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

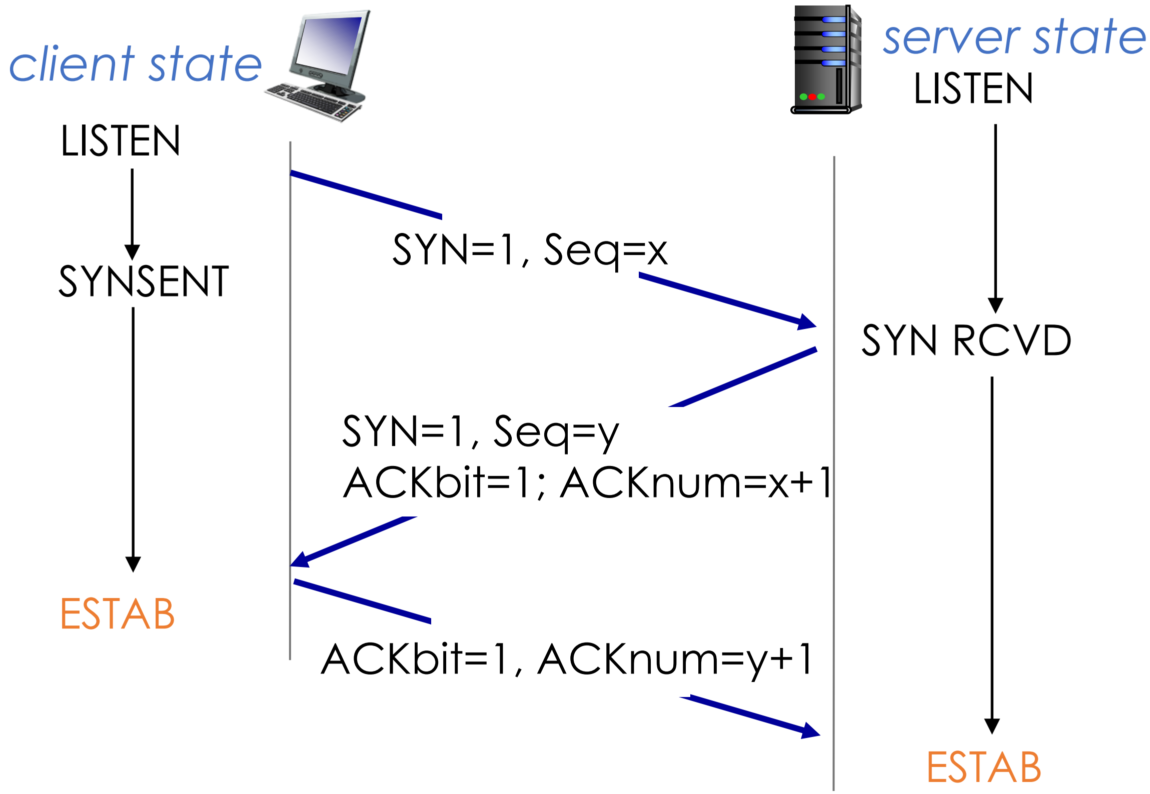
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- **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 steam:***
  - no "message boundaries"
  - **Pipelined**
  - TCP congestion and flow control set window size

Outline

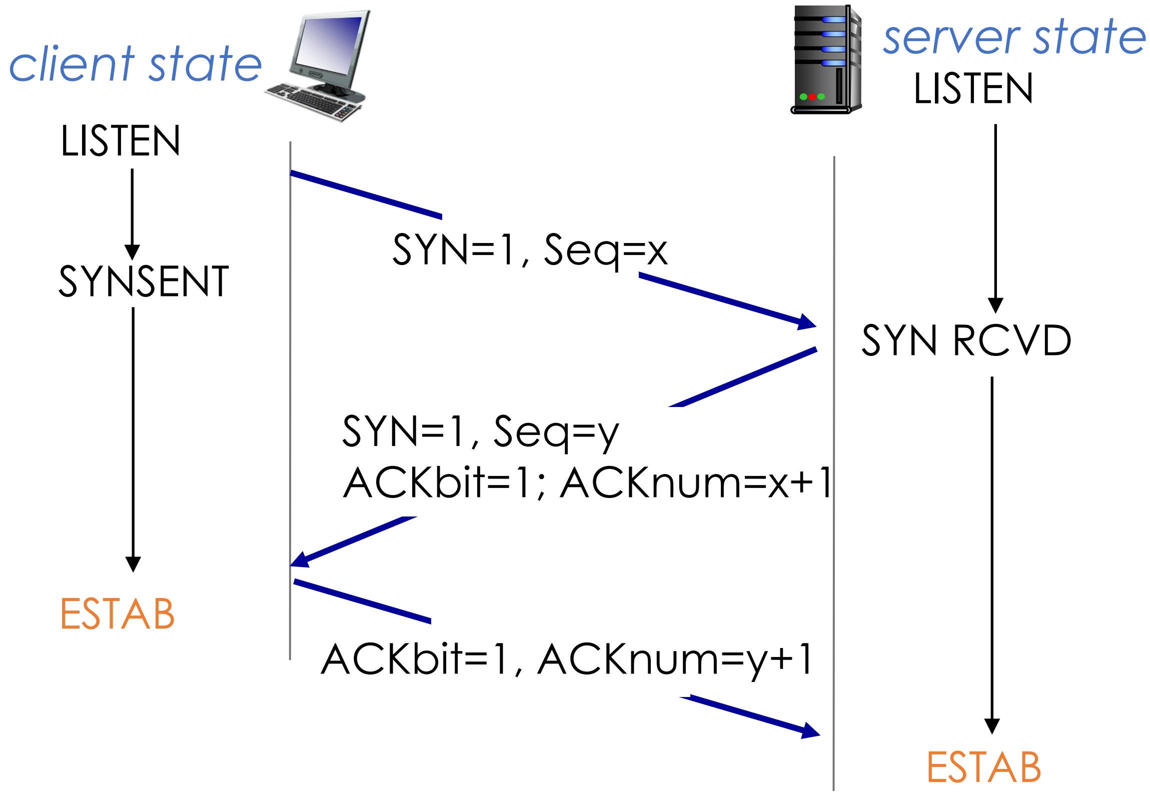
- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
  - **Connection management**
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Congestion Control

