

# Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

# Client/server socket interaction: UDP

## server (running on serverIP)

create socket, port= x:  
`serverSocket =  
socket(AF_INET,SOCK_DGRAM)`

↓  
read datagram from  
`serverSocket`

↓  
write reply to  
`serverSocket`  
specifying  
client address,  
port number

## client

create socket:  
`clientSocket =  
socket(AF_INET,SOCK_DGRAM)`

↓  
Create datagram with server IP and  
port=x; send datagram via  
`clientSocket`

↓  
read datagram from  
`clientSocket`

↓  
close  
`clientSocket`

# Example app: UDP client

## *Python UDPClient*

```
from socket import *
```

```
serverName = 'hostname'
```

```
serverPort = 12000
```

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

```
message = raw_input('Input lowercase sentence:')
```

```
clientSocket.sendto(message,(serverName, serverPort))
```

```
modifiedMessage, serverAddress =
```

```
clientSocket.recvfrom(1024)
```

```
print modifiedMessage
```

```
clientSocket.close()
```

# Example app: UDP server

## *Python UDPServer*

```
from socket import *
```

```
serverPort = 12000
```

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
```

```
serverSocket.bind(("", serverPort))
```

```
print "The server is ready to receive"
```

```
while 1:
```

```
    message, clientAddress = serverSocket.recvfrom(1024)
```

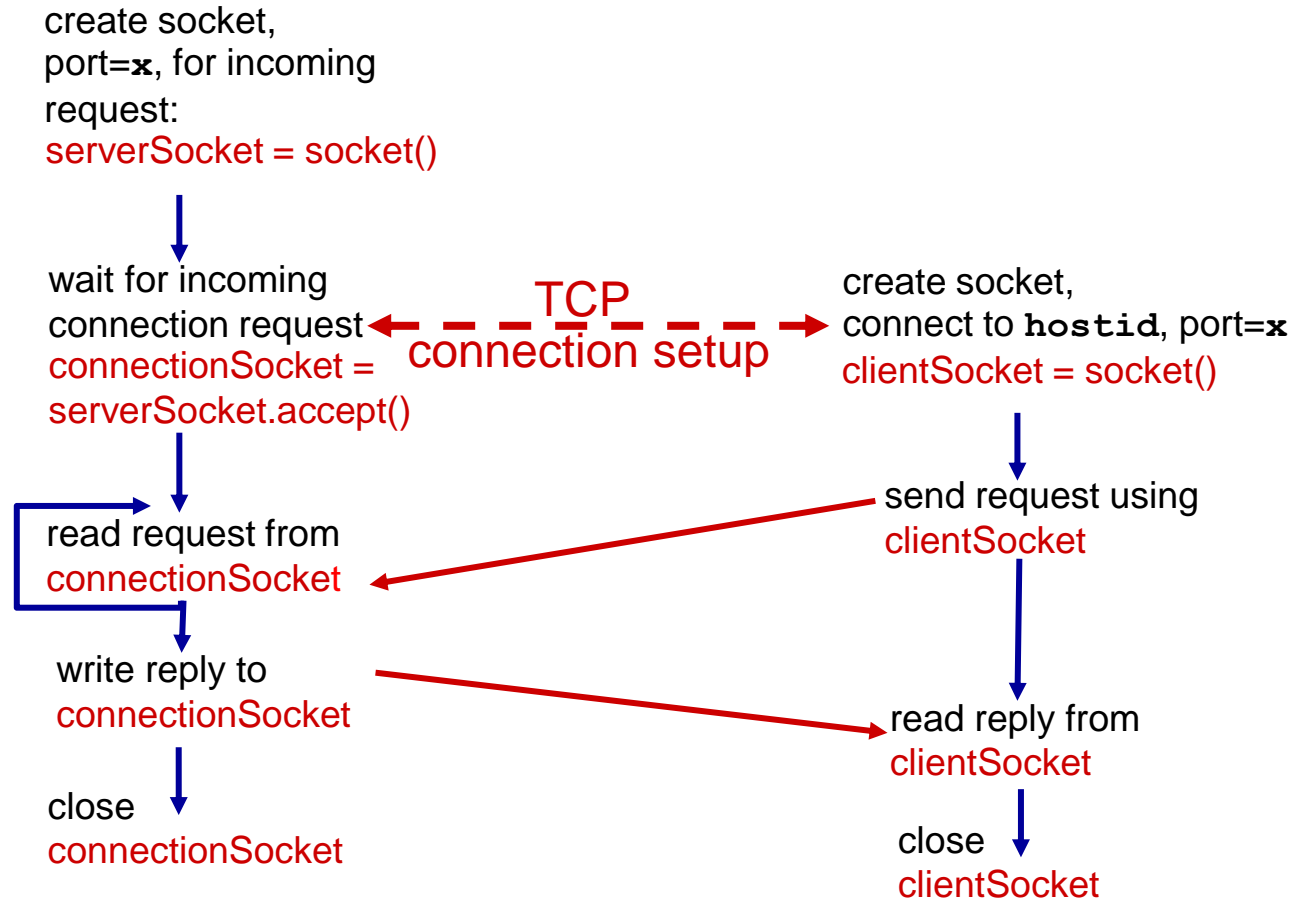
```
    modifiedMessage = message.upper()
```

```
    serverSocket.sendto(modifiedMessage, clientAddress)
```

# Client/server socket interaction: TCP

server (running on `hostid`)

client



# Example app:TCP client

## *Python TCPCClient*

```
from socket import *
```

```
serverName = 'servername'
```

```
serverPort = 12000
```

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

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

```
sentence = raw_input('Input lowercase sentence:')
```

```
clientSocket.send(sentence)
```

```
modifiedSentence = clientSocket.recv(1024)
```

```
print 'From Server:', modifiedSentence
```

```
clientSocket.close()
```

# Example app: TCP server

## *Python TCPServer*

```
from socket import *
```

```
serverPort = 12000
```

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

```
serverSocket.bind(('', serverPort))
```

```
serverSocket.listen(1)
```

```
print 'The server is ready to receive'
```

```
while 1:
```

```
    connectionSocket, addr = serverSocket.accept()
```

```
    sentence = connectionSocket.recv(1024)
```

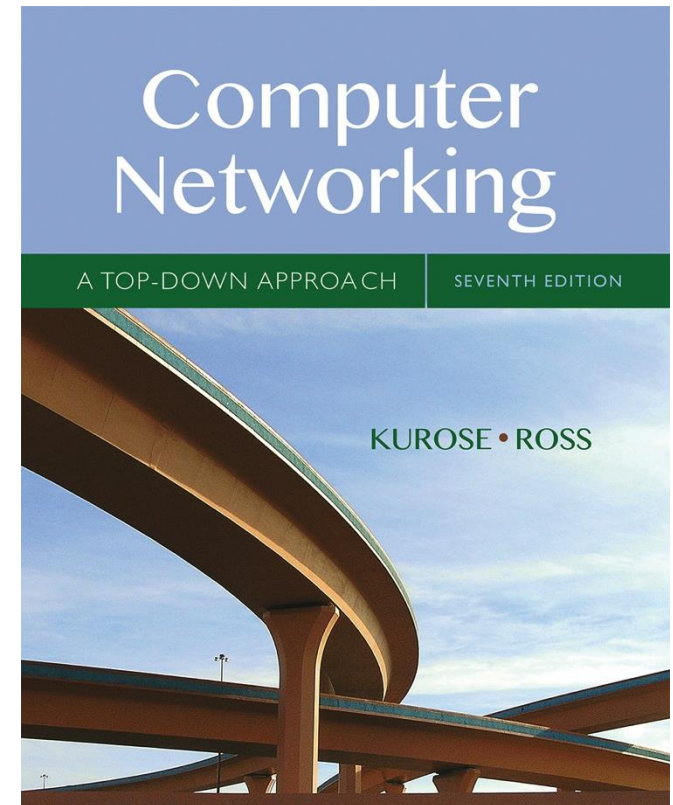
```
    capitalizedSentence = sentence.upper()
```

