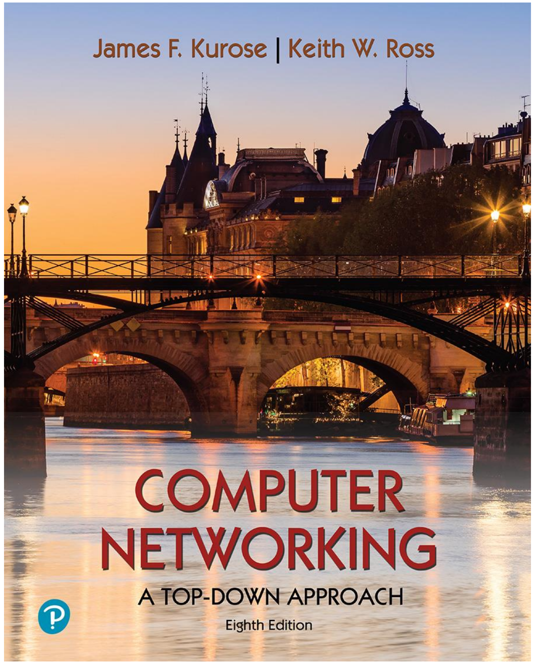
Chapter 3
Transport Layer

A note on the use of these PowerPoint slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: overview

Our goal:

- 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

if ethernet.

MSS = 1460 Bytes

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

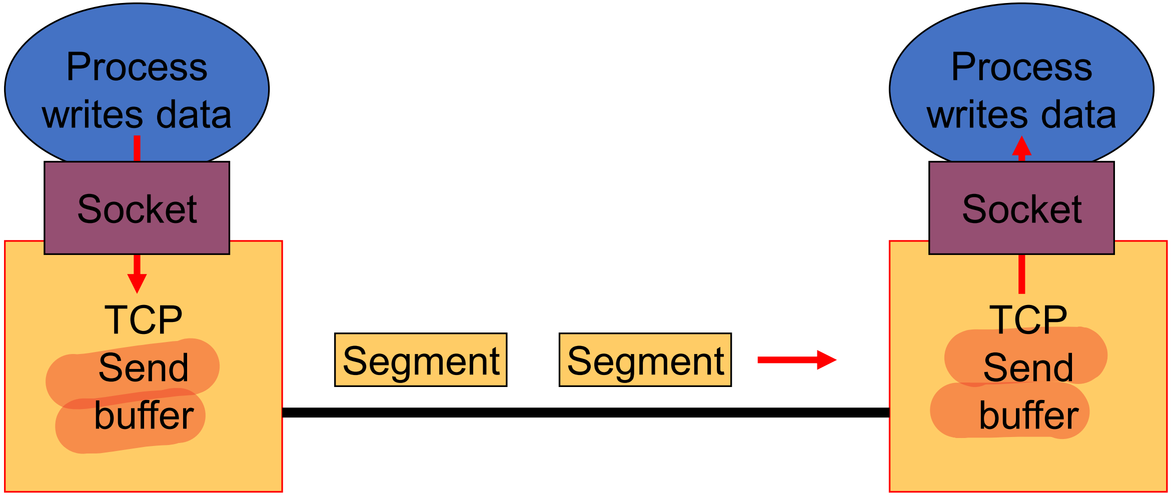
client

request.

"establish a "connection."

. three way handshake

TCP: overview RFCs: 793,1122, 2018, 5681, 7323



- The client process passes data through the socket.
- TCP directs data to the send's buffer. (memory allocate - pick up. receiving station)
- TCP performs three-way handshake. (establish a connection)
- TCP sends data in segments.
- Segment sized is limited by the maximum segment size (MSS).

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

logical point to point connection.

1
- point-to-point:
  - one sender, one receiver

2
- reliable, in-order *byte steam:*
  - no "message boundaries"

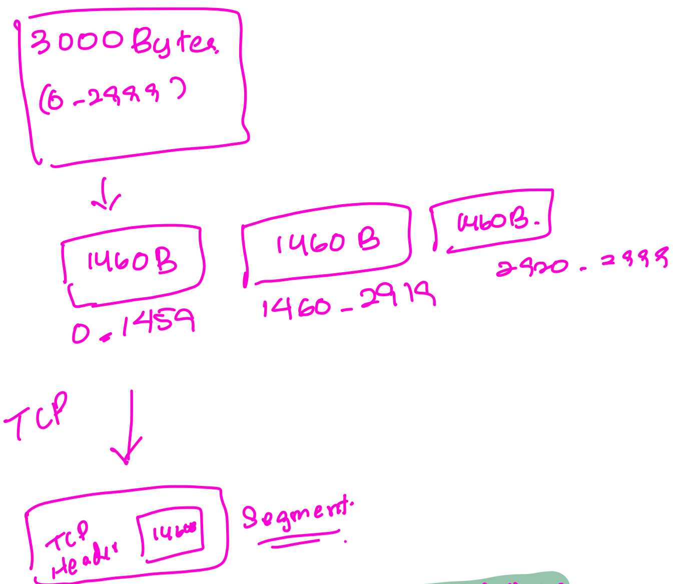
3
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size

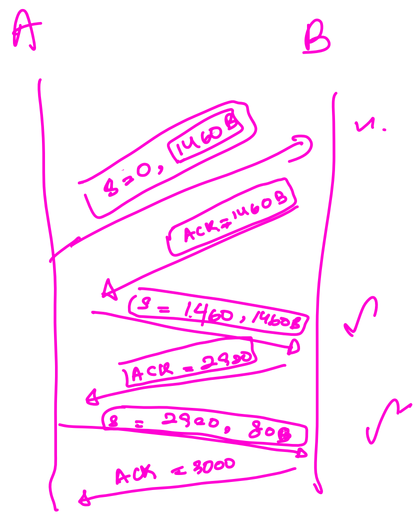
4
- cumulative ACKs

5
- pipelining:
  - TCP congestion and flow control set window size

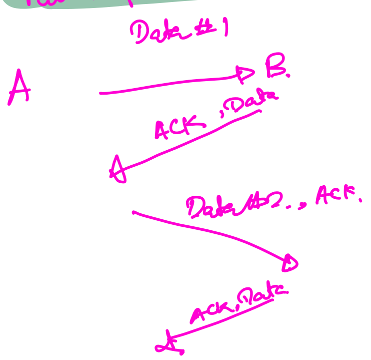
6
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange

7
- flow control:
  - sender will not overwhelm receiver



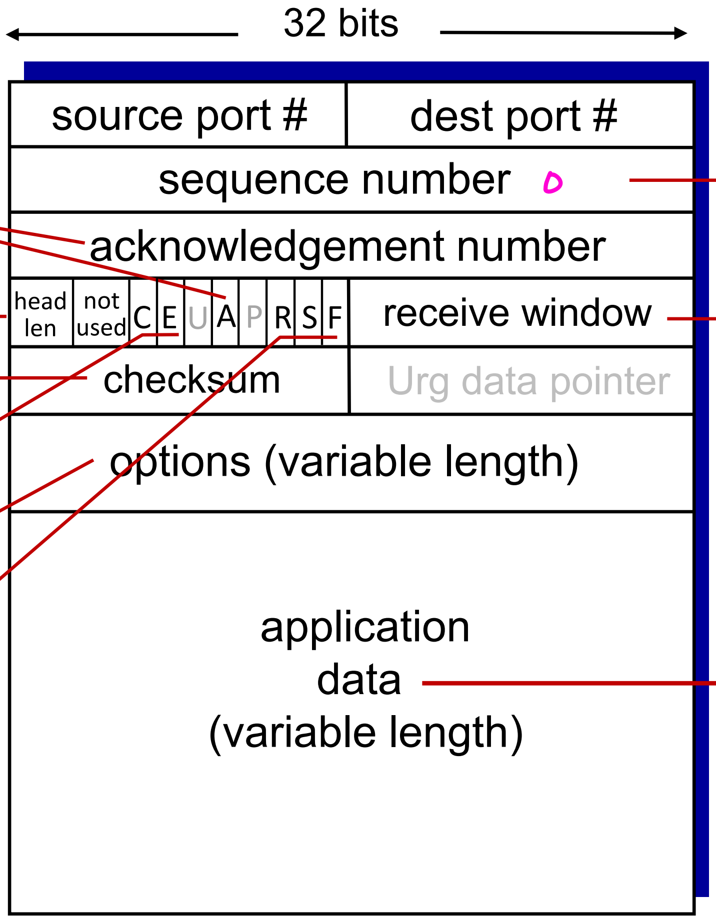


Full Duplex Data (Bidirectional flow)



TCP segment structure

0, 1460, 2920 seq number.
first byte no of the individual bytestream



ACK: seq # of next expected byte; A bit: this is an ACK

1460

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

reset sync finish Flags

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

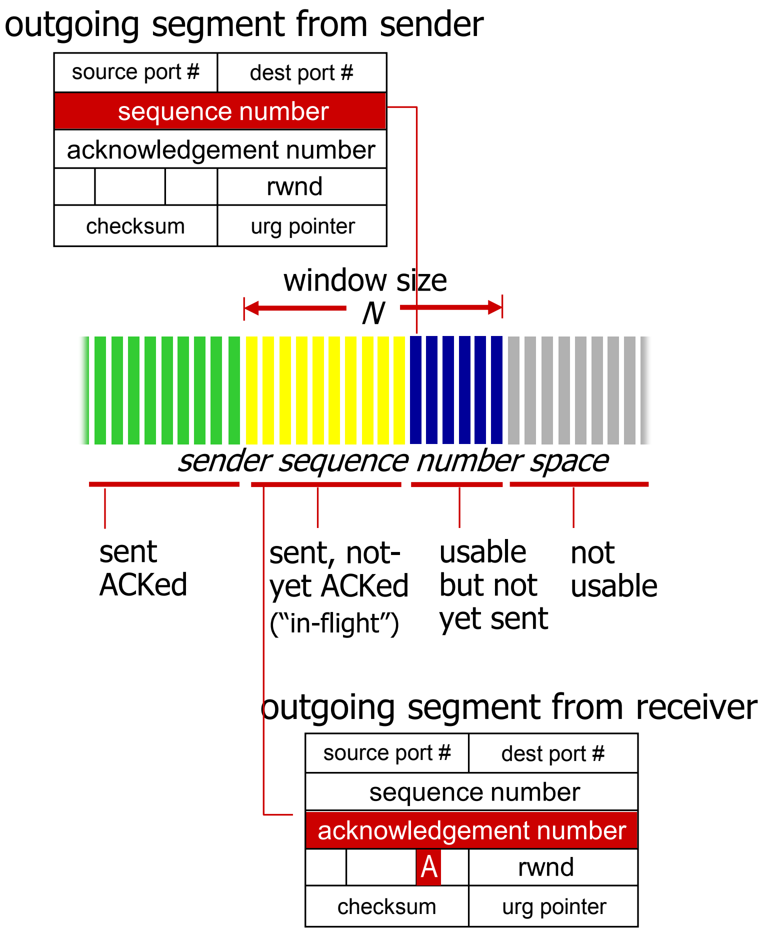
TCP Sequence No. and Acknowledge No.

