

Computer Networks

@CS.NCTU

Lecture 3: Transportation (TCP/UDP)

Instructor: Kate Ching-Ju Lin (林靖茹)

Slides modified from

“Computer Networking: A Top-Down Approach” 7th Edition

Outline

- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - Segment structure
 - Connection management
 - Reliable data transfer
 - Flow control
 - Congestion Control

TCP: Overview

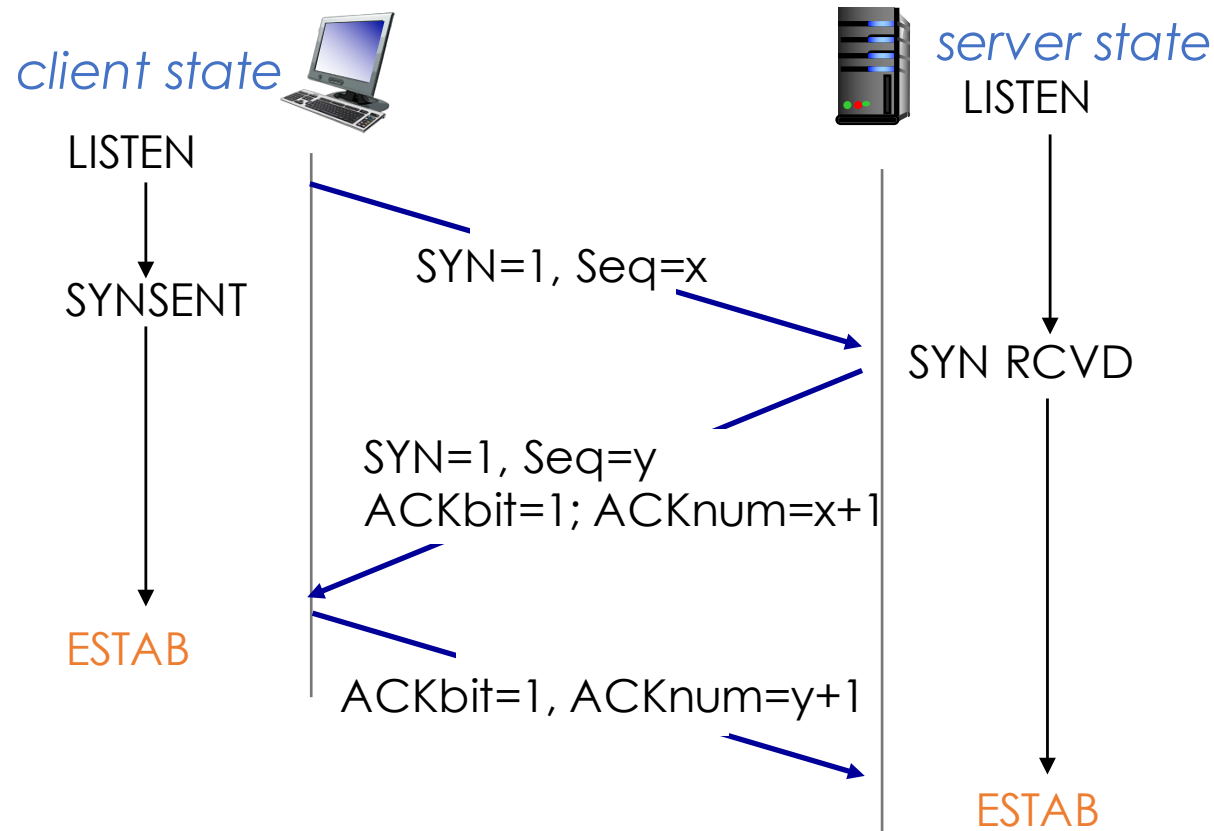
RFCs: 793,1122,1323, 2018, 2581

- **Full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- **Connection-oriented:**
 - **Three-way handshaking** (exchange of control msgs) inits sender, receiver state before data exchange
- **Flow controlled:**
 - sender will not overwhelm receiver
- **Point-to-point:**
 - one sender, one receiver
- **Reliable, in-order byte stream:**
 - no “message boundaries”
 - **Pipelined**
 - TCP congestion and flow control set window size

Outline

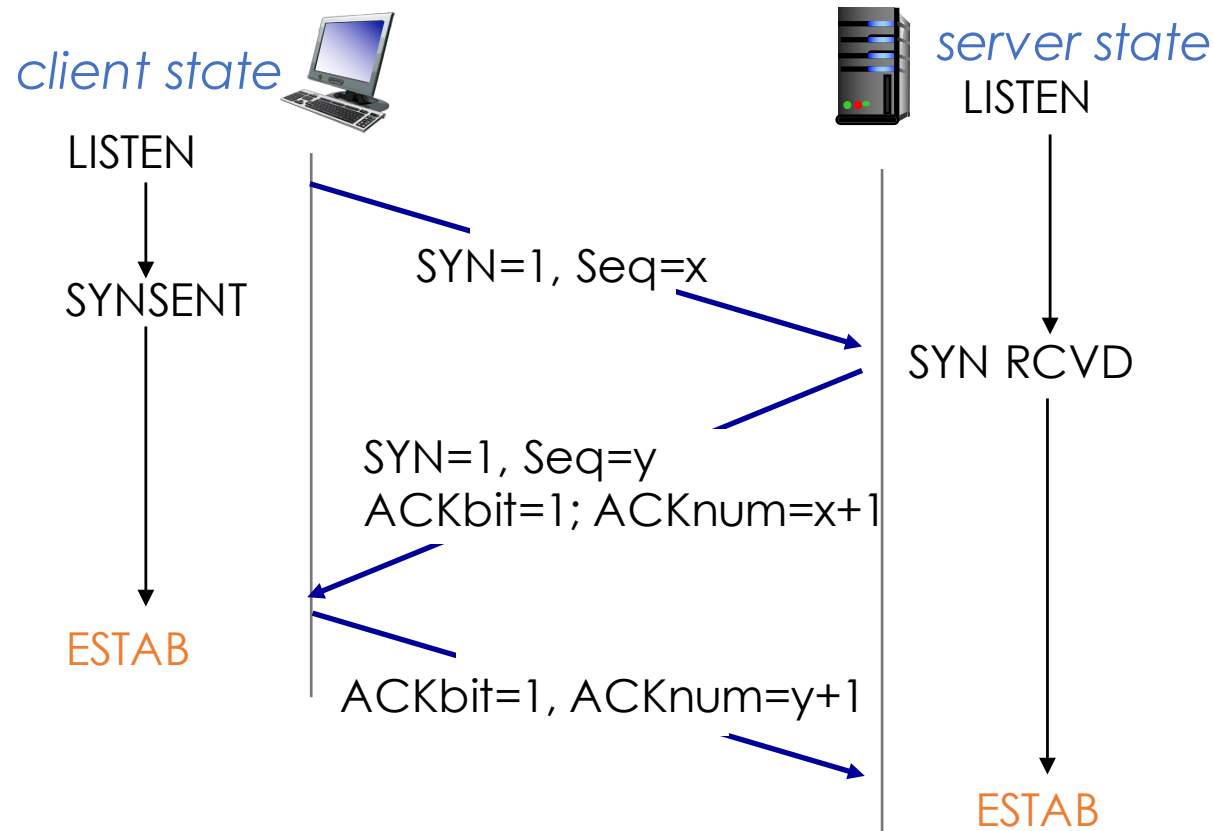
- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - **Connection management**
 - Segment structure
 - Reliable data transfer
 - Flow control
 - Congestion Control

Three-Way Handshaking



- msg1: SYN = 1, Seq x (from C) randomly selected
- msg2: SYN = 1, Seq y (from S) randomly selected
ACK# = x+1 (ready to recv next one)
- msg3: ACK# = y + 1 (ready to receive next one)

Three-Way Handshaking



- Why 3-way, not 2-way?
 - Bi-directional connection: both client and server can send to another site

Connection Termination

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 $2 \times \text{max}$
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