```
    connectionSocket.send(capitalizedSentence)
```

```
    connectionSocket.close()
```

# Chapter 3

## Transport Layer



## *Computer Networking: A Top Down Approach*

7<sup>th</sup> edition

Jim Kurose, Keith Ross

Pearson/Addison Wesley

April 2016

Slides adopted from original ones provided by the textbook authors.



# 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

# Transport vs. network layer

- ❖ **network layer:** logical communication between hosts
- ❖ **transport layer:** logical communication between processes
  - relies on, enhances, network layer services
- ❖ **two transport-layer protocols**
  - reliable, in-order delivery: TCP
  - unreliable, unordered delivery: UDP

# 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

# Demultiplexing

- ❖ UDP socket identified by 2-tuple:
  - dest IP address
  - dest port number
- ❖ TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

# 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

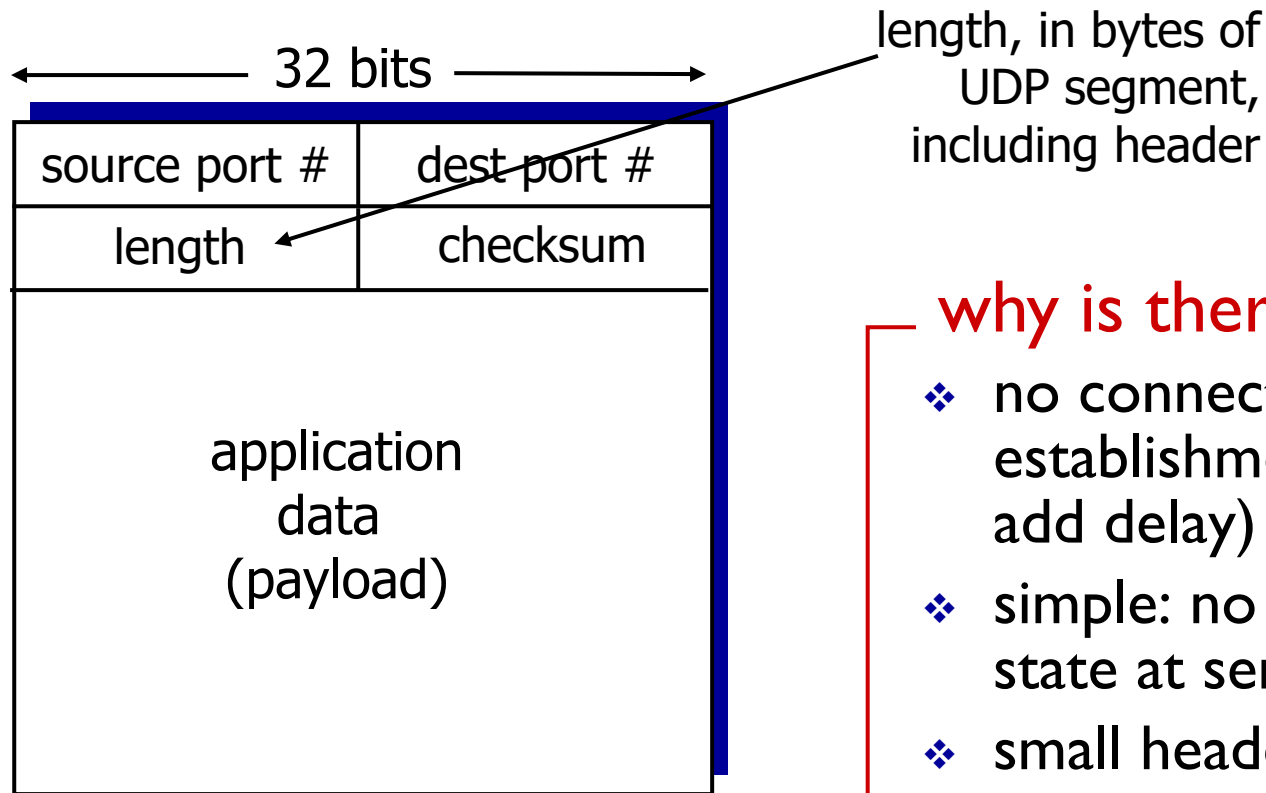
3.7 TCP congestion control

# UDP: User Datagram Protocol [RFC 768]

- ❖ “best effort” service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- ❖ *connectionless*:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- ❖ UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP

# UDP: segment header



UDP segment format

## why is there a UDP?

- ❖ no connection establishment (which can add delay)
- ❖ simple: no connection state at sender, receiver
- ❖ small header size
- ❖ no congestion control: UDP can blast away as fast as desired

# UDP checksum

*Goal:* detect “errors” (e.g., flipped bits) in transmitted segment

## sender:

- ❖ treat segment contents, including header fields, as sequence of 16-bit integers
- ❖ checksum: addition (one's complement sum) of segment contents
- ❖ sender puts checksum value into UDP checksum field

## receiver:

- ❖ compute checksum of received segment