Three-Way Handshaking



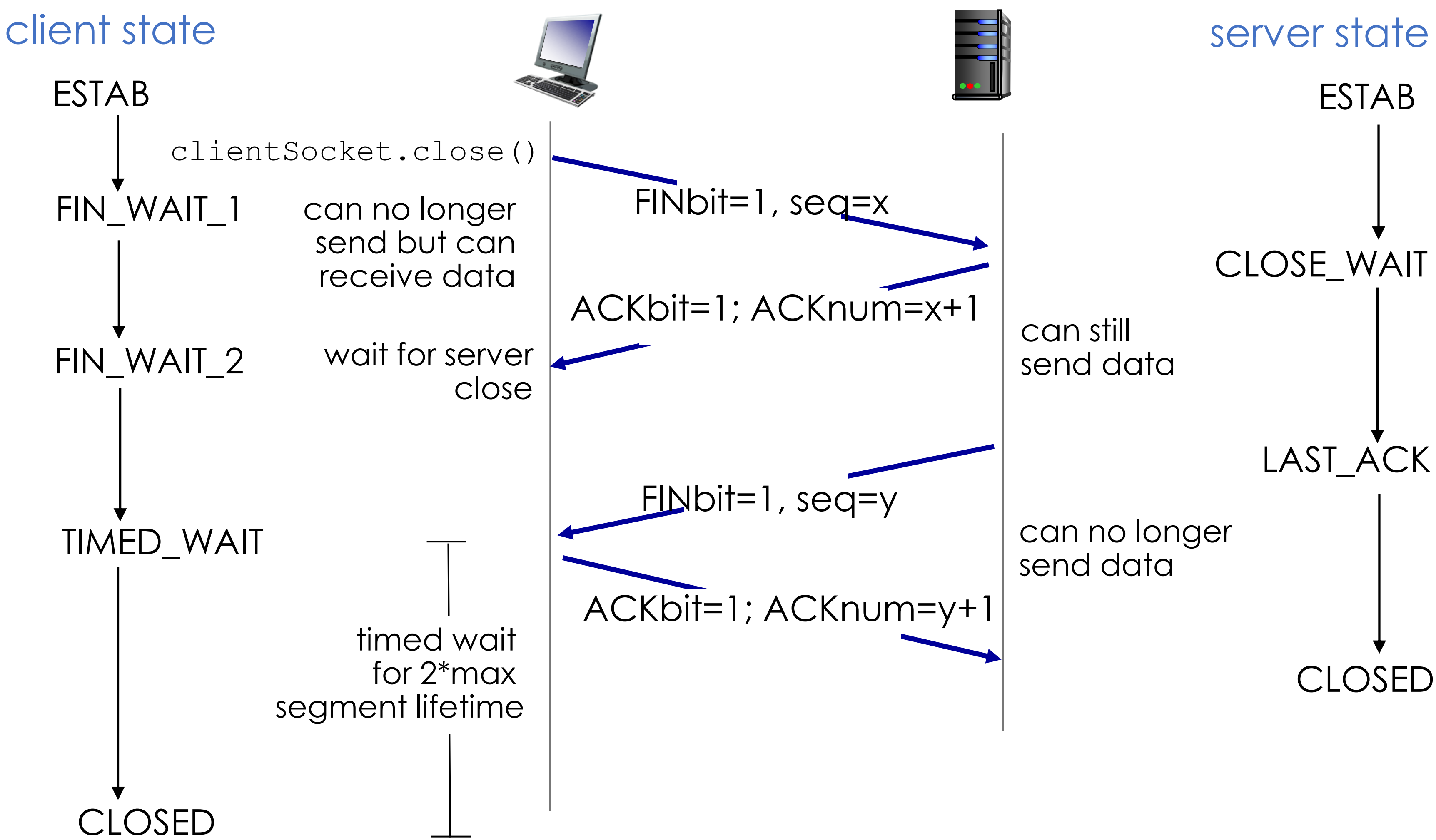
- msg1: SYN = 1, Seq x (from C) randomly selected
- msg2: SYN = 1, Seq y (from S) randomly selected
  ACK# = x+1 (ready to recv next one)
- msg3: ACK# = y + 1 (ready to receive next one)

