Multimedia Communications
@CS.NCTU

Lecture 3: Networking – TCP/UDP

[Computer Networking, Ch3]

Instructor: Kate Ching-Ju Lin (林靖茹)

Slides modified from
"Computer Networking: A Top-Down Approach" 6th Edition

Chapter 3: Transport Layer

our goals:

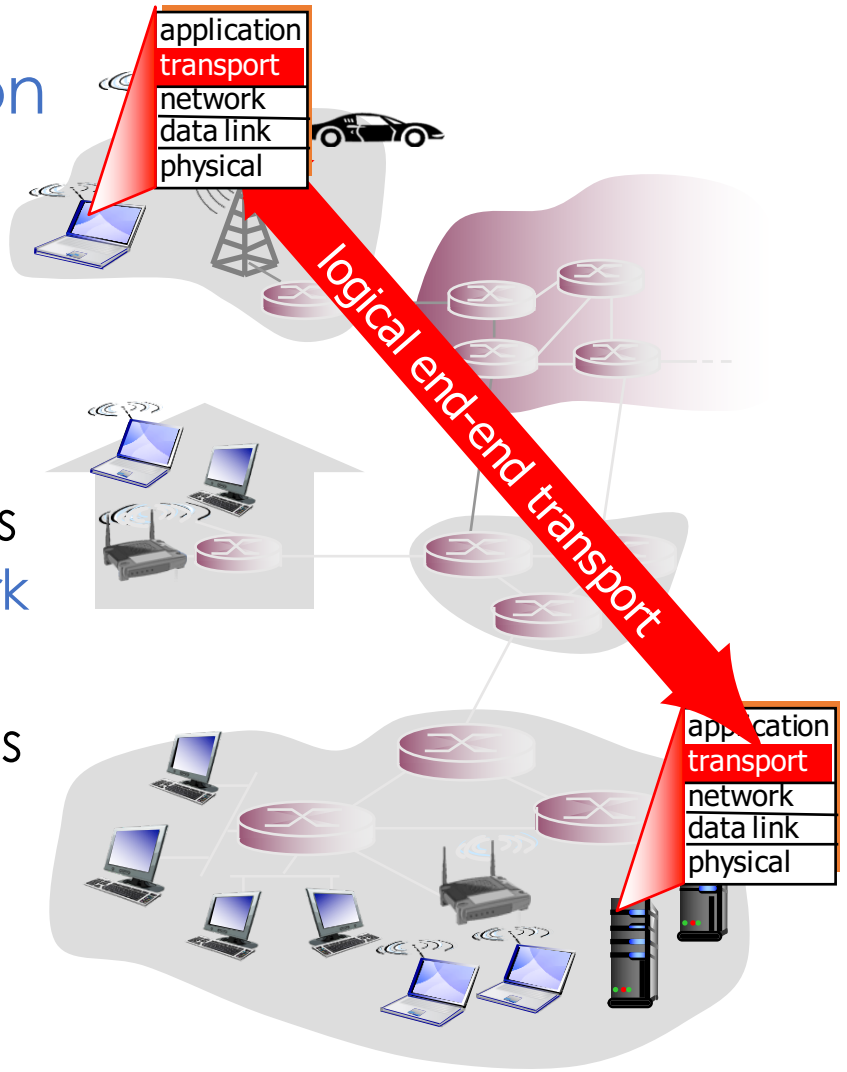
- Understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Outline

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

Transport Services and Protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - recv side: reassembles segments into messages, passes to app layer
- Available transport protocols
  - **TCP** and **UDP**



Transport vs. Network Layer

- **Network layer:**
  - logical communication between hosts
  - Host-to-host
- **Transport layer:**
  - logical communication between processes relies on, enhances, network layer services
  - End-to-end (process-to-process)

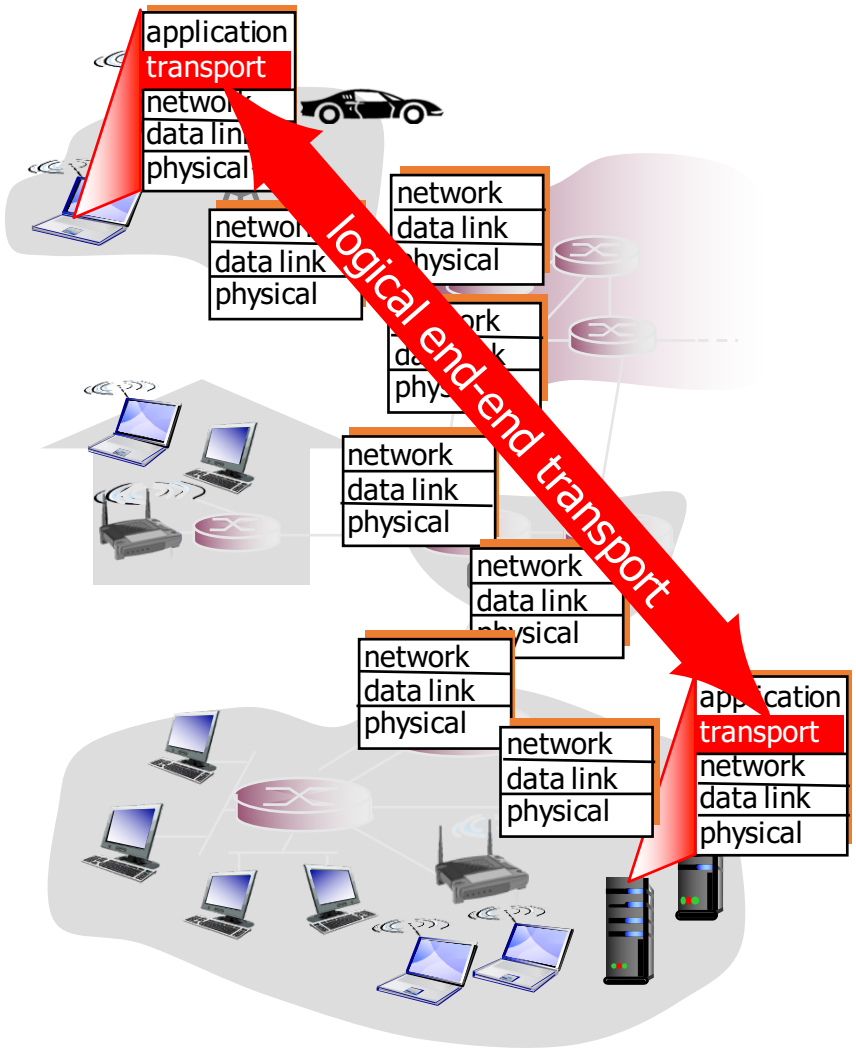
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet Transport Protocols

- Reliable, in-order delivery: TCP
  - congestion control
  - acknowledgement
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
  - Send as many as possible
- Services not available:
  - delay guarantees
  - bandwidth guarantees



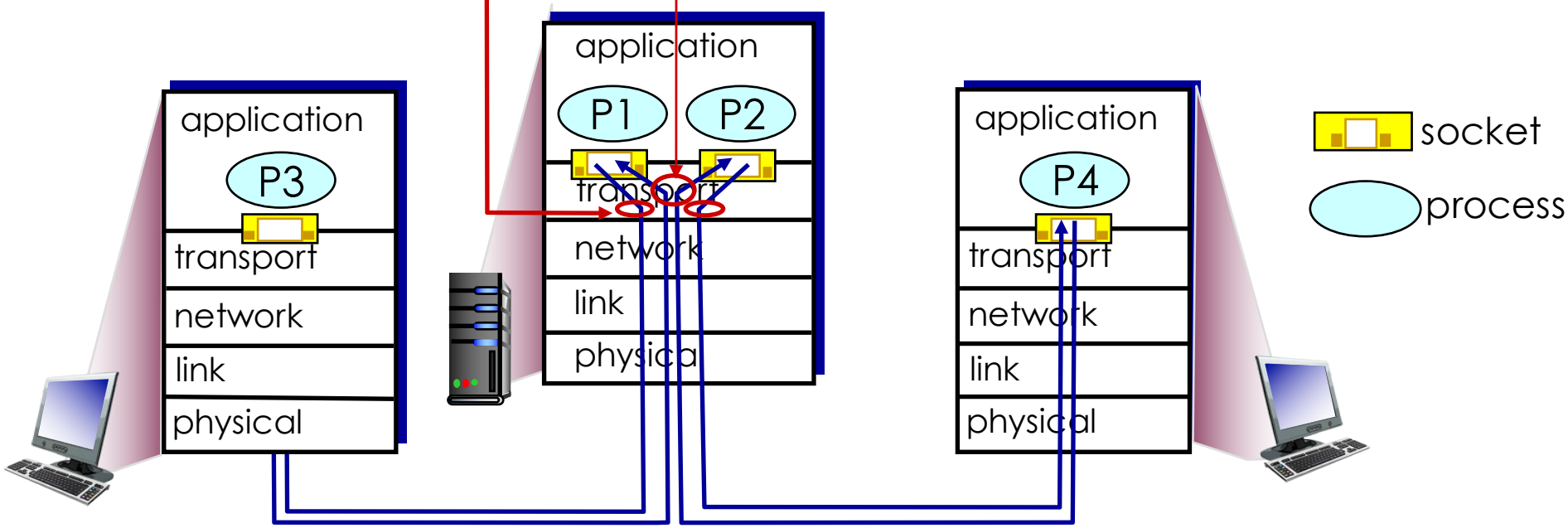
Outline

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

Multiplexing/Demultiplexing

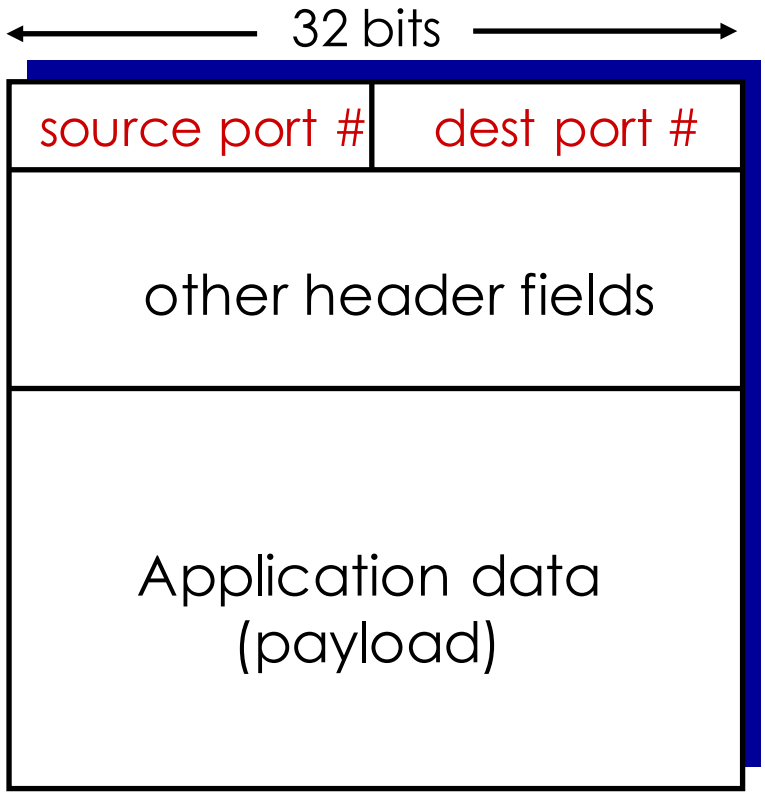
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct **socket** *(unique proc ID)*



How Demultiplexing Works?

- Host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

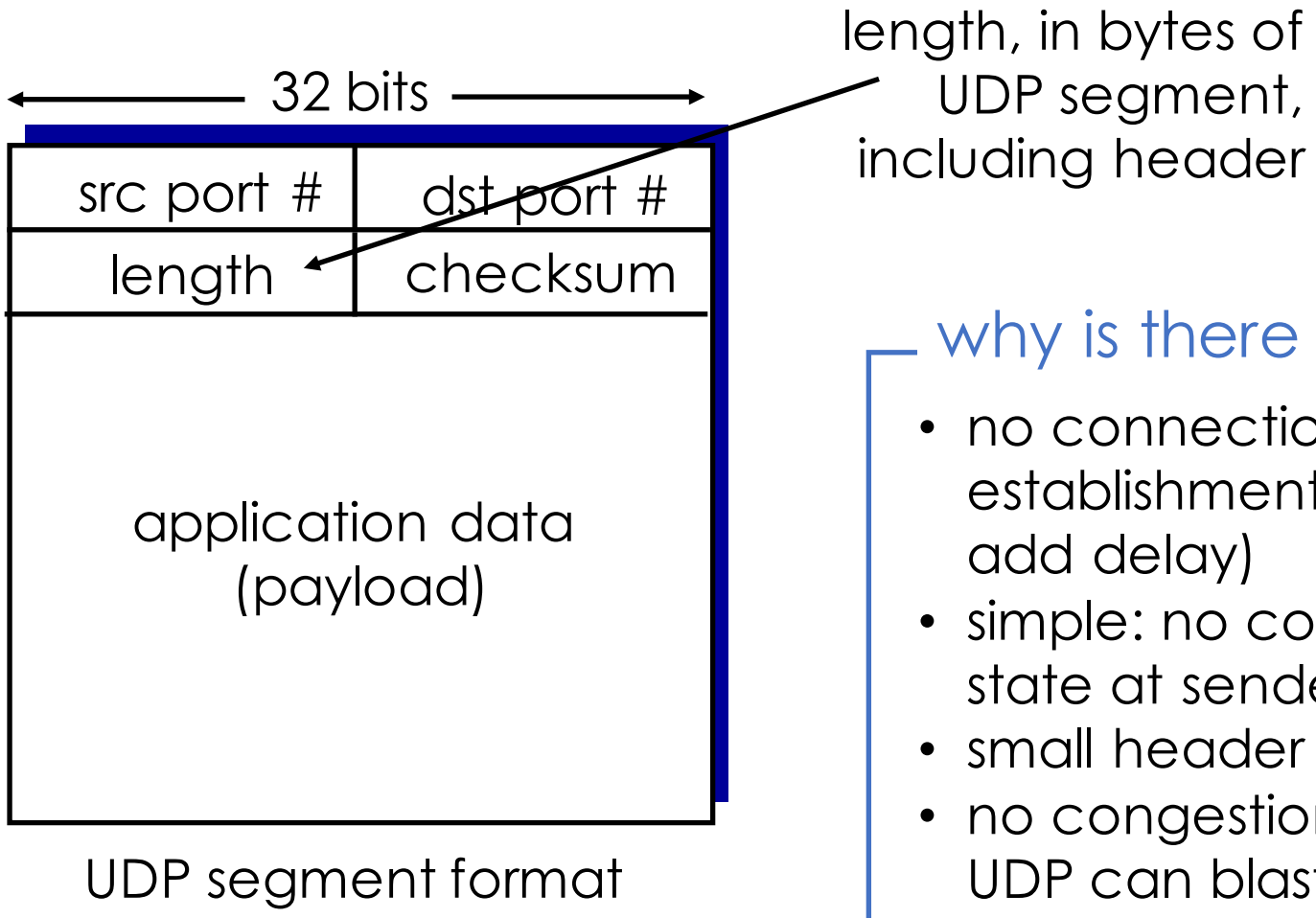
Outline

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

UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- "**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
- Pros:
  - low latency
  - no state → support more users
  - smaller packet header
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- Reliable transfer over UDP:
  - add reliability at application layer via error recovery