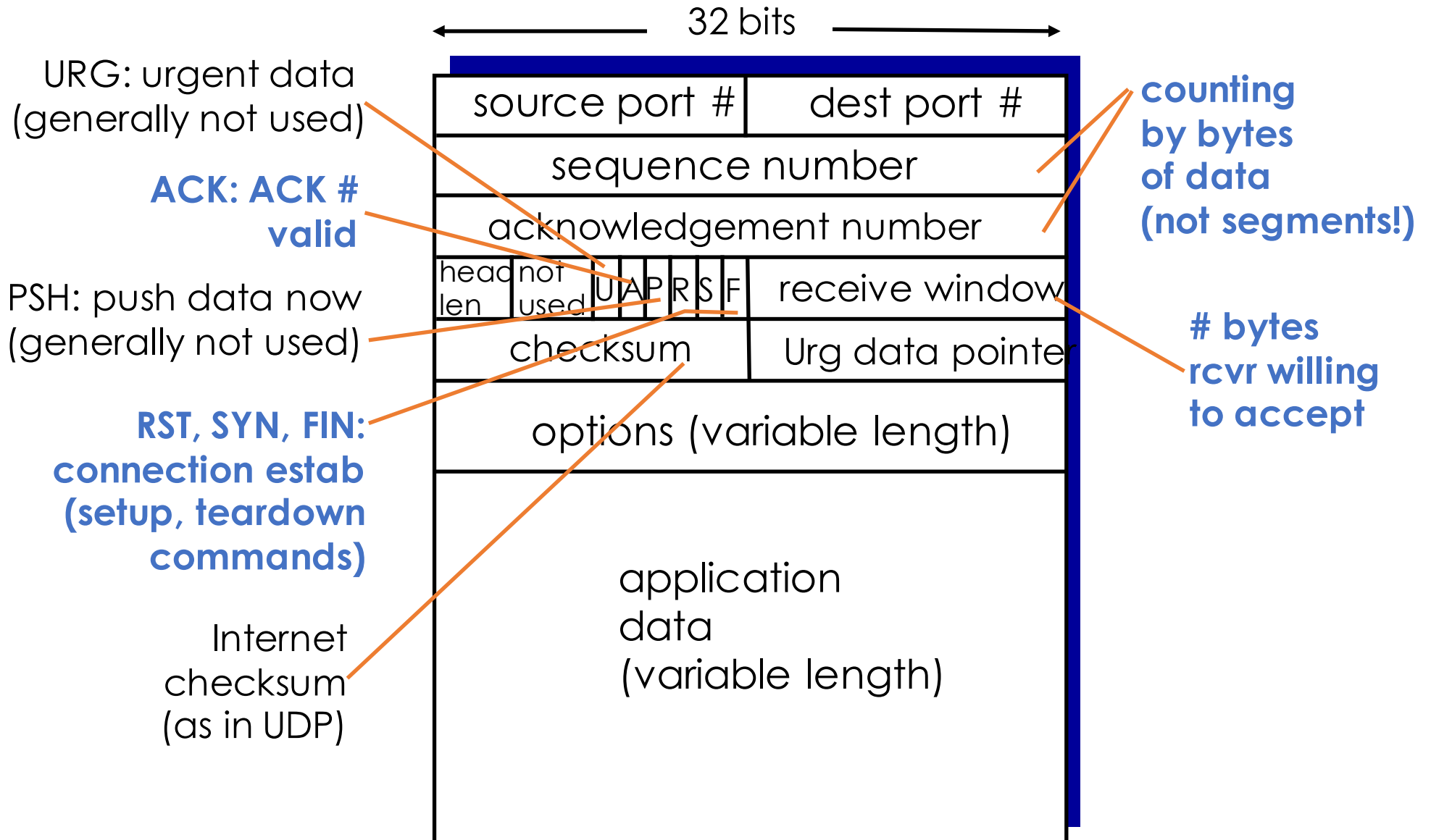
LAST_ACK

CLOSED

Outline

- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - Connection management
 - **Segment structure**
 - Reliable data transfer
 - Flow control
 - Congestion Control

TCP Segment Structure



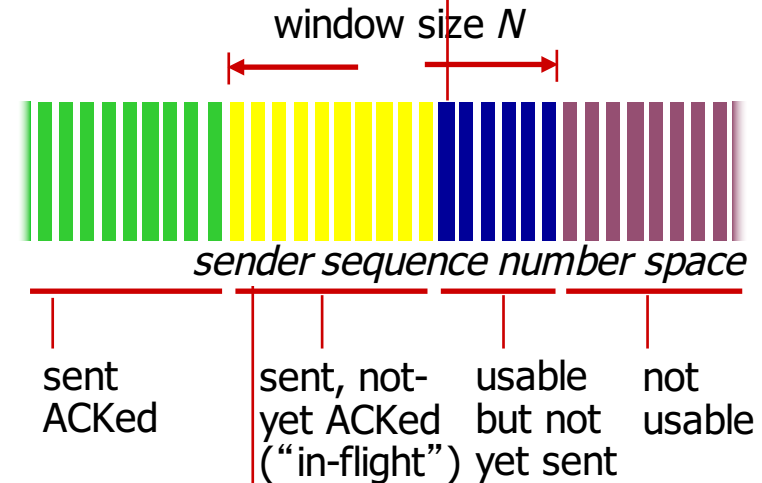
TCP Seq. Numbers, ACKs

- 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 implementer)

outgoing segment from sender

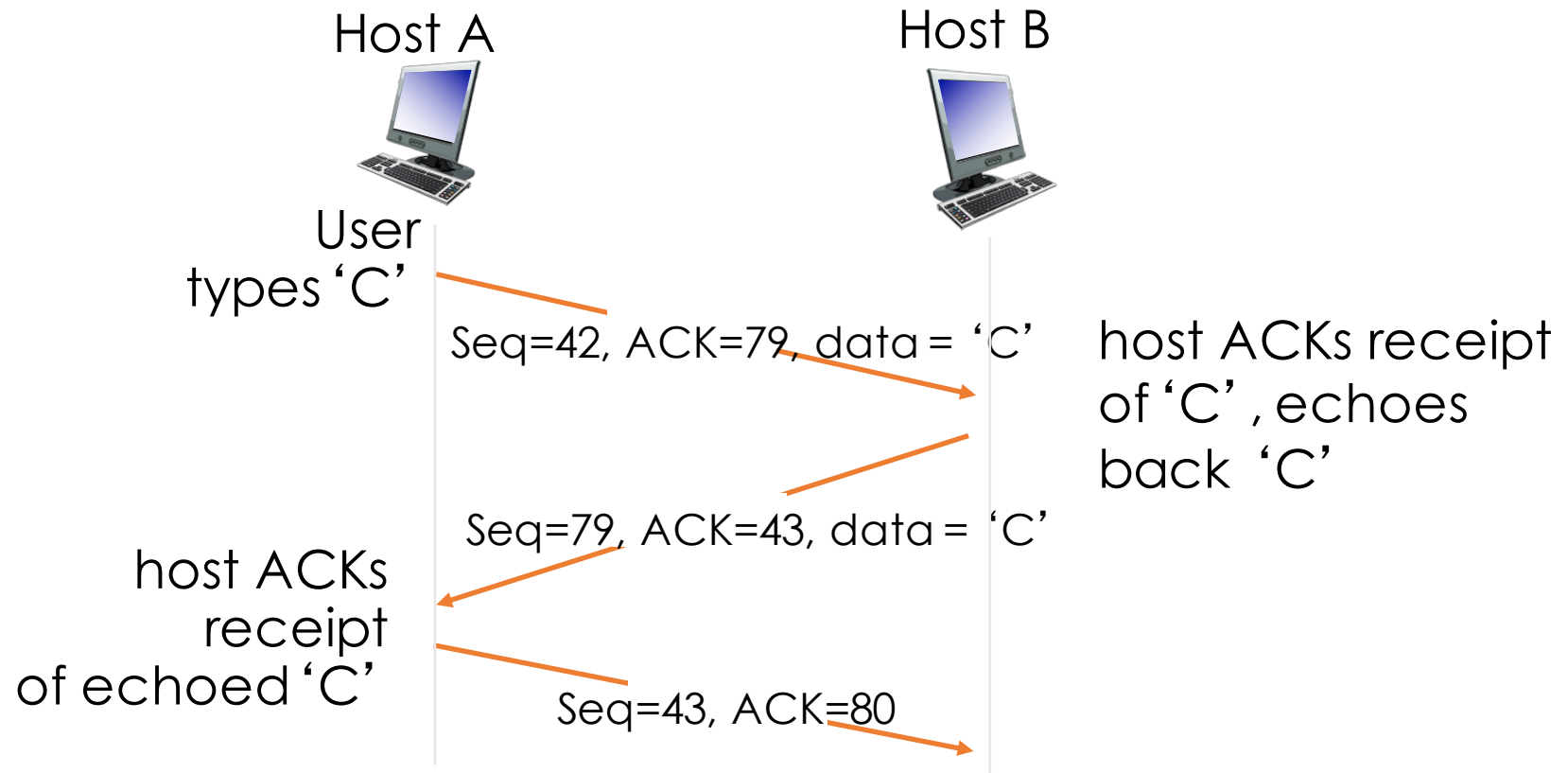
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