Three-Way Handshaking



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

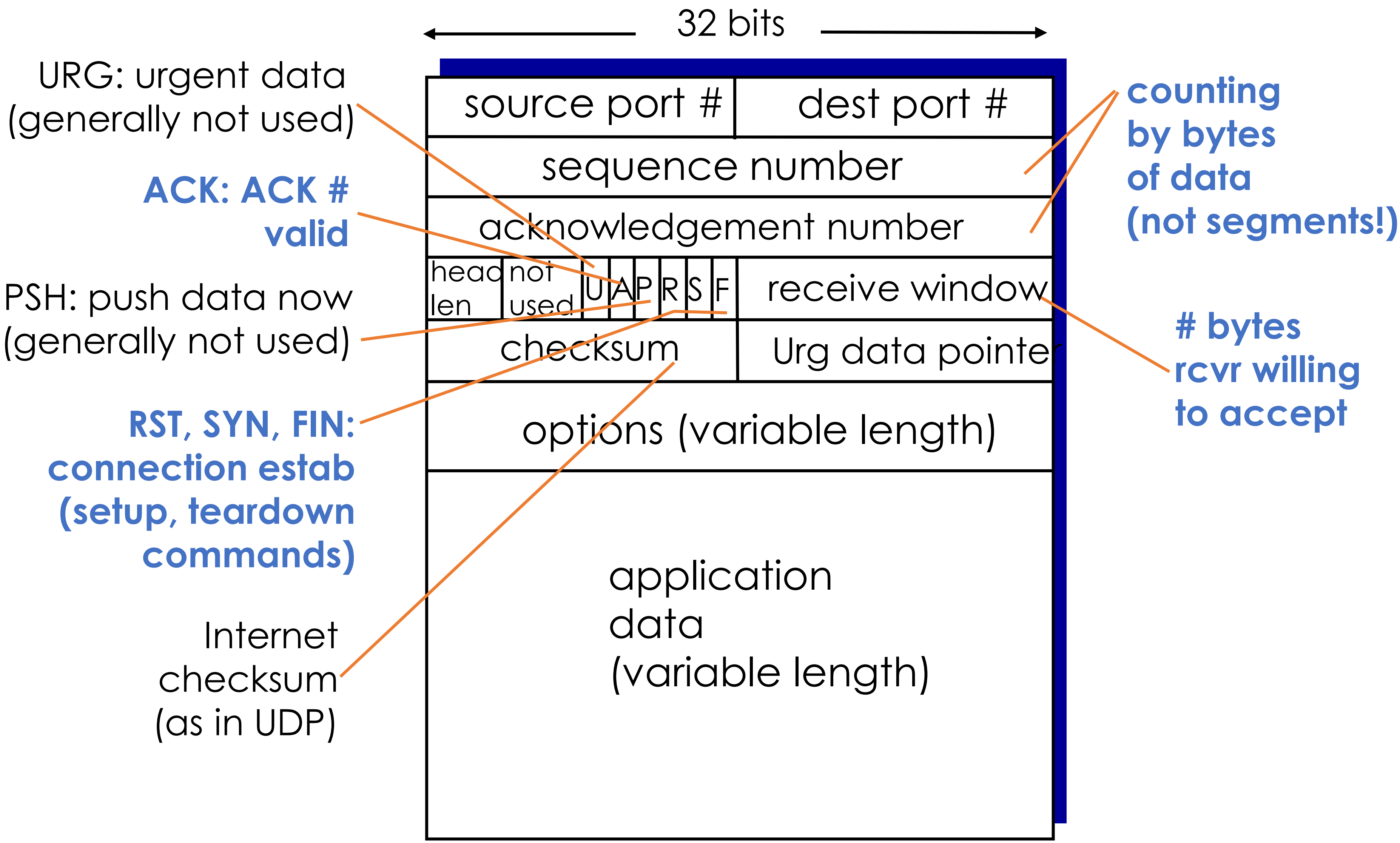
Connection Termination



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

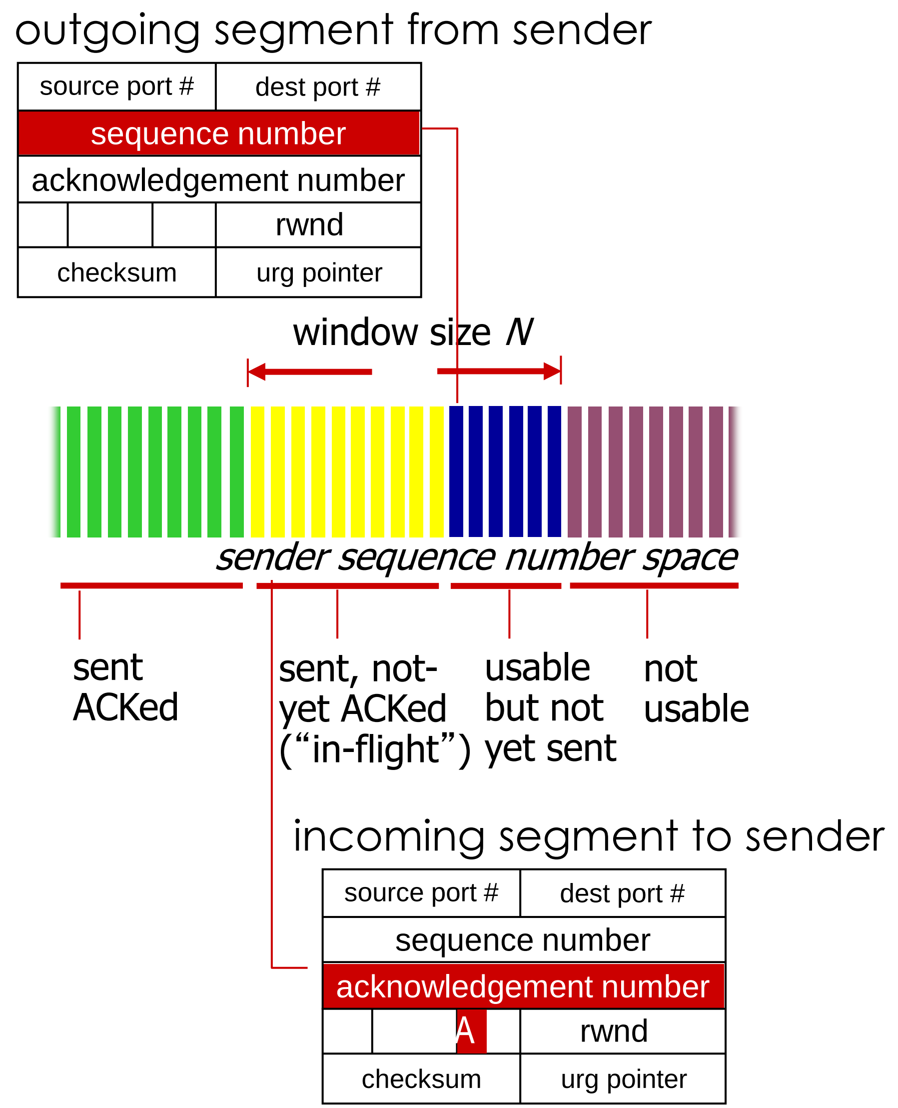


TCP Seq. Numbers, ACKs

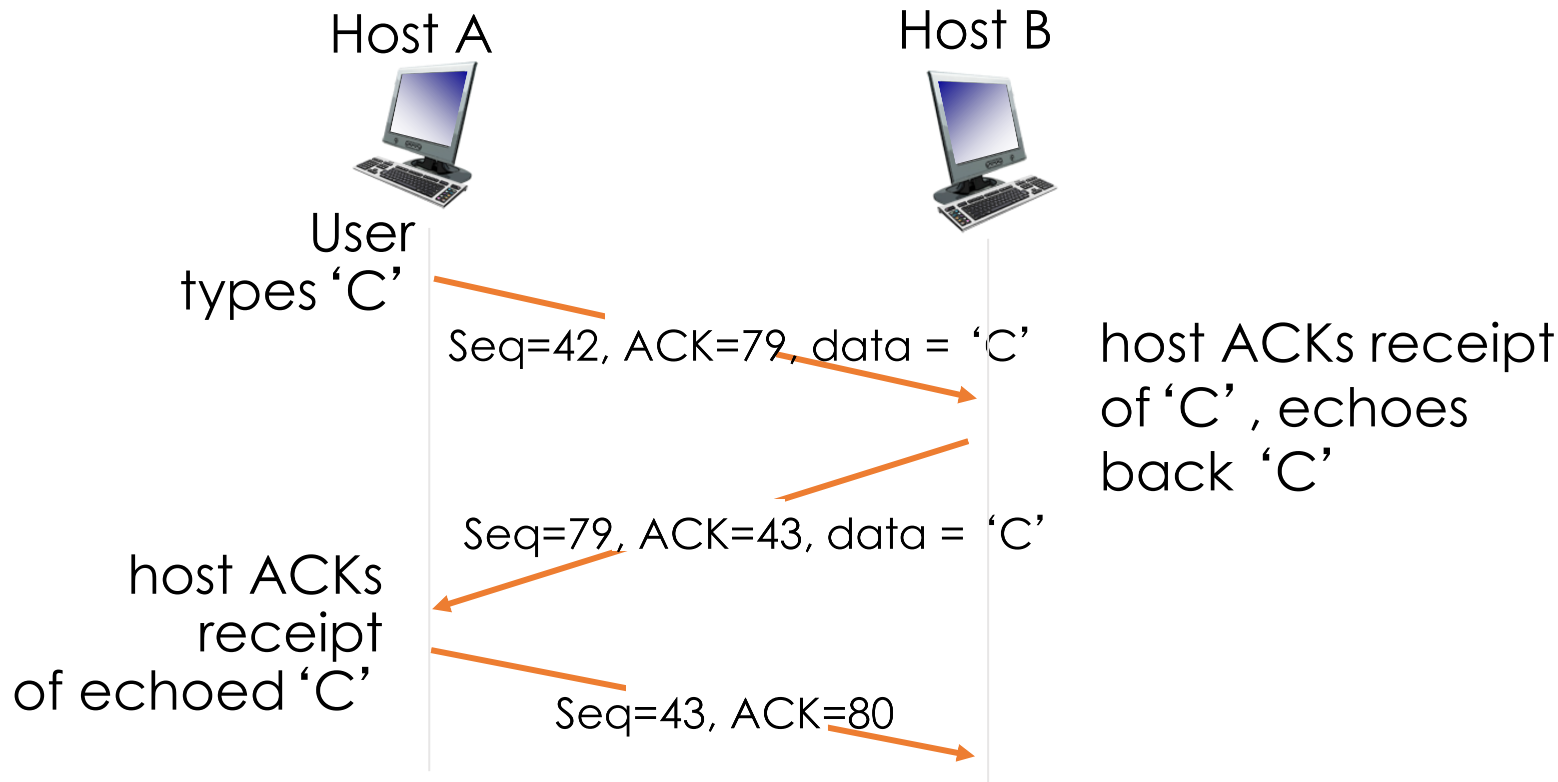
- **Sequence numbers:**
  - byte stream "number" of first byte in segment's data
- **Acknowledgements:**
  - seq # of next byte expected from other side
  - cumulative ACK

Q: how receiver handles out-of-order segments

- **A:** TCP spec doesn't say (up to implementer)



TCP Seq. Numbers, ACKs



simple telnet scenario

ACK Example

Outline

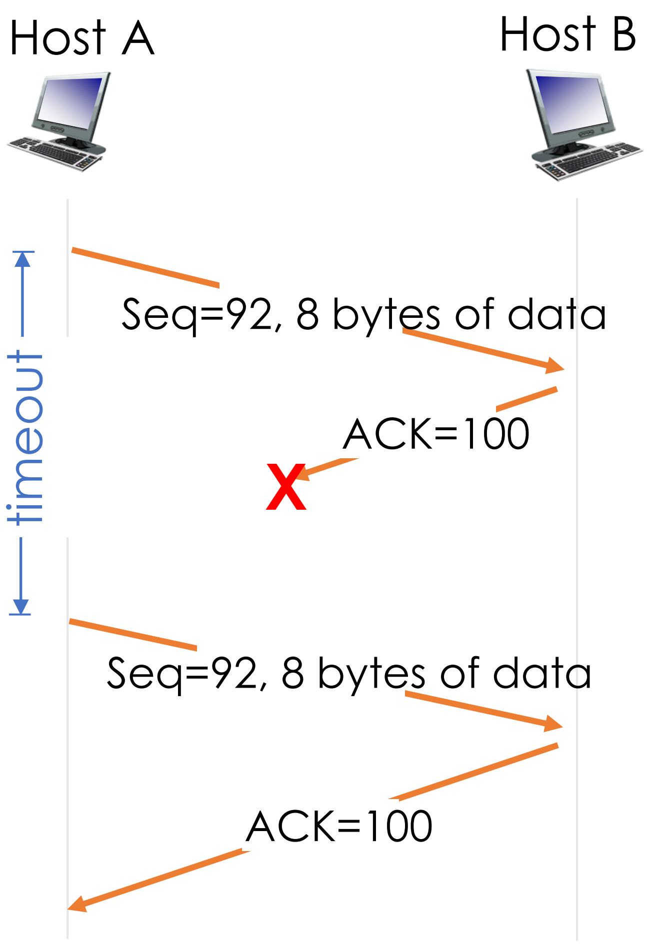
- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
  - Connection management
  - Segment structure
  - **Reliable data transfer**
  - Flow control
  - Congestion Control