- ❖ check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.  
*But maybe errors nonetheless? More later*  
....



# Internet checksum: example

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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

# 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

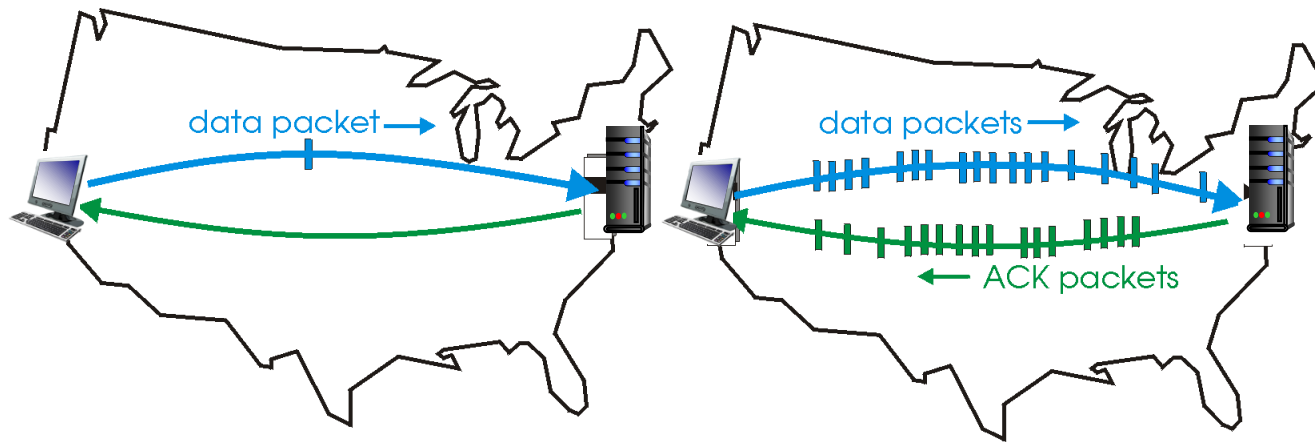
# Reliable Data Transfer

- ❖ **Challenge:** TCP requires reliable data transfer, but the underlying protocol IP is not reliable.
- ❖ Possible errors
  - channel with bit errors -> checksum, ACK/NAK, retransmission
  - with corrupted ACK/NAKs -> retransmission, sequence #
  - without NAKs -> ACK retransmission
  - channels with packet loss -> timer

# Pipelined protocols

**stop-and-wait:** one packet at a time

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



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

(b) a pipelined protocol in operation

**utilization:**

$$y_{\text{sender}} = \frac{n L / R}{RTT + L / R}$$



# 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

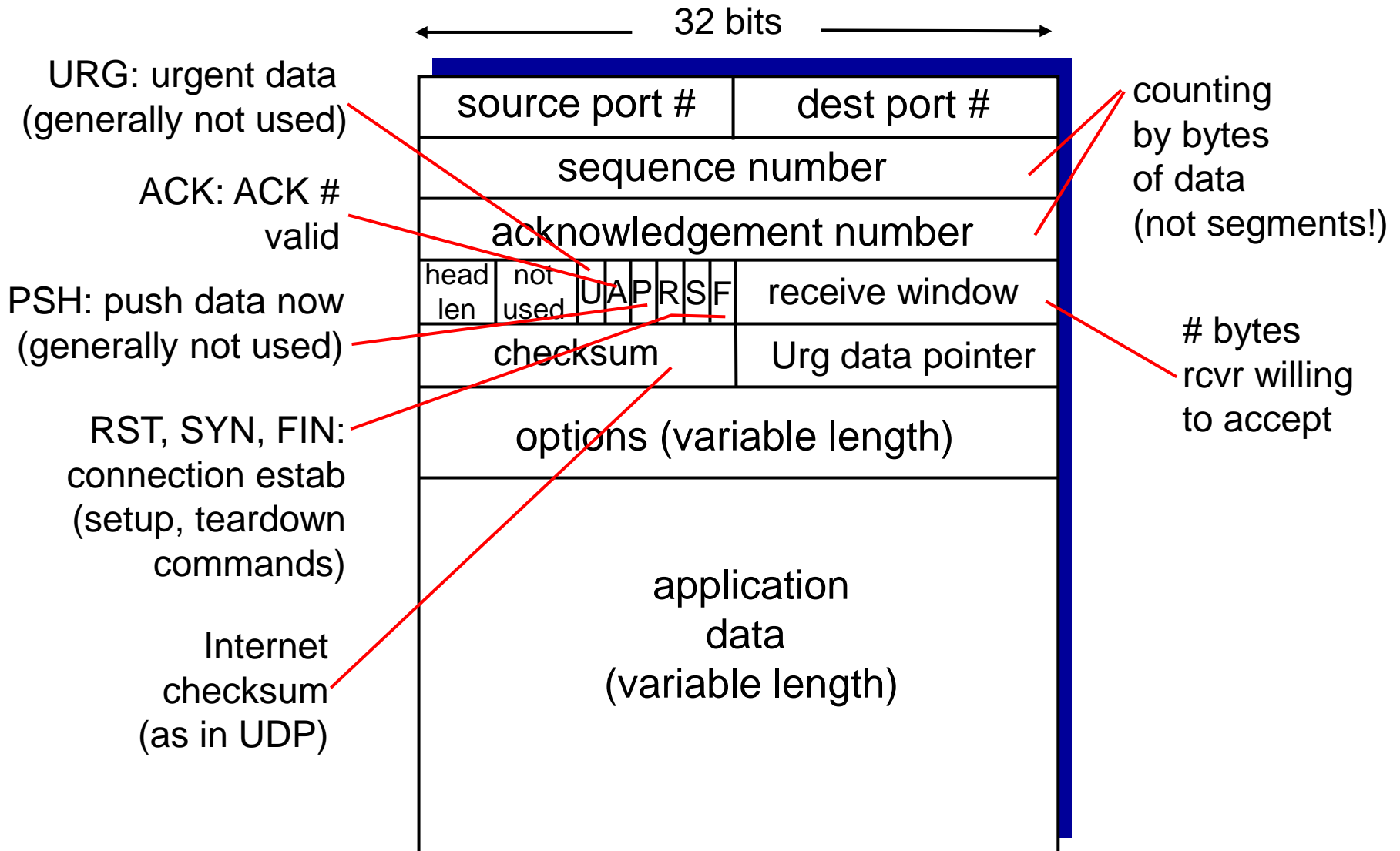
3.7 TCP congestion control

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order *byte stream*:**
  - 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) initializes sender, receiver state before data exchange
- ❖ **flow controlled:**
  - sender will not overwhelm receiver

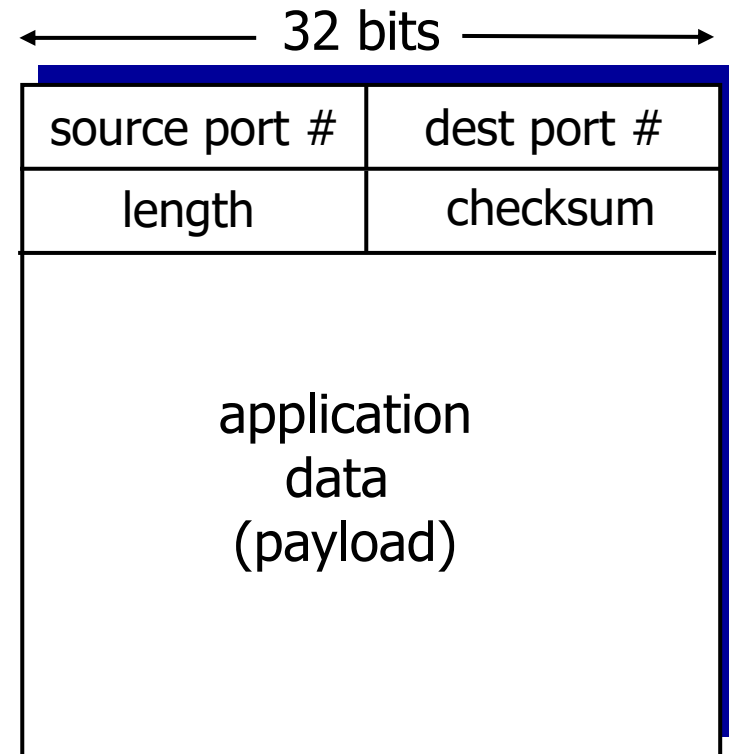
# TCP segment structure





# Review: UDP

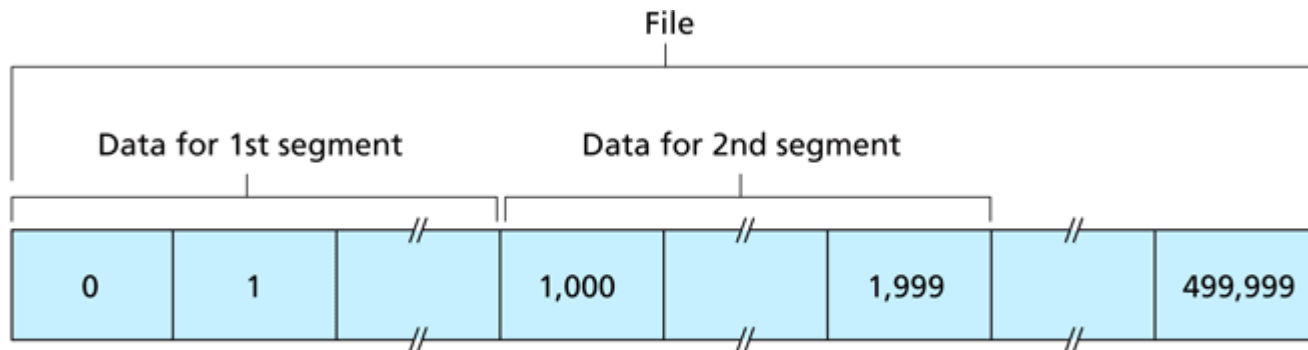
- ❖ “best effort” service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- ❖ *connectionless*:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others



UDP segment format

# Maximum segment size (MSS)

- ❖ MSS: maximum bytes of TCP payload
- ❖ Sequence #: byte-stream # of first byte in segment
- ❖ E.g. file size 500,000 bytes, MSS 1,000 bytes



**Figure 3.30** ♦ Dividing file data into TCP segments

# TCP seq. numbers, ACKs

## sequence numbers:

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