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" 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

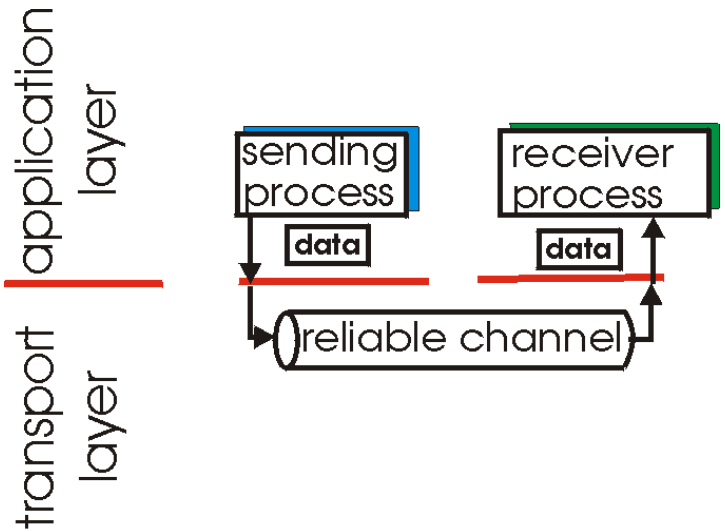
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Outline

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

What is Reliable Data Transfer?

- Important in application, transport, link layers
  - top-10 list of important networking topics!

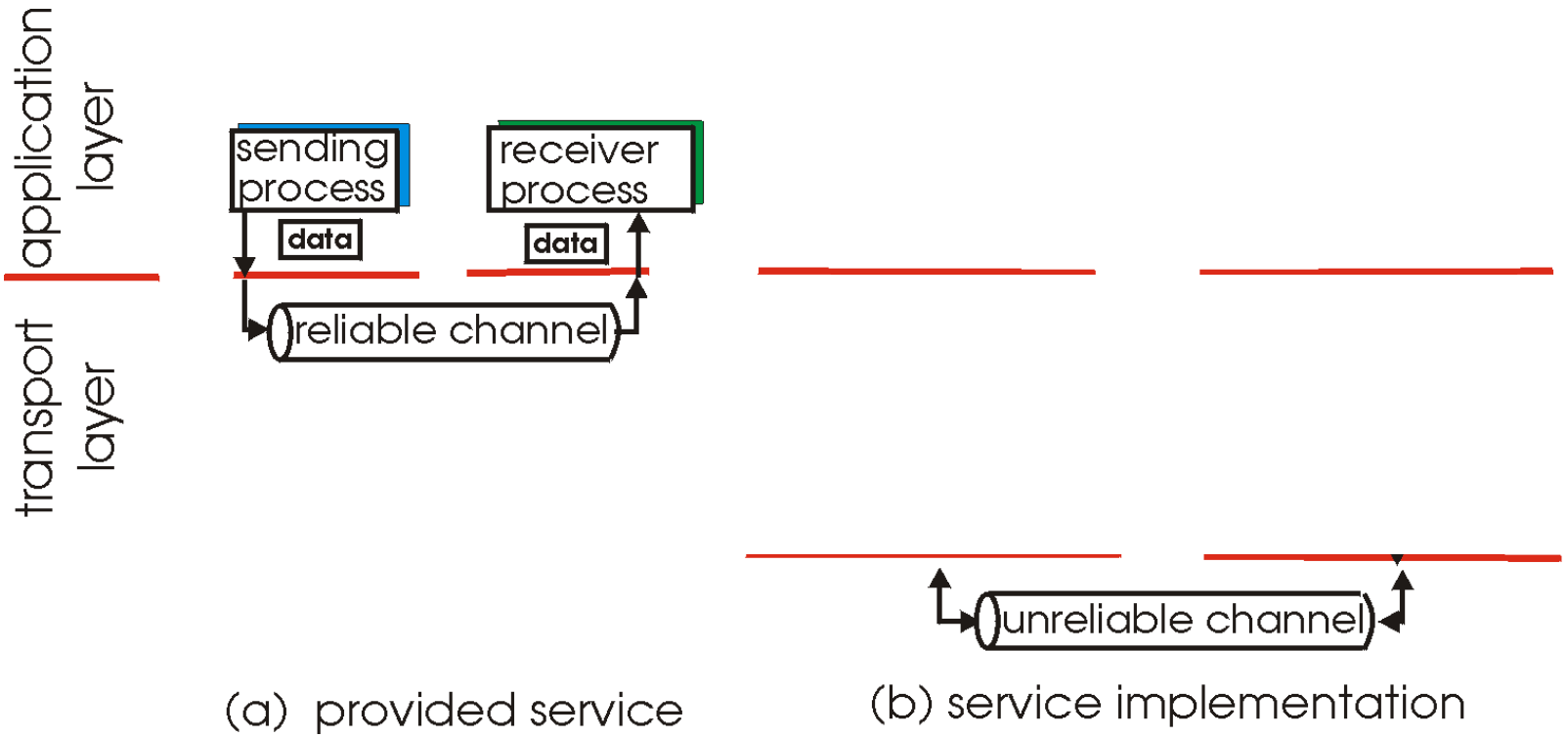


(a) provided service

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

What is Reliable Data Transfer?

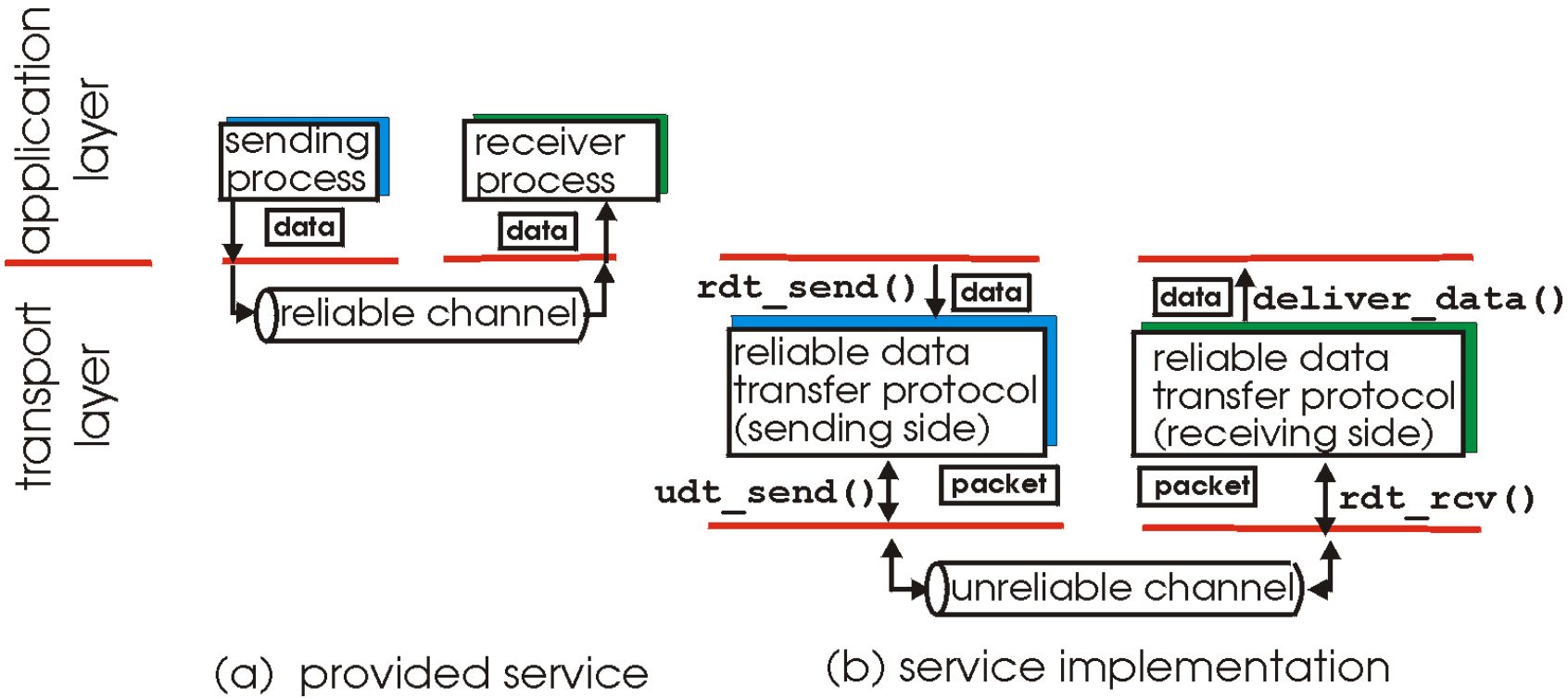
- Important in application, transport, link layers
  - top-10 list of important networking topics!



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

What is Reliable Data Transfer?

- Important in application, transport, link layers
  - top-10 list of important networking topics!

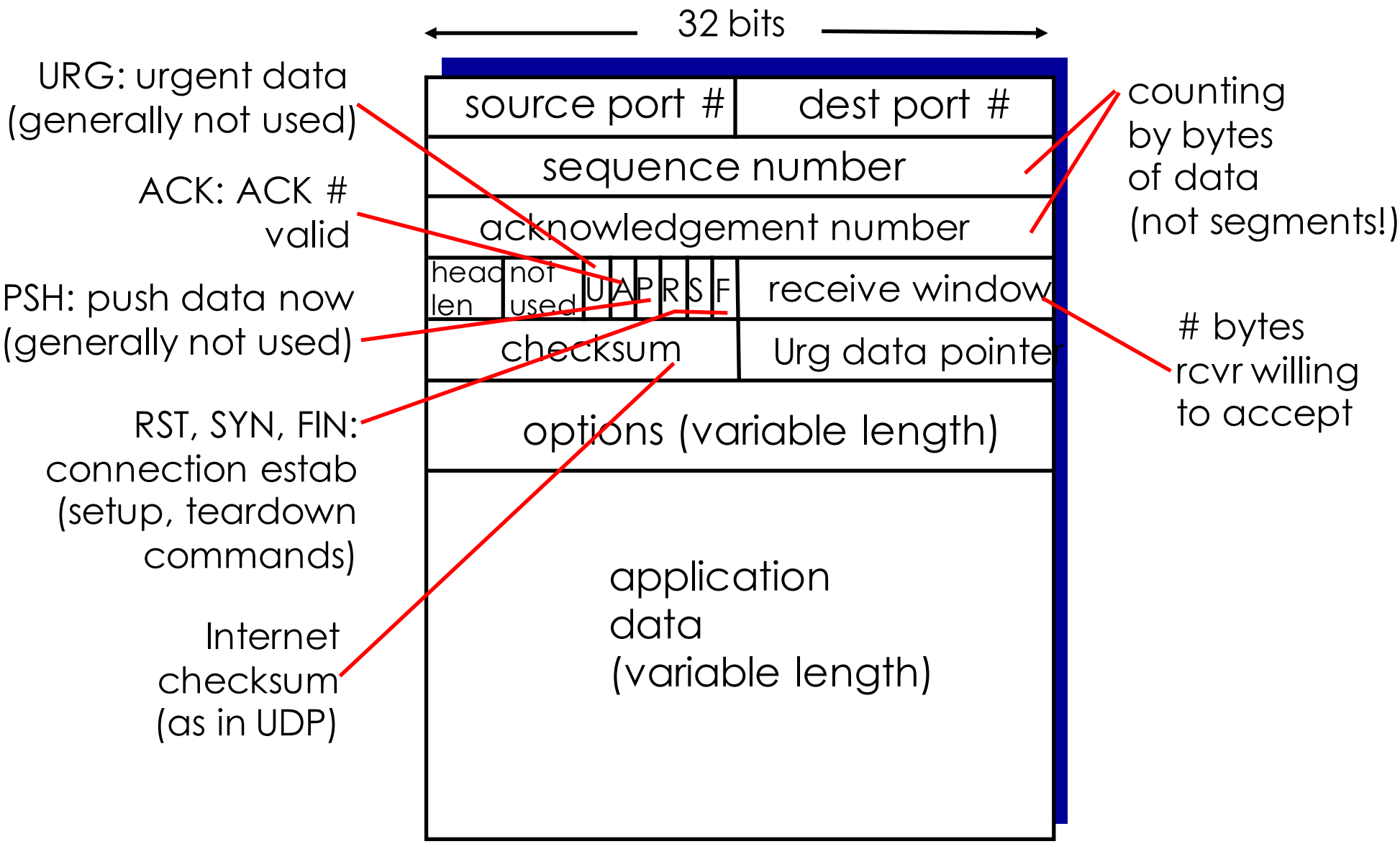


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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

