

# Chapter 3: Transport Layer – Part 2

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Miscellaneous

- HW2 deadline in a week

# Quick review

- Multiplexing and demultiplexing
  - UDP, DEMUX using only port number
  - TCP, 四元组, DEMUX using 4-tuple
- UDP
  - Connectionless
  - MUX and DEMUX
  - Checksum
- Design principles of reliable data transfer (RDT) protocol
  - RDT 1.0
  - RDT 2.0
  - RDT 3.0

# rdt3.0: channels with errors *and* loss

*Approach:* sender waits “reasonable” amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time



*timeout*

# Performance of rdt3.0 (stop-and-wait)

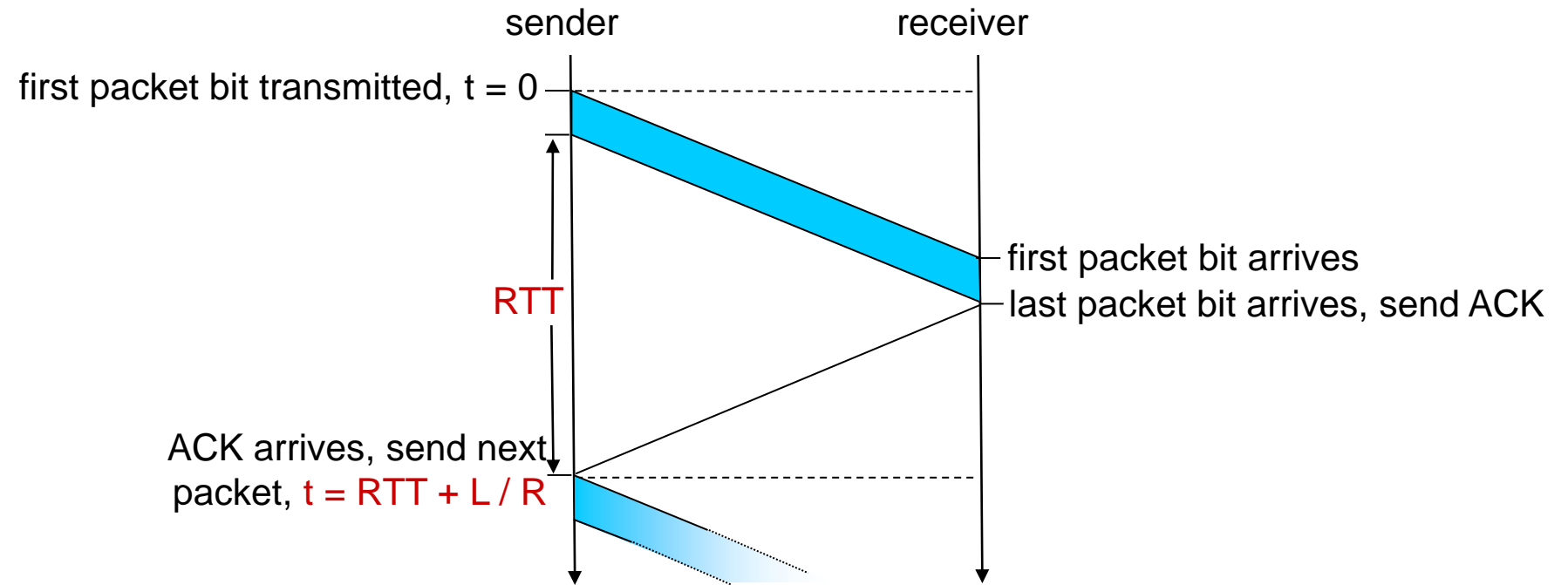


# Performance of rdt3.0 (stop-and-wait)

- $U_{sender}$ : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

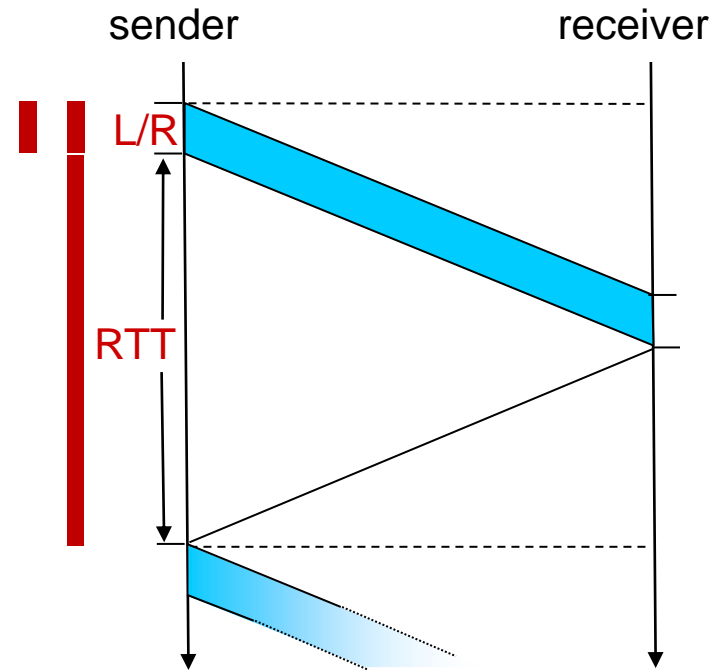
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

# rdt3.0: stop-and-wait operation



# rdt3.0: stop-and-wait operation

$$\begin{aligned}U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.00027\end{aligned}$$



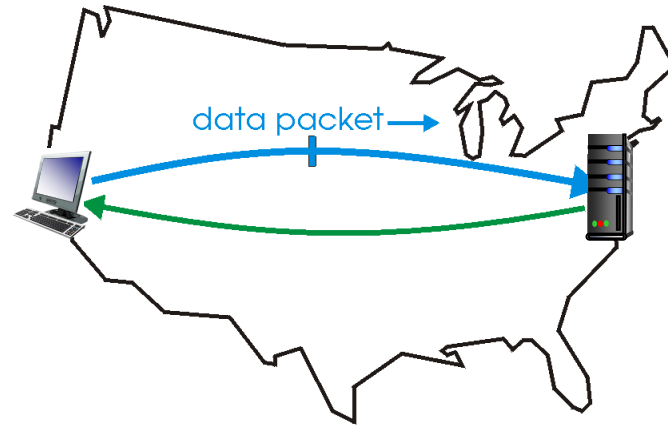
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)



# rdt3.0: pipelined protocols operation

**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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

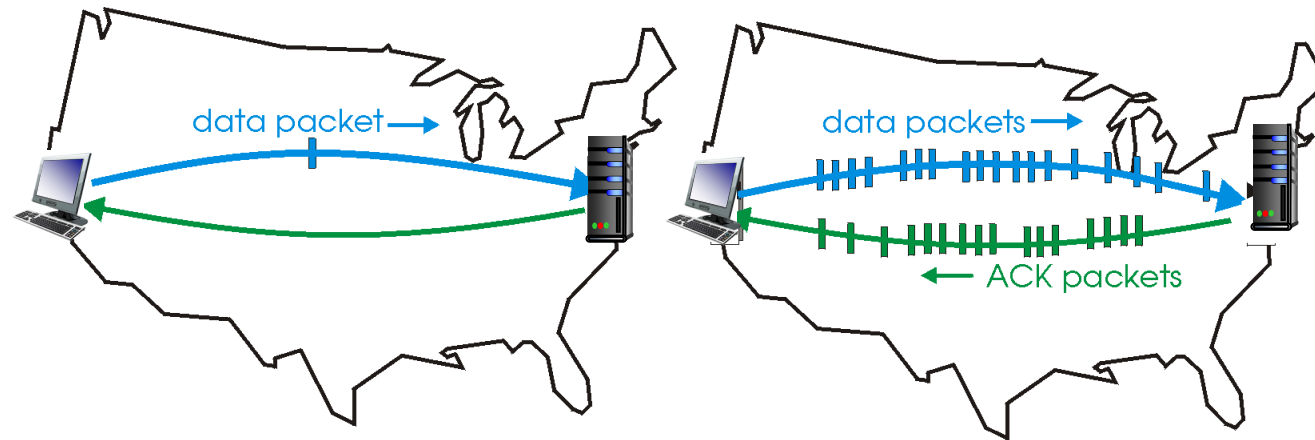


(a) a stop-and-wait protocol in operation

# rdt3.0: pipelined protocols operation

**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



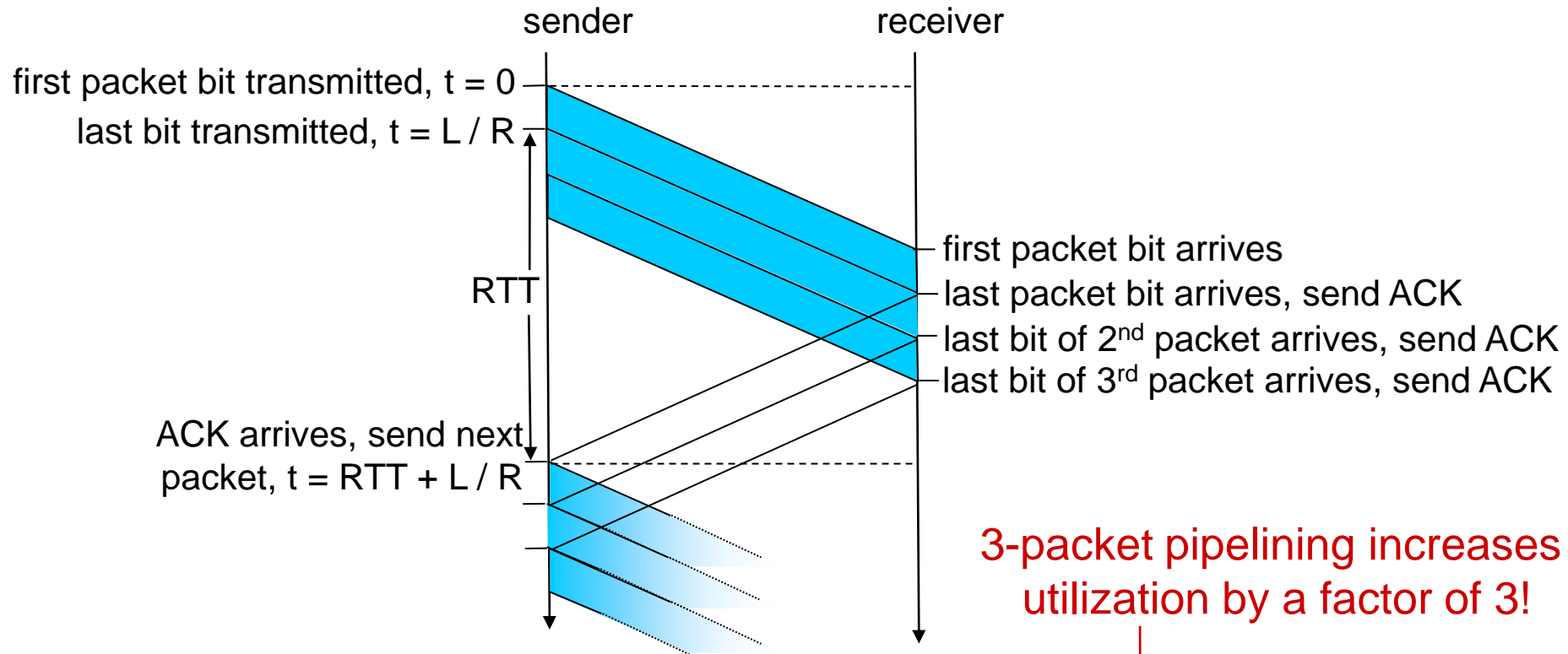
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

# rdt3.0: pipelined protocols operation



# Pipelining: increased utilization

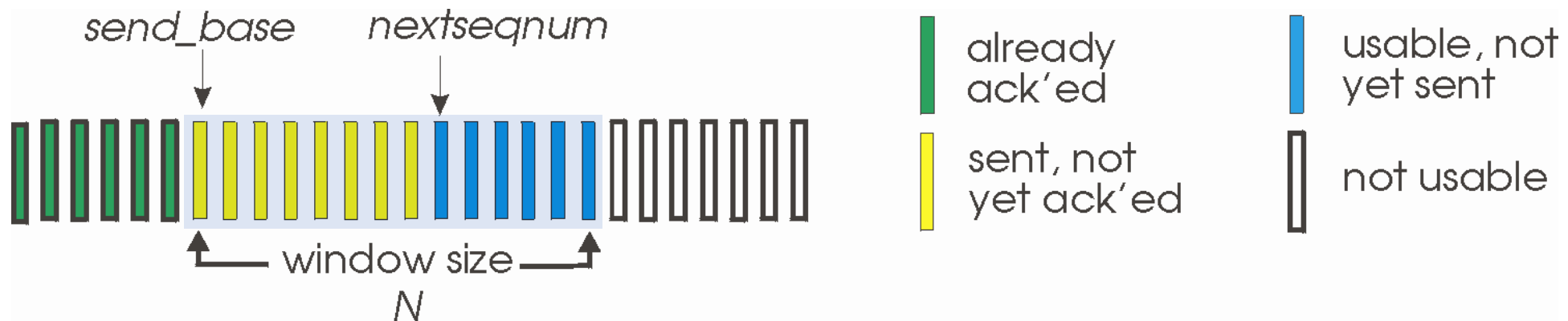


3-packet pipelining increases utilization by a factor of 3!

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

# Go-Back-N (回退N步) : sender

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

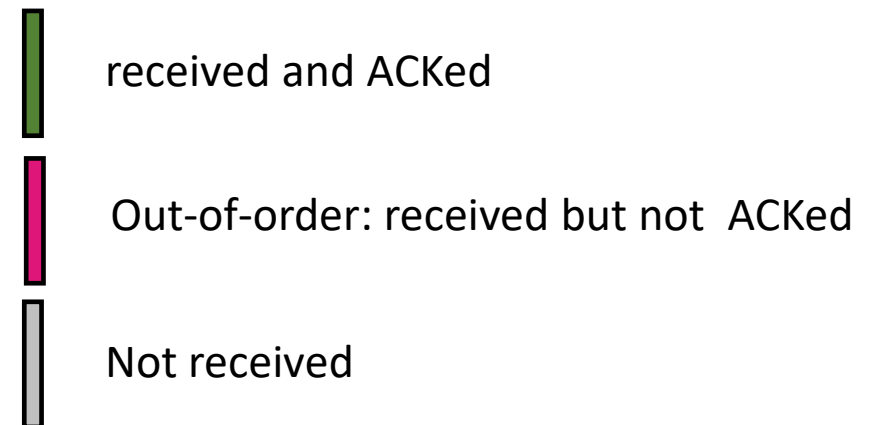
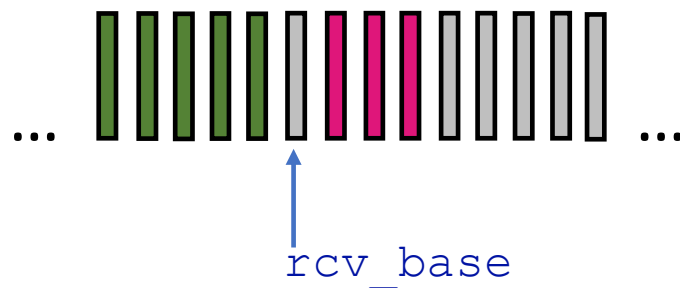


- **cumulative ACK:**  $ACK(n)$ : ACKs all packets up to, including seq #  $n$ 
  - on receiving  $ACK(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- **timeout(n):** retransmit packet  $n$  and all higher seq # packets in window