## acknowledgements:

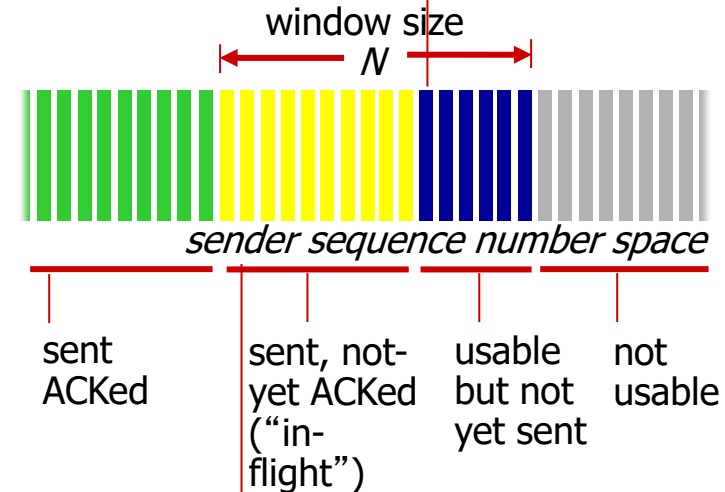
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments

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

outgoing segment from sender

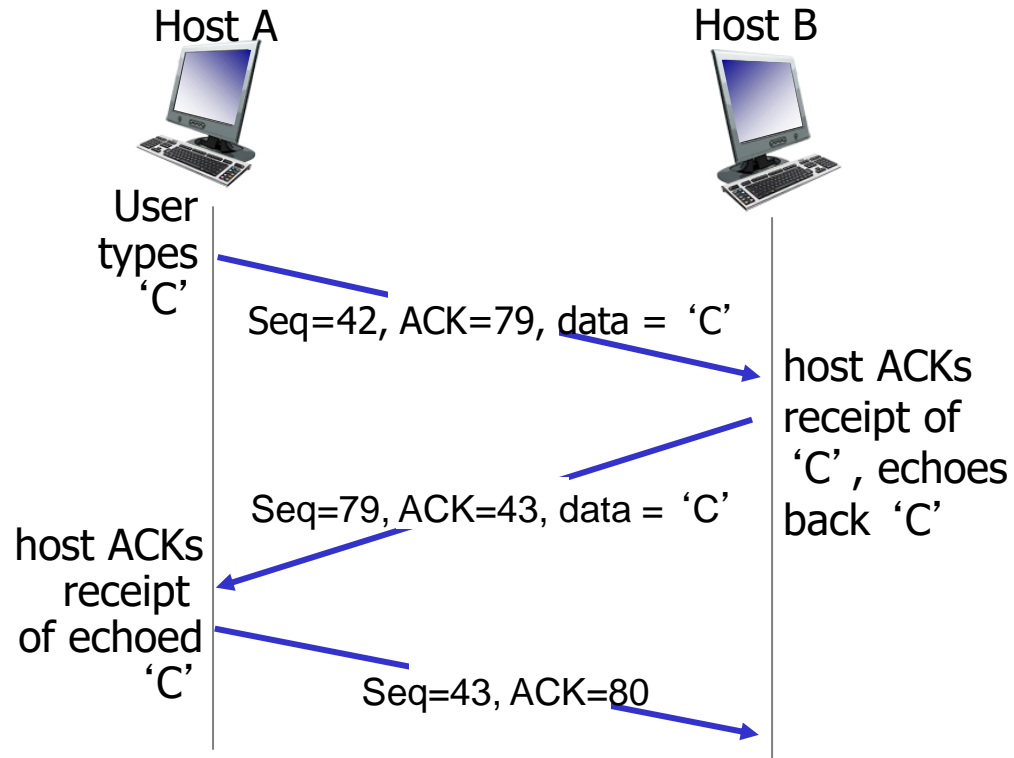
source port #		dest port #	
sequence number			
acknowledgement number			
			rwnd
checksum		urg pointer	



incoming segment to sender

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

# TCP seq. numbers, ACKs



simple telnet scenario

# Example

P26. Consider transferring an enormous file of  $L$  bytes from Host A to Host B. Assume an MSS of 536 bytes.

- a. What is the maximum value of  $L$  such that TCP sequence numbers are not exhausted?
- b. For the  $L$  you obtain in (a), find how long it takes to transmit the file. Assume that a total of 66 bytes of transport, network, and data-link header are added to each segment before the resulting packet is sent out via a 155 Mbps link.

# 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

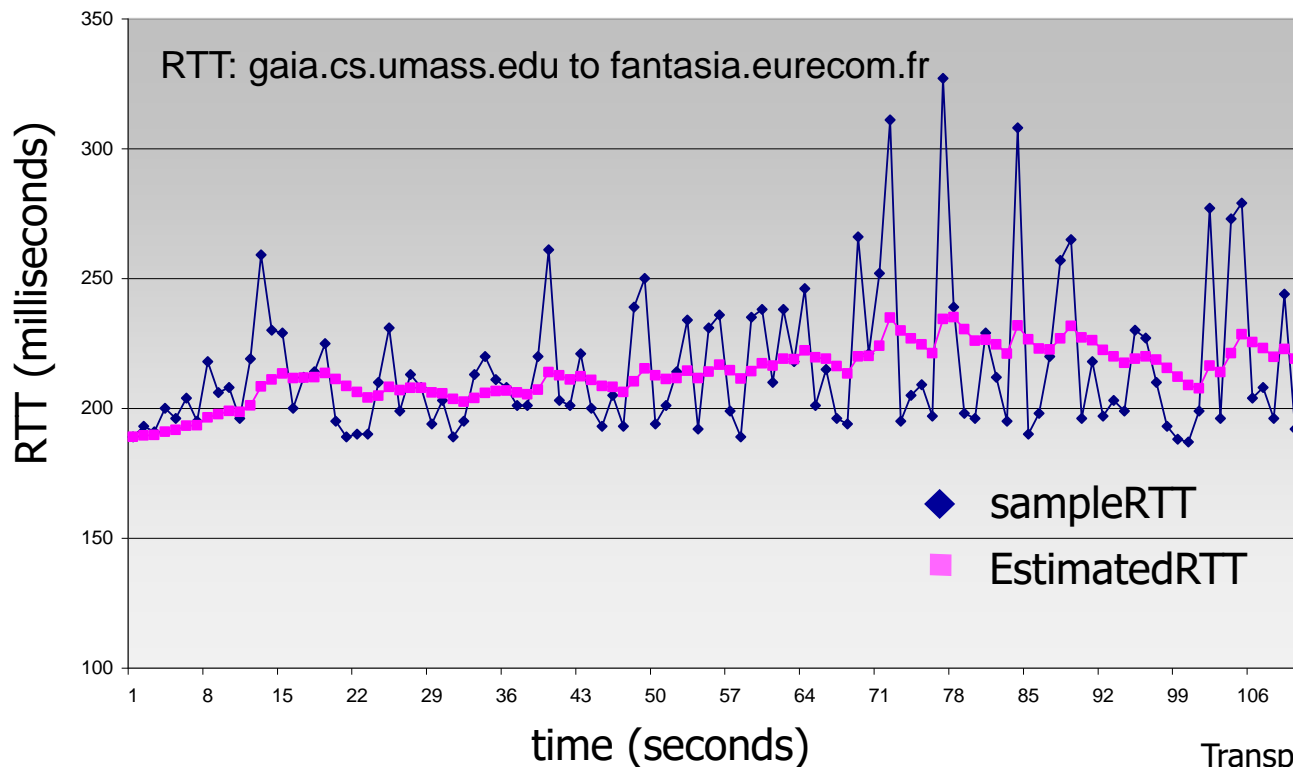
Q: how to estimate RTT?

- ❖ **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:  $\alpha = 0.125$



# TCP round trip time, timeout

- ❖ **timeout interval:** **EstimatedRTT** plus “safety margin”
  - large variation in **EstimatedRTT** -> larger safety margin
- ❖ estimate **SampleRTT** deviation from **EstimatedRTT**:

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

(typically,  $\beta = 0.25$ )

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



↑  
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`

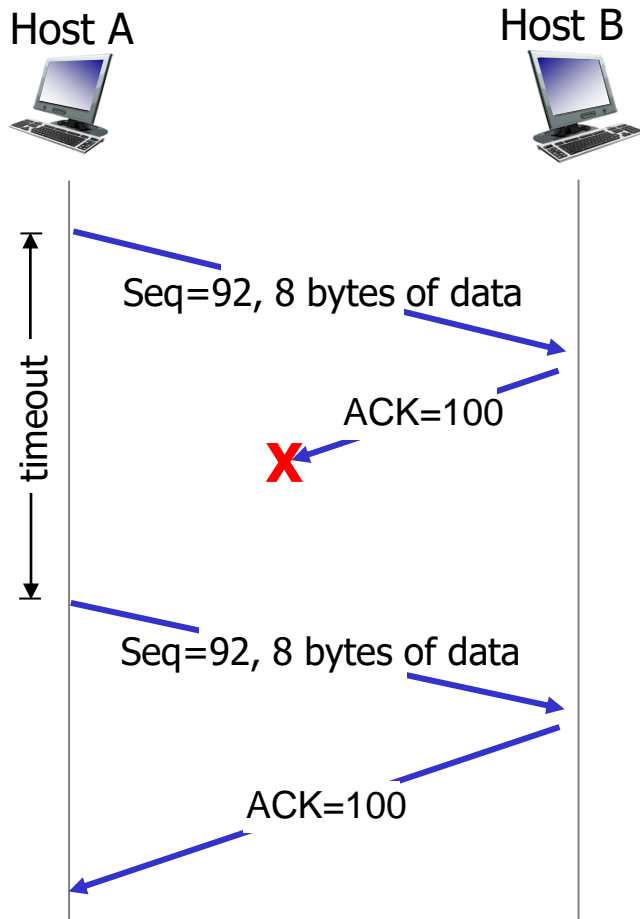
## *timeout:*

- ❖ retransmit segment that caused timeout
- ❖ restart timer

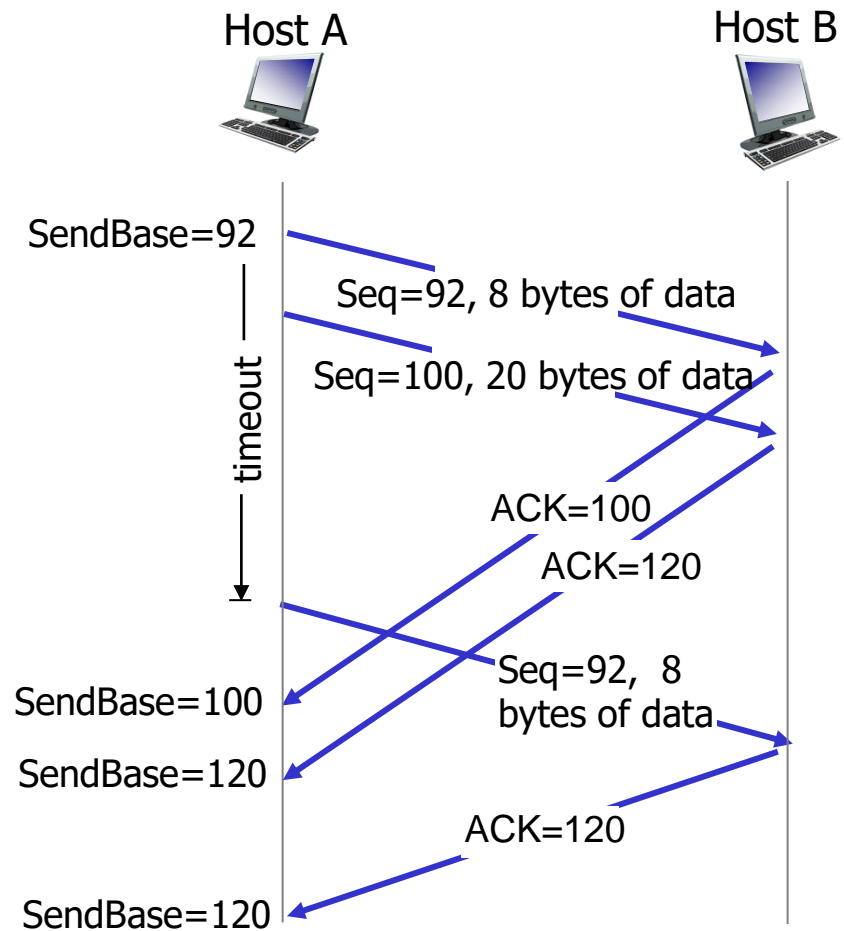
## *ack rcvd:*

- ❖ if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

# TCP: retransmission scenarios

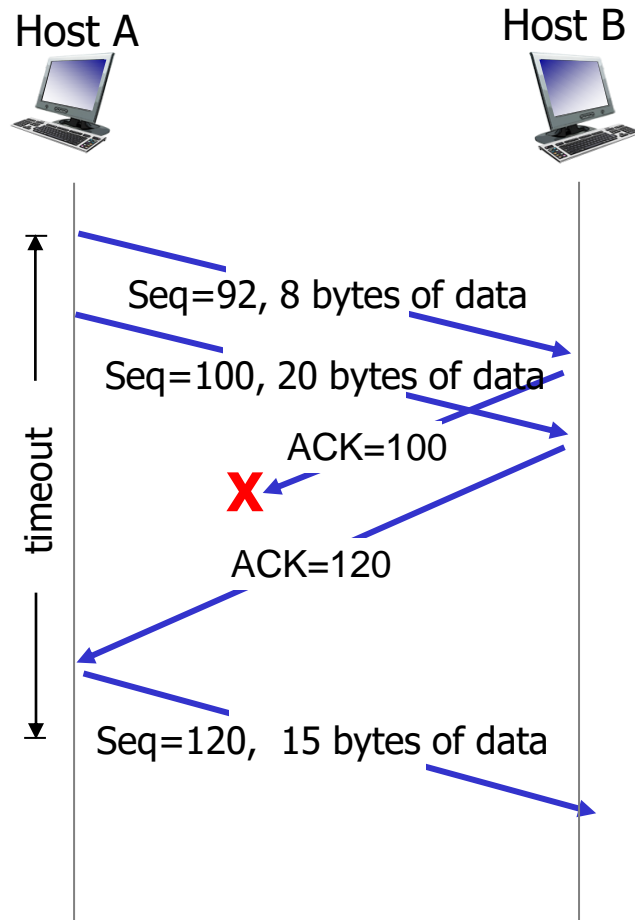