Sequence numbers:

- TCP views data as unstructured, but ordered stream of bytes.
- Sequence numbers are over bytes, not segments
  - Byte stream number of first byte in segment's data
- Initial sequence number is chosen randomly
- TCP is full duplex – numbering of data is independent in each direction

Acknowledgements:

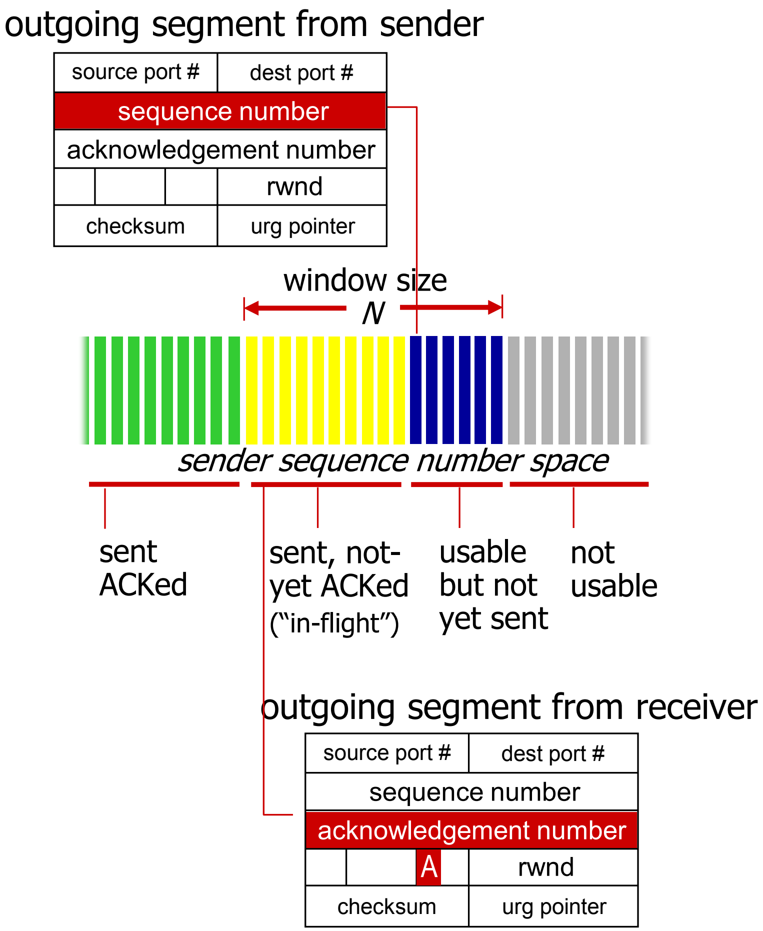
- Acknowledgement number – sequence number of the next byte expected from the sender
- ACKs are cumulative



TCP Sequence No. and Acknowledge No.

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs

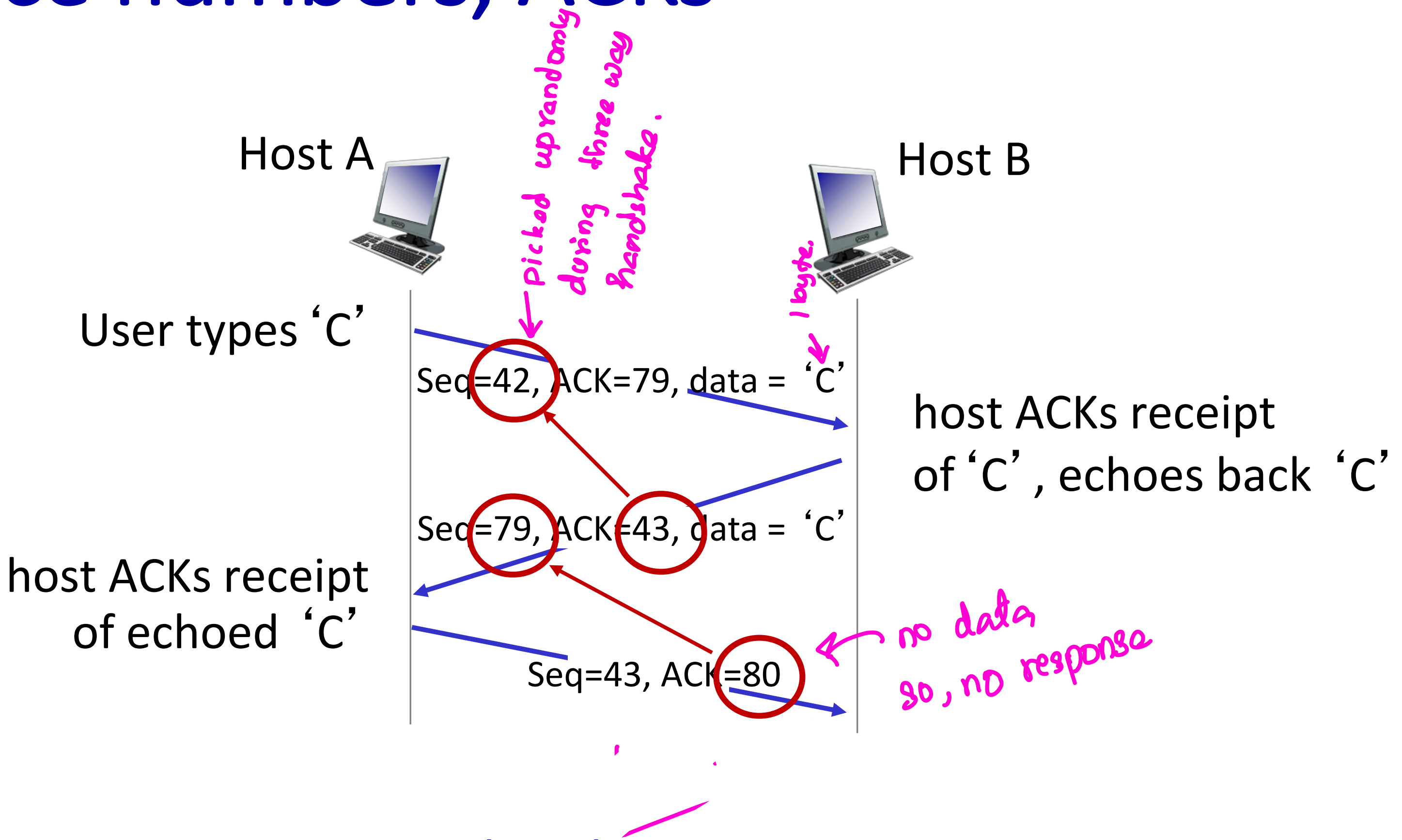
Seq. numbers:

- byte stream "number" of first byte in segment's data

ACKs:

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

Q: how receiver handles out-of-order ?



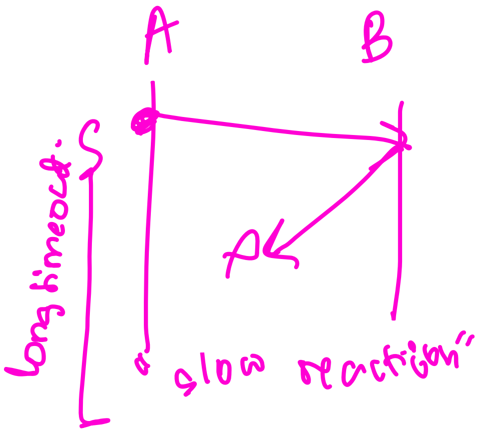
simple telnet scenario

TCP round trip time, timeout

round trip time may vary.

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



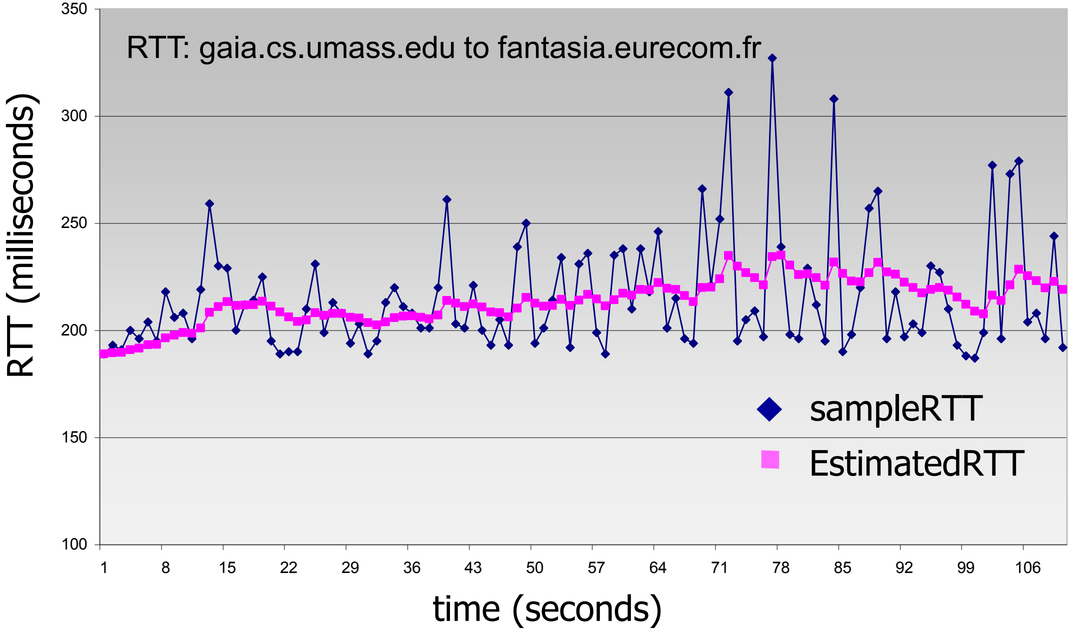
Q: how to estimate RTT?

- `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

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

- <u>e</u>xponential <u>w</u>eighted <u>m</u>oving <u>a</u>verage (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



?

TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`**: want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

(est rt - current rtt?

estimated RTT

"safety margin"

