Chapter 3: Transport Layer
Part B

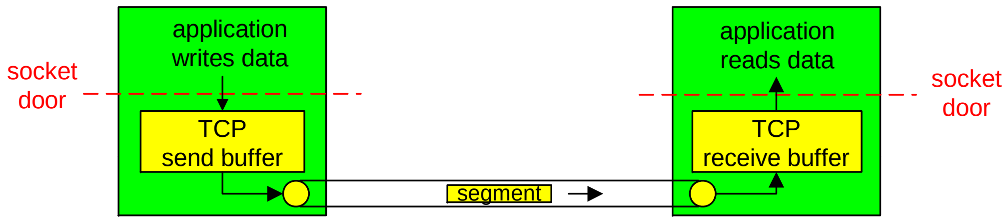
Course on Computer Communication and Networks, CTH/GU

The slides are adaptation of the slides made available by the authors of the course's main textbook

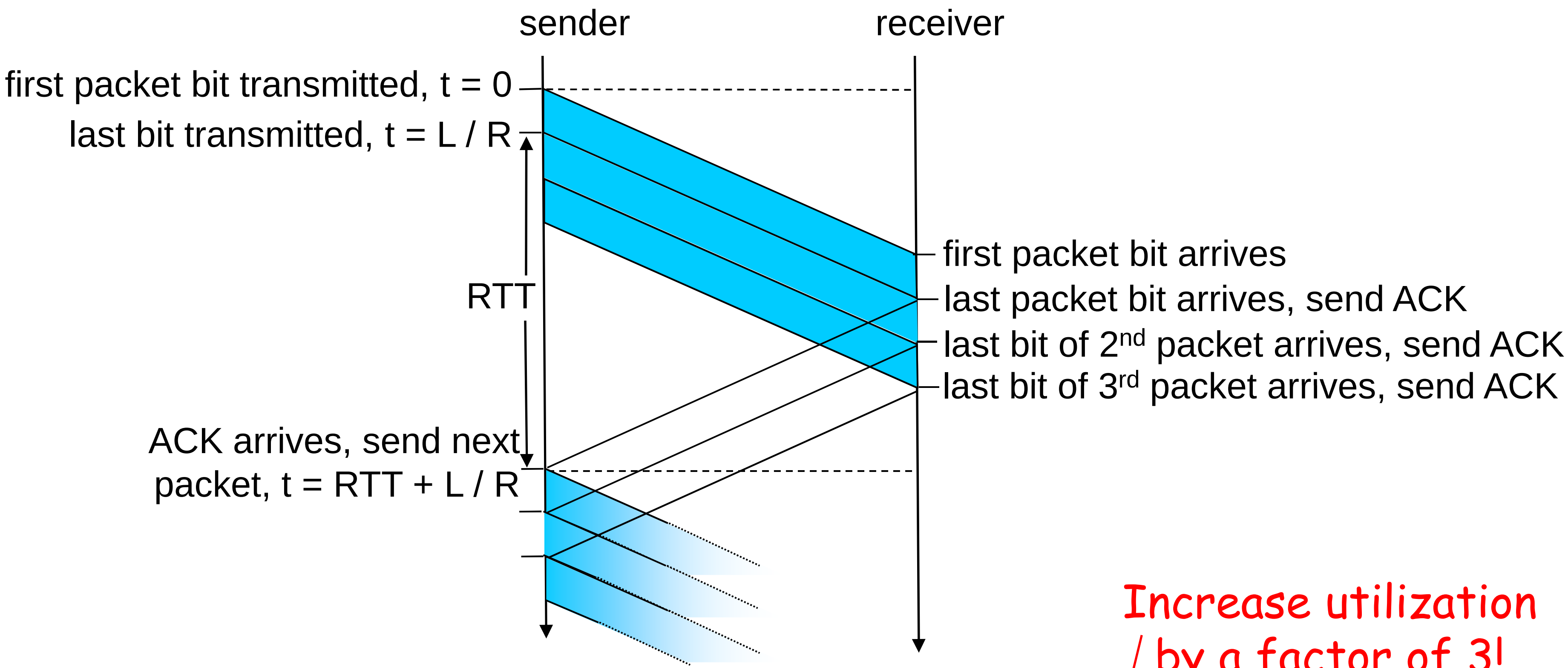
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- full duplex data:
  - bi-directional data flow in same connection
- point-to-point:
  - one sender, one receiver
- flow controlled:
  - sender will not overwhelm receiver
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange, MSS (maximum segment size)
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



Pipelining: increased utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

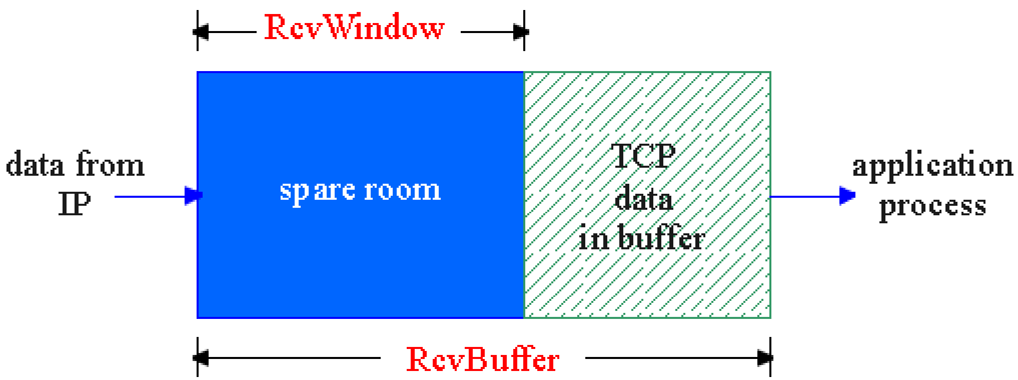
TCP Flow Control: Dynamic sliding windows

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

RcvBuffer = size or TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **RcvWindow field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- 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, cumulatively ACKed segments
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponential weighted average: influence of given sample decreases exponentially fast
- typical value of x: 0.1

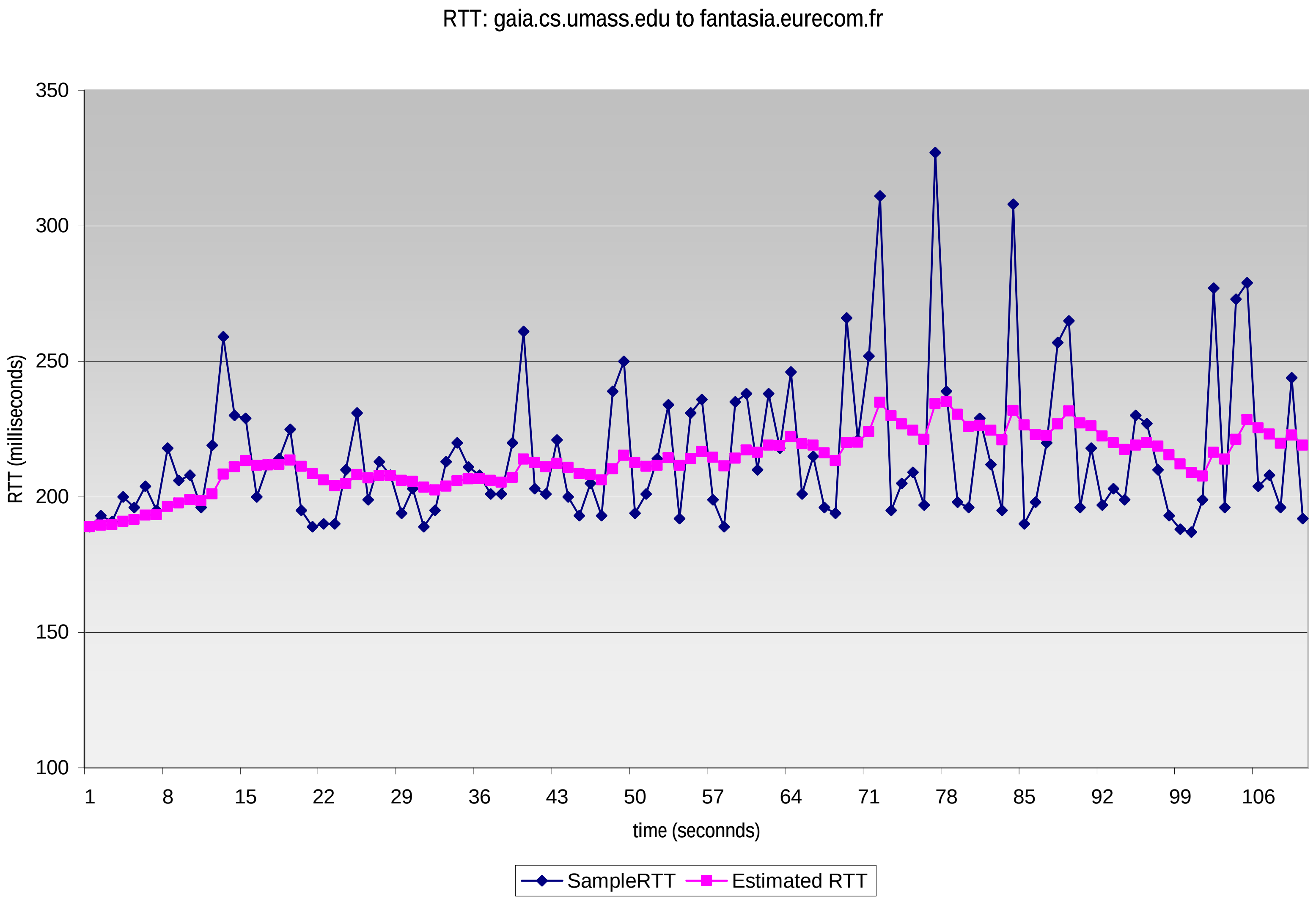
Setting the timeout

- **EstimtedRTT** plus "safety margin"
- large variation in **EstimatedRTT ->** larger safety margin

```
Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation +
             x*|SampleRTT-EstimatedRTT|
```