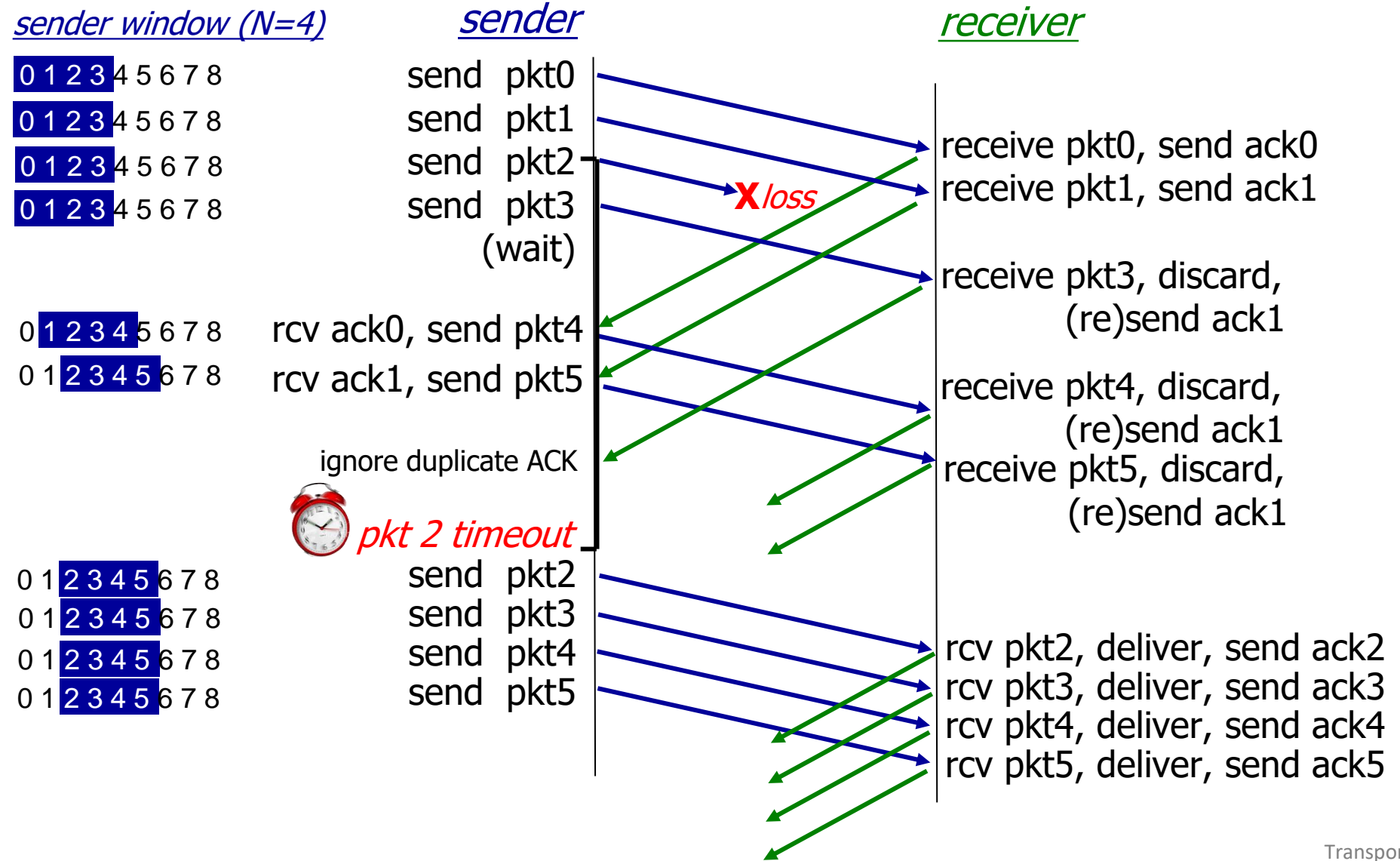
# Go-Back-N: receiver

- 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

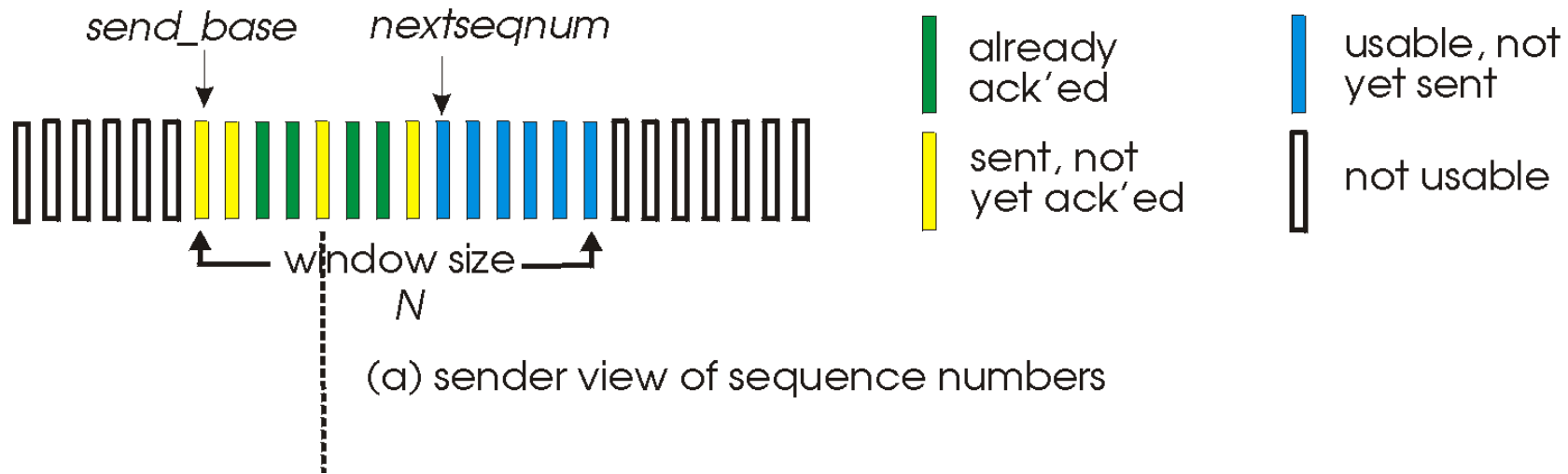


# Selective repeat (选择重传)

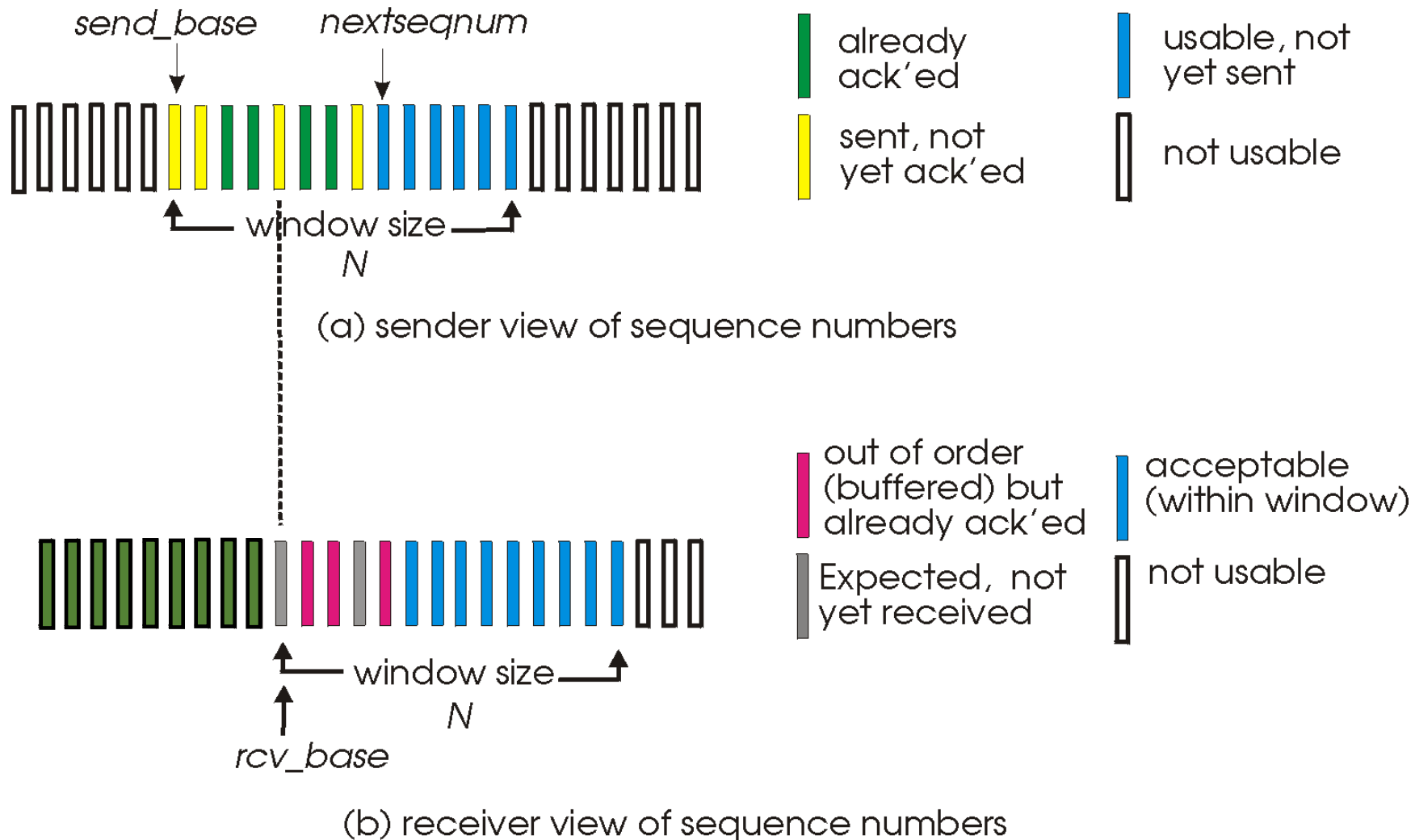
- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - $N$  consecutive seq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



# Selective repeat: sender, receiver windows



# Selective repeat: sender, 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  $n$  is 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

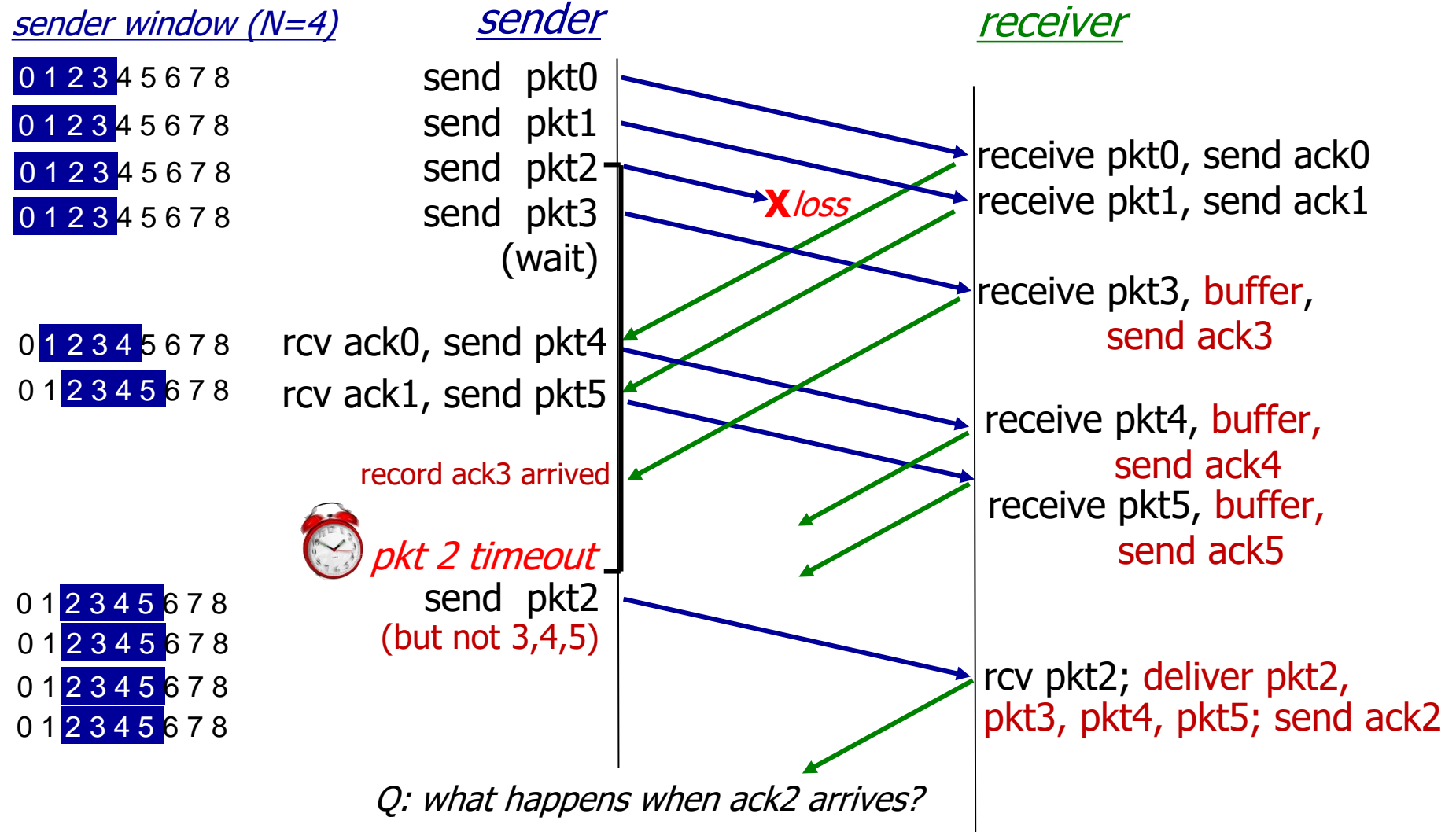
### packet $n$ in [rcvbase-N, rcvbase-1]

- ACK( $n$ )

### otherwise:

- ignore

# Selective Repeat in action



# 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
- Principles of congestion control
- TCP congestion control



# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

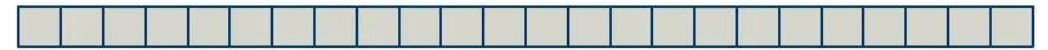
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream*:
  - no “message boundaries”
- full duplex (全双工) data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size

我爱你，爱着你，就像老鼠爱大米，巴拉巴拉... (省略一万字)

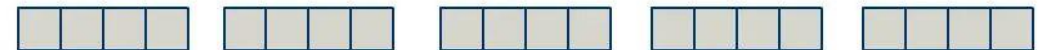


TCP

我管你是啥，到我这都是字节



我切切切切切!



1号

2号

3号

4号

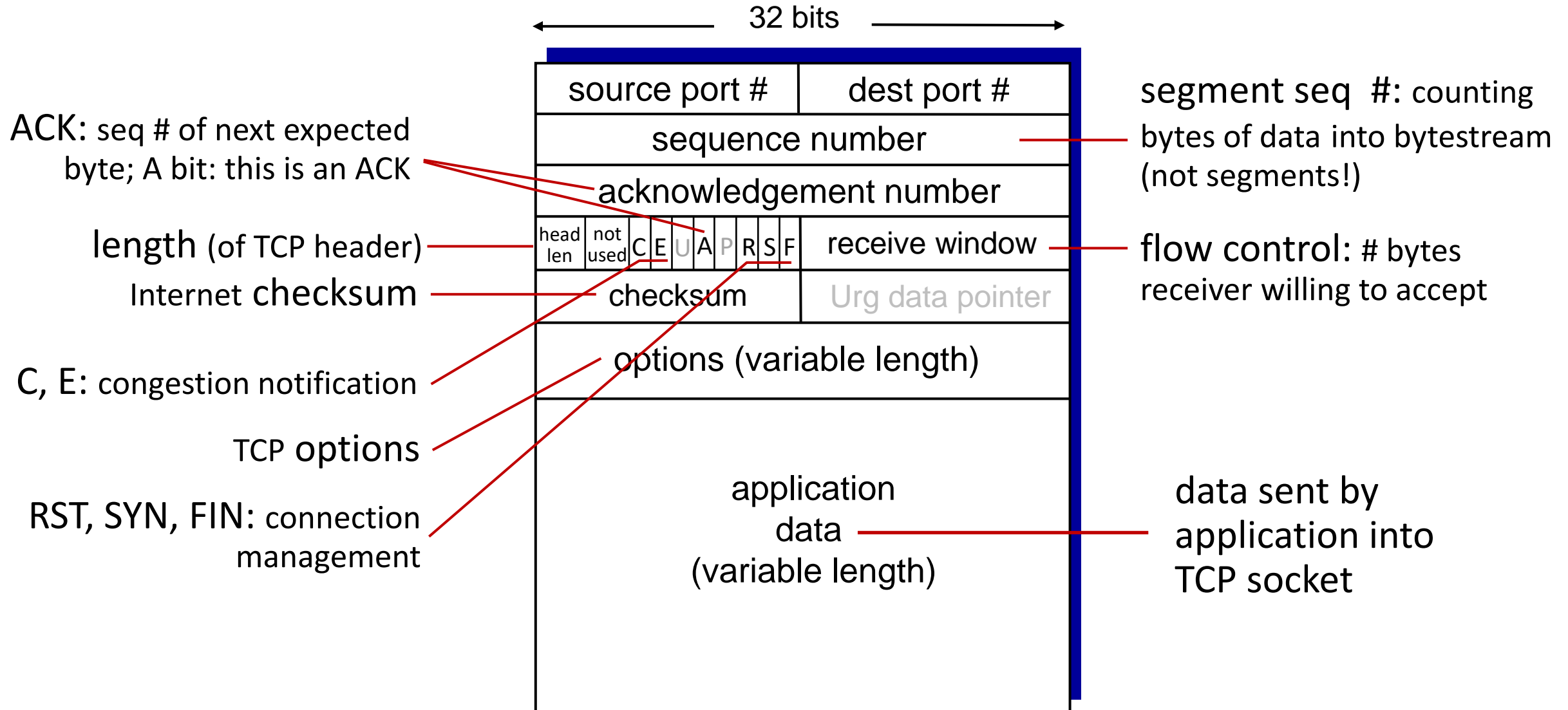
5号

# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

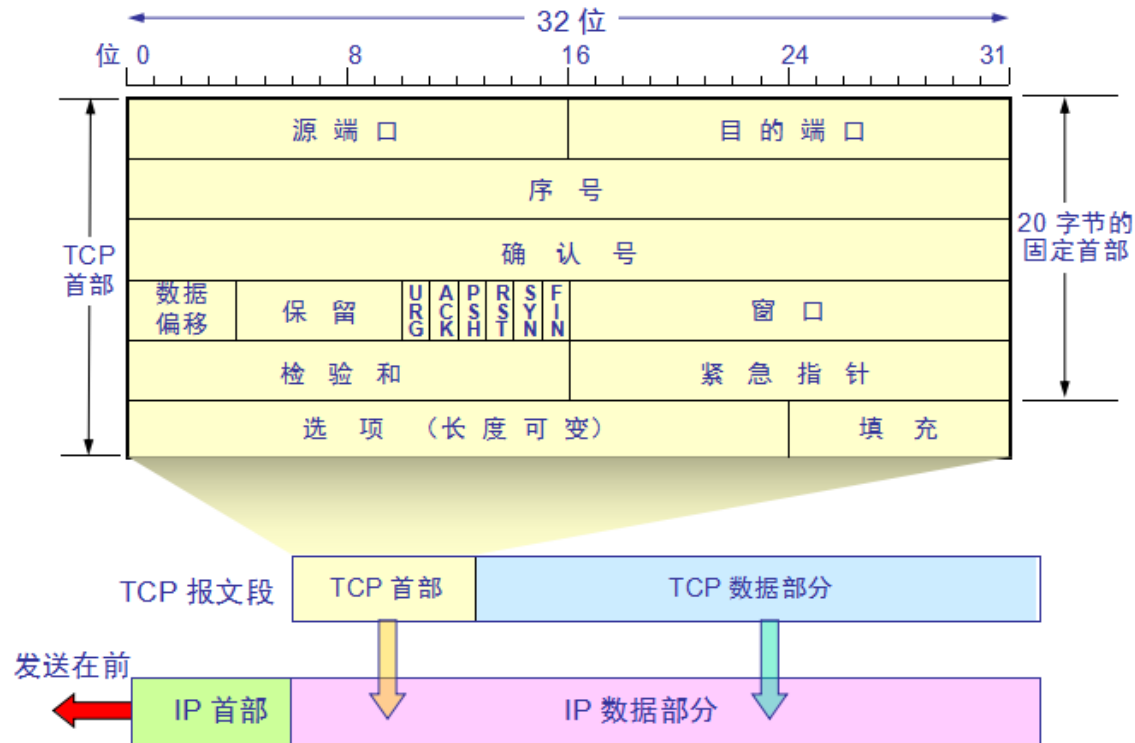
- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte stream*:**
  - no “message boundaries”
- **full duplex (全双工) data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- **cumulative ACKs**
- **pipelining:**
  - TCP congestion and flow control 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





# TCP segment structure



- The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so called **maximum transmission unit, MTU**)
- Ethernet and PPP link-layer protocols MTU: 1,500 bytes.
- A typical value of MSS: 1460 bytes

# TCP sequence numbers, ACKs

## Sequence numbers:

- byte stream “number” of first byte in segment’s data

## Acknowledgements:

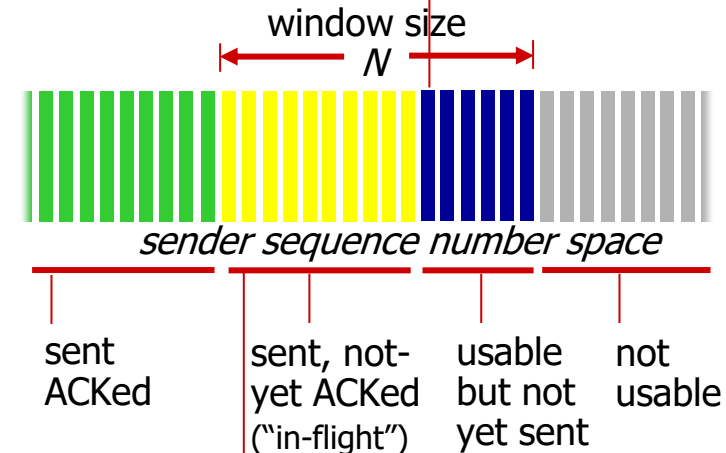
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

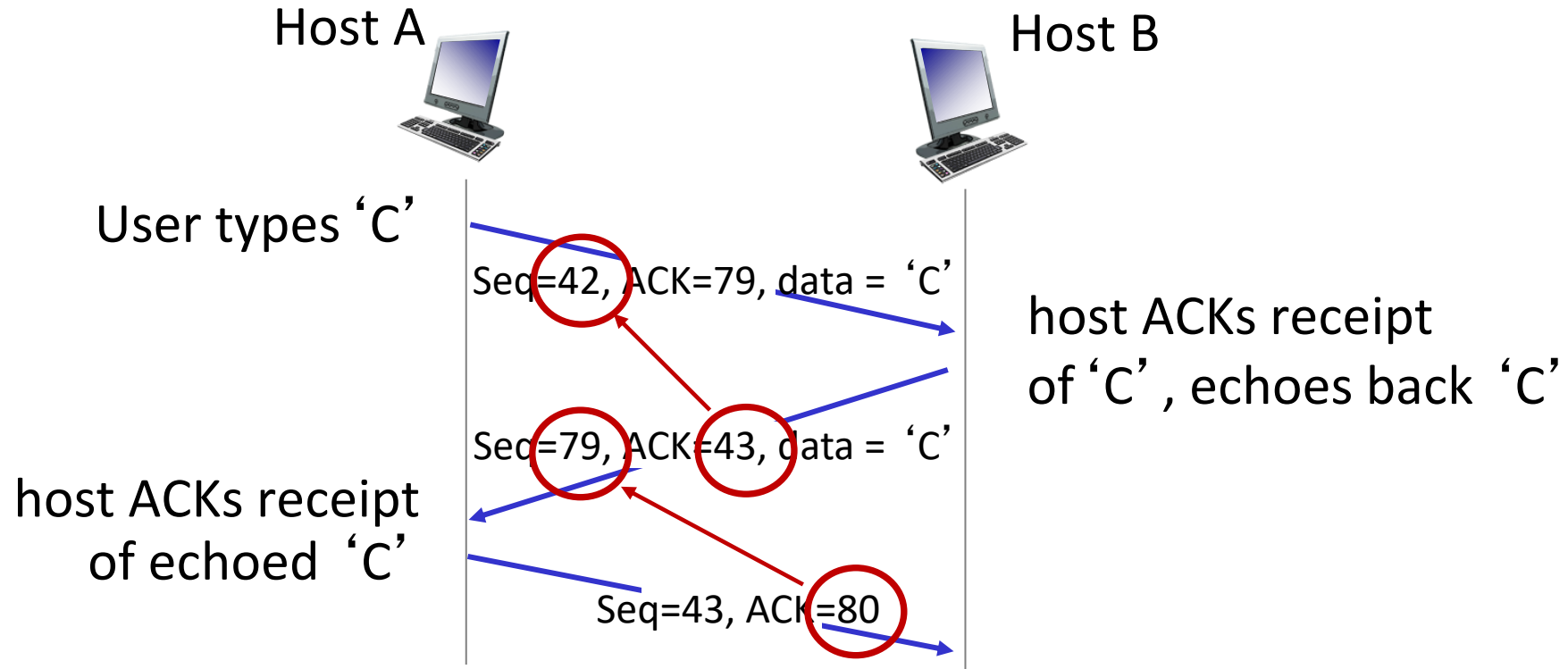
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

