Chapter III: Transport Layer
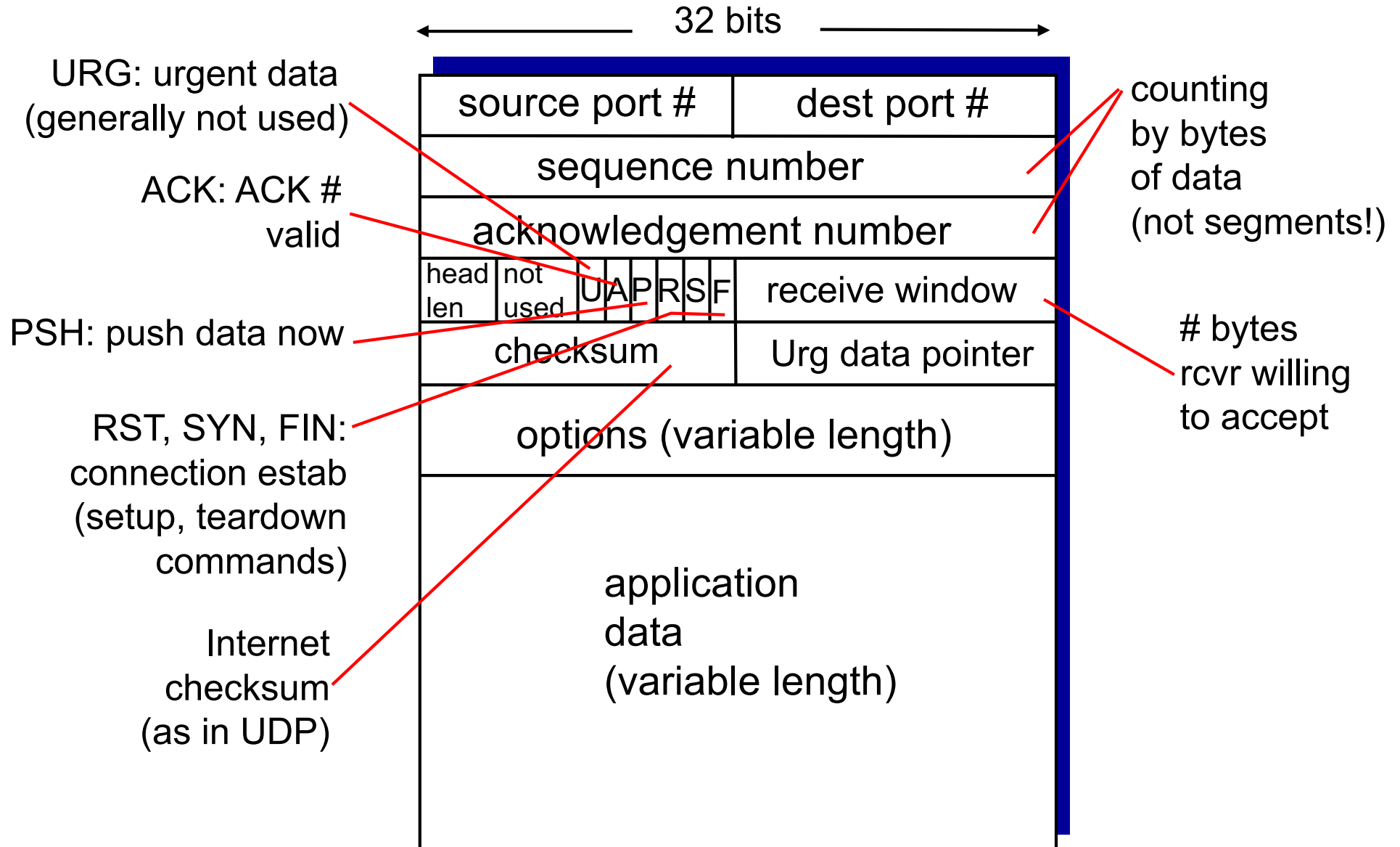
UG3 Computer Communications & Networks (COMN)

Myungjin Lee
myungjin.lee@ed.ac.uk

Slides copyright of Kurose and Ross

TCP: Overview   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

TCP segment structure

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now

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

Internet checksum (as in UDP)

32 bits

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

TCP seq. numbers, ACKs

sequence numbers:

– byte stream "number" of first byte in segment's data
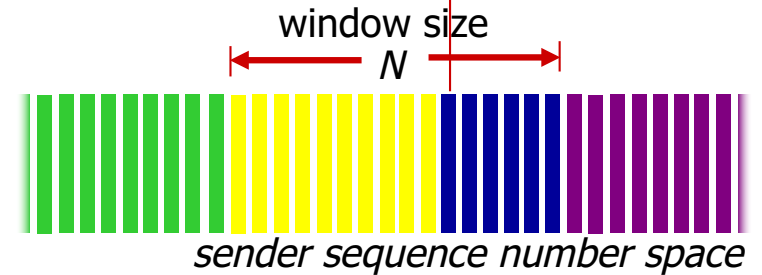
acknowledgements:

– 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

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

Byte stream in TCP

Cannot be transmitted now

HTTP Get Message (K bytes)

100th byte

M bytes

Window: N bytes

TCP header (seq no. = 100)	HTTP Get Messag

TCP seq. numbers, ACKs

Host A   Host B

User
types
'C'

    Seq=42, ACK=79, data = 'C'

                      host ACKs
                      receipt of
    Seq=79, ACK=43, data = 'C' 'C', echoes

host ACKs   back 'C'
receipt
of echoed
'C'   Seq=43, ACK=80
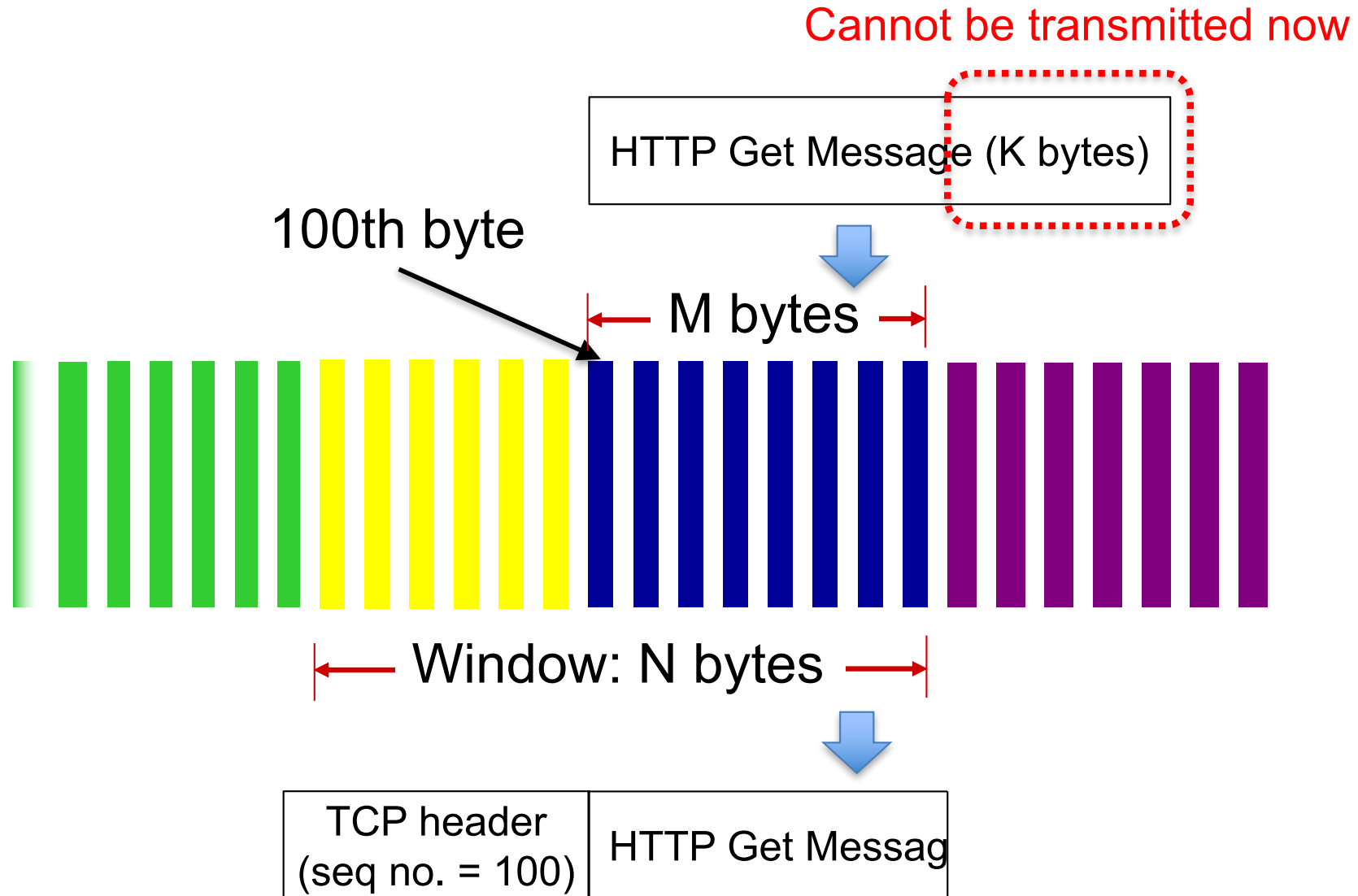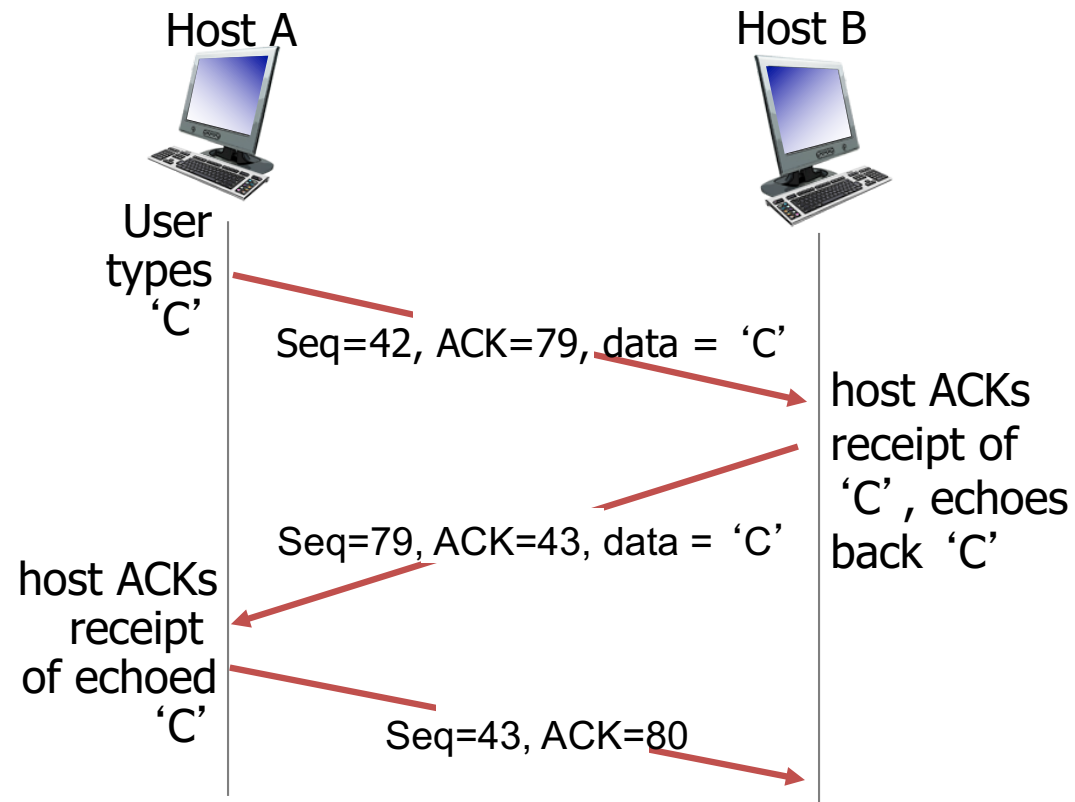
simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