Example RTT estimation:



TCP seq. #'s and ACKs

Seq. #'s: byte stream "number" of first byte in segment's data

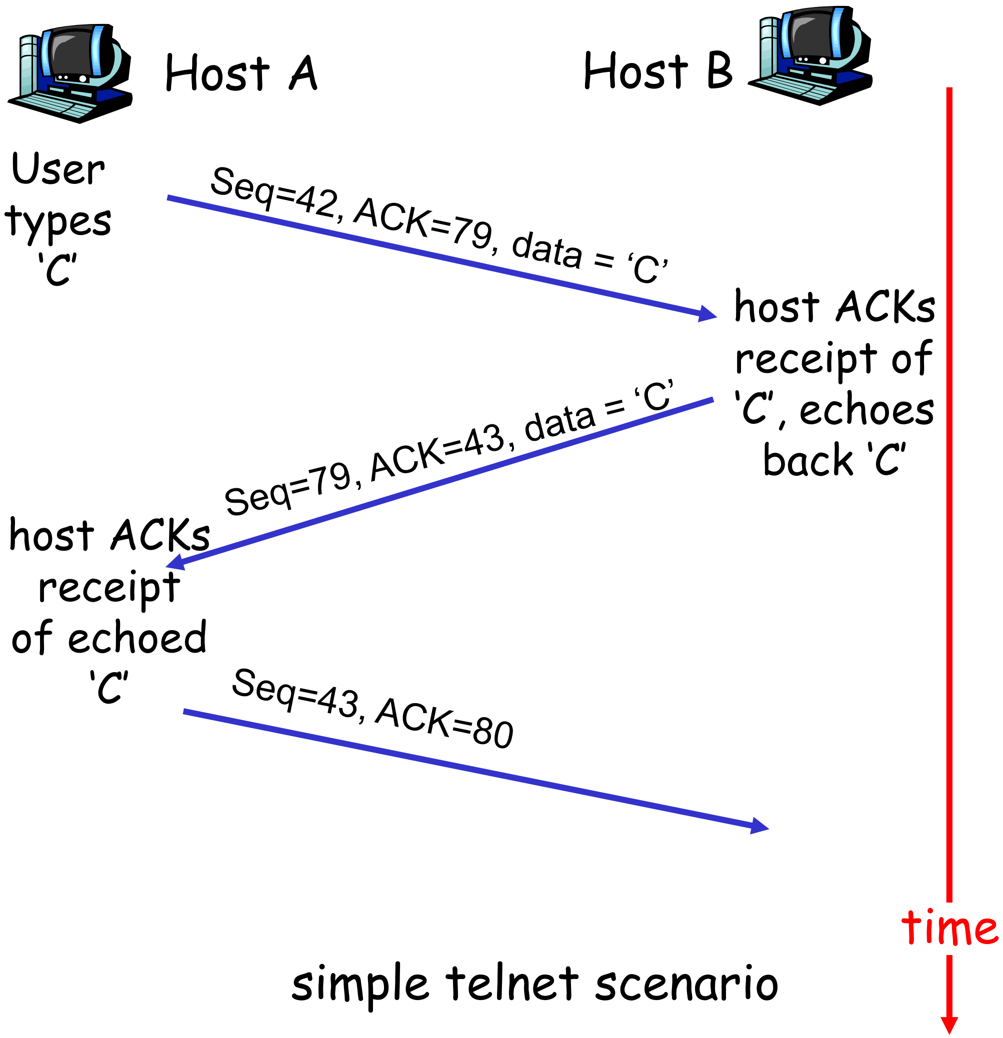
- initially random (to min. probability of conflict, with "historical" segments, buffered in the network)
- recycling sequence numbers?

ACKs: 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

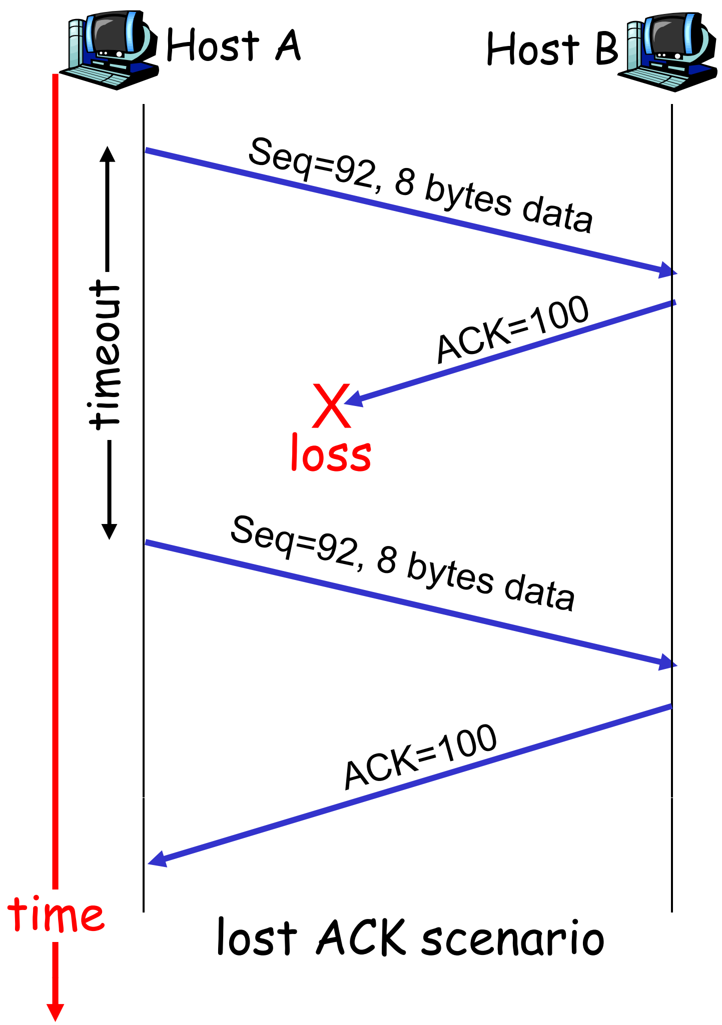


simple telnet scenario

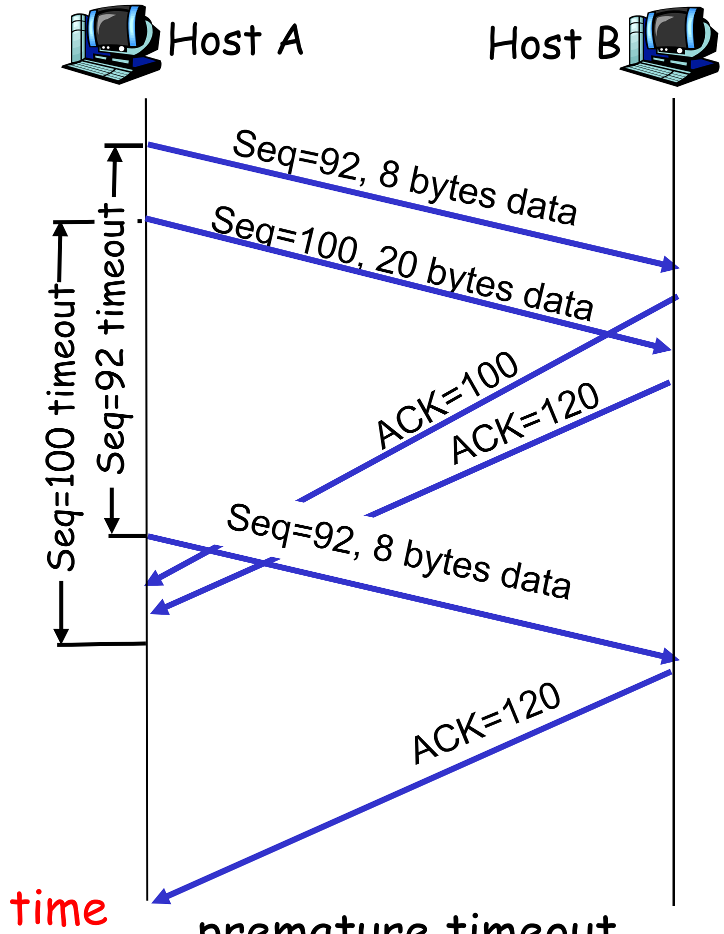
TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: retransmission scenaria



lost ACK scenario



premature timeout,
cumulative ACKs

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.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments - to initialize TCP variables

- *client:* connection initiator

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

- *server:* contacted by client

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

Note: connection is between processes (socket end-points); underlying network may be connectionless