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

TCP Seq. Numbers, ACKs



simple telnet scenario

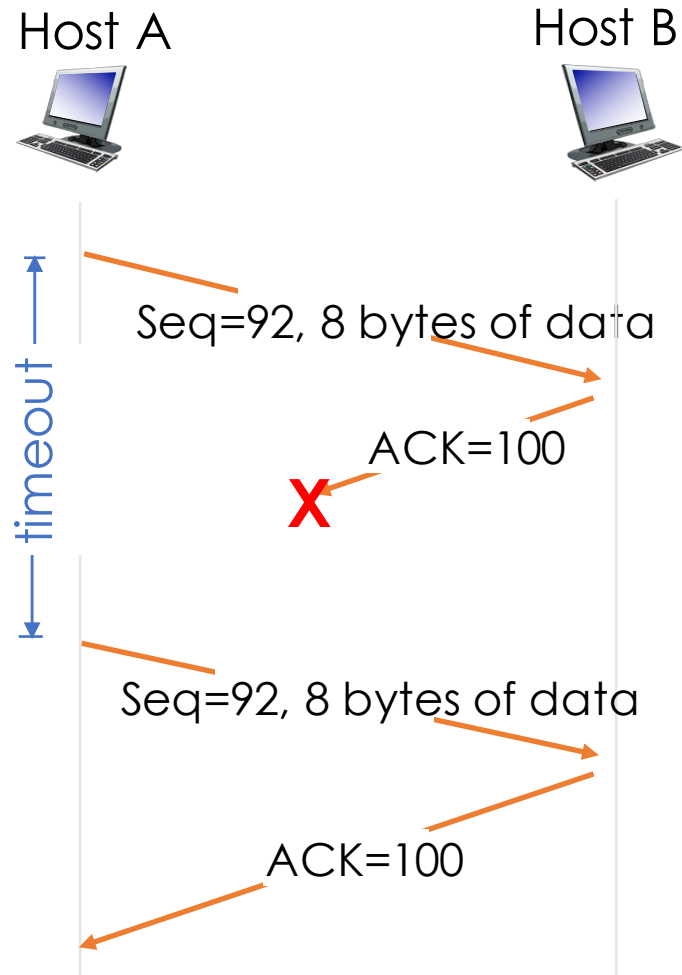
Outline

- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - Connection management
 - Segment structure
 - **Reliable data transfer**
 - Flow control
 - Congestion Control

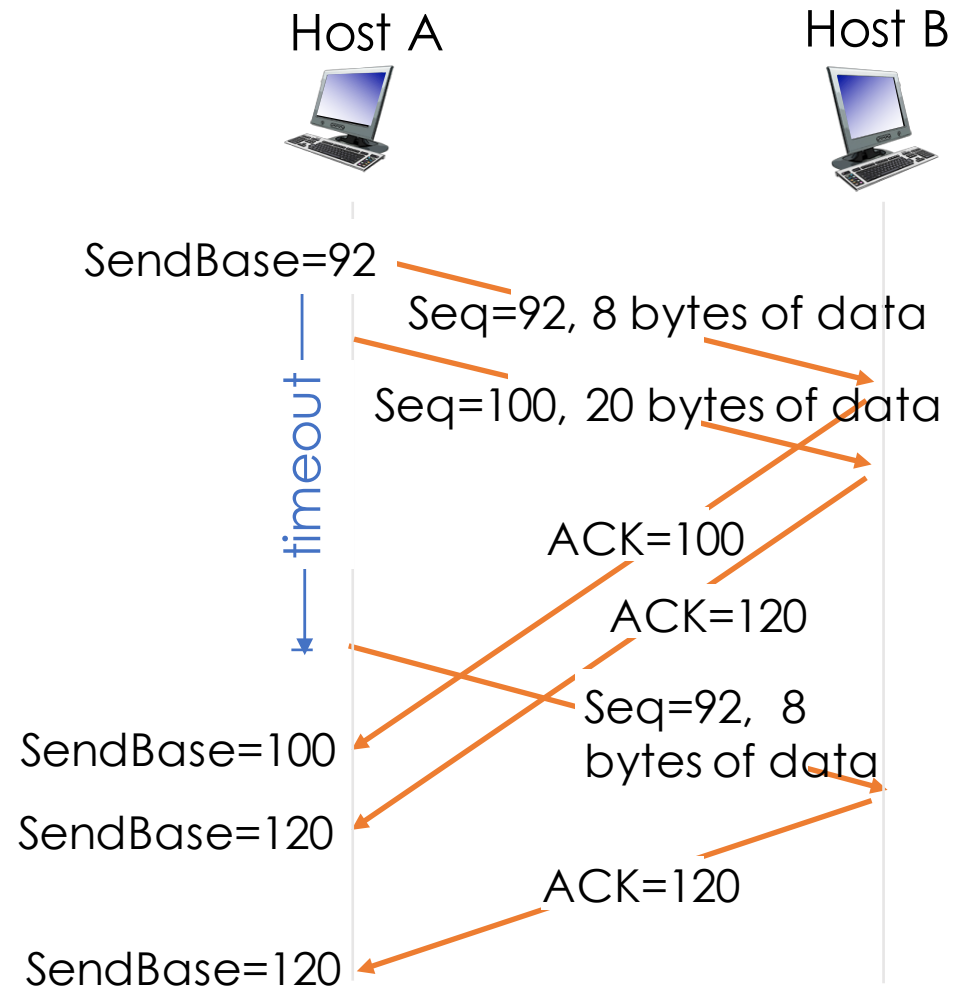
TCP Reliable Data Transfer

- 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 (fast retransmission)

TCP Retransmission Scenarios

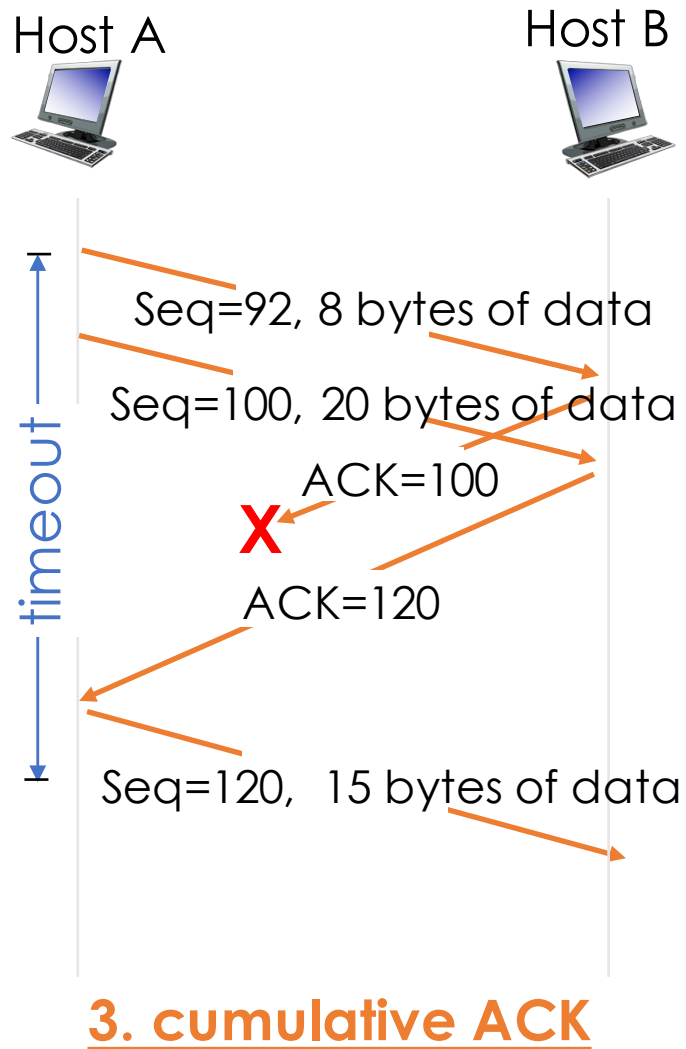


1. lost ACK scenario



2. premature timeout

TCP Retransmission Scenarios



TCP Fast Retransmit

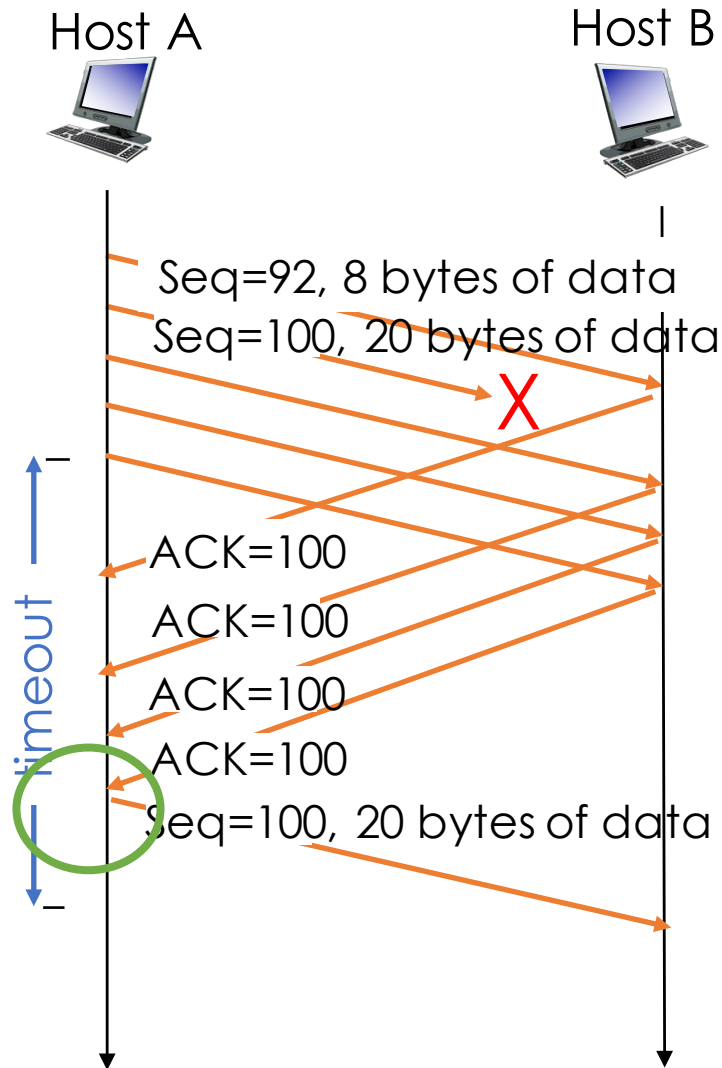
- Time-out period often relatively long:
 - long delay before resending lost packet
- Detect lost segments via duplicate ACKs
 - Sender often sends many segments back-to-back
 - If segment is lost, there will likely be many duplicate ACKs

TCP fast retransmit

if sender receives 3
ACKs for same data
→ immediately resend
unacked segment with
smallest seq #

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

TCP Fast Retransmit



Why this works?