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

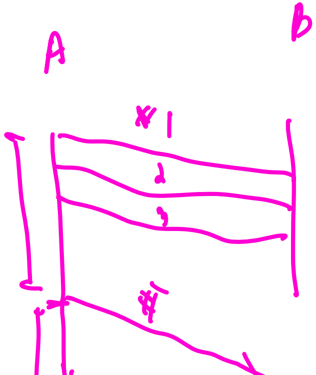
```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

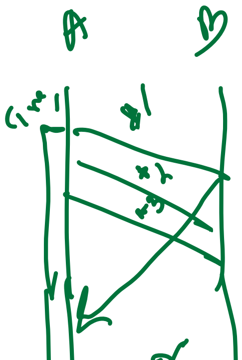
(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP Reliable Data Transfer

- TCP creates **rdt** service on top of IP's unreliable service
  - Pipelined segments (multiple UnAck in pipeline)
  - Cumulative ACKs
  - Single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate ACKs





Initially consider simplified TCP sender:
- ignore duplicate ACKs
- ignore flow control, congestion control

TCP Sender Events (1)

data rcvd from app:

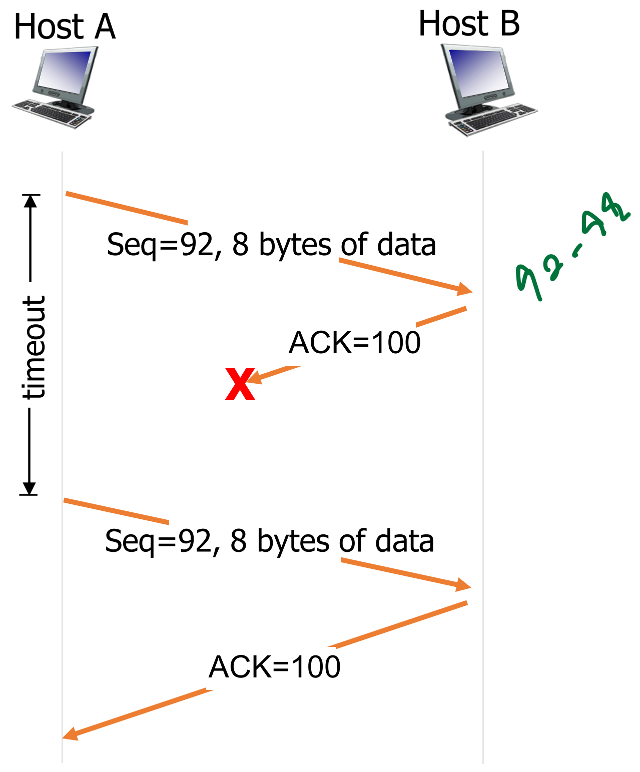
- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- **start timer** if not already running (think of timer as for oldest unacked segment) oldest one trigger
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments



Retransmission due to a lost acknowledgement

TCP Sender Events (2)

data rcvd from app:

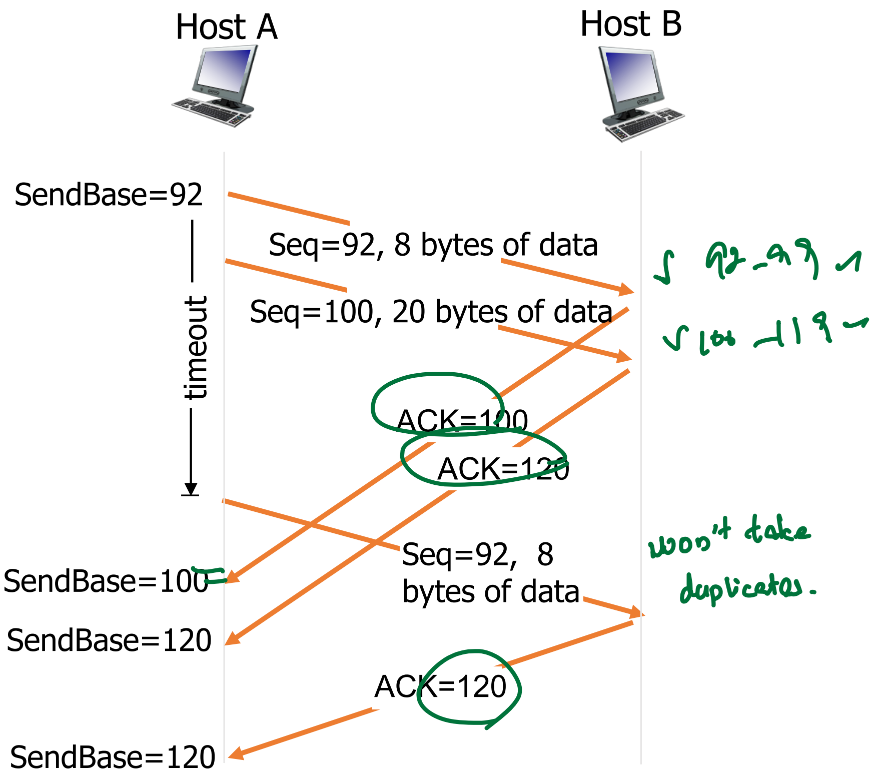
- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- **start timer** if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments



premature timeout & cumulative ACK

TCP Sender Events (3)

data rcvd from app:

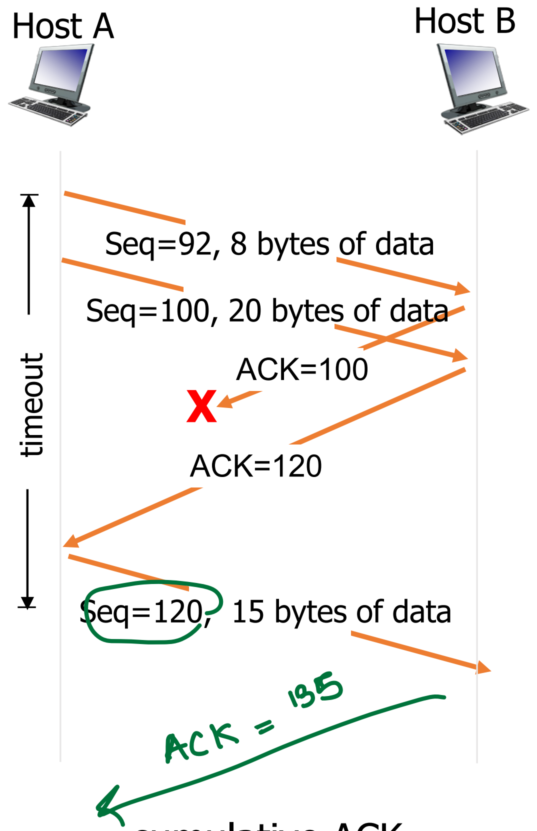
- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- **start timer** if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

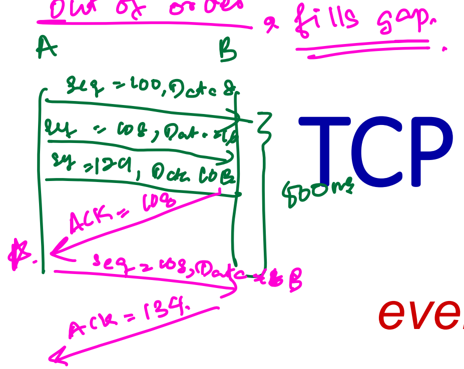
- retransmit segment that caused timeout
- restart timer

ack rcvd:

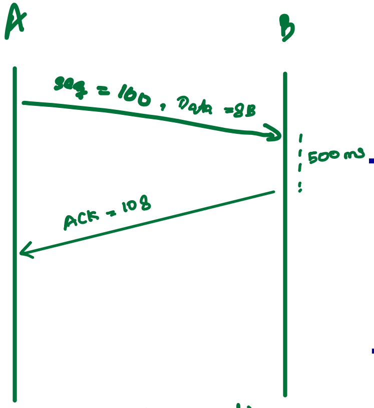
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments



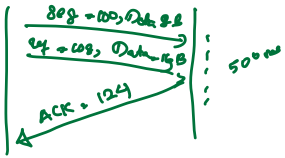
cumulative ACK
avoids retransmission of the first segment



TCP ACK Generation [RFC 1122, 2581, 5681, 7323]



received when pending



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

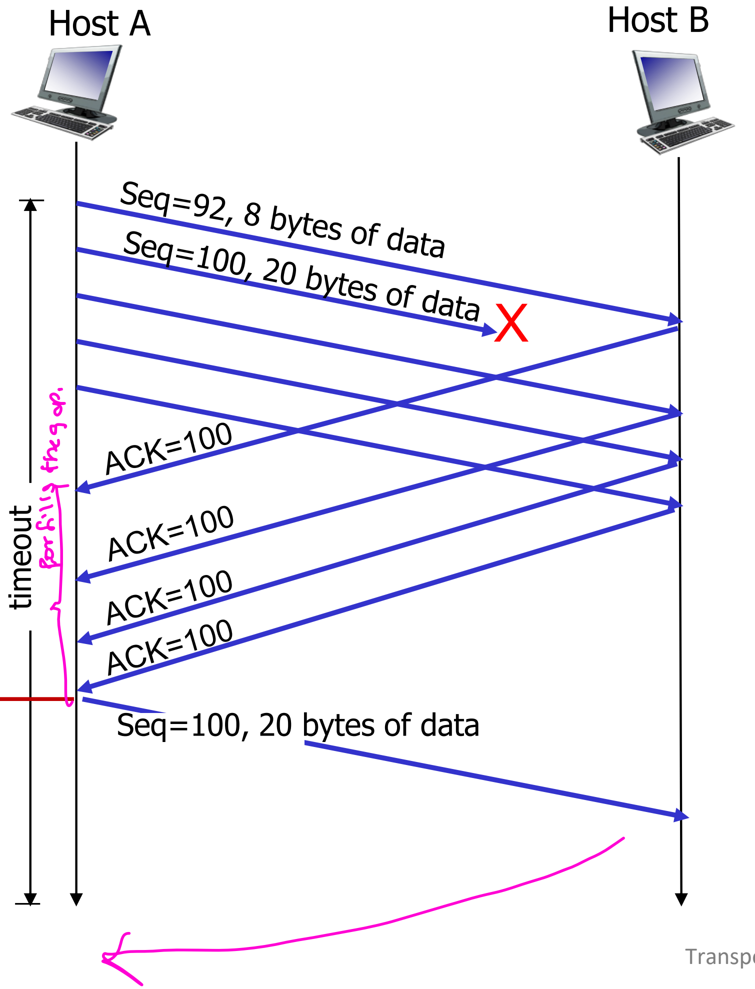
TCP fast retransmit

TCP fast retransmit

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

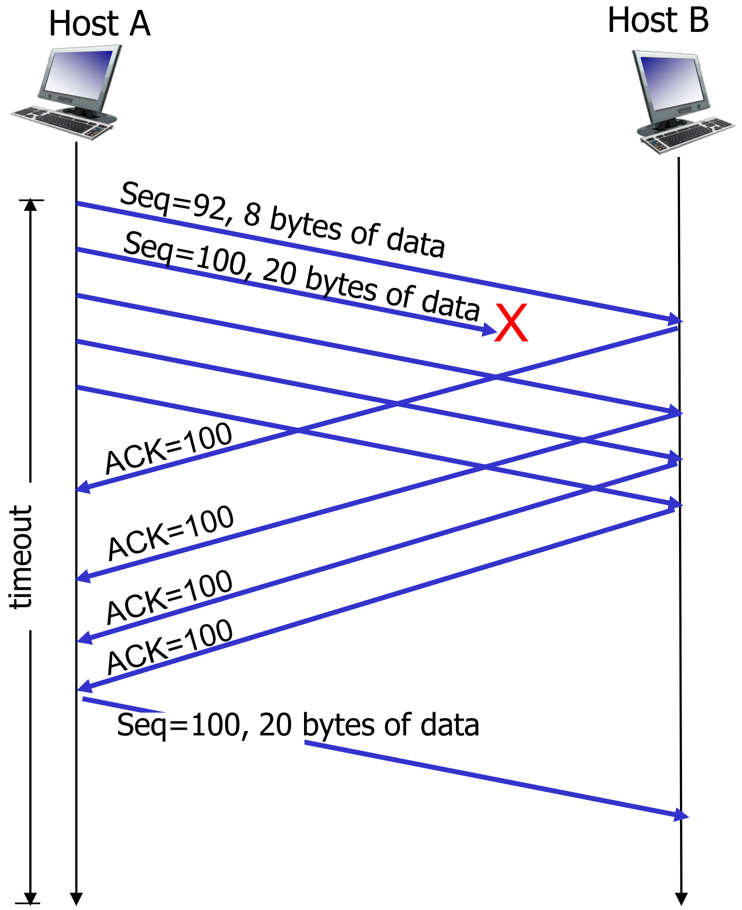
- likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



TCP fast retransmit

- Since the timeout interval is exponentially increased, increasing end-to-end delay.
  - Long delay before retransmission*
- Fortunately, the sender can often detect packet loss well before the timer expires by just **duplicate ACKs**.
  - sender often sends many segments back-to-back (send many segments one after another)
  - if one segment is lost, there will likely be many **duplicate ACKs**.
- If the TCP sender receives **three duplicate ACKs** for the same data, TCP performs a **fast retransmit**, retransmitting the missing segment before the timer expires.

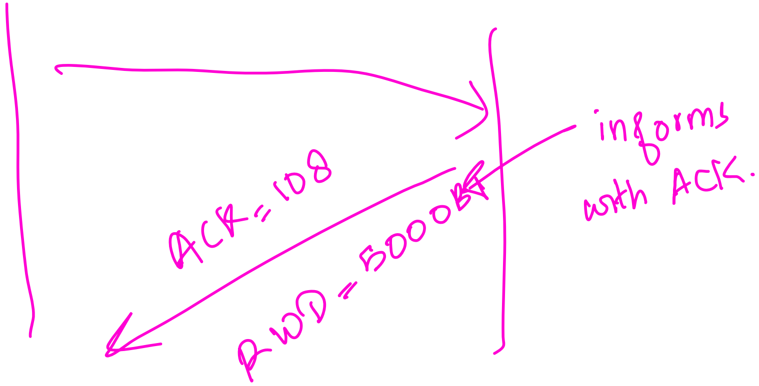


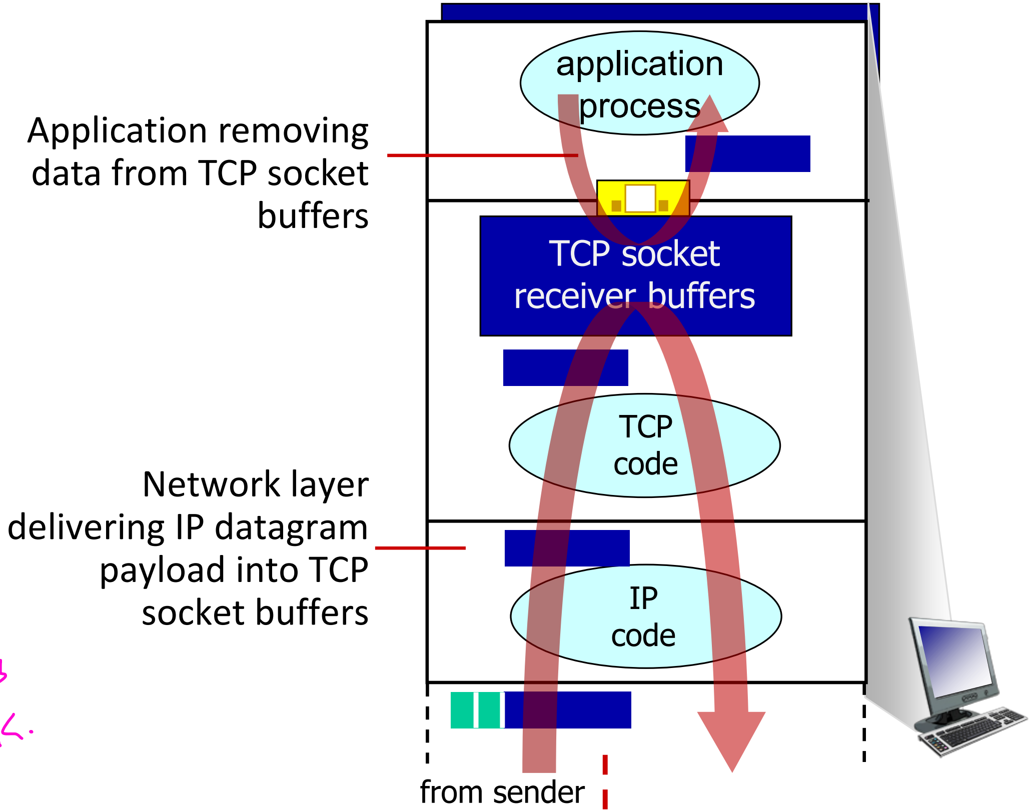
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

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

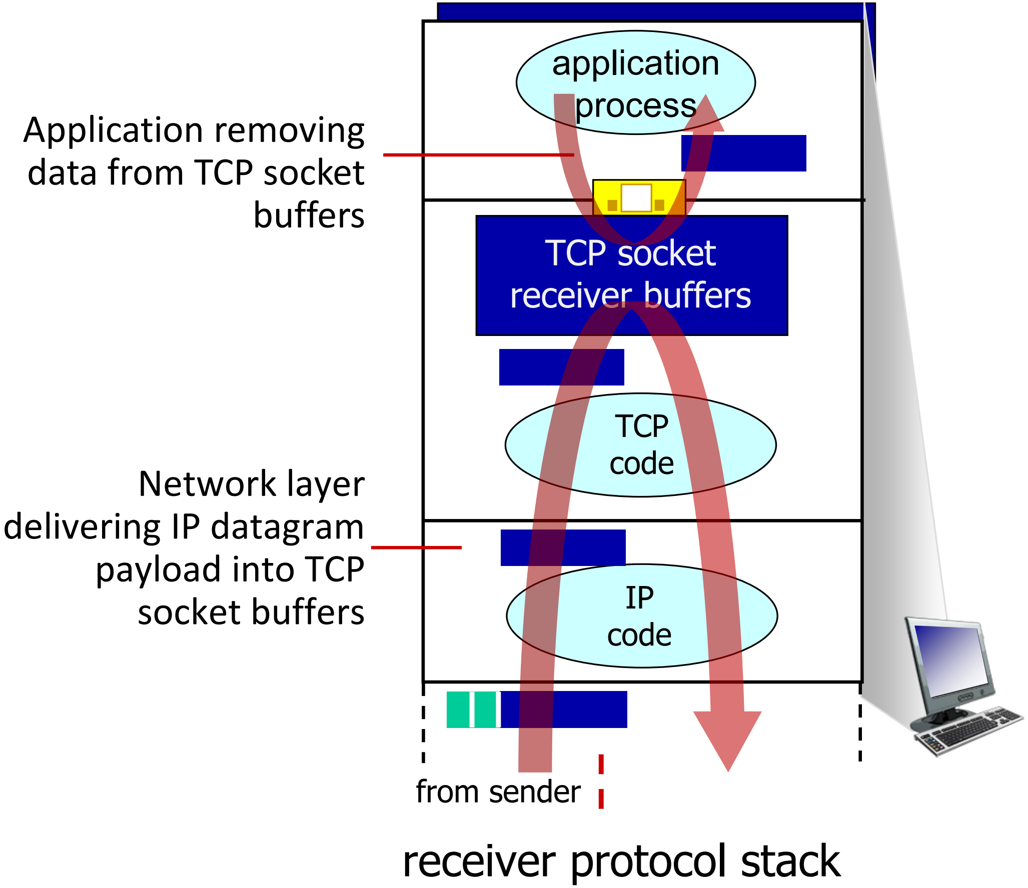




receiver protocol stack

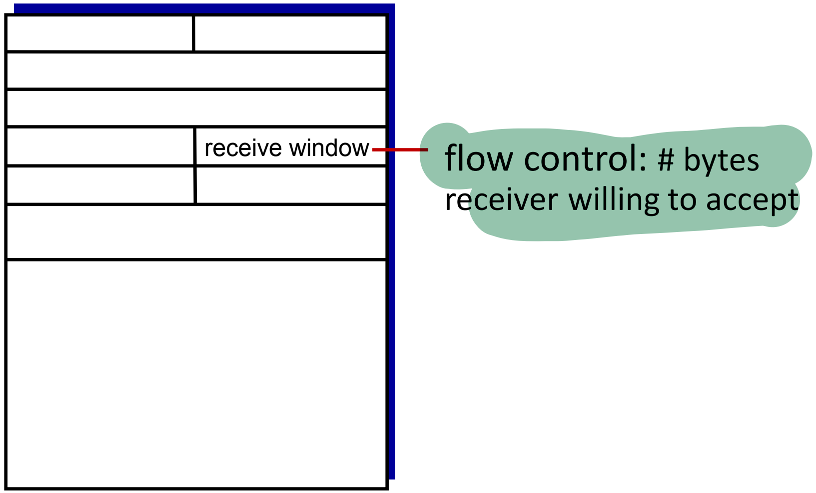
TCP flow control

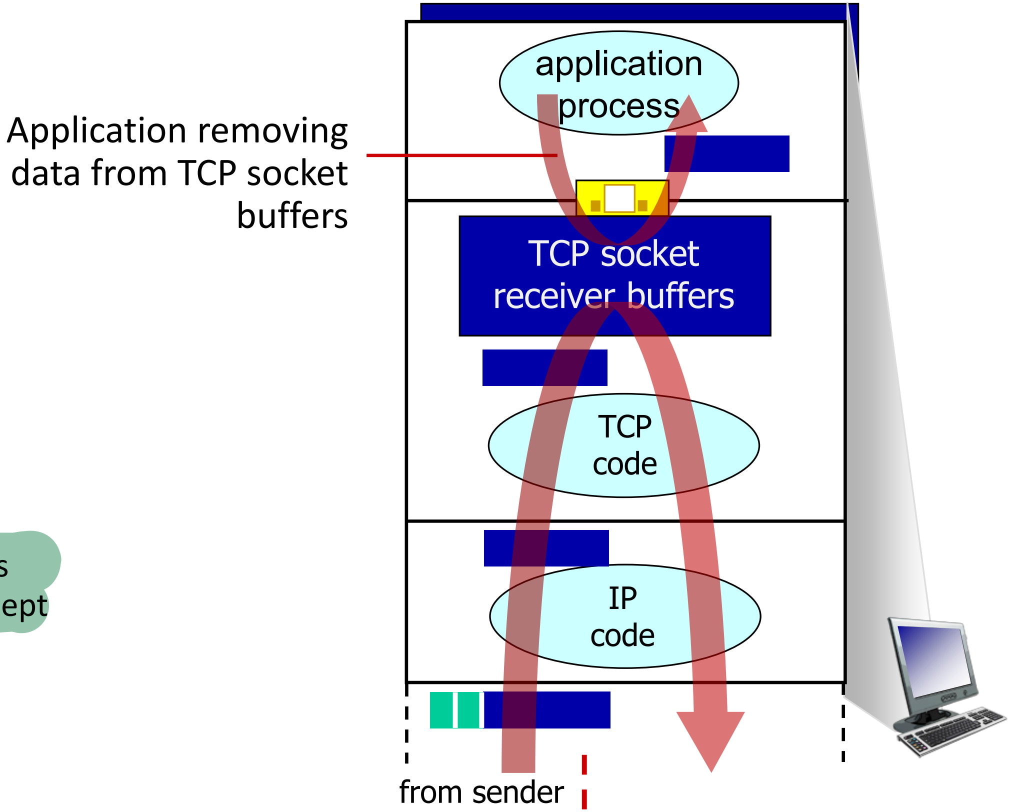
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers (receive buffers)?



TCP flow control

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





receiver protocol stack

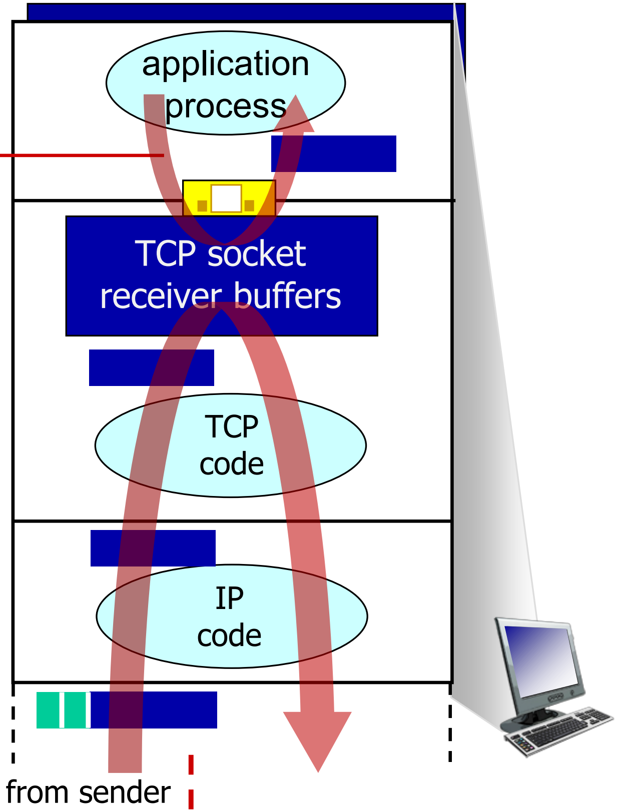