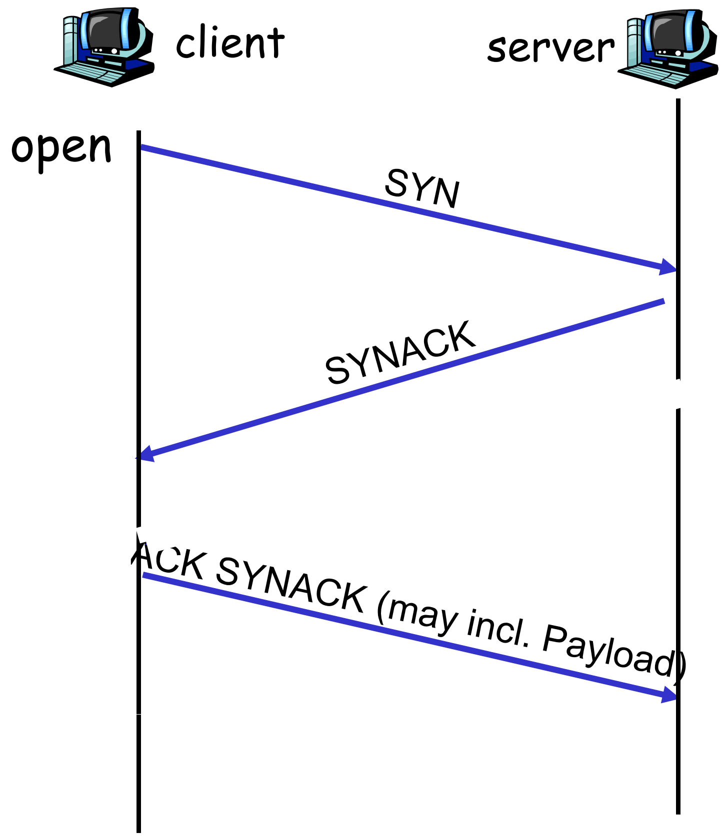
TCP Connection Management: Establishing a connection

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN:
- allocates buffers (can be "step4", cf. SYNflood attacks)
- specifies server-> client initial seq. #
- ACKs received SYN (SYNACK control segment)
- Negotiate MSS

Step 3: client receives SYNACK-segm:
- allocates buffers
- ACKs the SYNACK (segment may contain payload)



TCP Connection Management: Closing a connection

Requires distributed agreement (cf. also Byzantine generals problem)

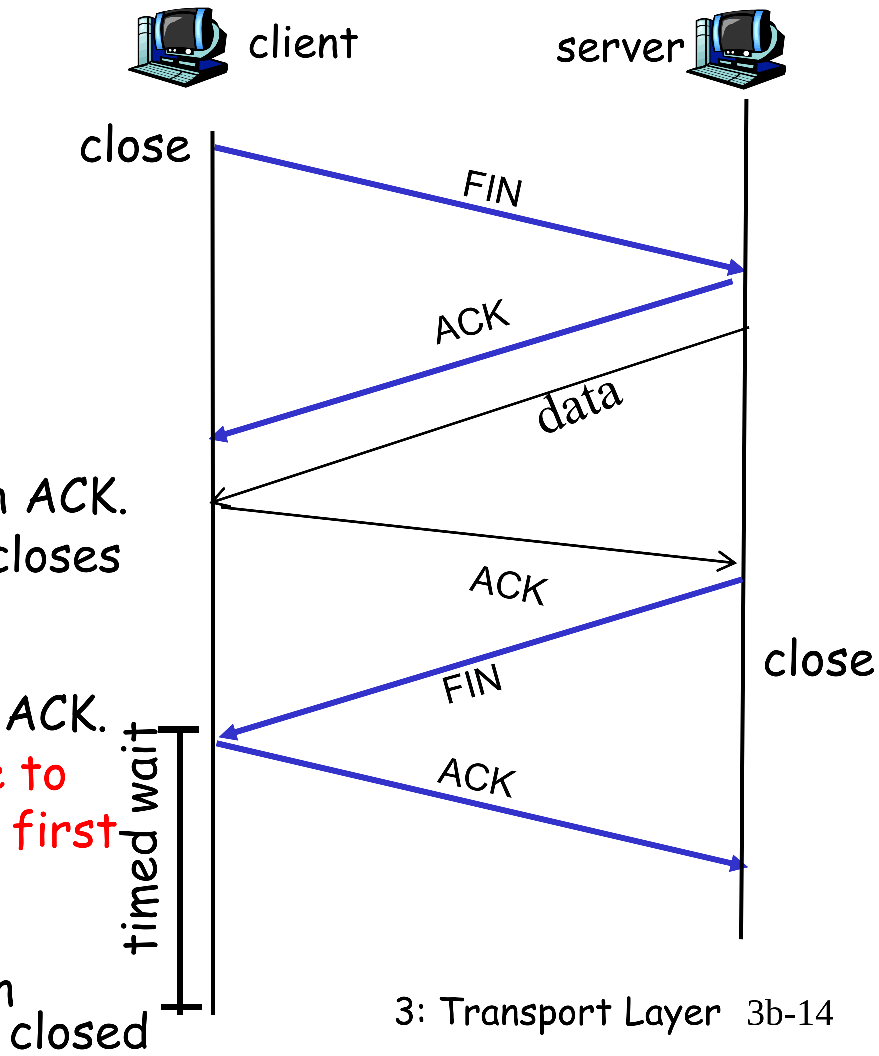
client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

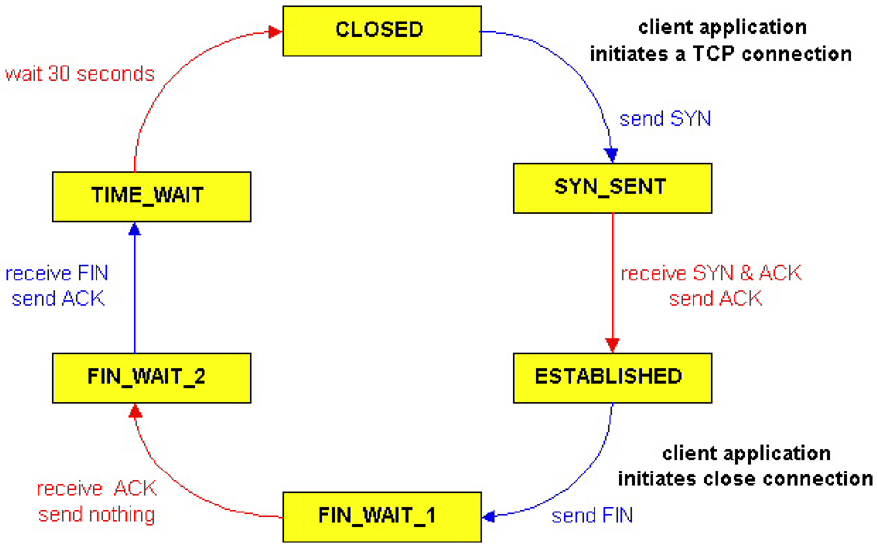
Step 2: server receives FIN, replies with ACK. (Possibly has more data to send; then closes connection, sends FIN.

Step 3: client receives FIN, replies with ACK. Enters "timed wait" (needed to be able to respond with ACK to received FINs, if first ACK was lost)

Step 4: server, receives ACK. Connection closed.

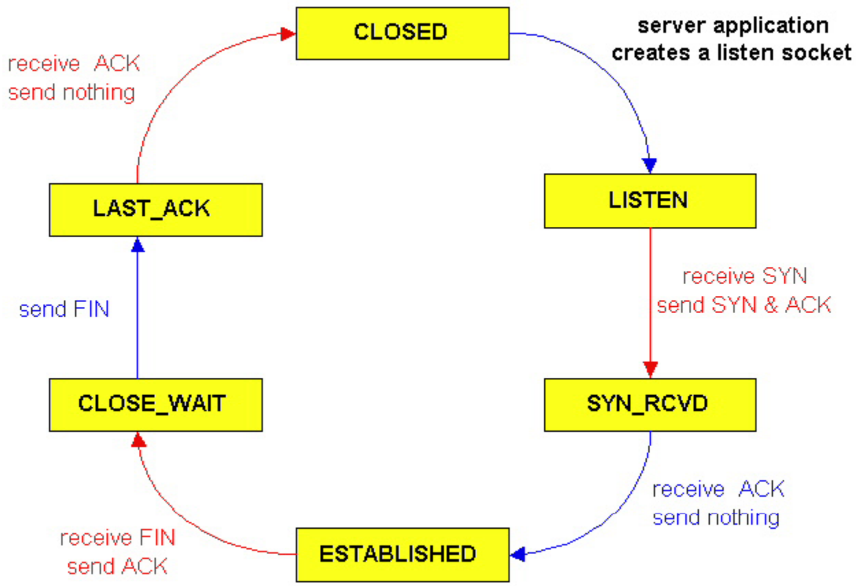


TCP Connection Management (cont)

