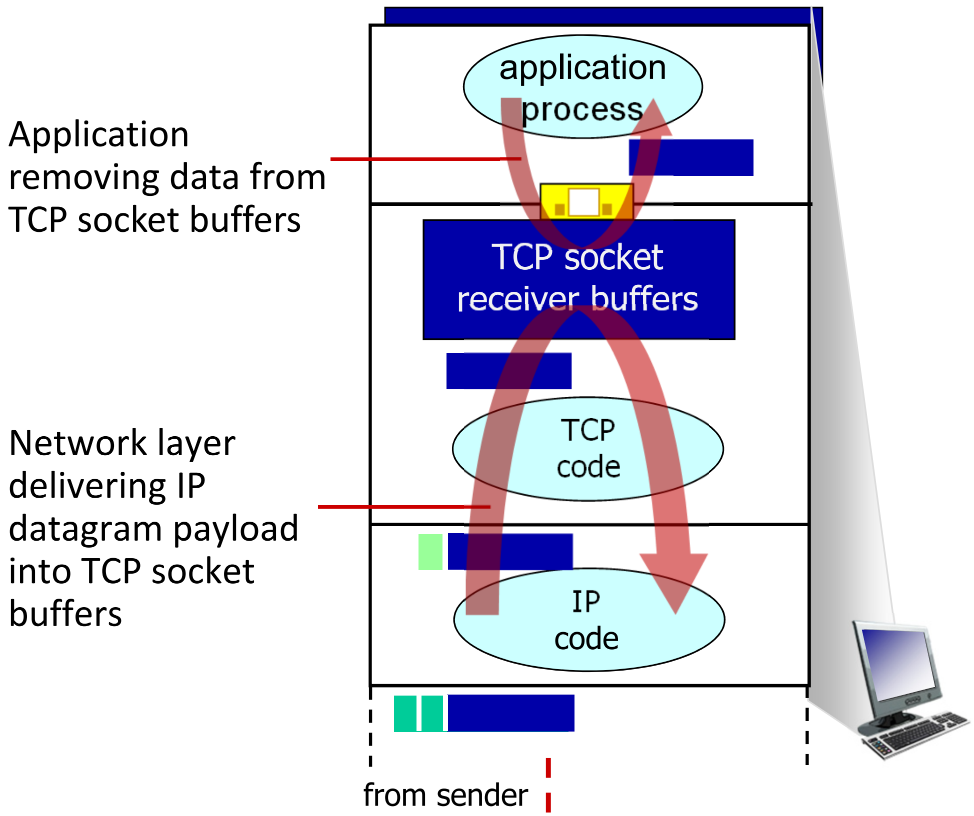
Chapter 3: roadmap

- 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

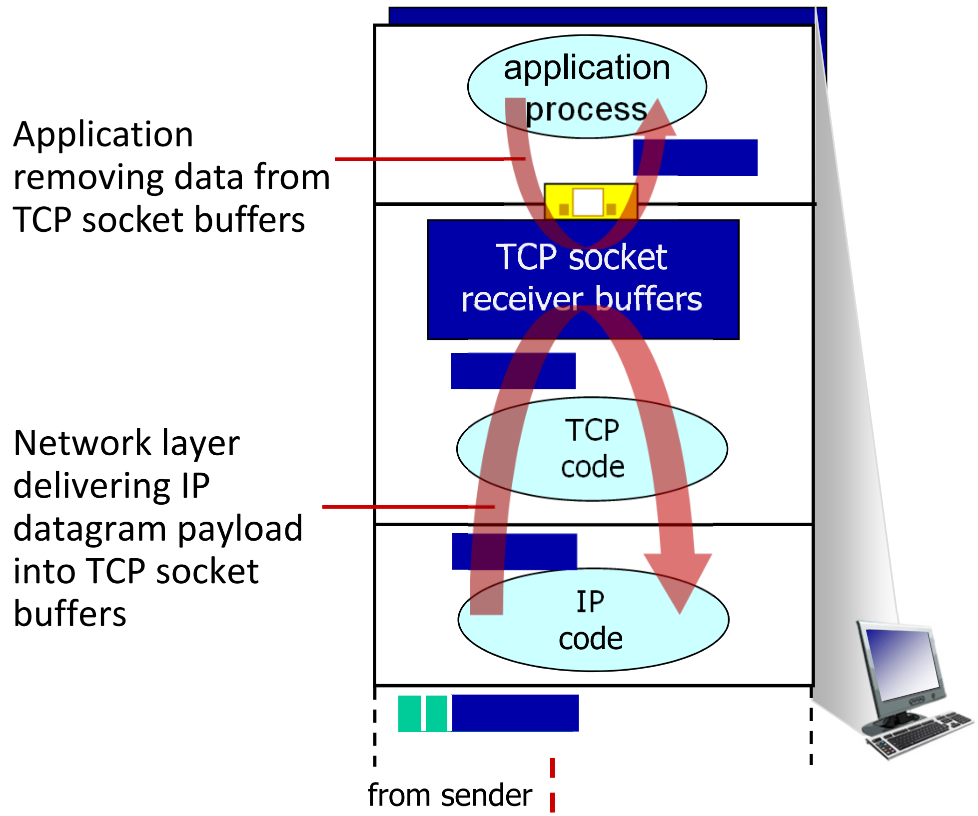
- At network and transport layers: payloads of new coming (3rd-layer) datagrams are
  - brought up to the transport layer
  - saved into TCP socket buffer
- at application layer: an application process
  - performs socket reads
  - removes data from TCP socket buffer



receiver protocol stack

TCP flow control

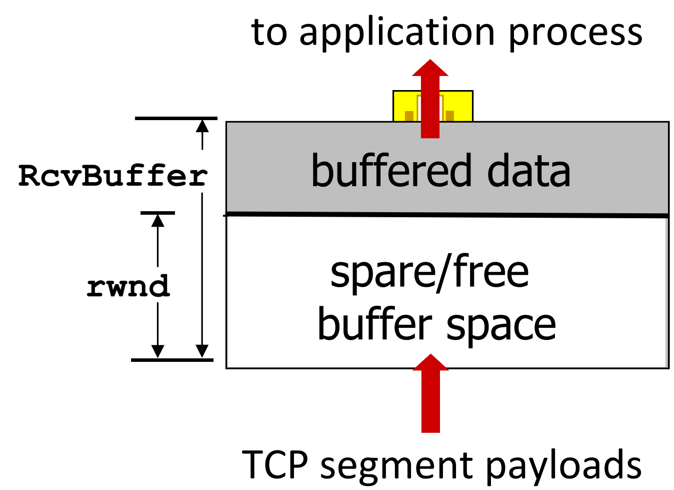
- what if network layer delivers data faster than application layer removes data from socket buffers?
  - buffer overflow
    - retransmissions
- flow control
  - receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

TCP flow control

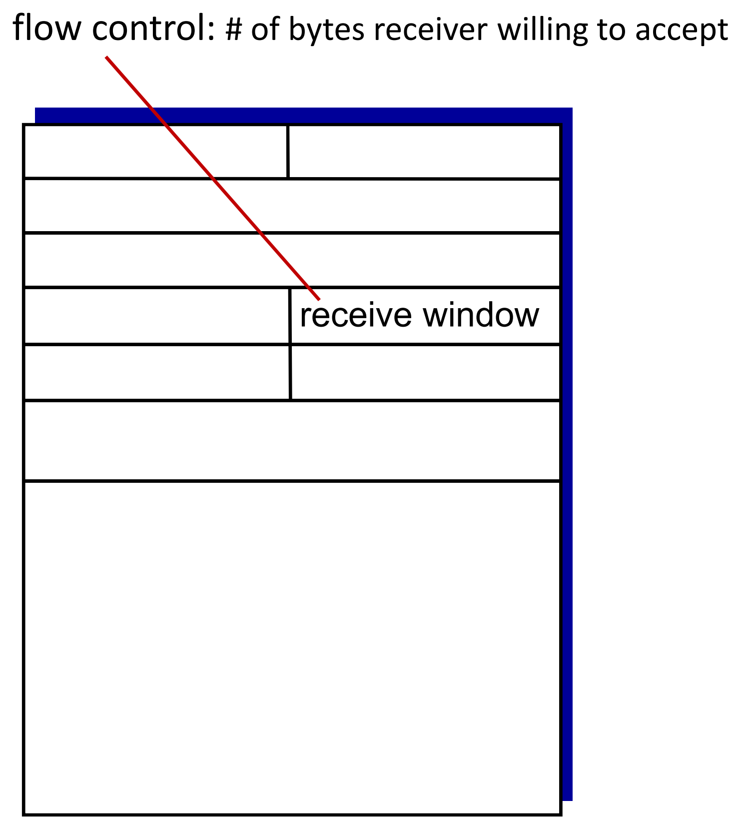
- **`RcvBuffer`** (receive buffer)
  - data can be buffered here
  - spare/free buffer space is *receive window* or **`rwnd`**
- the size of **`rwnd`** changes dynamically
  - buffer overflow should be avoided
  - but how?



TCP receiver-side buffering

TCP flow control

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

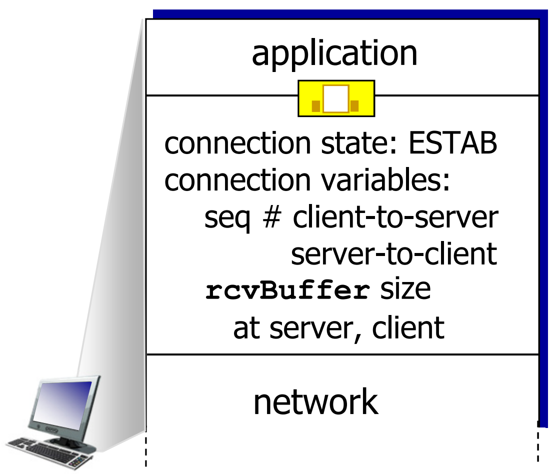


TCP segment format

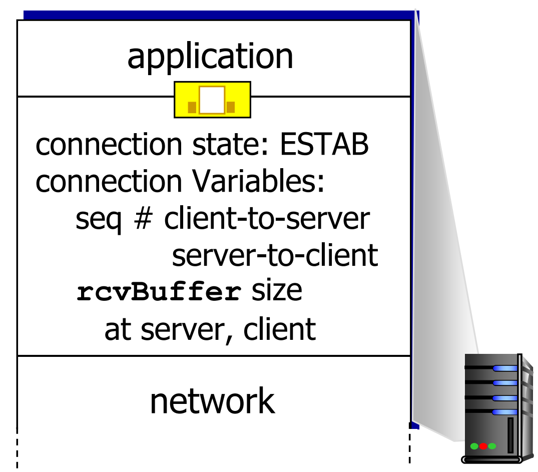
TCP connection management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., initial seq #)



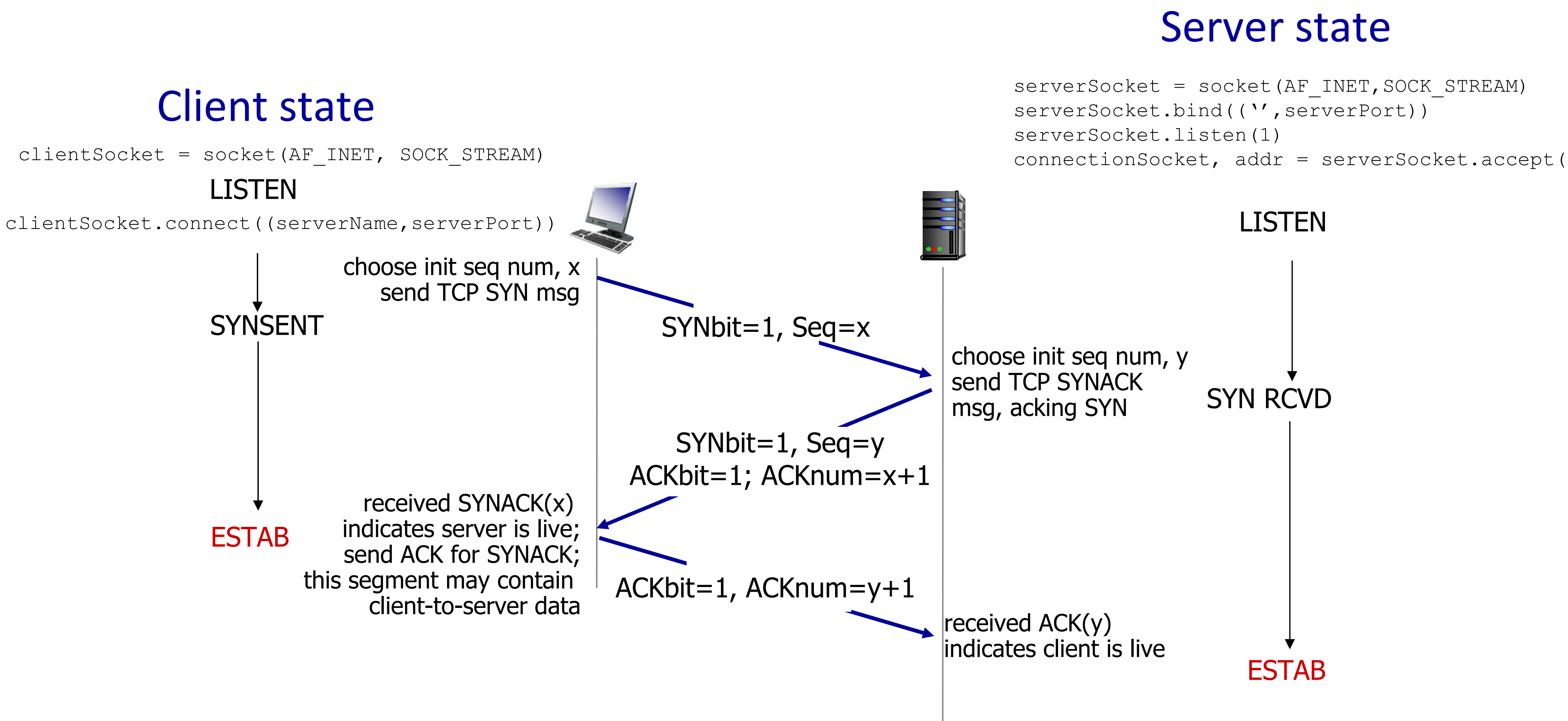

```
Socket clientSocket =
  newSocket("hostname","port number");
```




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

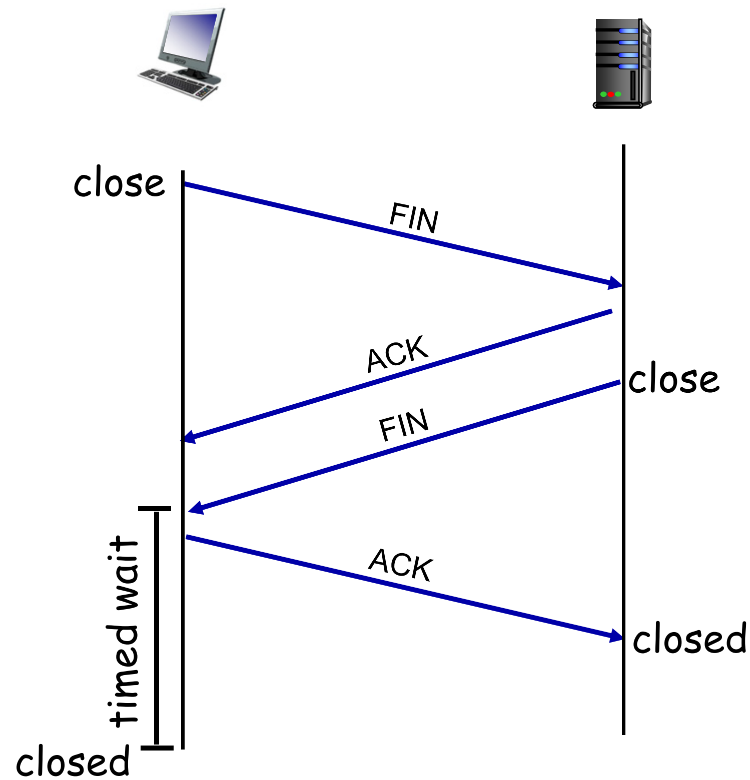
A human 3-way handshake protocol

TCP 3-way handshake



Closing a TCP 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



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

Principles of congestion control

Congestion:

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

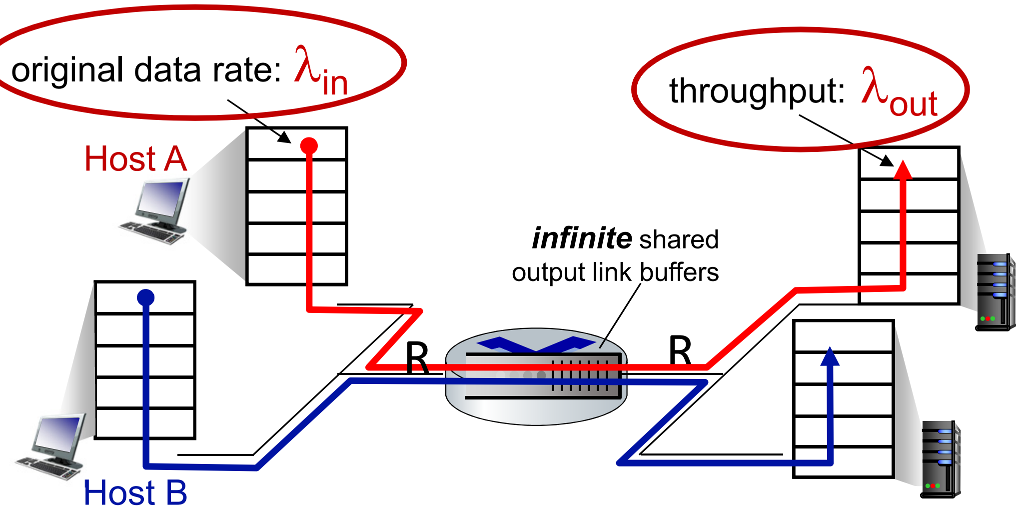
congestion control:
too many senders, sending too fast

flow control:
one sender too fast for one receiver

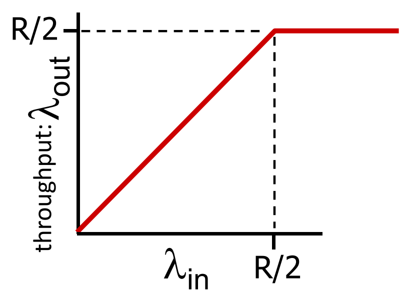
Causes/costs of congestion: scenario 1

Simplest scenario:

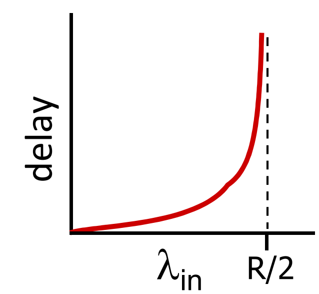
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed
  - $\lambda'_{in} = \lambda_{in}$



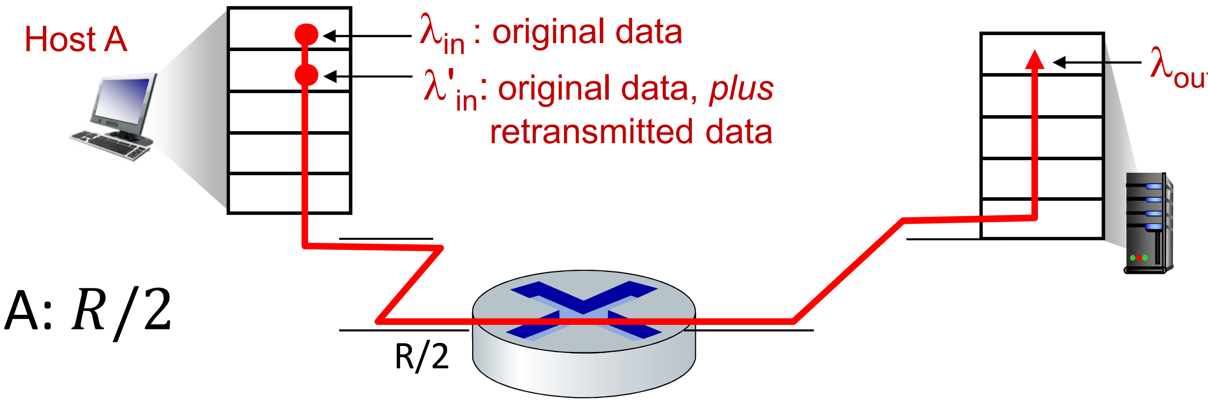
Q: What happens as arrival rate λ_{in} approaches R/2?



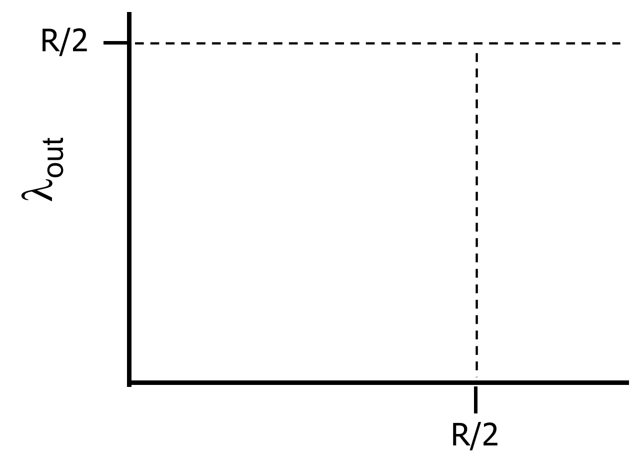
maximum per-connection throughput: R/2



large delays as arrival rate λ_{in} approaches capacity



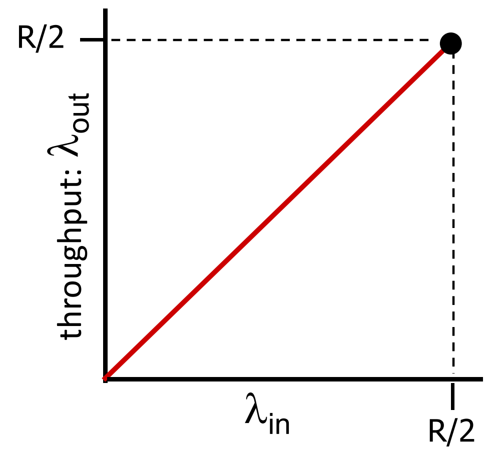
- bandwidth allocated to Host A: $R/2$
- throughput (at receiver): λ_{out}
  - $\lambda_{out} \leq R/2$
- original data rate: λ_{in}
  - $\lambda_{out} \leq \lambda_{in}$
- offered load: λ'_{in}
  - data rate including *retransmissions*
  - $\lambda'_{in} \geq \lambda_{in}$

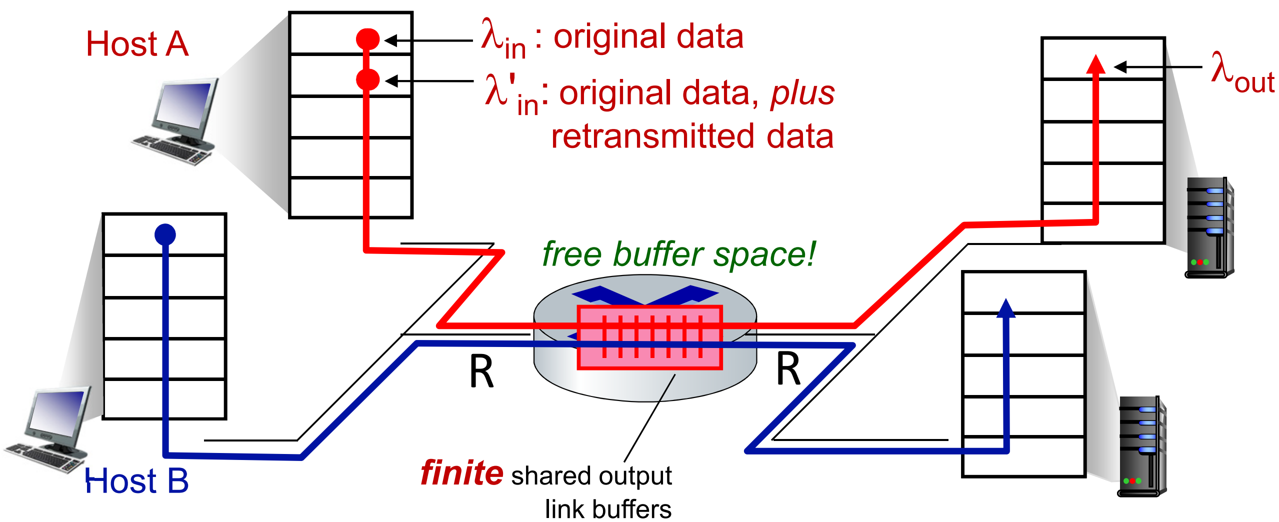


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available
  - no packet drop at router, no retransmission at sender
  - $\lambda'_{in} = \lambda_{in}$
  - $\lambda_{out} = \lambda'_{in} = \lambda_{in}$

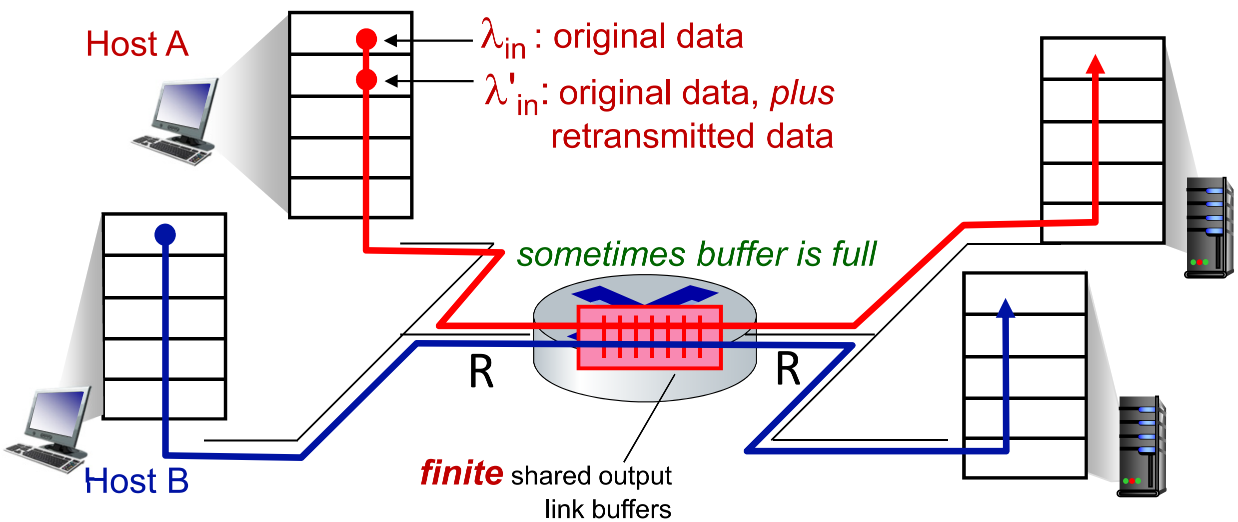


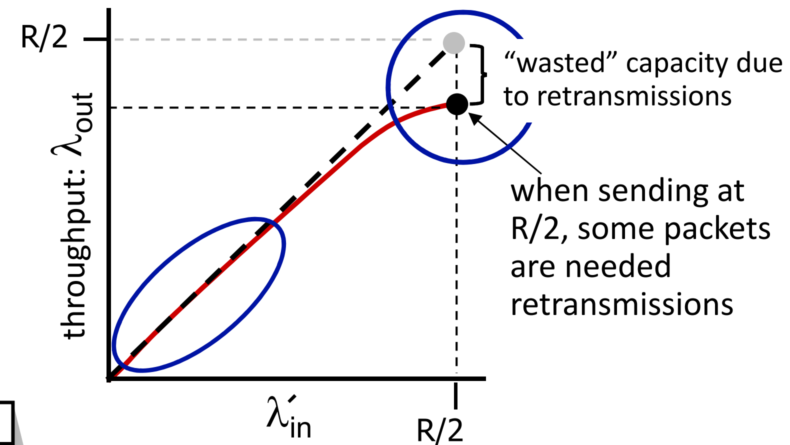


Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- a part of packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends the packets *known* to be lost

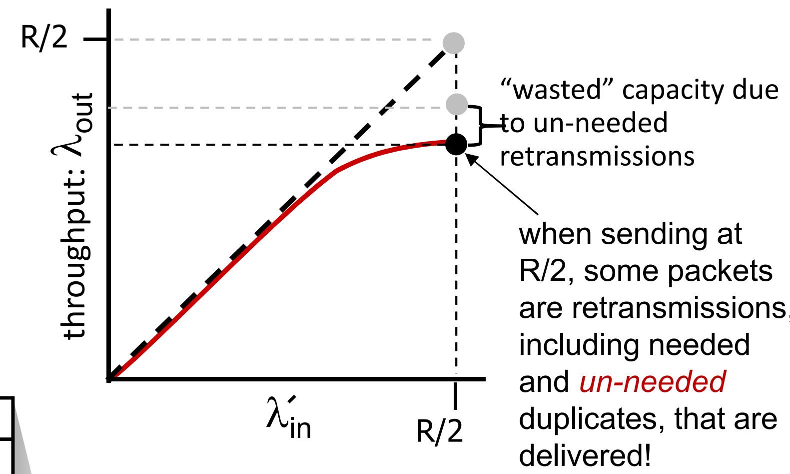


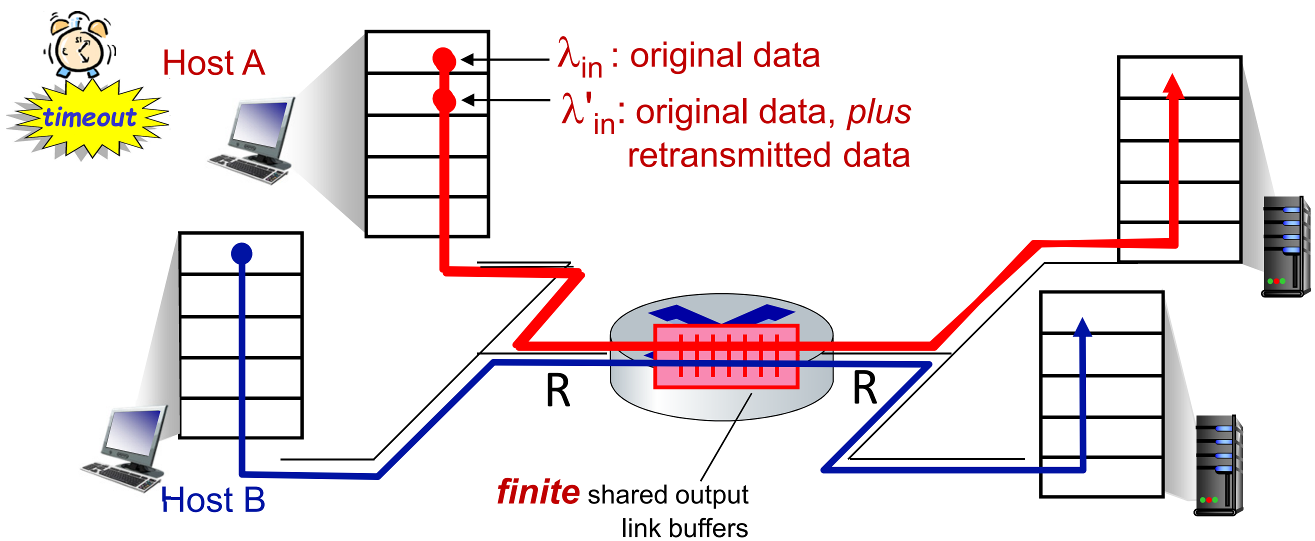


Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- Besides retransmissions caused by packet drops at router due to buffer overflow,
- sender sometimes can time out prematurely, sending *two* copies, *both* of which are delivered



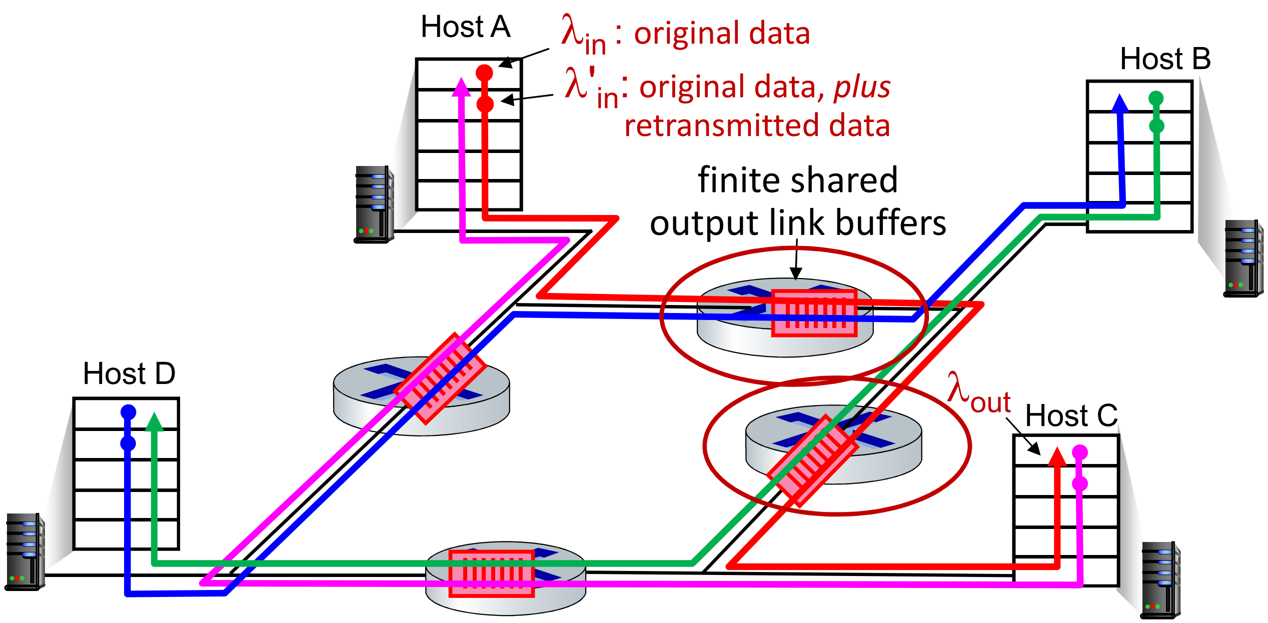


Causes/costs of congestion: scenario 3

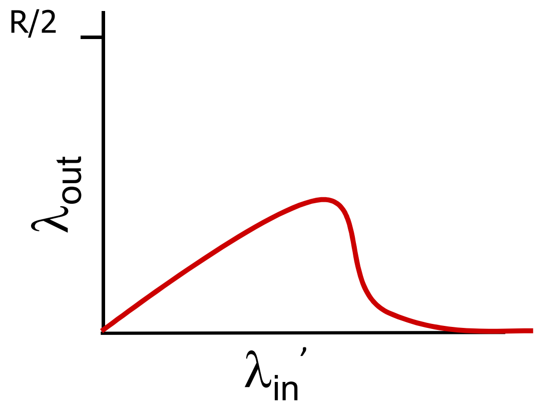
- *four* senders
- *multi-hop* paths
- timeout/retransmit

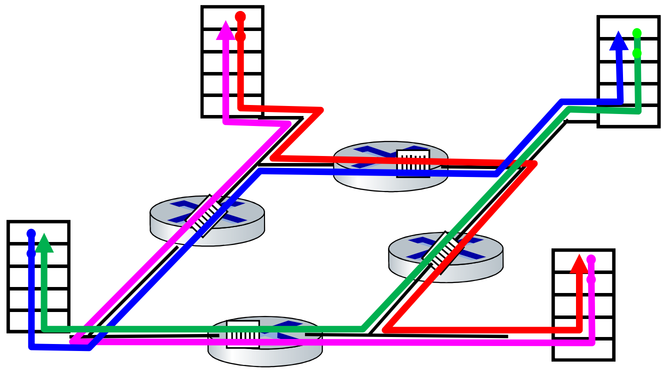
Q: what happens as λ_{in} and λ'_{in} increase ?

A: as red λ'_{in} increases, all blue pkts arriving at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3



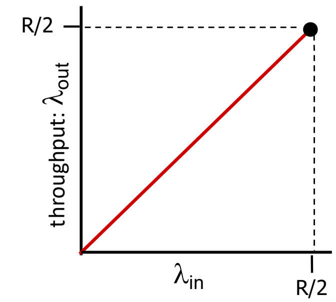


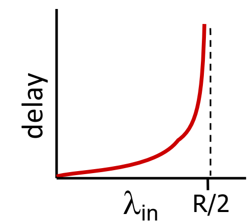
another "cost" of congestion:

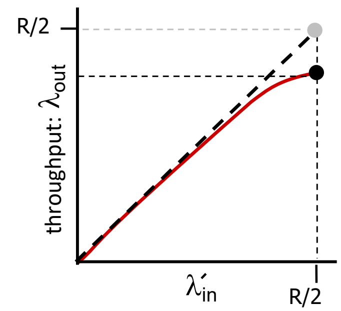
- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

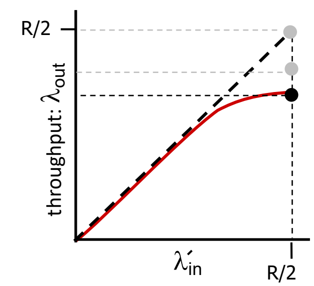
Causes/costs of congestion: insights

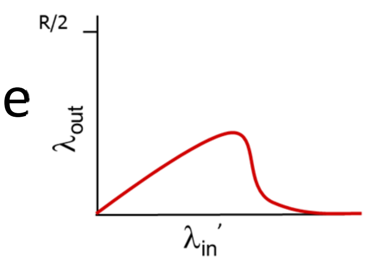
- (effective) throughput (λ_{out}) can't exceed
  - the capacity
  - the original data rate (λ_{in})
- delay increases as λ_{in} increases
- packet loss causes retransmission, which
  - increases the offer load $\lambda'_{in} = \lambda_{in} + \lambda_{\mathrm{reTX}}$
  - decreases (effective) throughput due to loss
    - un-needed duplicates further decreases (effective) throughput λ_{out}
- (upstream) transmission capacity / buffering wasted for packets lost in the downstream

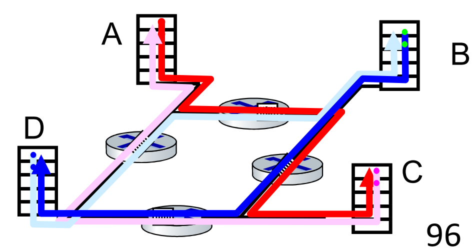








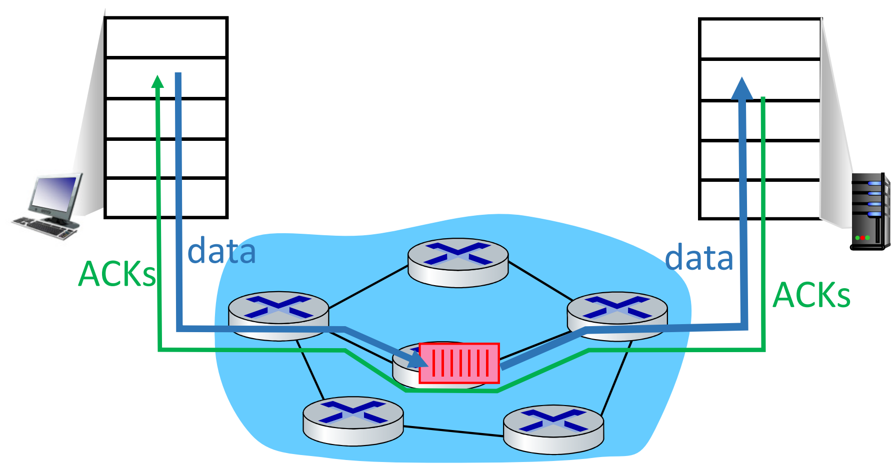




Approaches towards congestion control

End-to-end congestion control:

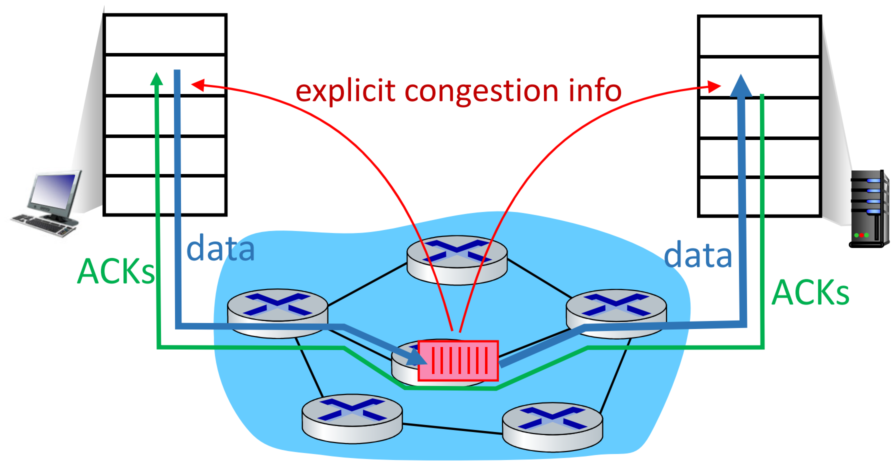
- no explicit feedback from network
- congestion *inferred* from
  - observed loss, overlong delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- congested routers provide *direct* feedback to sending/receiving hosts
  - may indicate congestion level or explicitly set sending rate



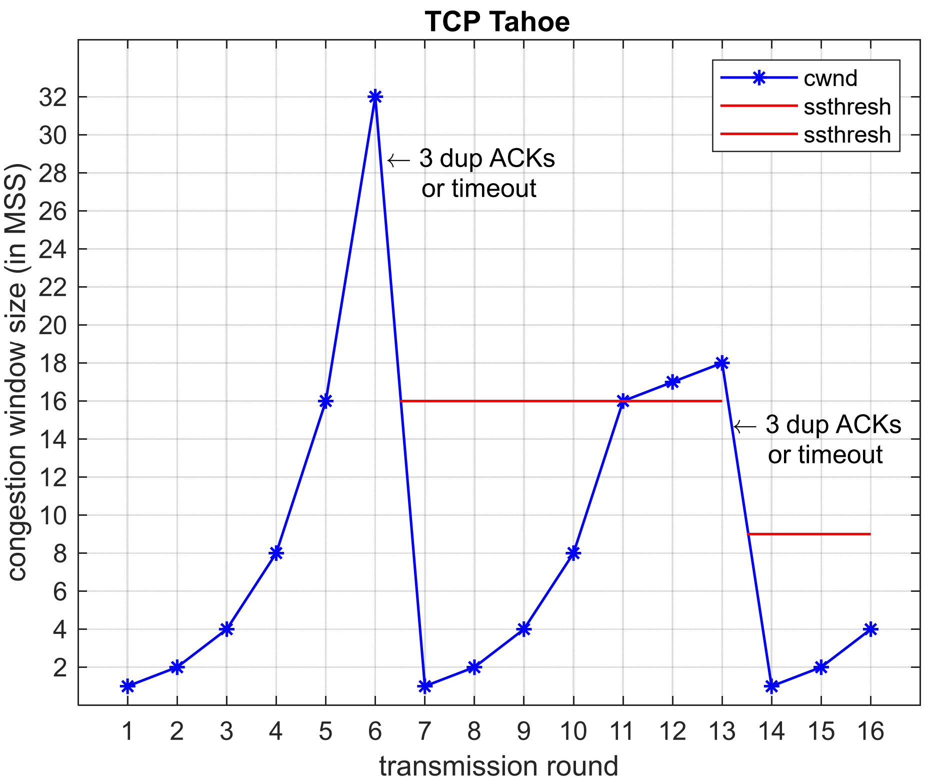
- IP ECN (explicit congestion notification) & TCP ECE (ECN-Echo)
  - router sets a mark (in IP header) to signal congestion
  - receiver echoes back (in TCP header) to sender
  - sender reduces its transmission rate as for a packet drop

Chapter 3: roadmap

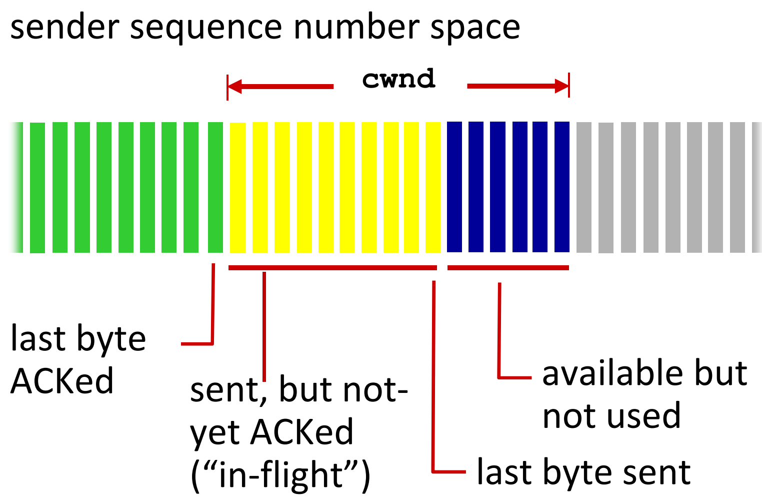
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

TCP congestion control: overview

- sender adjusts `cwnd` dynamically to limit the tx rate (`cwnd` ∝ rate)
- sender perceives congestion by
  - timeout and 3 duplicate ACKs
- TCP Tahoe:
  - "slow start" phase
    - start with `cwnd` = 1 (MSS)
    - if no loss occurs, double `cwnd` every RTT (until `cwnd` reaches `ssthresh` and goes to "congestion avoidance")
    - if loss occurs, reset `cwnd` = 1
  - "congestion avoidance" phase
    - if no loss occurs, `cwnd` grows linearly
    - if loss occurs, go to "slow start"



TCP congestion window: details

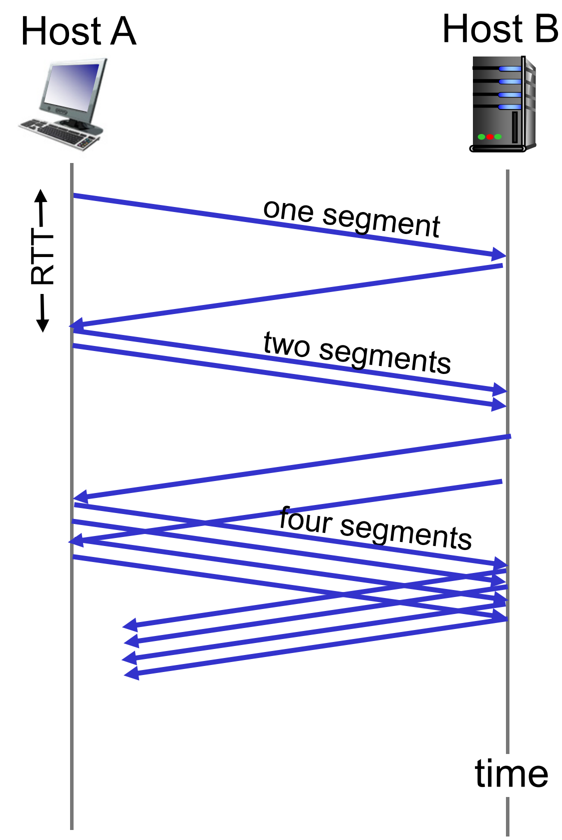


- TCP sender limits transmission: `LastByteSent - LastByteAcked ≤ cwnd`
  - *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

TCP slow start: details

- 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
- initial rate is slow, but ramps up exponentially fast



Improvement for congestion avoidance: AIMD

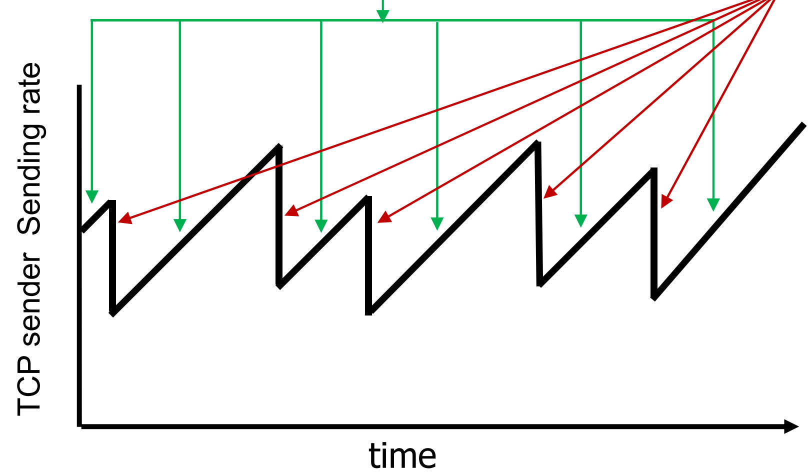
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event in a AIMD manner

Additive Increase

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

Multiplicative Decrease

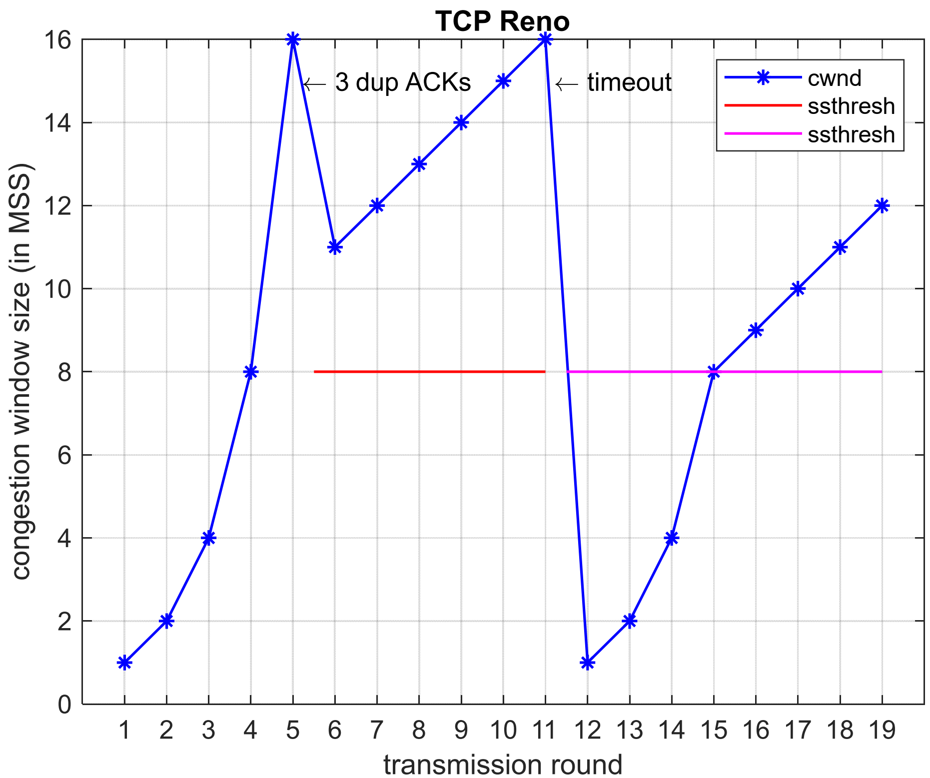
cut sending rate in half at each loss event



TCP Reno

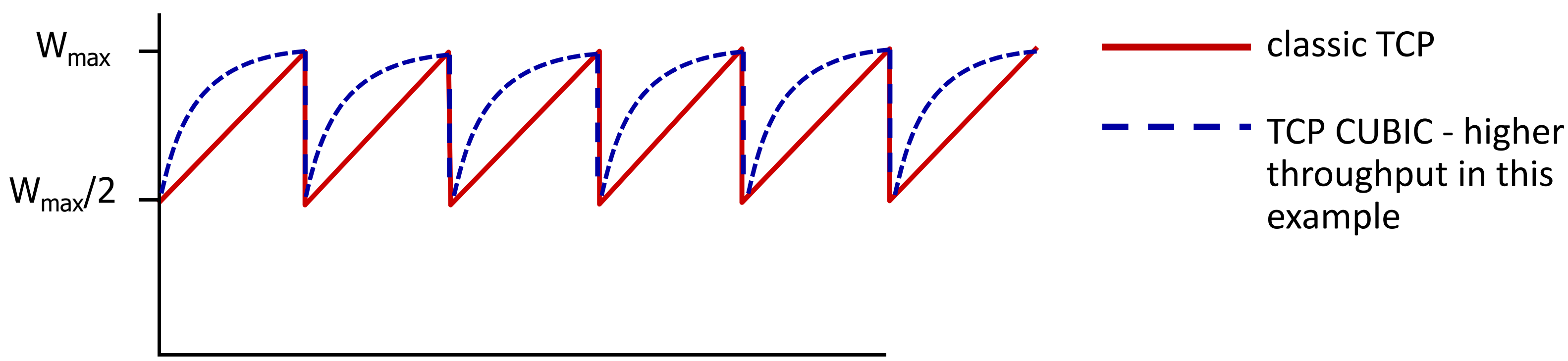
TCP Reno: cwnd is

- Cut to 1 MSS when loss detected by **timeout**
  - then enter “slow start”
  - $\text{ssthresh} \leftarrow \text{cwnd}/2$
    $\text{cwnd} \leftarrow 1 \cdot \text{MSS}$
- Cut in half on loss detected by **3 duplicate ACK**
  - then enter “congestion avoidance”
  - more precisely,
    $\text{ssthresh} \leftarrow \text{cwnd}/2$
    $\text{cwnd} \leftarrow \text{cwnd}/2 + 3 \cdot \text{MSS}$



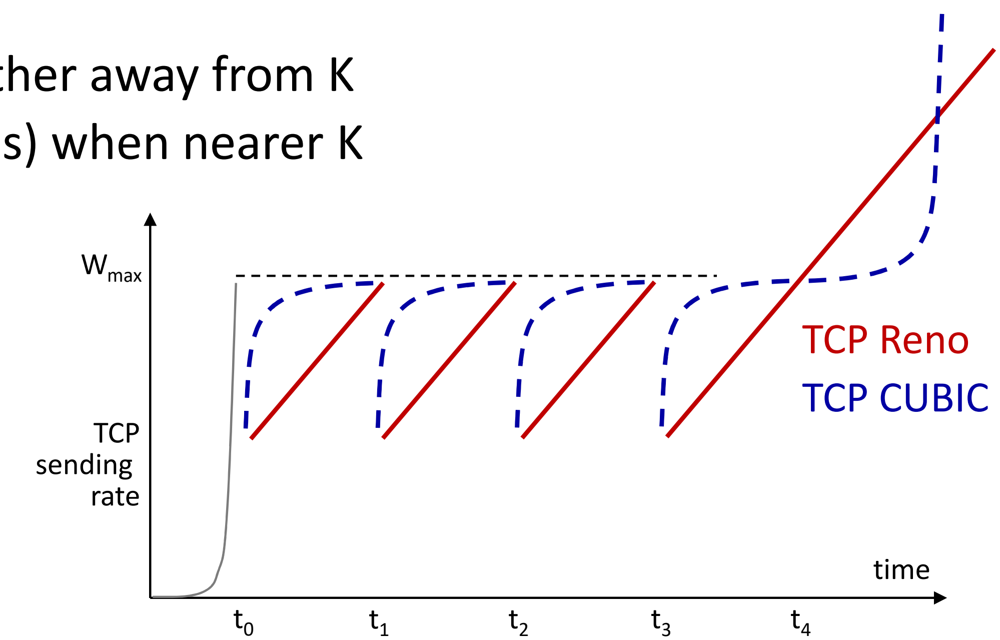
TCP CUBIC (default in Linux)

- 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*



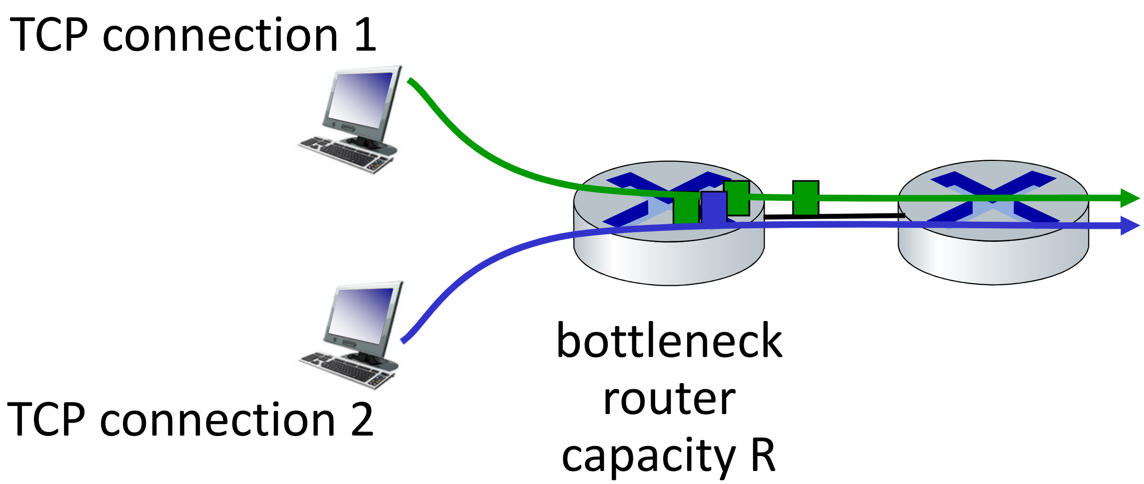
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- 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 fairness

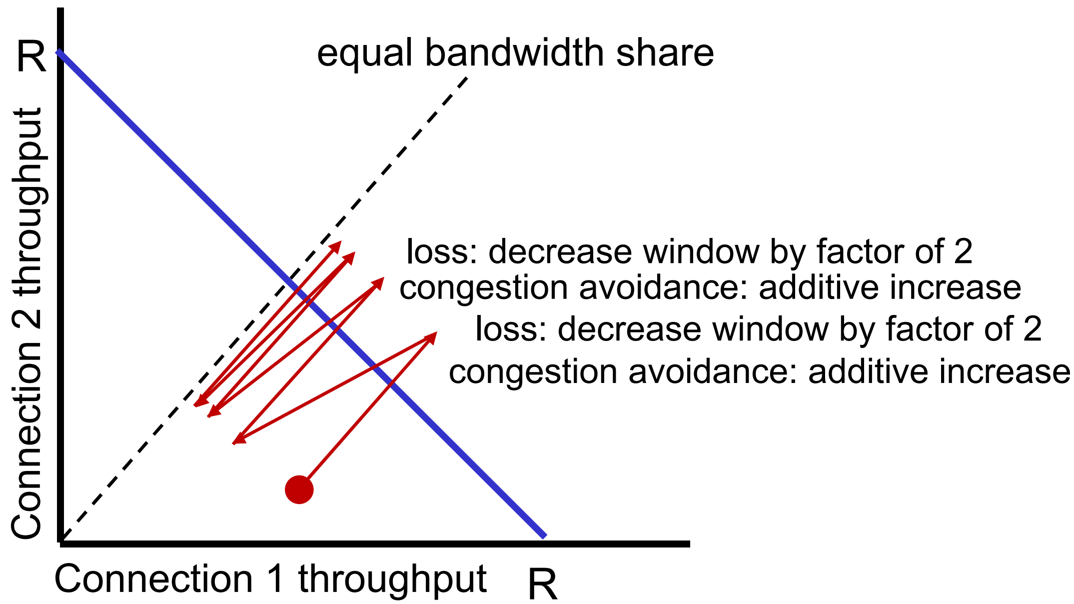
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Reno Fair?

Example: two competing TCP connections:

- 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 1R/10
  - new app asks for 11 TCPs, gets 11R/20