TCP flow control

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

flow control

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

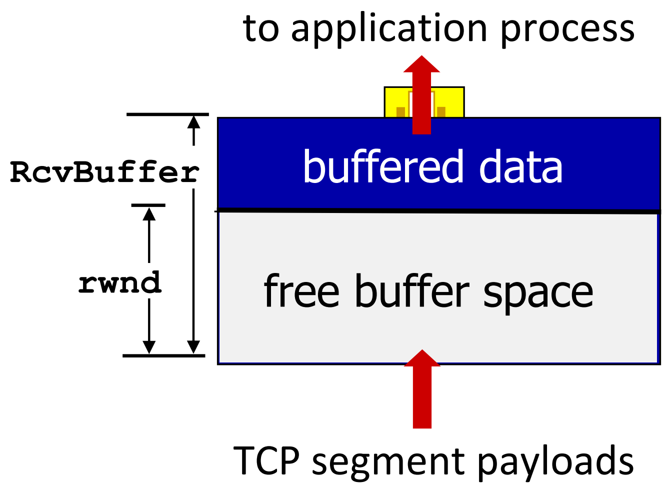
Application removing data from TCP socket buffers



receiver protocol stack

TCP flow control

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

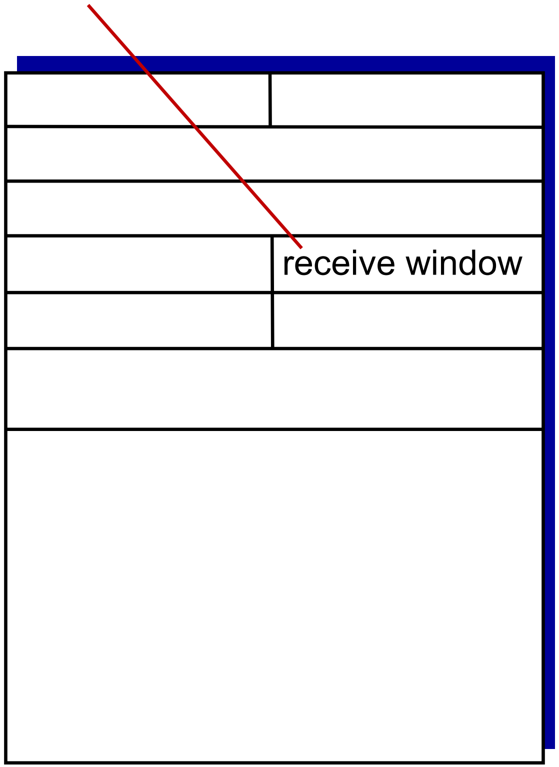


TCP receiver-side buffering

TCP flow control

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

flow control: # bytes receiver willing to accept

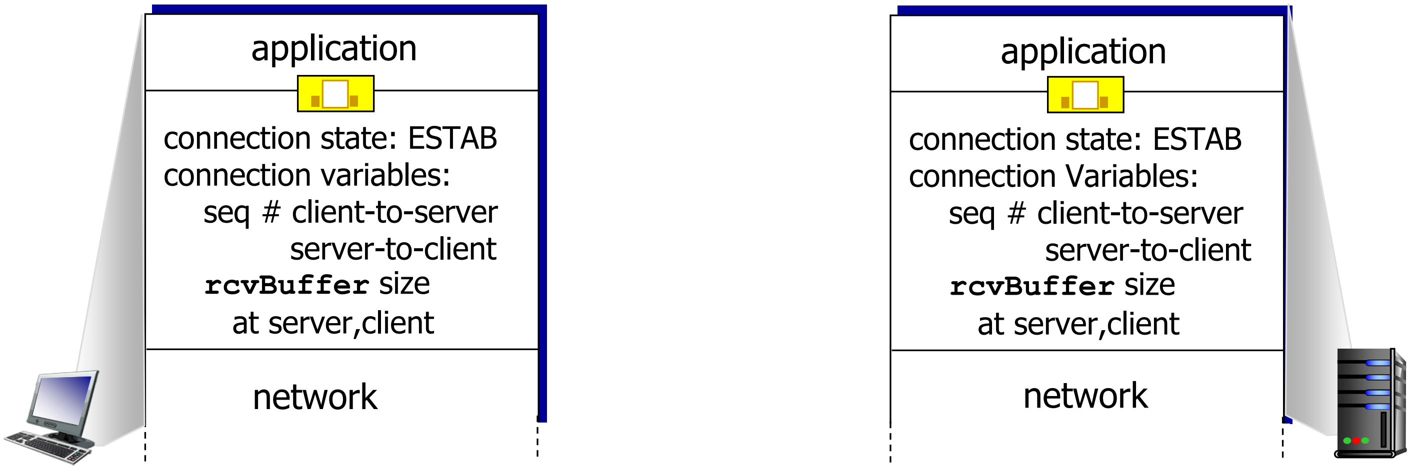


TCP segment format

TCP connection management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (**starting seq #** and **`rwnd`**)

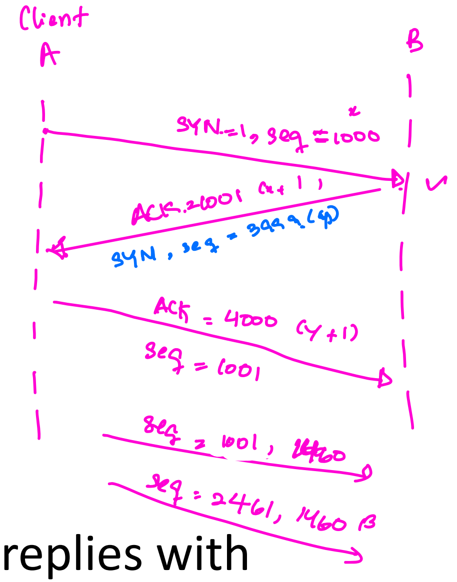



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

(review!

TCP Connection Management



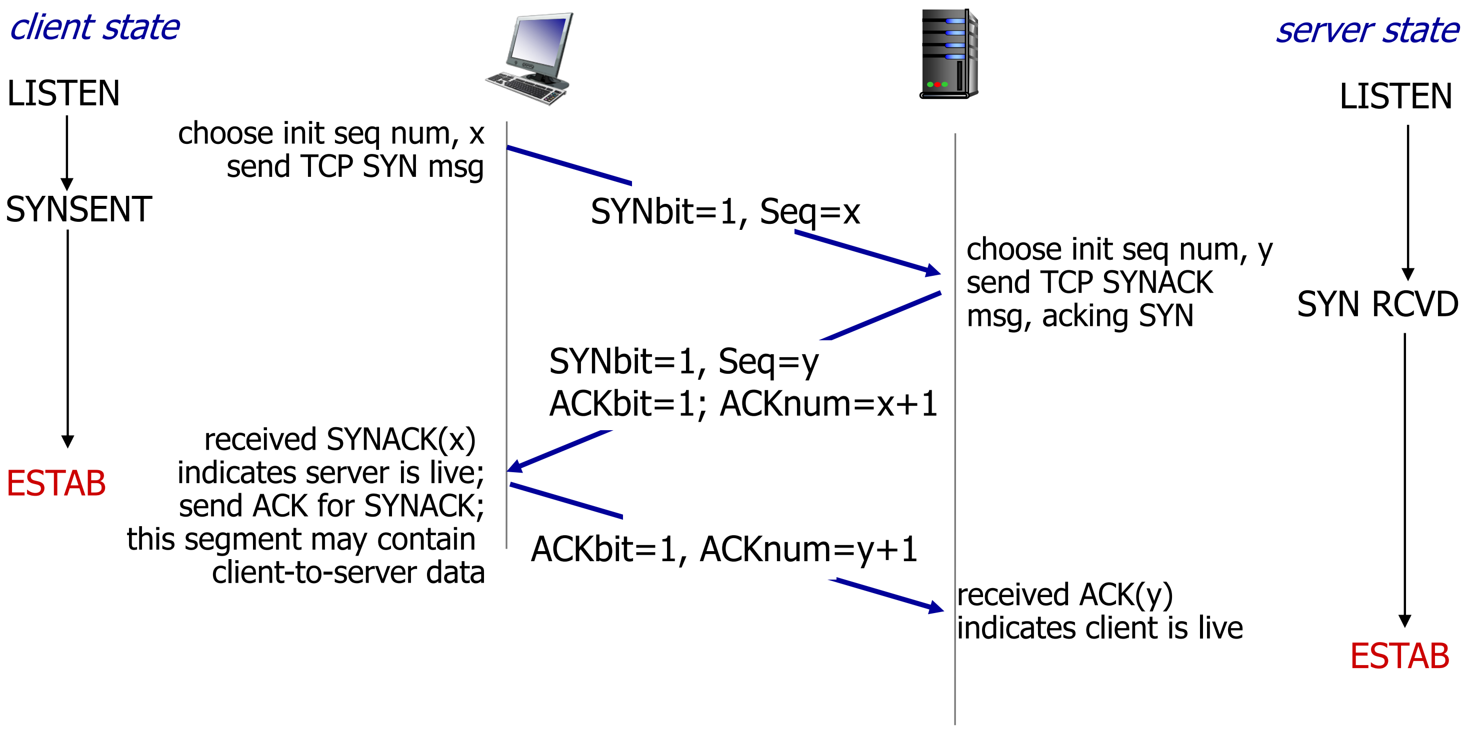
3-Way Handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, (if want to communicate) replies with SYN/ACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYN/ACK, replies with ACK segment, which may contain data

TCP 3-Way Handshake

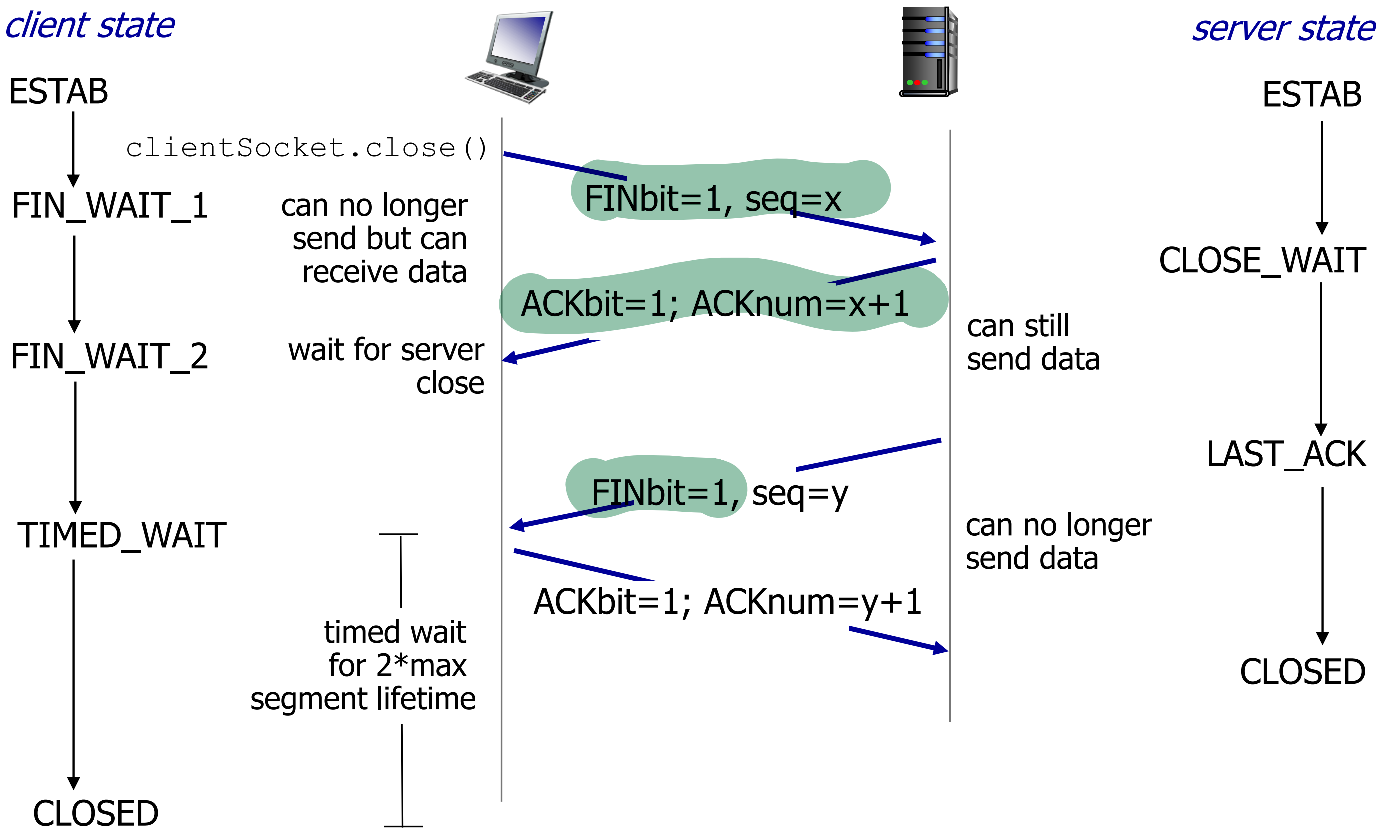


A human 3-way handshake protocol

(review!

TCP Connection Termination

- client, server each close their side of connection
- send TCP segment with FIN bit = 1
- respond to received FIN with ACK
- on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled



Chapter 3: roadmap

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

(slows down sending rate of the sender?)

Principles of congestion control

observes network condition.

Congestion:

- informally: "too many sources sending too much data too fast for ***network*** to handle"
- manifestations: buffer full, bandwidth full
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!

~~one sender and not overflow its receiver~~

priority queue → not priority throw away.
new comers → tail box queue
queuing policy.

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

TCP Congestion Control: Overview

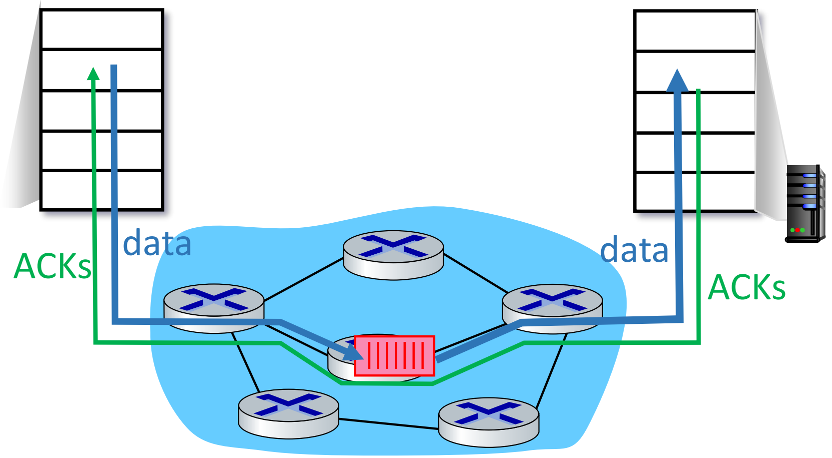
- TCP uses end-to-end congestion control.
- It limits the sender's sending rate.
- If the sender perceives that there is **little (no) congestion** on the path, the TCP sender **increases its send rate**.
- If the sender perceives that there is **congestion** on the path, the TCP sender **reduces its send rate**.

main actors → router

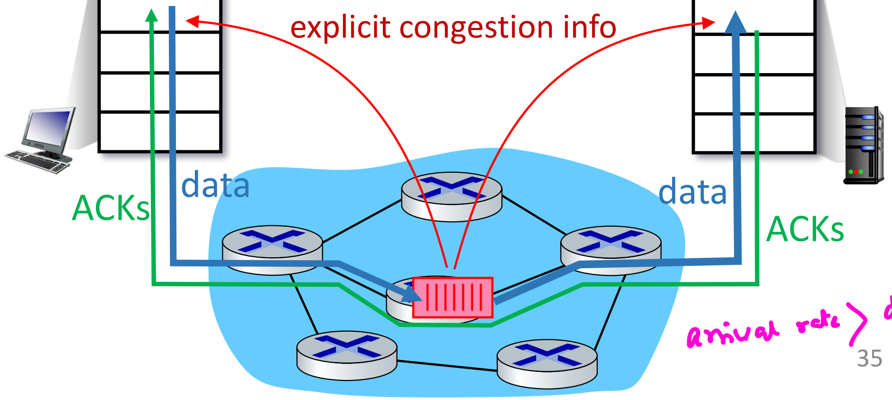
→

Congestion Control: Approaches

- Goal: Throttle senders as needed to ensure load on the network is "reasonable"
- 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
  - explicit rate sender should send at
  - TCP ECN, ATM, DECbit protocols



if packet is discarded.



arrival rate > departure rate.

Chapter 3: roadmap

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

TCP congestion control: AIMD

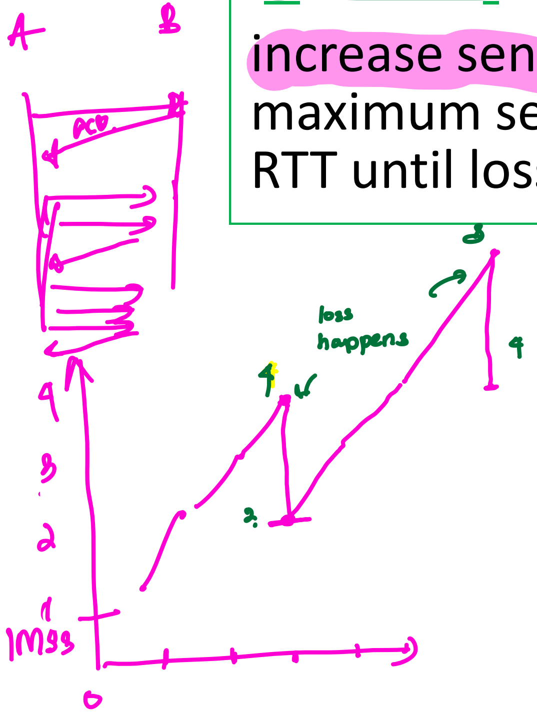
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

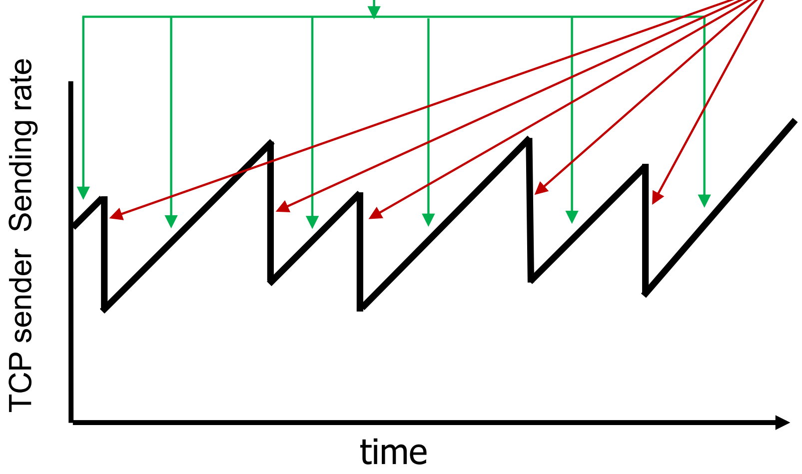
Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event





AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

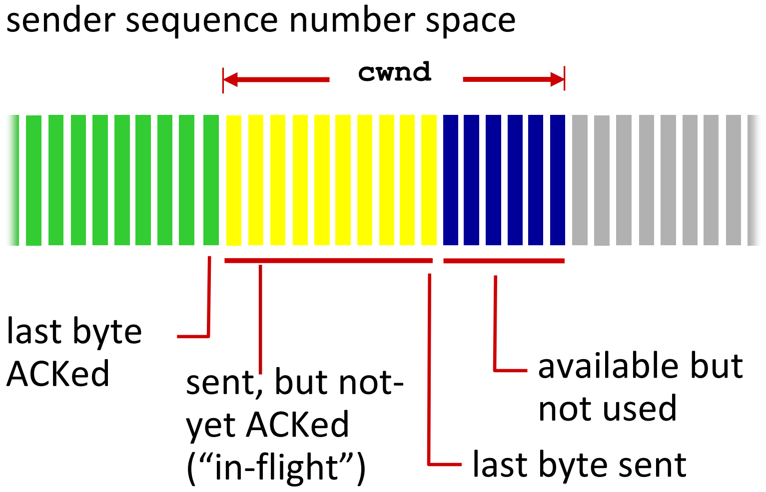
Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ~~ACK (TCP~~ Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (~~TCP Tahoe~~)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details