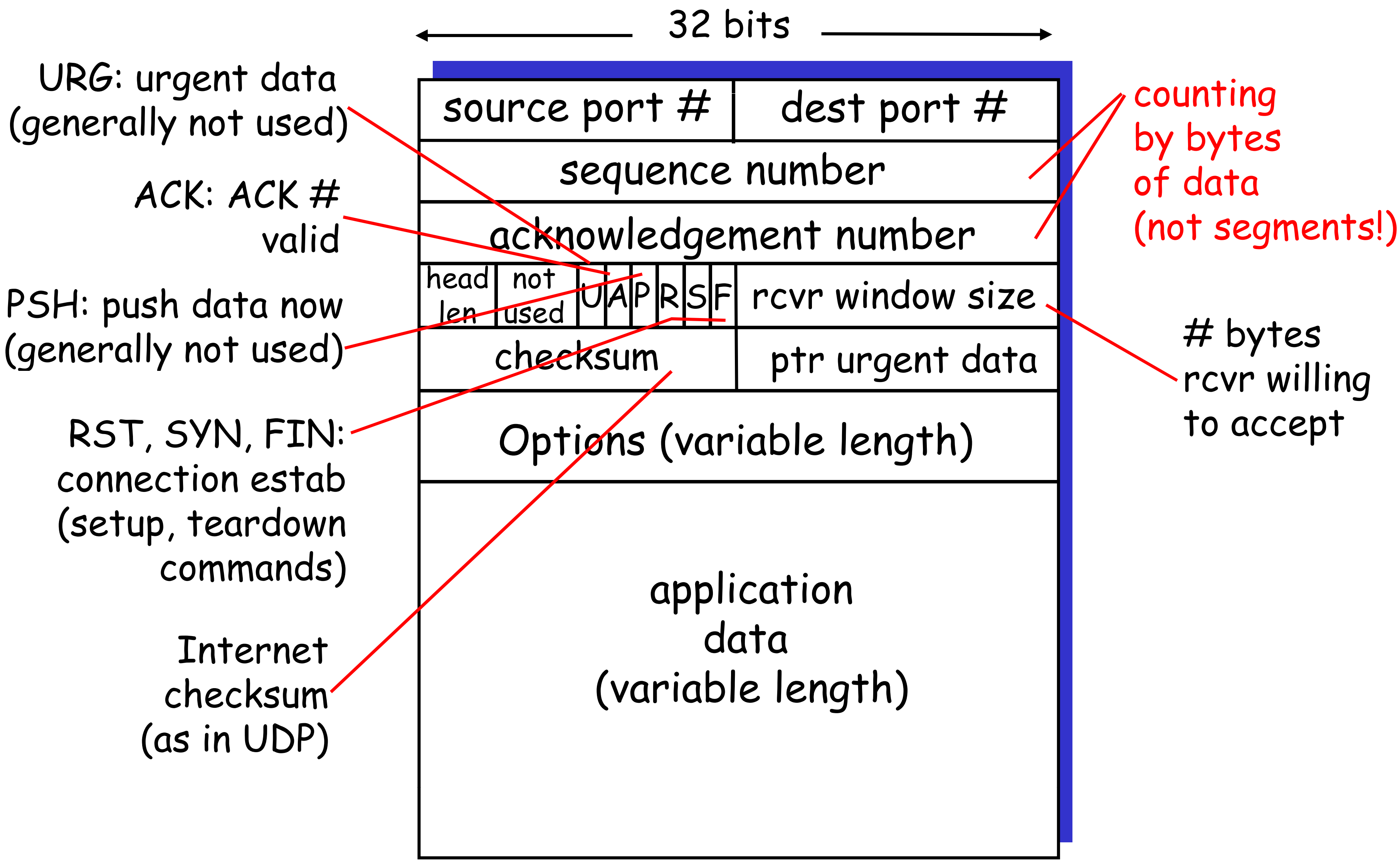


TCP client lifecycle

TCP server lifecycle



TCP segment structure



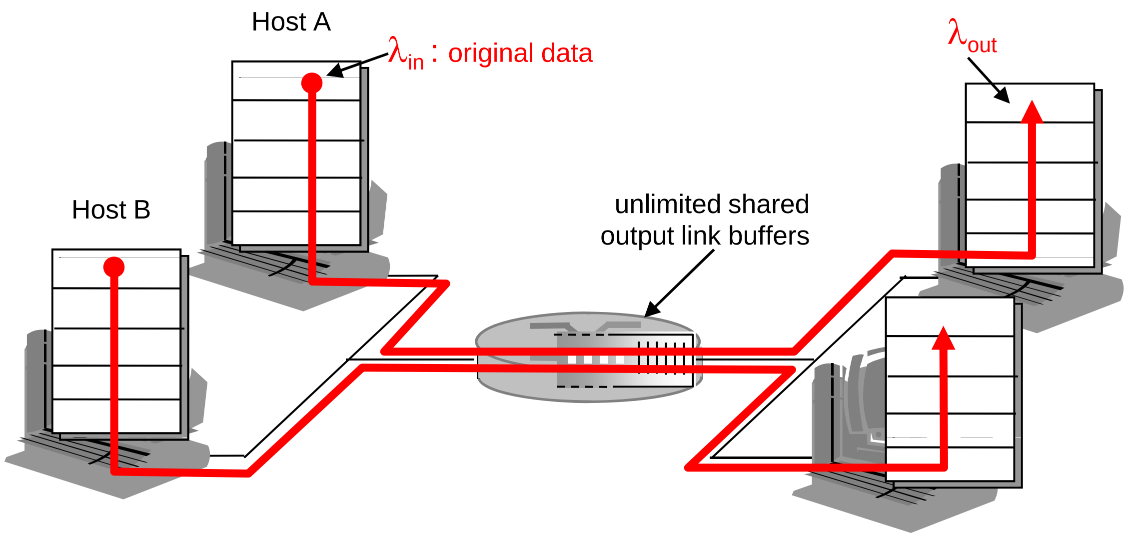
Principles of Congestion Control

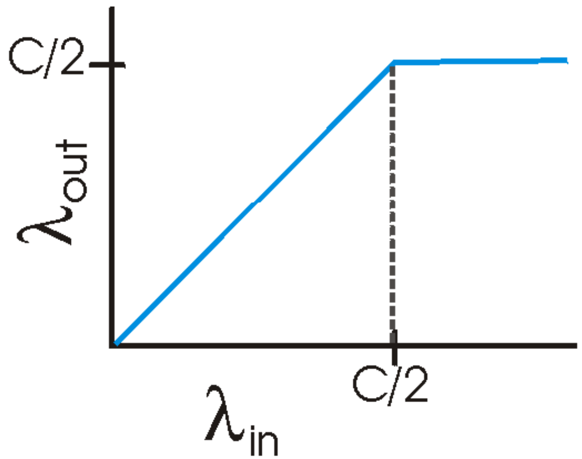
Congestion: a top-10 problem!

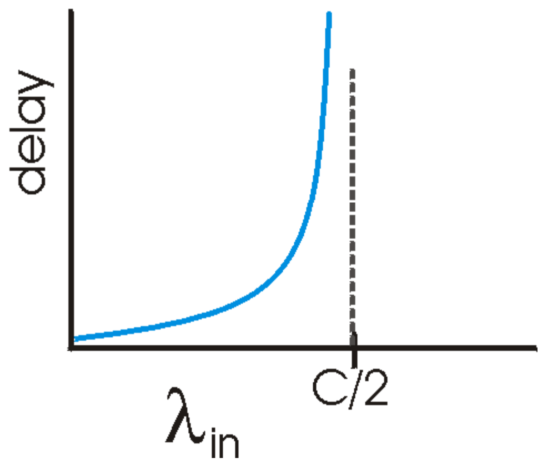
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