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



cumulative ACK

# TCP ACK generation [RFC 1122, RFC 2581]

<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
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

# TCP fast retransmit

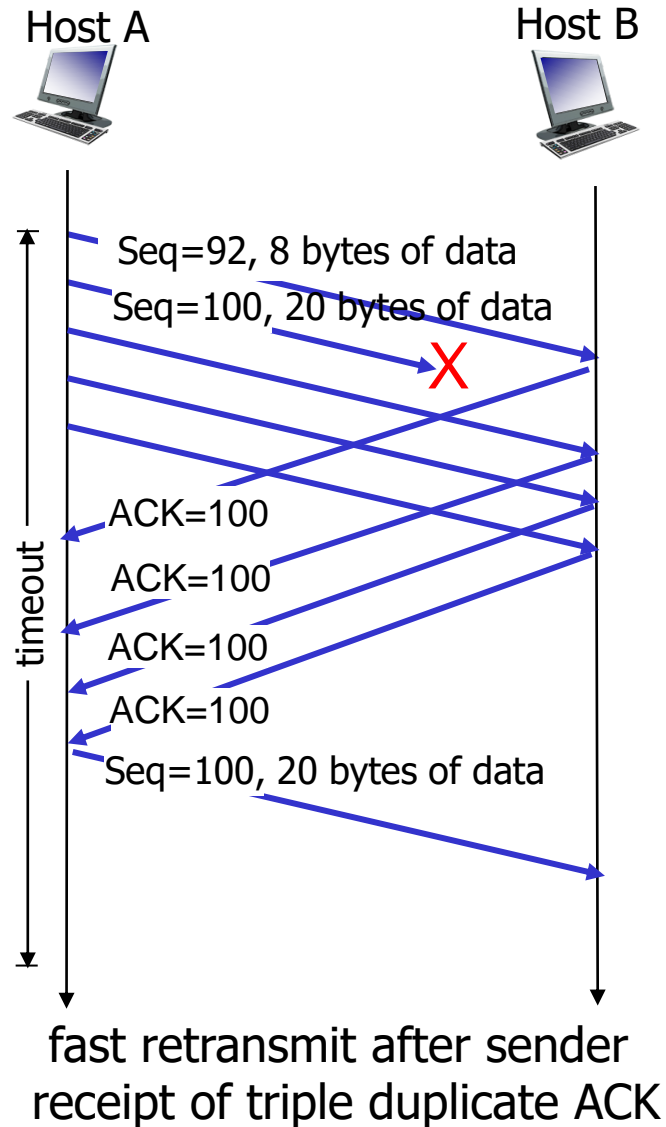
- ❖ time-out period often relatively long:
  - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

## *TCP fast retransmit*

if sender receives 3 duplicate ACKs for (“triple duplicate ACKs”), (“triple duplicate ACKs”), resend unacked segment with smallest seq #

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

# TCP fast retransmit





# Example

R15. Suppose Host A sends two TCP segments back to back to Host B over a TCP connection. The first segment has sequence number 90; the second has sequence number 100.

- a. How much data is in the first segment?
- b. Suppose that the first segment is lost but the second segment arrives at B. In the acknowledgement that Host B sends to Host A, what will be the acknowledgement number?

# 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

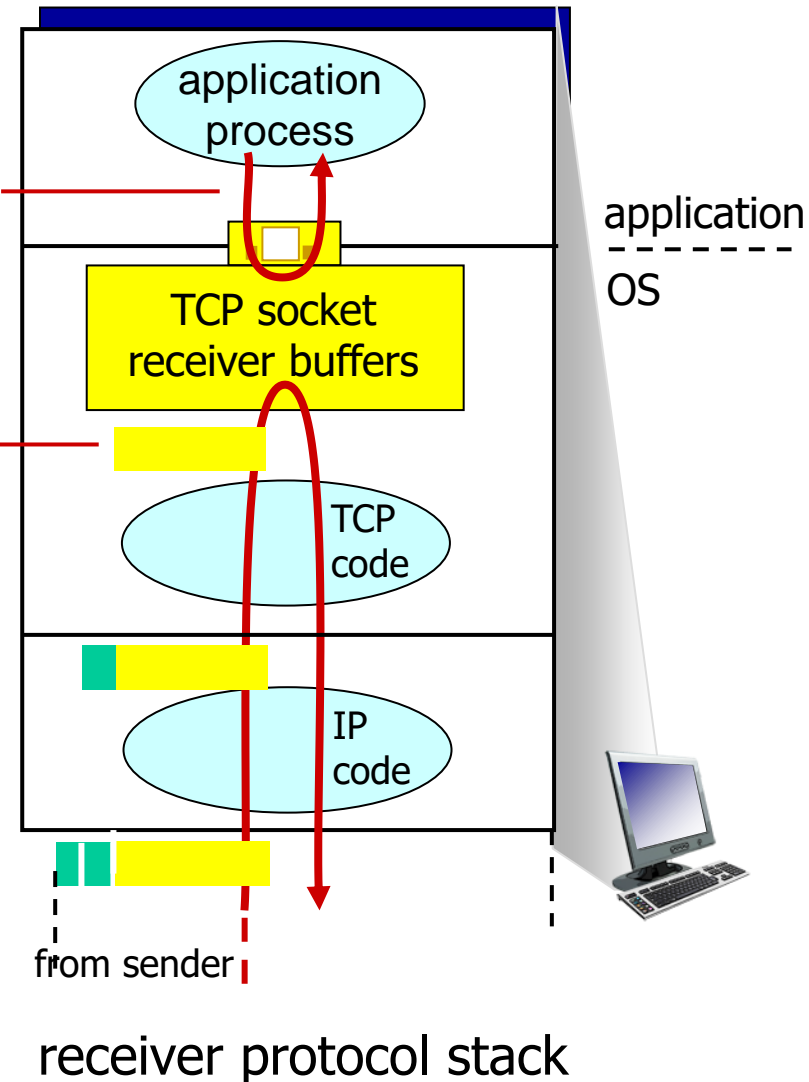
3.7 TCP congestion control

# TCP flow control

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

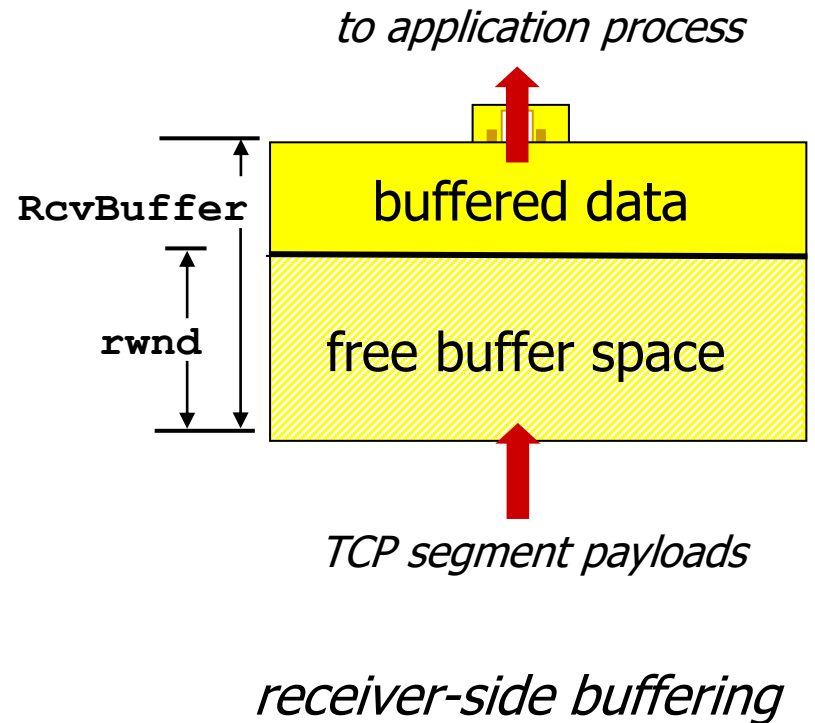
... slower than TCP  
receiver is delivering  
(sender is sending)

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



# 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



# 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

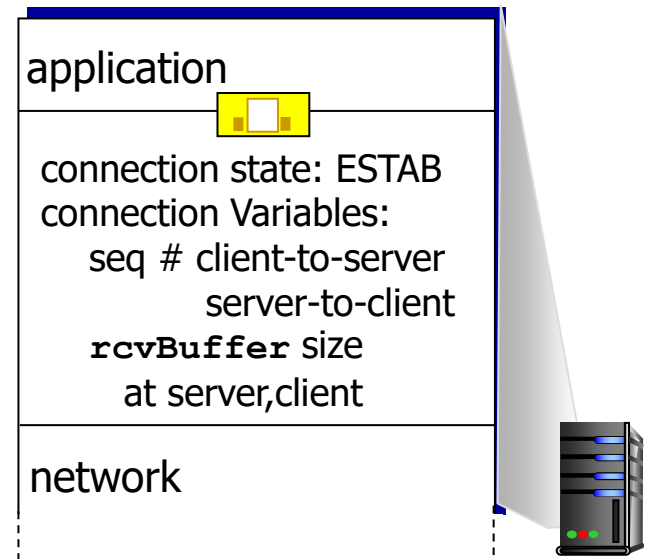
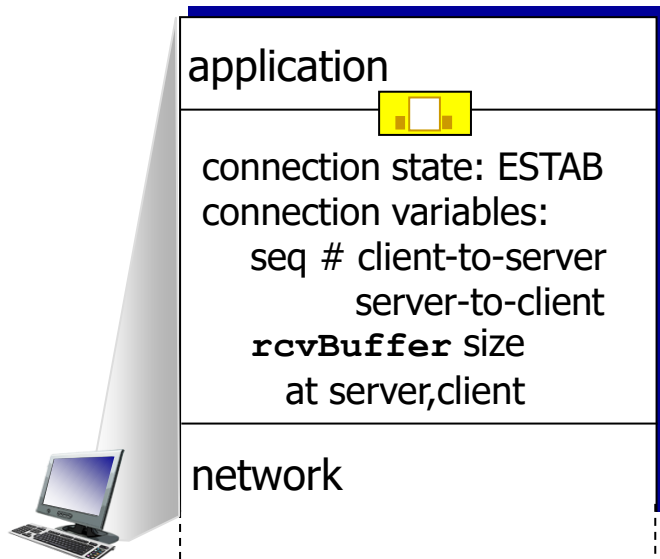
3.6 principles of congestion control

3.7 TCP congestion control

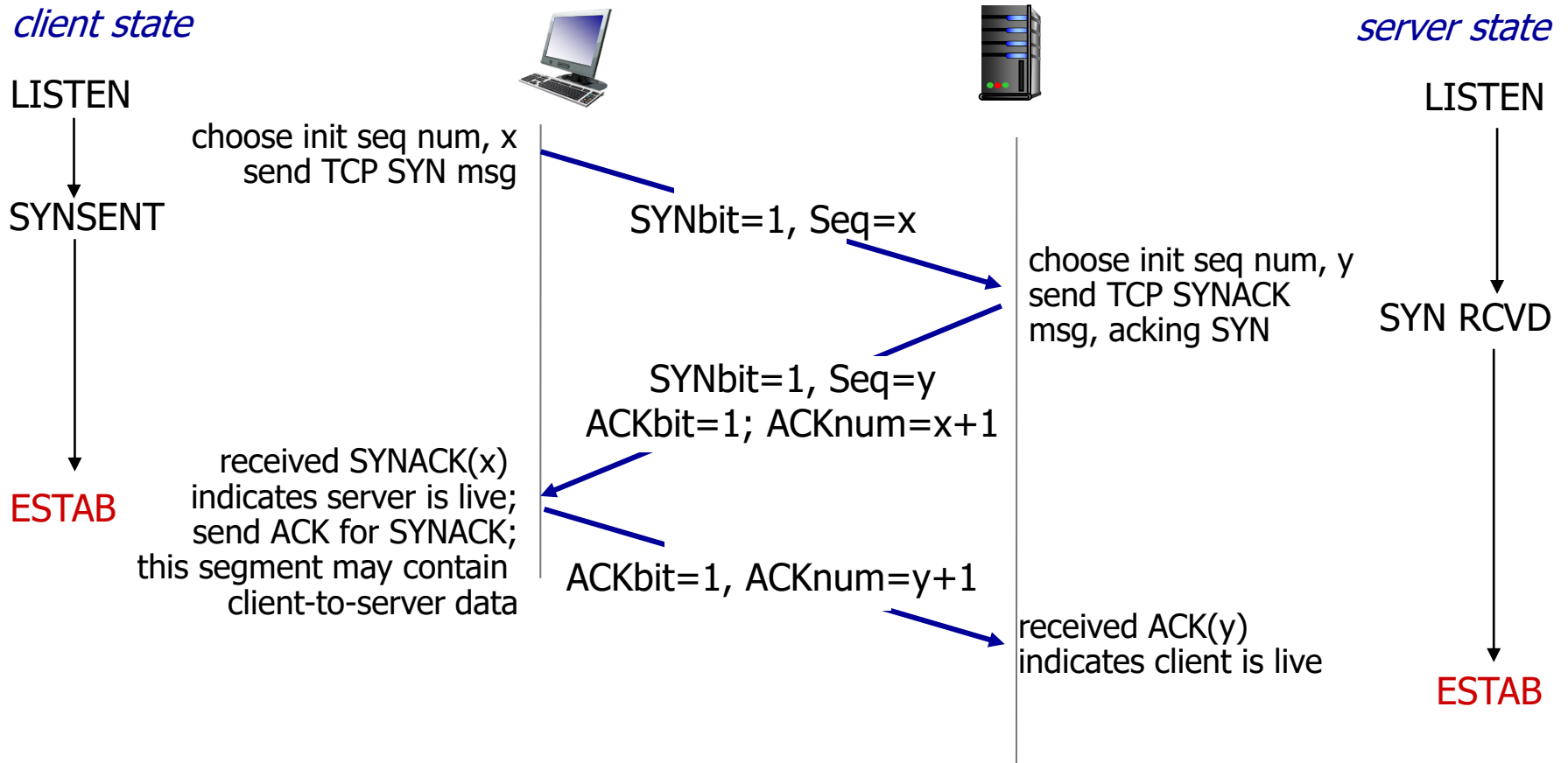
# Connection Management

before exchanging data, sender/receiver “handshake”:

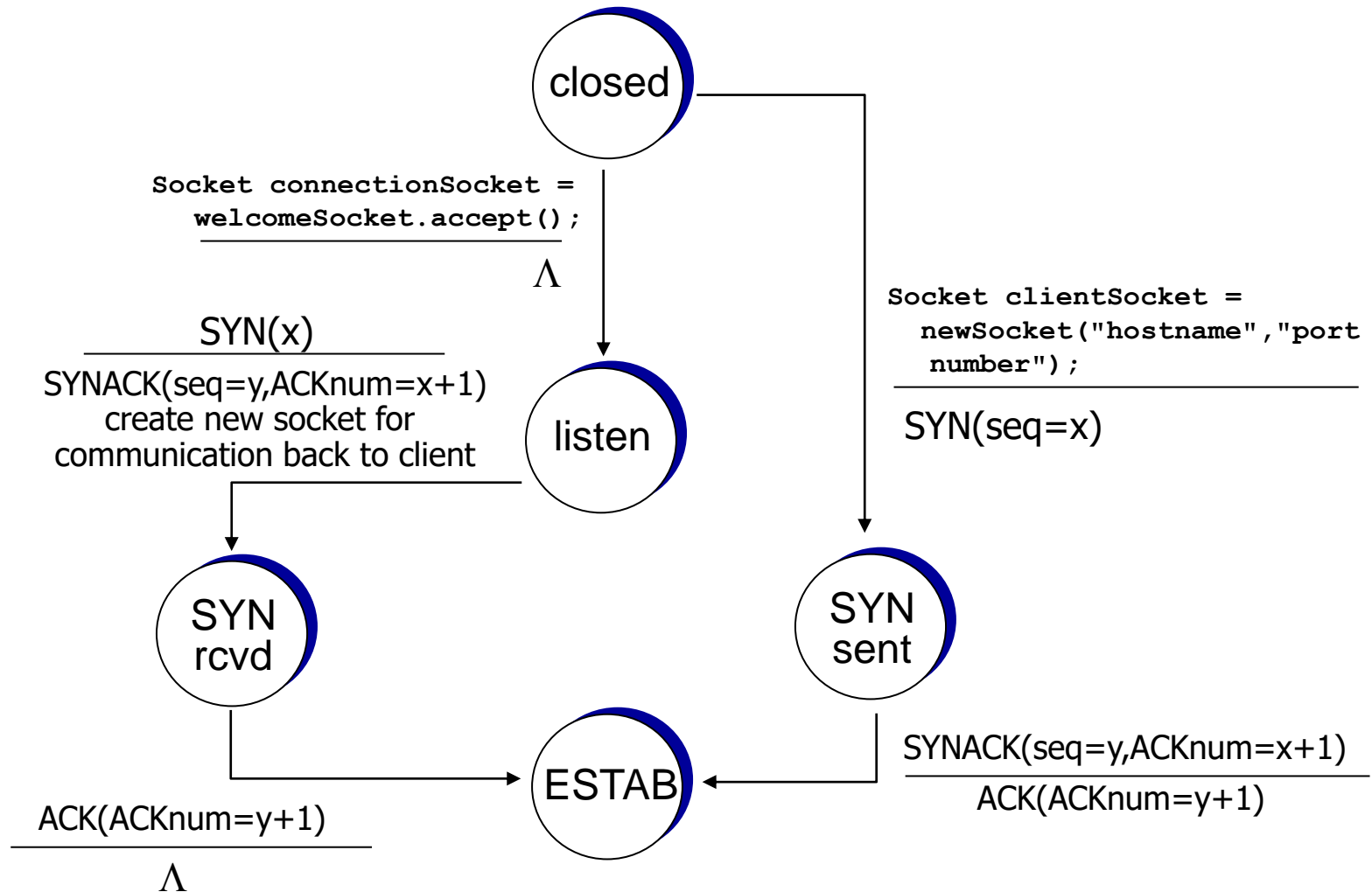
- ❖ agree to establish connection (each knowing the other willing to establish connection)
- ❖ agree on connection parameters