TCP Segment Structure

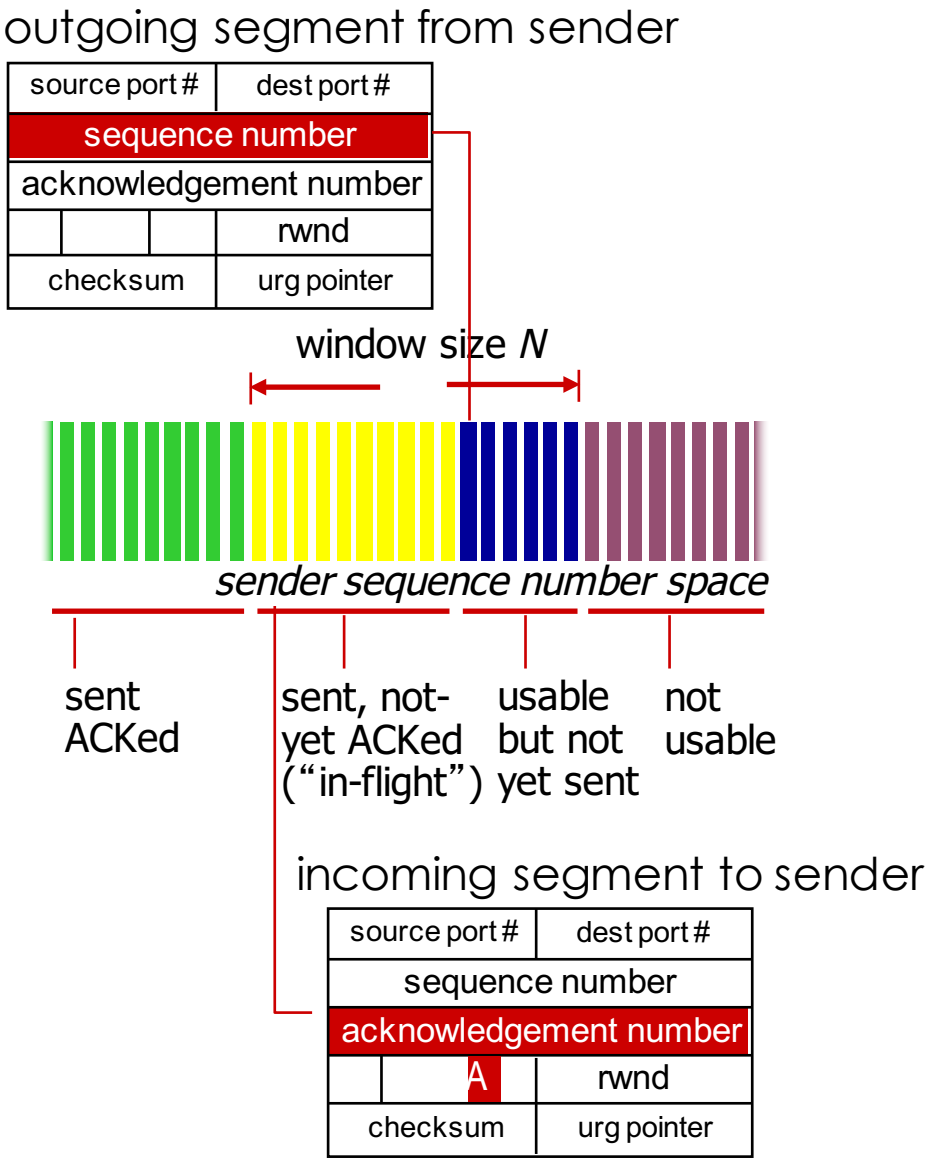


TCP Seq. Numbers, ACKs

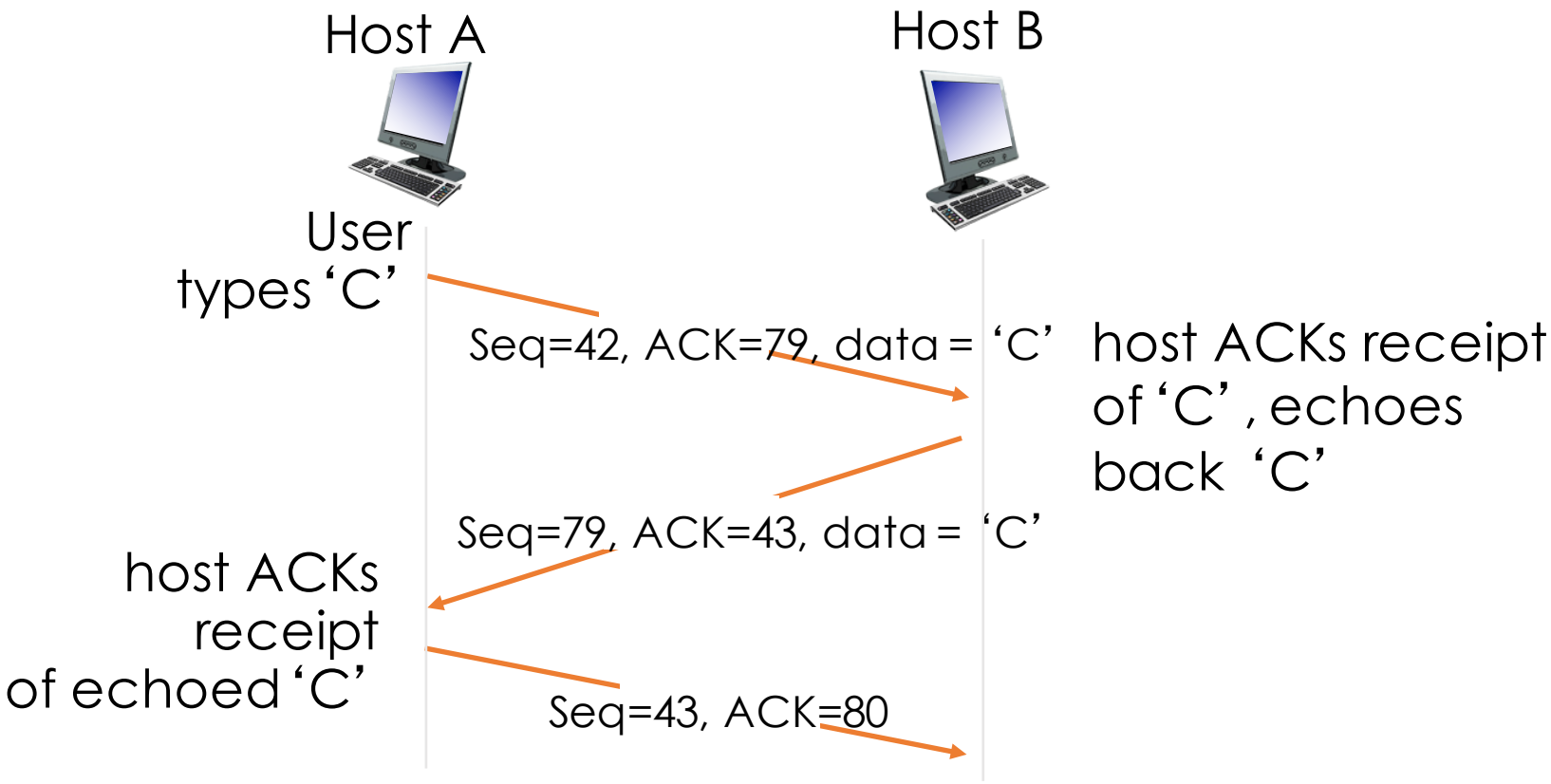
- **Sequence numbers:**
  - byte stream "number" of first byte in segment's data
- **Acknowledgements:**
  - seq # of next byte expected from other side
  - cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP Seq. 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?

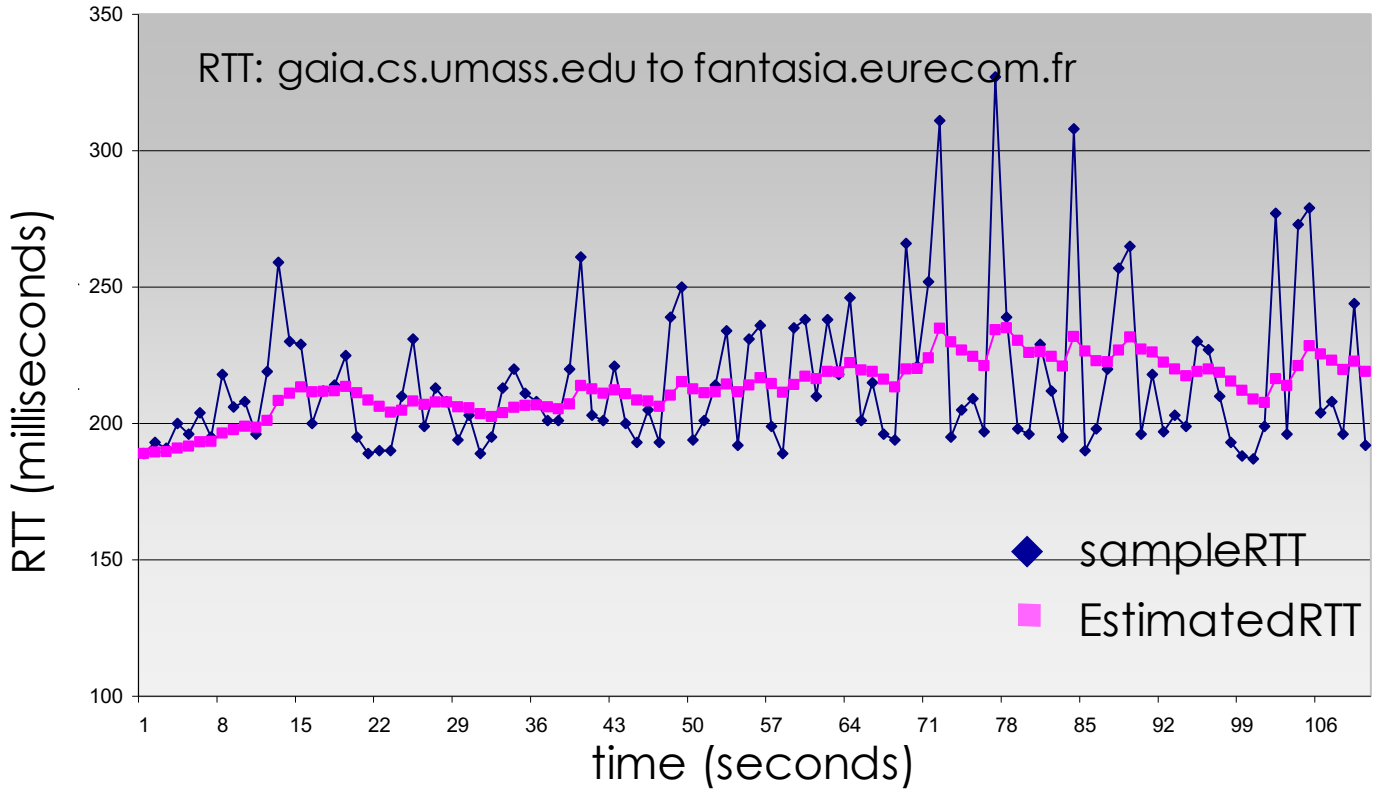
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP Round Trip Time, Timeout

exponential weighted moving average (EWMA):

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

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



TCP Round Trip Time, Timeout

- **Q:** what value should be used for TCP's timeout interval?
  - 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
- Connectionless transport: UDP
- connection-oriented transport: TCP
  - Segment structure
  - **Reliable data transfer**
  - Flow control
  - Connection management
- 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`**

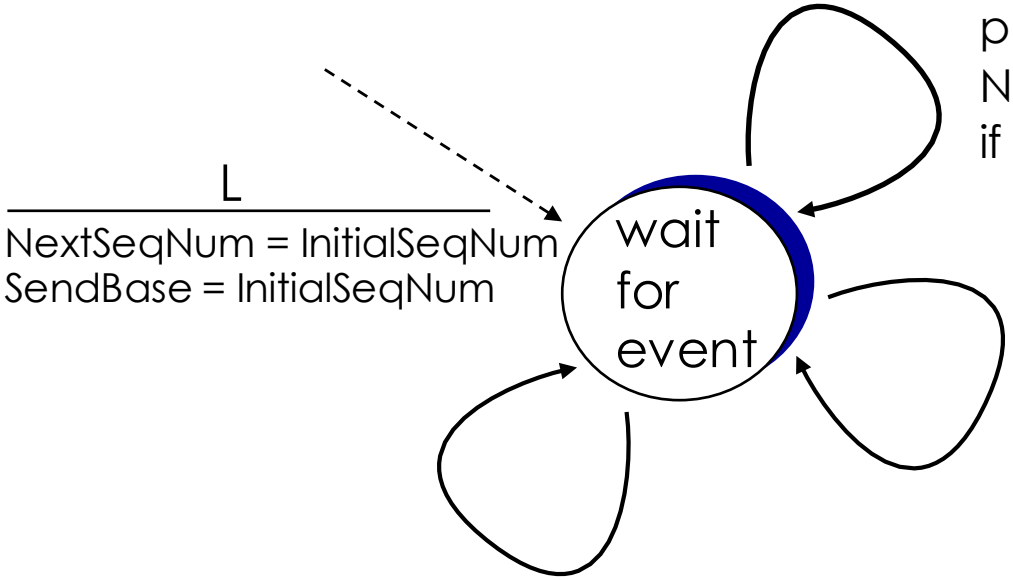
timeout:

- 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 Sender (Simplified)