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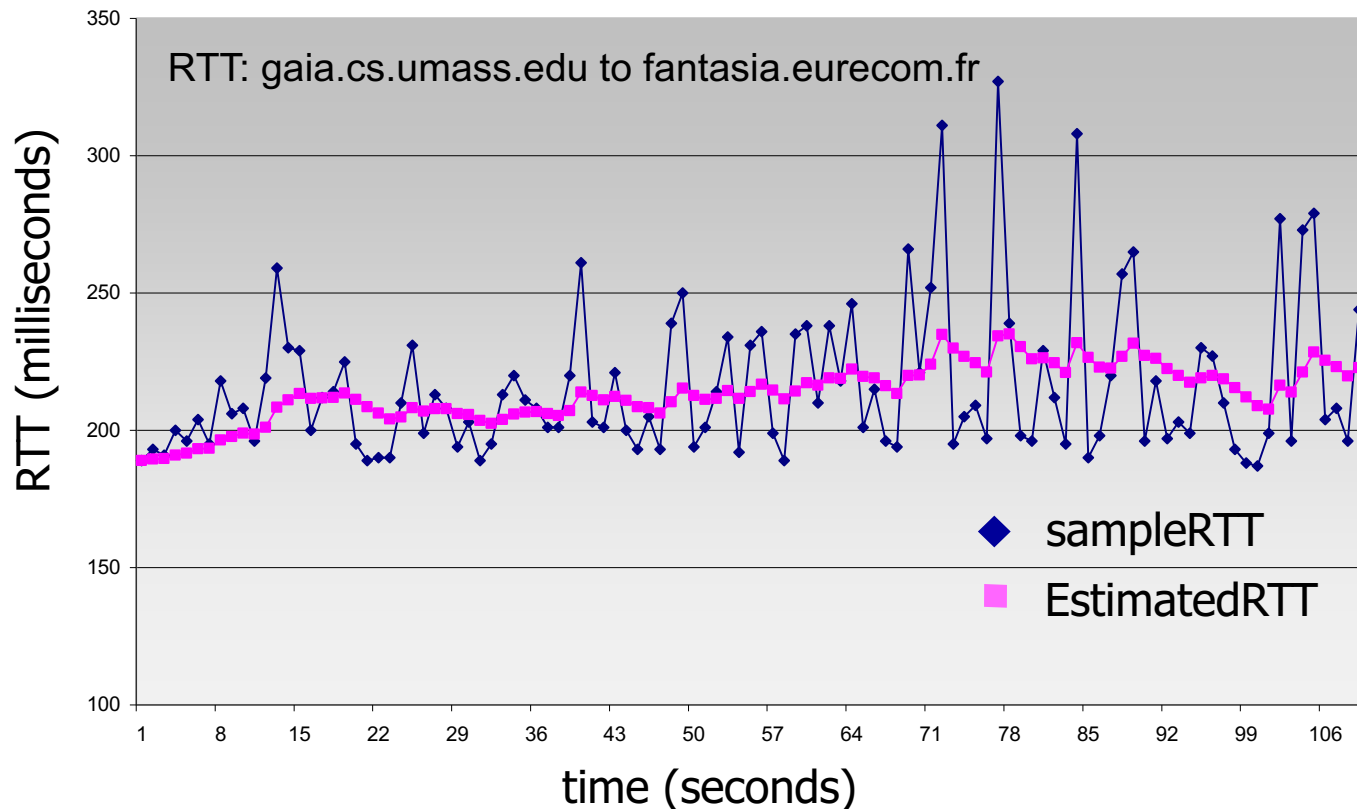- *too long:* slow reaction to segment loss

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

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

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

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

RTT (milliseconds) vs time (seconds)

◆ sampleRTT
■ EstimatedRTT

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"

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
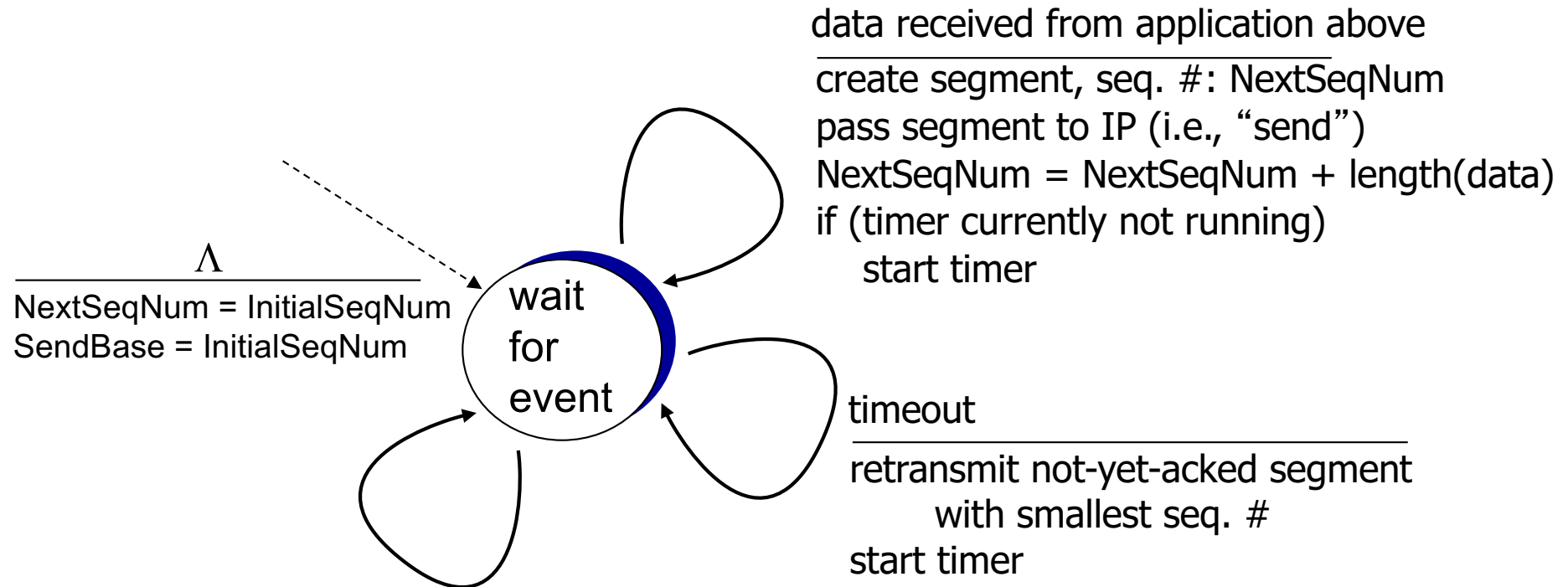 - expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
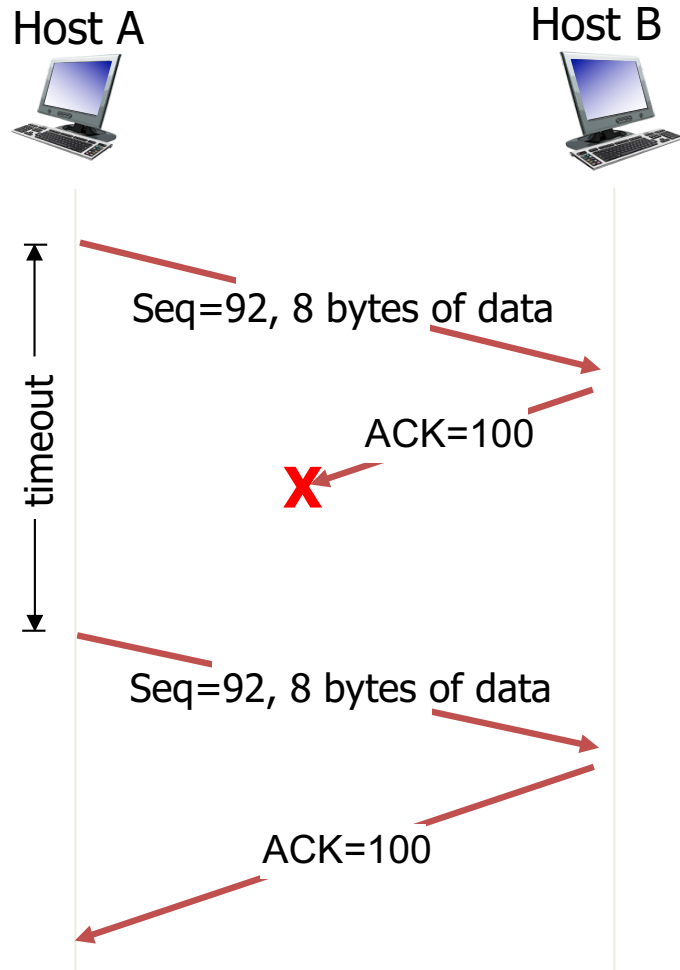 - start timer if there are still unacked segments
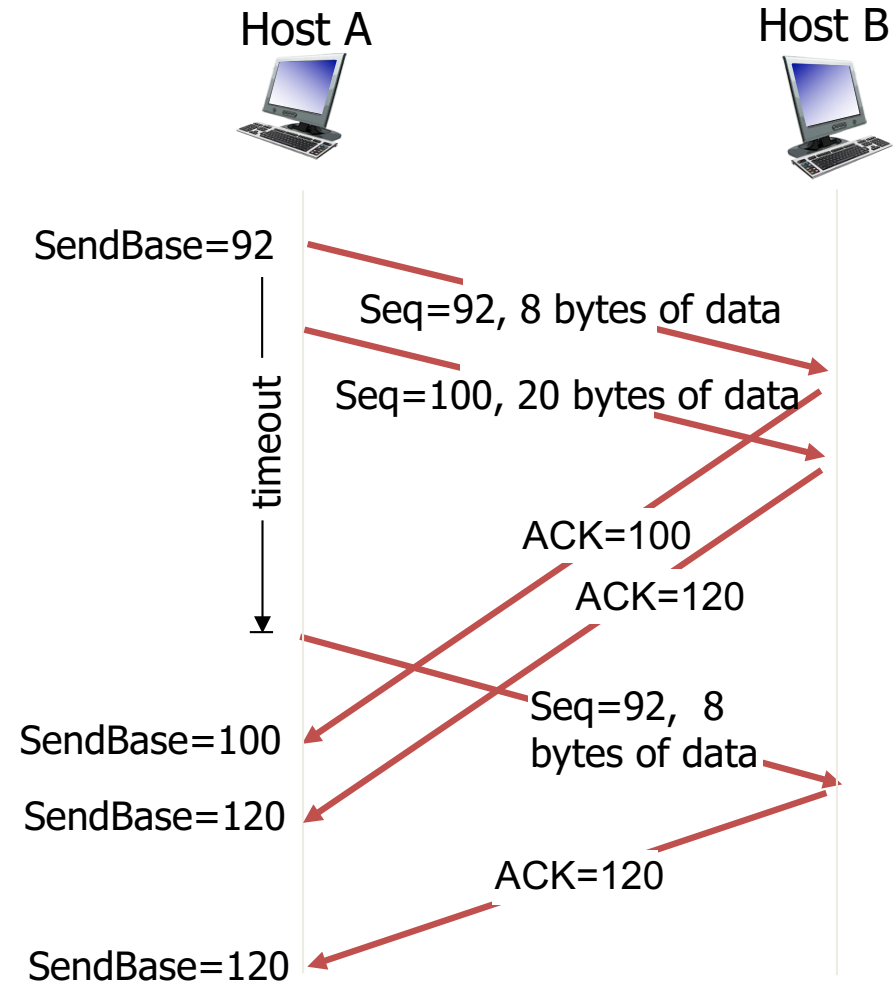
TCP sender (simplified)