Sender sends packets back-to-back

→ Each packet triggers an ACK

→ Dup ACK implies something lost

fast retransmit after sender receipt of 3 duplicate 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 <i>single cumulative ACK</i> , 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

Timeout Configuration

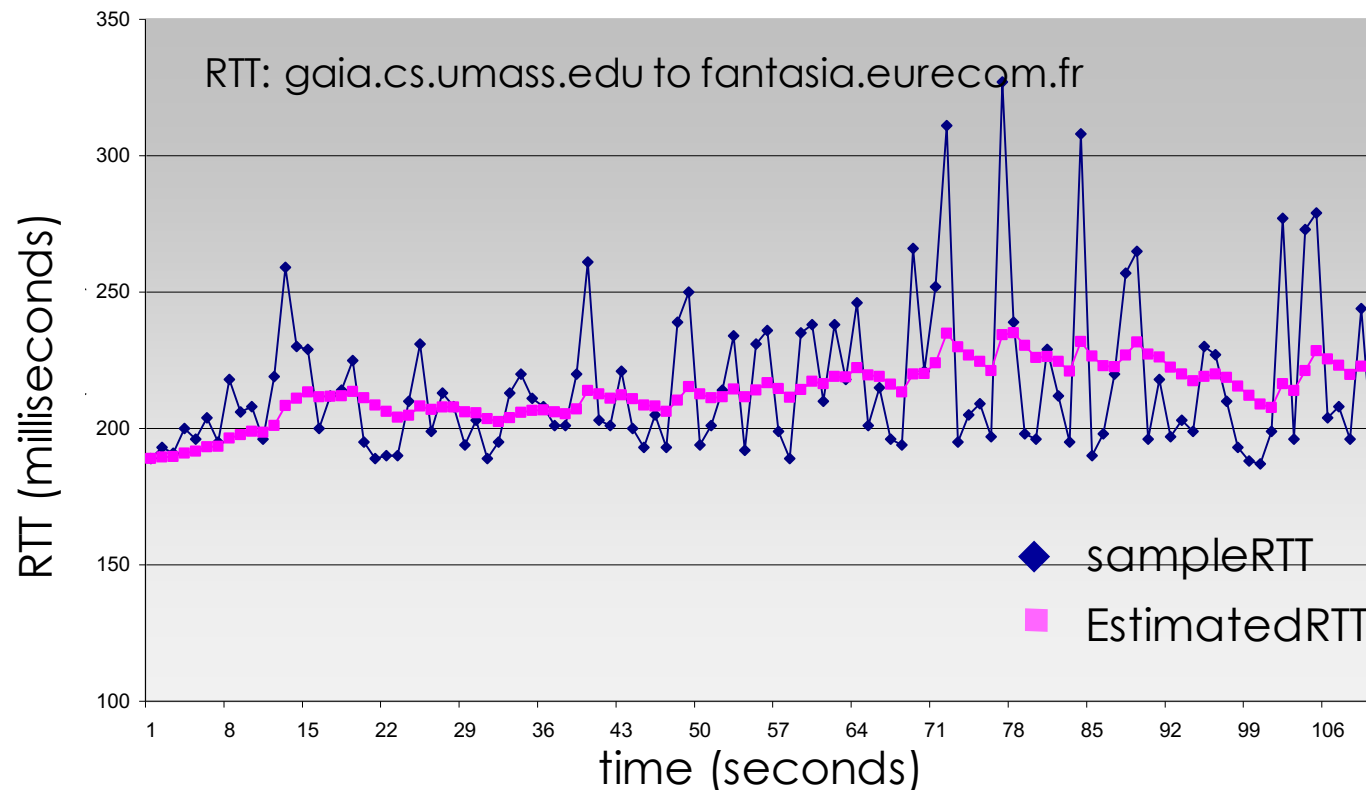
- How to configure a proper timeout that
 - Improve bandwidth utilization (short enough)
 - Avoid unnecessary retransmissions (but not so short)
- Key idea: **RTT (round trip time)**
 - Ideally, an ACK should be returned to sender after RTT
 - **Q:** how to measure RTT? What if RTT fluctuates?

Practical RTT Measurement

exponential weighted moving average (EWMA):

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

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



RTT Distribution

- Q: how to consider RTT variation?
 - timeout interval: **EstimatedRTT** plus “safety margin”
 - large variation in **EstimatedRTT** → larger safety margin

retransmission timeout interval:

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



estimated RTT “safety margin”

RTT deviation:

$$\text{DevRTT} = (1 - b) * \text{DevRTT} + b * | \text{SampleRTT} - \text{RTT} |$$

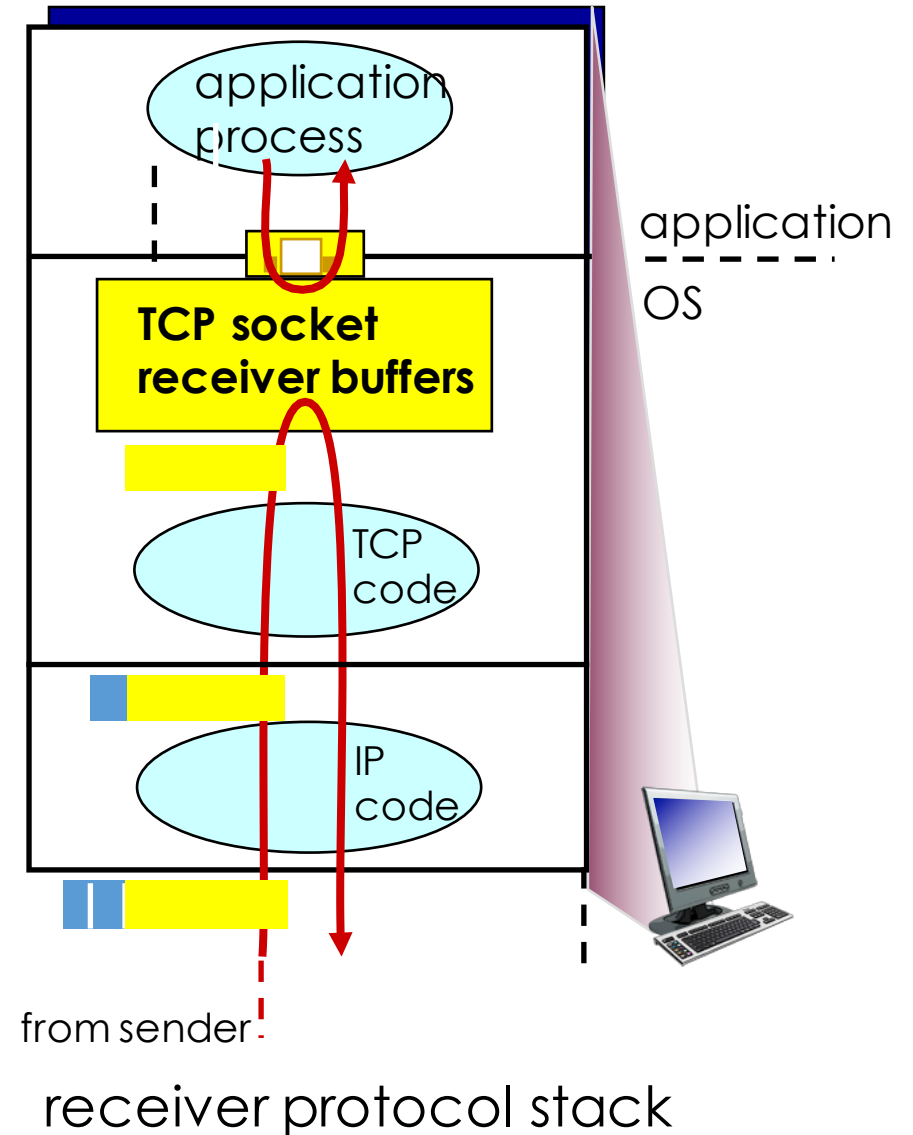
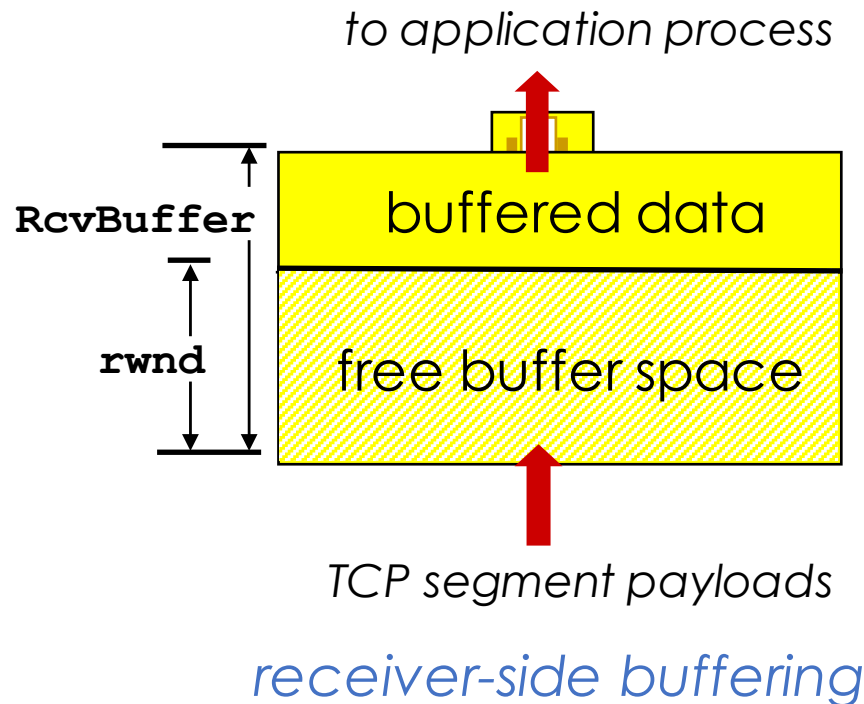
Outline

- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - Connection management
 - Segment structure
 - Reliable data transfer
 - **Flow control**
 - Congestion Control

TCP Flow Control

flow control

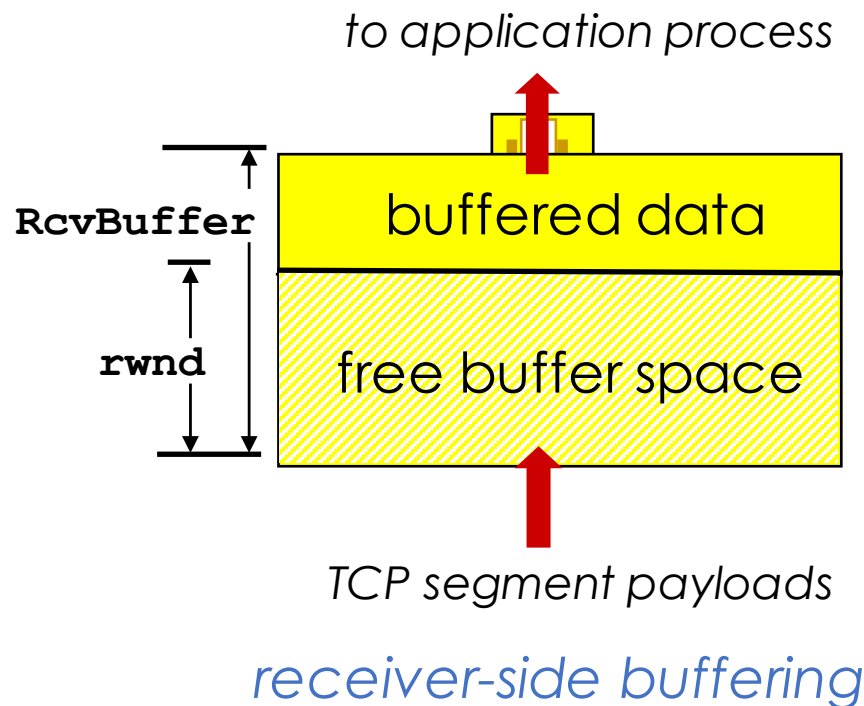
receiver controls sender, so
sender won't overflow
receiver's buffer by
transmitting too much, too fast



TCP Flow Control

- Sender limits unacked (“in-flight”) data

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



How about UDP?

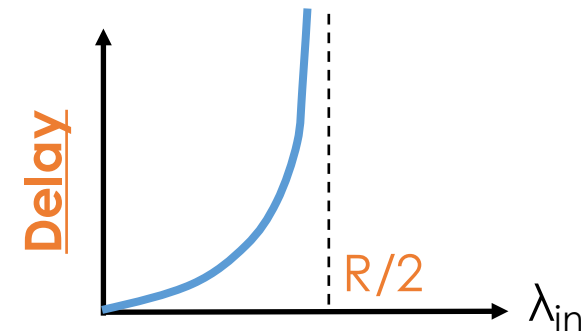
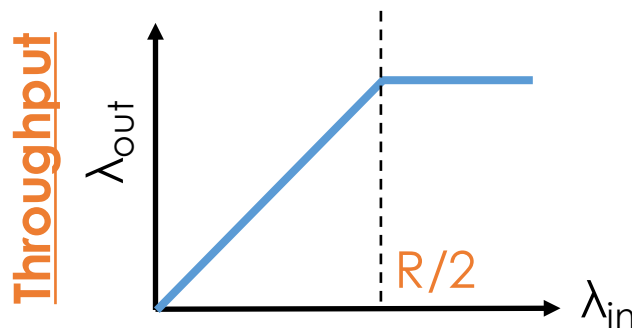
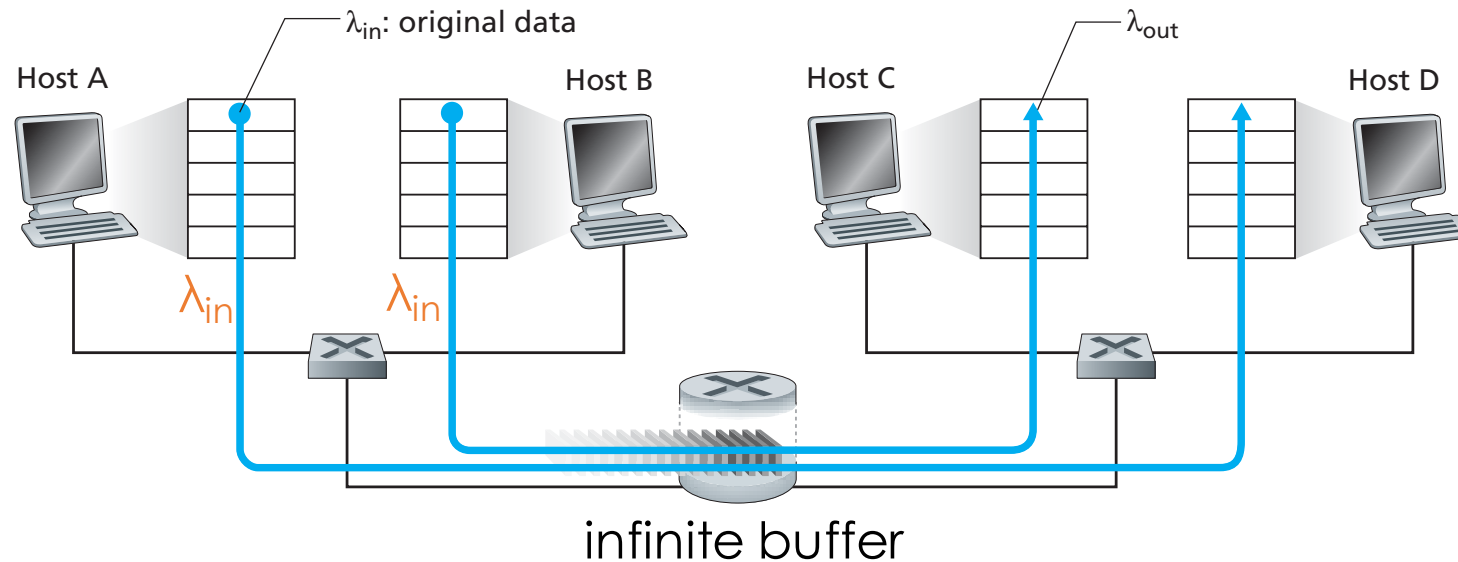
Does not support flow control
→ Segments may be received but dropped by Rx due to buffer overflow

Outline

- Transport-layer services
- Multiplexing and demultiplexing
 - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
 - Connection management
 - Segment structure
 - Reliable data transfer
 - Flow control
 - **Congestion Control**

Causes of Congestion (Case 1)

