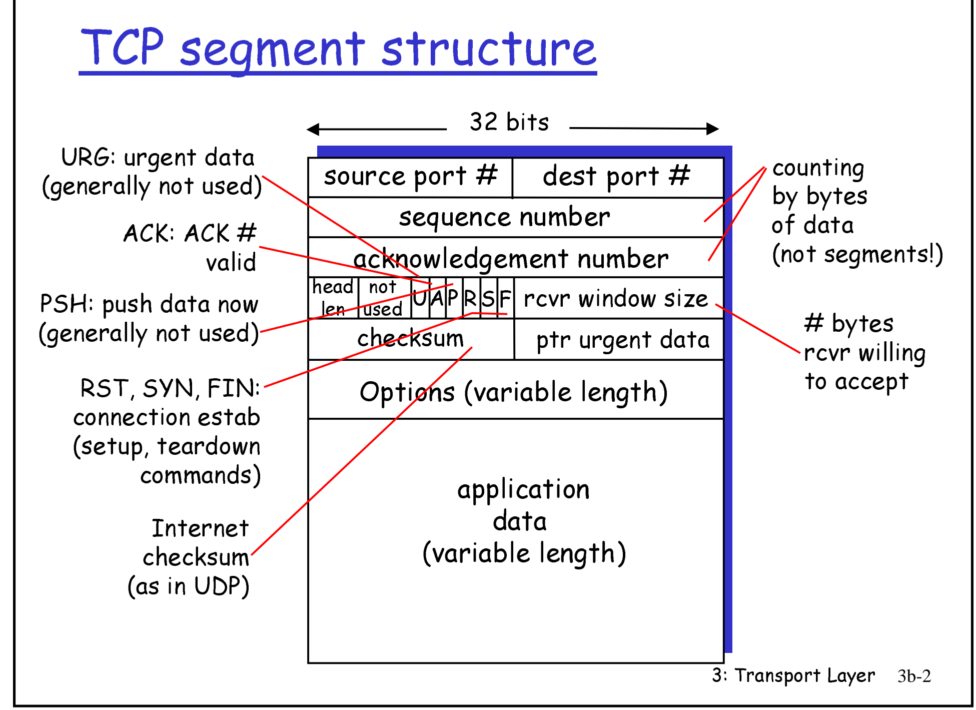
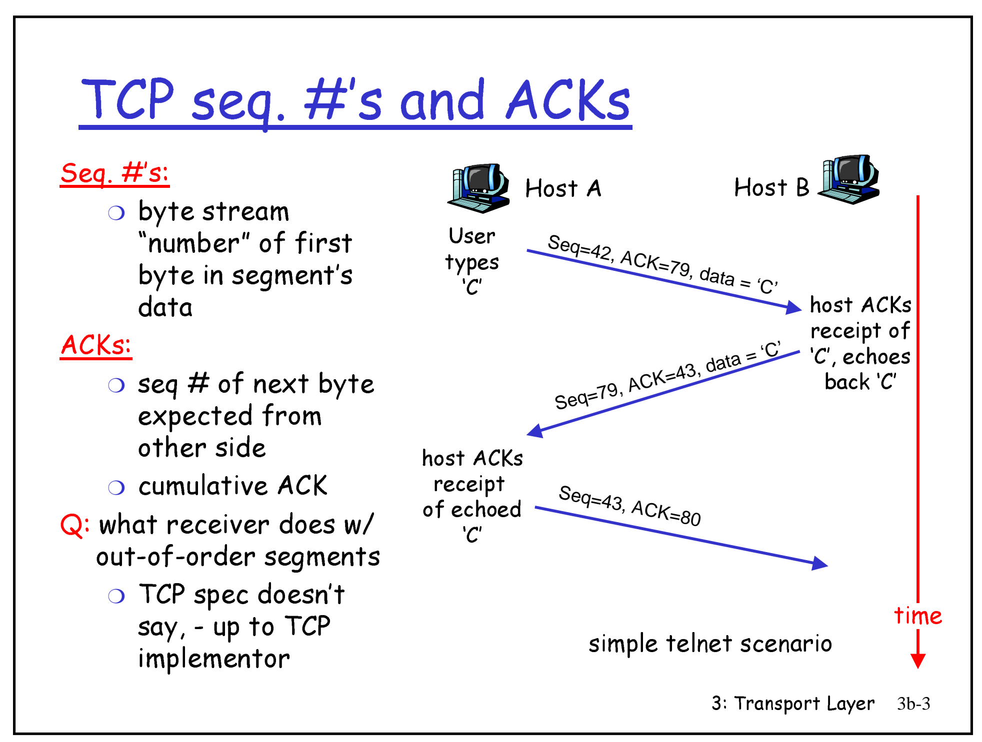
TCP: Overview

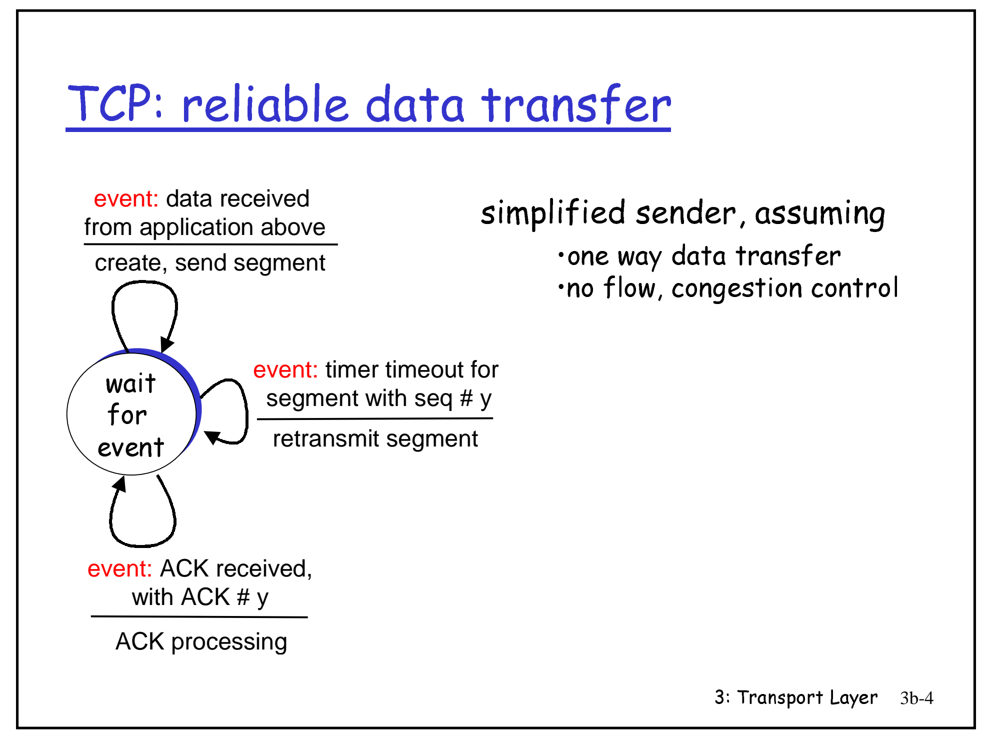
RFCs: 793, 1122, 1323, 2018, 2581

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

TCP segment structure




TCP seq. #'s and ACKs
Seq. #'s:
byte stream "number" of first byte in segment's data
ACKs:
seq # of next byte expected from other side
cumulative ACK
Q: what receiver does w/ out-of-order segments
TCP spec doesn't say, - up to TCP implementor
Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80
time
simple telnet scenario
3: Transport Layer 3b-3


TCP: reliable data transfer
event: data received from application above
create, send segment
wait for event
event: timer timeout for segment with seq # y
retransmit segment
event: ACK received, with ACK # y
ACK processing
simplified sender, assuming
•one way data transfer
•no flow, congestion control
3: Transport Layer 3b-4

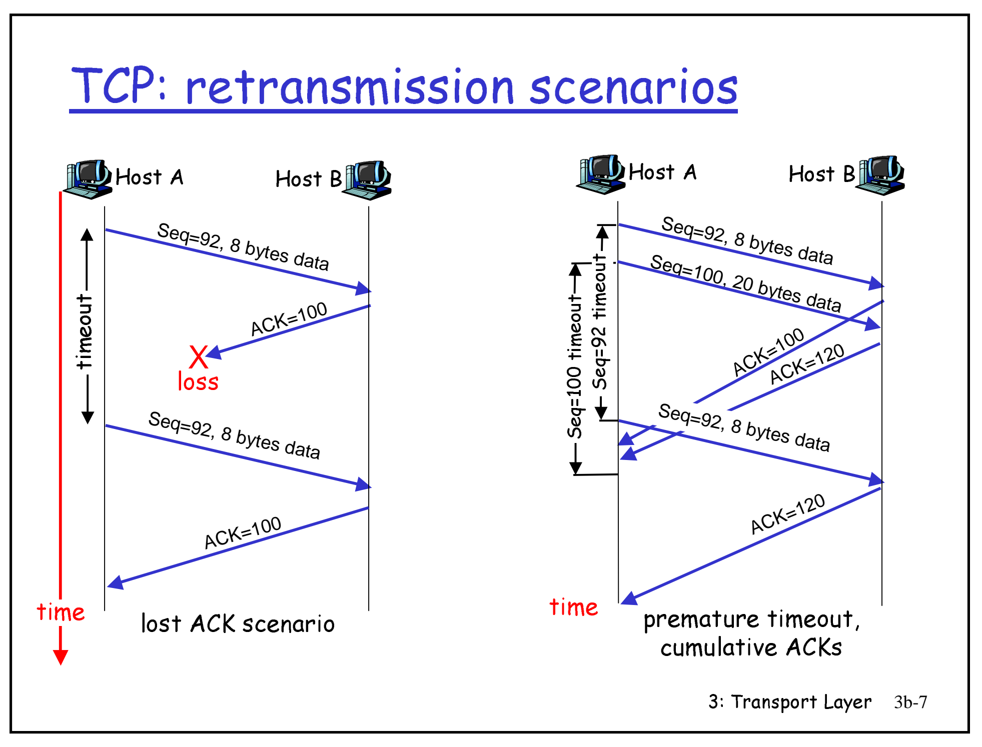
TCP: reliable data transfer

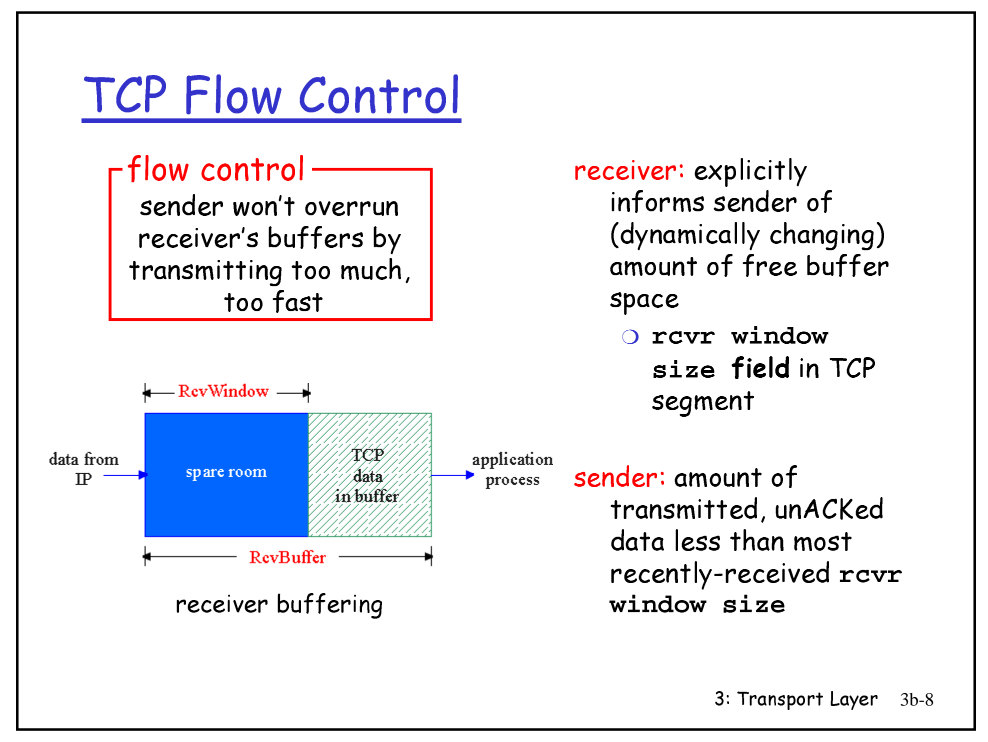
Simplified TCP sender

```
00   sendbase = initial_sequence number
01   nextseqnum = initial_sequence number
02
03   loop (forever) {
04     switch(event)
05     event: data received from application above
06          create TCP segment with sequence number nextseqnum