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

Λ
___
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait
for
event

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
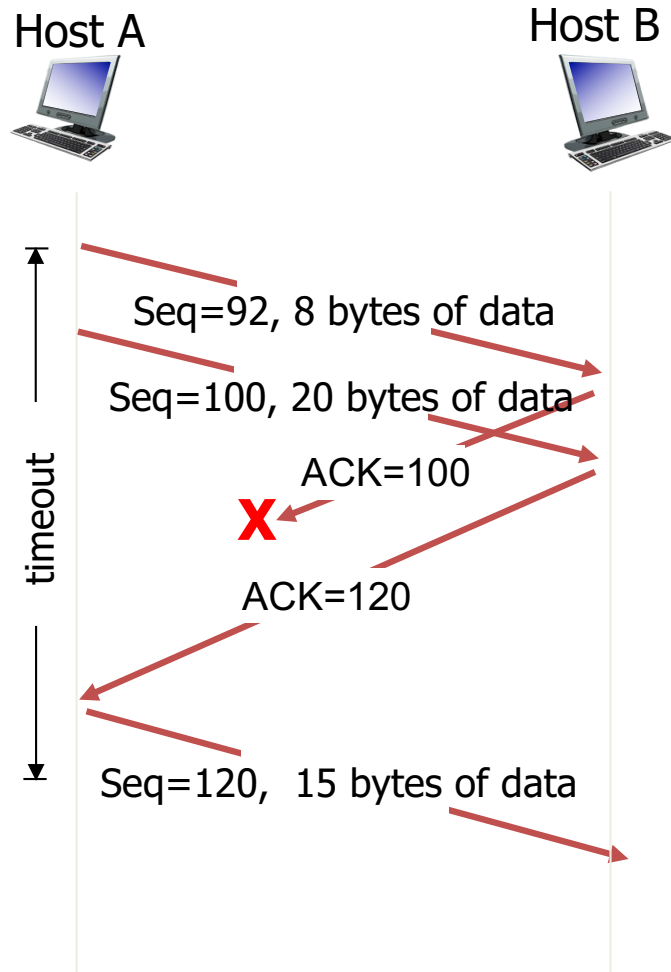   else stop timer
    }

TCP: 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 ACK generation [RFC 5861]

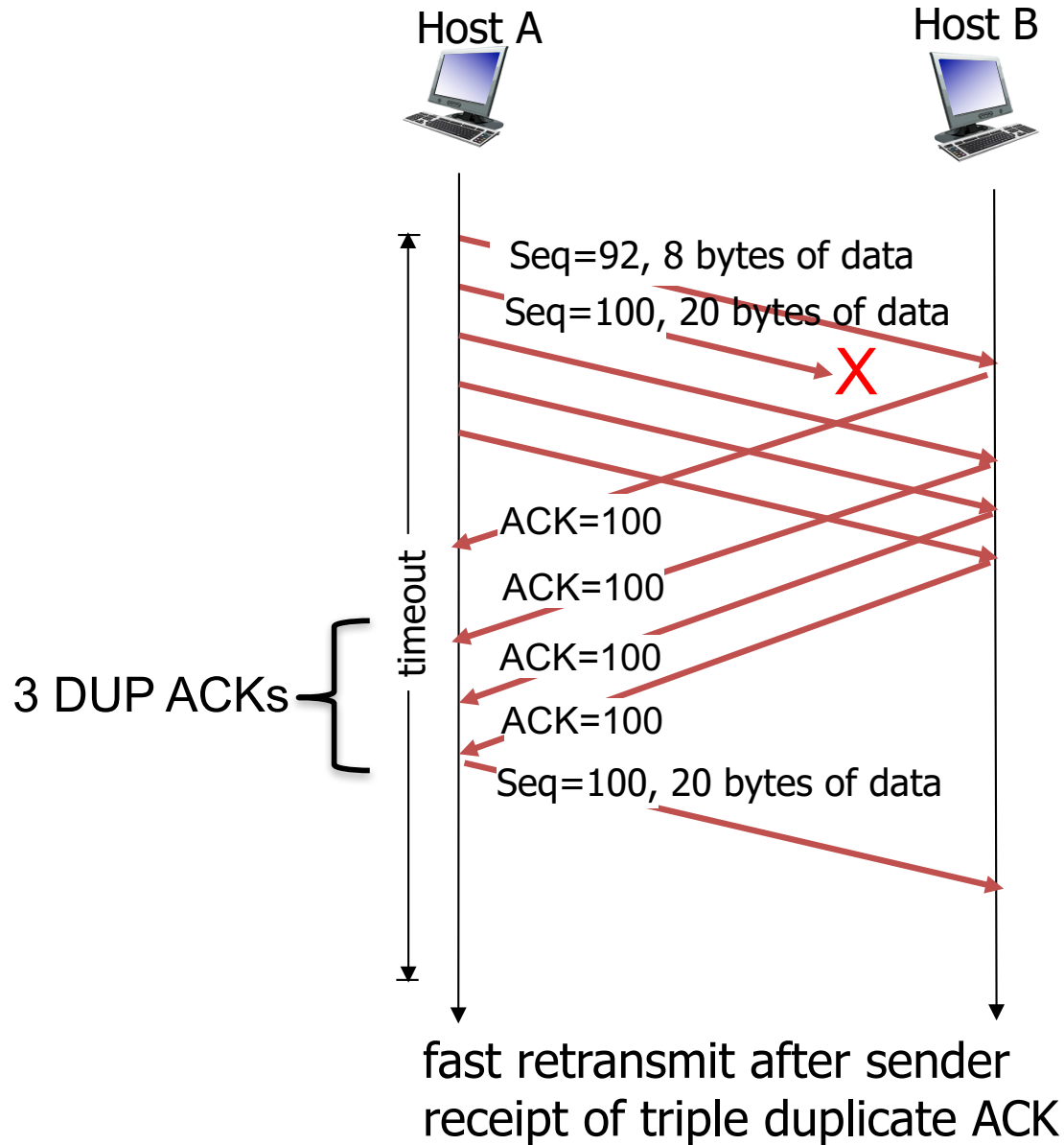| event at receiver	TCP receiver action
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

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
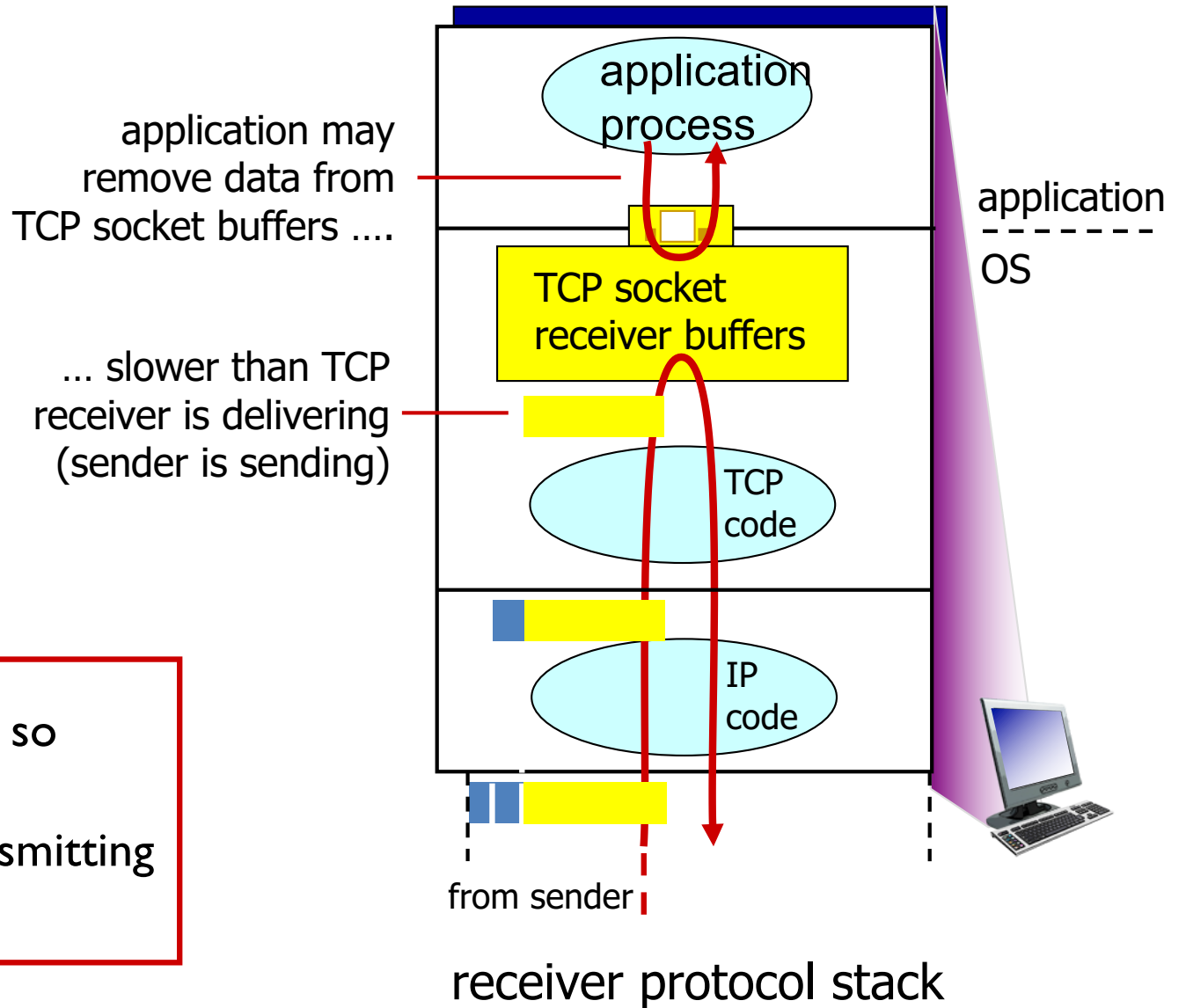
TCP fast retransmit

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
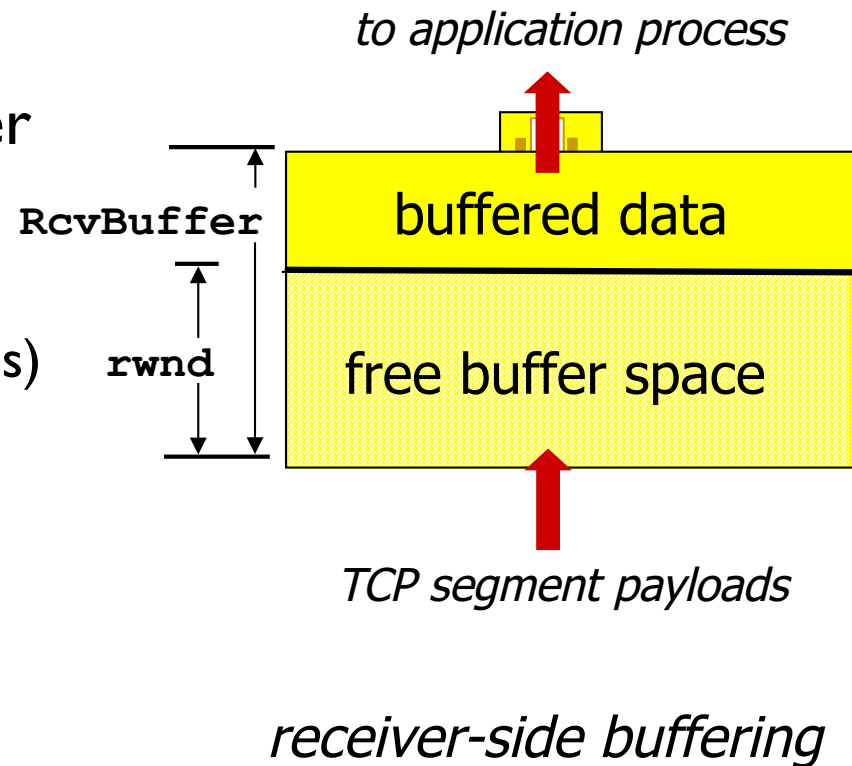
82