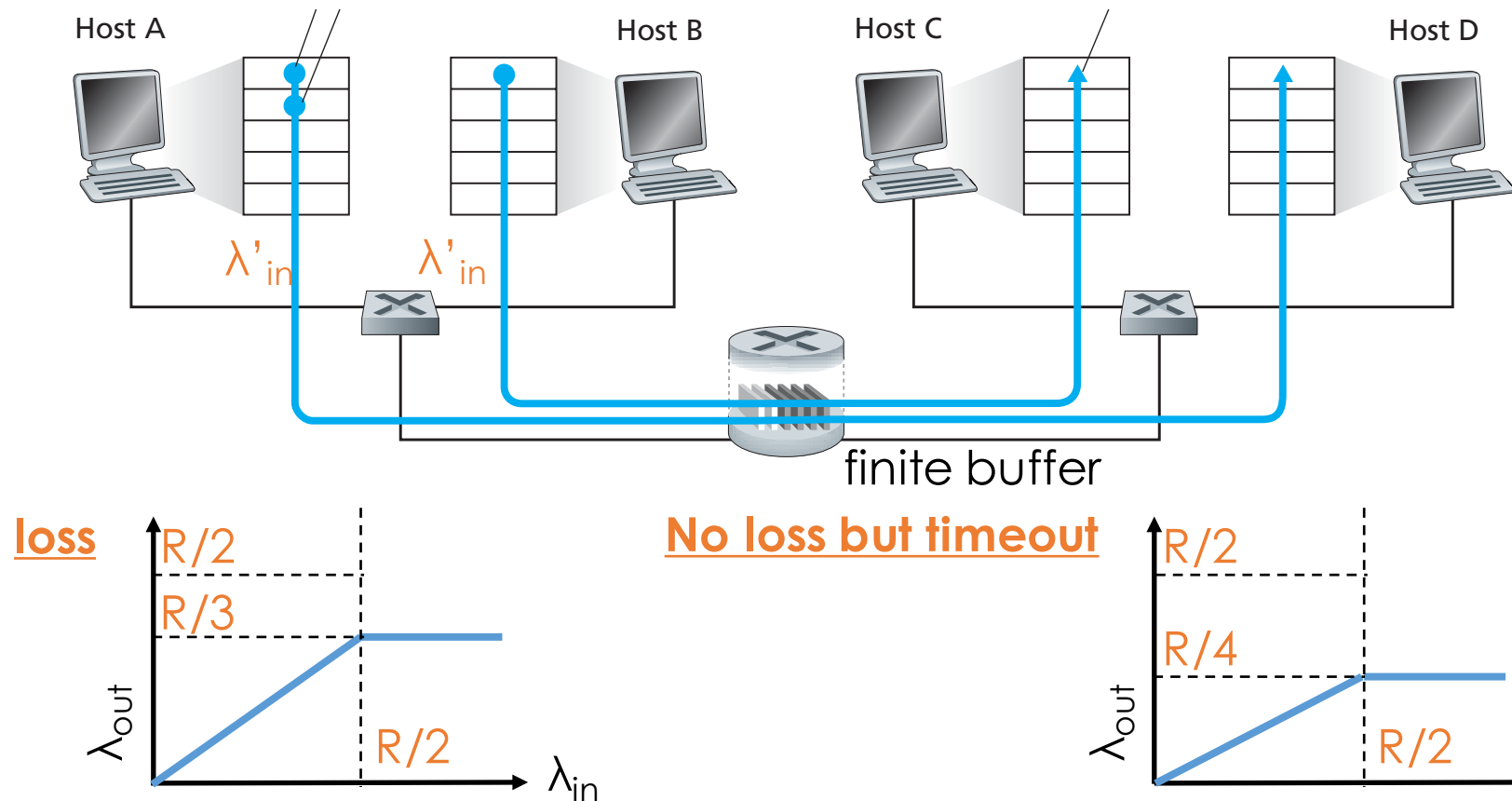
2 connections share a reliable link with infinite buffer



- When λ_{in} exceeds $R/2$, the average number of queued packets in the router is unbounded \rightarrow delay becomes infinite

Causes of Congestion (Case 2)

2 connections with finite buffer and retransmission enabled



- With retransmission, **offered load** becomes λ'_{in} larger than λ_{in}
- Capacity wastes: 1) **packet loss: retransmission**, 2) **timeout: unnecessary retransmissions**

TCP Congestion Control

- End-to-end control, rather than network-assisted control
- Idea: TCP sender determines the rate
 - No congestion → increase the rate
 - Congestion → reduce the rate
- Questions:
 - How to limit the rate?
 - How to determine whether there is congestion?
 - How to change the rate?

TCP Congestion Control

- How to limit the rate?
 - track a variable, **congestion window**, called **cwnd**

$$\#unACKed = LastByteSent - LastByteAcked \leq \min(rwnd, cwnd)$$
$$rate \approx cwnd / RTT$$

- How to determine congestion?
 - Buffer overflow → losses
 - How to detect? 1) **timeout**, or 2) **3 dup-ACK**
 - How to change the rate?
 - Arrival of ACK → "nothing wrong"
 - Missing ACK → congestion
 - Use **ACKs to update cwnd** → **self clocking**
- Q: how to adjust the value of cwnd?**

Bandwidth Probing

Key idea of TCP's congestion control

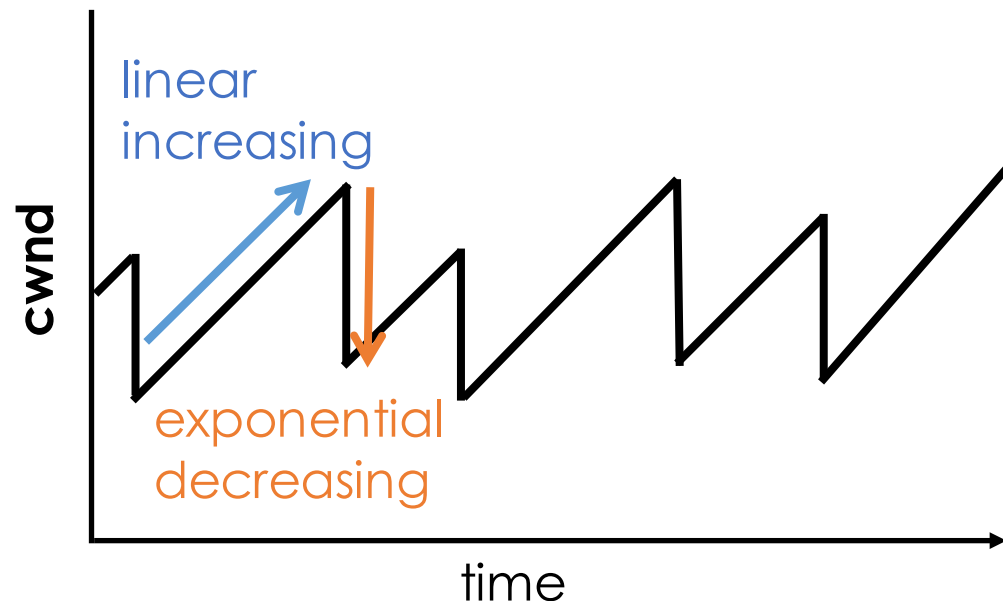
- Get ACK → **cwnd**↑ → rate ↑
- Packet losses → timeout or dup ACK → **cwnd**↓

TCP's congestion control algorithm [RFC 5681]

- Slow start
- Congestion avoidance
- Fast recovery

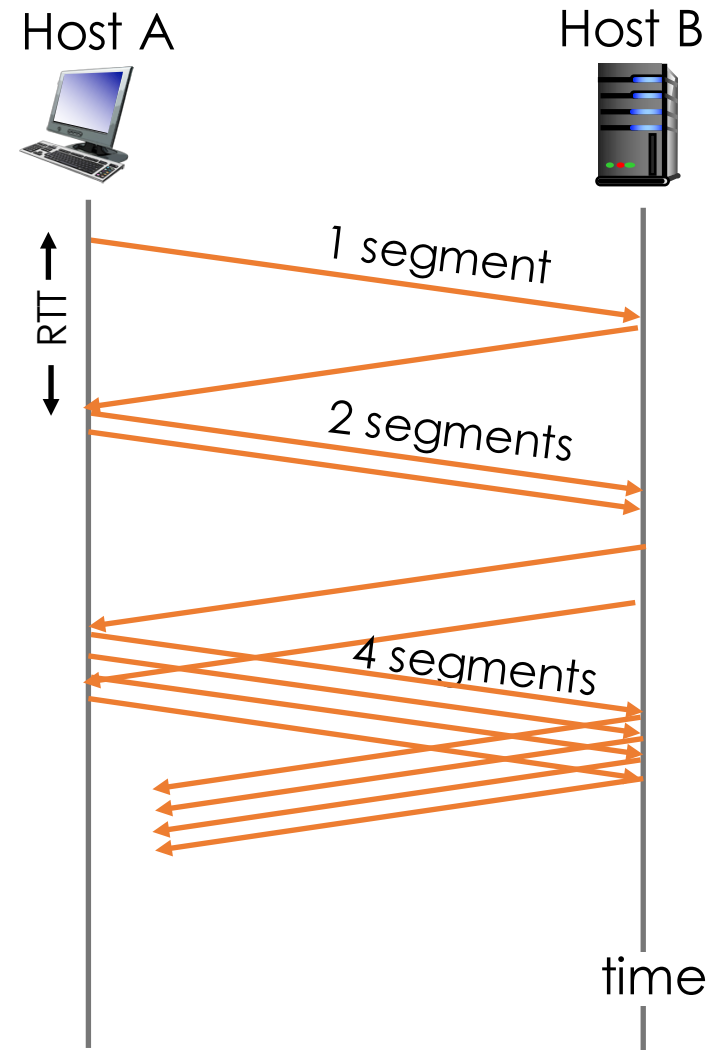
TCP Congestion Control

- Sender **increases** the rate (**cwnd**), probing for usable bandwidth, **until loss occurs**
- **Additive Increase Multiplicative Decrease (AIMD)**
 - additive increase: $\text{cwnd} = \text{cwnd} + 1 * \text{MSS}$ every RTT until loss detected
 - multiplicative decrease: $\text{cwnd} = 0.5 * \text{cwnd}$ after loss