07          start timer for segment nextseqnum
08          pass segment to IP
09          nextseqnum = nextseqnum + length(data)
10      event: timer timeout for segment with sequence number y
11          retransmit segment with sequence number y
12          compute new timeout interval for segment y
13          restart timer for sequence number y
14      event: ACK received, with ACK field value of y
15          if (y > sendbase) { /* cumulative ACK of all data up to y */
16              cancel all timers for segments with sequence numbers < y
17              sendbase = y
18              }
19          else { /* a duplicate ACK for already ACKed segment */
20              increment number of duplicate ACKs received for y
21              if (number of duplicate ACKS received for y == 3) {
22                 /* TCP fast retransmit */
23                 resend segment with sequence number y
24                 restart timer for segment y
25              }
26     }  /* end of loop forever */
```


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 (missing) byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap


TCP: retransmission scenarios
Host A
Host B
Seq=92, 8 bytes data
ACK=100
X
loss
timeout
Seq=92, 8 bytes data
ACK=100
time
lost ACK scenario
Host A
Host B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
ACK=120
Seq=92 timeout
Seq=100 timeout
Seq=92, 8 bytes data
ACK=120
time
premature timeout,
cumulative ACKs
3: Transport Layer 3b-7


TCP Flow Control
flow control
sender won't overrun receiver's buffers by transmitting too much, too fast
RcvWindow
data from IP
spare room
TCP data in buffer
application process
RcvBuffer
receiver buffering
receiver: explicitly informs sender of (dynamically changing) amount of free buffer space
rcvr window size field in TCP segment
sender: amount of transmitted, unACKed data less than most recently-received rcvr window size
3: Transport Layer 3b-8

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
  - or timestamp pkts
- `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 moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout

- RTT plus "safety margin"
- large variation in `EstimatedRTT` -> larger safety margin

```
Timeout = EstimatedRTT + 4*Deviation

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

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator

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

- *server:* contacted by client

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

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, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

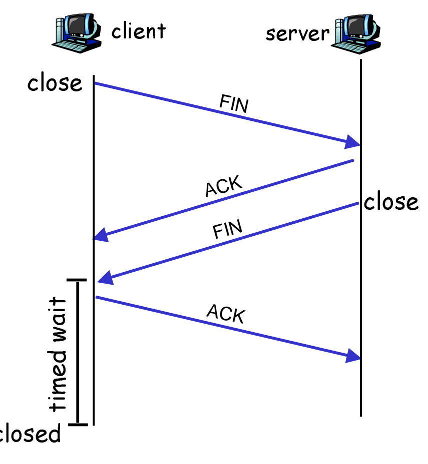
TCP Connection Management (cont.)