TCP fast retransmit

Host A

Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

X

ACK=100

ACK=100

3 DUP ACKs

ACK=100

ACK=100

timeout

Seq=100, 20 bytes of data

fast retransmit after sender
receipt of triple duplicate ACK

TCP flow control

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

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

application
process

TCP socket
receiver buffers

application
OS

TCP
code

flow control

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

IP
code

from sender

receiver protocol stack
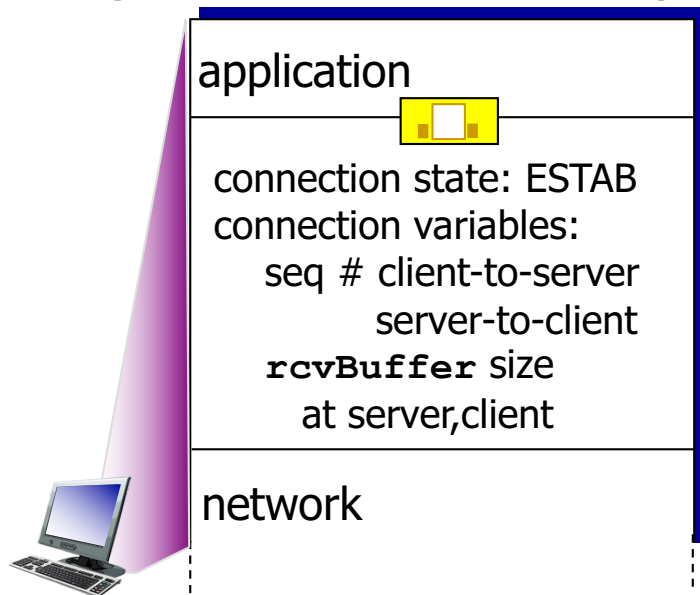
84

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
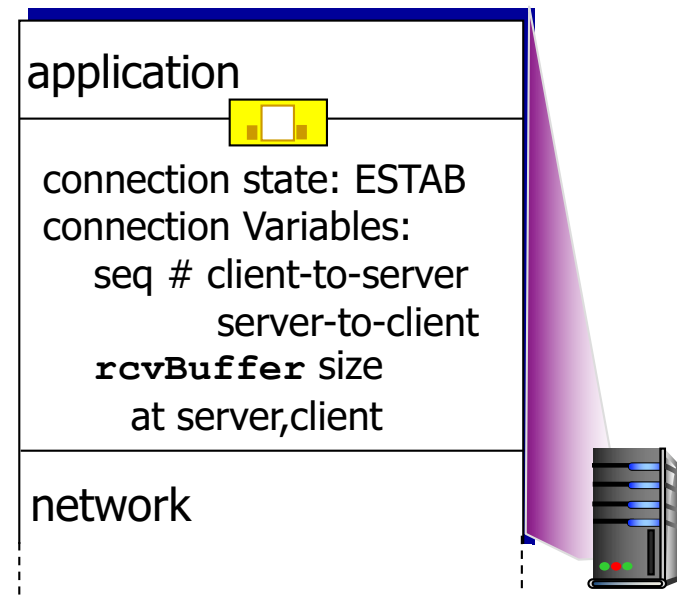
receiver-side buffering

85

Connection Management

before exchanging data, sender/receiver "handshake":