TCP Slow Start

- When connection begins, increase rate **exponentially** until first loss event:
 1. initially **cwnd** = 1 MSS
 2. **cwnd** = 2 * **cwnd**
 3. Update for every RTT (i.e., ACK received)
- Summary: initial rate is slow but ramps up exponentially fast
→ slow start is not slow @@



How to Detect Loss?

- Loss indicated by **timeout**:

Timeout \rightarrow **cwnd** = 1 (MSS)

- begin the slow start process anew

- Loss indicated by 3 duplicate ACKs: **TCP RENO**

- enter the **fast recovery** state

Dup ACK \rightarrow **cwnd** = **cwnd**/2 + 3
 \rightarrow increase **cwnd** linearly

- “add 3” is to account for the three dup ACK
- more conservative as compared to timeout

- **TCP Tahoe** always sets **cwnd** to 1, no matter timeout or 3 duplicate ACKs

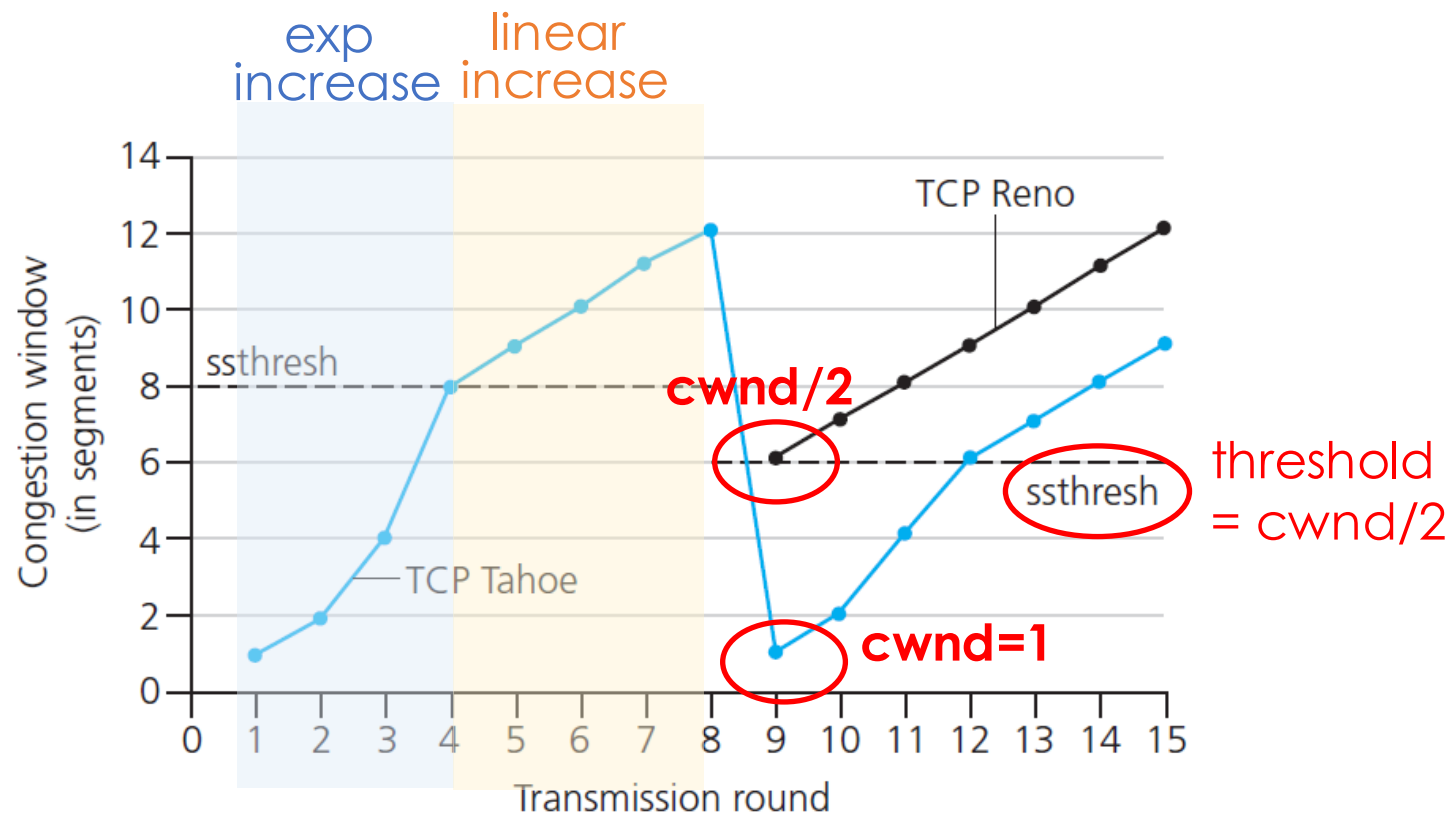
Congestion Avoidance

- switch to the congestion-avoidance mode
 - When $cwnd = ssthresh$
 - Increase **cwnd** linearly
 - $cwnd = cwnd + 1$
- When and how to update the value of **ssthresh**?
 - When: when a loss event occurs
 - How: $ssthresh = cwnd/2$
- Why congestion avoidance?
 - Prevent from saturating the available bandwidth

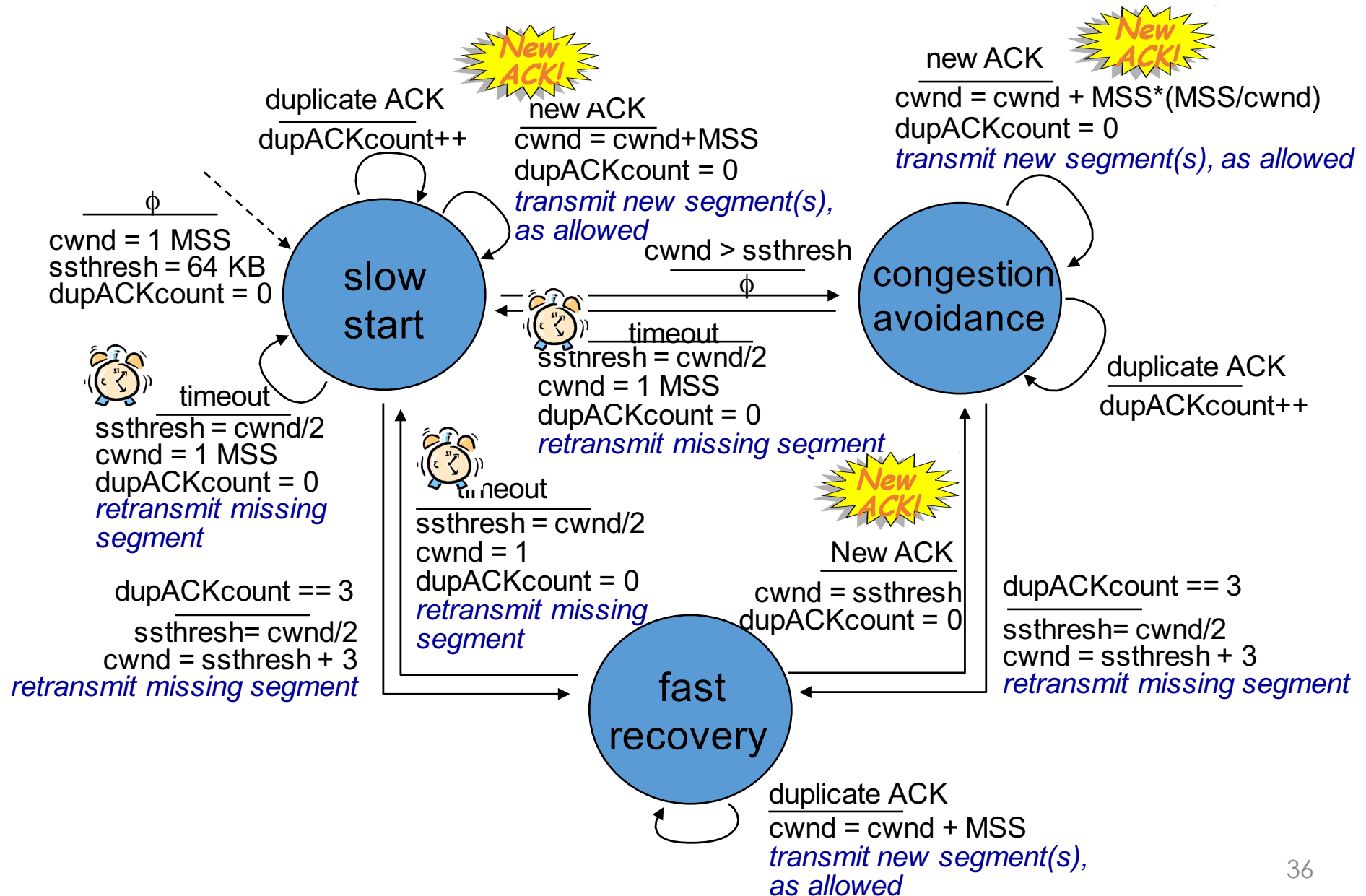
Congestion Window Adaptation

Q: when should exponential increase switch to linear?