TCP Reliable Data Transfer

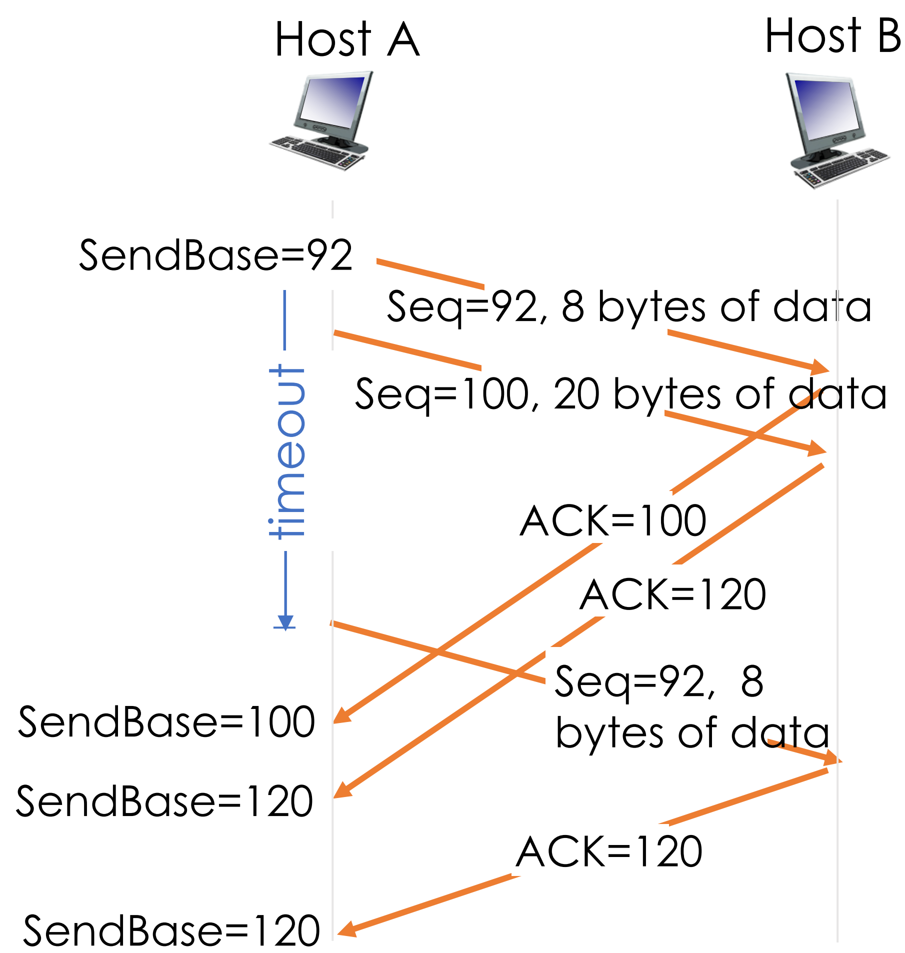
- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative ACKs
  - single retransmission timer

- Retransmissions triggered by
  - timeout events
  - duplicate ACKs (fast retransmission)

TCP Retransmission Scenarios

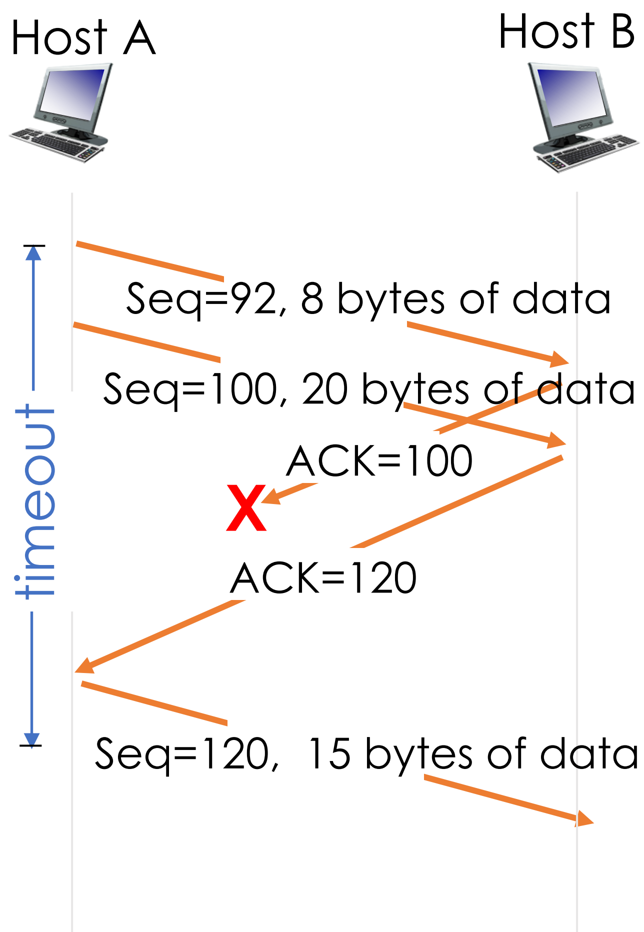


1. lost ACK scenario



2. premature timeout

TCP Retransmission Scenarios



3. cumulative ACK

TCP Fast Retransmit

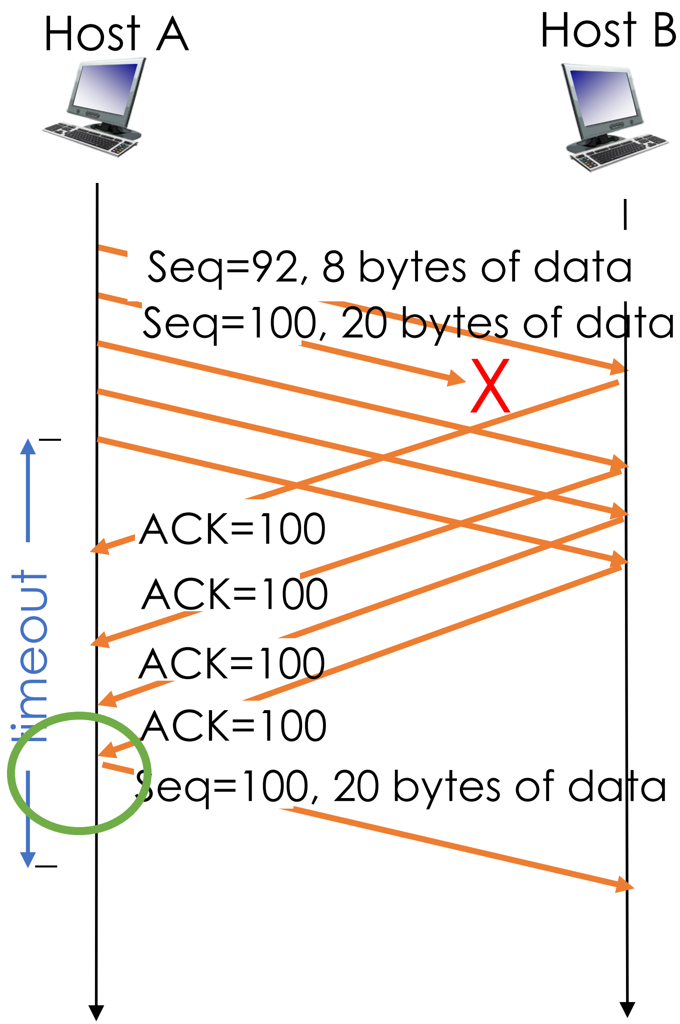
- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs

TCP ***fast retransmit***

if sender receives 3 ACKs for same data
→ 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]

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

Delayed ACK

- As described in [RFC 1122], a host may delay sending an ACK response by up to 500 ms.
- Give the application the opportunity to update the TCP receive window
- Reduce the number of responses
- Issue
  - Long delay if the sender is not continuously sending data

Timeout Configuration

- How to configure a proper timeout that
  - Improve bandwidth utilization (short enough)
  - Avoid unnecessary retransmissions (but not so short)