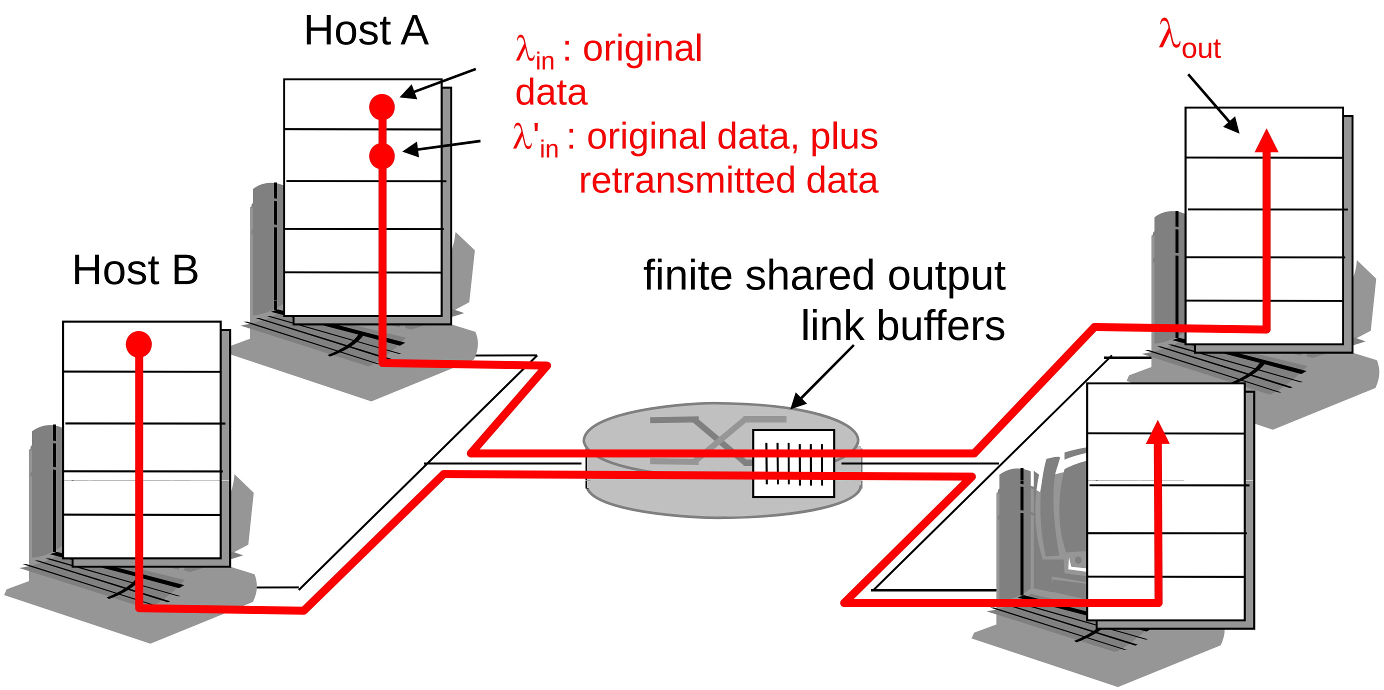




- large delays when congested
- maximum achievable throughput

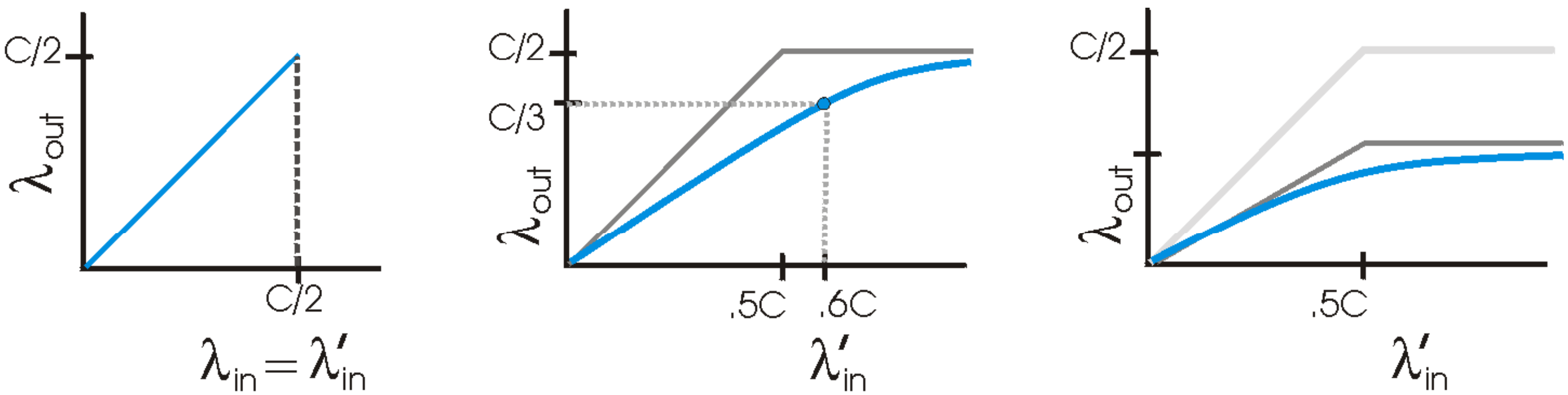
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmits lost packets



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



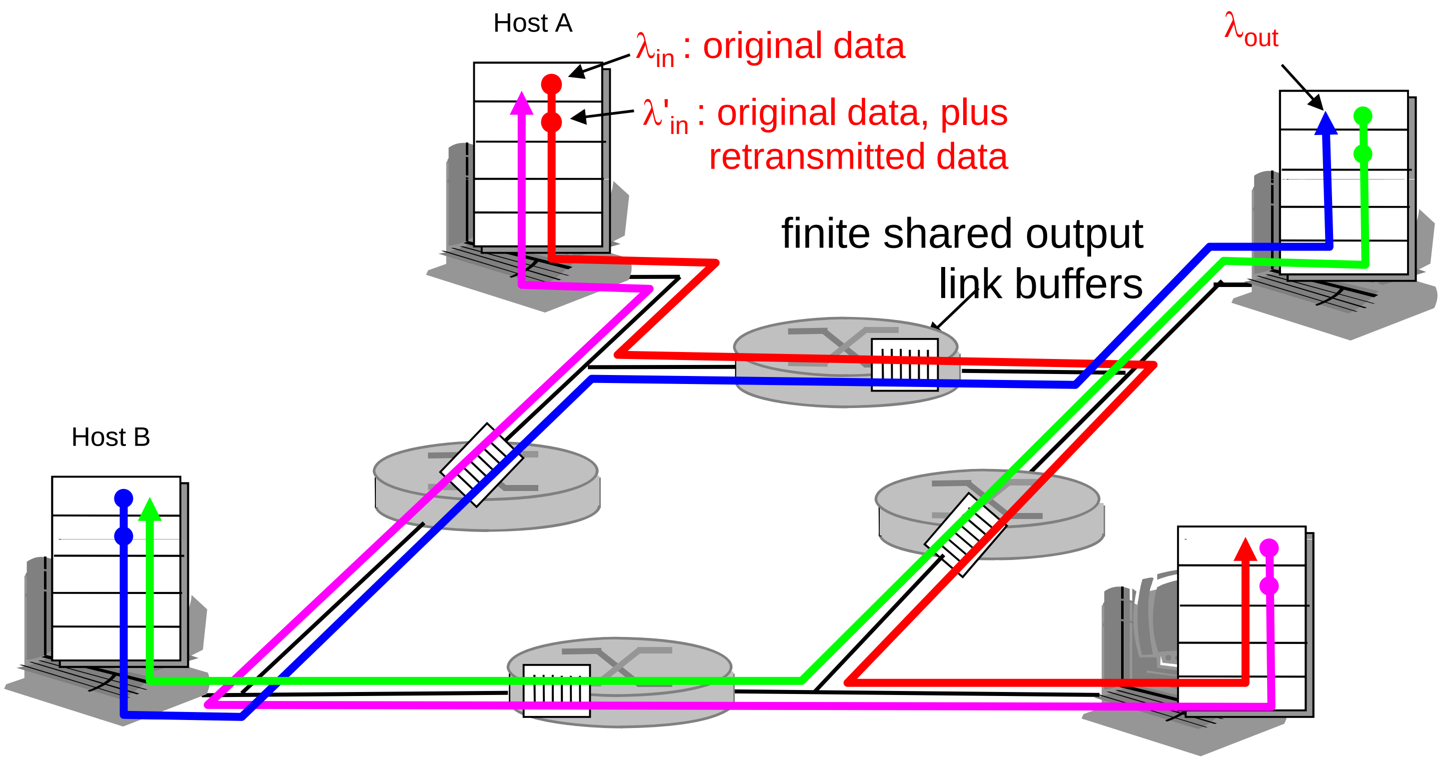
"costs" of congestion: (more congestion ☹)

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

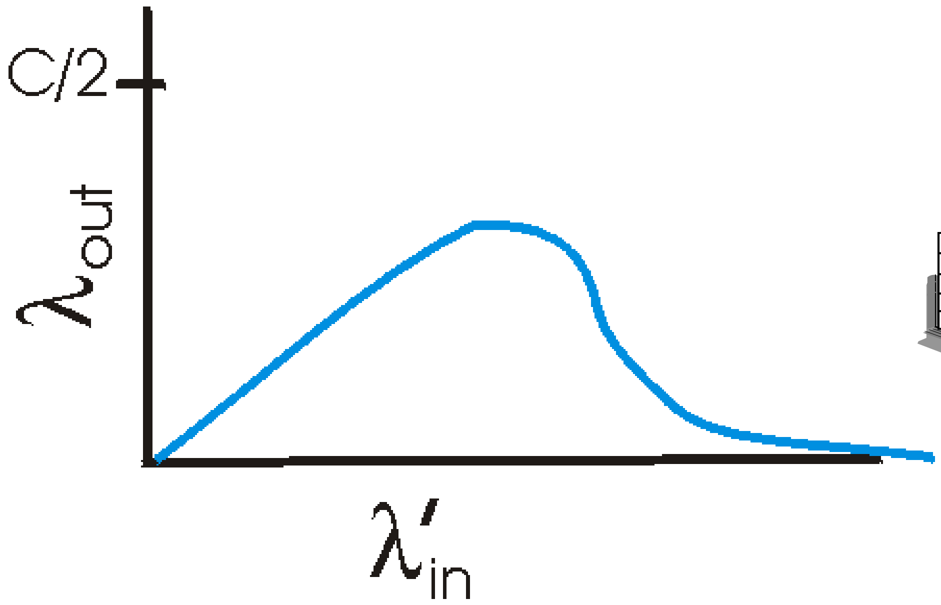
Causes/costs of congestion: scenario 3

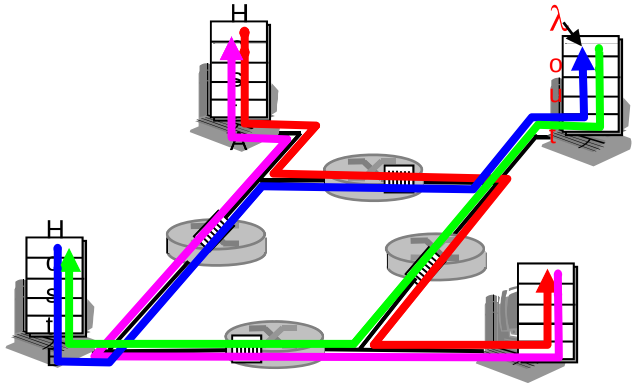
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Summary causes of Congestion:

- Bad network design (bottlenecks)
- Bad use of network : feed with more than can go through
- … congestion ☺ (bad congestion-control policies e.g. dropping the wrong packets, etc)

Two broad approaches towards congestion control

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP (focus here)

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
- routers may serve flows with parameters, may also apply admission control on connection-request
- (*see later, in assoc. with N/W layer, ATM policies, multimedia apps & QoS, match of traffic needs with use of the N/W*)

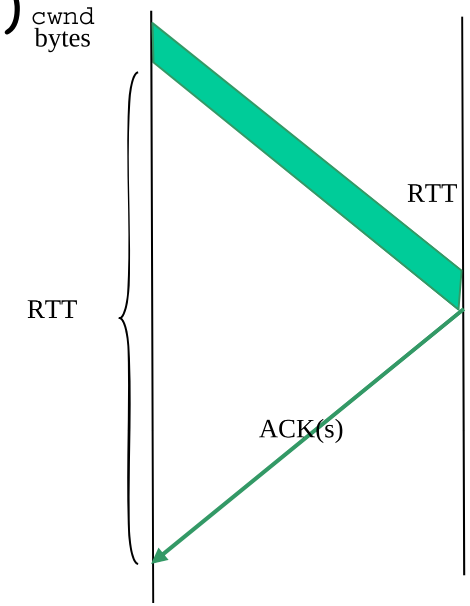
TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:

  LastByteSent-LastByteAcked ≤ CongWin
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** is dynamic, function of perceived network congestion (NOTE: different than receiver's window!)