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

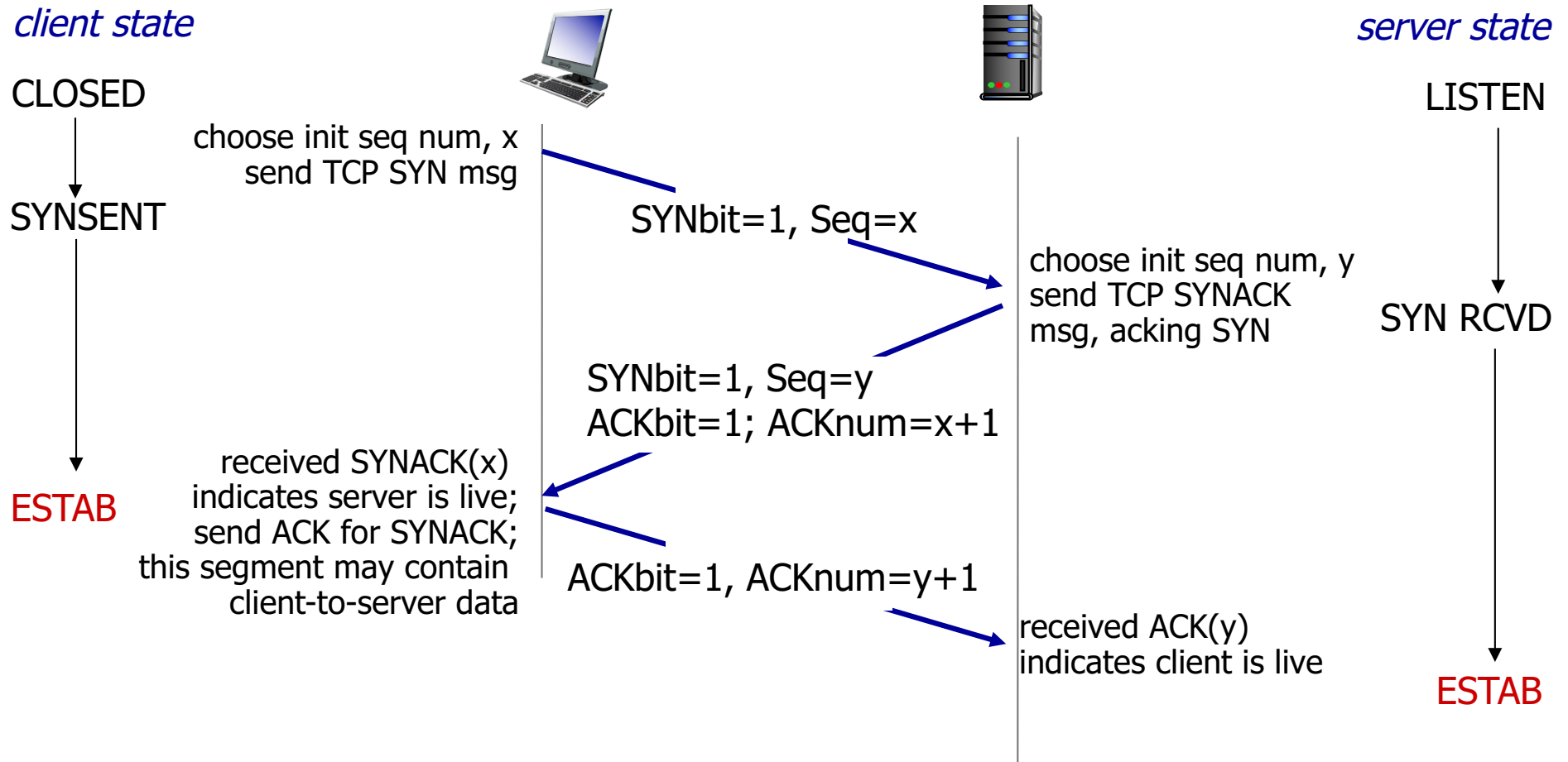- agree on connection parameters

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();
```

TCP 3-way handshake

client state

CLOSED

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

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

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

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

87

TCP 3-way handshake: FSM

closed

Socket connectionSocket =
  welcomeSocket.accept();
_____
              Λ

Socket clientSocket =
  newSocket("hostname","port
  number");
_____
 SYN(seq=x)

listen

SYN(x)
_____
SYNACK(seq=y,ACKnum=x+1)
create new socket for
communication back to client

SYN
rcvd

SYN
sent

ESTAB

ACK(ACKnum=y+1)
_____
              Λ

SYNACK(seq=y,ACKnum=x+1)
_____
  ACK(ACKnum=y+1)

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()`

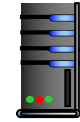FIN_WAIT_1 — can no longer send but can receive data

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!
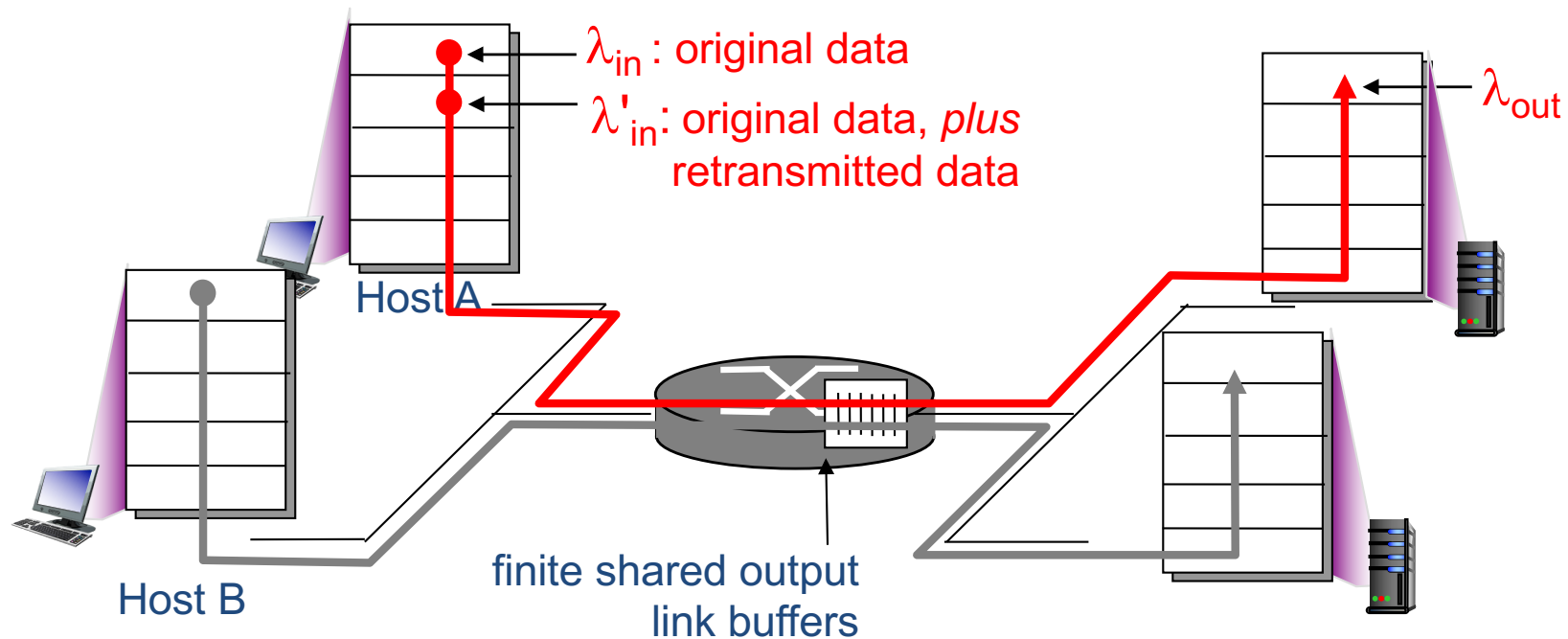
Causes/costs of congestion: scenario 1

original data: λ_{in}

throughput: λ_{out}

- two senders, two receivers

- one router, infinite buffers

- output link capacity: R

- no retransmission

Host A

Host B

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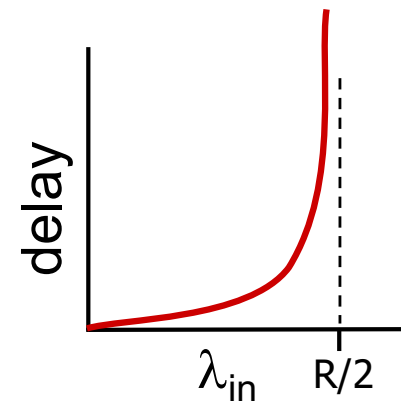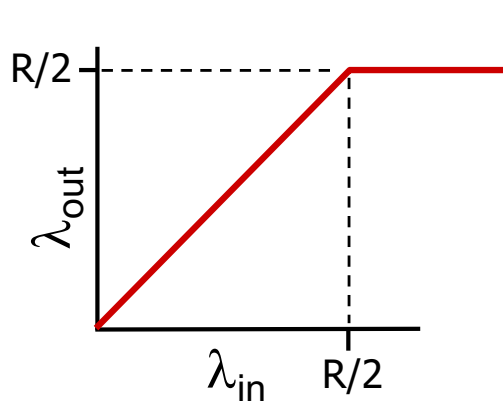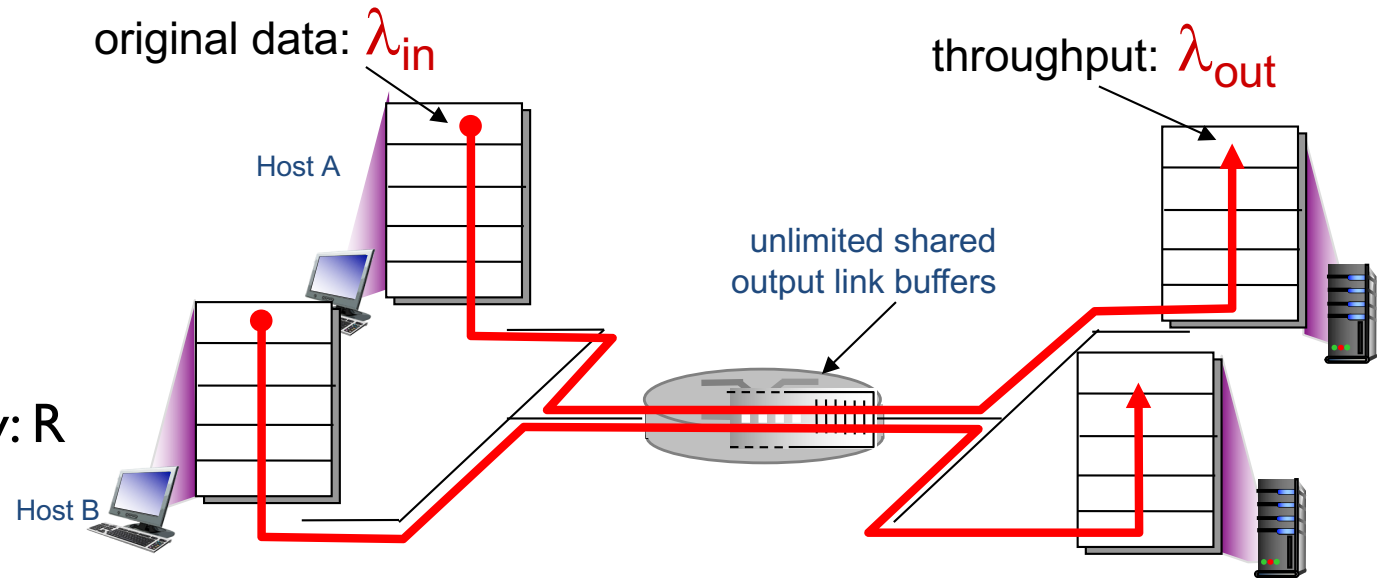
- maximum per-connection throughput: R/2

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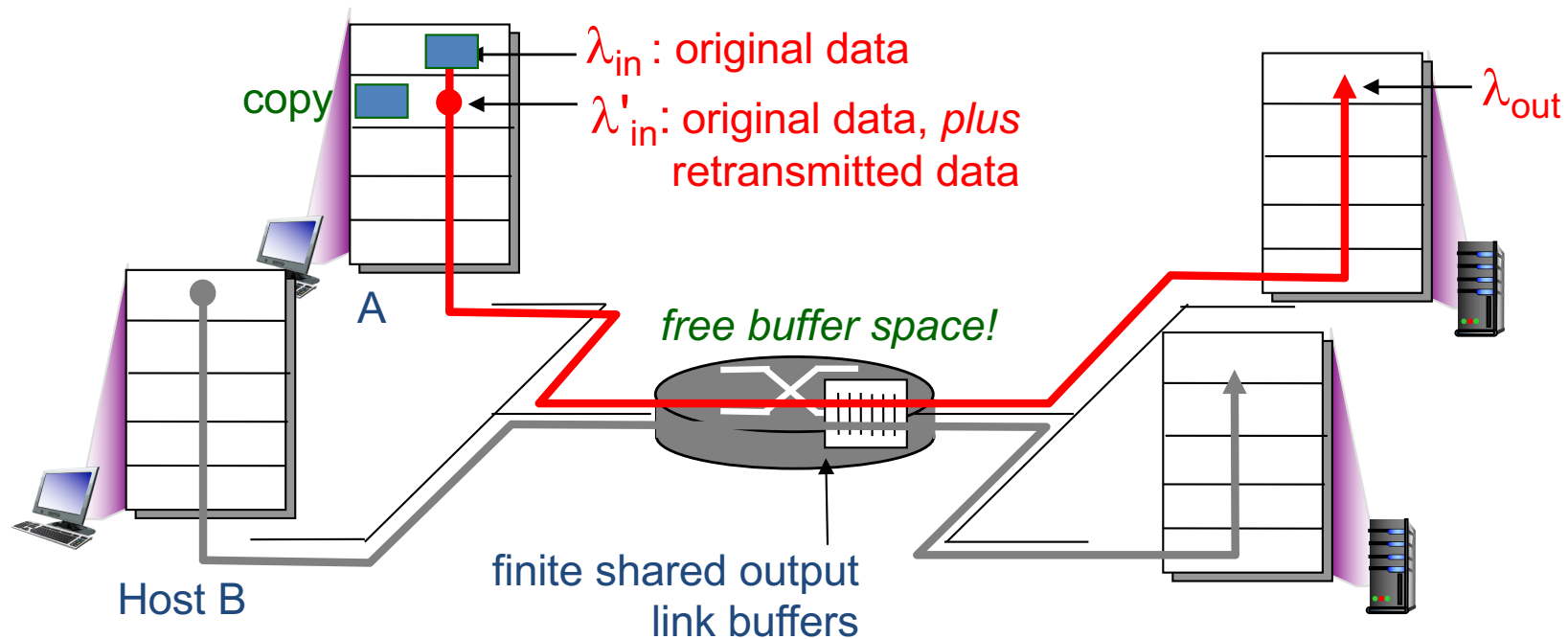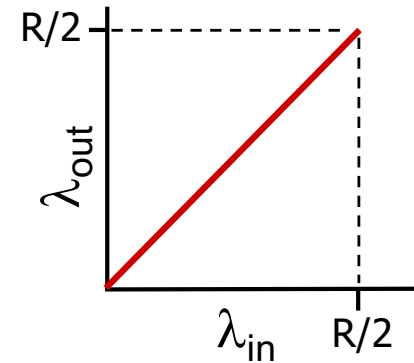
Causes/costs of congestion: scenario 2

- one router, *finite* buffers

- sender retransmission of timed-out packet
  - application-layer input = application-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

copy

A

Host B

free buffer space!

finite shared output link buffers

λ_{out}

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}
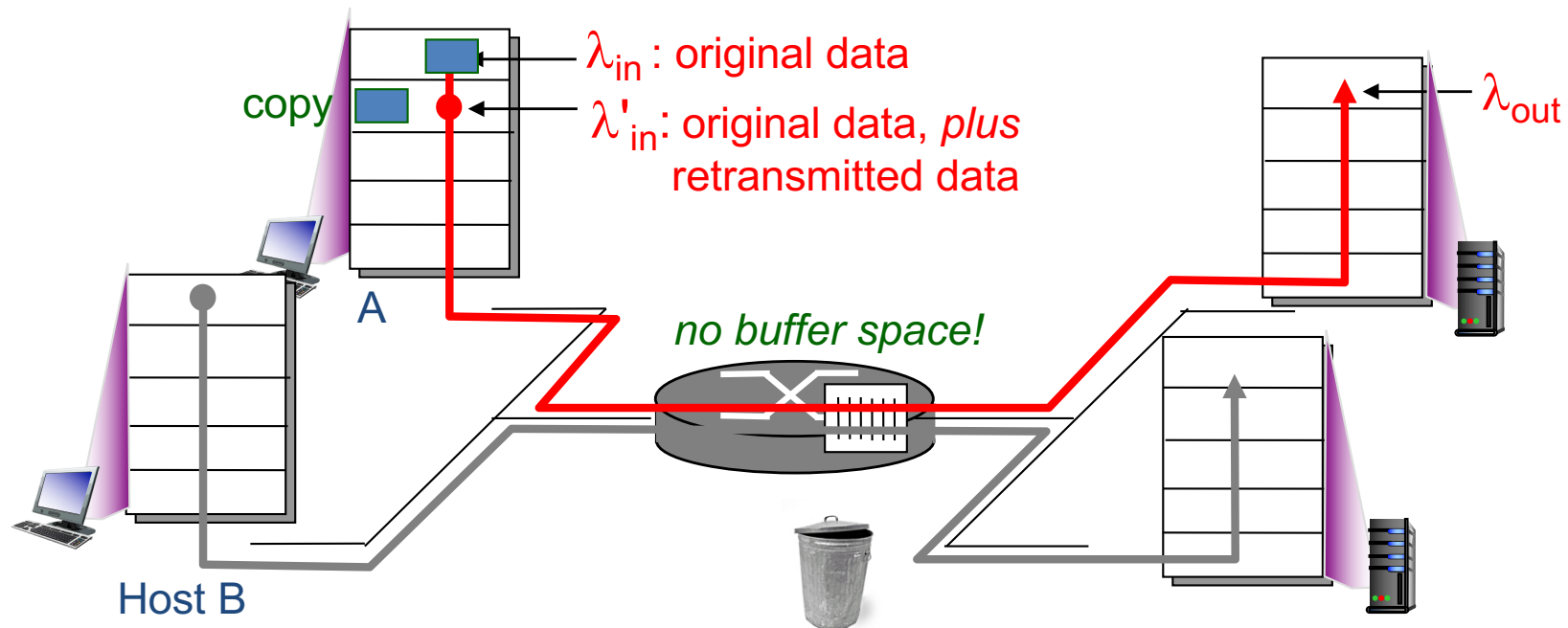
copy

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

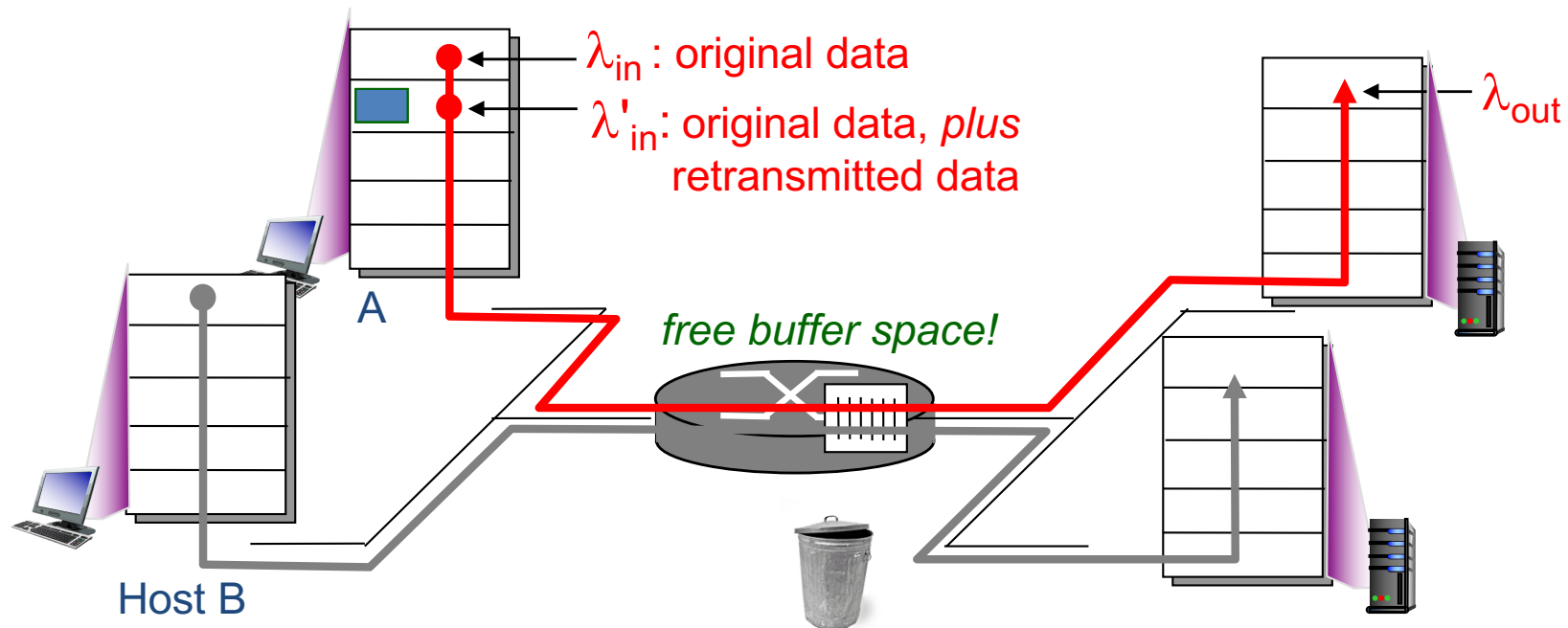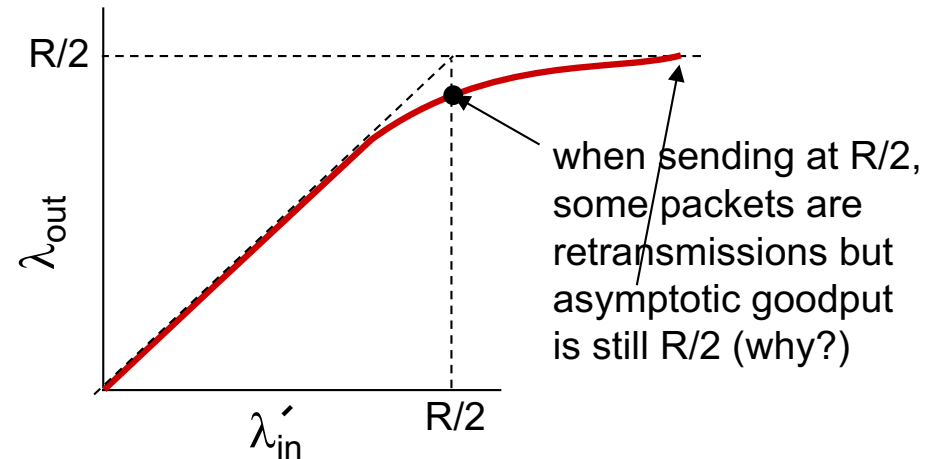- sender only resends if packet *known* to be lost

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

λ_{in} : original data

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

λ_{out}

A

free buffer space!

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!

Causes/costs of congestion: scenario 2

Realistic: duplicates

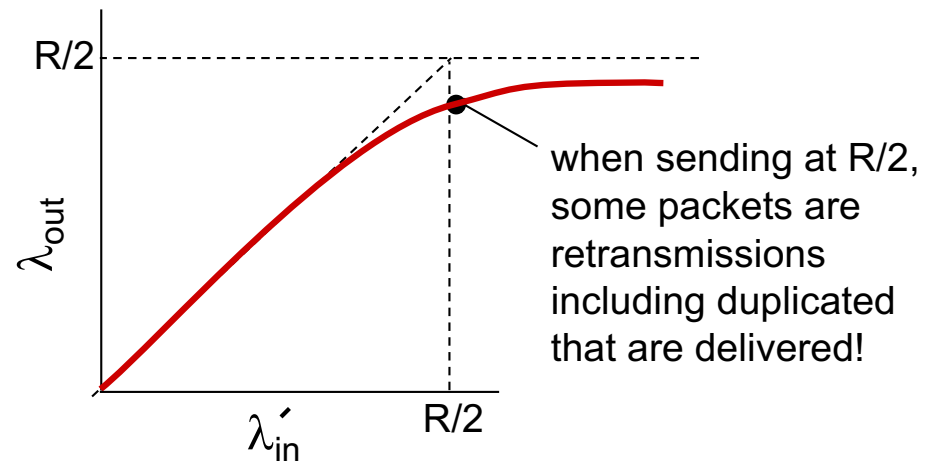❖ packets can be lost, dropped at router due to full buffers

❖ 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!
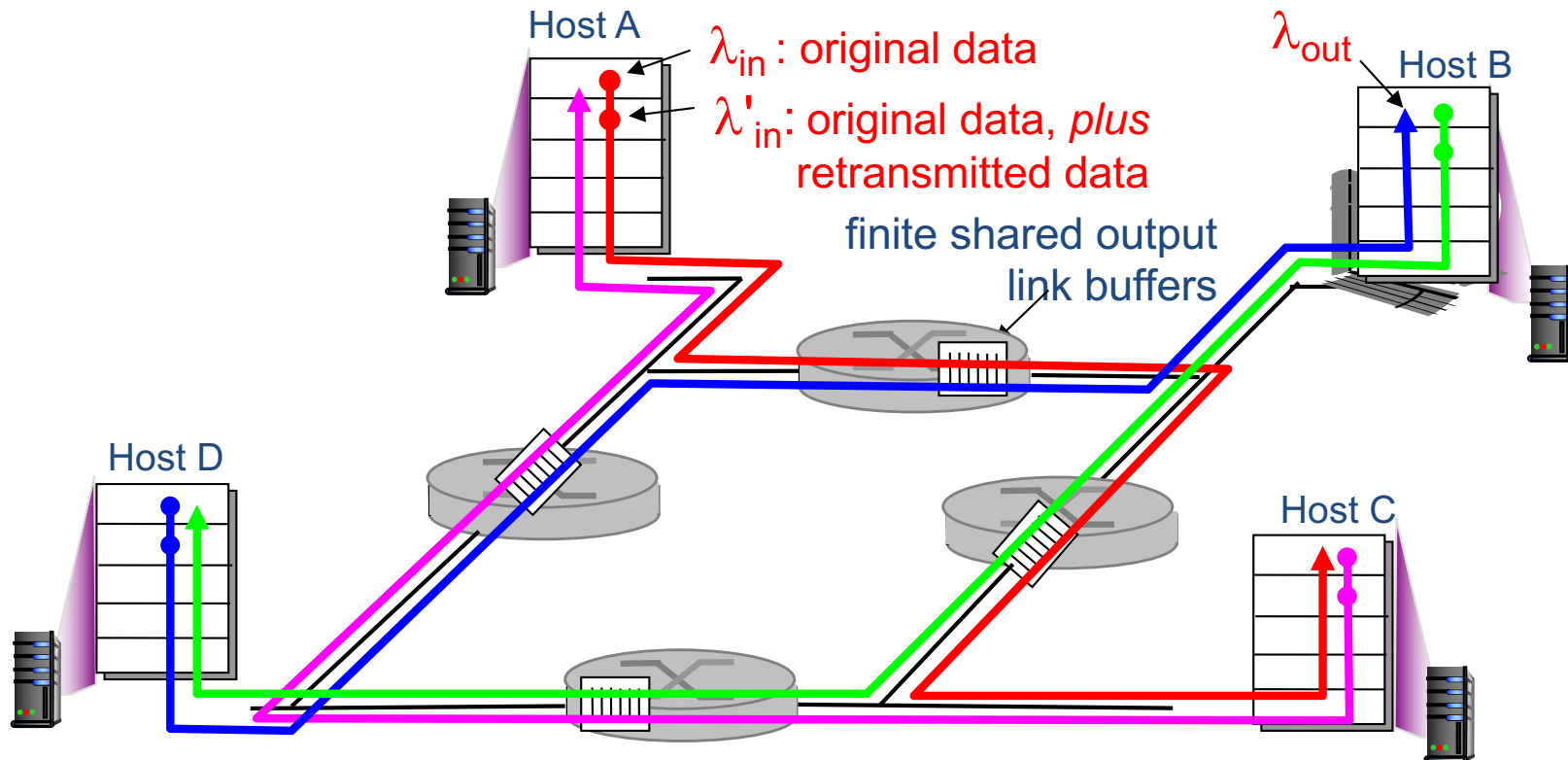
"costs" of congestion:

❖ more work (retrans) for given "goodput"

❖ unneeded retransmissions: link carries multiple copies of pkt
- decreasing goodput

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 $\to$ 0

Host A

λ_{in} : original data

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

finite shared output link buffers

λ_{out}

Host B

Host D

Host C

Causes/costs of congestion: scenario 3

Throughput by blue traffic

λ_{out}

R/2

λ_{in}'   R/2

Offered load by Host A

Bandwidth wastage
for packets dropped at
the 2nd router
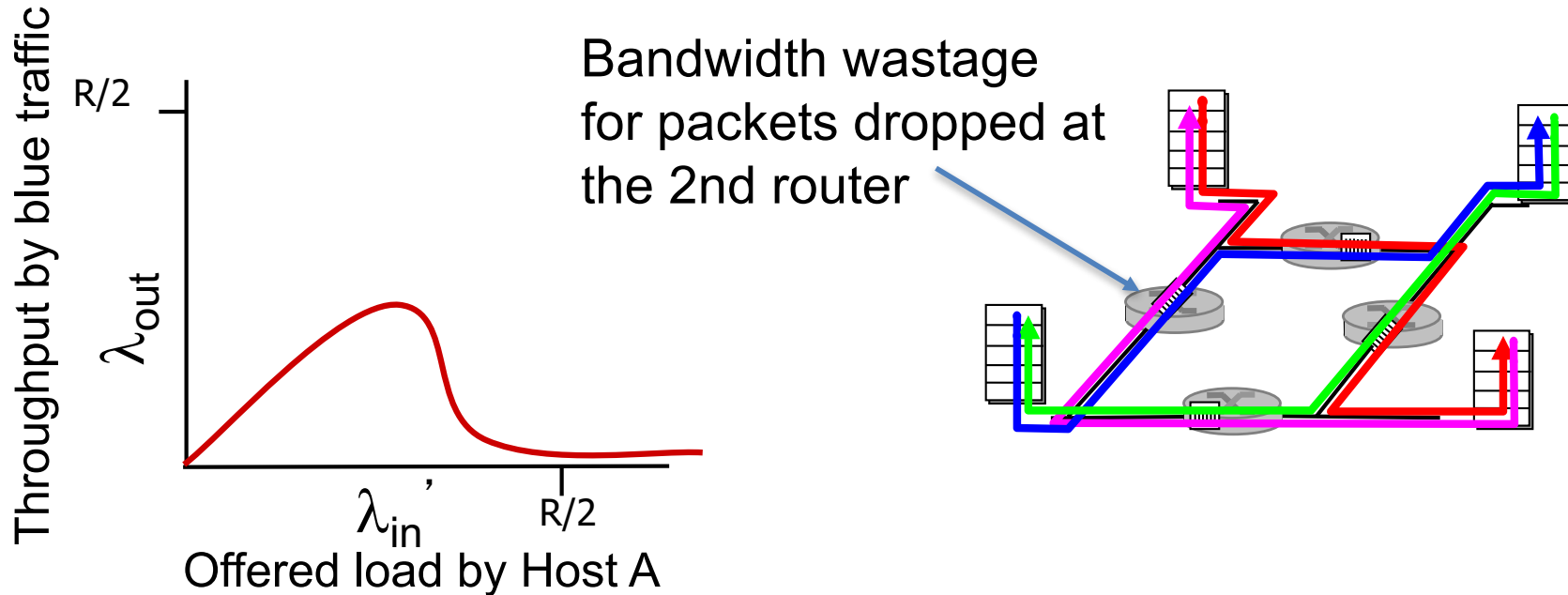
another "cost" of congestion:

❖ 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
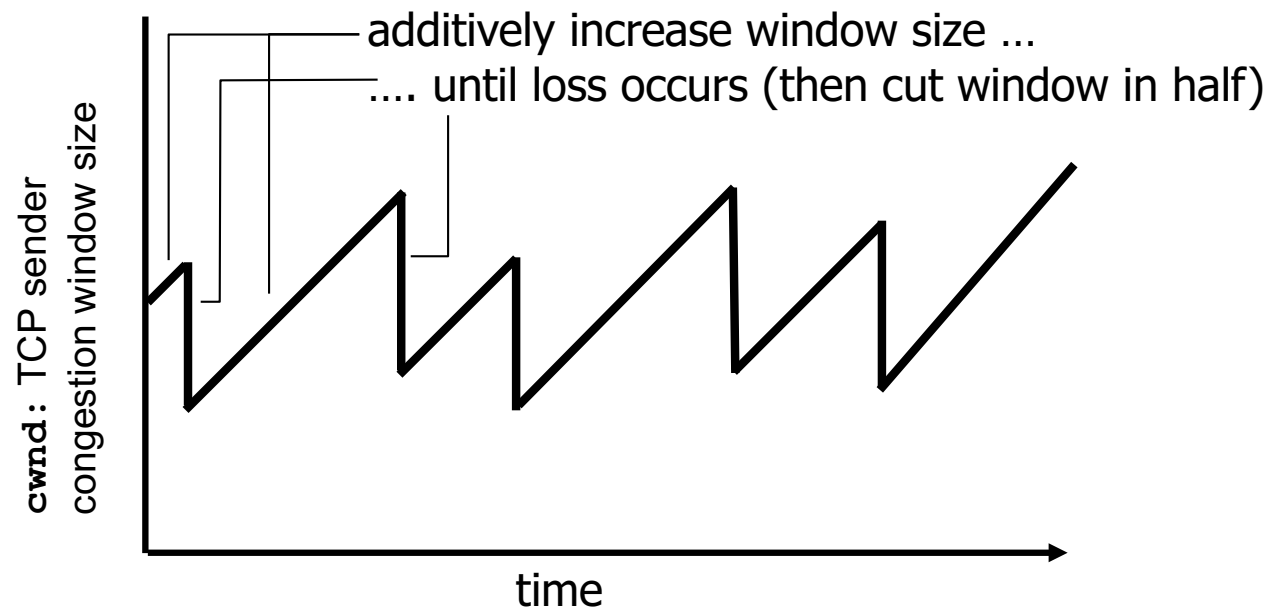
- approach taken by TCP

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 (AIMD)

❖ *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

last byte ACKed

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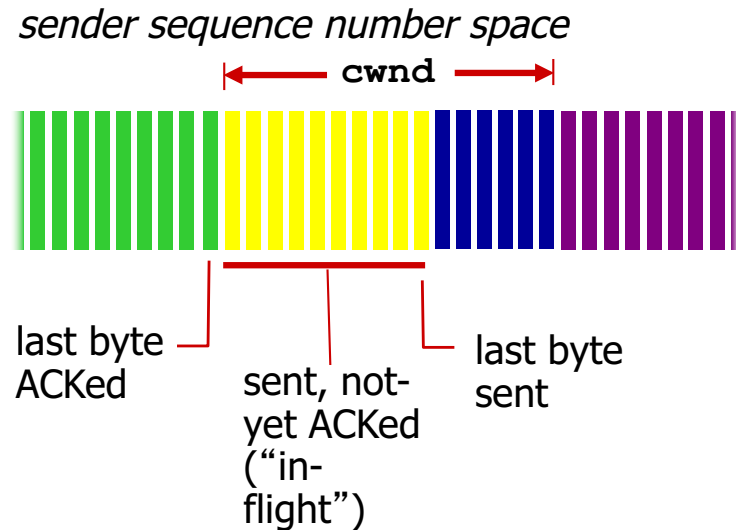
last byte sent

- 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

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

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
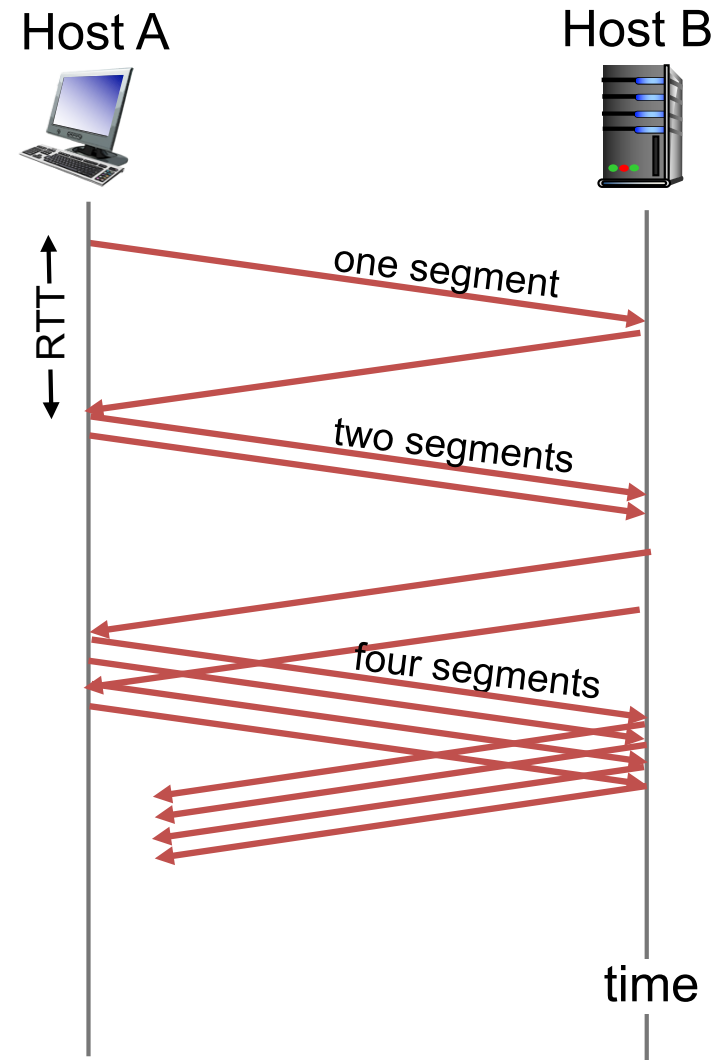
time

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)
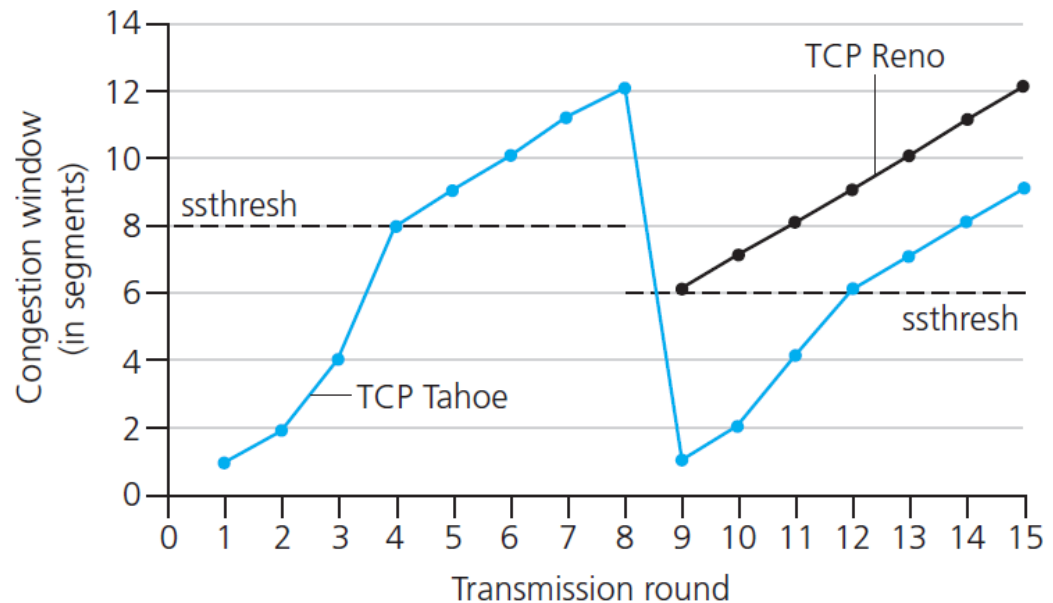
TCP: switching from slow start to CA

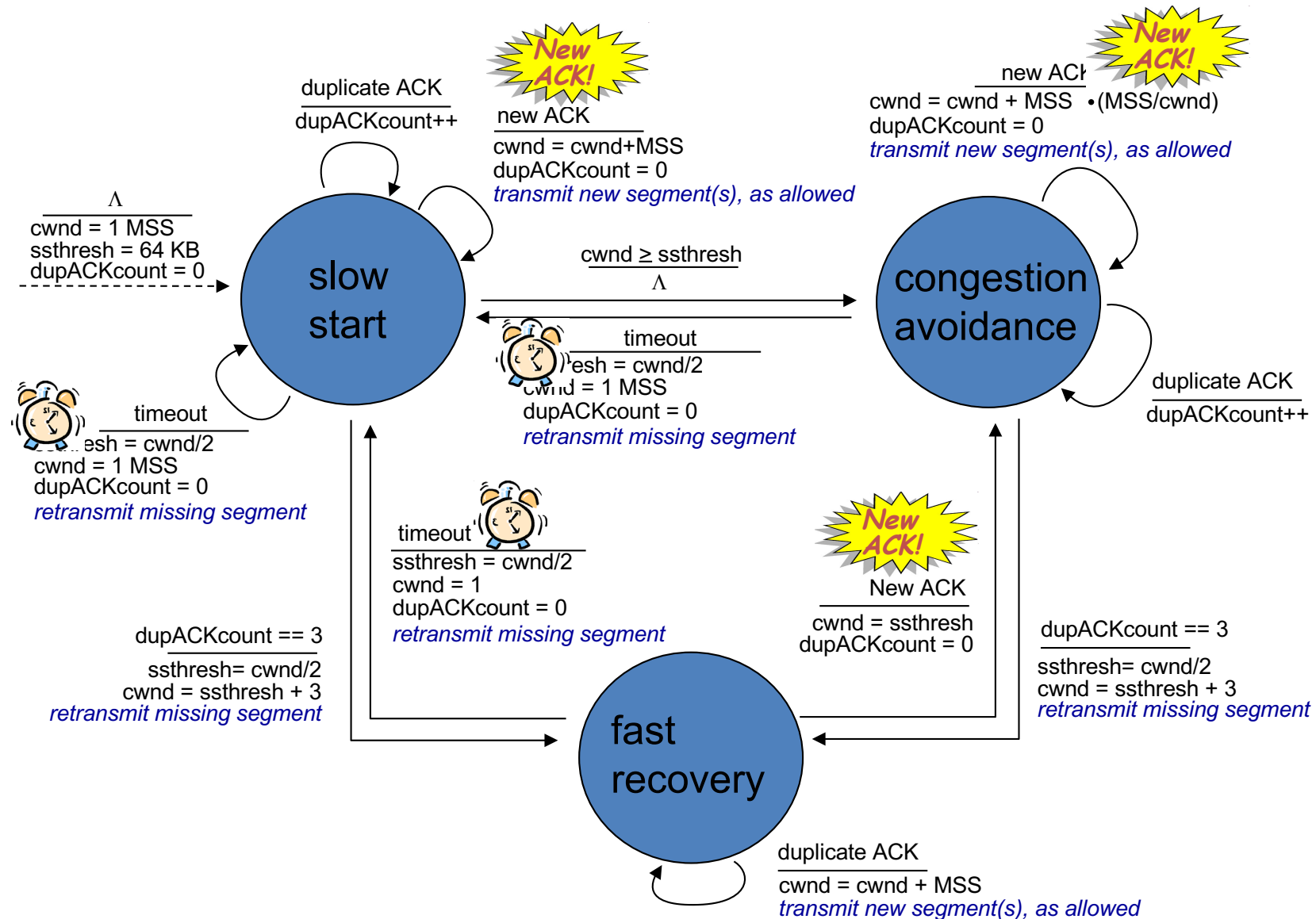**Q:** when should the exponential increase switch to linear?

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

Summary: TCP Congestion Control

slow start

duplicate ACK / dupACKcount++

Λ / cwnd = 1 MSS; ssthresh = 64 KB; dupACKcount = 0

New ACK! new ACK / cwnd = cwnd+MSS; dupACKcount = 0; *transmit new segment(s), as allowed*

timeout / ssthresh = cwnd/2; cwnd = 1 MSS; dupACKcount = 0; *retransmit missing segment*