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

# 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

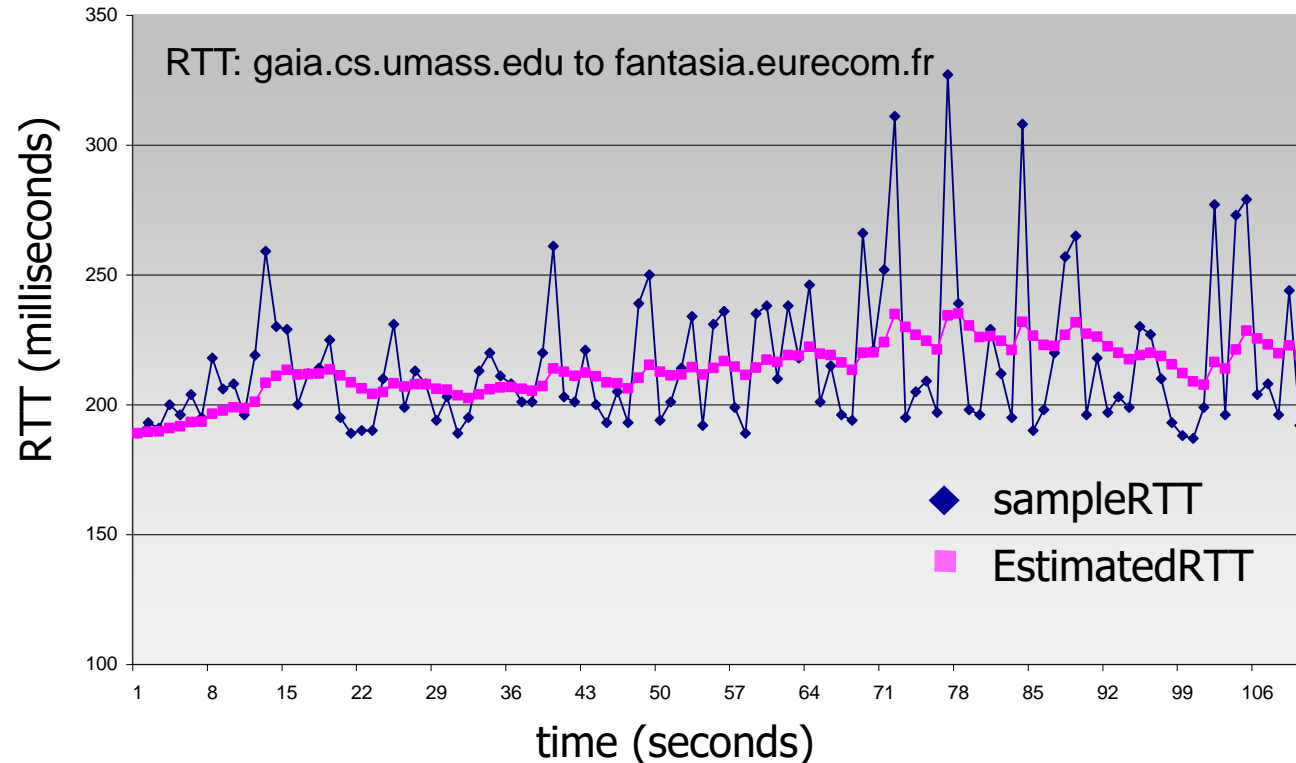
Q: how to estimate RTT?

- *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

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

- 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

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑  
estimated RTT

↑  
“safety margin”

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

$$\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

# 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**

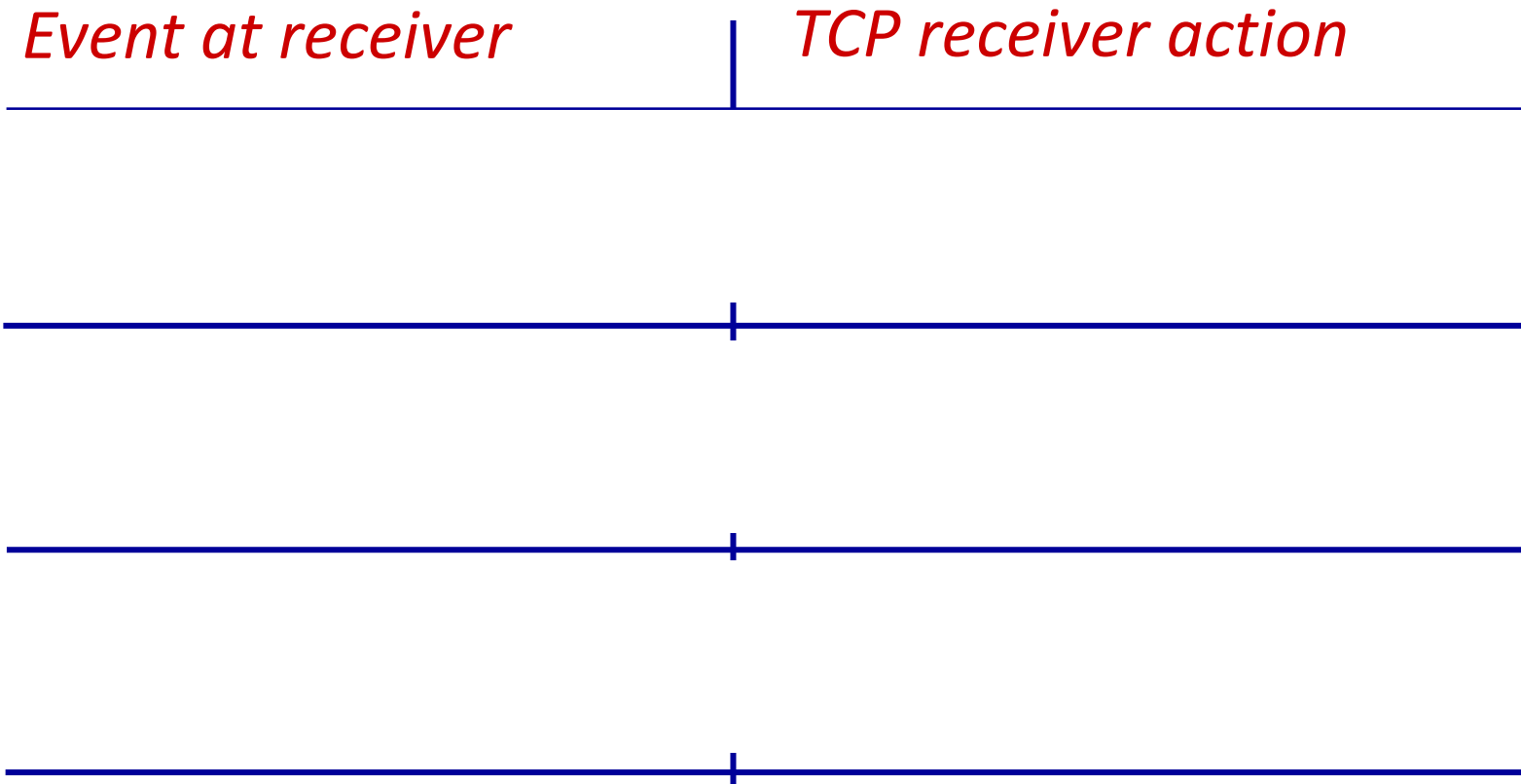
event: *timeout*

- 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]

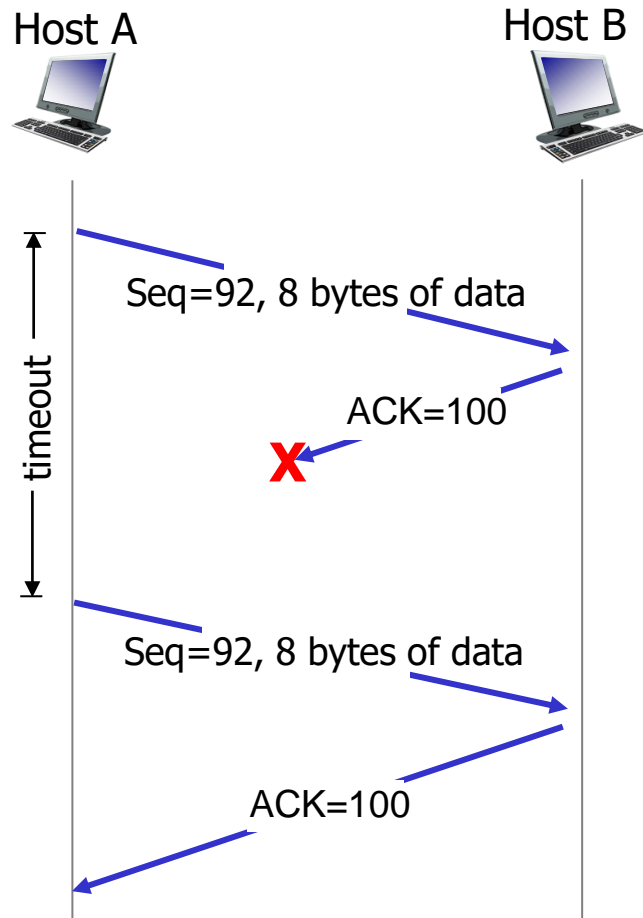




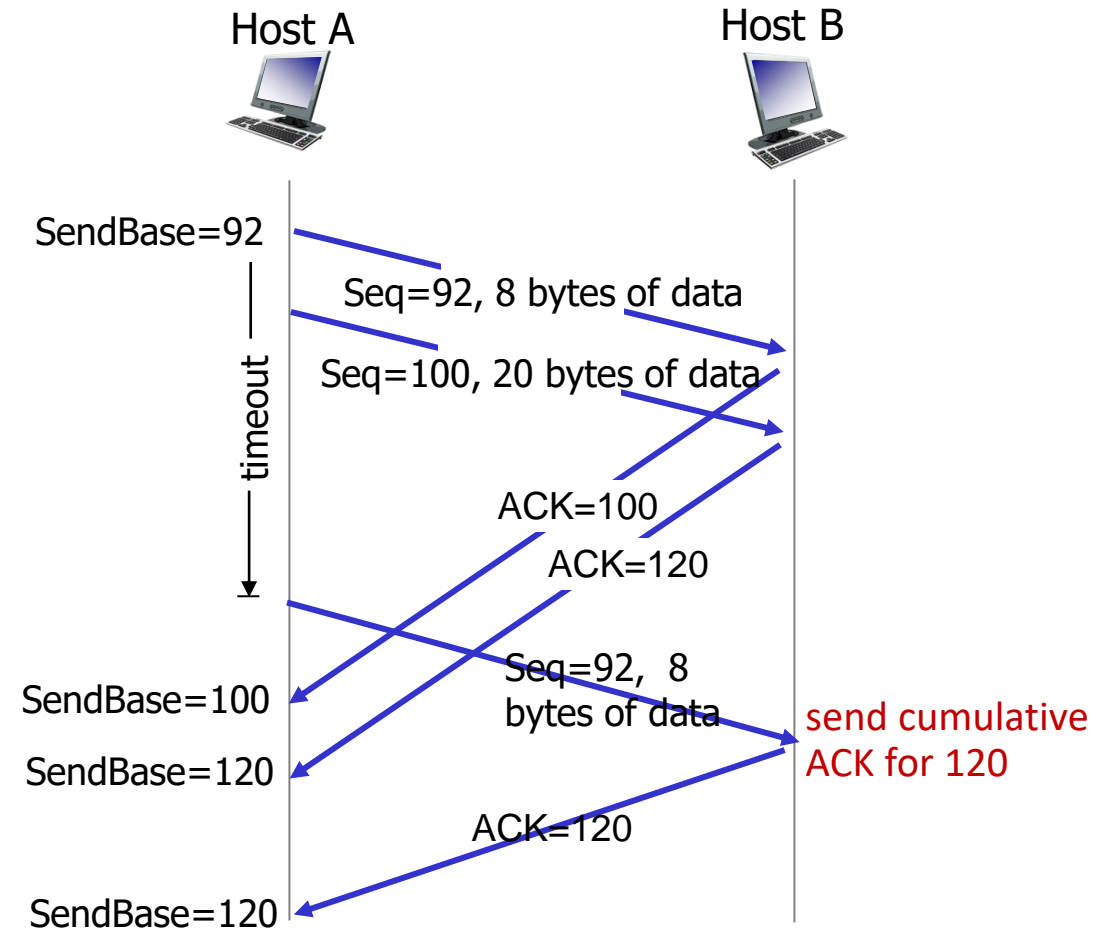
# TCP Receiver: ACK generation [RFC 5681]

<i>Event at receiver</i>	<i>TCP receiver action</i>
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 <i>duplicate ACK</i> , indicating seq. # of next expected byte

# TCP: retransmission scenarios

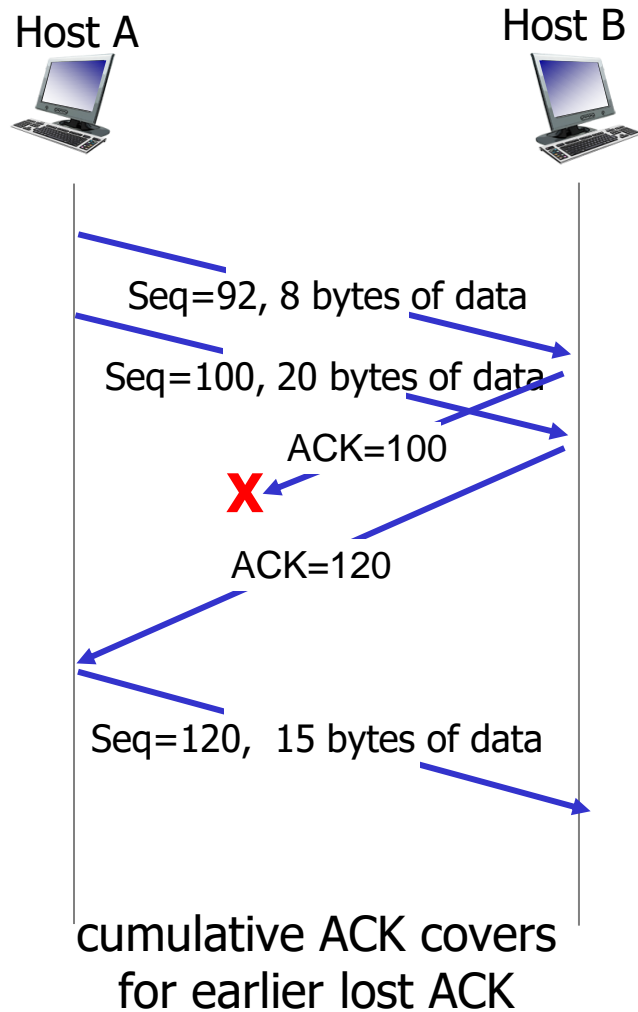


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



# TCP fast retransmit

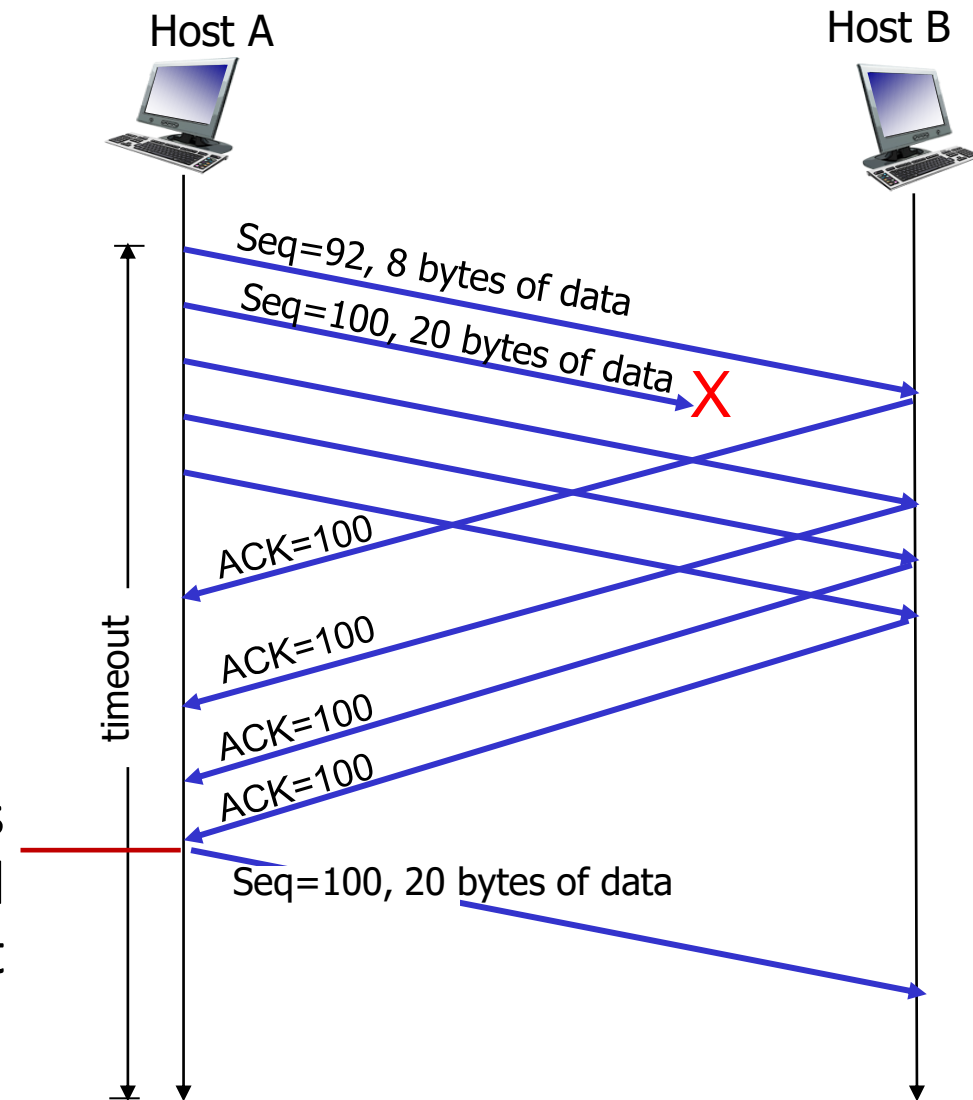
## *TCP fast retransmit*

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!



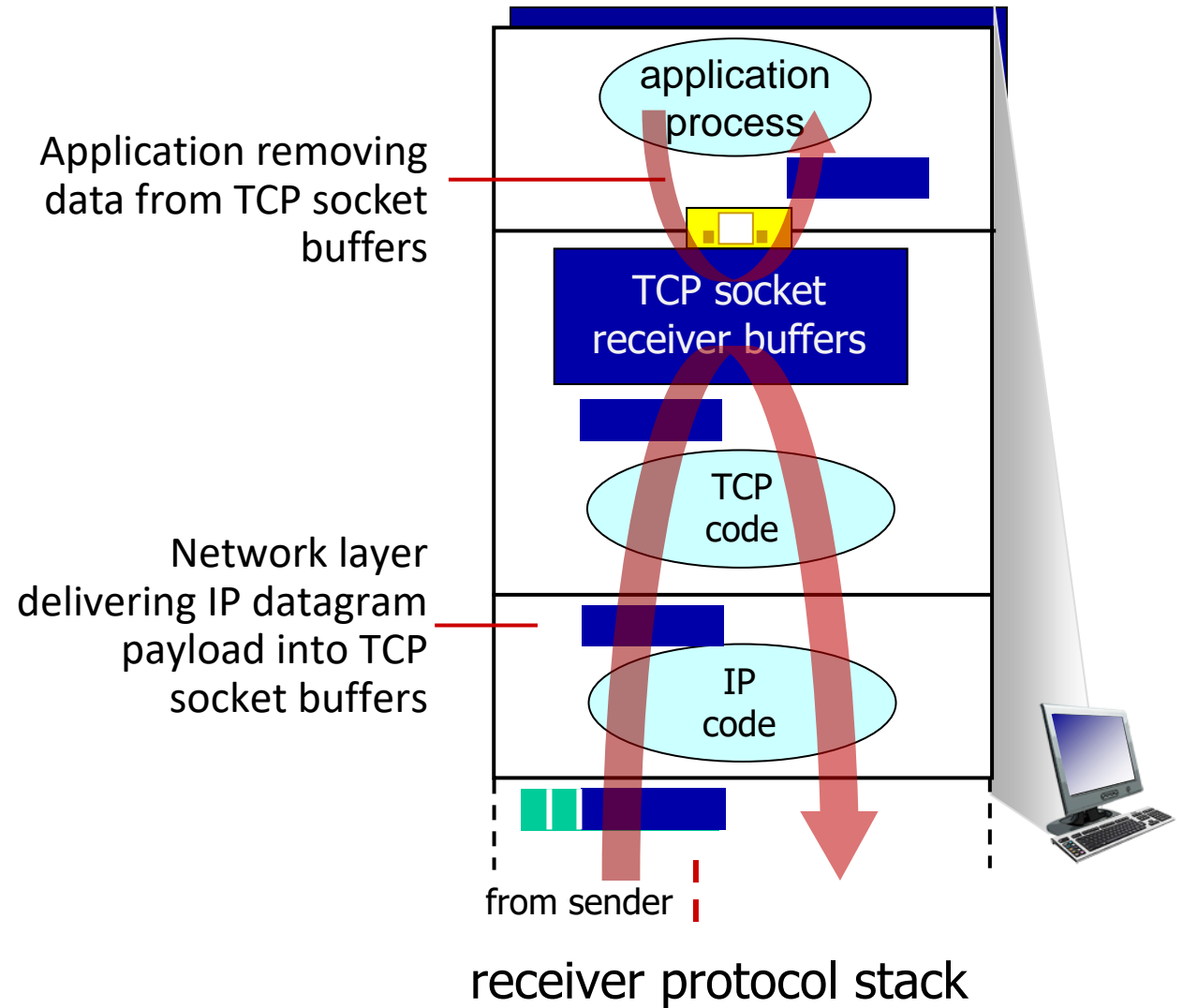
# Chapter 3: roadmap

- 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
- Principles of congestion control
- TCP congestion control



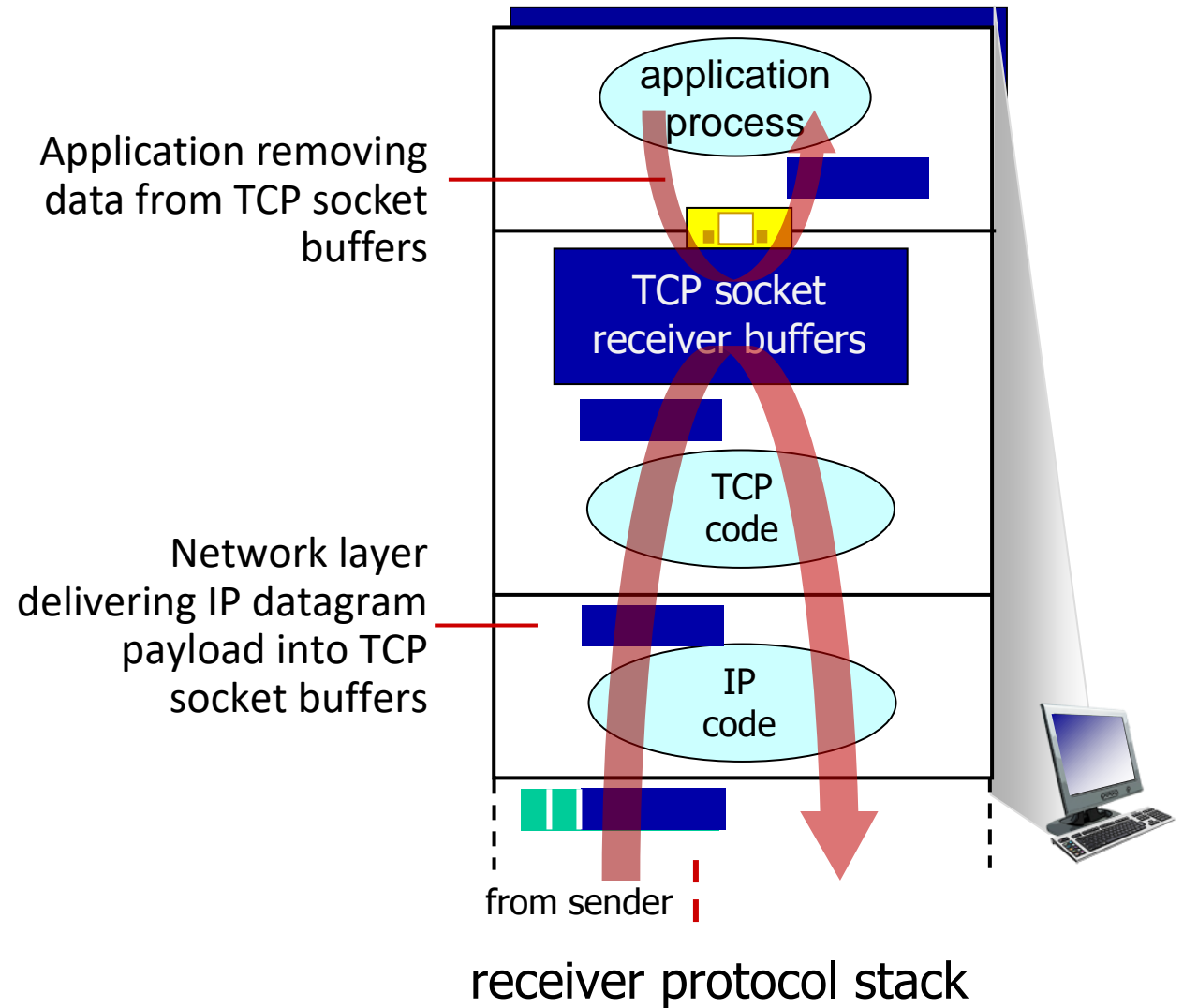
# TCP flow control (流量控制)

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?

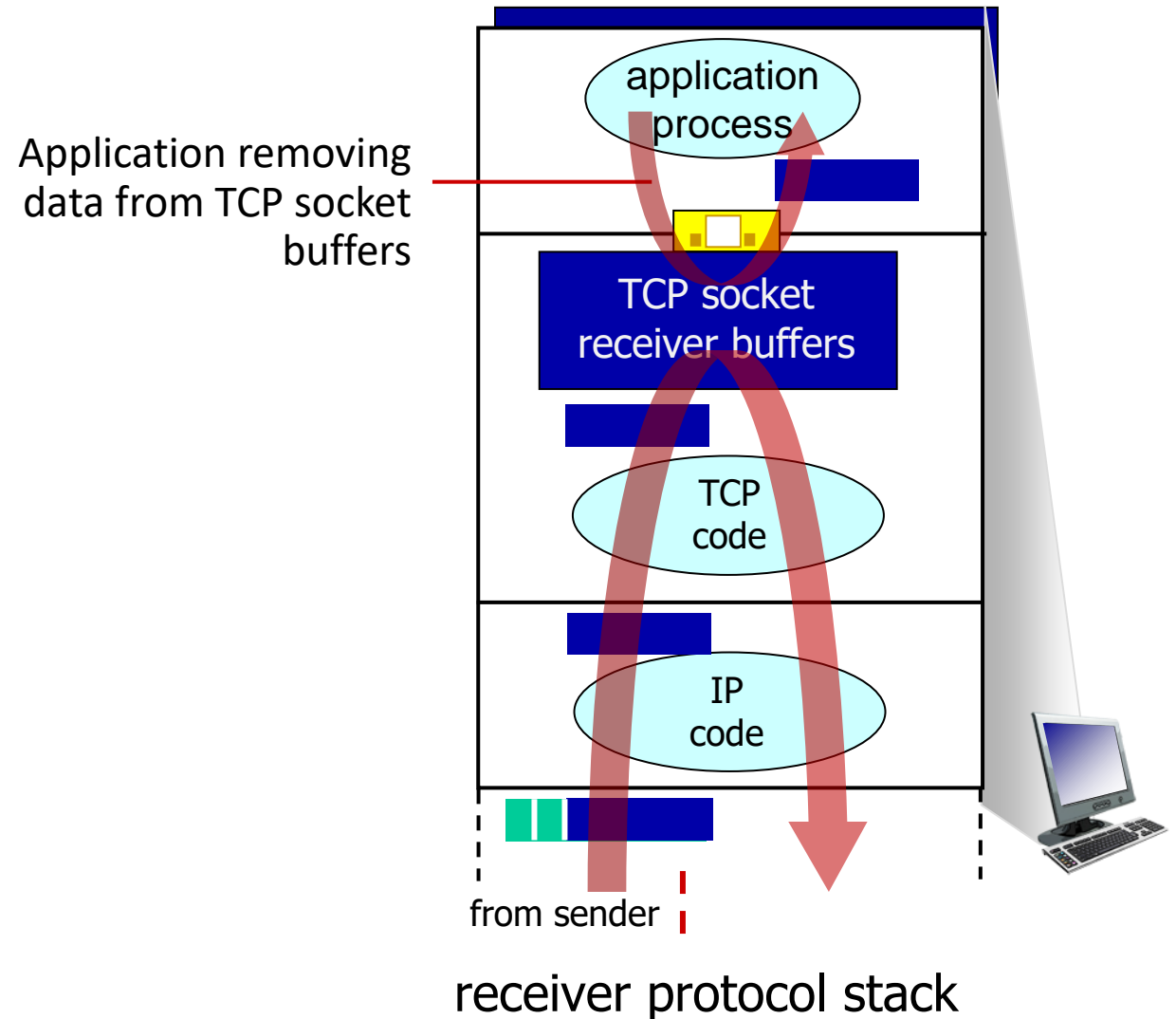


# TCP flow control

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

## —flow control—

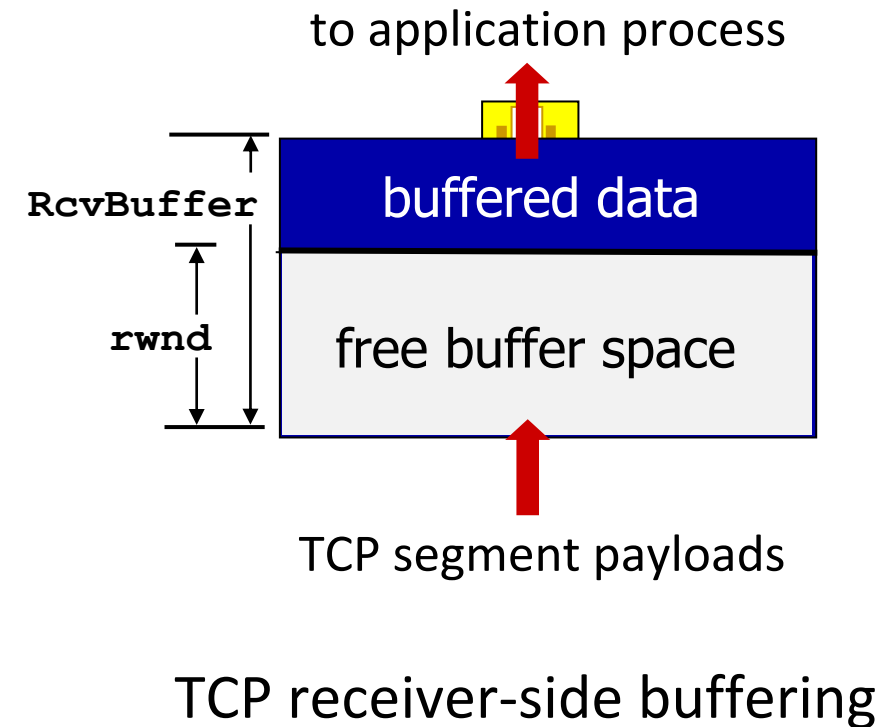
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast





# TCP flow control

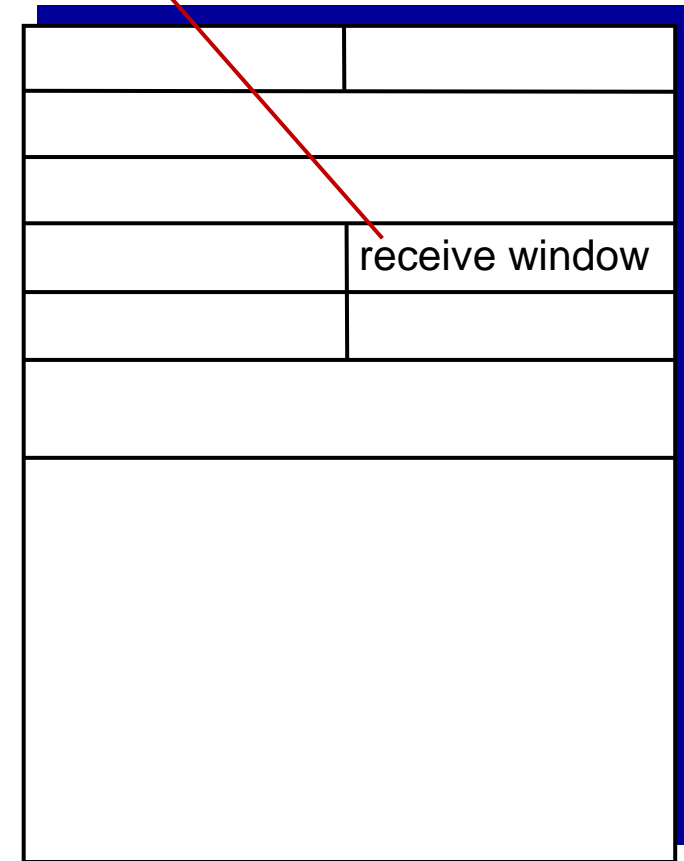
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive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



# TCP 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

flow control: # bytes receiver willing to accept

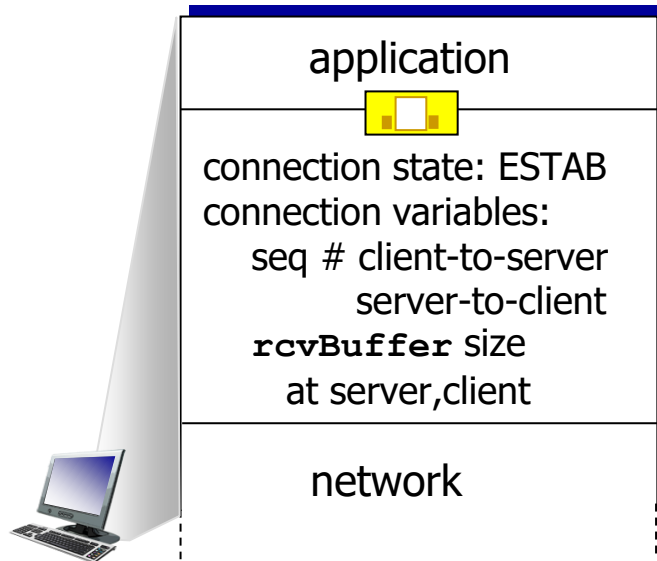


TCP segment format

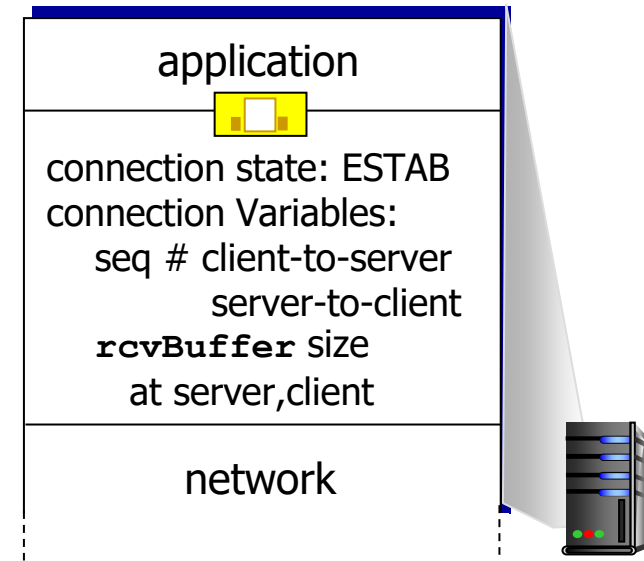
# TCP connection management

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)



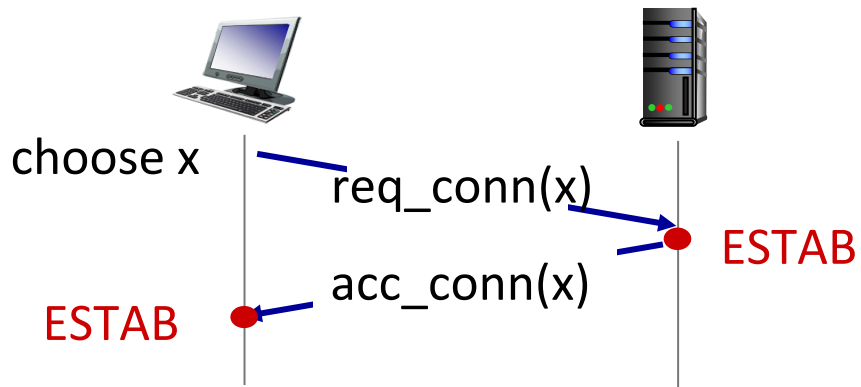
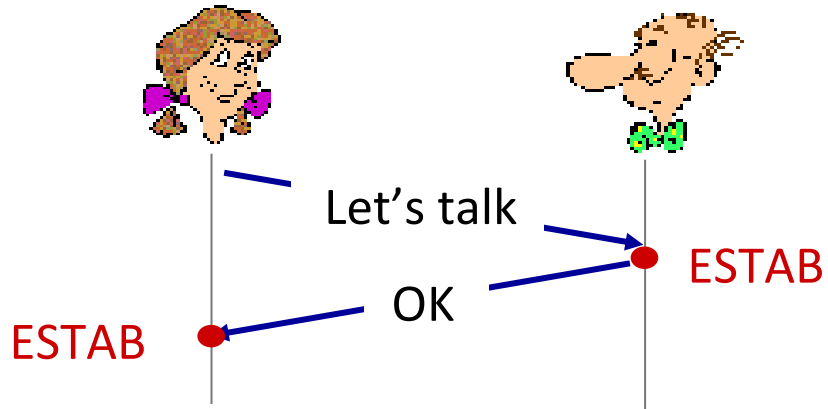
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

# Agreeing to establish a connection

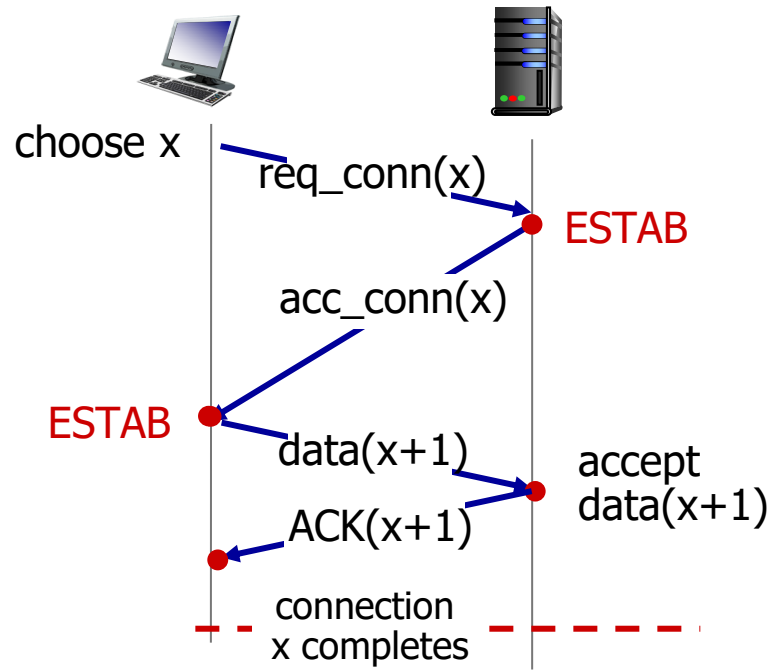
2-way handshake:



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

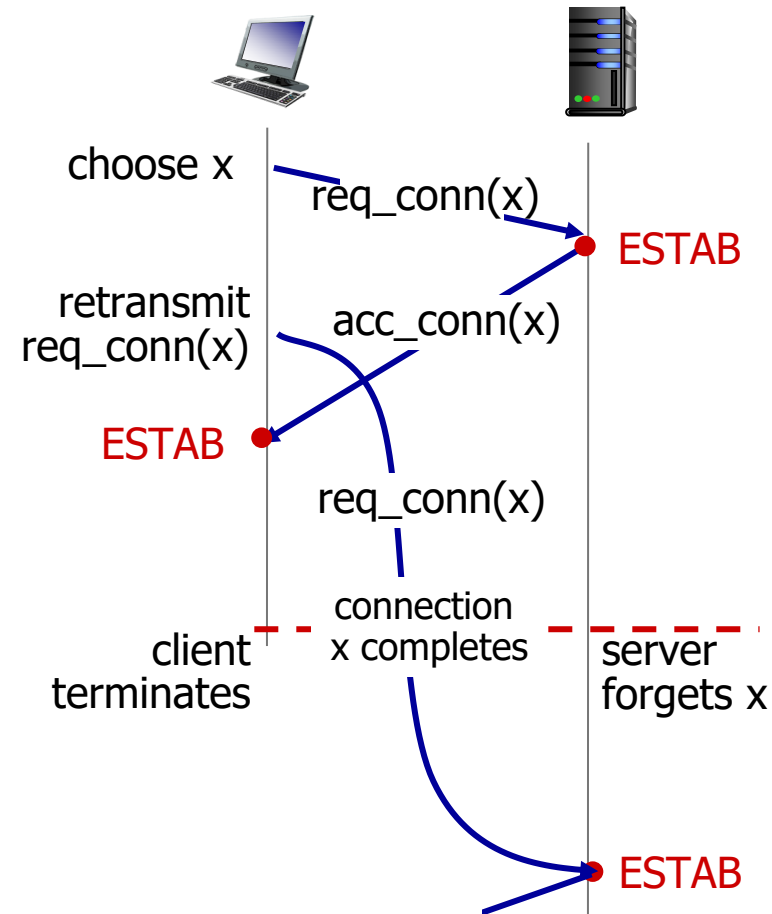
# 2-way handshake scenarios




No problem!

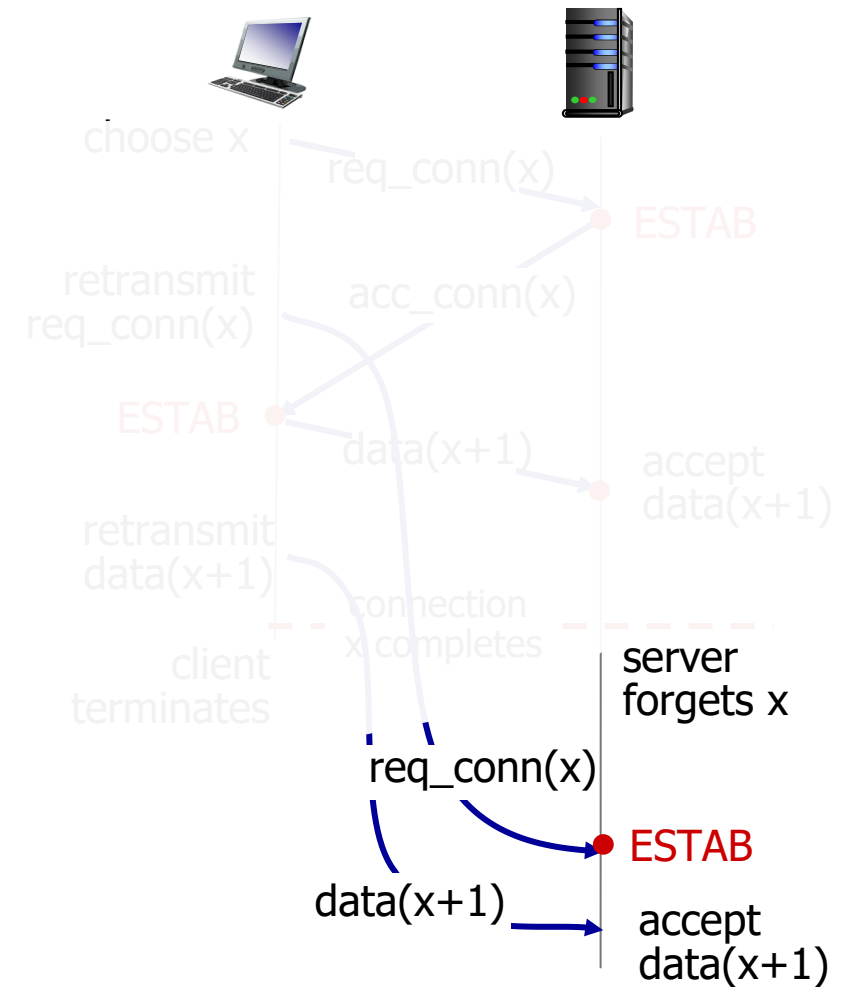


# 2-way handshake scenarios



 Problem: half open connection! (no client)

# 2-way handshake scenarios



 Problem: dup data accepted!

# TCP 3-way handshake (三次握手)

## Server state

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

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

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

received ACK(y)  
indicates client is live

LISTEN

SYN RCVD

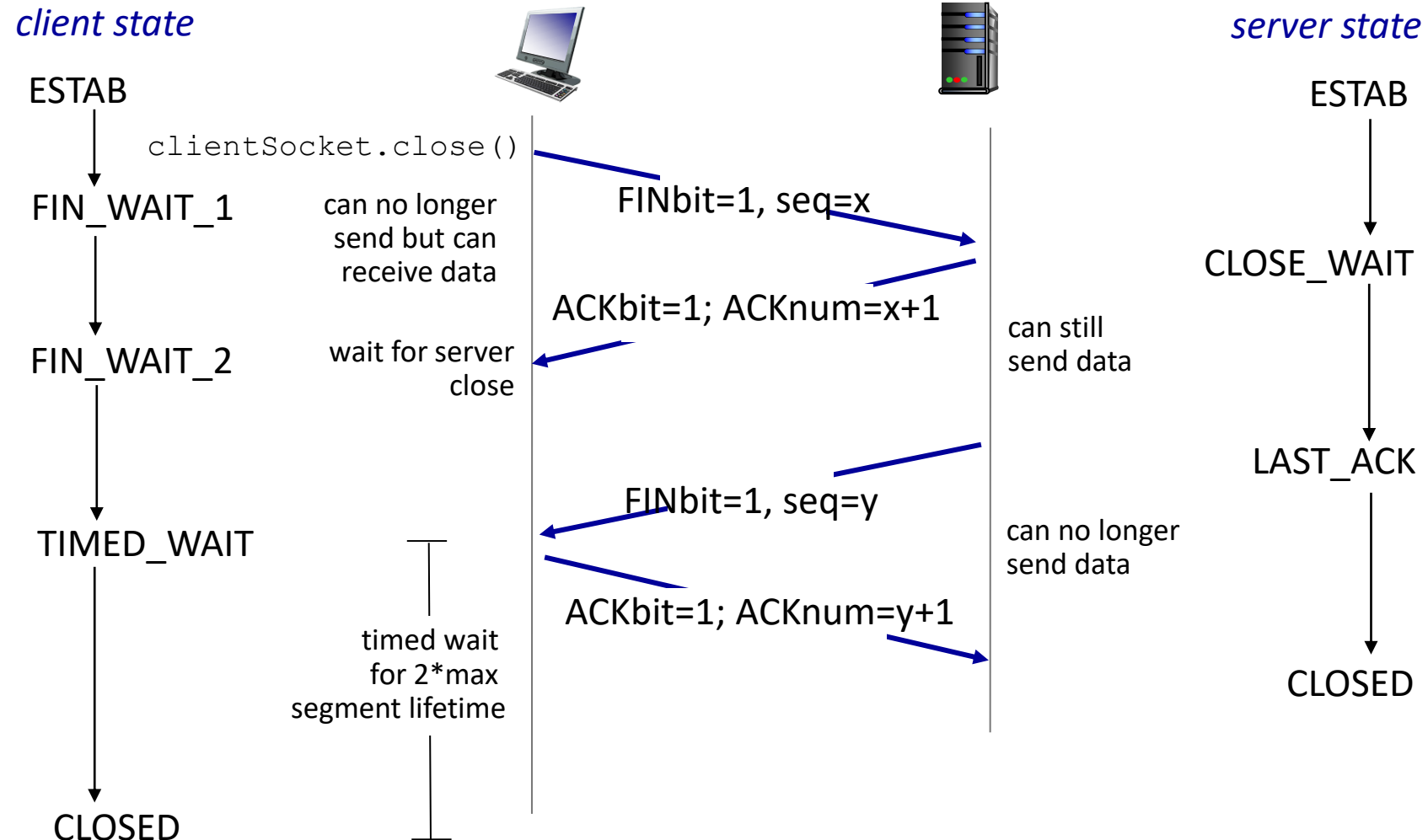
ESTAB



# A human 3-way handshake protocol



# TCP: closing a connection

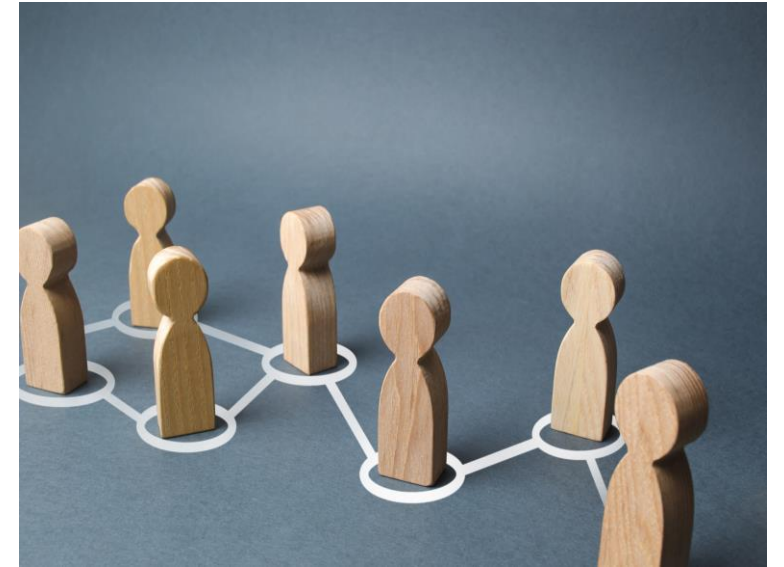


# Closing a TCP connection

- 4-way handshake (四次挥手)
- 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

# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality



# 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!
- a top-10 problem!



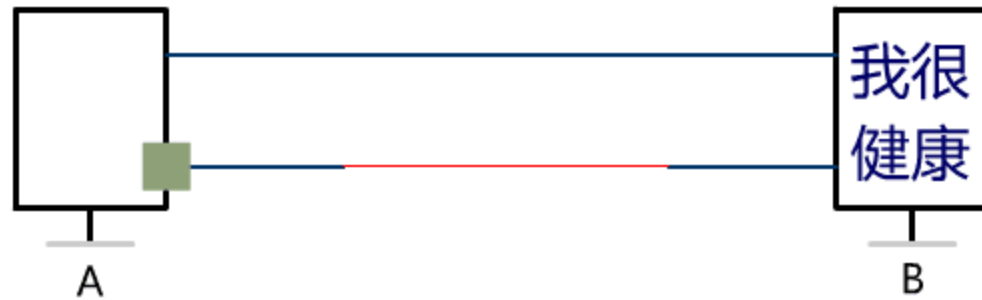
**congestion control:**

too many senders,  
sending too fast



**flow control:** one sender  
too fast for one receiver

# Principles of congestion control

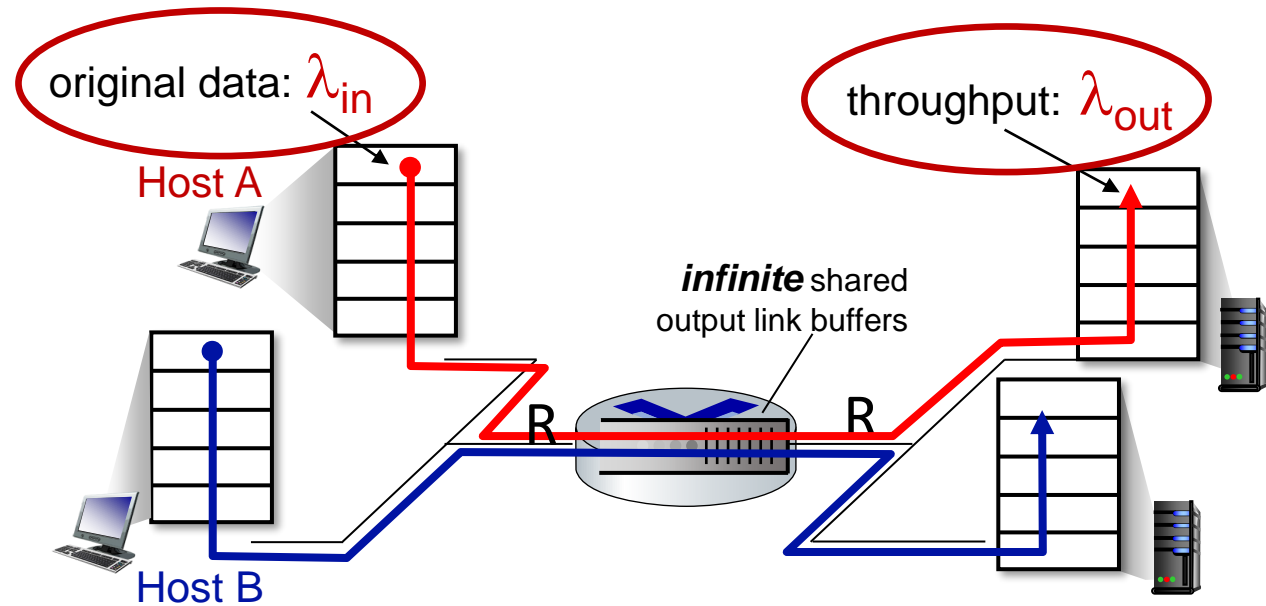




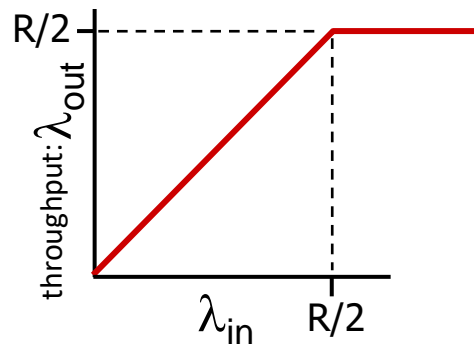
# Causes/costs of congestion: scenario 1

Simplest scenario:

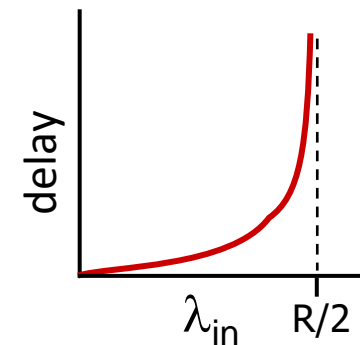
- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed



**Q:** What happens as arrival rate  $\lambda_{in}$  approaches  $R/2$ ?



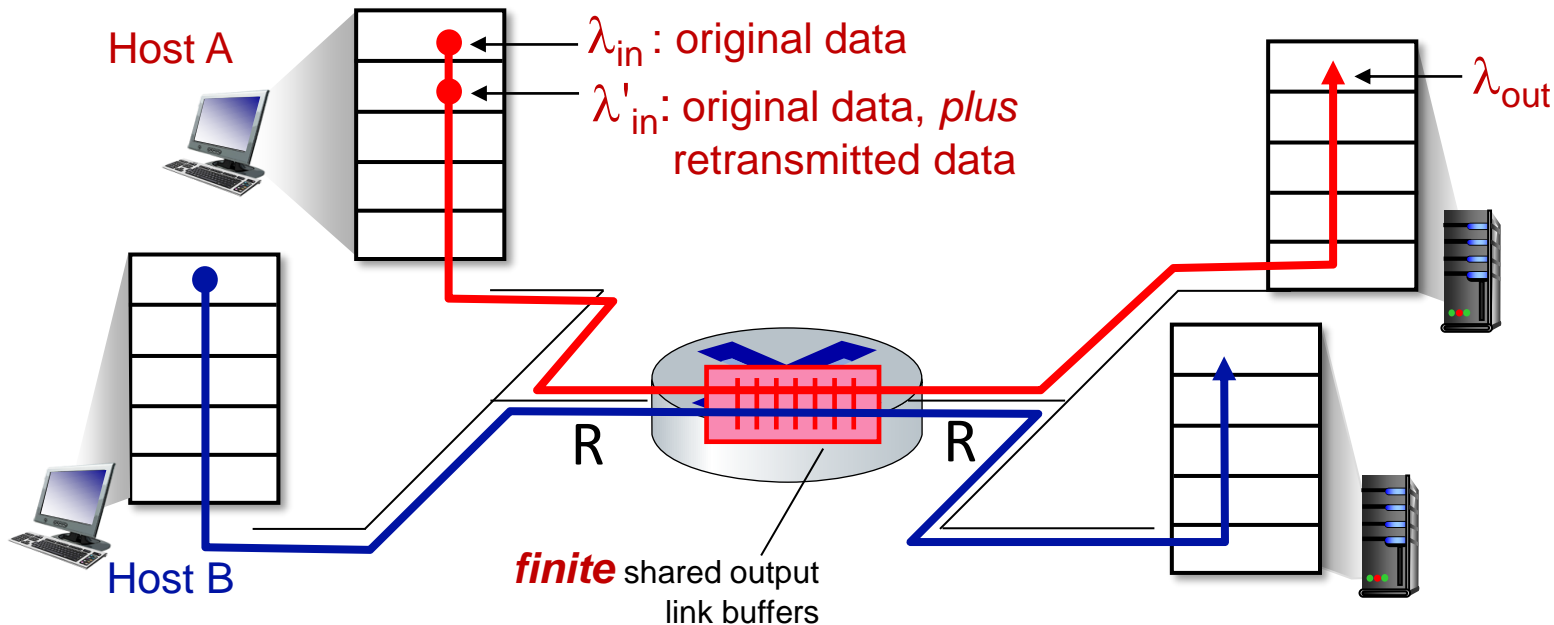
maximum per-connection throughput:  $R/2$



large delays as arrival rate  $\lambda_{in}$  approaches capacity

# Causes/costs of congestion: scenario 2

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

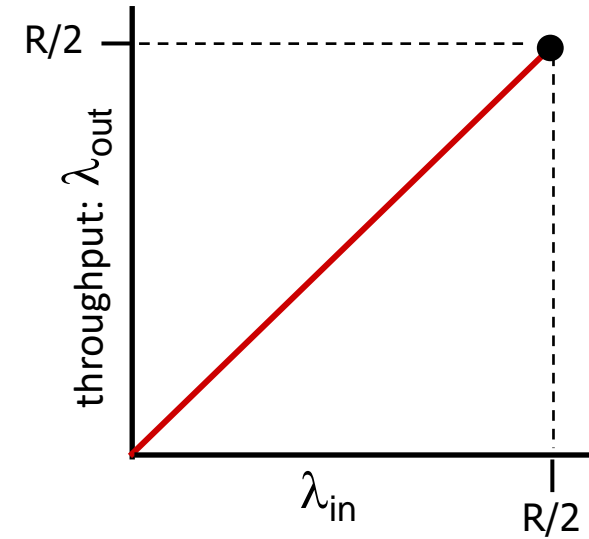
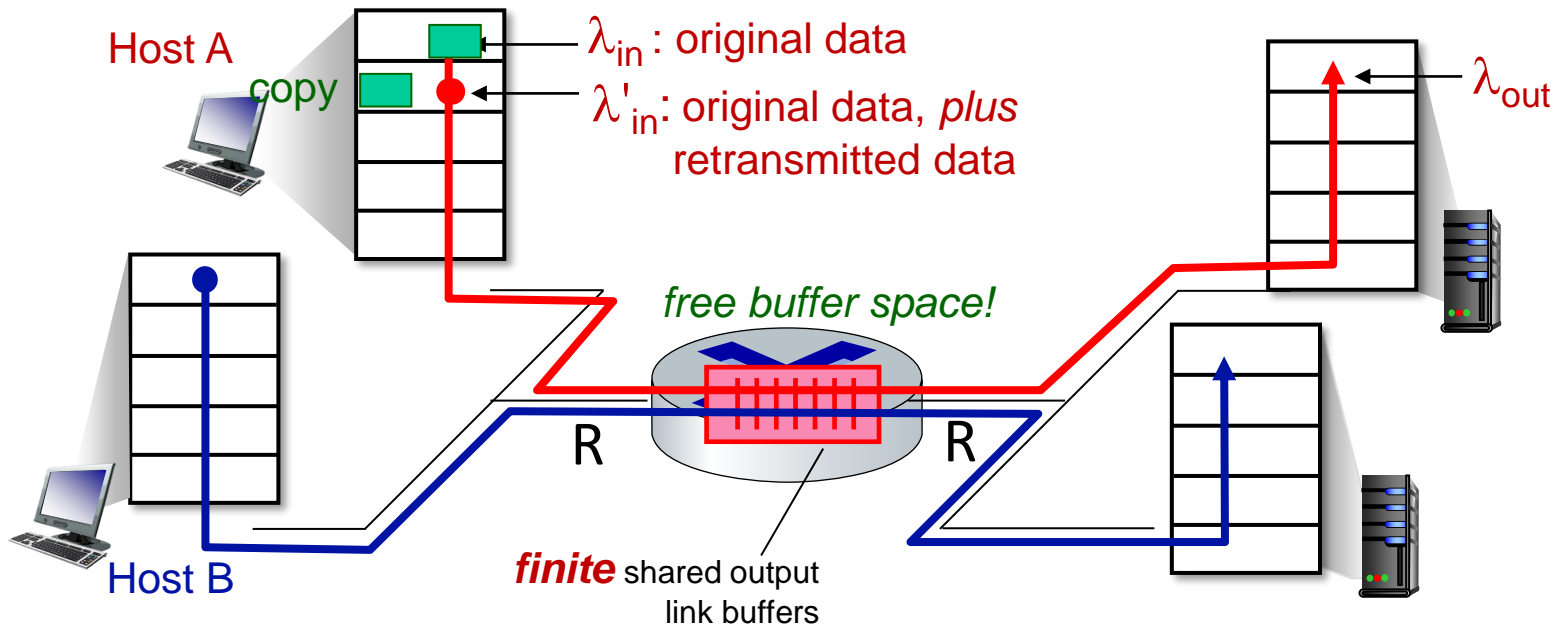




# Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

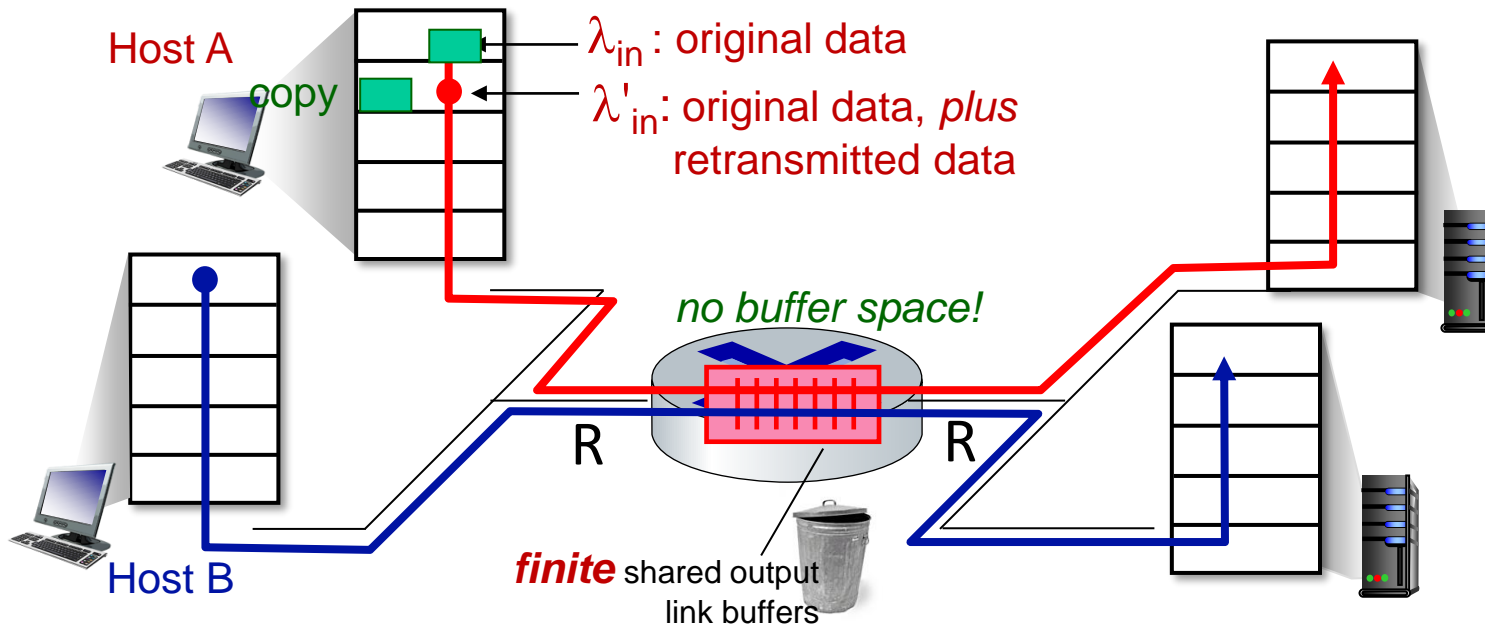
- sender sends only when router buffers available



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

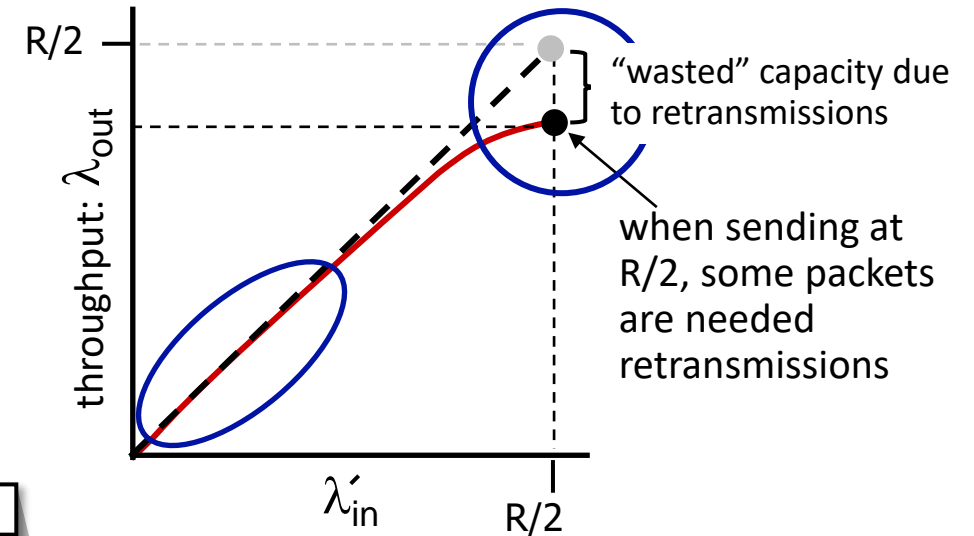
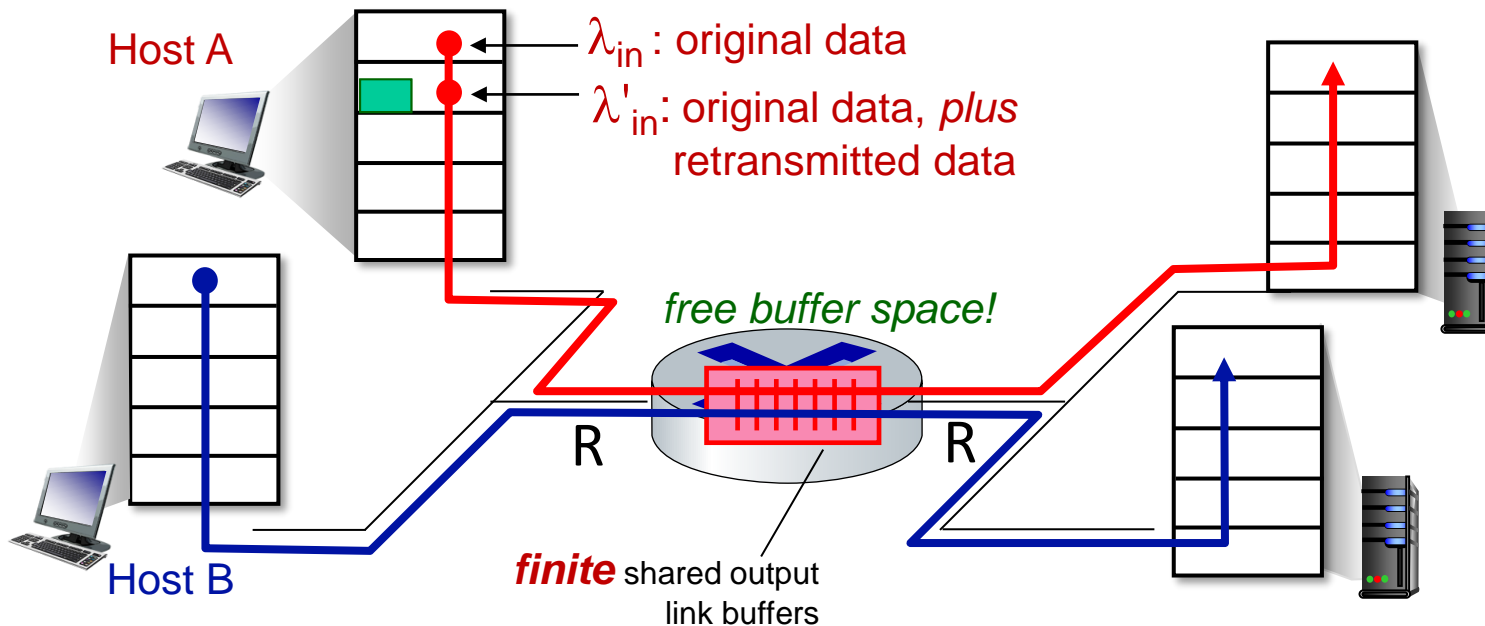
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Idealization: *some* perfect knowledge

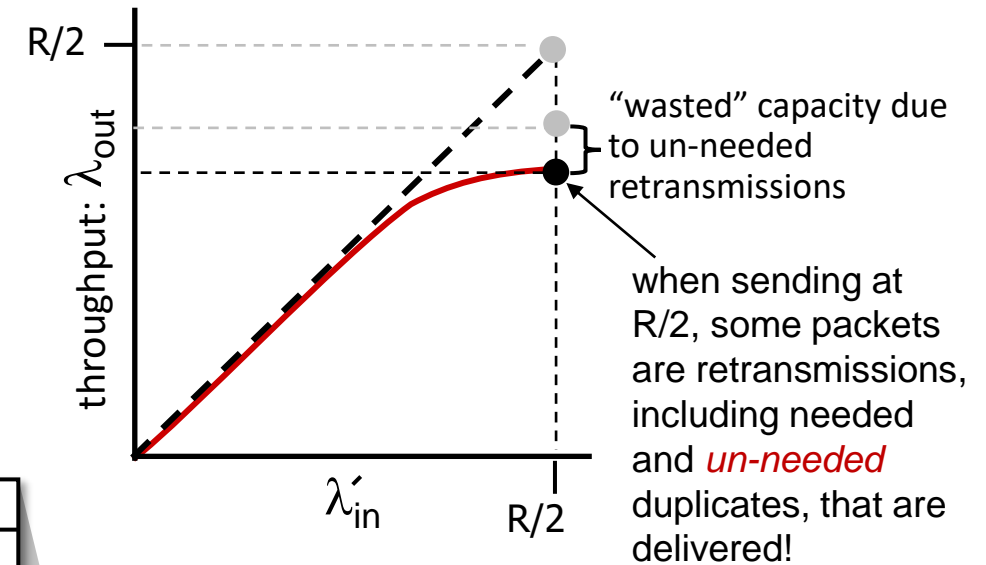
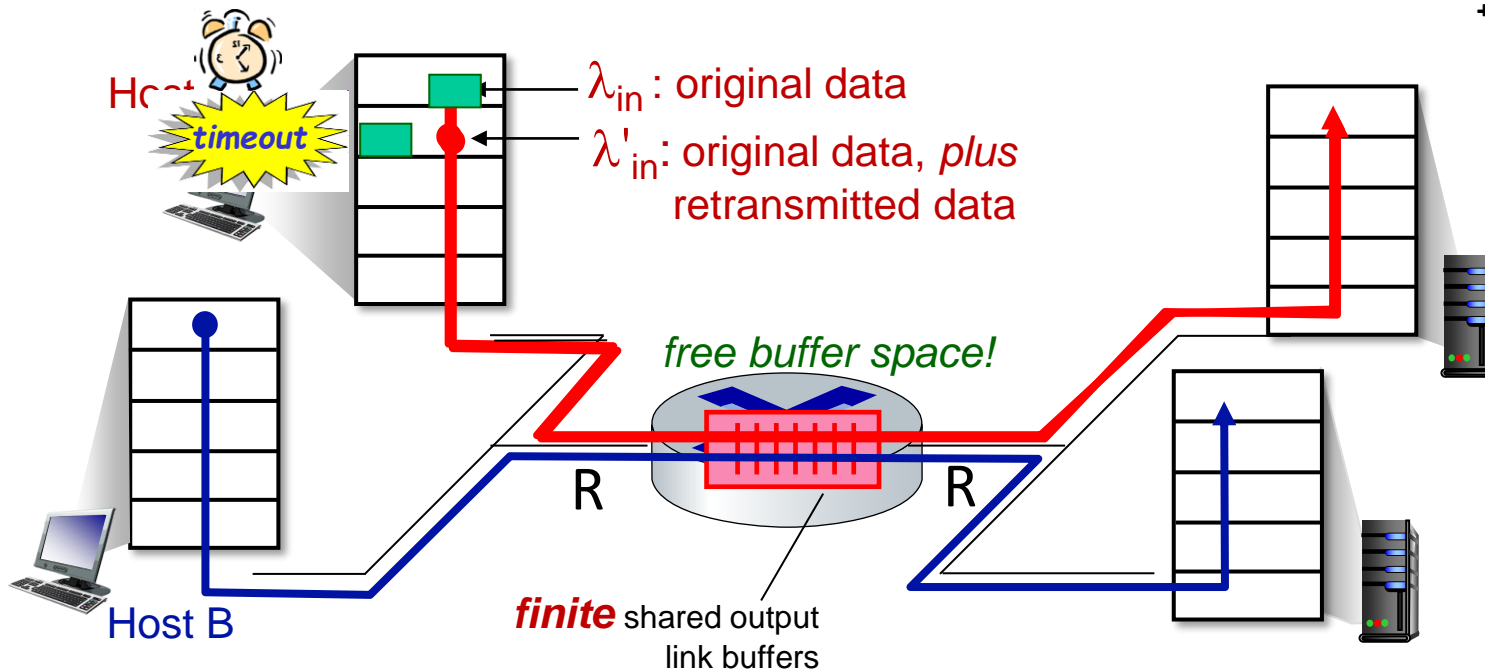
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

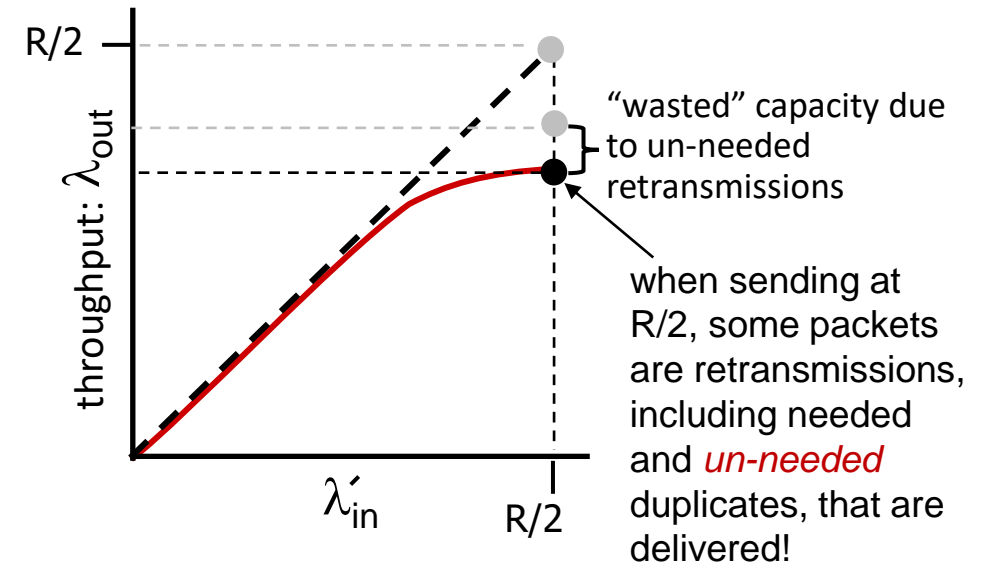
- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



## "costs" of congestion:

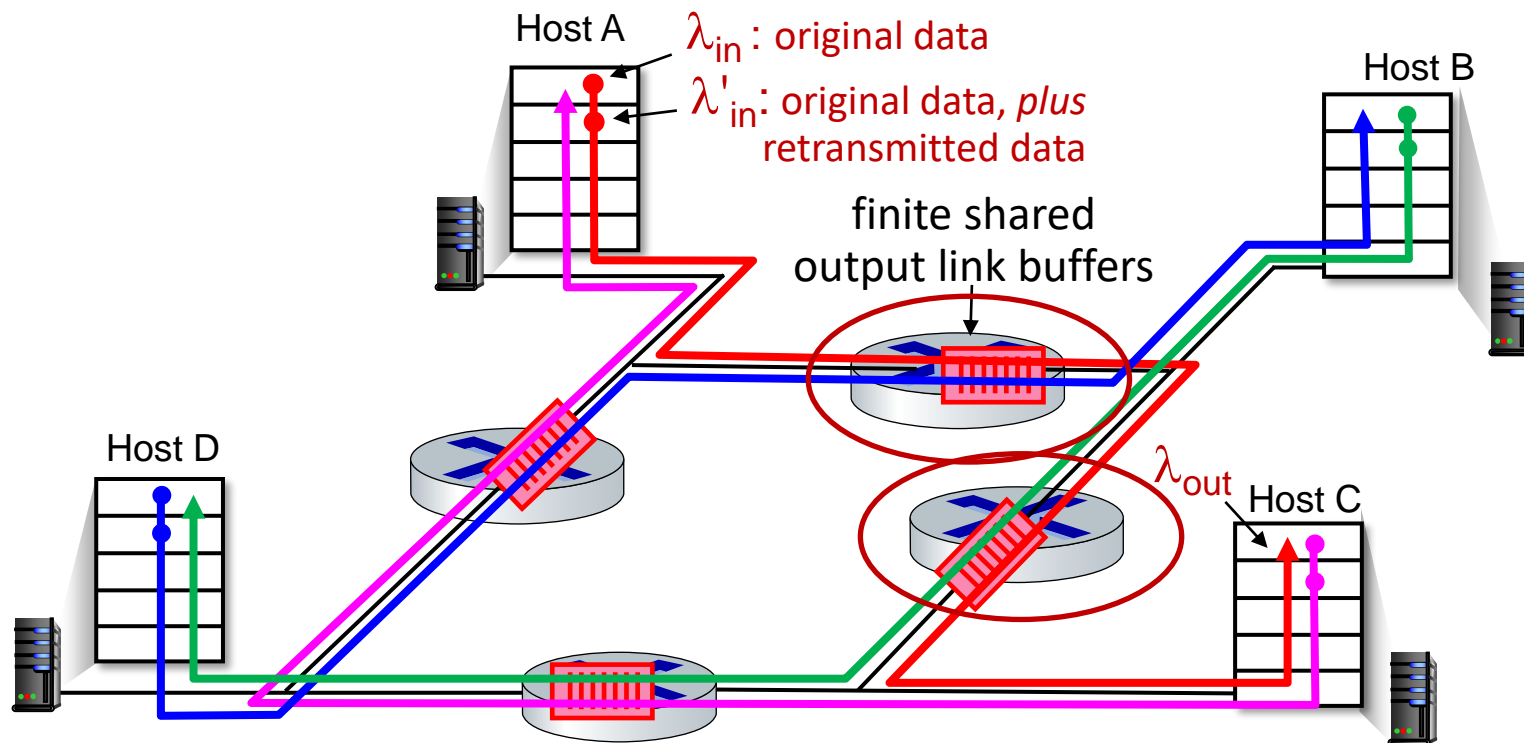
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

# Causes/costs of congestion: scenario 3

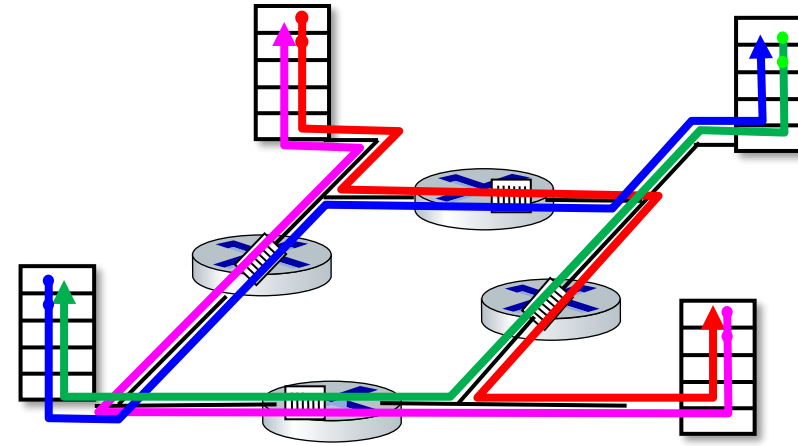
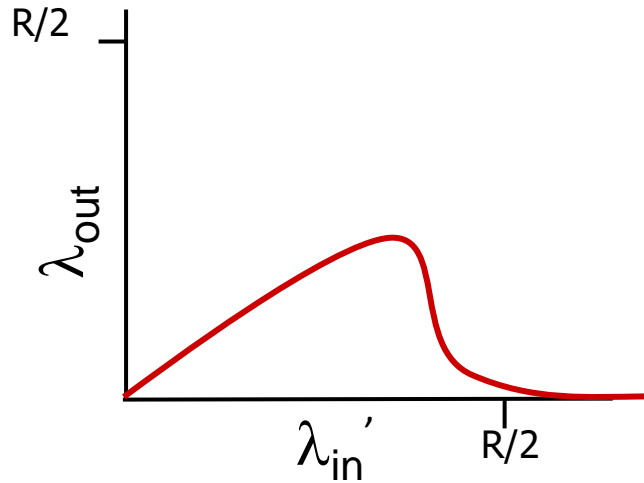
- *four* senders
- *multi-hop* paths
- timeout/retransmit

Q: what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

A: as red  $\lambda'_{in}$  increases, all arriving blue pkts at upper queue are dropped, blue throughput  $\rightarrow 0$



# Causes/costs of congestion: scenario 3

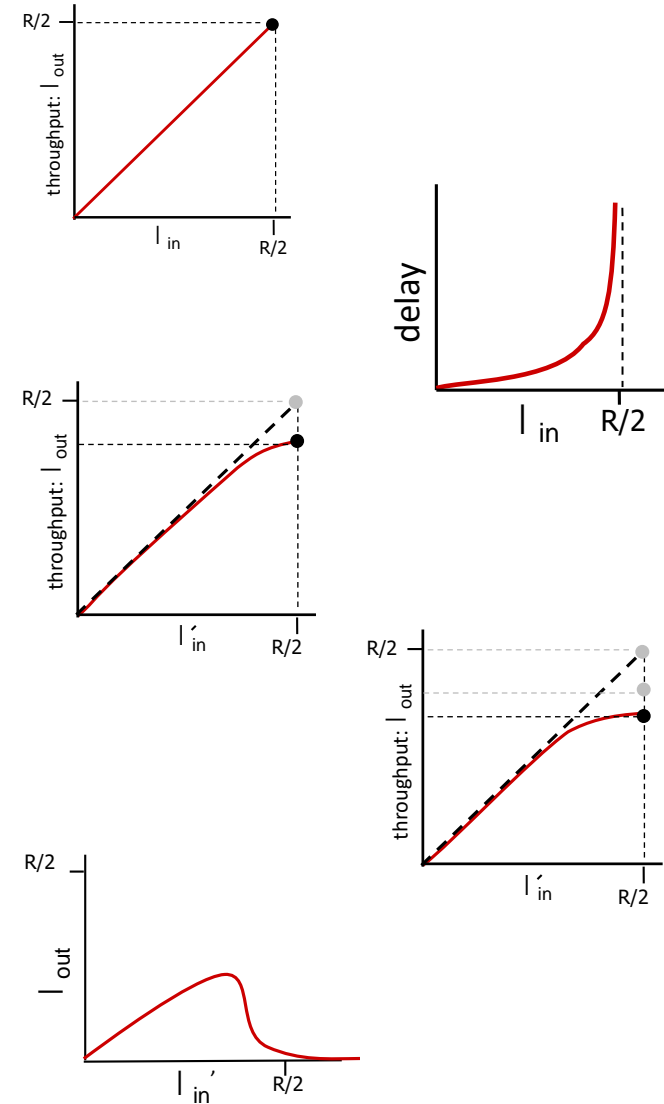


another “cost” of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

# Causes/costs of congestion: insights

- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream

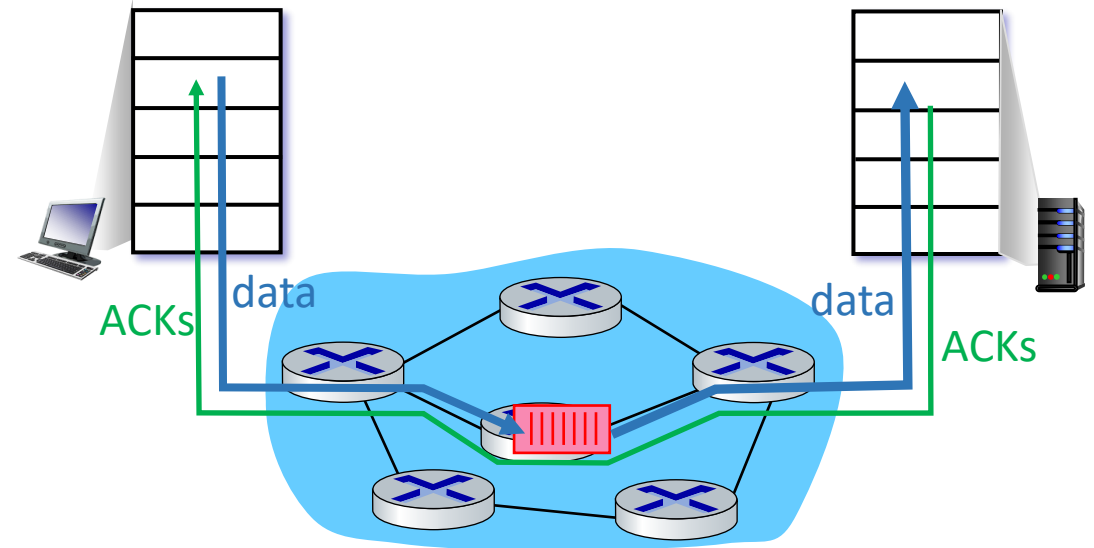




# Approaches towards congestion control

## End-end congestion control:

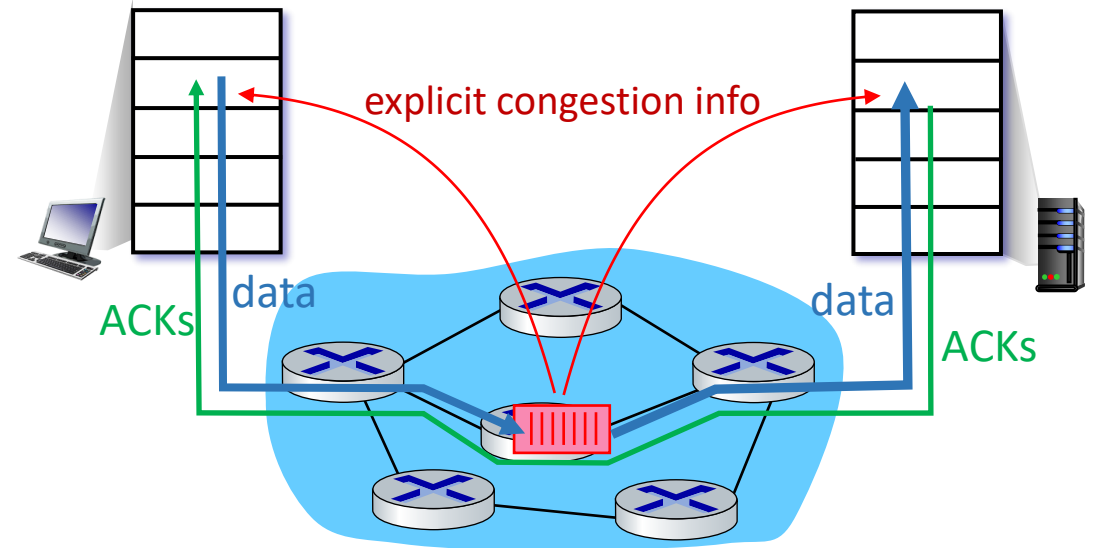
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



# Approaches towards congestion control

## Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- **TCP congestion control**
- Evolution of transport-layer functionality



# TCP congestion control: AIMD

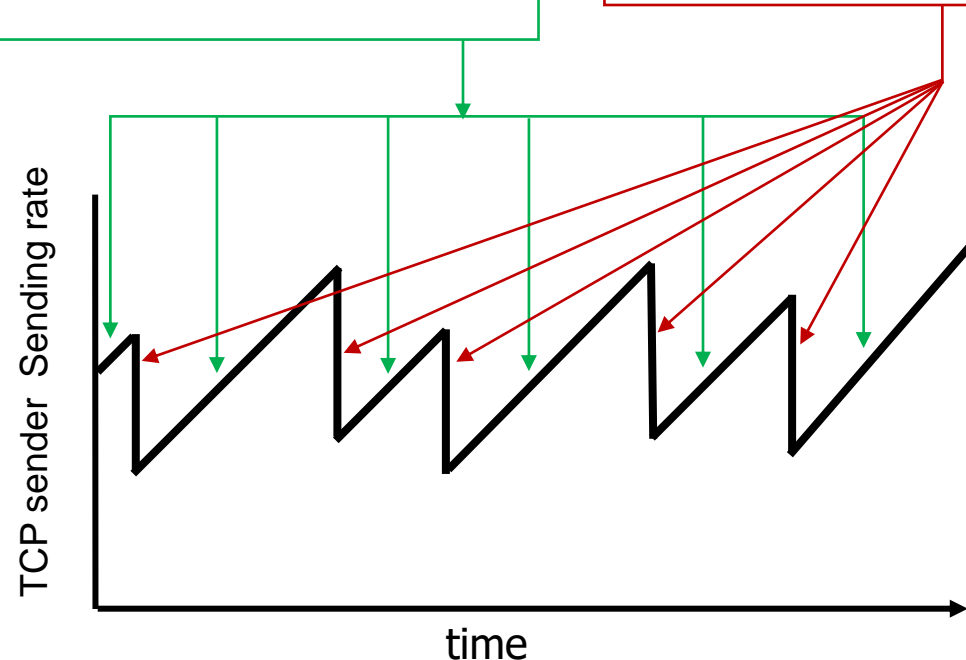
- *approach*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

## 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

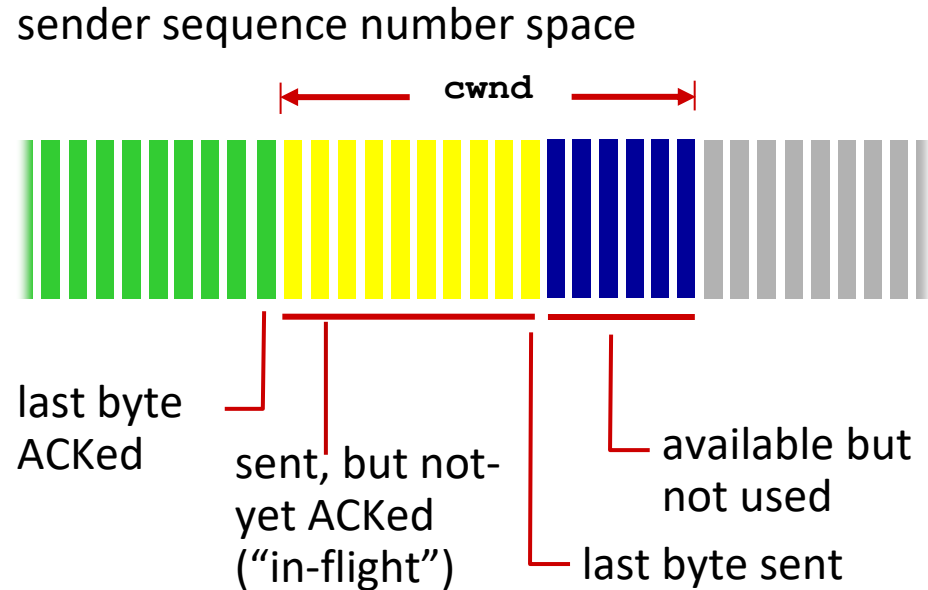
*Multiplicative decrease* detail: sending rate is

- 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



TCP sending behavior:

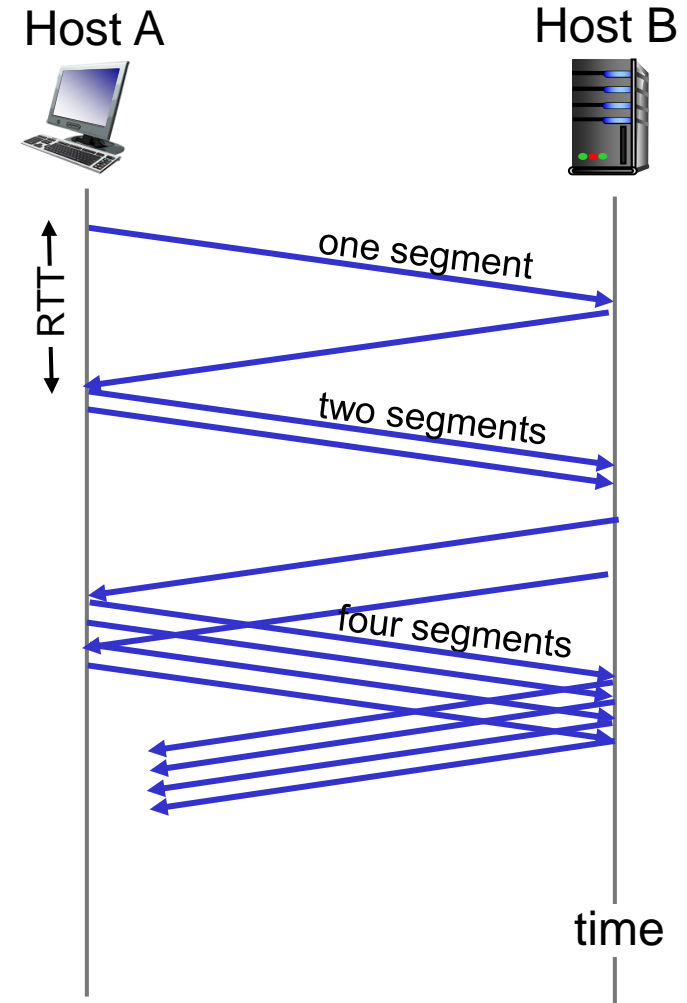
- *roughly*: send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission:  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

# 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



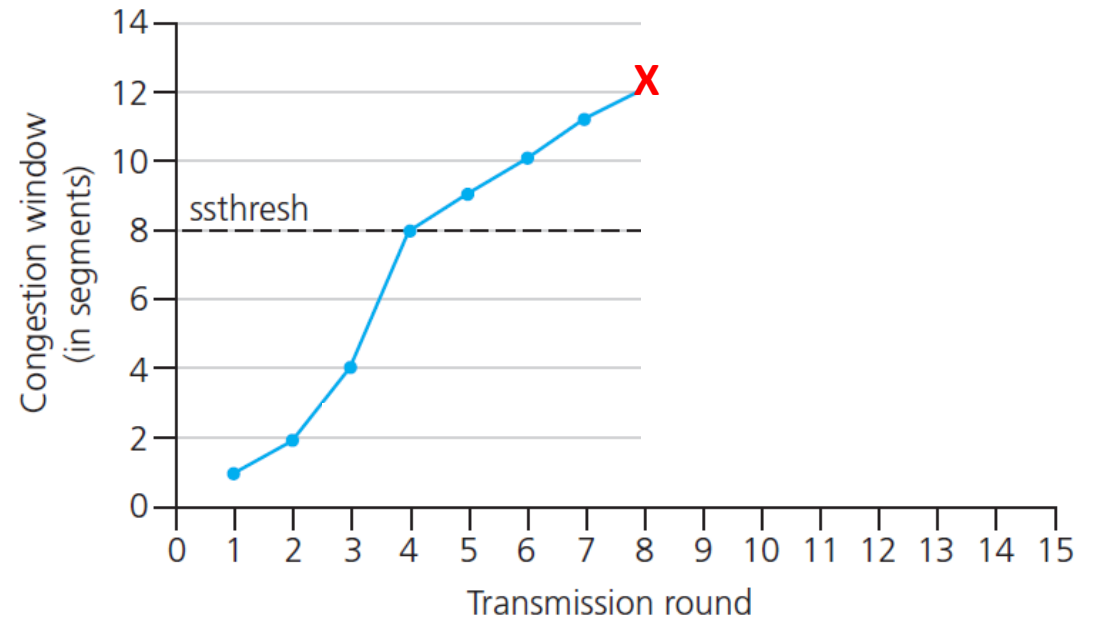
# TCP: from slow start to congestion avoidance

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

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

## Implementation:

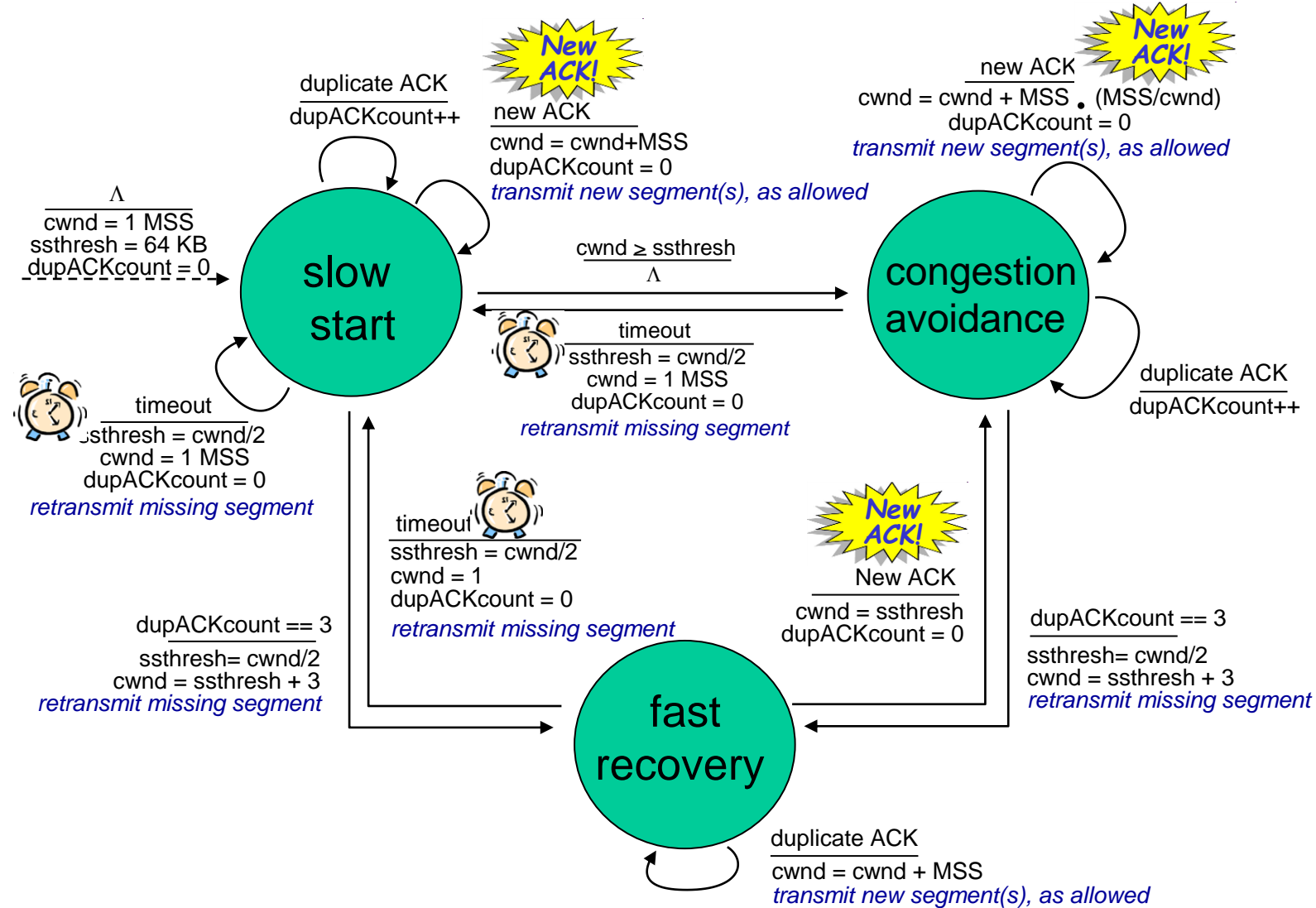
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

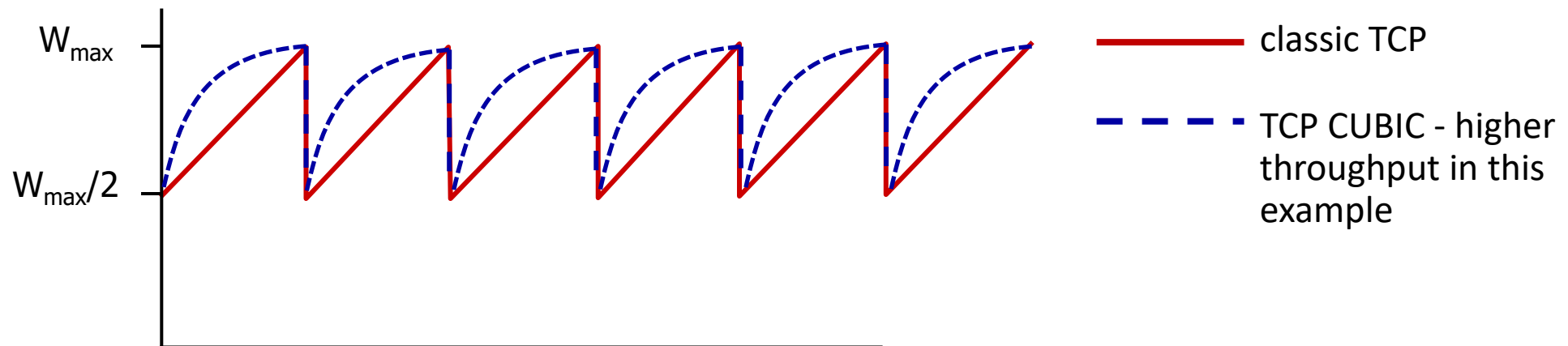


# Summary: TCP congestion control



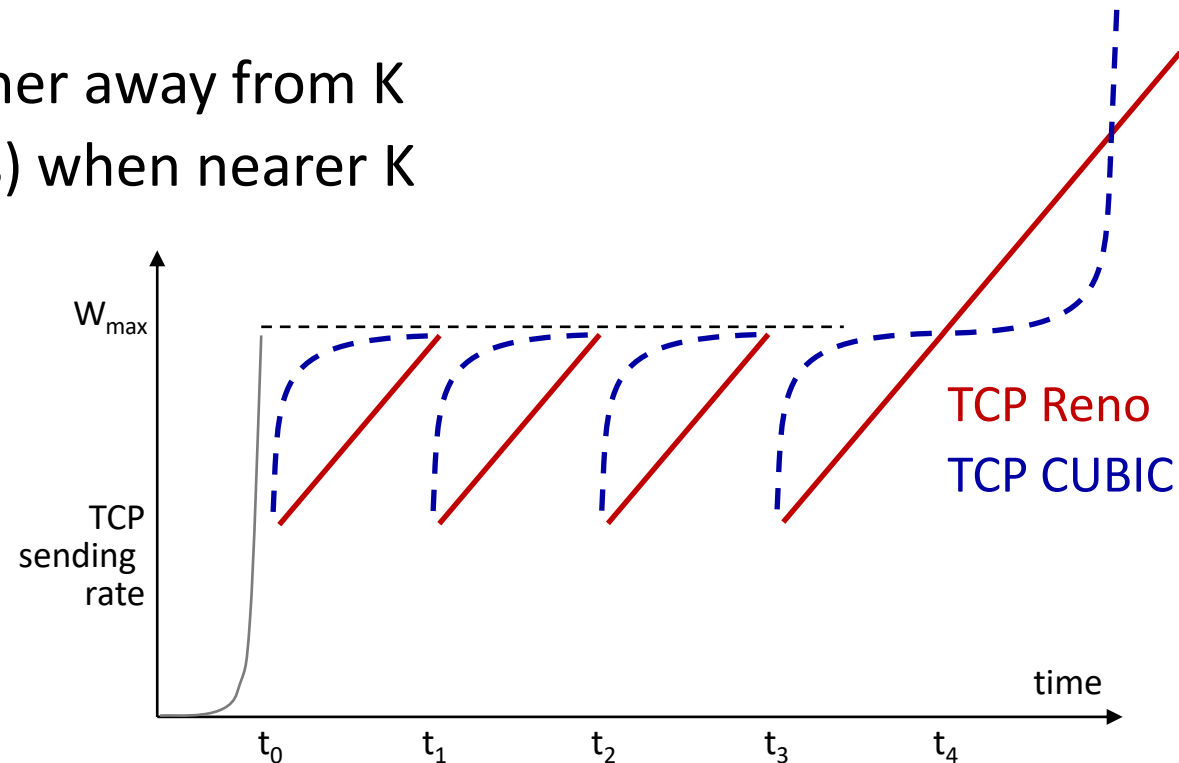
# TCP CUBIC

- 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*



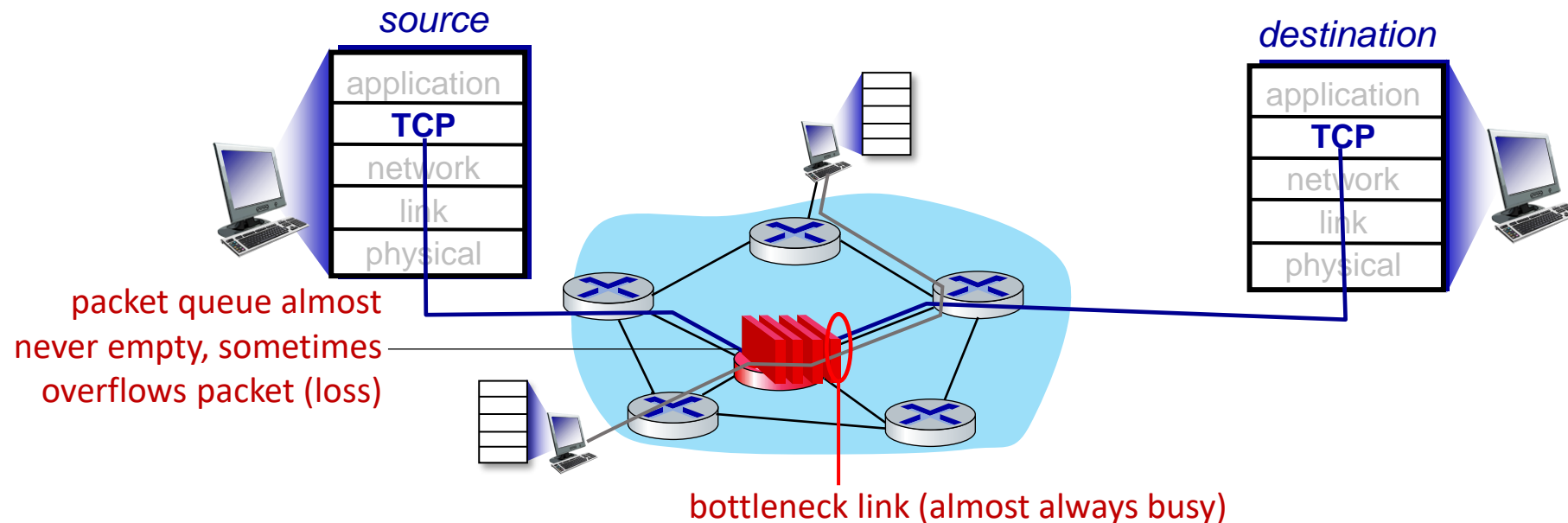
# TCP CUBIC

- 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



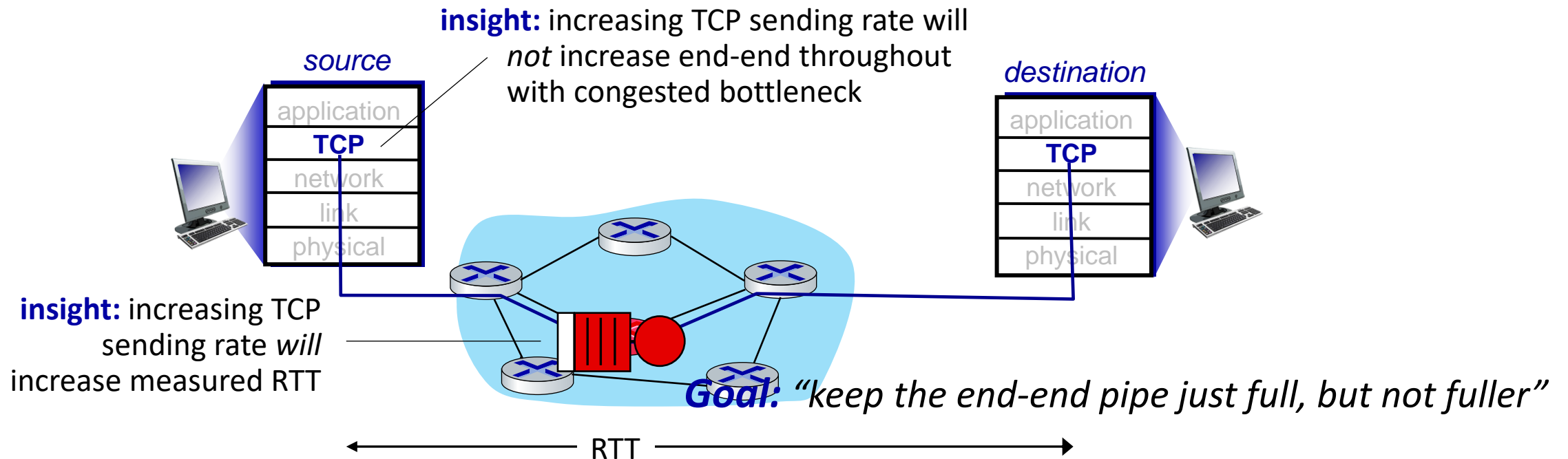
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP’s sending rate until packet loss occurs at some router’s output: the *bottleneck link*



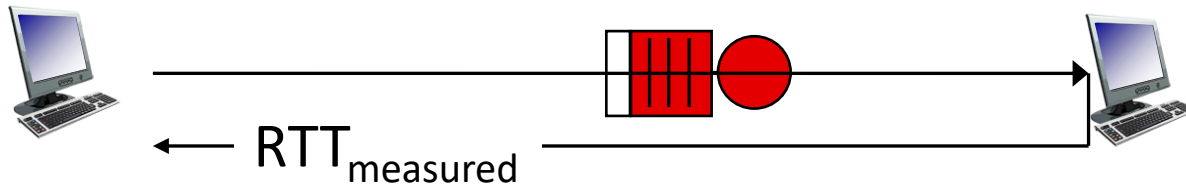
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP’s sending rate until packet loss occurs at some router’s output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



# Delay-based TCP congestion control

Keeping sender-to-receiver pipe “just full enough, but no fuller”: keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{\text{measured}}}$$

## Delay-based approach:

- $\text{RTT}_{\min}$  - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window  $\text{cwnd}$  is  $\text{cwnd}/\text{RTT}_{\min}$

if measured throughput “very close” to uncongested throughput  
    increase  $\text{cwnd}$  linearly                      /\* since path not congested \*/  
else if measured throughput “far below” uncongested throughput  
    decrease  $\text{cwnd}$  linearly                    /\* since path is congested \*/

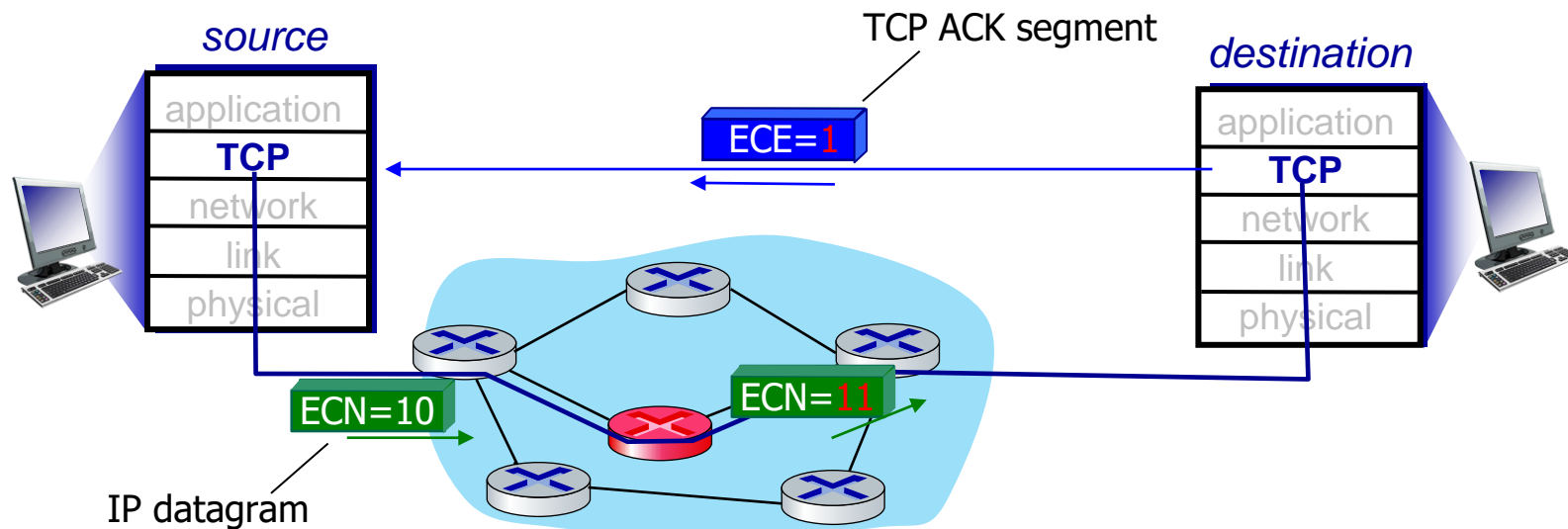
# Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughput (“keeping the just pipe full... ”) while keeping delay low (“...but not fuller”)
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google’s (internal) backbone network

# Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

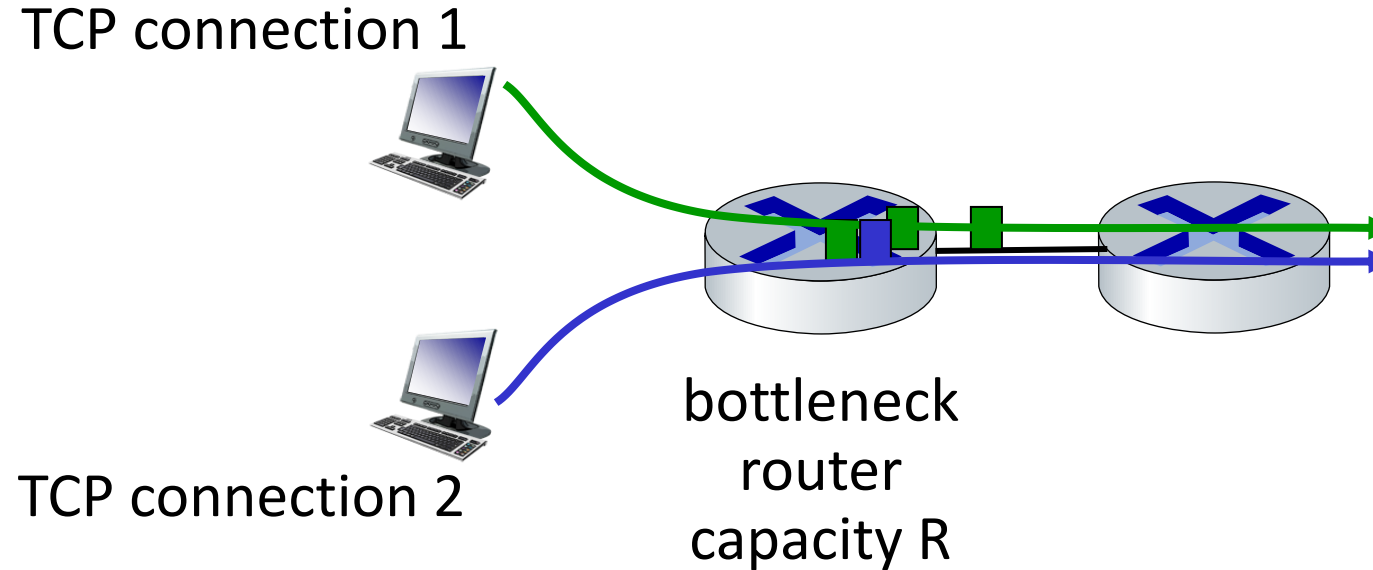
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)





# TCP fairness

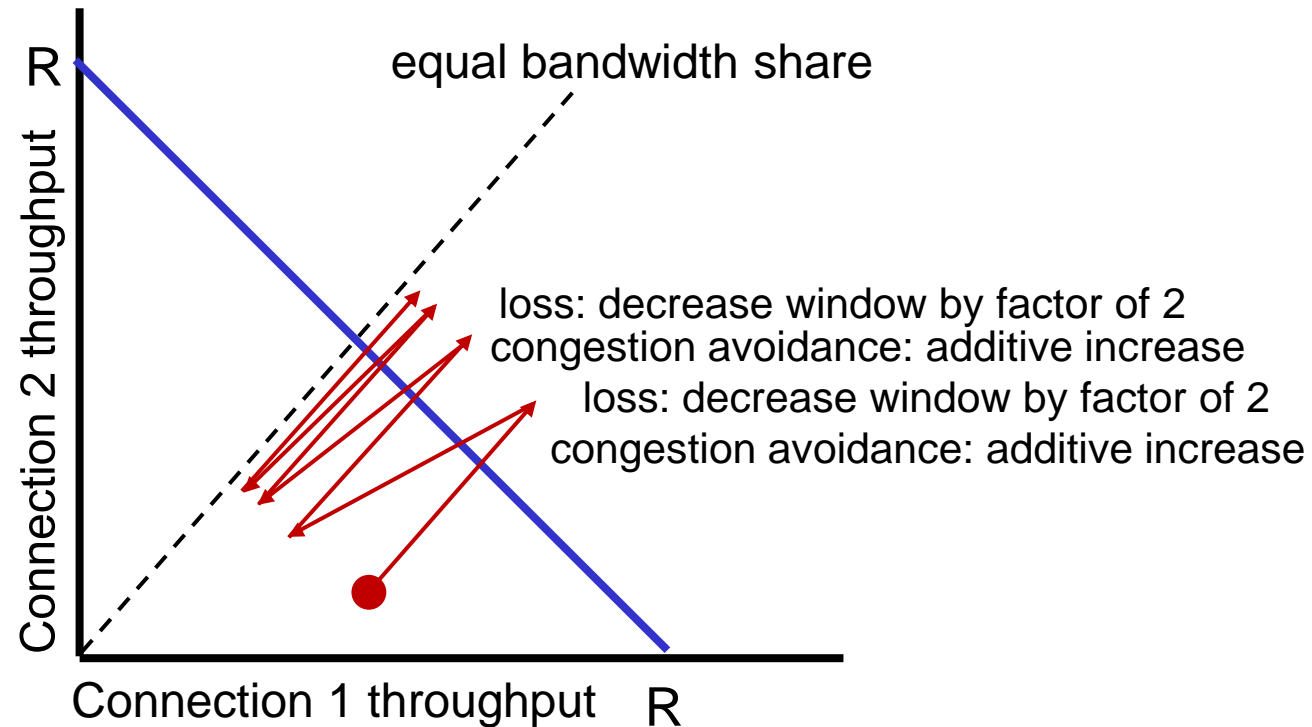
**Fairness goal:** if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally



*Is TCP fair?*

**A:** Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

# Fairness: must all network apps be “fair”?

## 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
- there is no “Internet police” policing use of congestion control

## 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  $R/2$