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

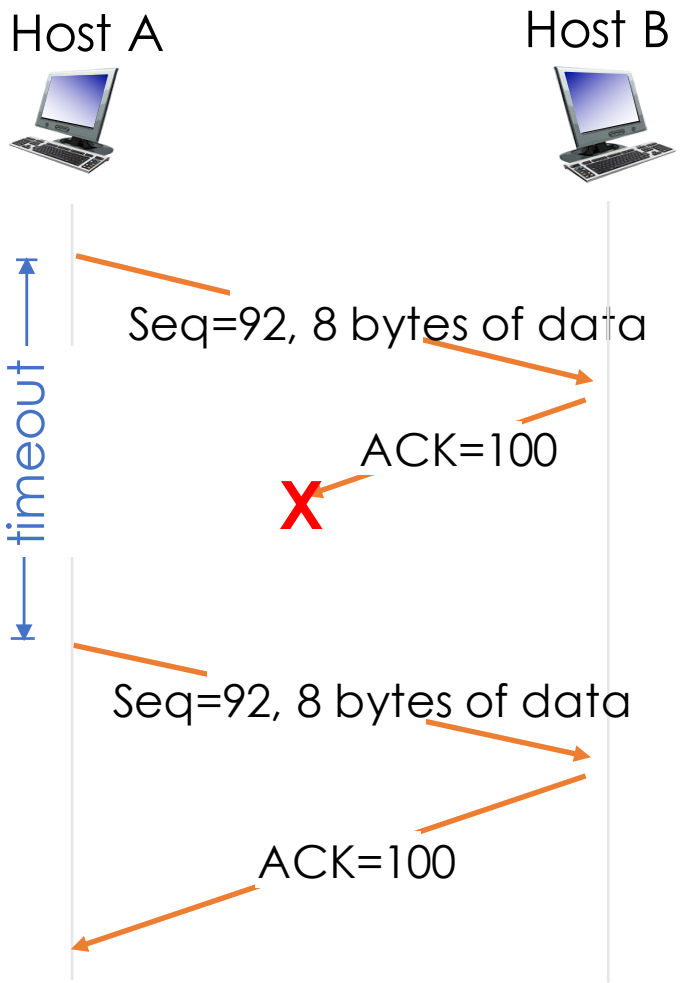
timeout

retransmit not-yet-acked
segment with smallest seq. #
start timer

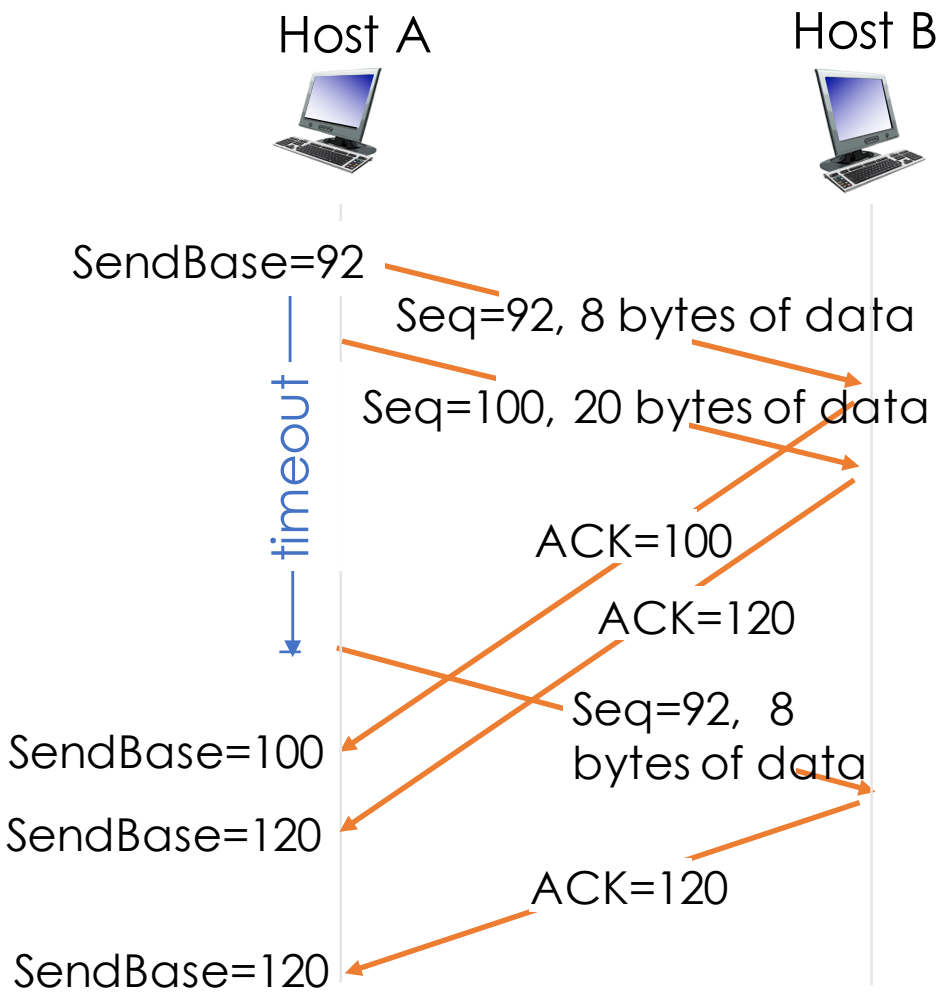
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
    else stop timer
}
```

TCP: Retransmission Scenarios

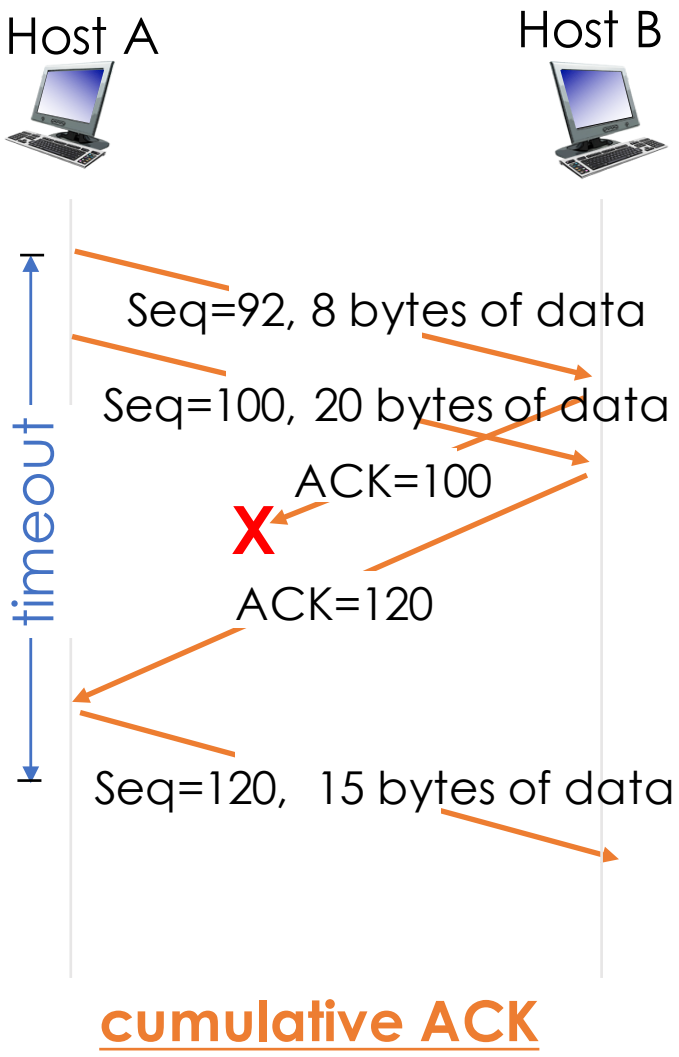


lost ACK scenario



premature timeout

TCP: Retransmission Scenarios



cumulative 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

TCP Fast Retransmit

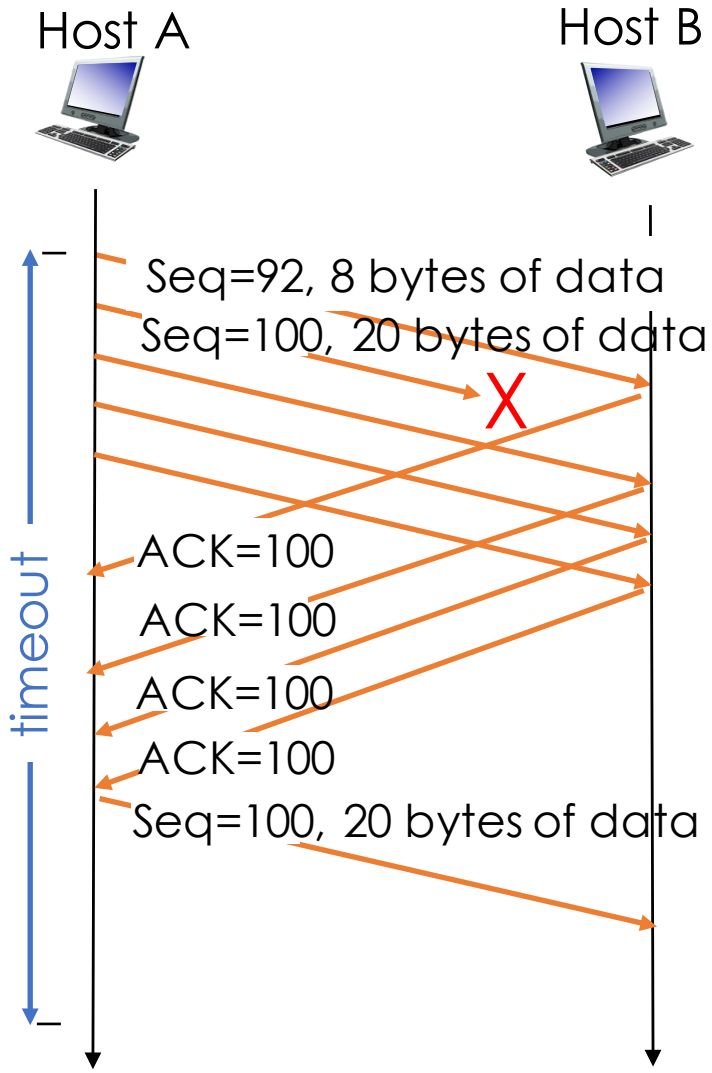
- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), immediately resend unacked segment with smallest seq #

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

TCP Fast Retransmit

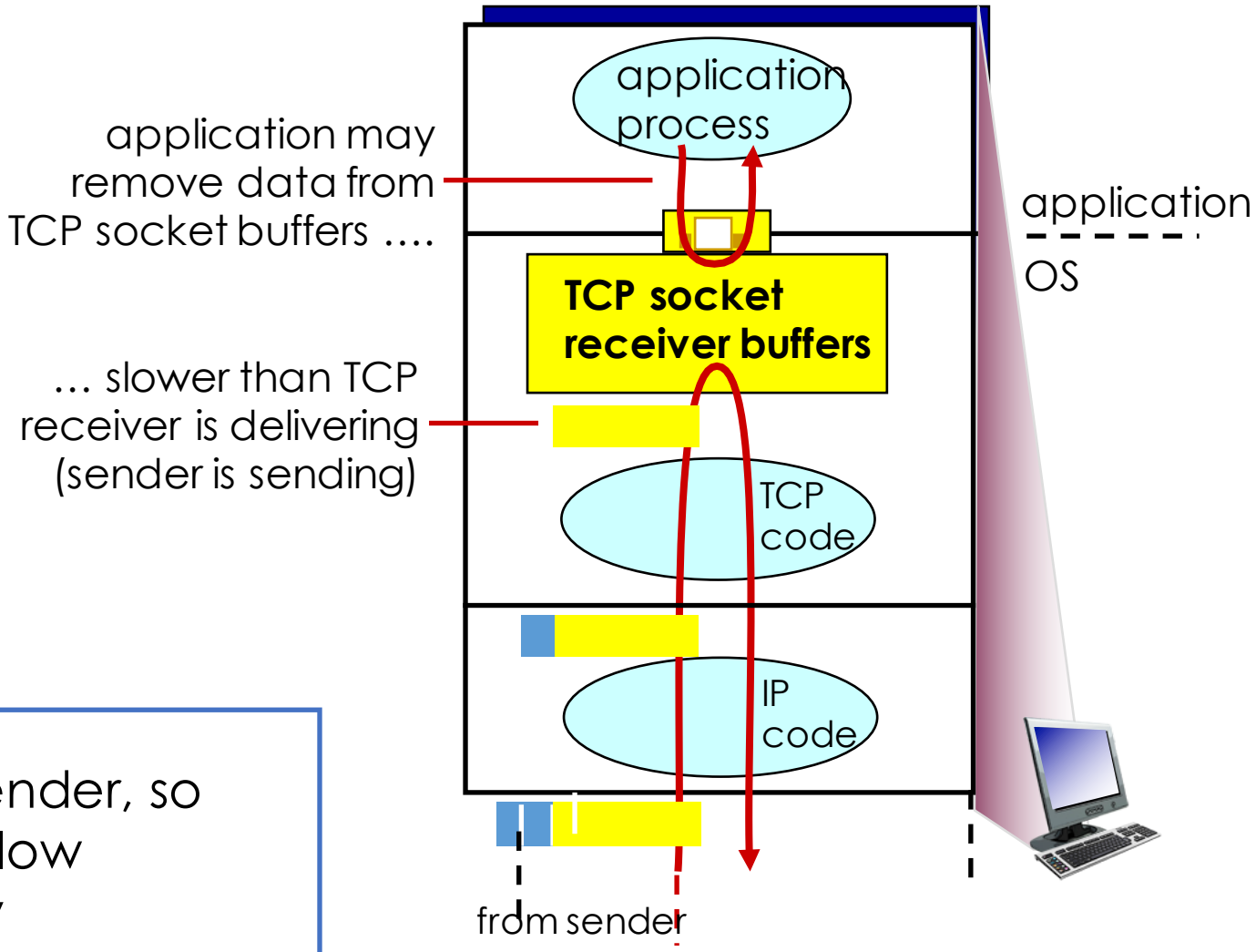