How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

Q: any problem with this?

three mechanisms:

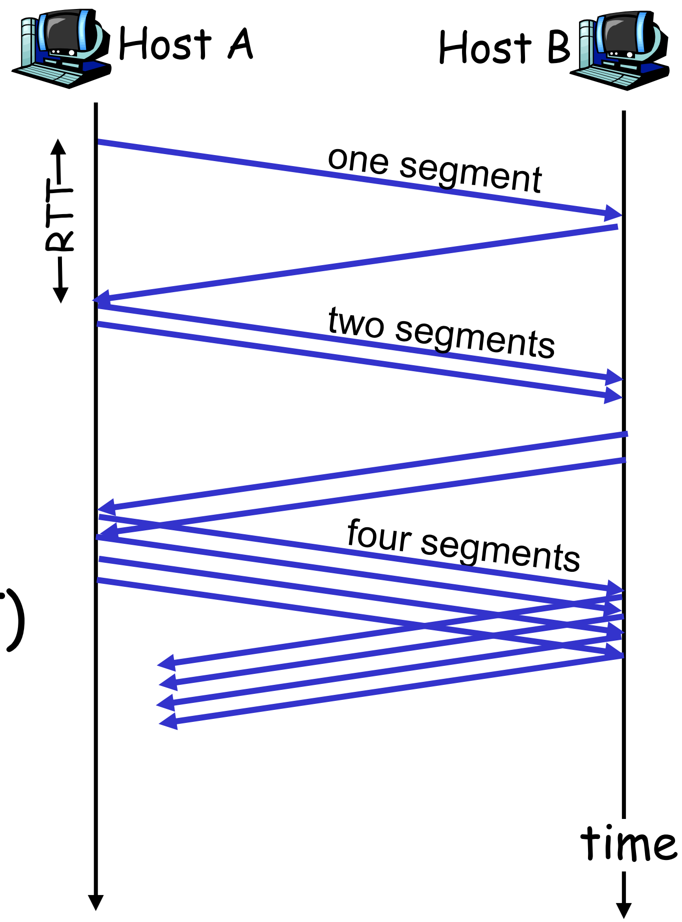
- AIMD
- slow start
- conservative after timeout events

TCP Slowstart

Slowstart algorithm

```
initialize: Congwin = 1
for (each segment ACKed)
      Congwin = 2 * Congwin
until (loss event OR
        CongWin > threshold)
```

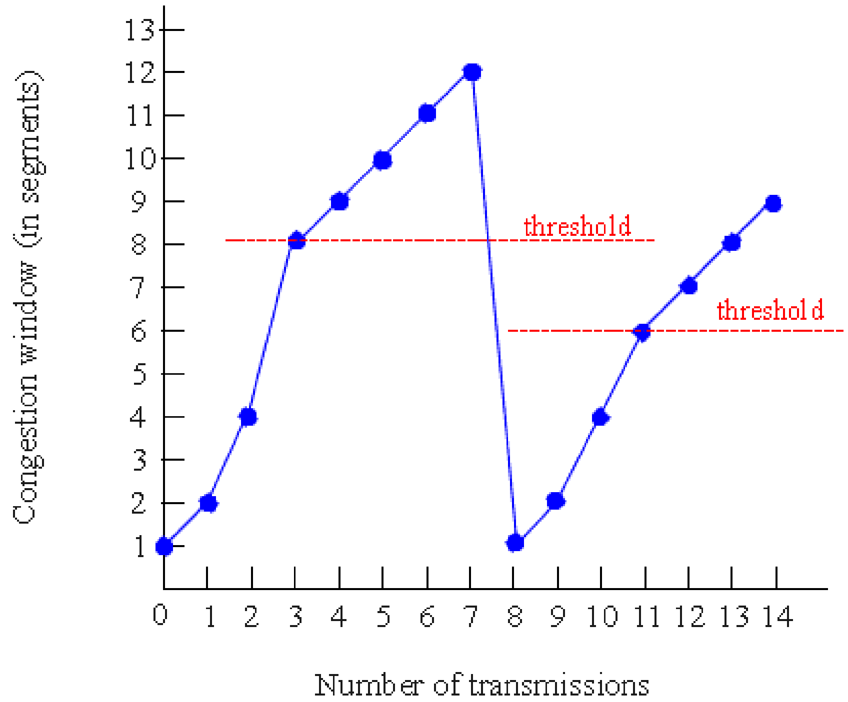
- exponential increase (per RTT) in window size (not so slow !?)
- loss event = timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)