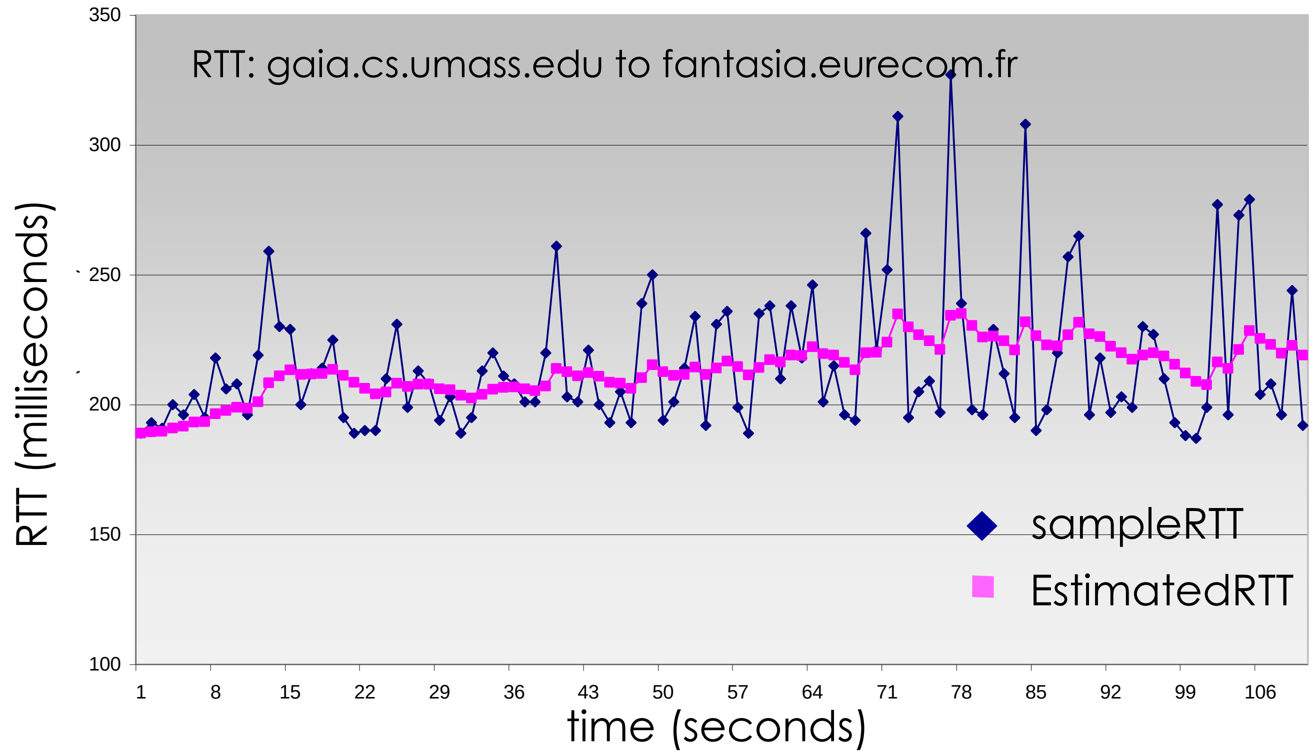
- Key idea: **RTT (round trip time)**
  - Ideally, an ACK should be returned to sender after RTT
  - **Q**: how to measure RTT? What if RTT fluctuates?

Practical RTT Measurement

exponential weighted moving average (EWMA):

RTT = (1-a) * RTT + a * SampleRTT

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

TimeoutInterval = RTT + 4 * DevRTT

estimated RTT "**safety margin**"

RTT deviation:

DevRTT = (1 - b)*DevRTT + b * |SampleRTT - RTT |

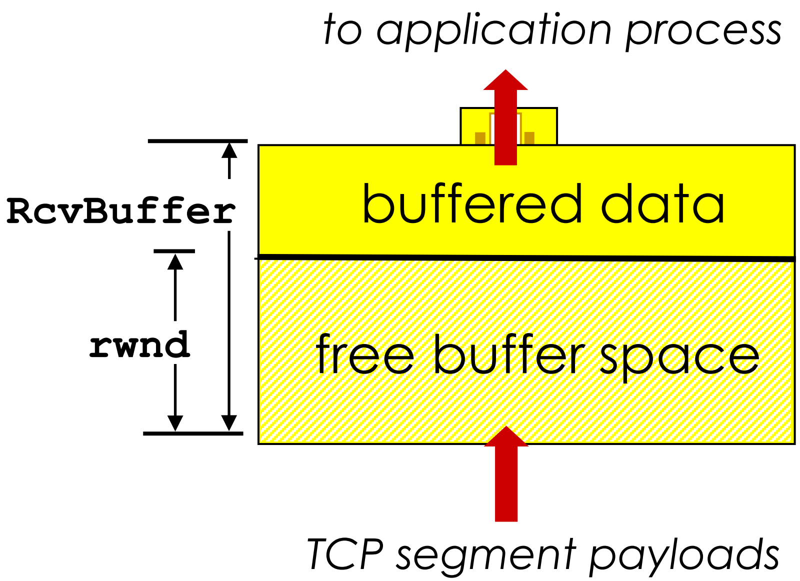
Outline

- 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

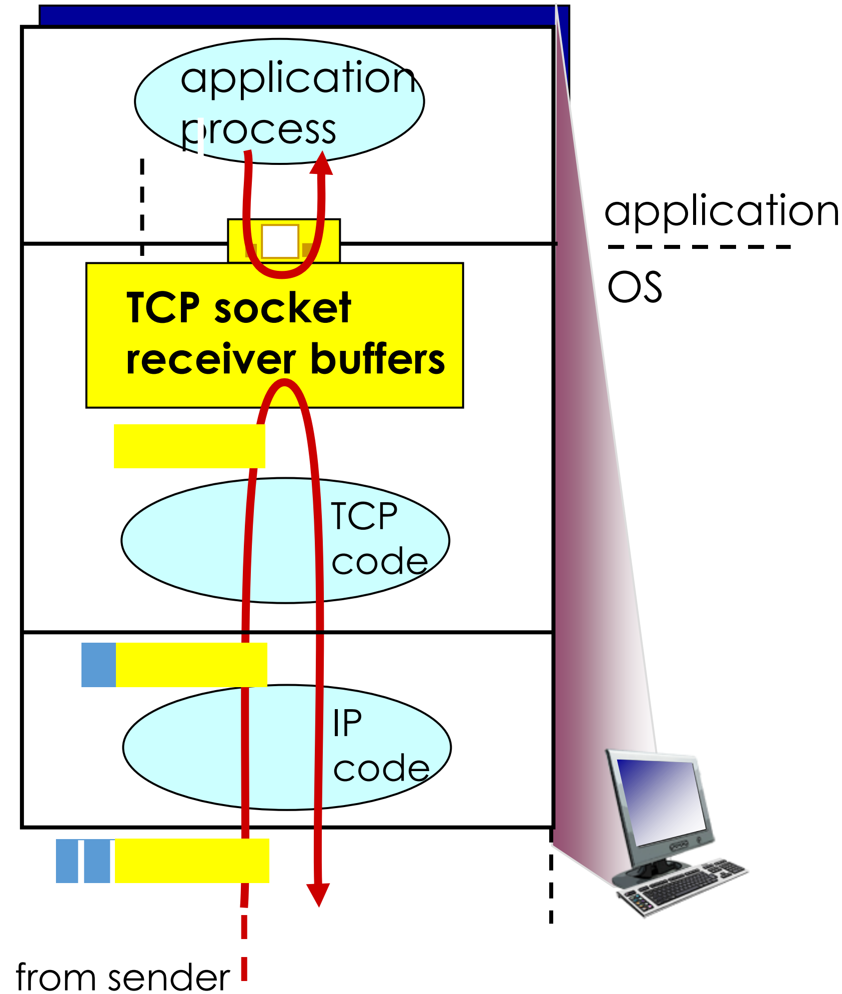
TCP Flow Control

flow control

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



receiver-side buffering

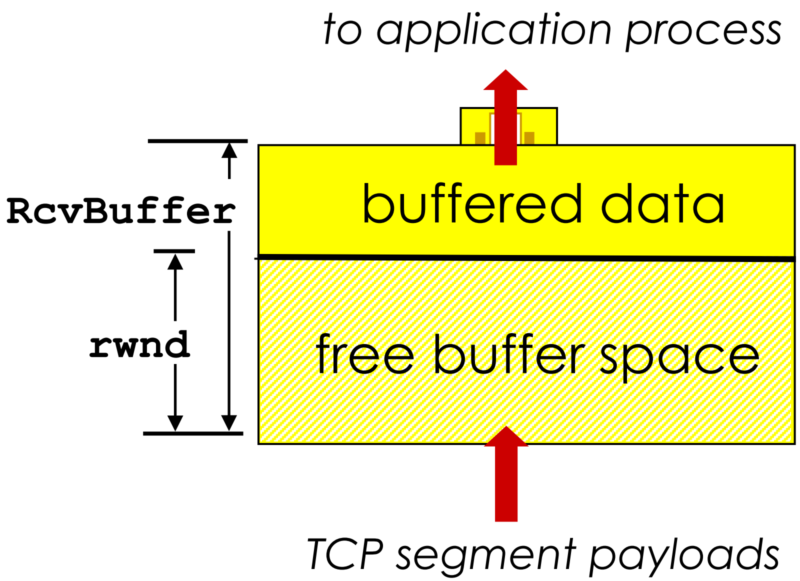


receiver protocol stack

TCP Flow Control

- Sender limits unacked ("in-flight") data

`LastByteSent – LastByteAcked ≤ rwnd`



receiver-side buffering

How about UDP?

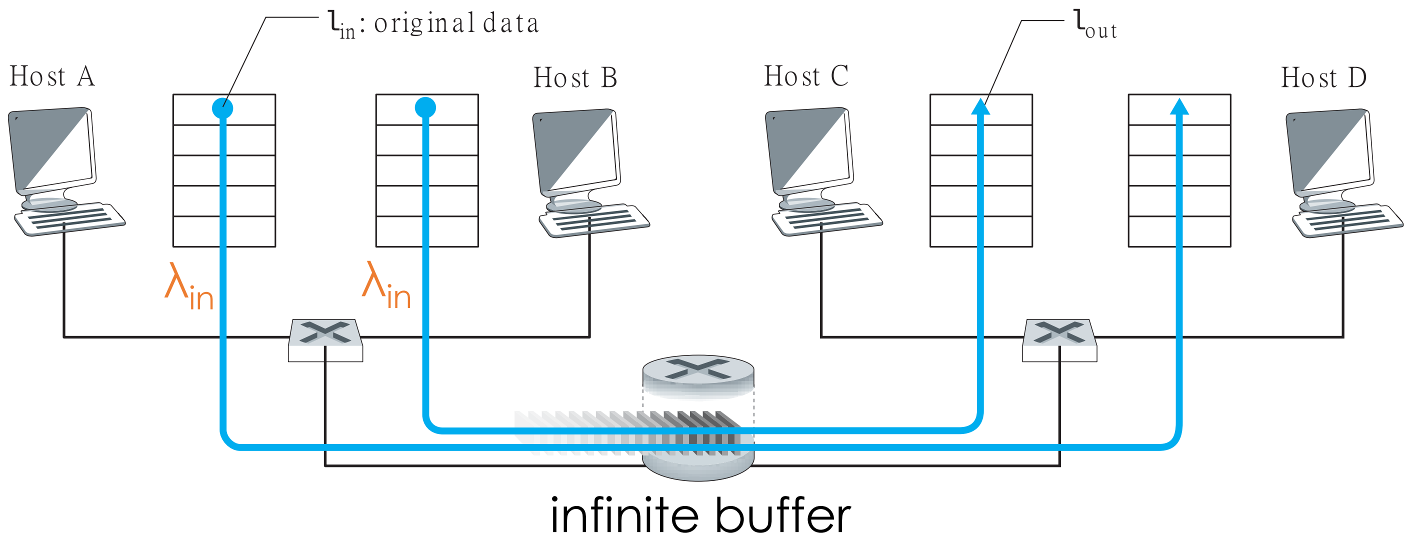
Does not support flow control
→ Segments may be received but dropped by Rx due to buffer overflow

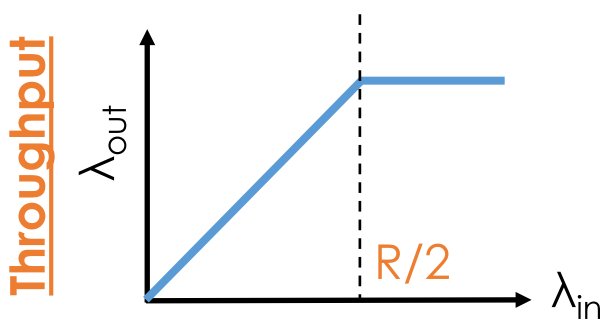
Outline

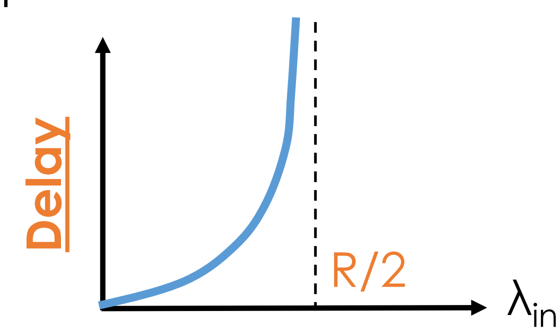
- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- **Connection-oriented transport: TCP**
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
  - **Congestion Control**

Causes of Congestion (Case 1)

2 connections share a reliable link with infinite buffer



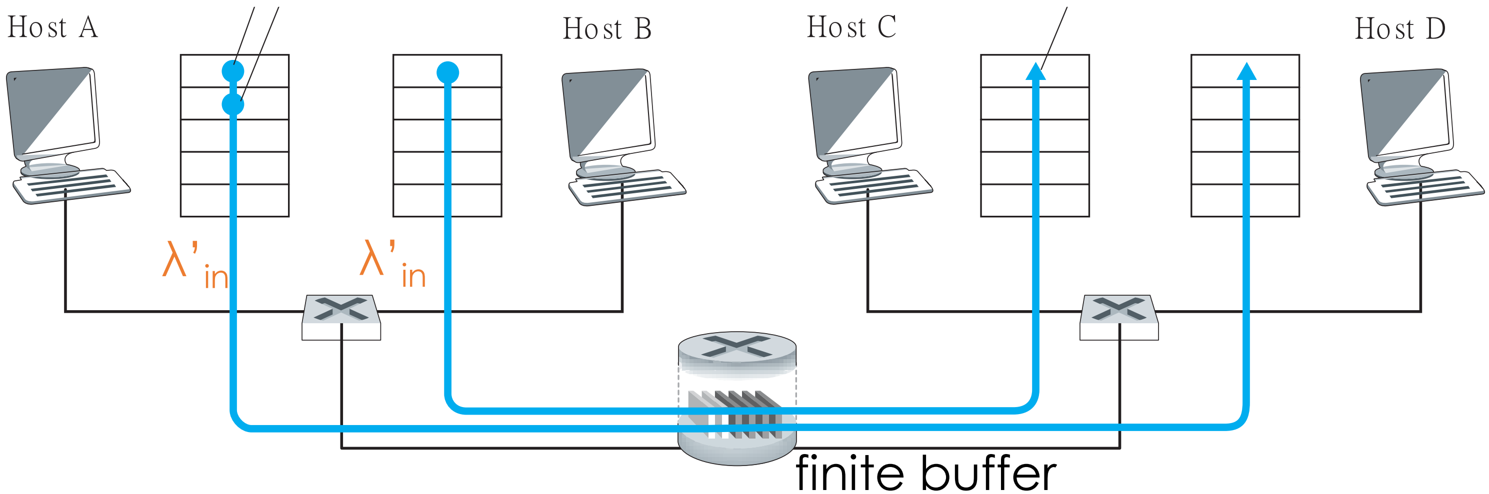




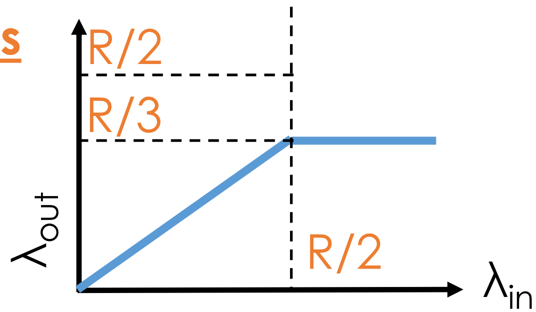
- When λ_{in} exceeds R/2, the average number of queued packets in the router is unbounded → delay becomes infinite

Causes of Congestion (Case 2)

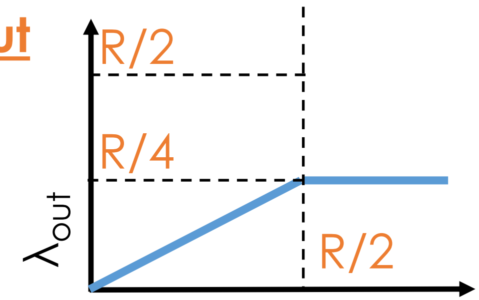
2 connections with **finite buffer** and **retransmission enabled**



loss



No loss but timeout



- With retransmission, **offered load** becomes λ'_{in} larger then 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` ≤ **`min(rwnd,cwnd)`**

rate ≈ **`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

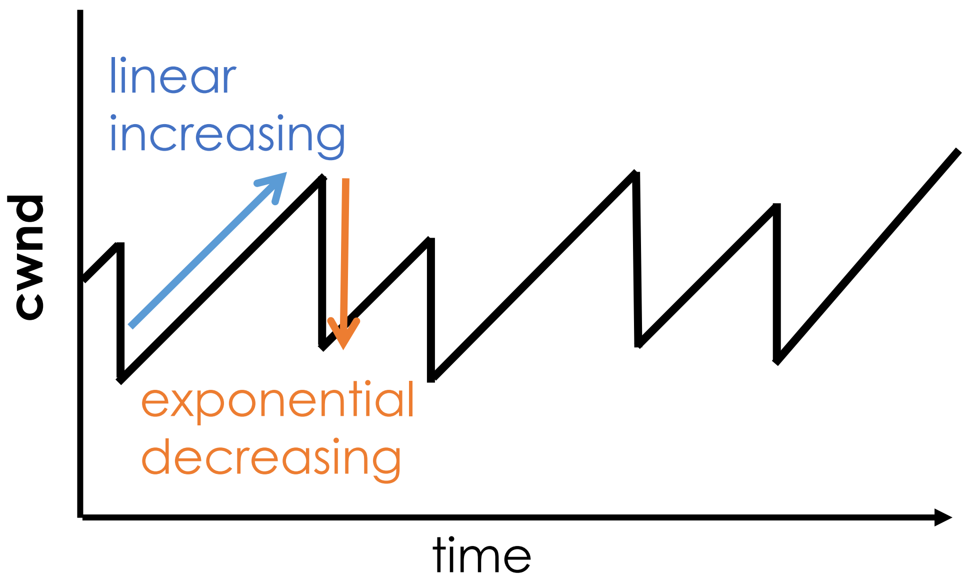
- Get ACK → **`cwnd`**↑ → rate ↑
- Packet losses → timeout or dup ACK → **`cwnd`**↓

TCP's congestion control algorithm [RFC 5681]

- Slow start
- Congestion avoidance
- Fast recovery

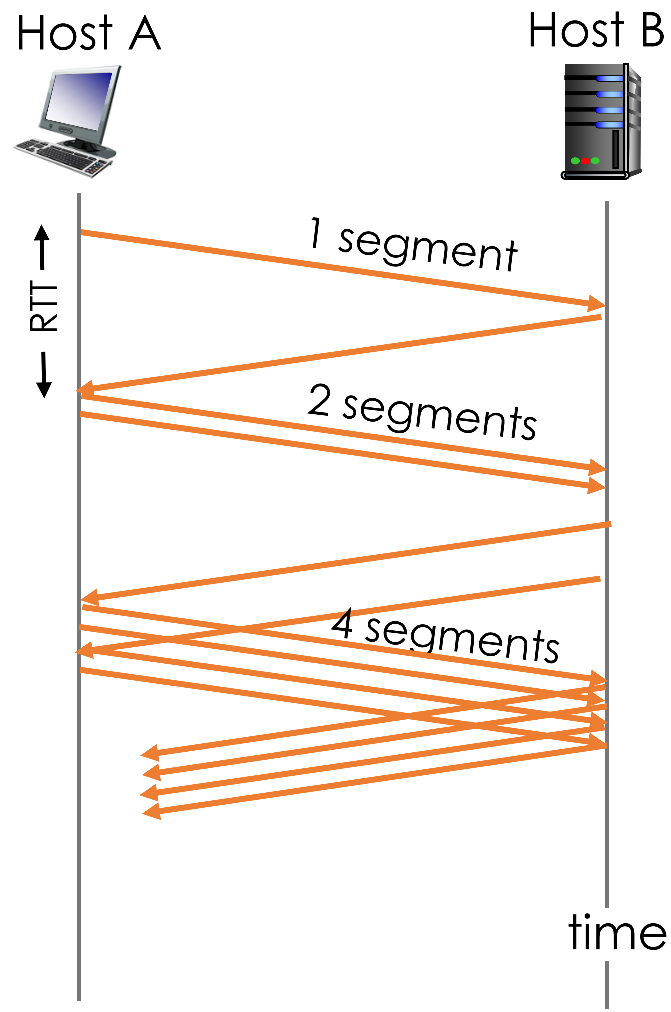
TCP Congestion Control

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

  - begin the slow start process anew

- Loss indicated by 3 duplicate ACKs: **TCP RENO**
  - enter the **fast recovery** state

Dup ACK → **`cwnd = cwnd/2 + 3`** → 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