fast retransmit after sender receipt of triple duplicate ACK

Outline

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

TCP Flow Control

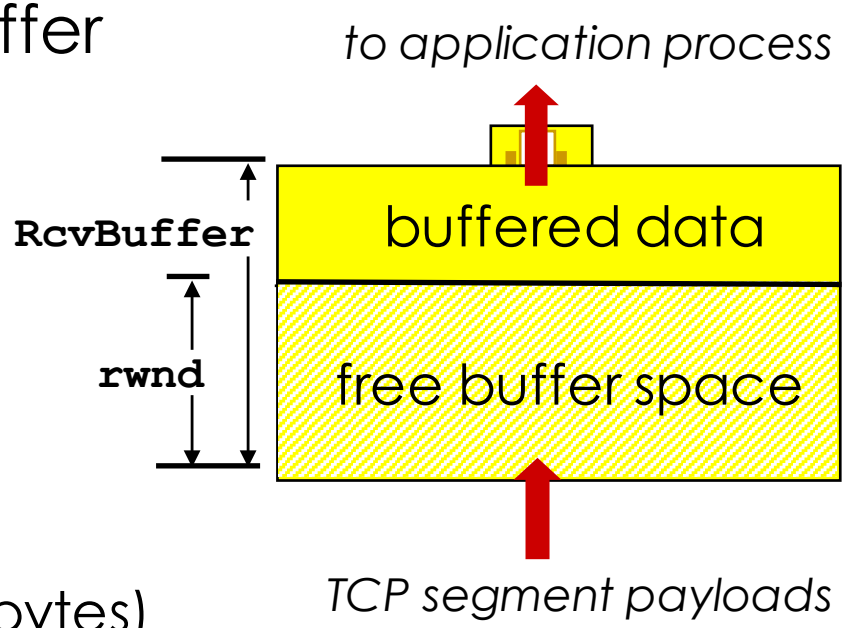


flow control

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

TCP Flow Control

- Why? guarantees receive buffer will not overflow
- 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 auto-adjust **`RcvBuffer`**
- Sender limits the amount of unacked ("in-flight") data to receiver's **`rwnd`** value



receiver-side buffering

`LastByteSent - LastByteAcked ≤ rwnd`

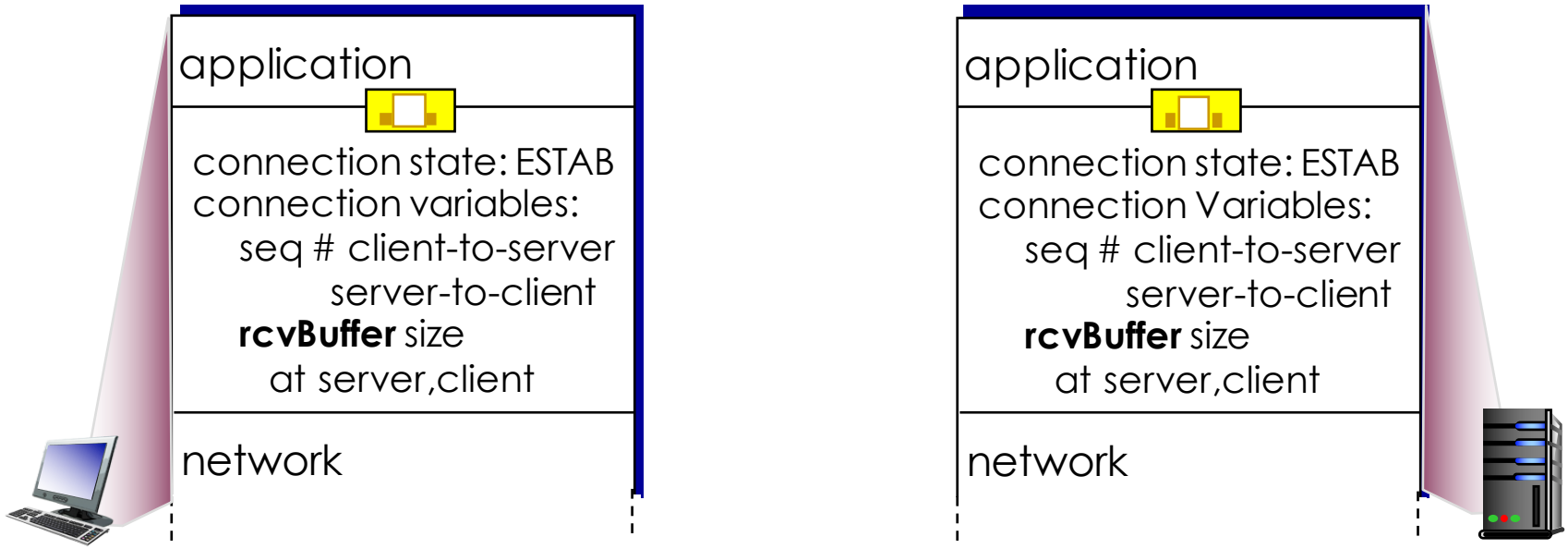
How about UDP?

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - **Connection management**
- 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

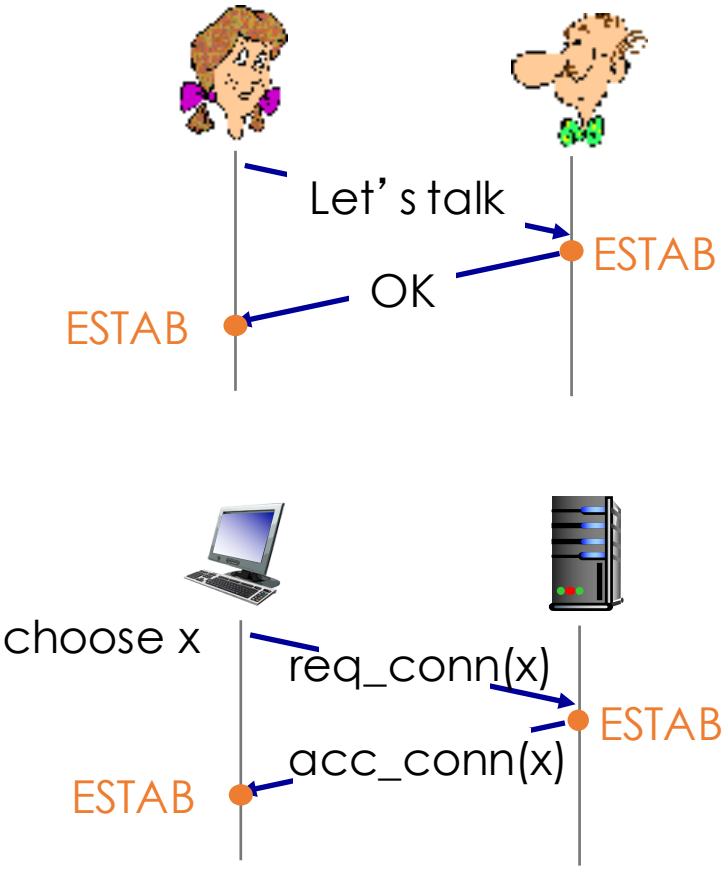