cwnd ≥ ssthresh / Λ

timeout / ssthresh = cwnd/2; cwnd = 1 MSS; dupACKcount = 0; *retransmit missing segment*

congestion avoidance

New ACK! new ACK / cwnd = cwnd + MSS ·(MSS/cwnd); dupACKcount = 0; *transmit new segment(s), as allowed*

duplicate ACK / dupACKcount++

dupACKcount == 3 / ssthresh= cwnd/2; cwnd = ssthresh + 3; *retransmit missing segment*

timeout / ssthresh = cwnd/2; cwnd = 1; dupACKcount = 0; *retransmit missing segment*

dupACKcount == 3 / ssthresh= cwnd/2; cwnd = ssthresh + 3; *retransmit missing segment*

New ACK! New ACK / cwnd = ssthresh; dupACKcount = 0

fast recovery

duplicate ACK / cwnd = cwnd + MSS; *transmit new segment(s), as allowed*

TCP throughput

- avg. TCP throuput 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

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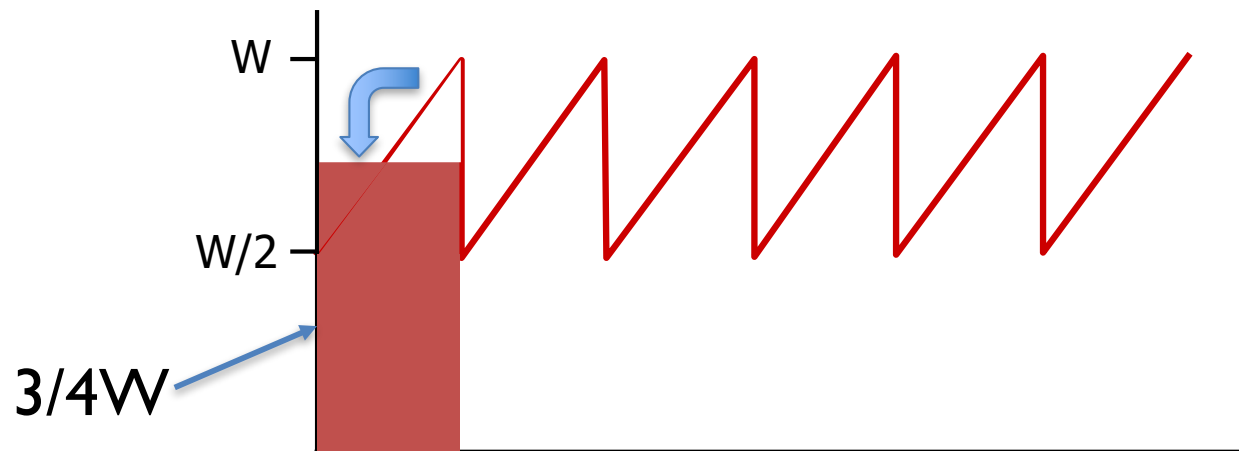
TCP Fairness

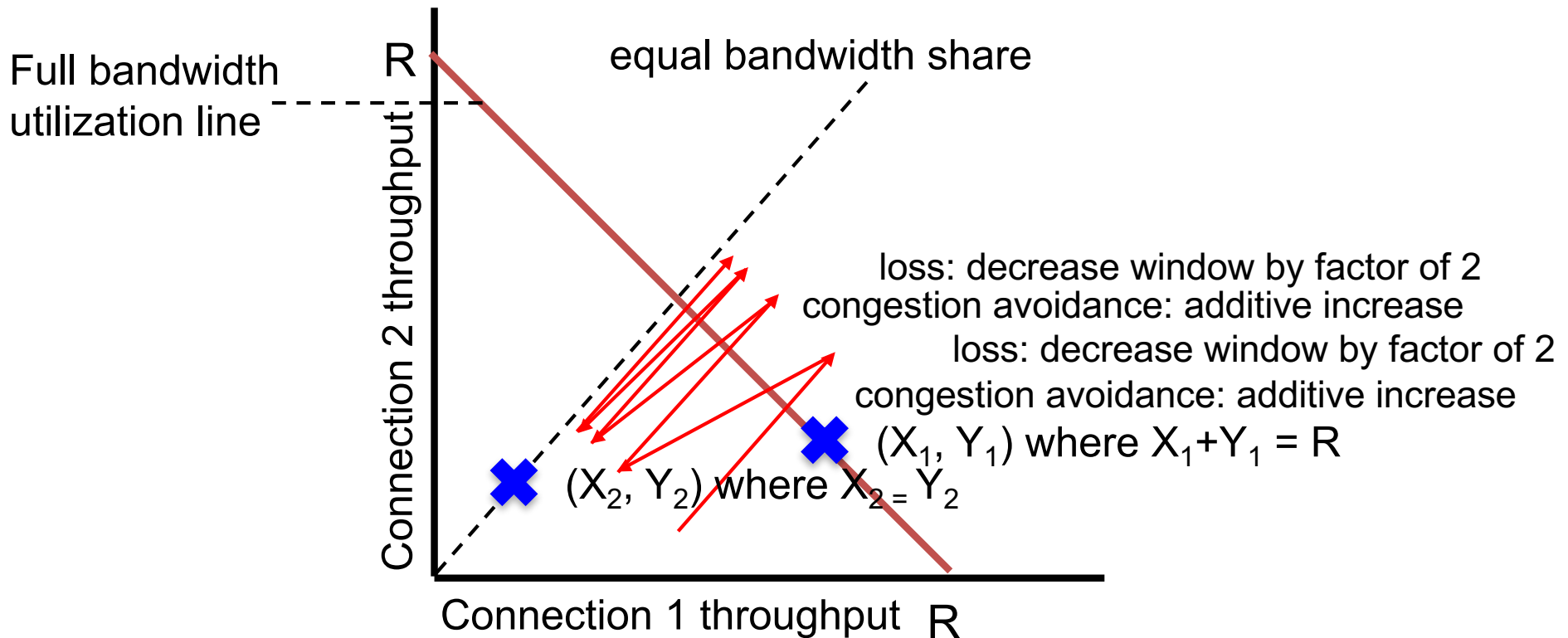*fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

TCP connection 1

TCP connection 2

bottleneck router capacity R
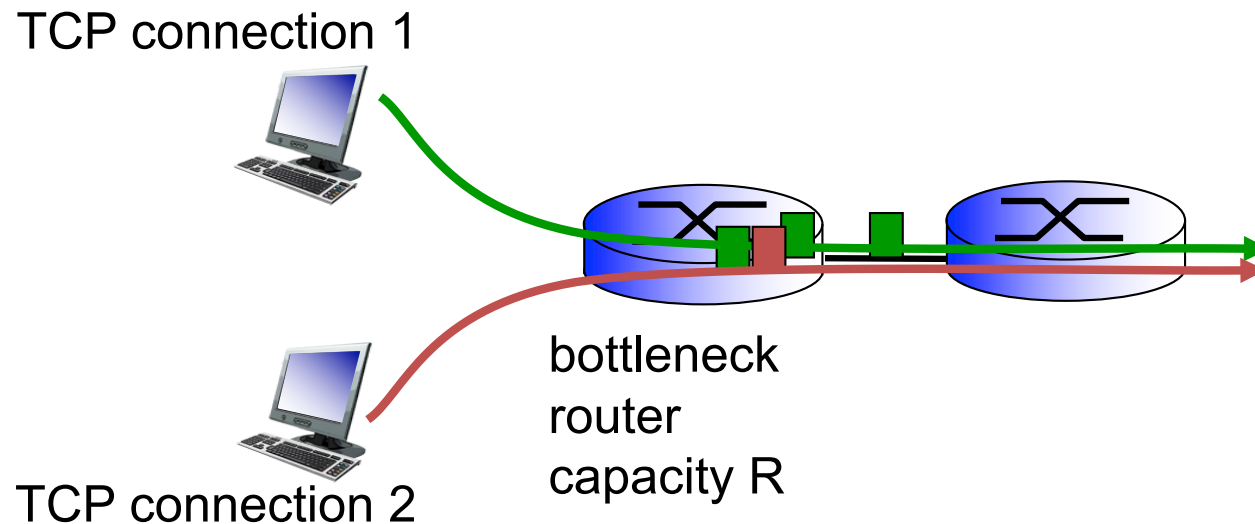
Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughout increases

- multiplicative decrease decreases throughput proportionally

Full bandwidth utilization line

R

equal bandwidth share

Connection 2 throughput

loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase

(X_1, Y_1) where $X_1 + Y_1 = R$

(X_2, Y_2) where $X_2 = Y_2$

Connection 1 throughput R

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