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



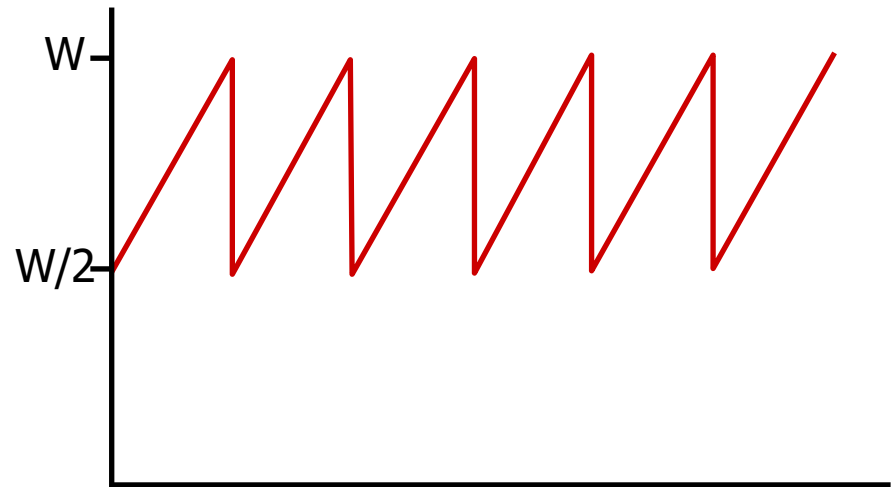
TCP Congestion Control



TCP Average Throughput

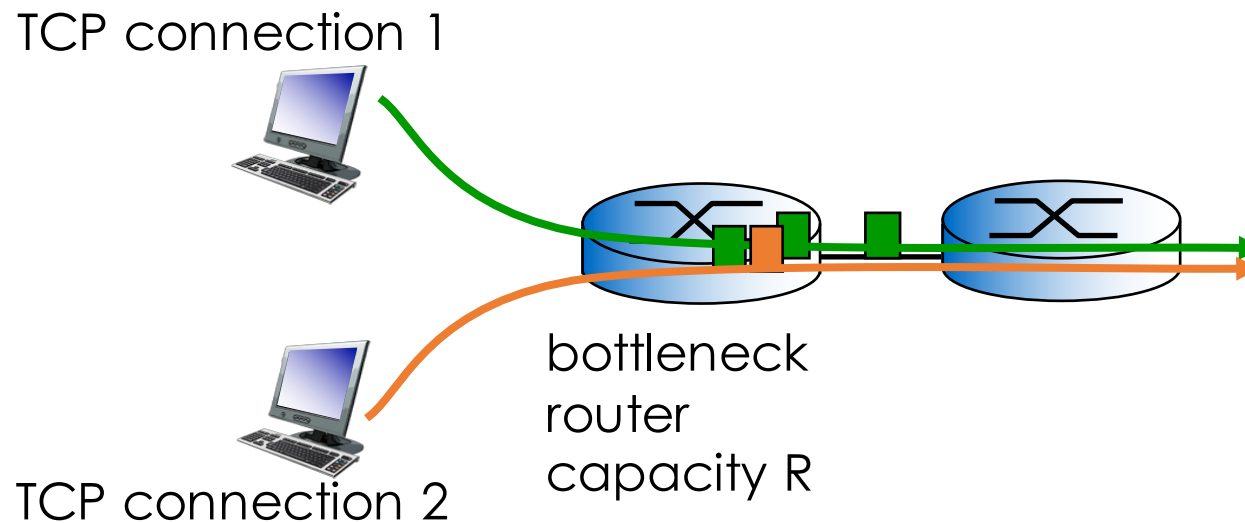
- avg. TCP throughput as function of window size, RTT?
 - ignore slow start, assume always data to send
- W: window size where loss occurs
 - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
 - avg. throughput is $\frac{3}{4}W$ per RTT

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



TCP Fairness

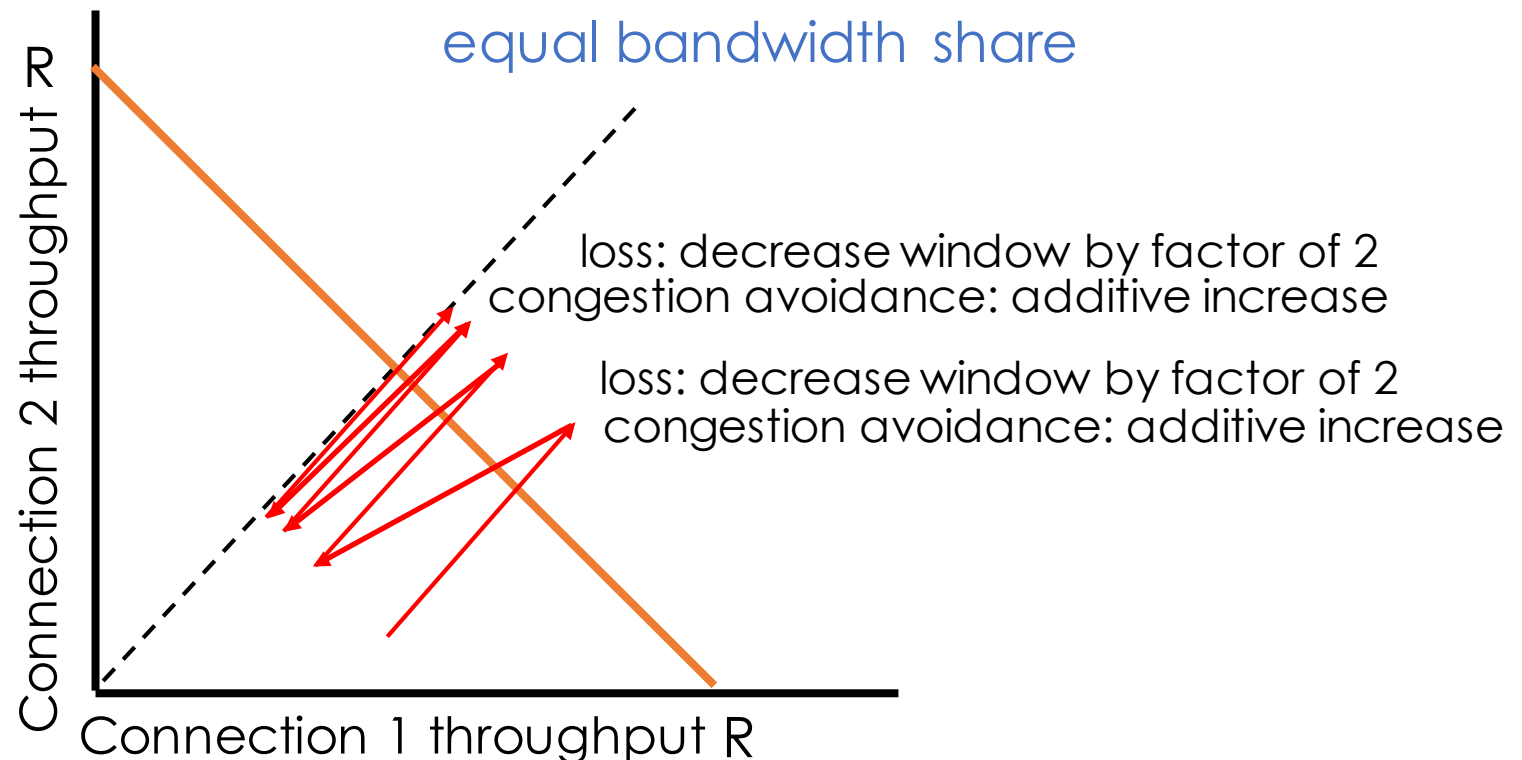
- *fairness goal*: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Why is TCP Fair?

Simple example: two competing sessions

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



Quiz

- Retransmissions can be trigger by which (two) events?