```
Socket clientSocket =
 newSocket("hostname","port number");
```

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

Agreeing to Establish a Connection

2-way handshake:



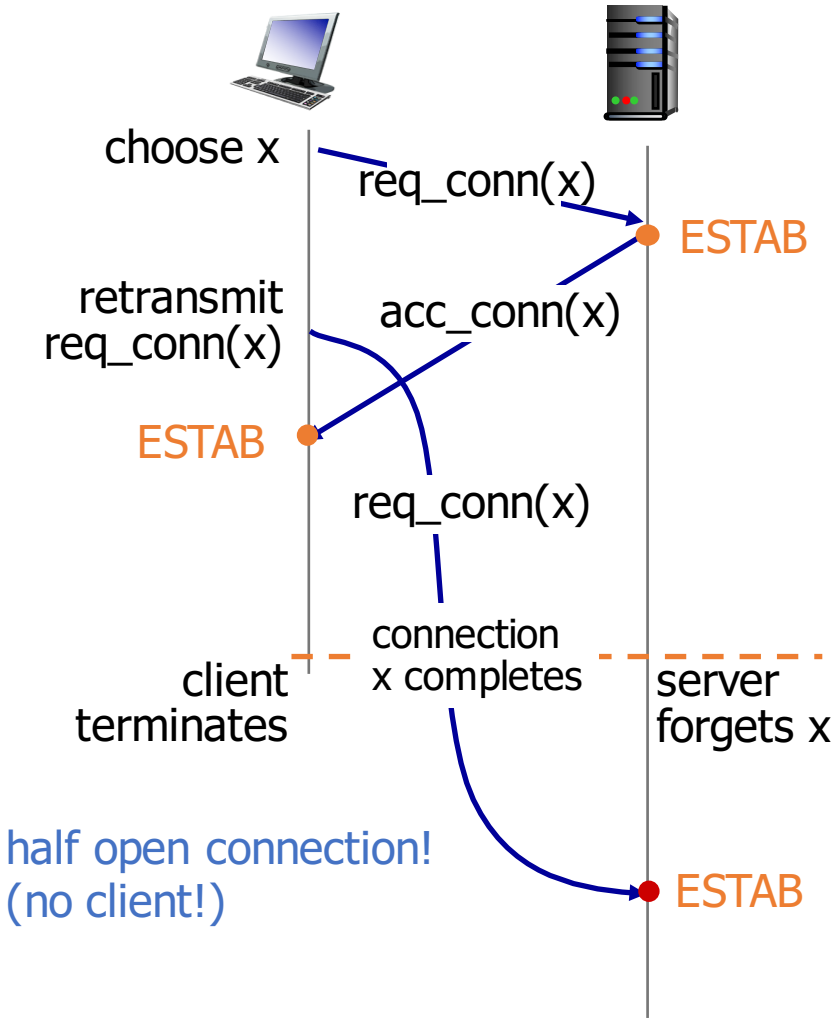
Q: will 2-way handshake always work in network?

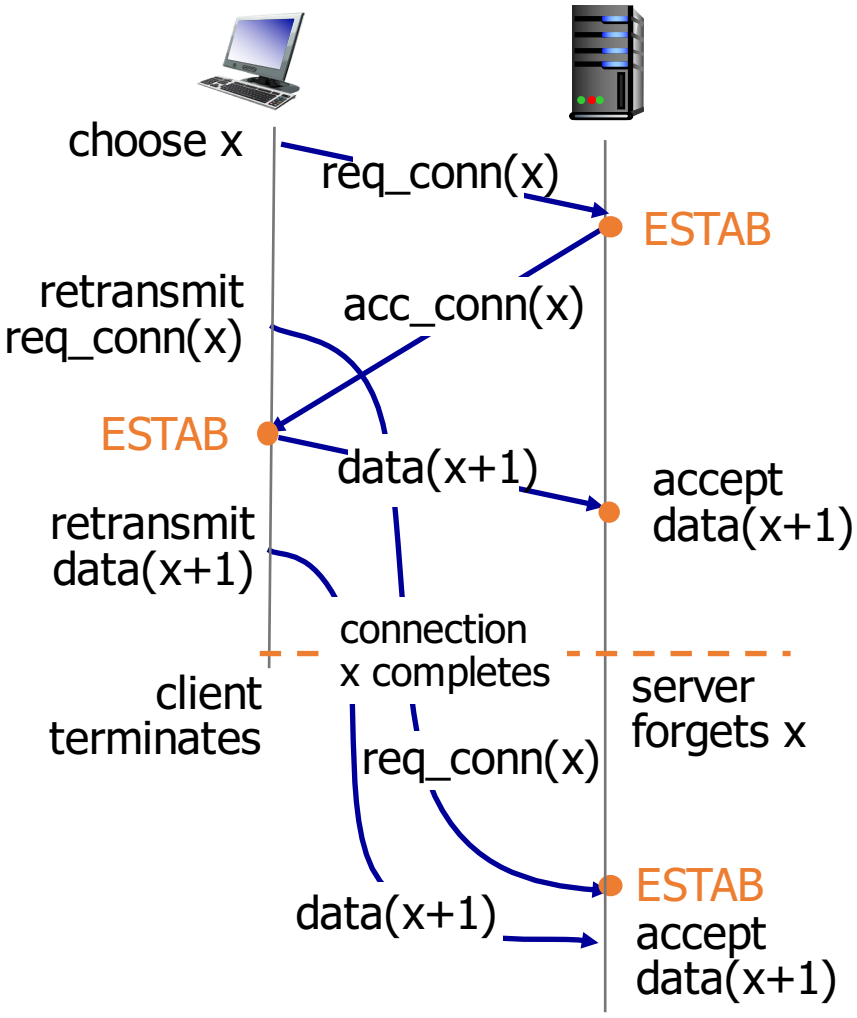
A: No

- TCP is bidirectional connection
- Both sides randomly pick their initial sequence numbers

Agreeing to Establish a Connection

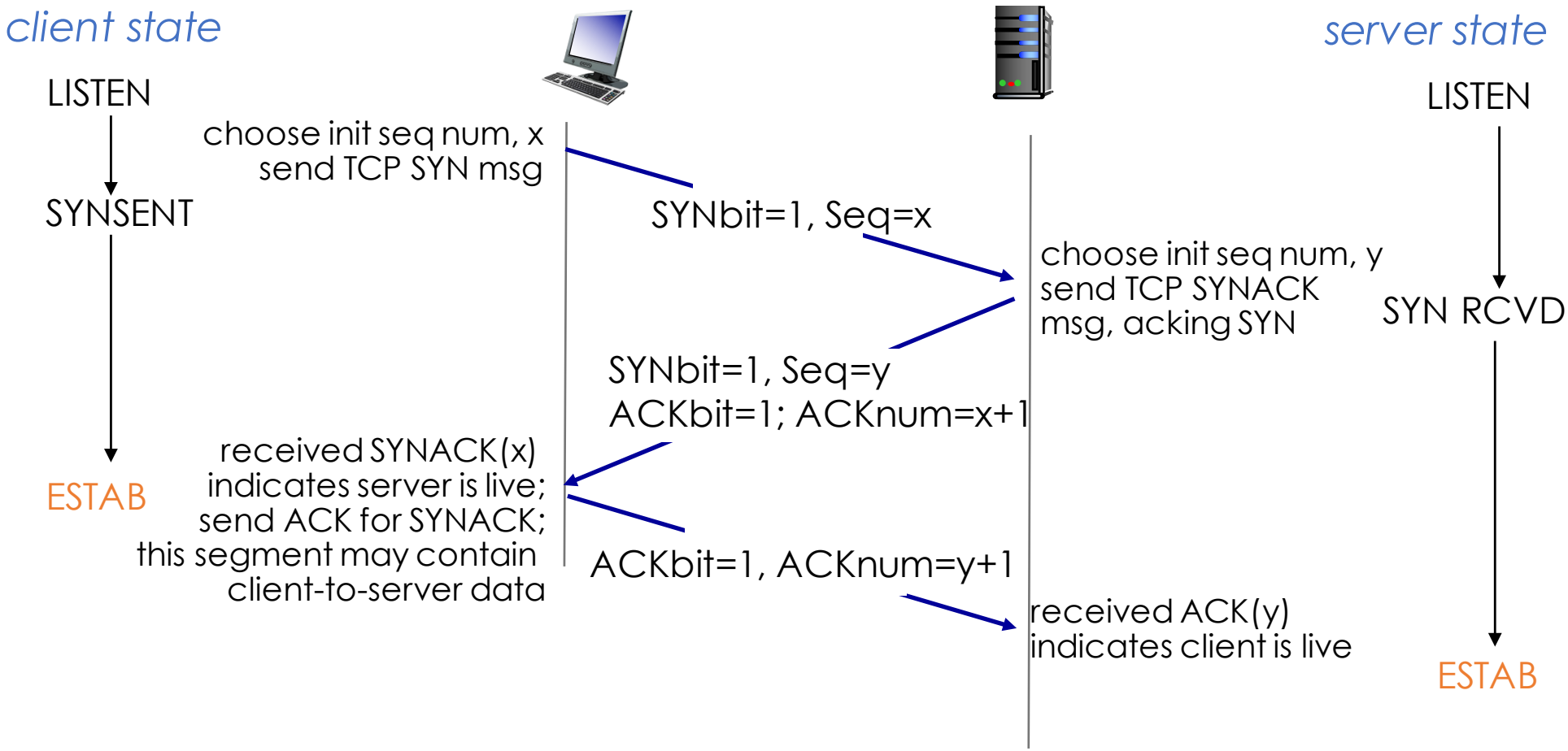
2-way handshake failure scenarios:



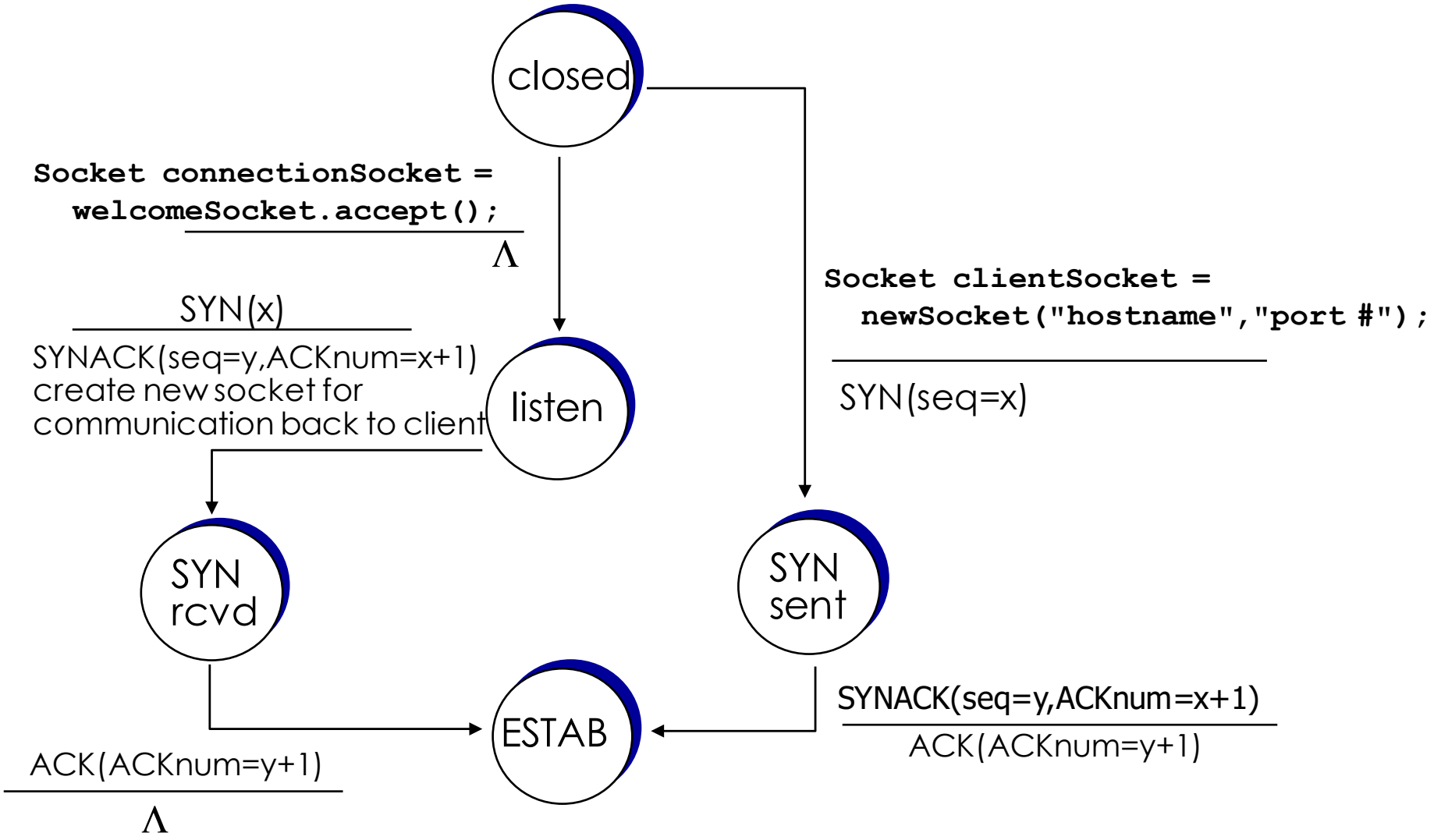


Connection terminated due to long delay

TCP 3-way Handshake



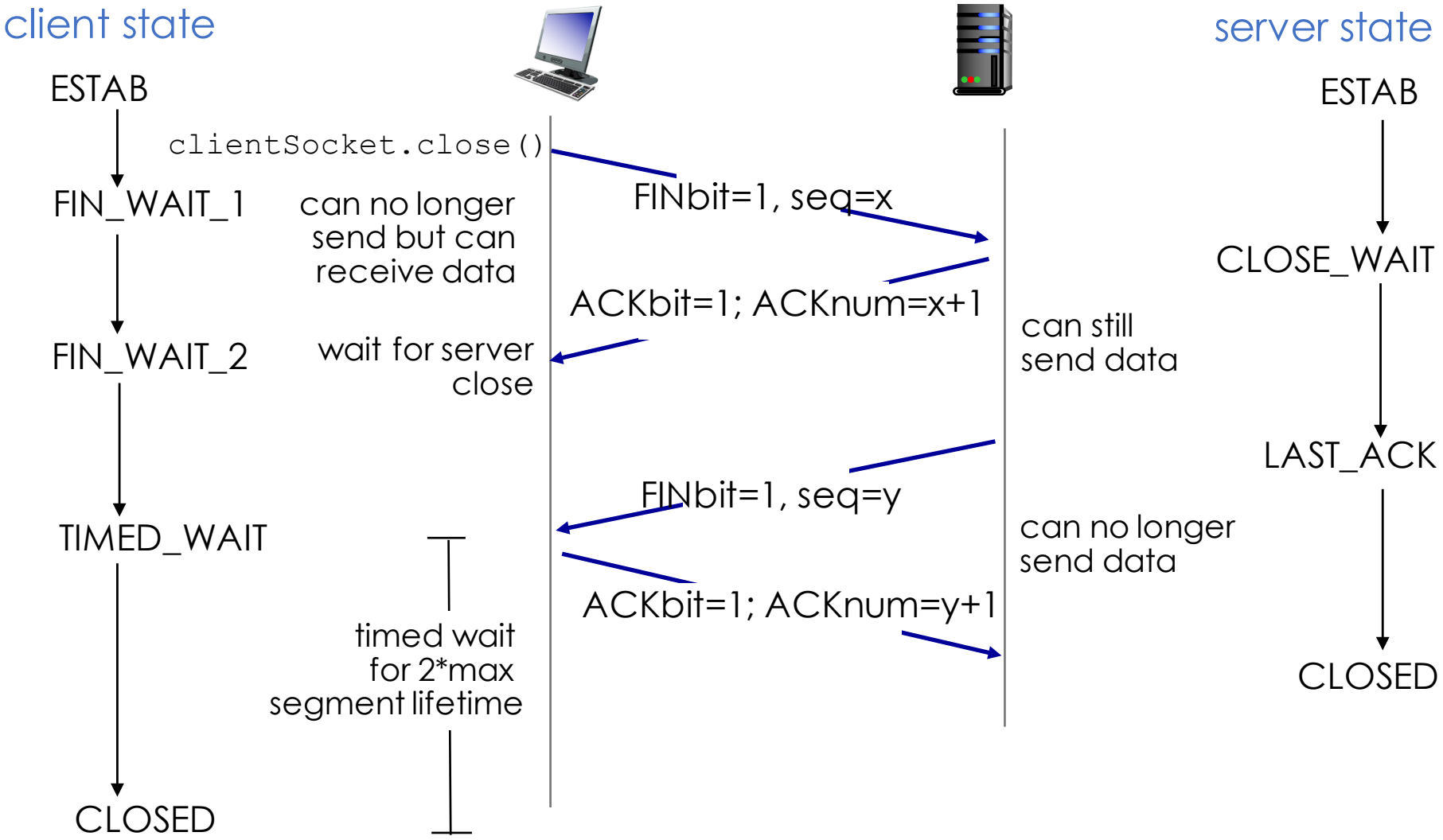
TCP 3-way Handshake: FSM



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

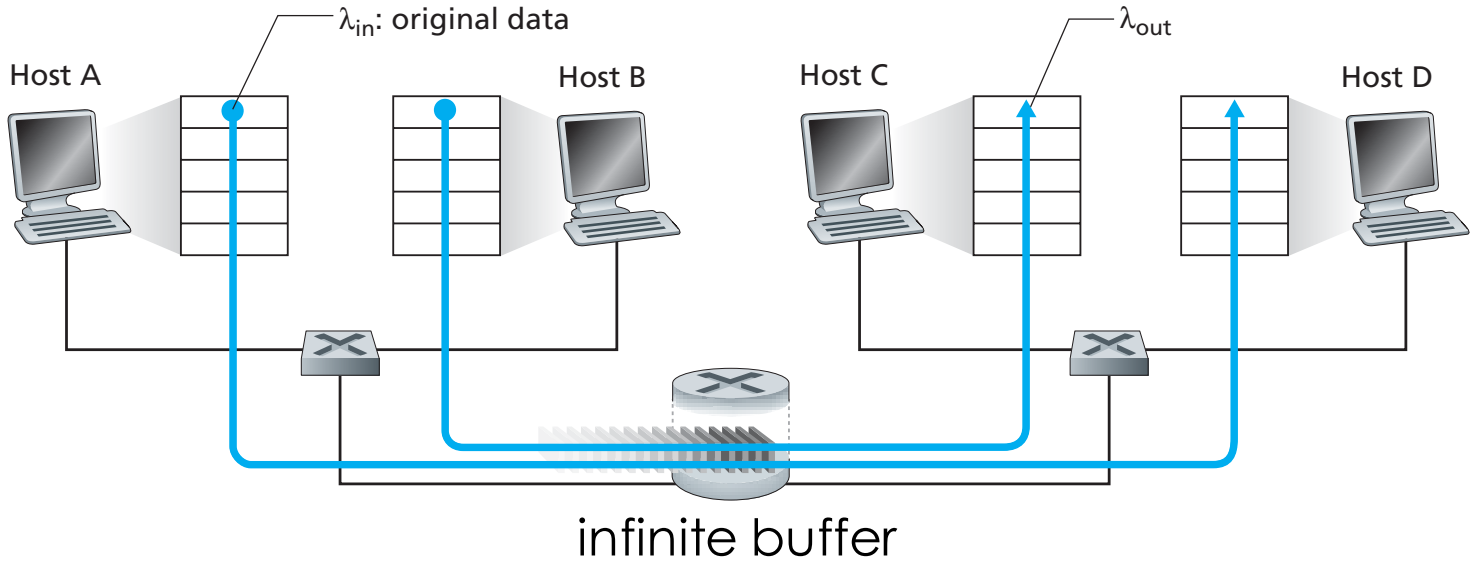


Outline

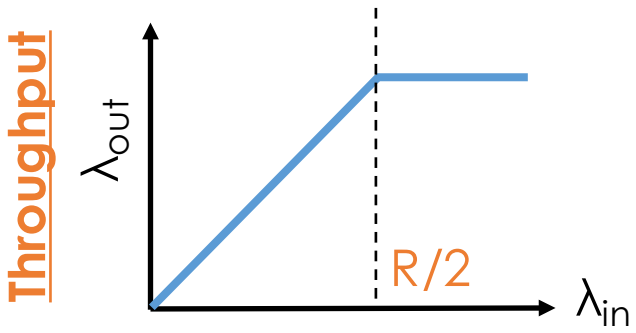
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- **Congestion Control**

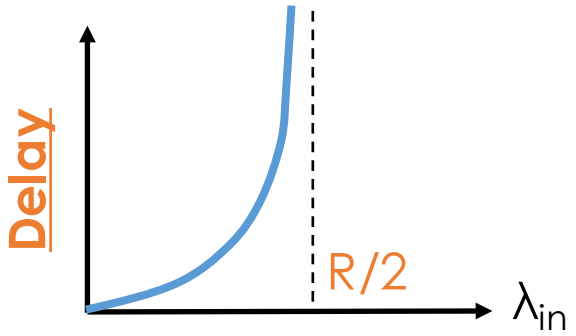
Congestion Example: Infinite Buffer

Two connections share a link with infinite buffer



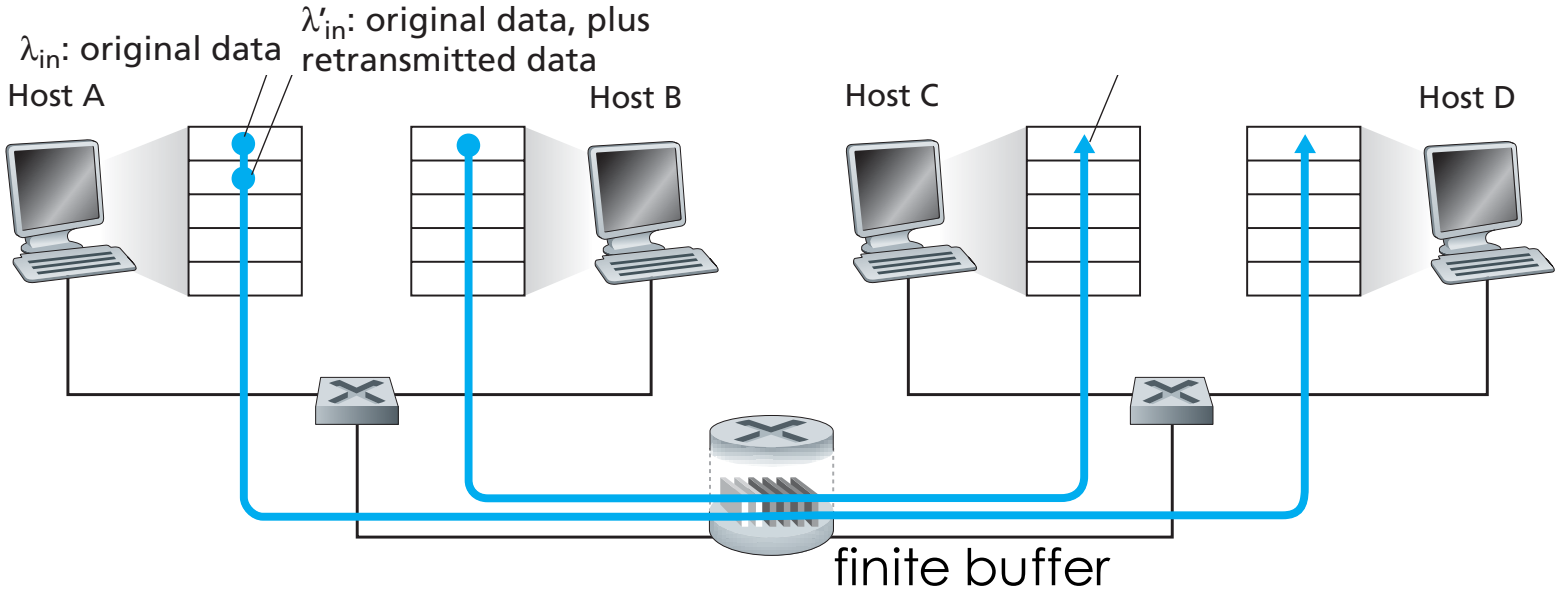
- Say both A and B send λ_{in} bytes/sec
- When λ_{in} exceeds R/2, the average number of queued packets in the router is unbounded → delay becomes infinite





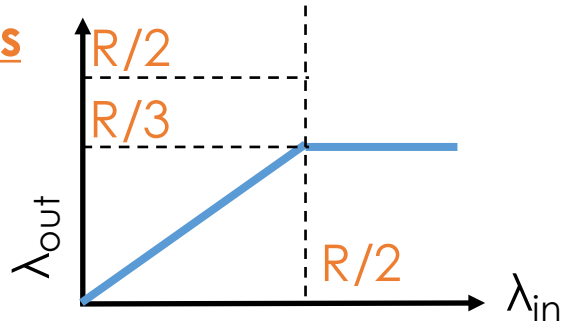
Congestion Example: Finite Buffer

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

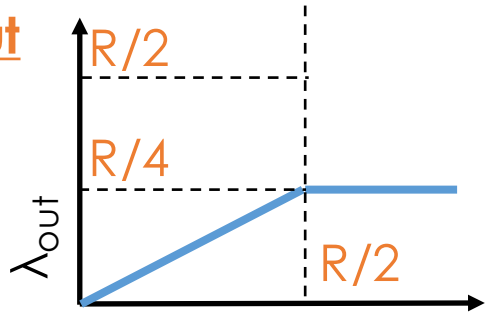


- With retransmission, **offered load** becomes λ'_{in} larger then λ_{in}
- Capacity wastes: 1) packet loss: retransmission, 2) timeout: unnecessary retransmissions

loss



No loss but timeout



TCP Congestion Control

- End-to-end control, rather then 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 packets cnanot exceed **cwnd**

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \min(\texttt{rwnd}, \texttt{cwnd})$$