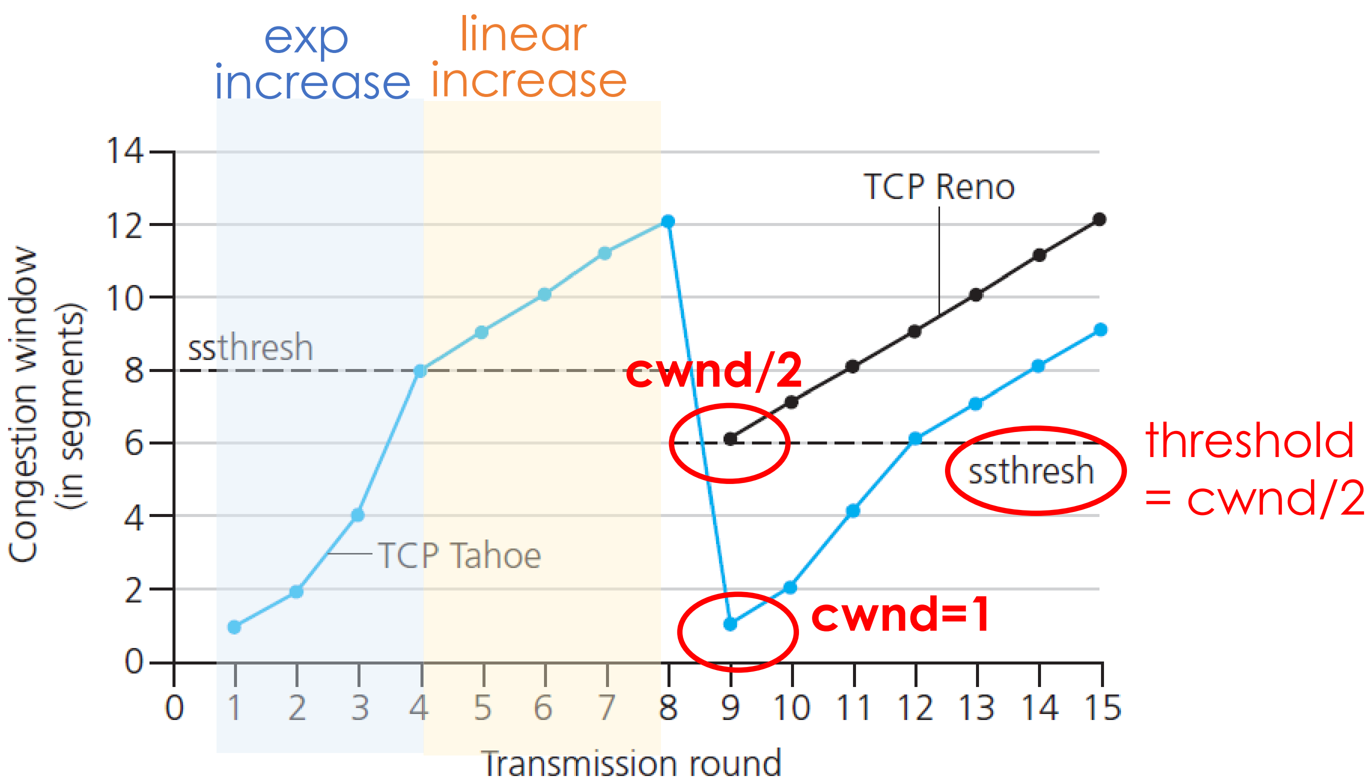
Congestion Avoidance

- 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

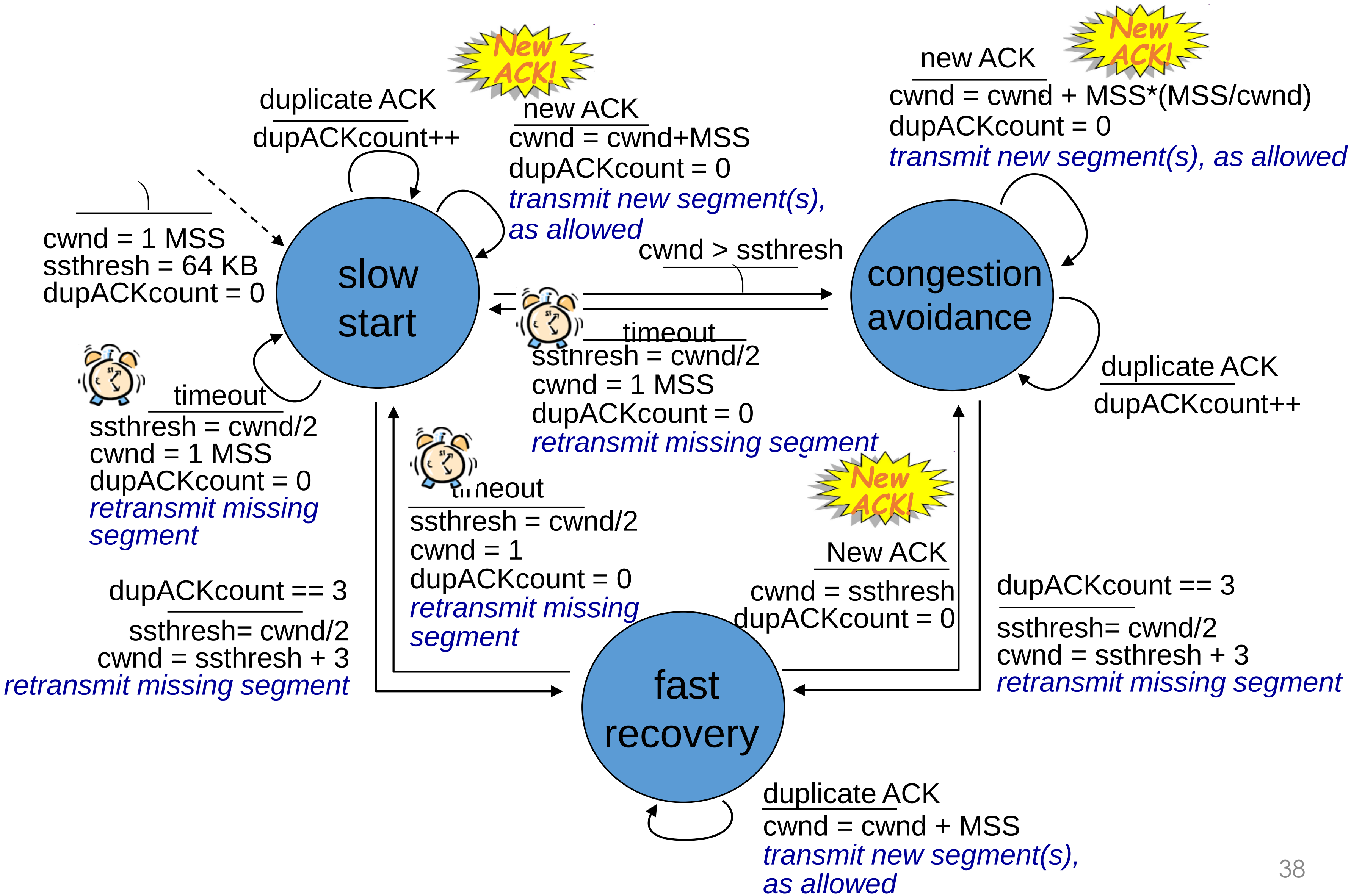
Congestion Window Adaptation

Q: when should exponential increase switch to linear?

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



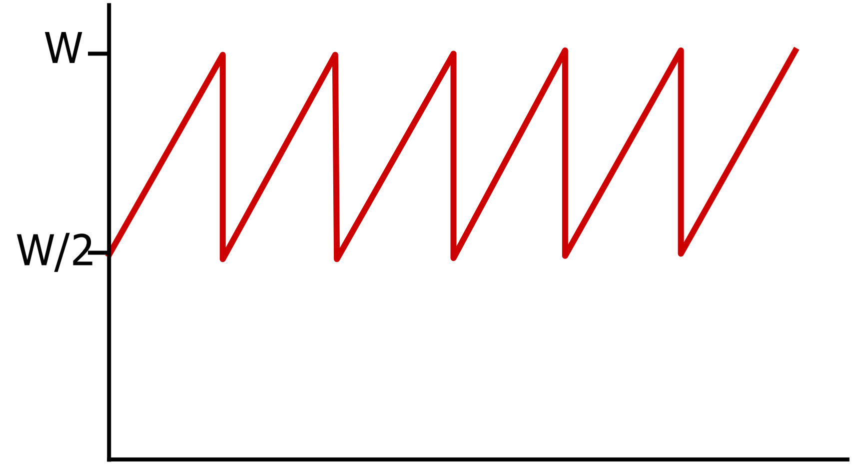
TCP Congestion Control



TCP Average Throughput

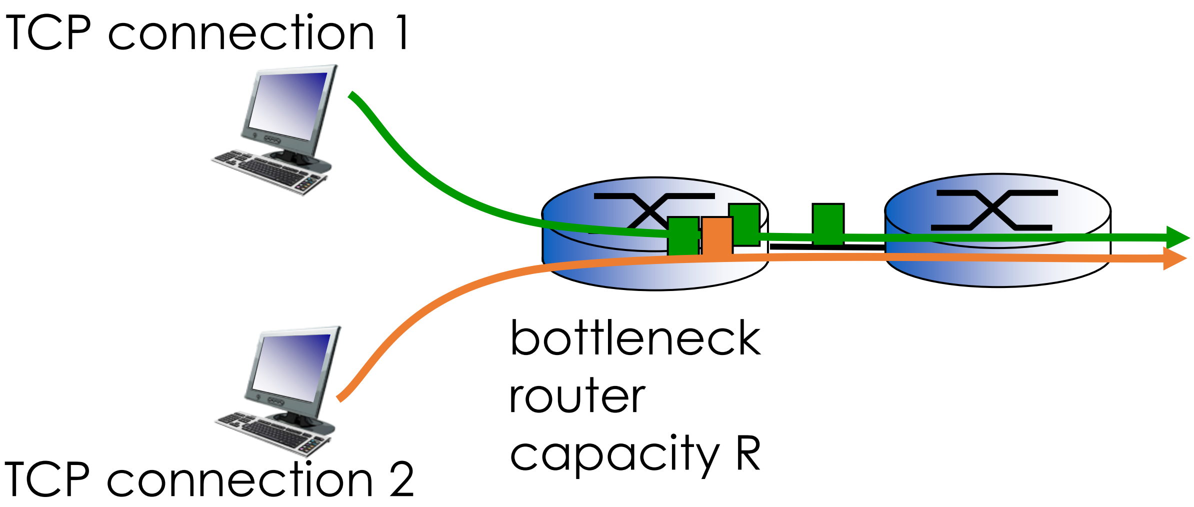
- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT

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



TCP Fairness

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



Why is TCP Fair?

Simple example: two competing sessions

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