TCP Congestion Avoidance

Congestion avoidance

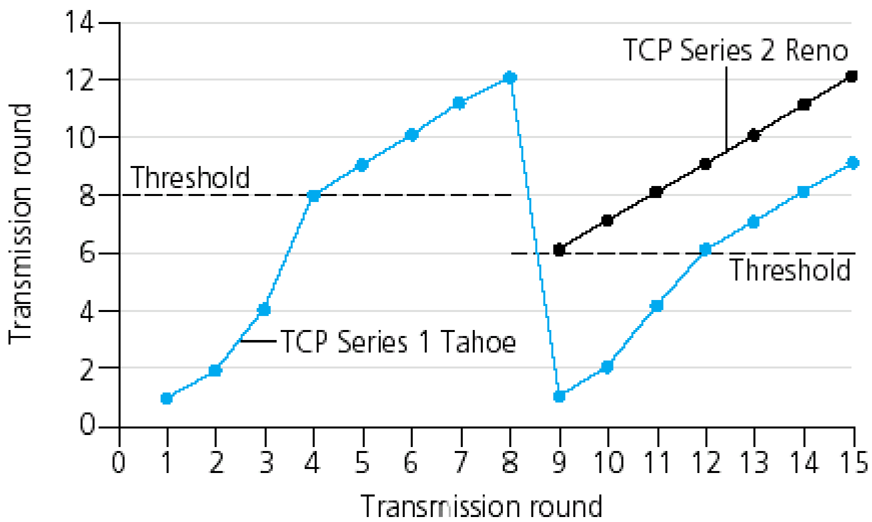
```
/* slowstart is over        */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



Refinement (Reno)

Avoid slow starts!

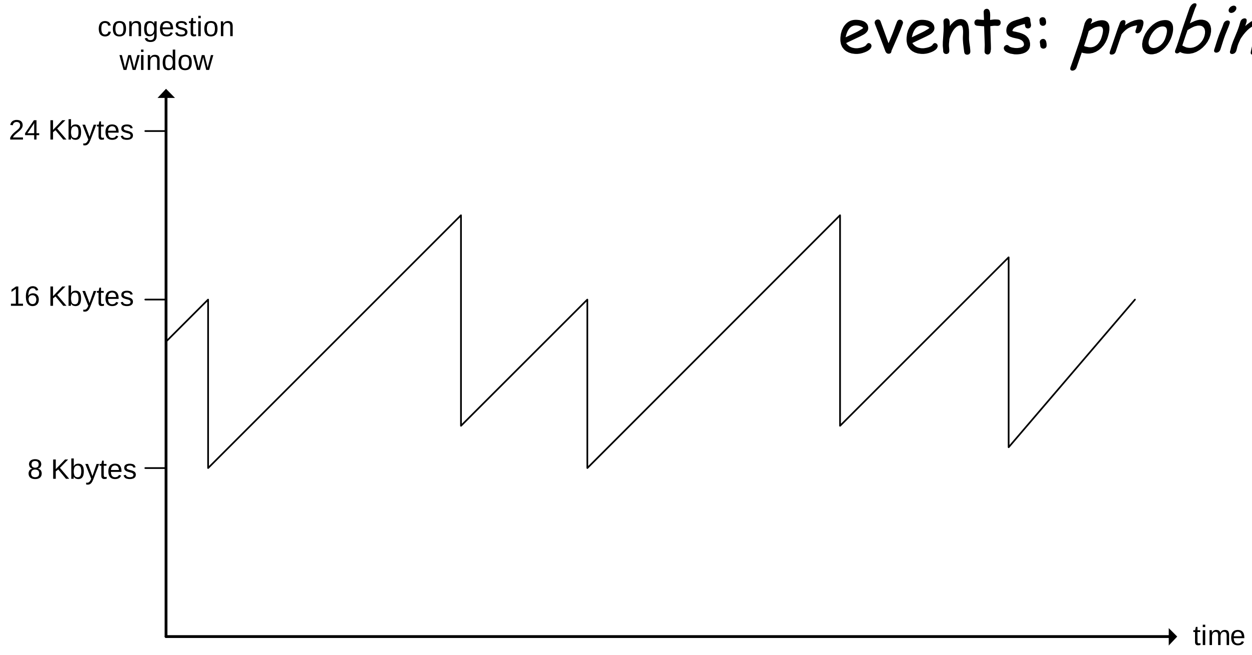
Go to linear increase after 3rd duplicate ack, starting from window of size (1/2 window before change)



TCP AIMD

multiplicative decrease: cut **CongWin** in half after loss event

additive increase: increase **CongWin** by 1 MSS every RTT in the absence of loss events: *probing*



Long-lived TCP connection

Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

TCP sender congestion control

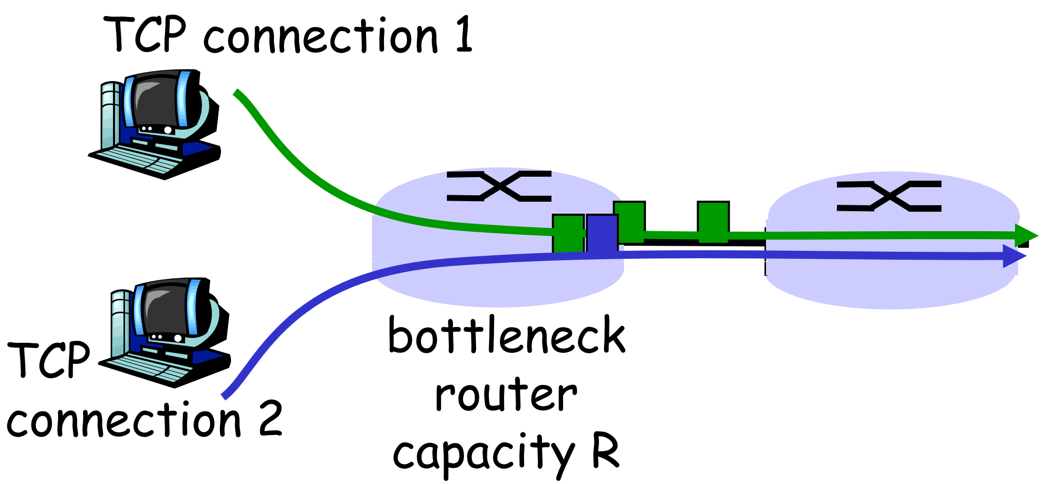
Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP Fairness

TCP's congestion avoidance effect: AIMD: *additive increase, multiplicative decrease*

- increase window by 1 per RTT
- decrease window by factor of 2 on loss event

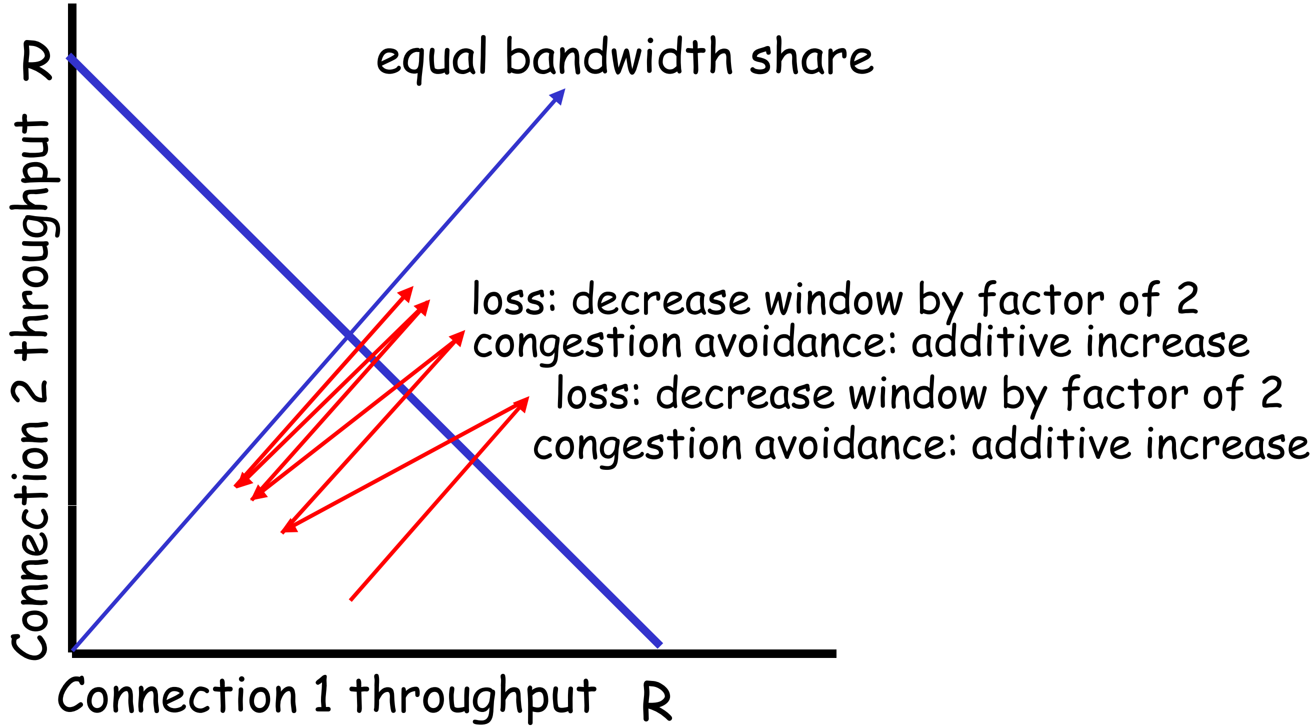
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

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 rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel cnctions between 2 hosts.
- Web browsers do this ....

TCP delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

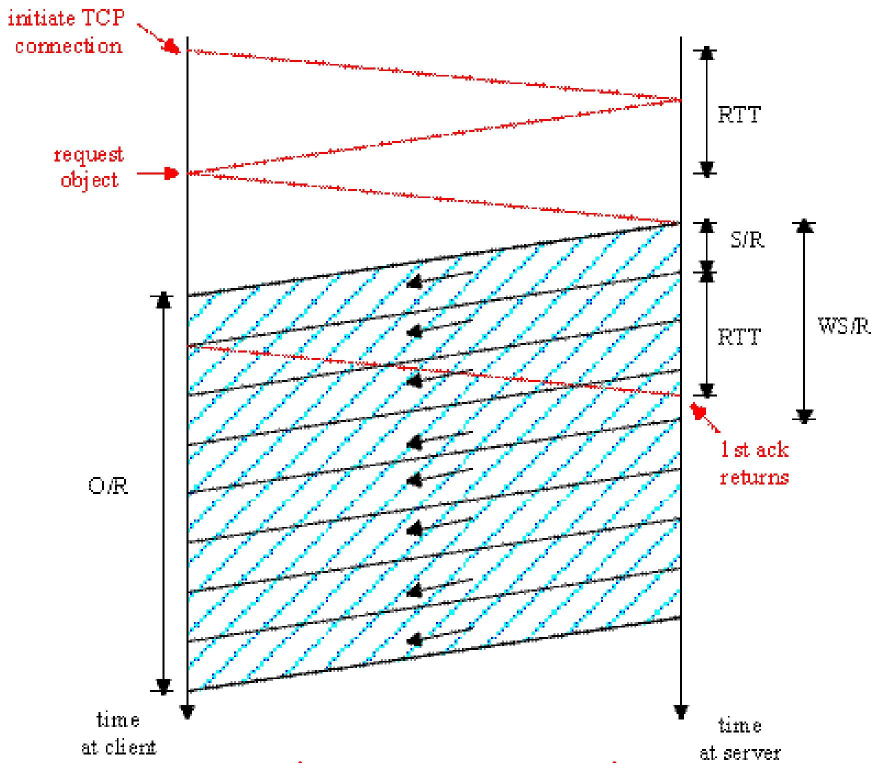
- TCP connection establishment
- data transfer delay

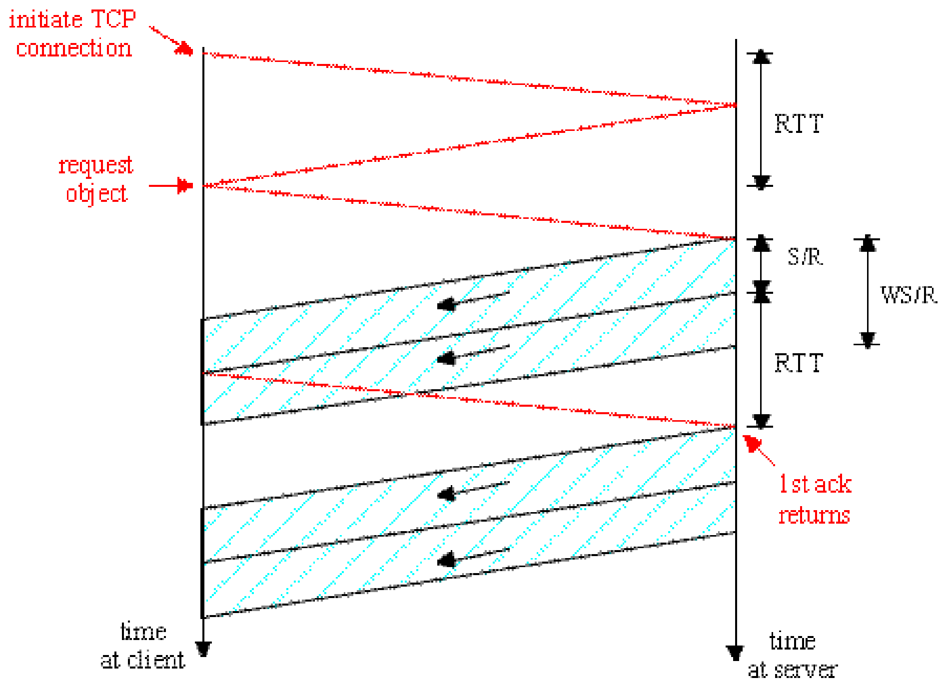
Notation, assumptions:

- Assume one link between client and server of rate R
- Assume: fixed congestion window, W segments
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

TCP delay Modeling: Fixed window

K:= O/WS





Case 1: WS/R > RTT + S/R:
ACK for first segment in window returns before window's worth of data sent

latency = 2RTT + O/R

Case 2: WS/R < RTT + S/R:
wait for ACK after sending window's worth of data sent

latency = 2RTT + O/R
+ (K-1)[S/R + RTT - WS/R]

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where P is the number of times TCP stalls at server:

$$P = \min\{Q, K-1\}$$