$$\text{rate} \approx \frac{\texttt{cwnd}}{\text{RTT}}$$

- How to determine whether there is congestion?
  - Buffer overflow leads to losses
  - How to detect? 1) timeout, or 2) receiving 3 dup-ACK
- How to change the rate?
  - Arrival of ACK indicates "nothing wrong"
  - Missing ACK implies congestion
  - Use ACKs to trigger an increase in **cwnd** → self clocking
    Q: how to adjust the value of **cwnd**?

Bandwidth Probing

Key idea of TCP's congestion control

- Keep increasing the rate (value of **`cwnd`**) in response to arriving ACKs
- Decrease the rate (value of **`cwnd`**) if loss event occurs

TCP's congestion control algorithm [RFC 5681]

- Slow start
- Congestion avoidance
- Fast recovery

analogy

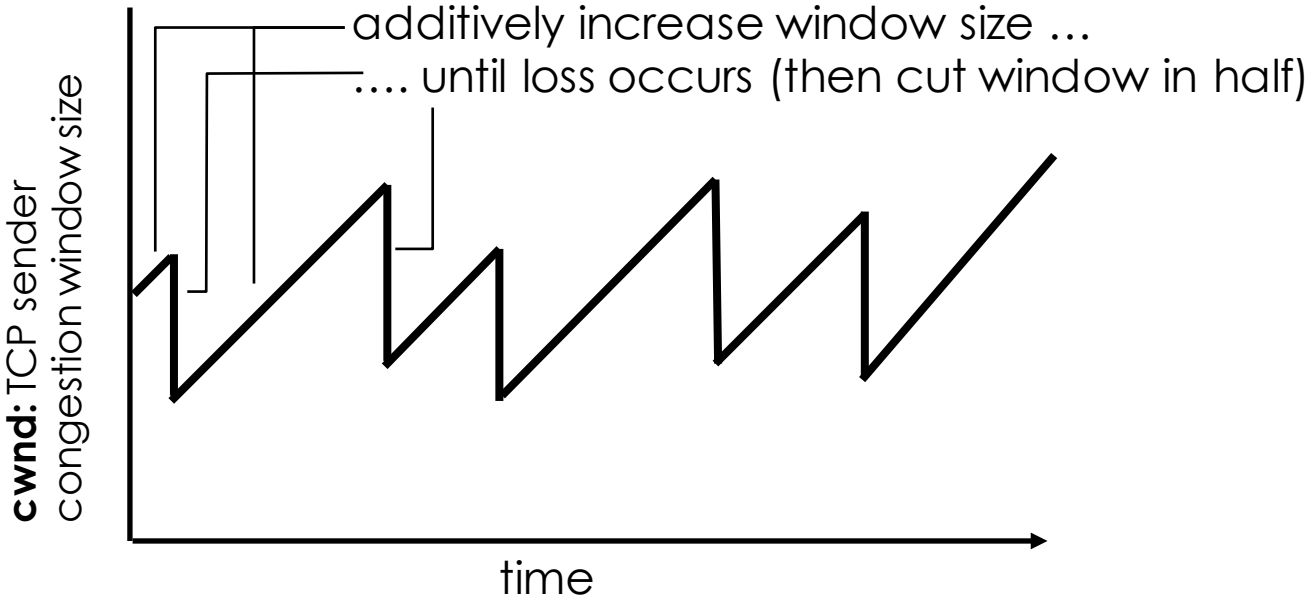
Kids request for goodies

- More and more until the parents finally say "NO"
- Back off a bit

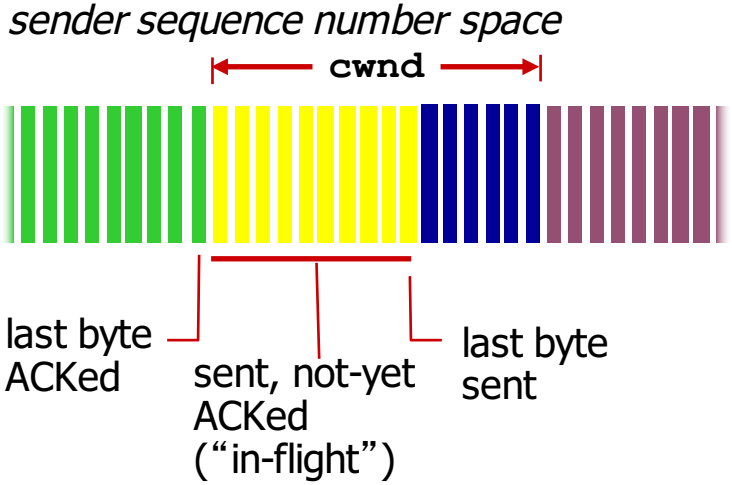
TCP Congestion Control

- sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
- **How?**
  Additive Increase Multiplicative Decrease (AIMD)
  - *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's Achievable Rate



- sender limits transmission:

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

- **cwnd** is dynamic, function of perceived network congestion

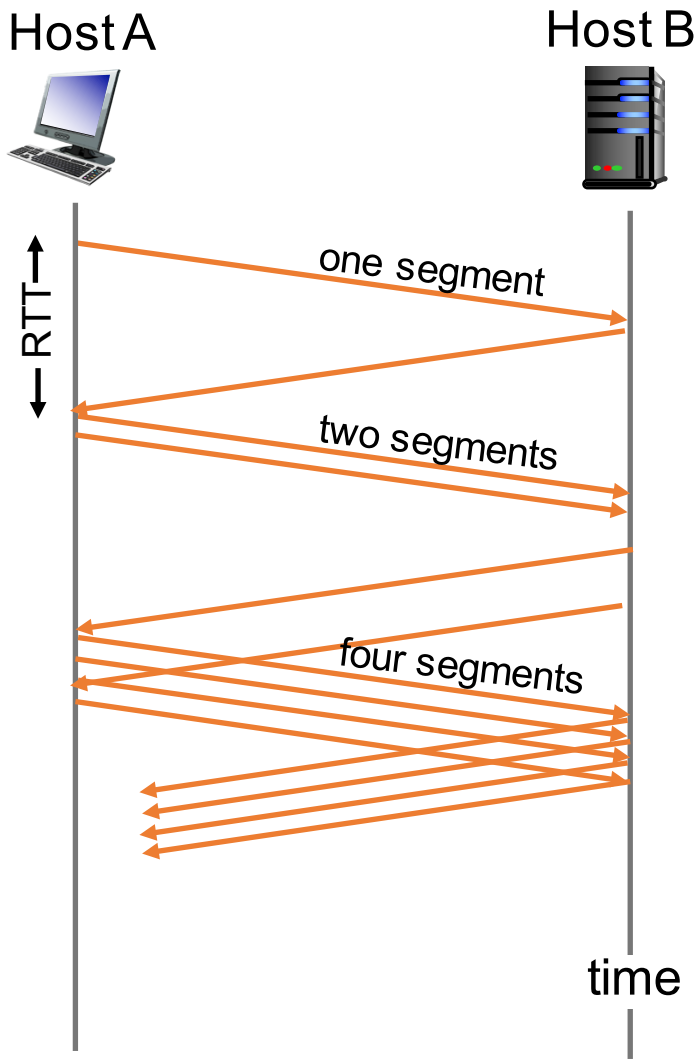
TCP sending rate:

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

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

TCP 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



Detecting, Reacting to Loss

Depend on how we define a "loss" event

- Loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - begin the slow start process anew → **`cwnd`** grows exponentially
  - switch to the **congestion-avoidance** mode when **`cwnd`** = threshold → **`ssthresh = cwnd/2`**; increase **`cwnd`** linearly
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - enter the **fast recovery** state → **`cwnd`** is cut in half window then grows linearly
- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

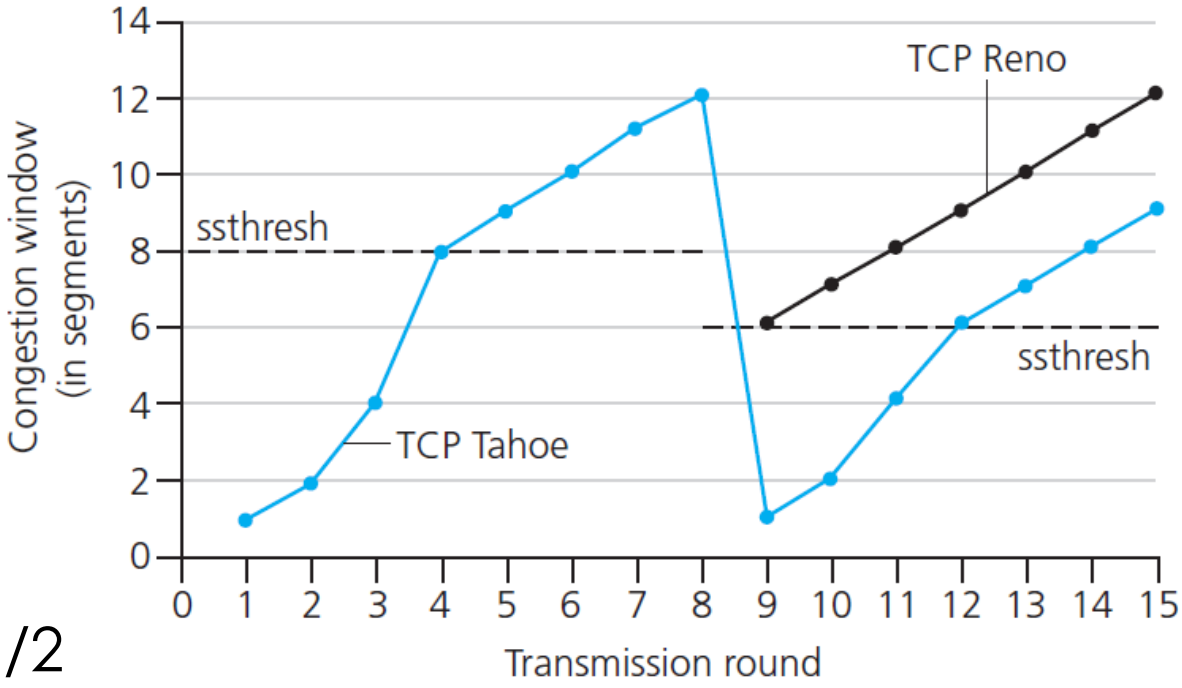
Recap

- **Congestion Avoidance (CA)**
  - When **`Cwnd`** is approaching the level of congestion (i.e., a timeout event), we should increase **`Cwnd`** more conservatively
    → grow linearly, instead of exponentially
  - Use a threshold called **`ssthresh`** to determine whether to enter the CA mode by setting **`ssthresh = Cwnd/2`**
- **Fast Recovery (FR)**
  - **`Cwnd`** is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state
  - Recommended, but not required
  - TCP Tahoe: unconditionally set **`Cwnd = 1`** and restart slow start
  - TCP Reno: halve **`Cwnd`** and then increase **`Cwnd`** linearly

From Slow Start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

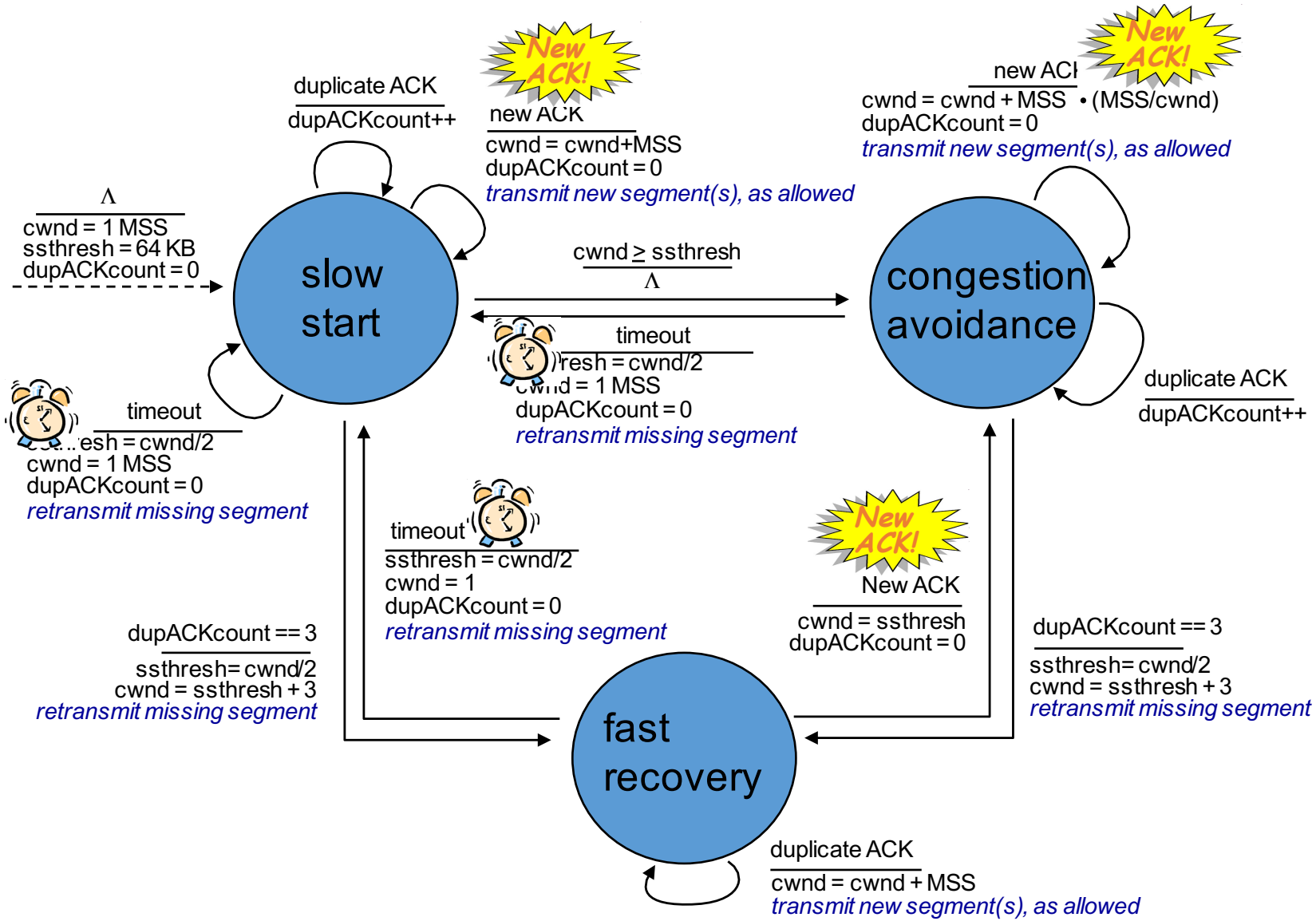
A: when **`cwnd`** gets to 1/2 of its value before timeout



Implementation:

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

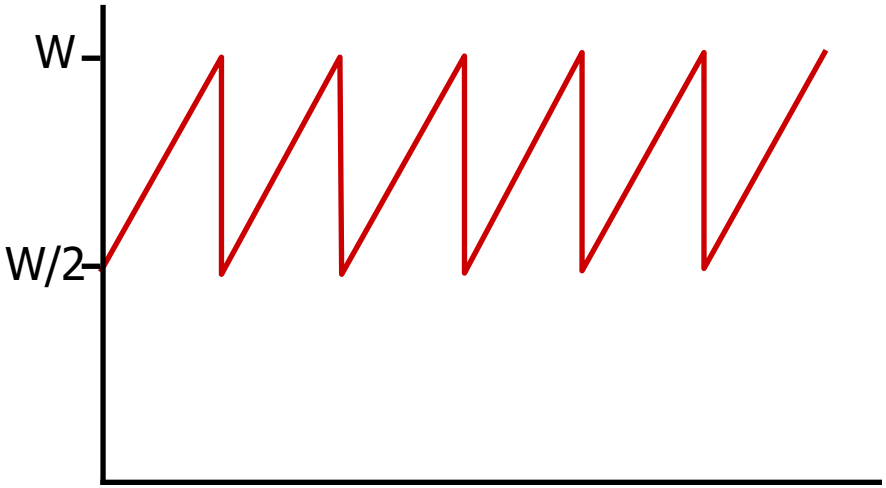
TCP Congestion Control



TCP Average Throughput

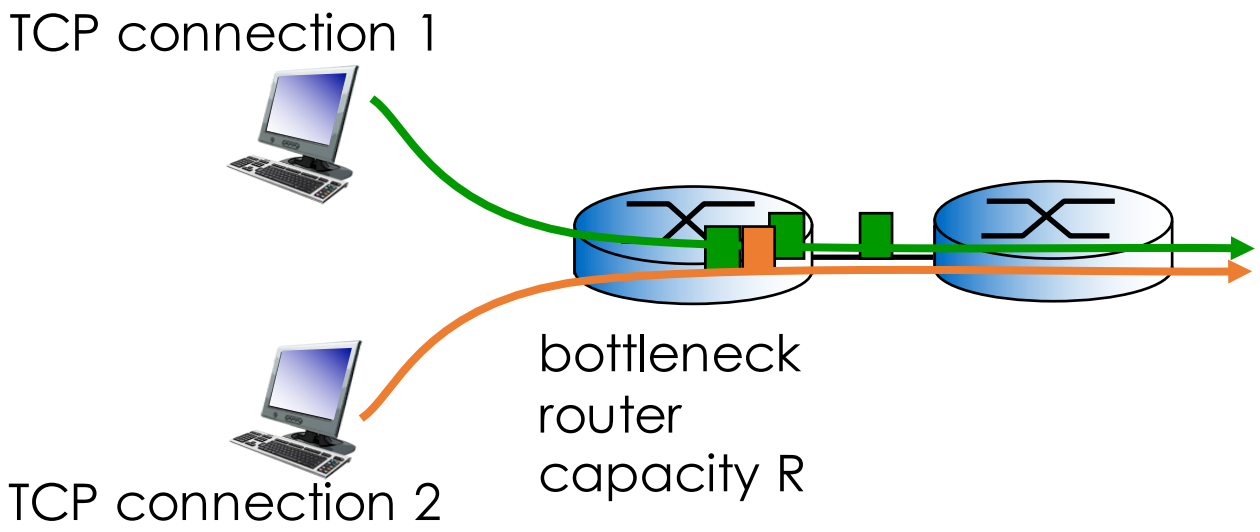
- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Fairness

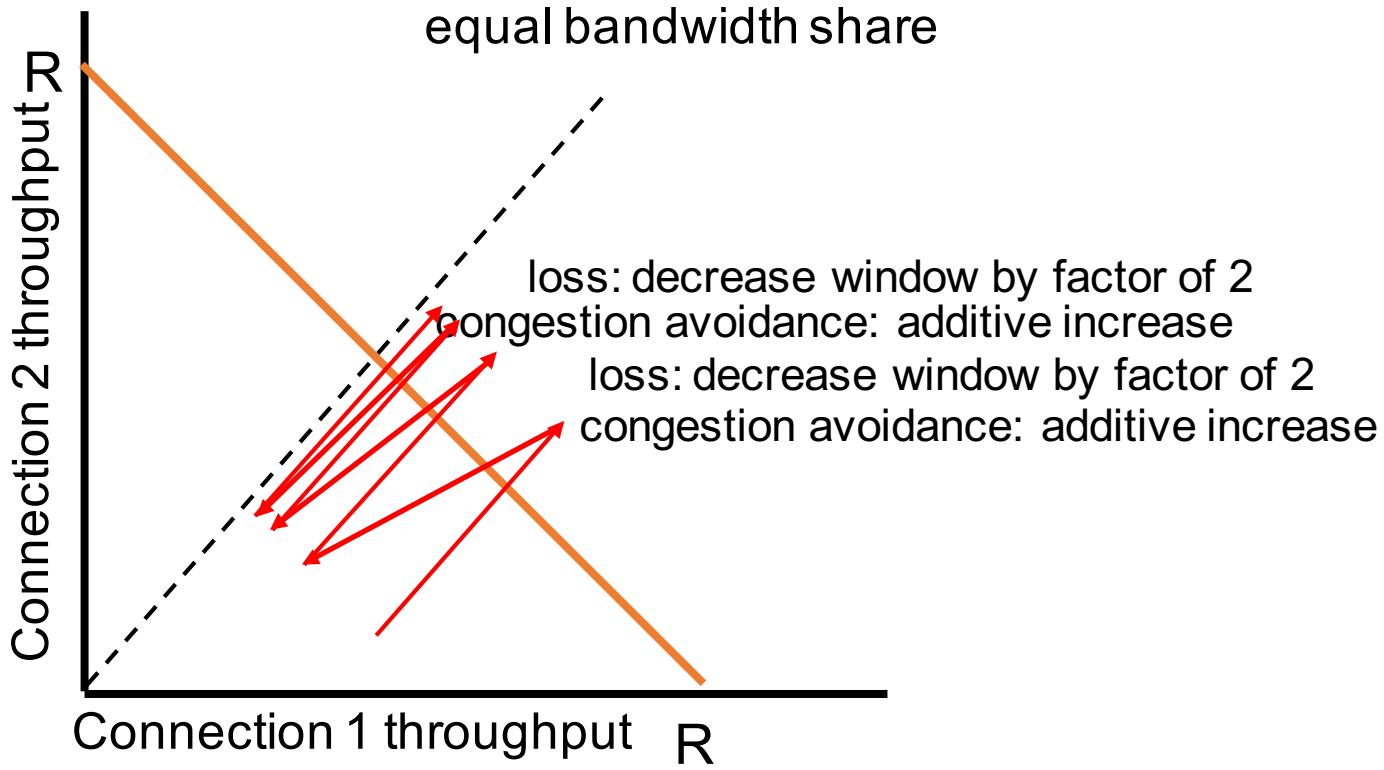
- *fairness goal:* if **K** TCP sessions share same bottleneck link of bandwidth **R**, each should have average rate of **R/K**



Why is TCP Fair?

Simple example: two competing sessions

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



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want the rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- Fairness between TCP and UDP? (later lecture)

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

Summary

- Principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- Instantiation, implementation in the Internet
  - UDP
  - TCP