Closing a connection:

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. Closes connection, sends FIN.



TCP Connection Management (cont.)

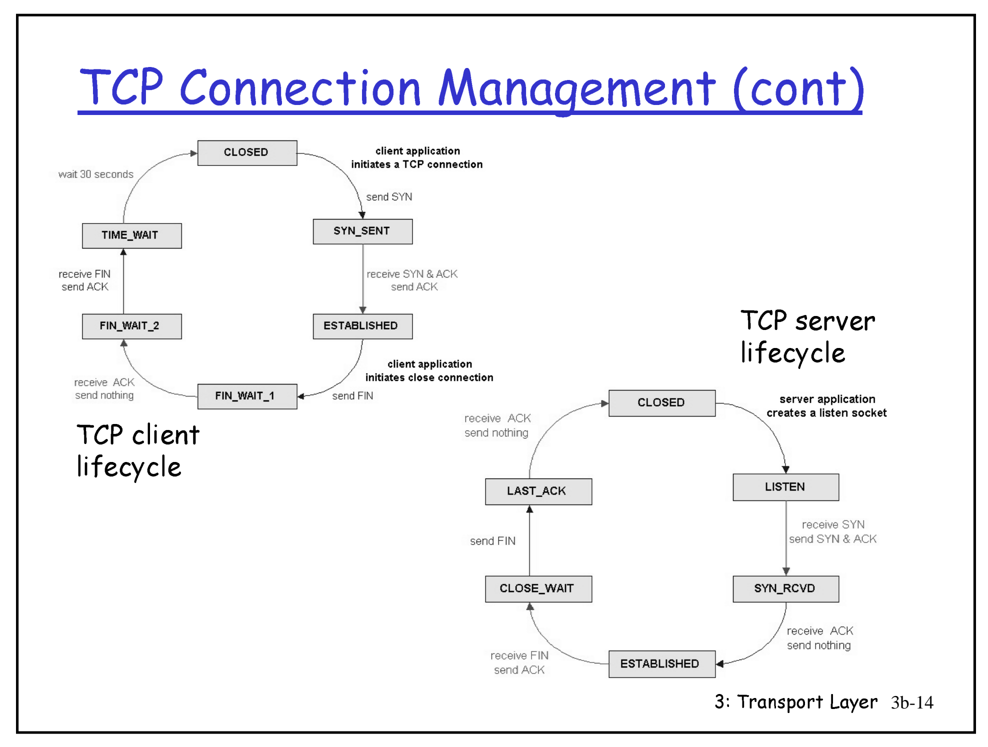
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

client
server
closing
FIN
ACK
closing
FIN
timed wait
ACK
closed
closed

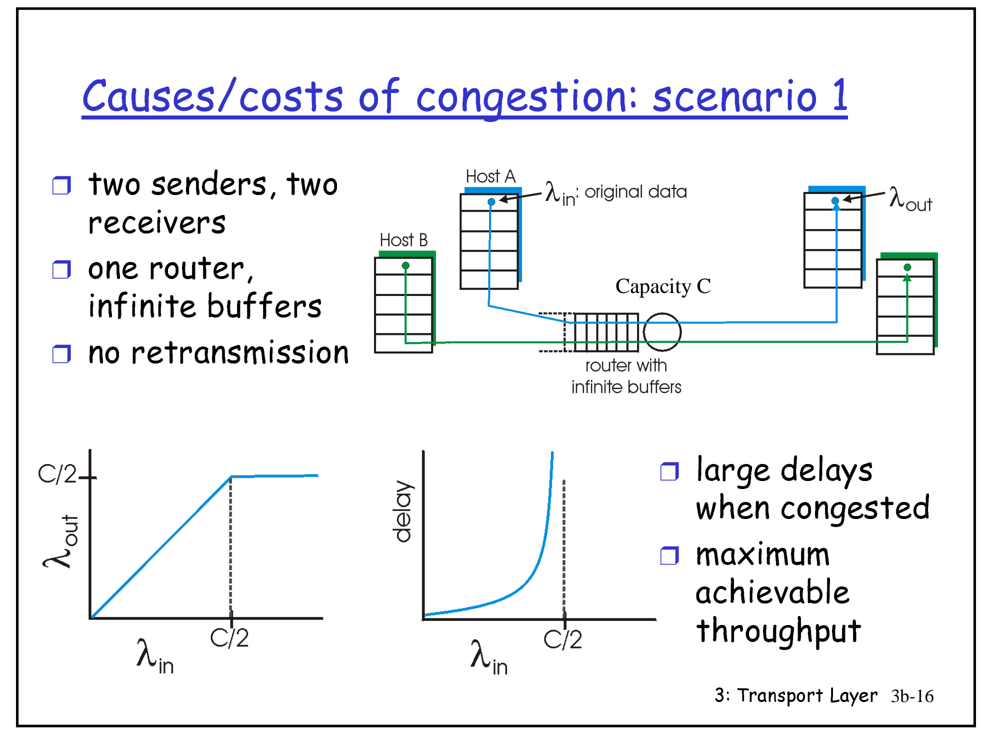


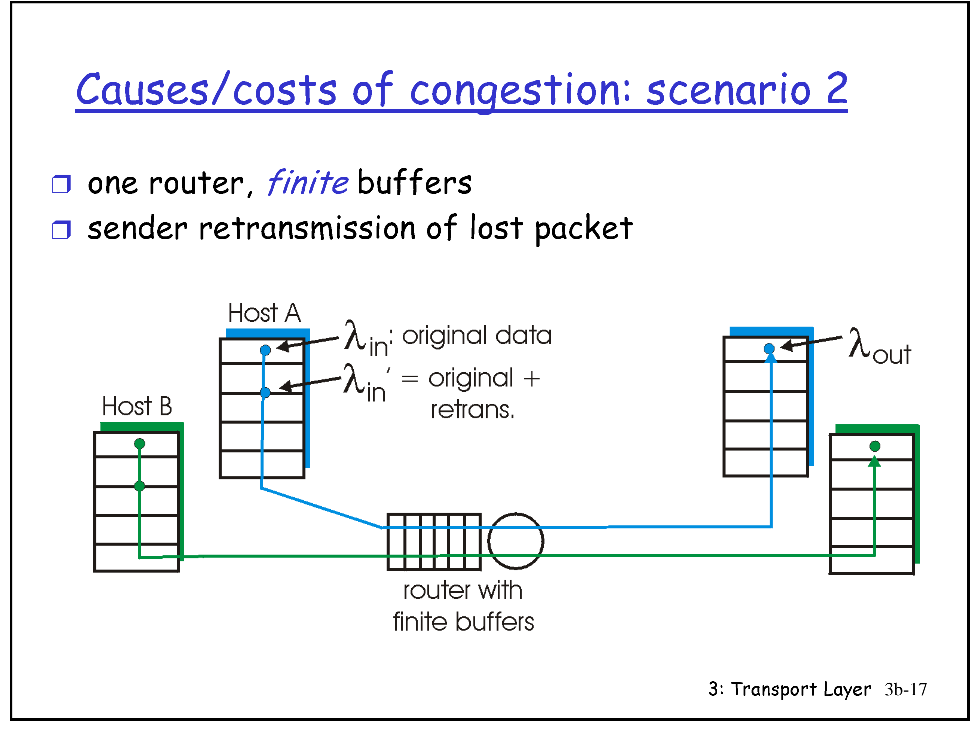
Principles of Congestion Control

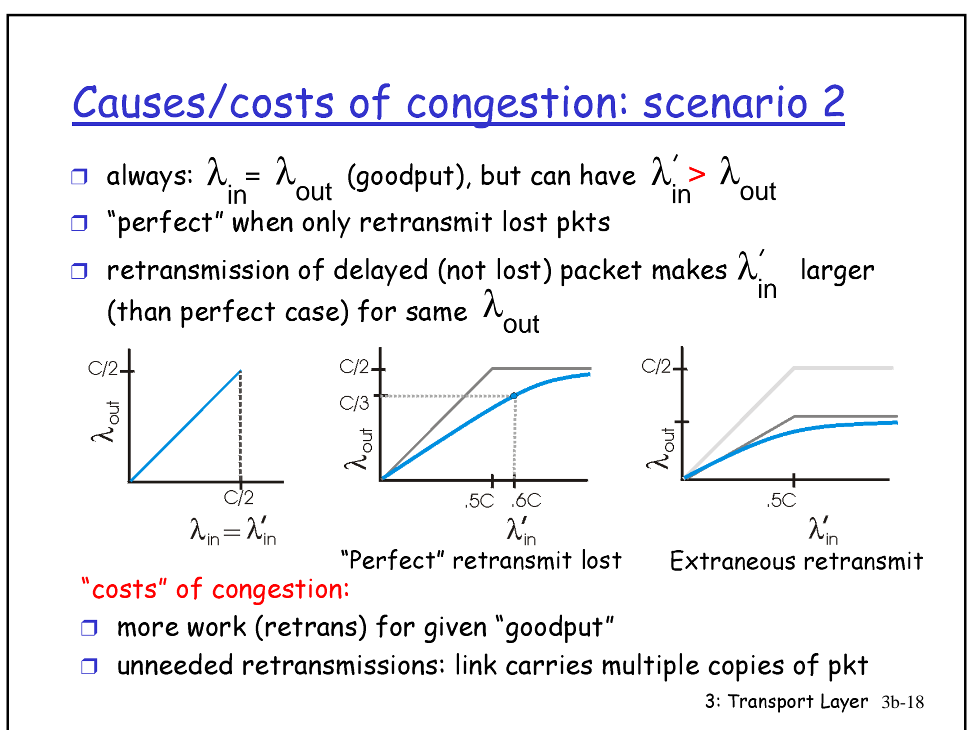
Congestion:

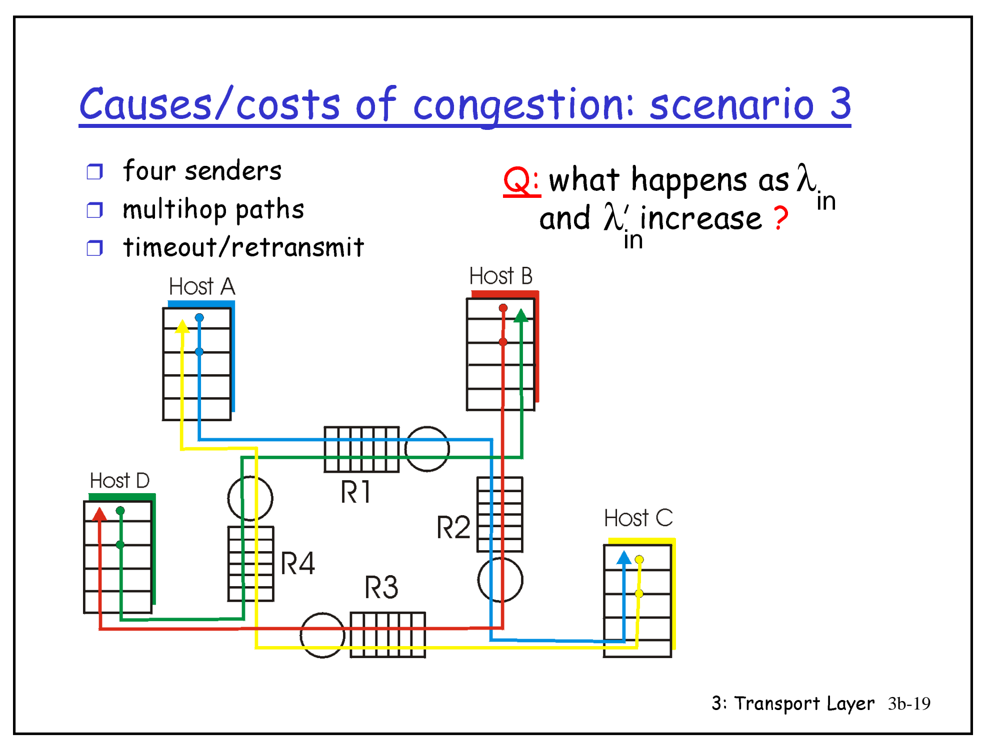
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

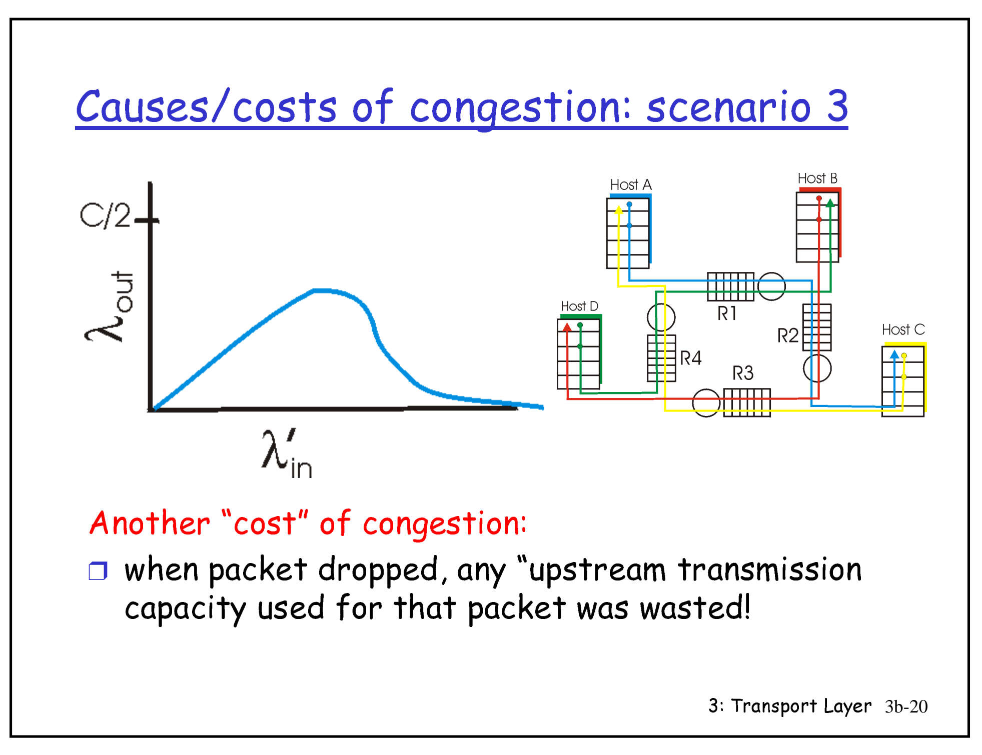
3: Transport Layer 3b-15




Causes/costs of congestion: scenario 2
one router, finite buffers
sender retransmission of lost packet
Host A
Host B
λ_{in}: original data
λ'_{in} = original + retrans.
λ_{out}
router with finite buffers
3: Transport Layer 3b-17


Causes/costs of congestion: scenario 2
always: $\lambda_{in} = \lambda_{out}$ (goodput), but can have $\lambda'_{in} > \lambda_{out}$
"perfect" when only retransmit lost pkts
retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}
C/2
λ_{out}
C/2
$\lambda_{in} = \lambda'_{in}$
C/2
C/3
λ_{out}
.5C
.6C
λ'_{in}
"Perfect" retransmit lost
C/2
λ_{out}
.5C
λ'_{in}
Extraneous retransmit
"costs" of congestion:
more work (retrans) for given "goodput"
unneeded retransmissions: link carries multiple copies of pkt
3: Transport Layer 3b-18


Causes/costs of congestion: scenario 3
four senders
multihop paths
timeout/retransmit
Q: what happens as λ_{in} and λ'_{in} increase ?
Host A
Host B
Host D
Host C
R1
R2
R3
R4
3: Transport Layer 3b-19


Causes/costs of congestion: scenario 3
C/2
λ_{out}
λ'_{in}
Host A
Host B
Host D
Host C
R1
R2
R3
R4
Another "cost" of congestion:
when packet dropped, any "upstream transmission capacity used for that packet was wasted!
3: Transport Layer 3b-20

Approaches towards congestion control

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

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

Case study: ATM ABR congestion control

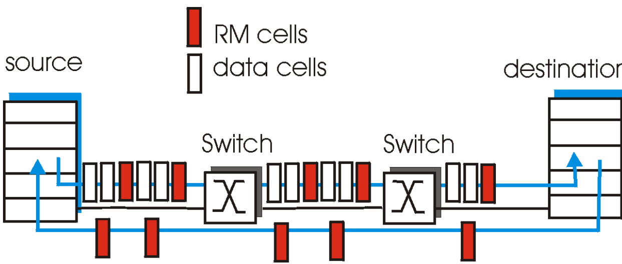
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

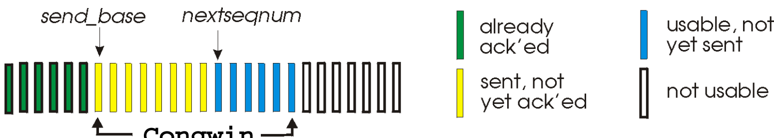
Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, CI bit set in returned RM cell

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, `Congwin`, over segments:



- w segments, each with MSS bytes sent in one RTT:

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

- Congestion Window never larger than rcvr-advertised window

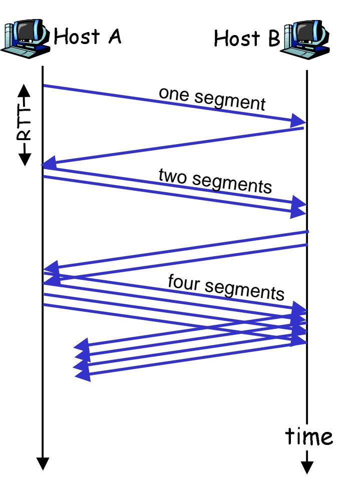
TCP congestion control:

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (`Congwin` as large as possible) without loss
  - *increase* `Congwin` until loss (congestion)
  - loss: *decrease* `Congwin`, then begin probing (increasing) again
- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - `Congwin`
  - `threshold:` defines threshold between slow start phase and congestion control phase

TCP Slowstart

Slowstart algorithm

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

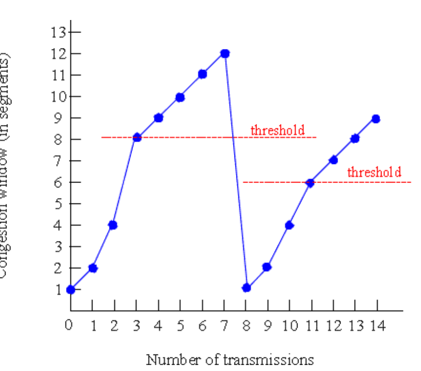


- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

TCP Congestion Avoidance (Tahoe)

Congestion avoidance

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



1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

TCP Reno

- Most of today's TCPs are Reno
- Same behavior as Tahoe on timeout
- On Triple-duplicate ACK:
  - Tahoe does nothing (no window change)
  - Reno:
    - Threshold = congwin / 2
    - congwin = congwin / 2

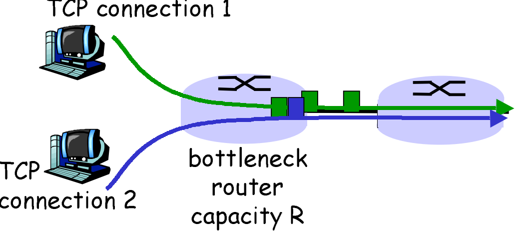
AIMD

TCP congestion avoidance:

- AIMD: *additive increase, multiplicative decrease*
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event

TCP Fairness

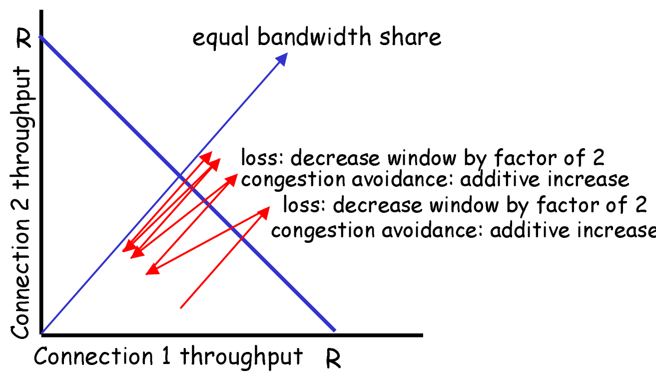
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

Two competing sessions:

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



Effects of TCP latencies

Q: client latency from object request from WWW server to receipt?

- TCP connection establishment
- data transfer delay

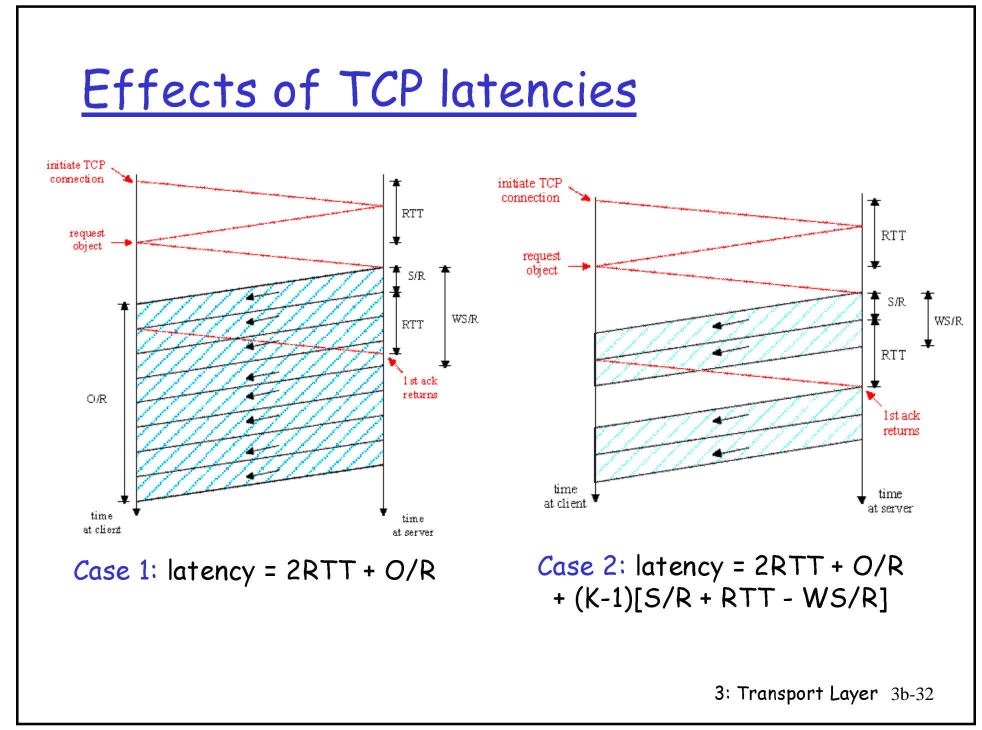
Notation, assumptions:

- Assume: fixed congestion window, W, giving throughput of R bps
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Two cases to consider:

- WS/R > RTT + S/R: ACK for first segment in window before window's worth of data sent
- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

3: Transport Layer 3b-31



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"