where
- Q = number of times the server would stall until cong. window grows larger than a "full-utilization" window (if the object were of unbounded size).

- K = number of (incremental-sized) congestion-windows that "cover" the object.

TCP Delay Modeling: Slow Start (2)

Delay components:

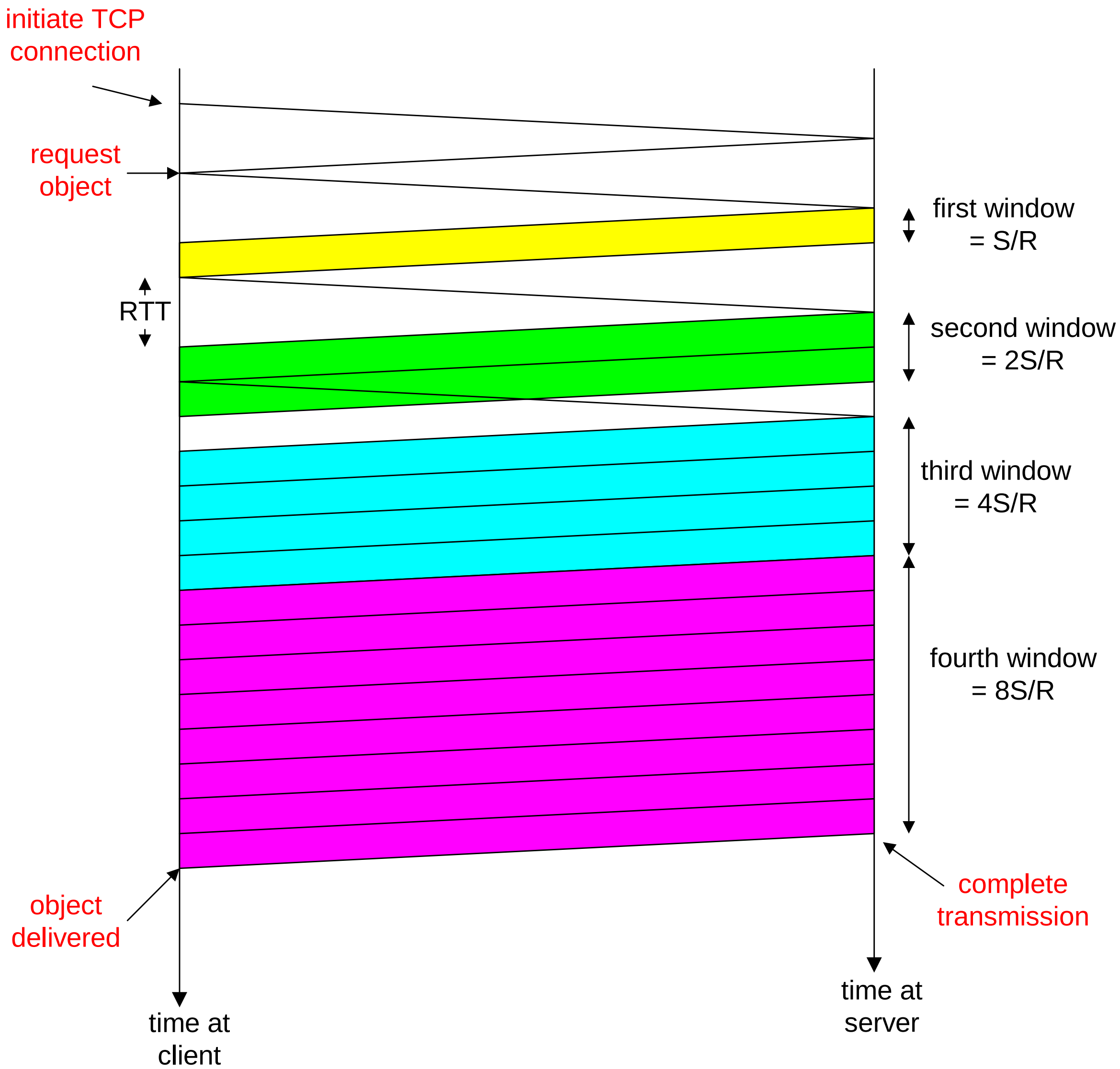
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
P = min{K-1,Q} times

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2

Server idles P=2 times

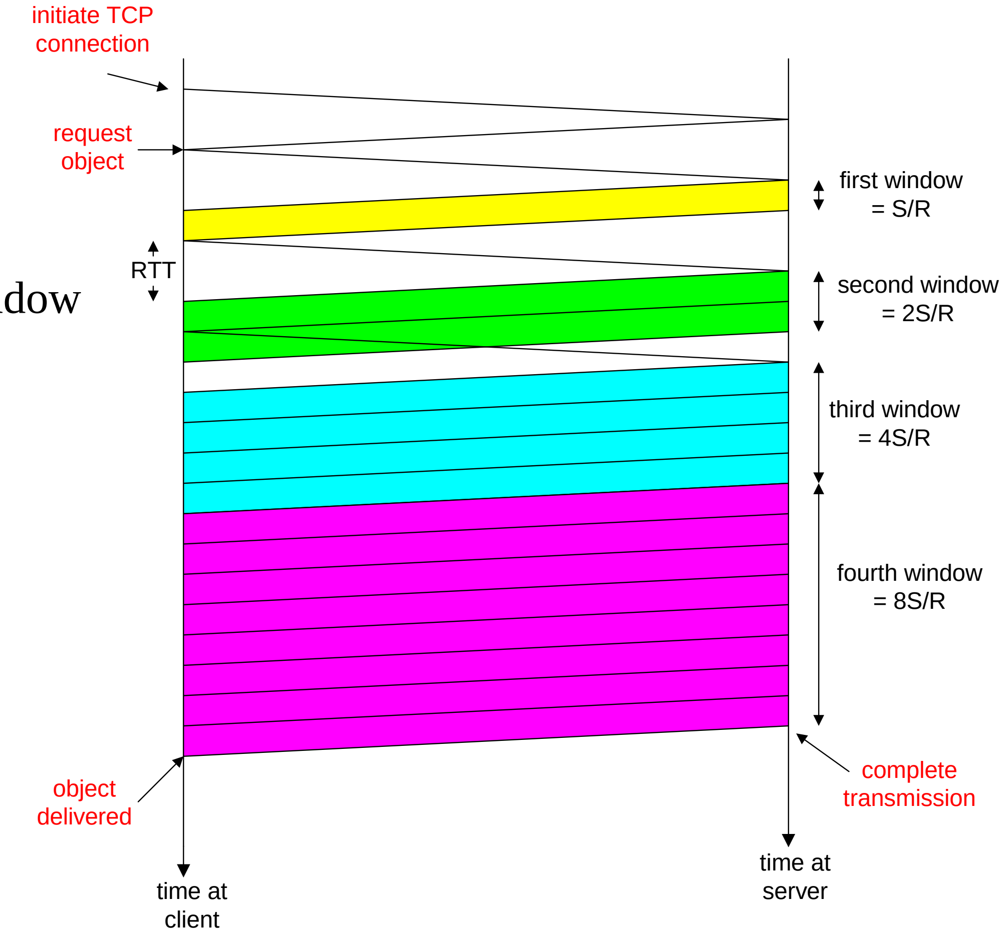


TCP Delay Modeling (3)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement}$$

$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^+ = \text{idle time after the } k\text{th window}$$



$$\begin{aligned}
\text{delay} &= \frac{O}{R} + 2RTT + \sum_{p=1}^{P} idleTime_p \\
&= \frac{O}{R} + 2RTT + \sum_{k=1}^{P} [\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}] \\
&= \frac{O}{R} + 2RTT + P[RTT + \frac{S}{R}] - (2^P - 1)\frac{S}{R}
\end{aligned}$$

TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

Calculation of Q, number of idles for infinite-size object, is similar.

Wireless TCP

Problem: higher data error-rate destroys congestion control principle (assumption)

Possible solutions:

- **Non-transparent (indirect):** manage congestion-control in 2 sub-connections (one wired, one wireless). *But …* the semantics of a connection changes: ack at the sender means that base-station, (not the receiver) received the segment
- **Transpartent:** use extra rules at the base-station (network layer retransmissions...) to "hide" the errors of the wireless part from the sender. *But …* the sender may still timeout in the meanwhile and think that there is congestion ...
- **Vegas algorithm**: observe RTT estimation and reduce transmission rate when in danger of loss

Chapter 3: Summary

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

Next:

- leaving the network "edge" (application transport layer)
- into the network "core"