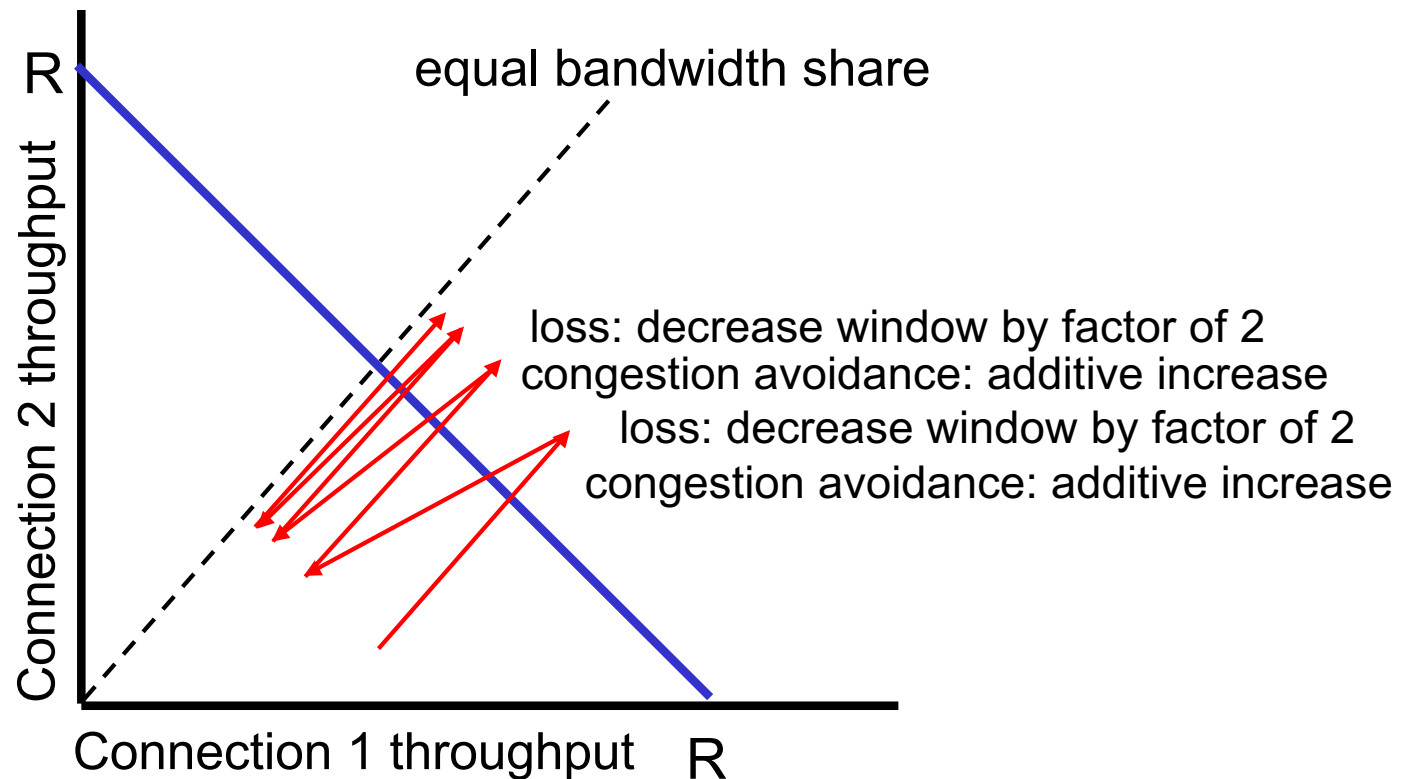
Fairness (more)

Fairness and UDP

❖ multimedia apps often do not use TCP
  ▪ do not want rate throttled by congestion control

❖ instead use UDP:
  ▪ send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

❖ application can open multiple parallel connections between two hosts

❖ web browsers do this

❖ e.g., link of rate R with 9 existing connections:
  ▪ new app asks for 1 TCP, gets rate R/10
  ▪ new app asks for 11 TCPs, gets R/2