TCP sending behavior:

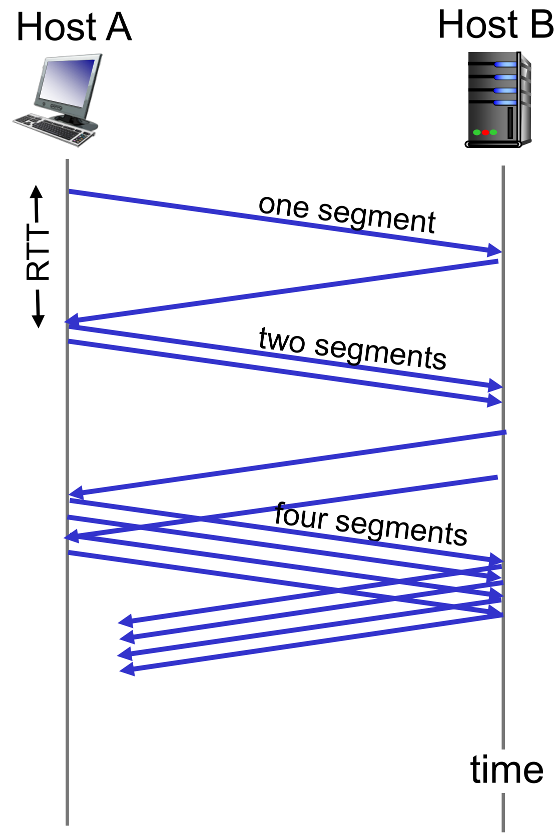
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

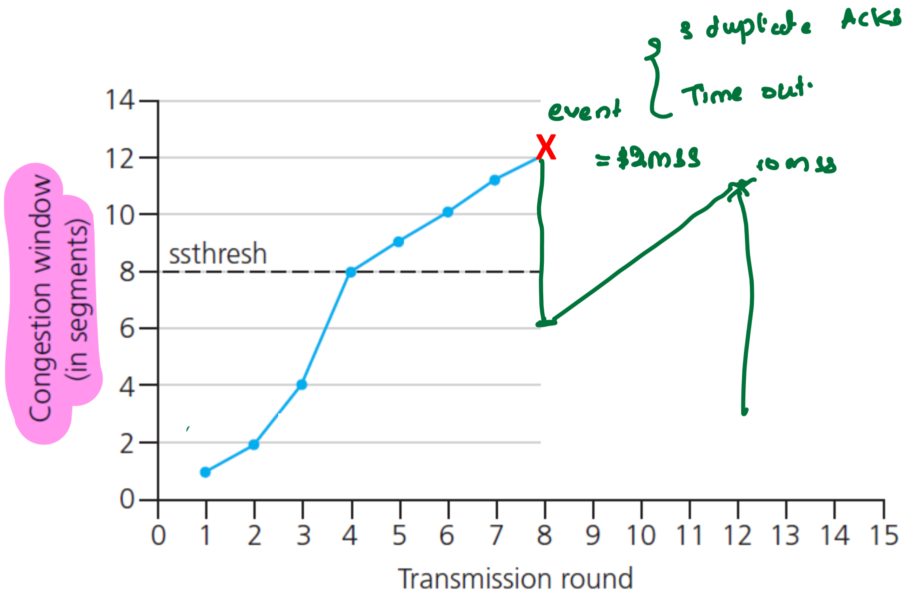
- when connection begins, increase rate **exponentially** until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary*: initial rate is slow, but ramps up exponentially fast



TCP: from Slow Start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

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

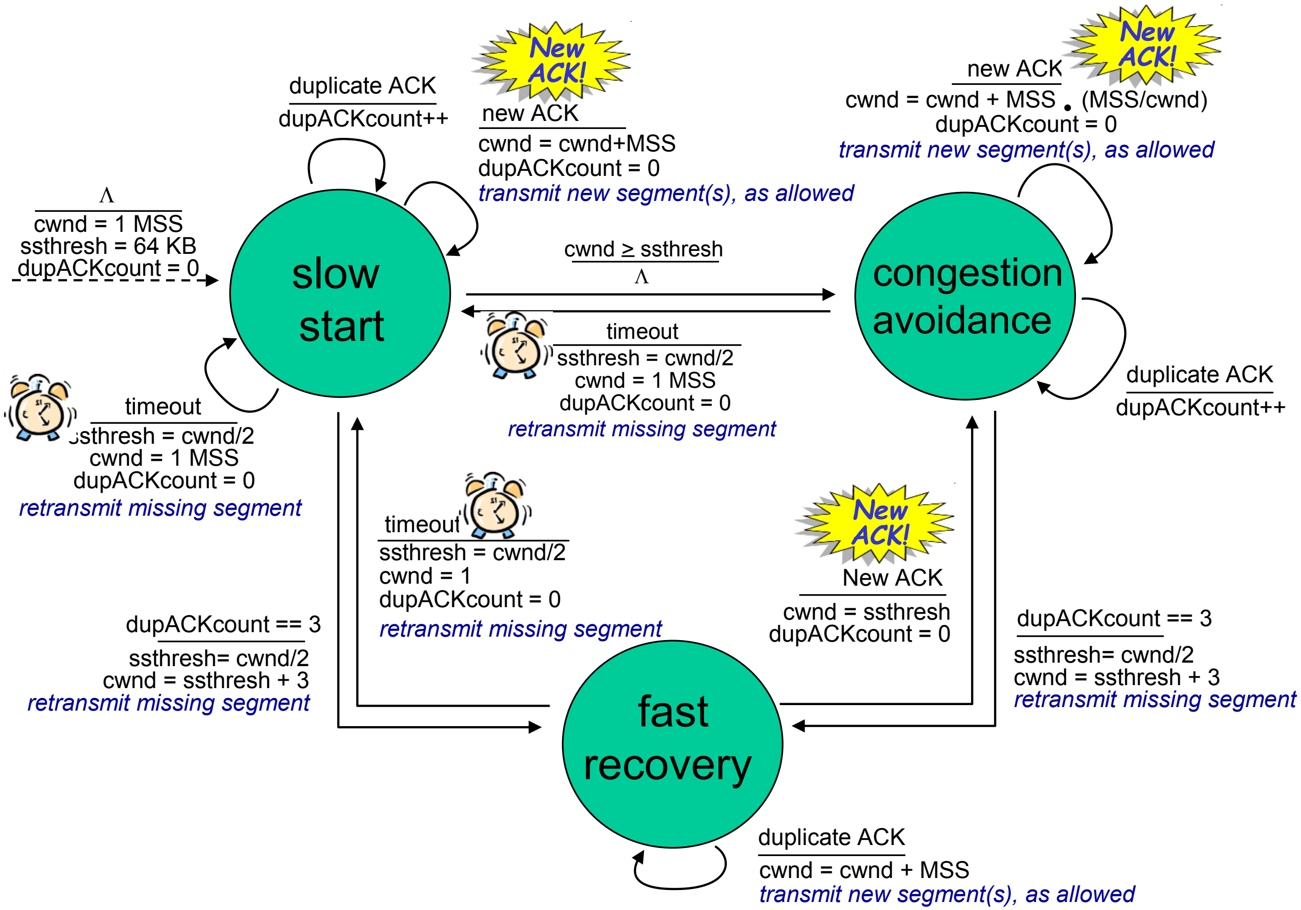


Implementation:

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

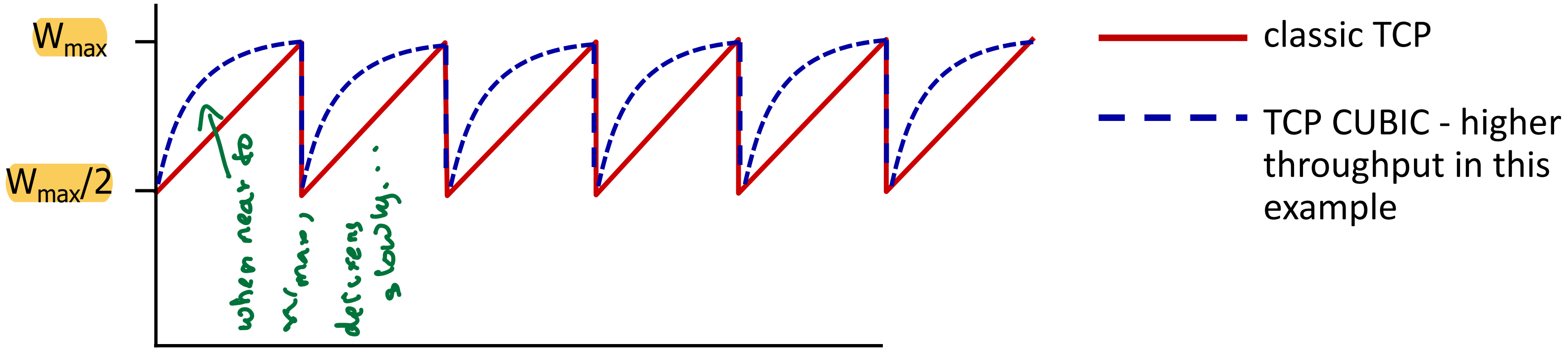
Summary: TCP congestion control



TCP CUBIC

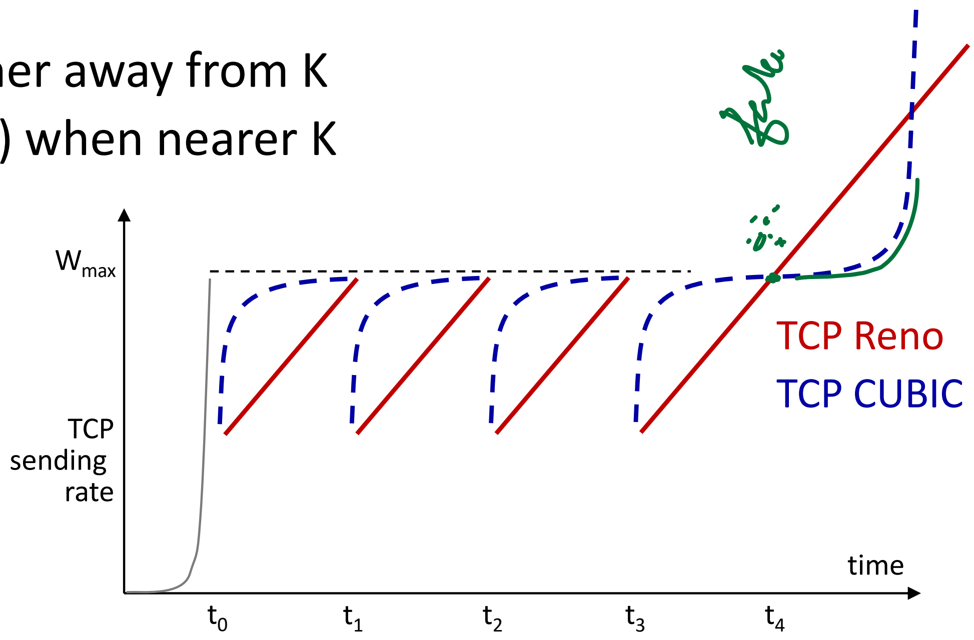
- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*

congestion occurs



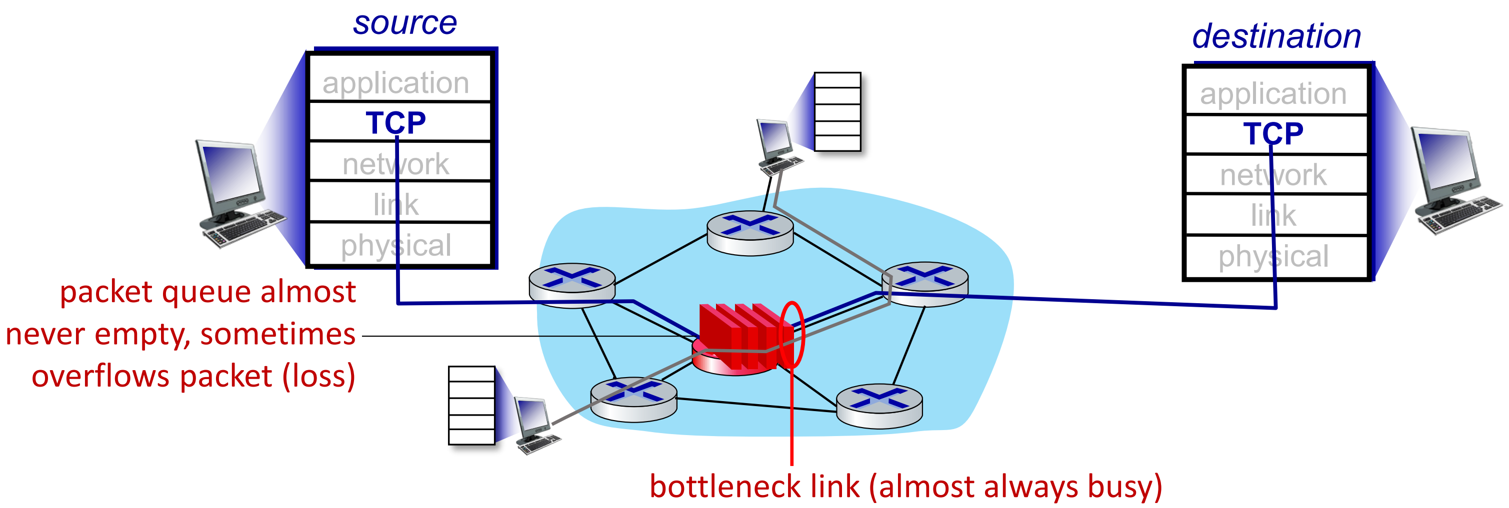
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tunable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



TCP and the congested "bottleneck link"

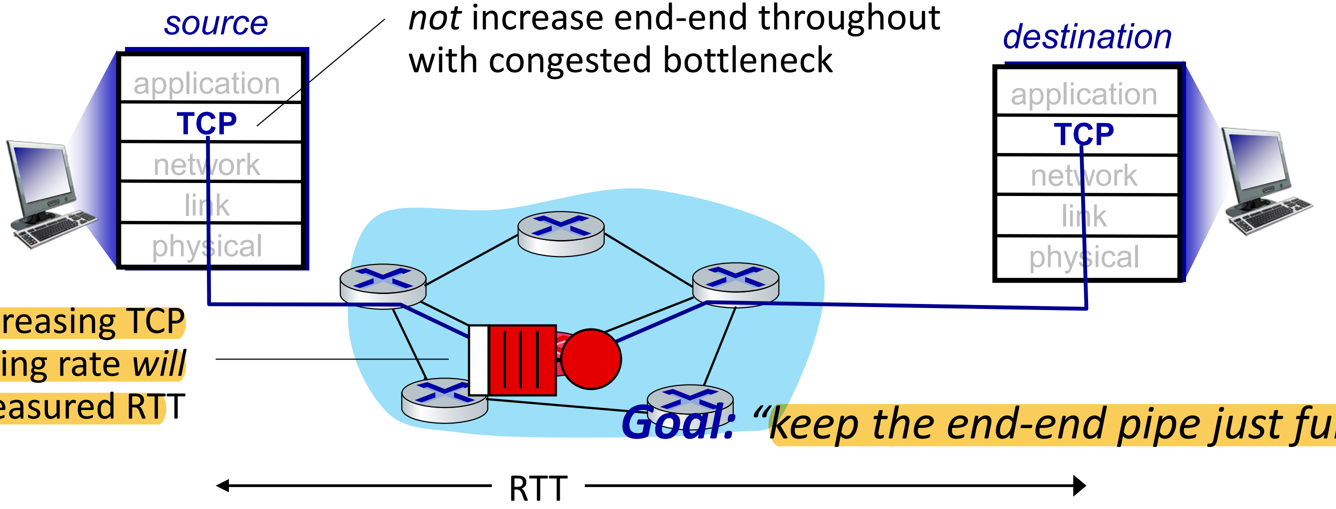
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



TCP and the congested "bottleneck link"

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

insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck



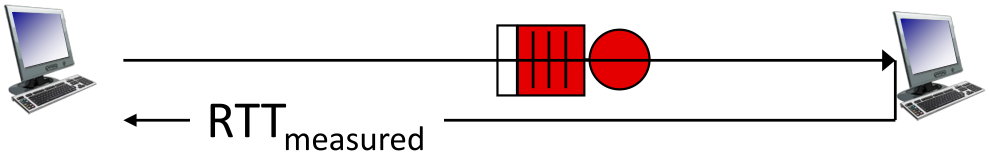
insight: increasing TCP sending rate *will* increase measured RTT

Goal: *"keep the end-end pipe just full, but not fuller"*

Don't wait until loss, Act when delay happens.

Delay-based TCP congestion control

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