# TCP 3-way handshake



# TCP 3-way handshake: FSM





# 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  $2 * \text{max}$   
segment lifetime

CLOSED



FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

can still  
send data

can no longer  
send data

*server state*

ESTAB

CLOSE\_WAIT

LAST\_ACK

CLOSED

# 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

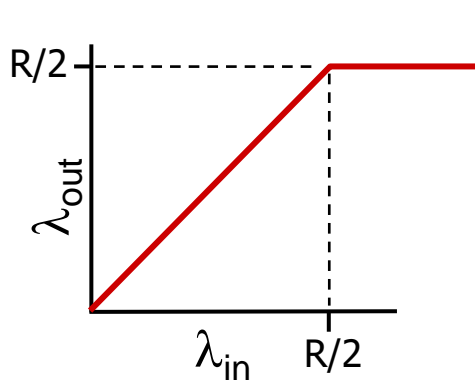
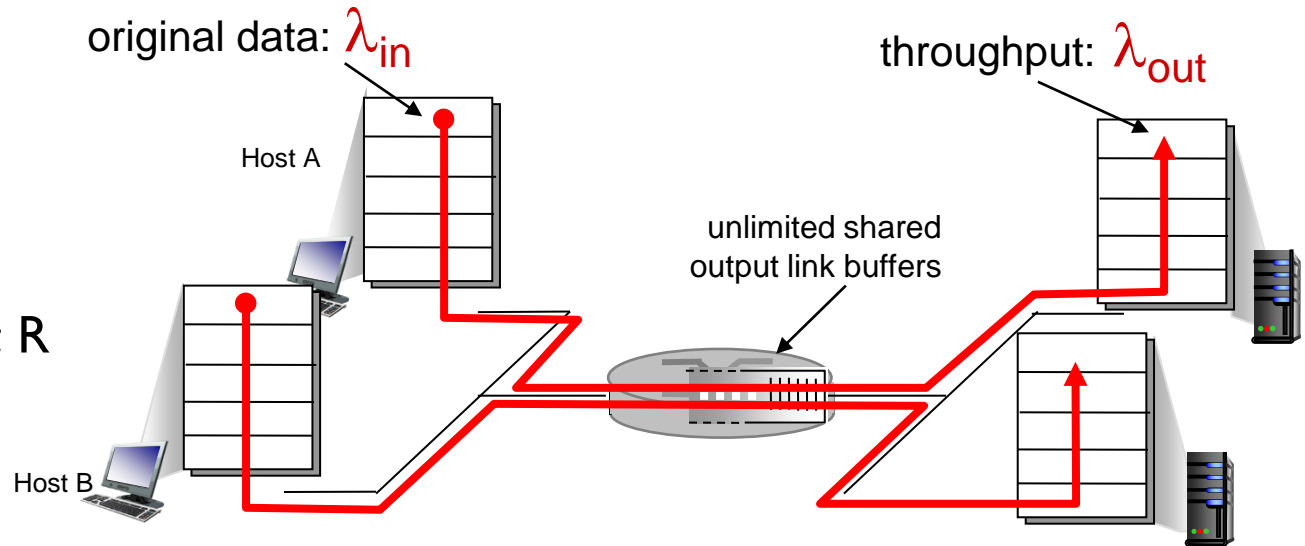
# Principles of congestion control

## *congestion:*

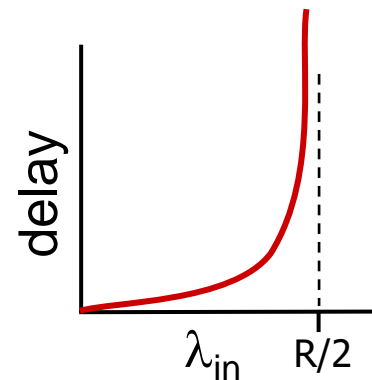
- ❖ informally: “too many sources sending too much data too fast for *network* to handle”
- ❖ different from flow control!
- ❖ manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

# Causes/costs of congestion: scenario I

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



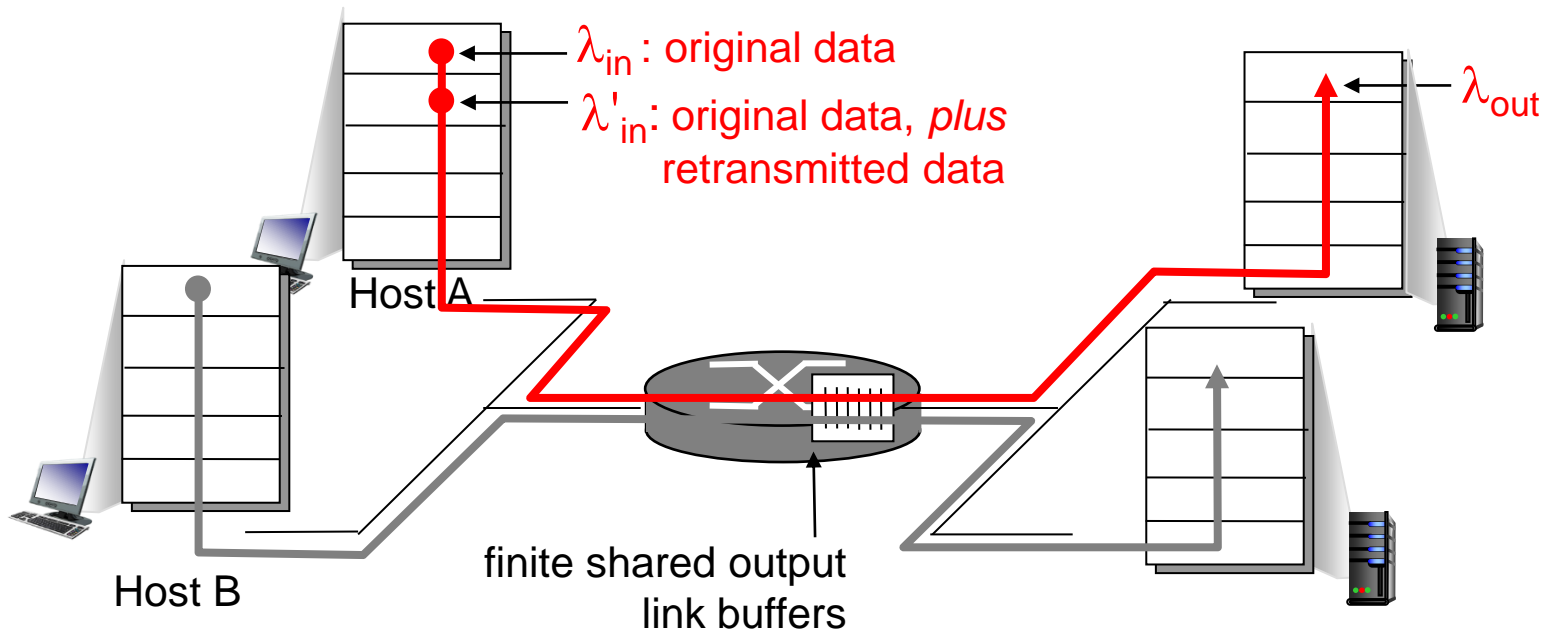
- ❖ maximum per-connection throughput:  $R/2$



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

# Causes/costs of congestion: scenario 2

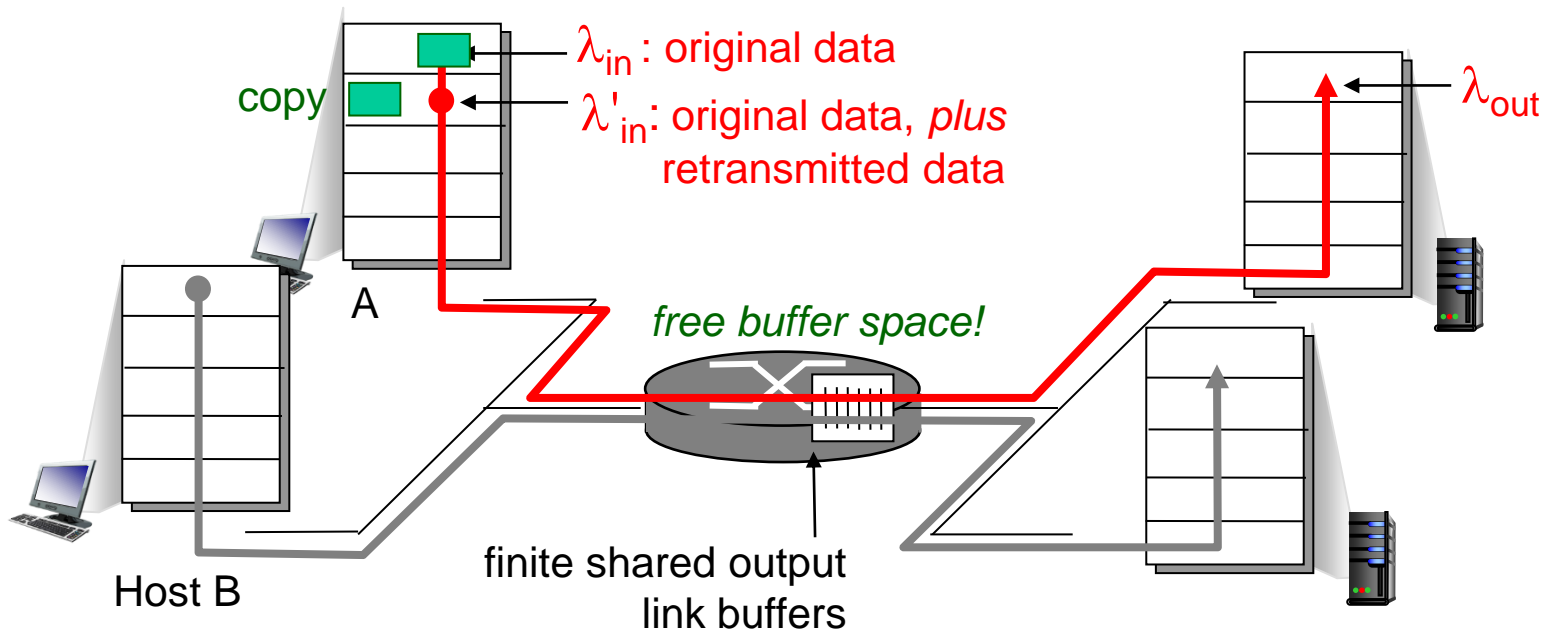
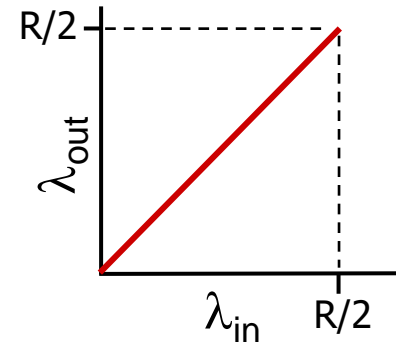
- ❖ one router, *finite* buffers
- ❖ sender retransmission of timed-out packet
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- ❖ sender sends only when router buffers available

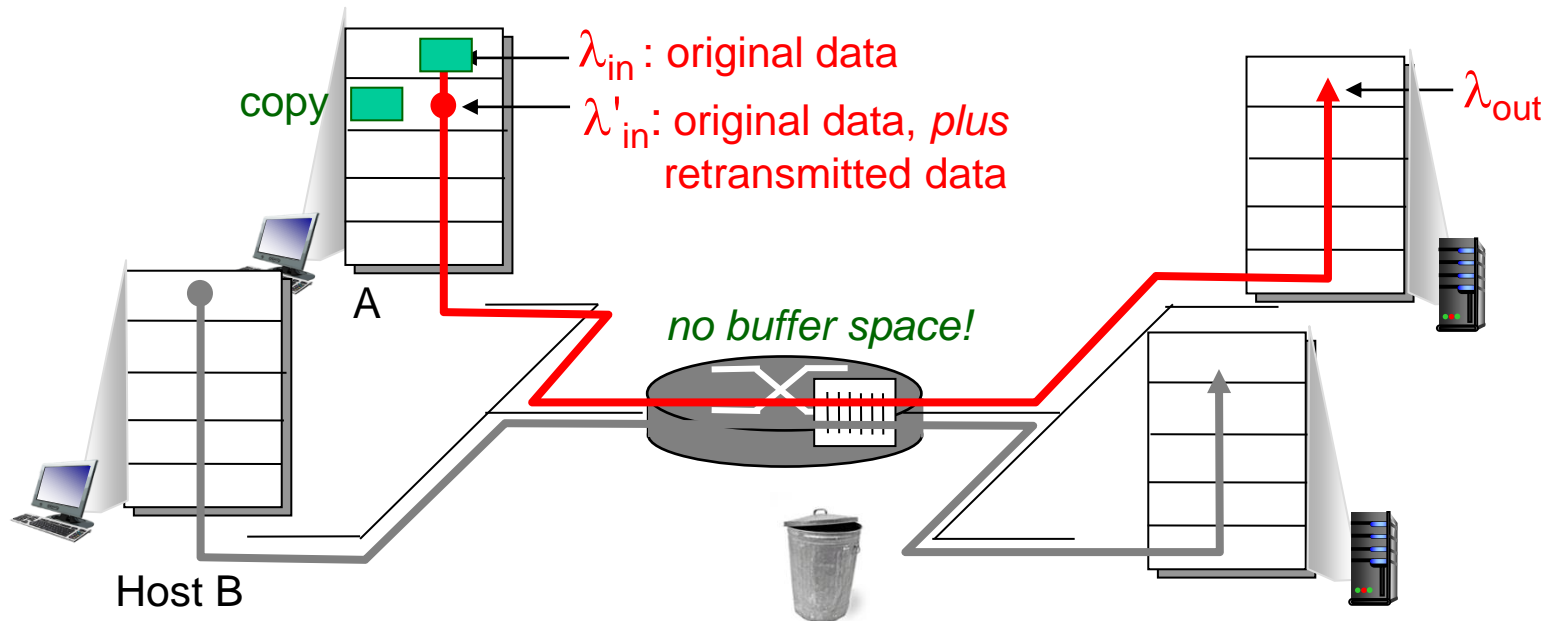


# Causes/costs of congestion: scenario 2

## *Idealization: known loss*

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

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



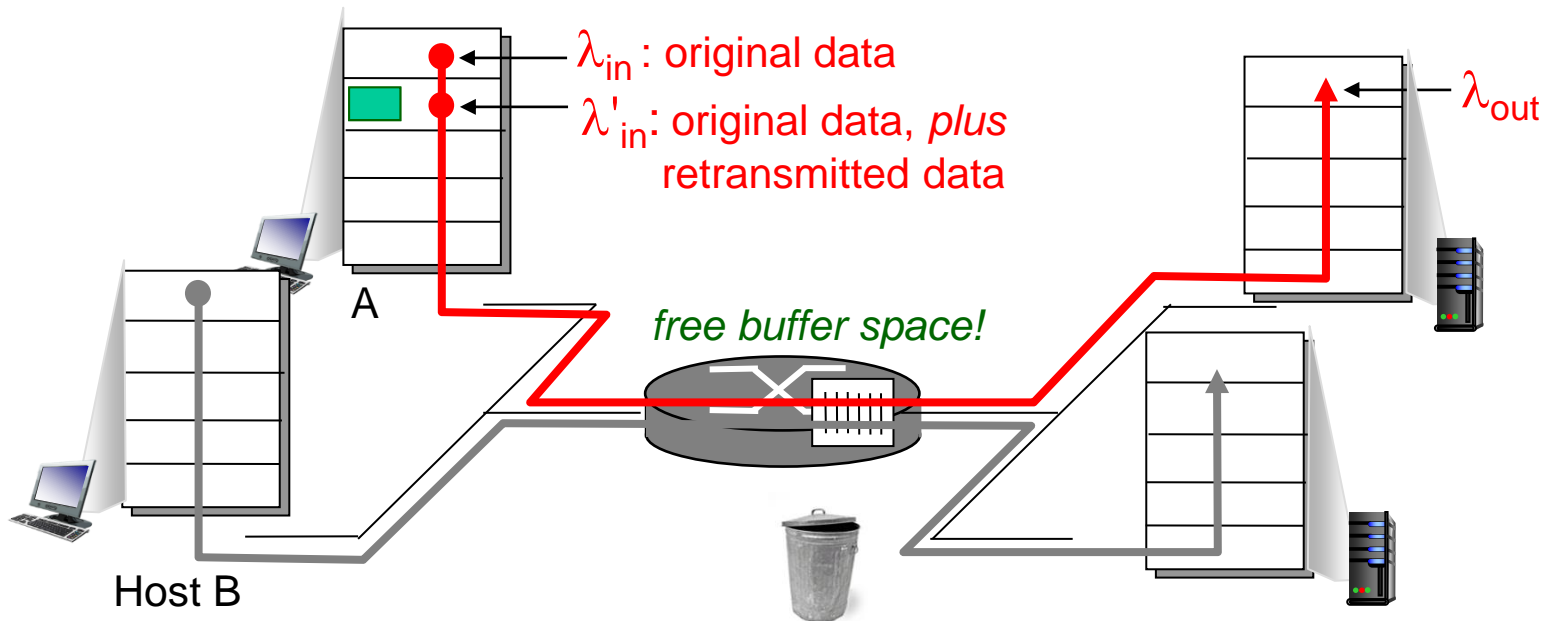
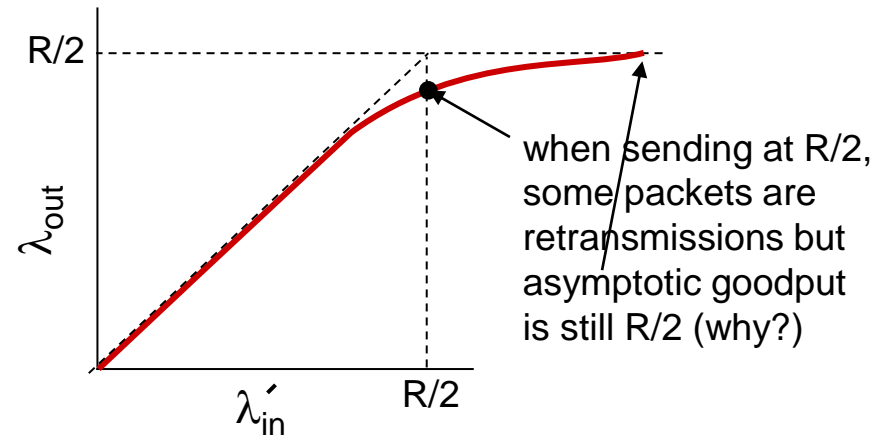


# Causes/costs of congestion: scenario 2

## *Idealization: known loss*

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

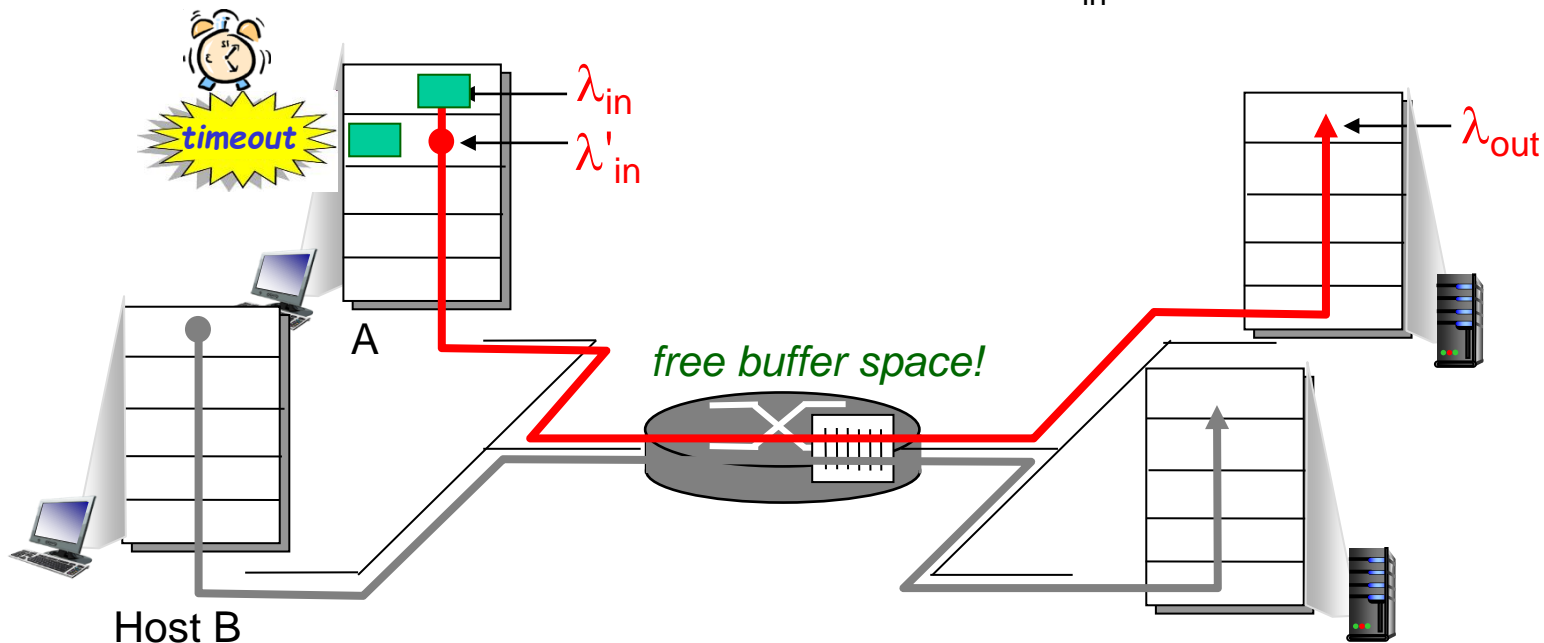
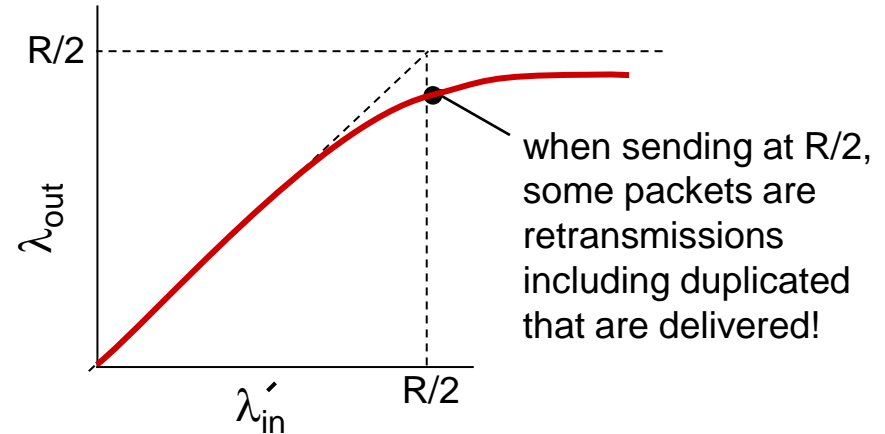
- ❖ sender only resends if  
packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic: *duplicates*

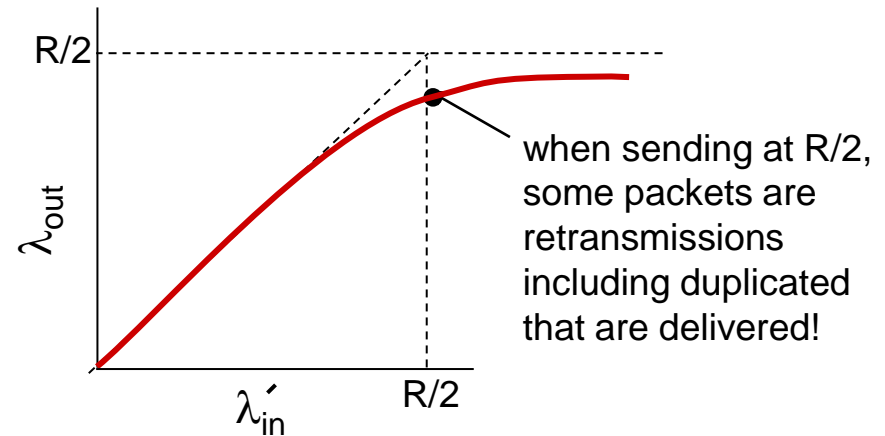
- ❖ packets can be lost, dropped at router due to full buffers
- ❖ sender times out prematurely, sending *two* copies, both of which are delivered



# 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



## “costs” of congestion:

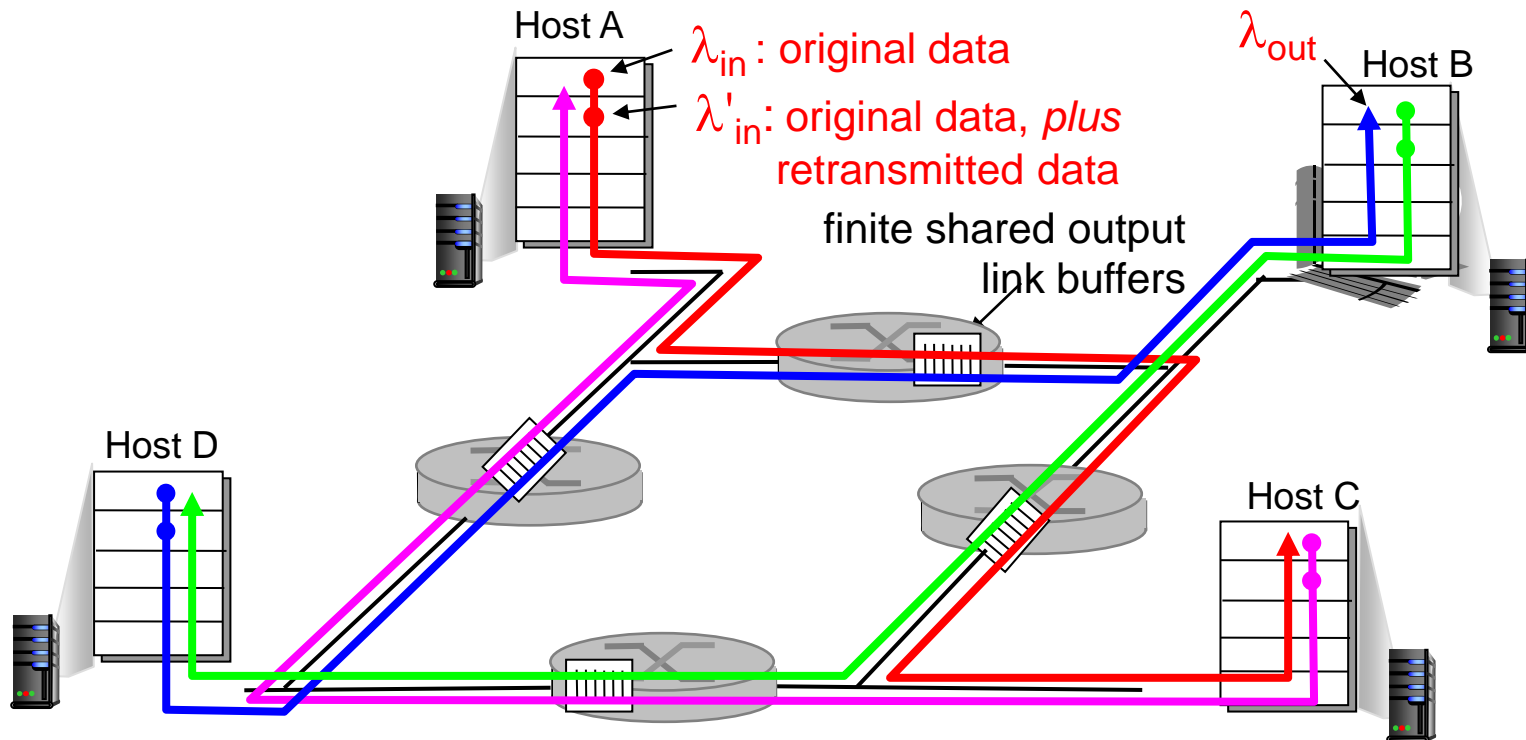
- ❖ more work (retrans) for given “goodput”
- ❖ unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

# Causes/costs of congestion: scenario 3

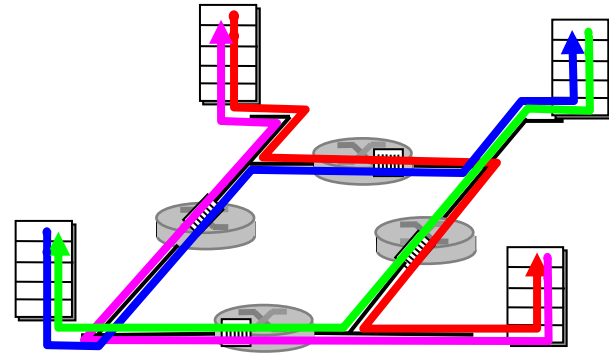
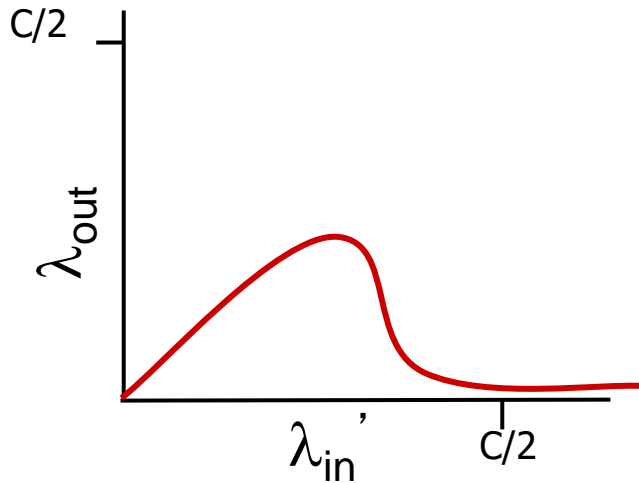
- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit

Q: what happens as  $\lambda_{in}$  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 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

# Case study: ATM ABR congestion control

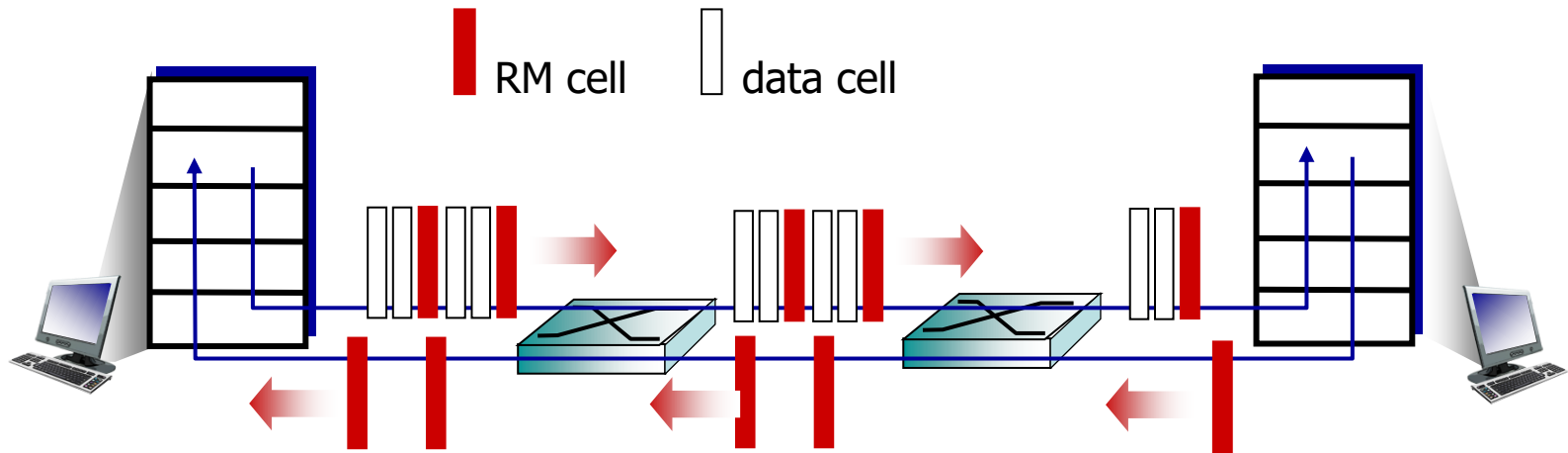
## ABR: available bit rate:

- ❖ “elastic service”
- ❖ if sender's path “underloaded”:
  - sender should use available bandwidth
- ❖ if sender's path congested:
  - sender throttled to minimum guaranteed rate

## RM (resource management) cells:

- ❖ sent by sender, interspersed with data cells
- ❖ bits in RM cell set by switches (“*network-assisted*”)
  - *NI bit*: no increase in rate (mild congestion)
  - *CI bit*: congestion indication
- ❖ RM cells returned to sender by receiver, with bits intact

# Case study: ATM ABR congestion control



- ❖ two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- ❖ EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell



# 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

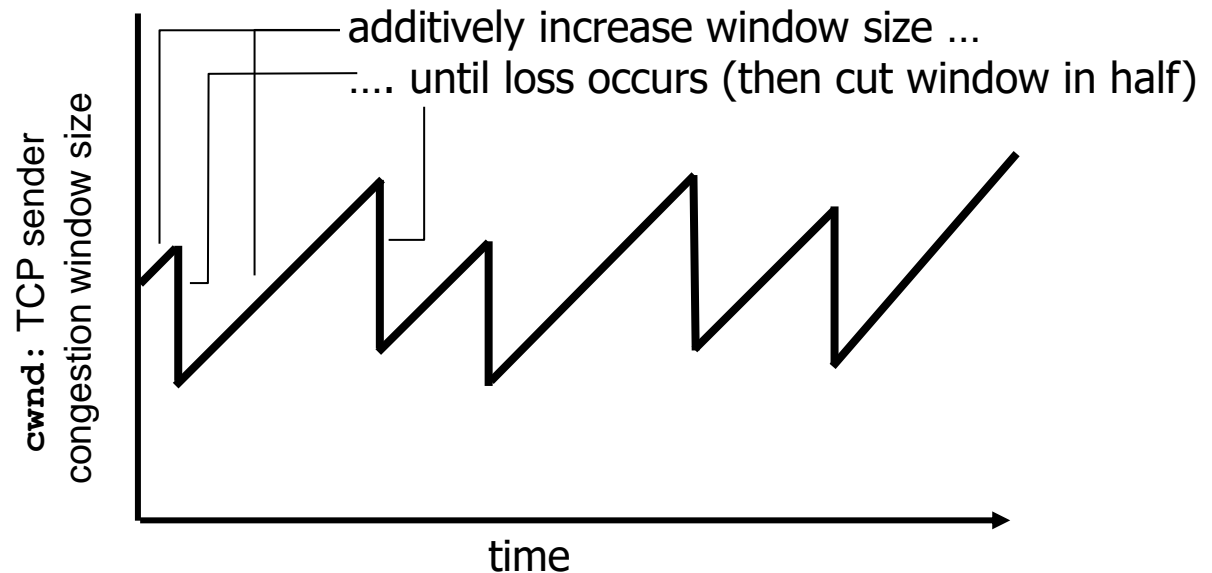
3.6 principles of congestion control

3.7 TCP congestion control

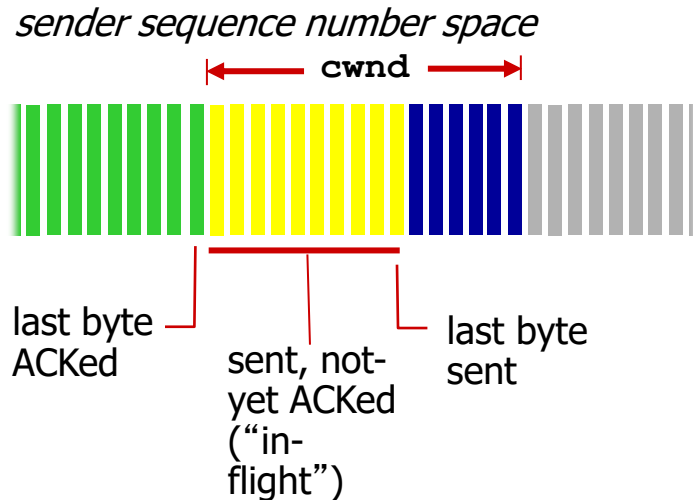
# TCP congestion control: additive increase multiplicative decrease

- ❖ *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



# TCP Congestion Control: details



- ❖ sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- ❖ **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 Congestion Control

## ❖ Phases:

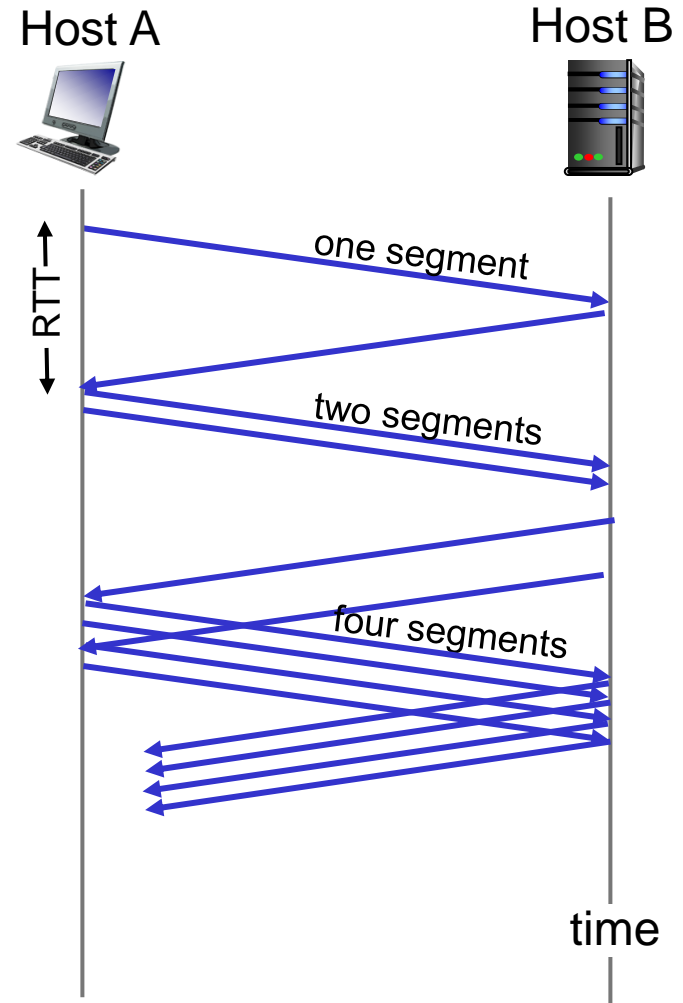
- Slow start
- Congestion avoidance
- Reaction to congestion

## ❖ Parameters:

- cwnd: congestion window size
- threshold: slow start threshold

# Slow Start

- ❖  $cwnd < threshold$ 
  - when connection begins
  - after congestion
- ❖ increase rate exponentially until first loss event:
  - initially  $cwnd = 1 \text{ MSS}$
  - double  $cwnd$  every RTT
  - done by incrementing  $cwnd$  for every ACK received



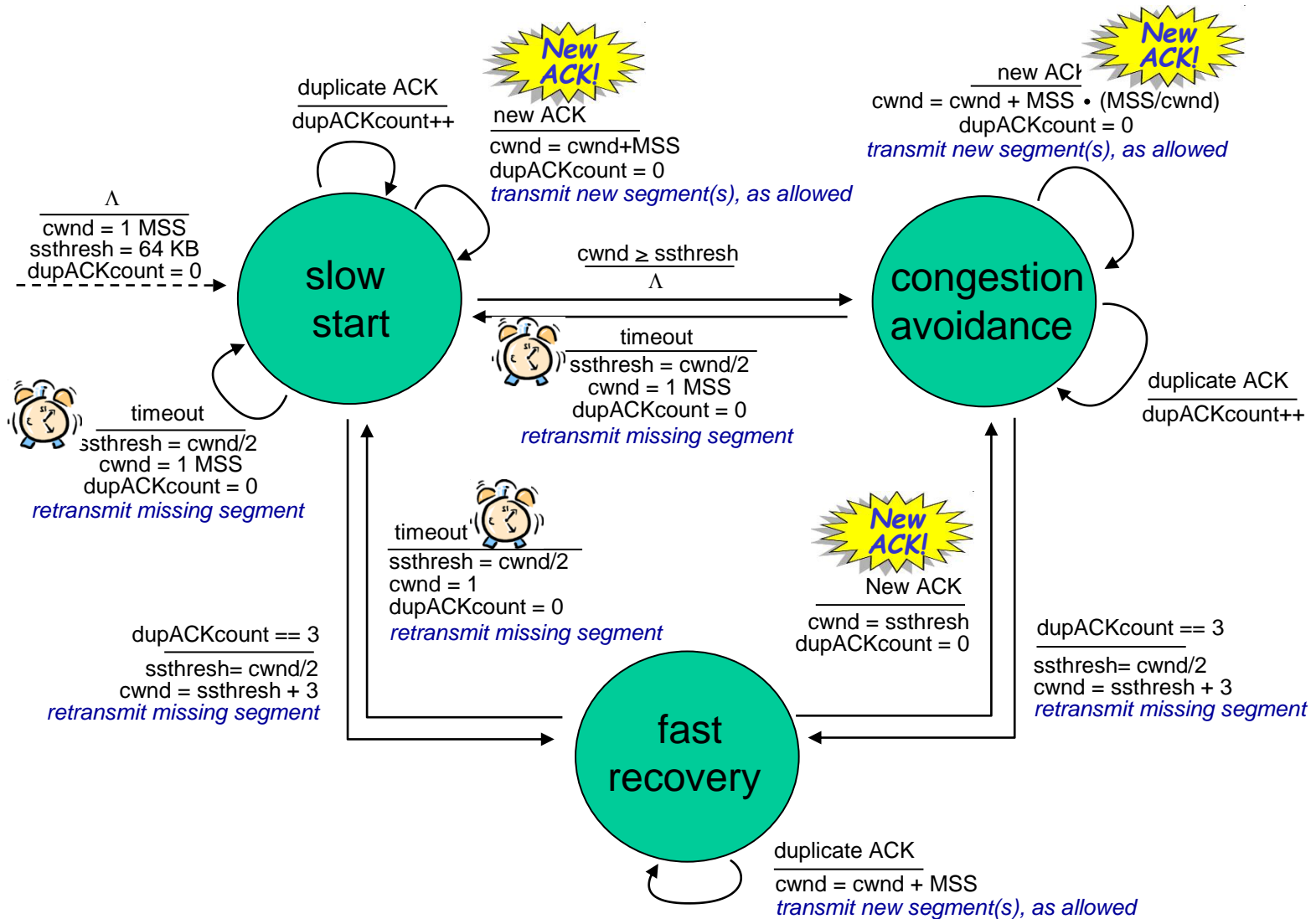
# Congestion Avoidance

- ❖ **cwnd**  $\geq$  threshold: transition from slow start to congestion avoidance phase
- ❖ In the congestion avoidance phase **cwnd** is increased linearly

# 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)

# Summary: TCP Congestion Control





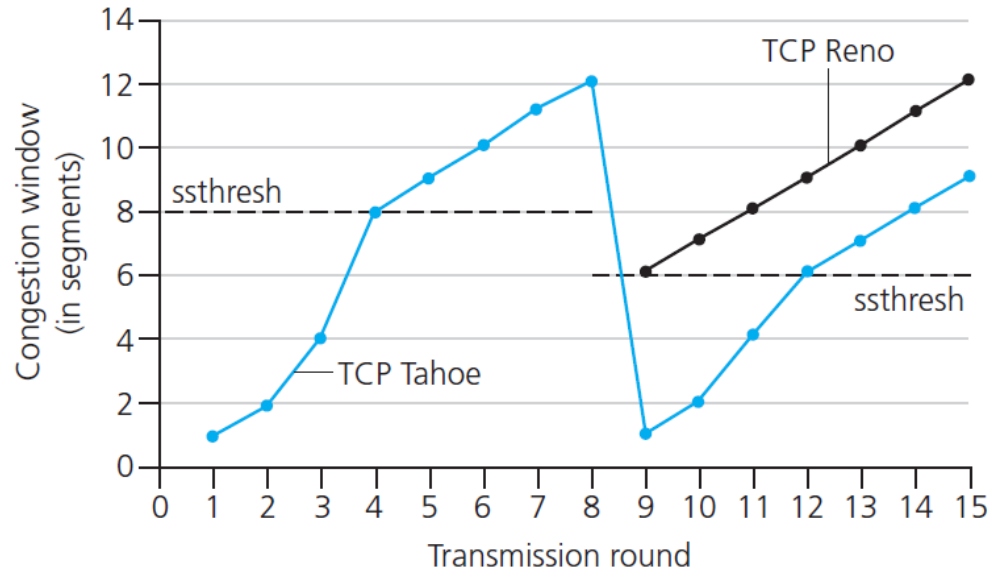
# TCP: switching from slow start to CA

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

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

## Implementation:

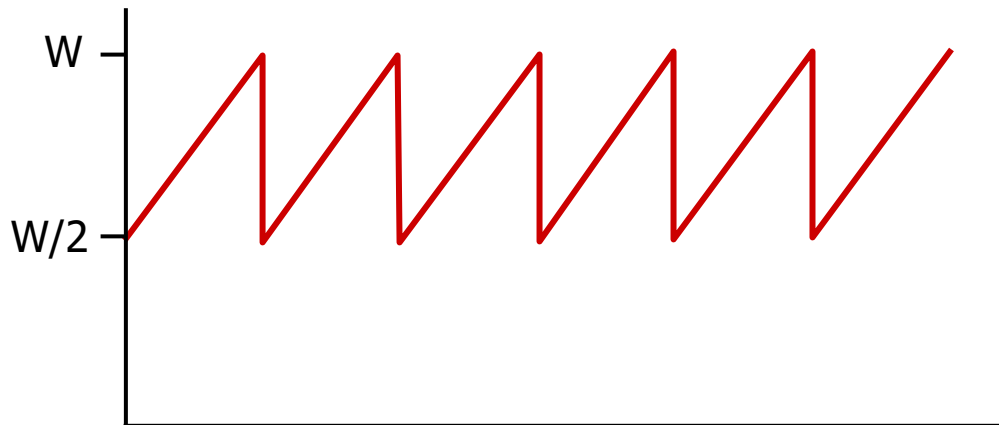
- ❖ variable **ssthresh**
- ❖ on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



# TCP throughput

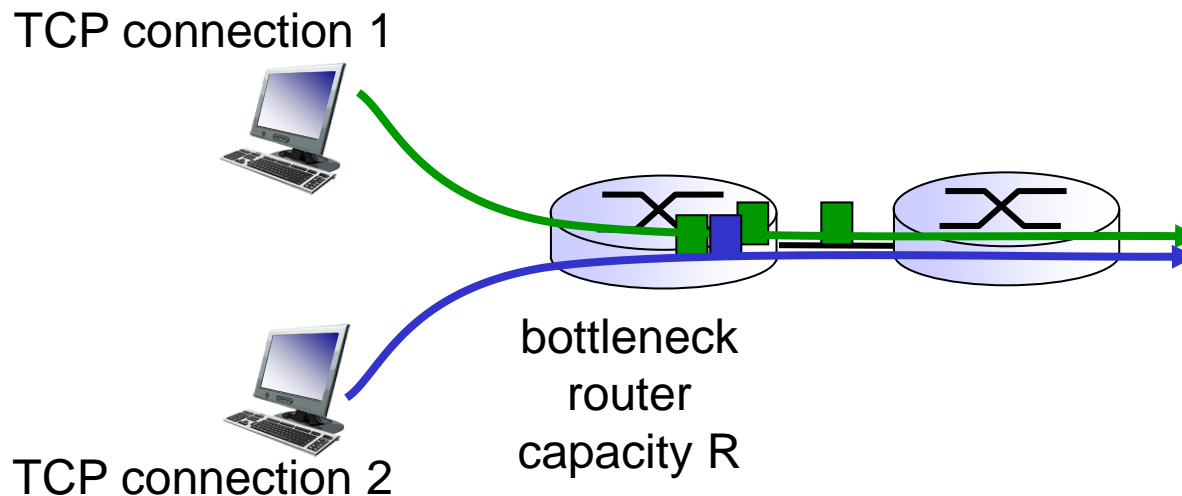
- ❖ avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- ❖ **W: window size** (measured in bytes) **where loss occurs**
  - avg. window size (# in-flight bytes) is  $W \frac{3}{4}$
  - avg. thruput is  $W \frac{3}{4}$  per RTT

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



# TCP Fairness

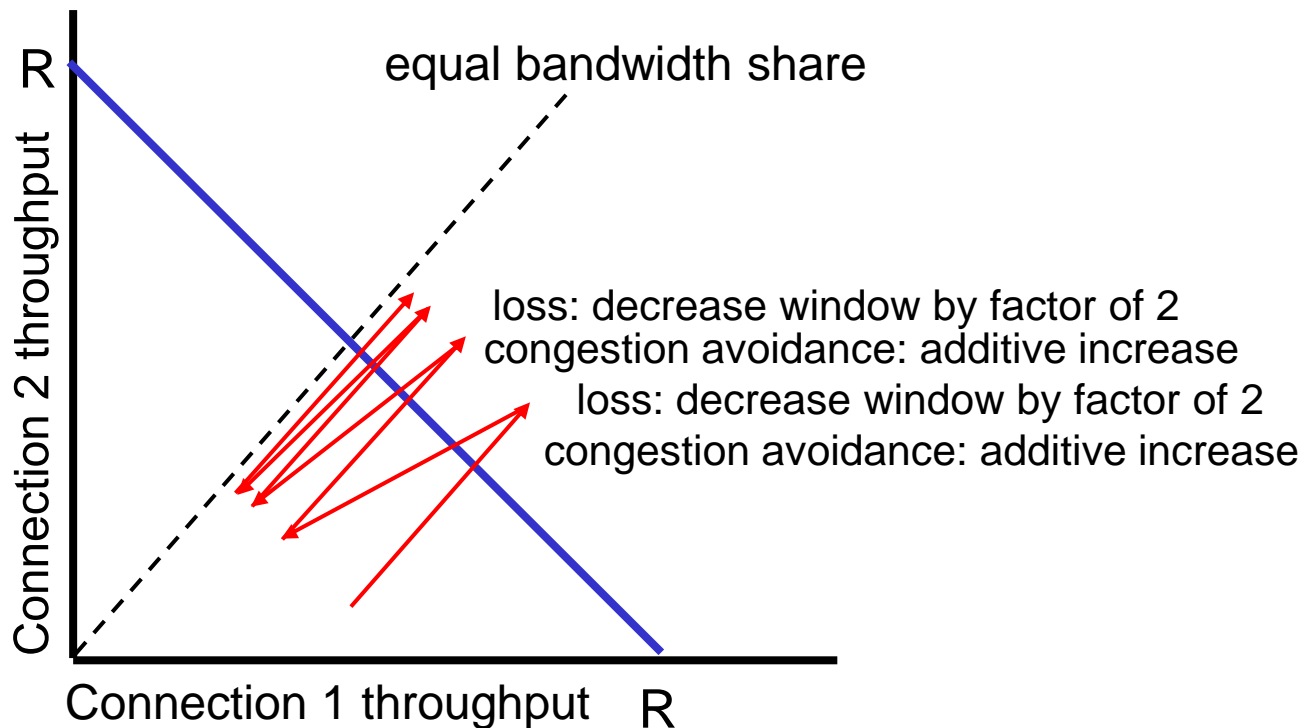
*fairness goal:* if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Why is TCP fair?

two competing sessions:

- ❖ additive increase gives slope of 1, as throughput 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  $R/2$

# Chapter 3: summary

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

## next:

- ❖ leaving the network “edge” (application, transport layers)
- ❖ into the network “core”