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

roundtrip

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

if measured throughput "very close" to uncongested throughput
    increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease `cwnd` linearly /* since path is congested */

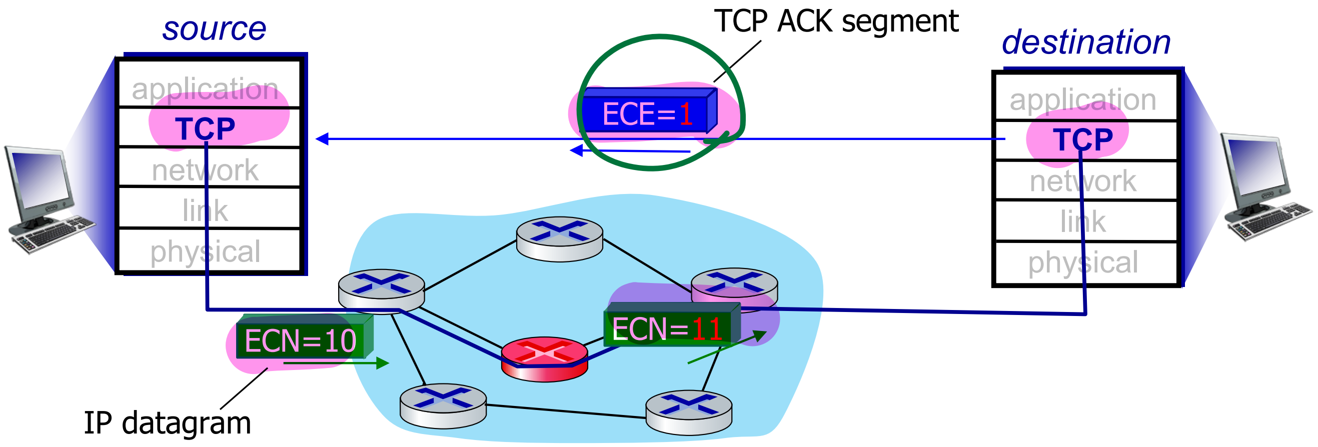
Delay-based TCP congestion control

- **congestion control without inducing/forcing loss**
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

Explicit congestion notification (ECN)

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

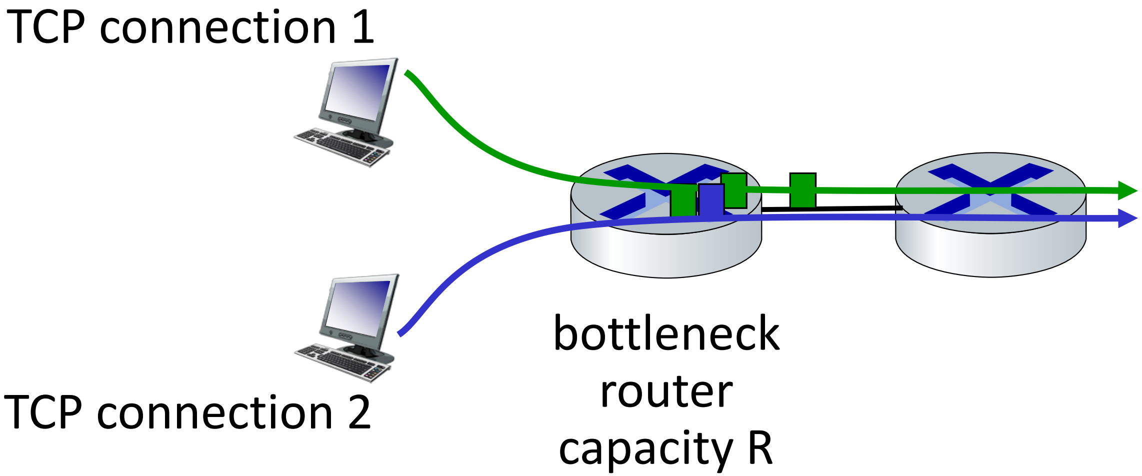
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN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



TCP fairness

no fairness between different TCP connections

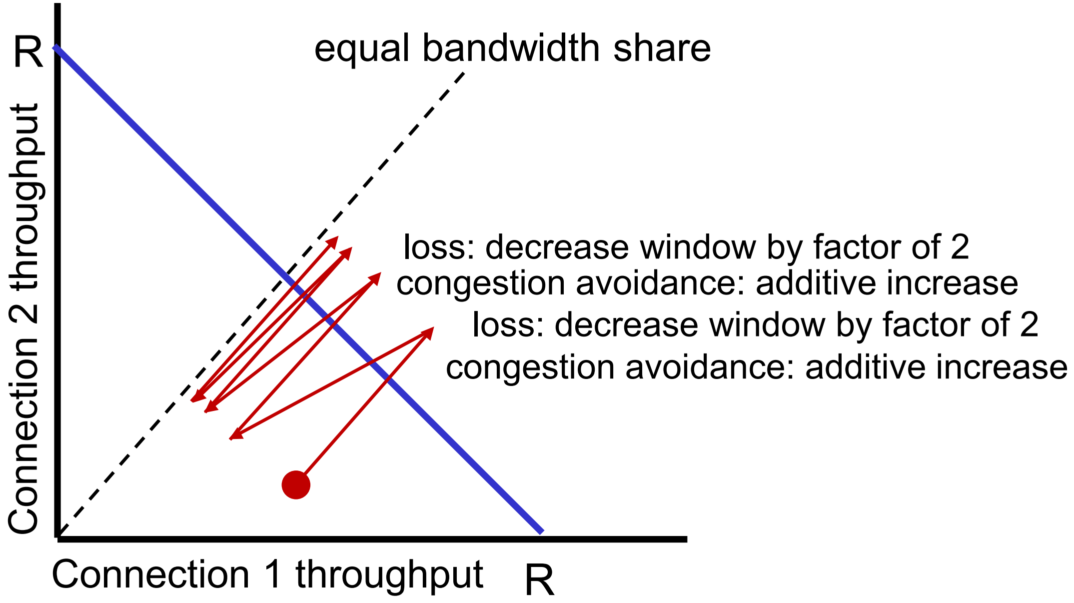
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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



Is TCP fair?

A: Yes, under idealized assumptions:

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

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

(Doesn't

→ UDP has more benefit over network.

Transport layer: roadmap

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

Evolving transport-layer functionality

- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

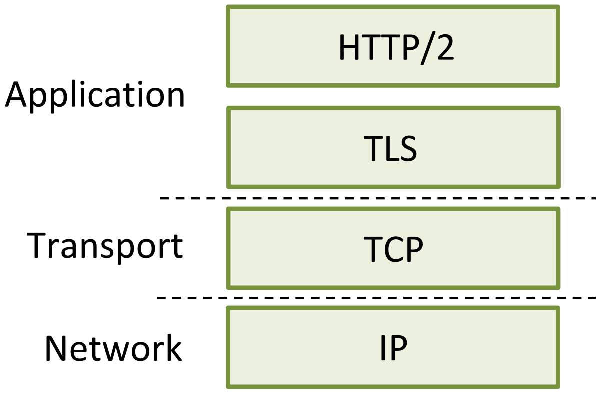
Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
    - HTTP/2 & HTTP/3: QUIC

research topic.

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

Chapter 3: summary

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

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane