

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

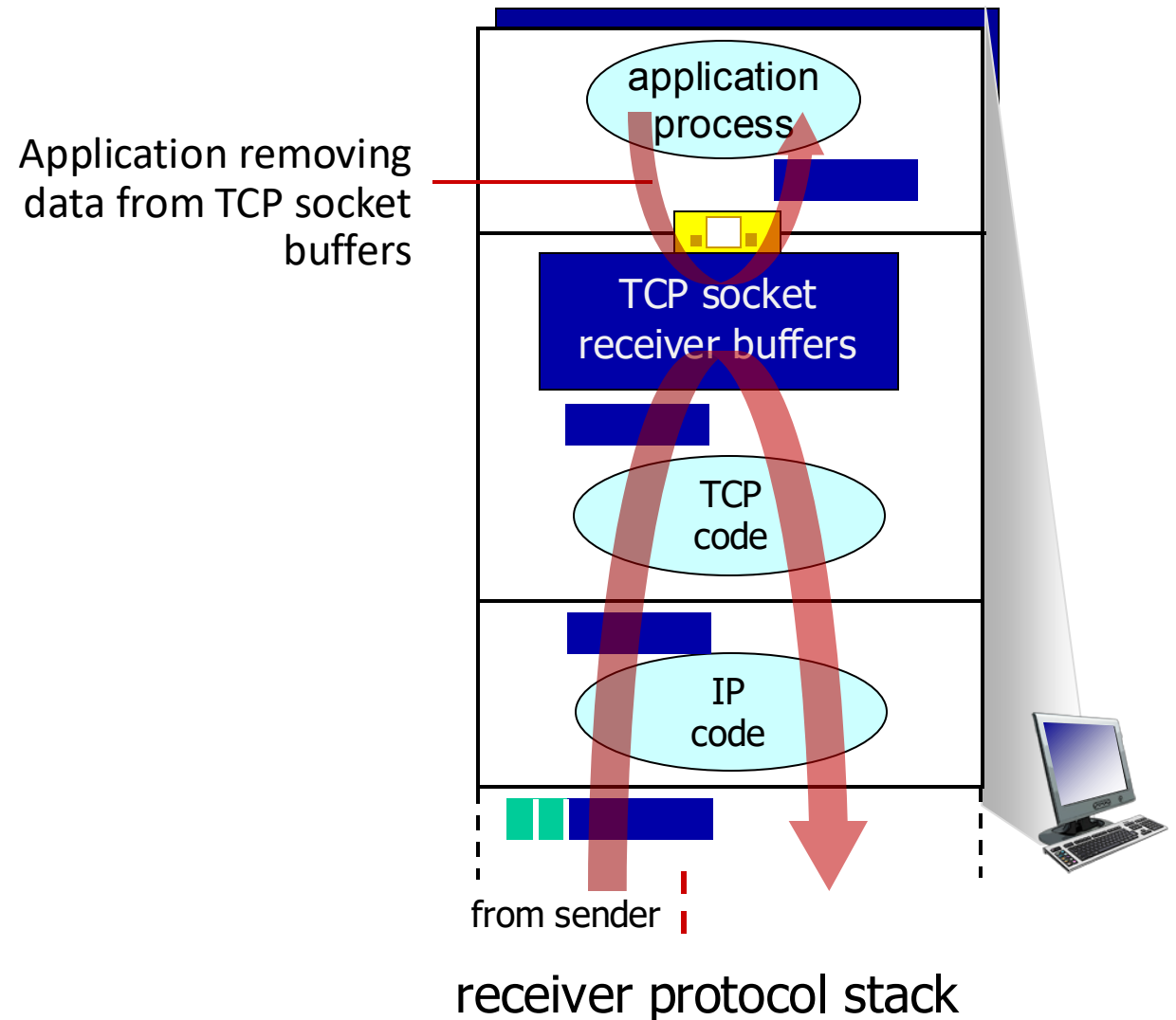


TCP flow control

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

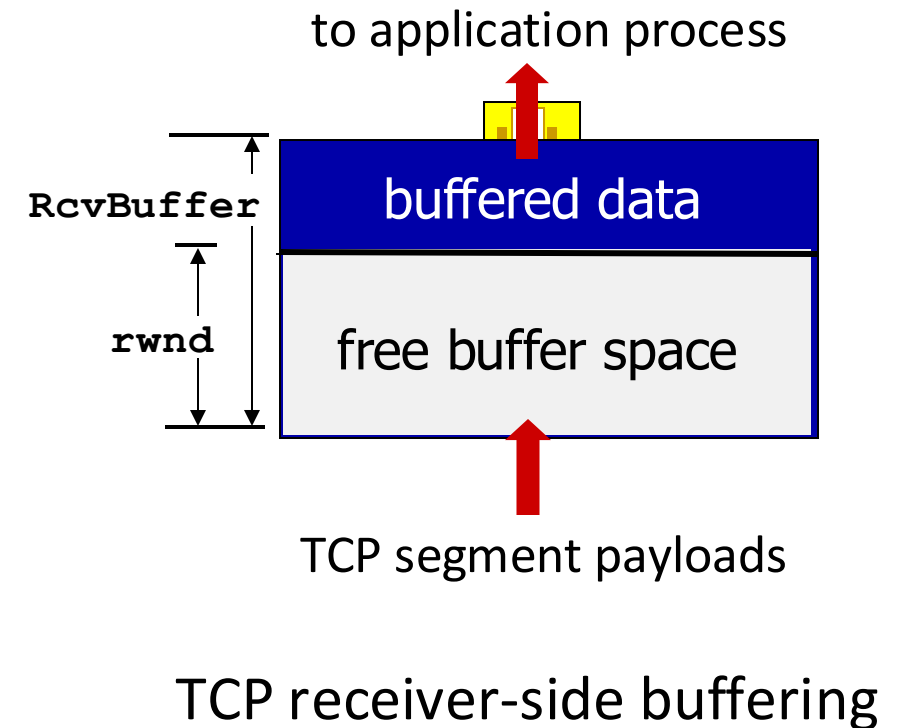
—flow control—

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



TCP flow control

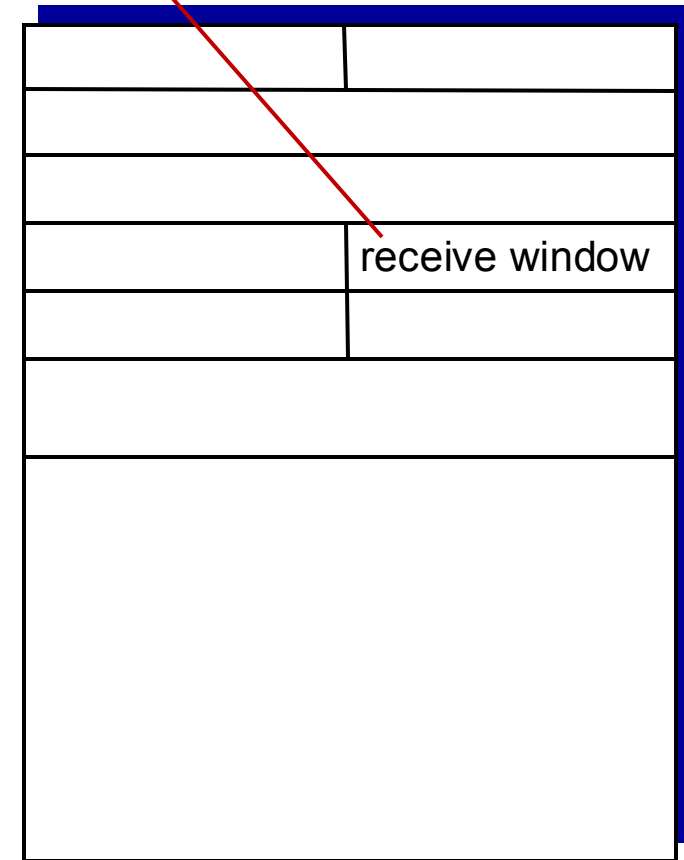
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
 - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
 - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive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



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 autoadjust **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

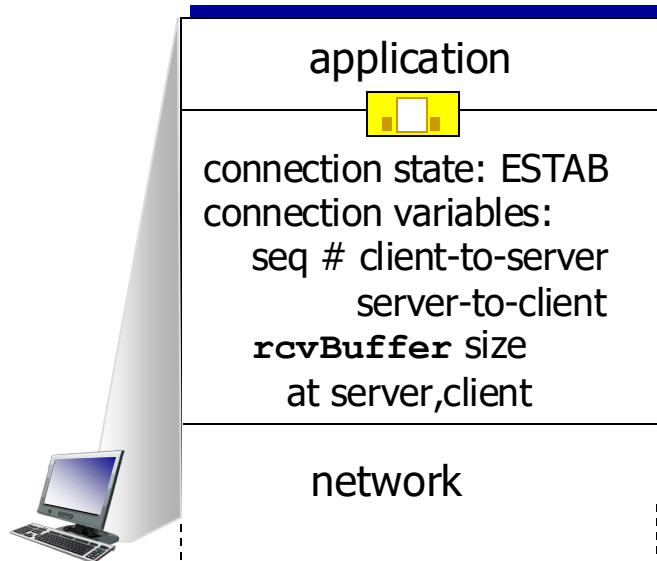


TCP segment format

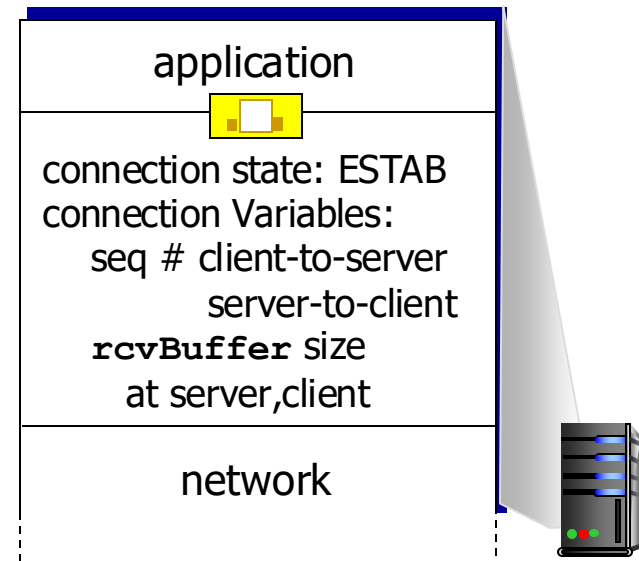
TCP connection management

before exchanging data, sender/receiver “handshake”:

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



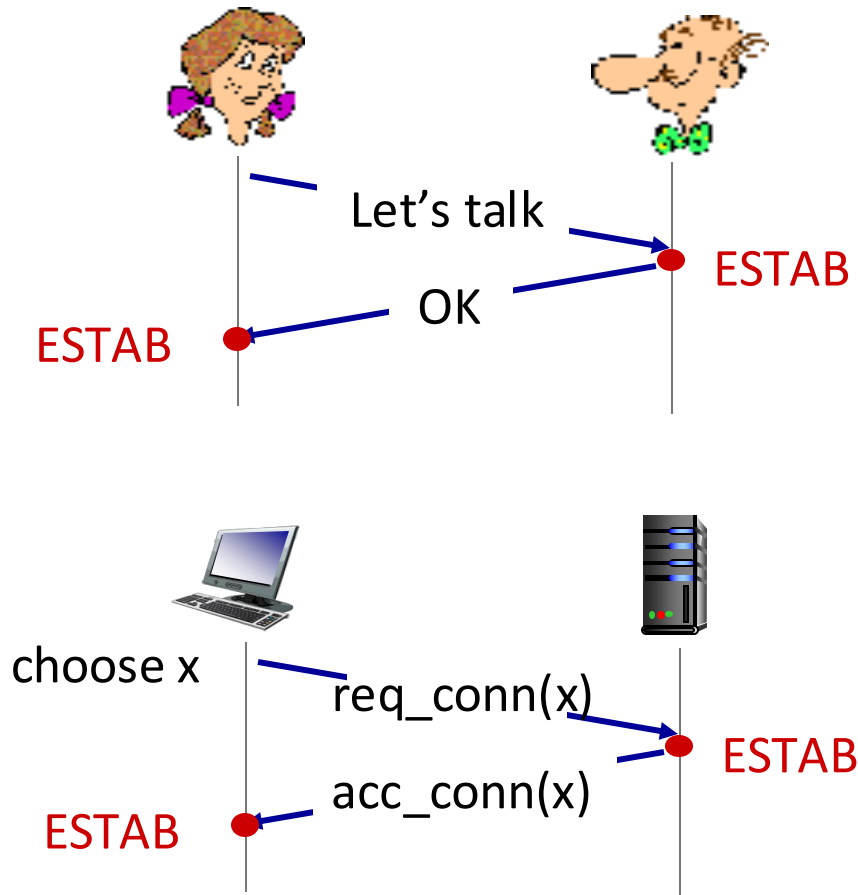
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

Agreeing to establish a connection

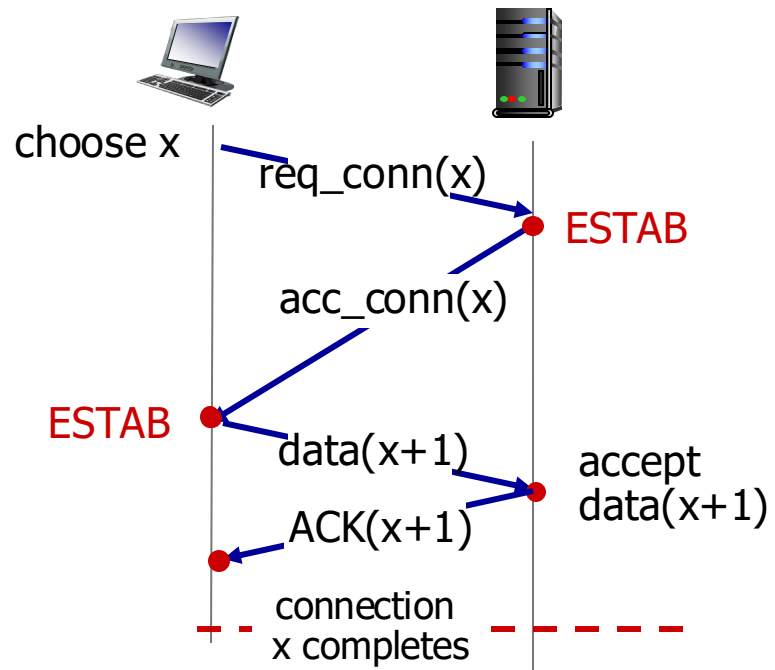
2-way handshake:



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

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

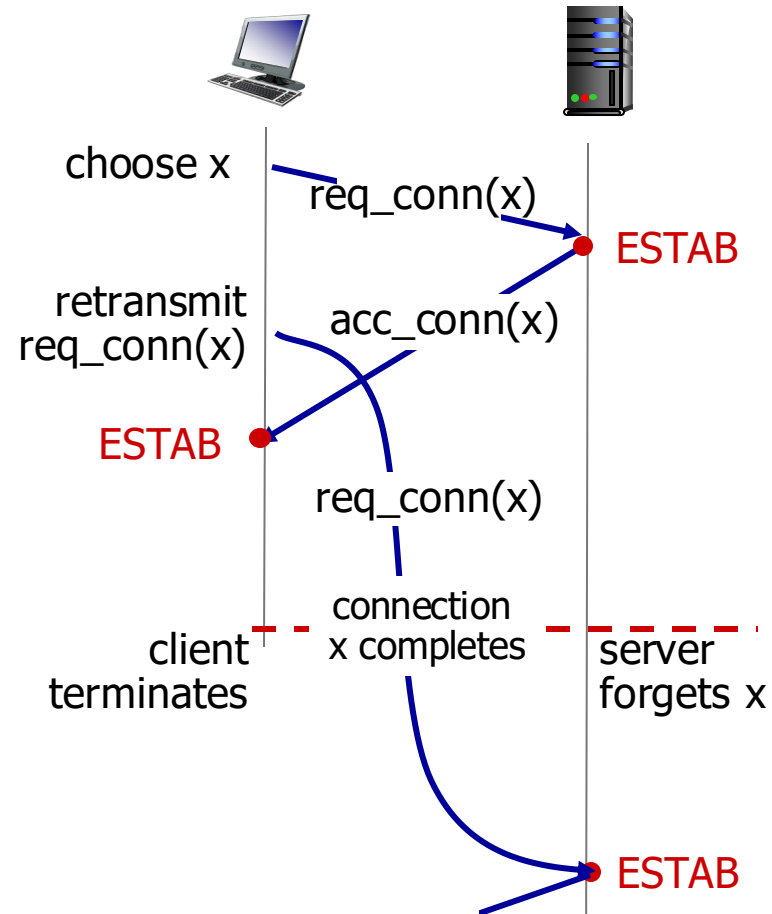
2-way handshake scenarios



No problem!

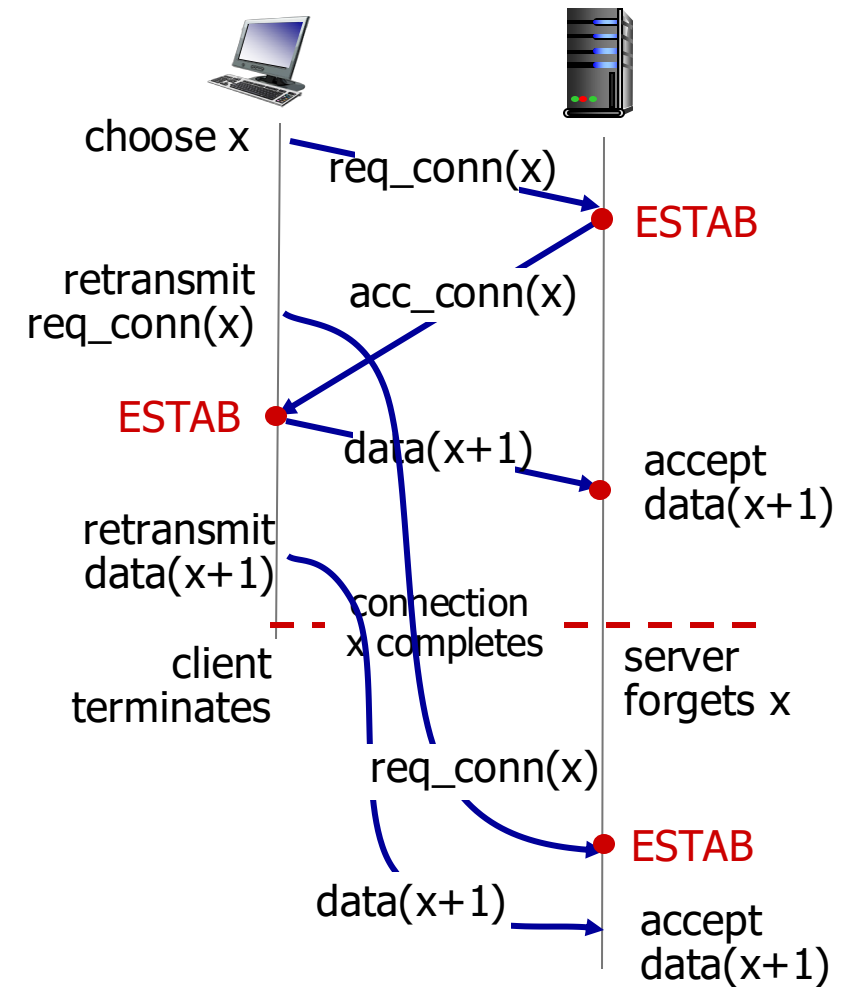


2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios



 Problem: dup data accepted!

TCP 3-way handshake

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data



SYNbit=1, Seq=x

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1



choose init seq num, y
send TCP SYNACK
msg, acking SYN

received ACK(y)
indicates client is live

Server state

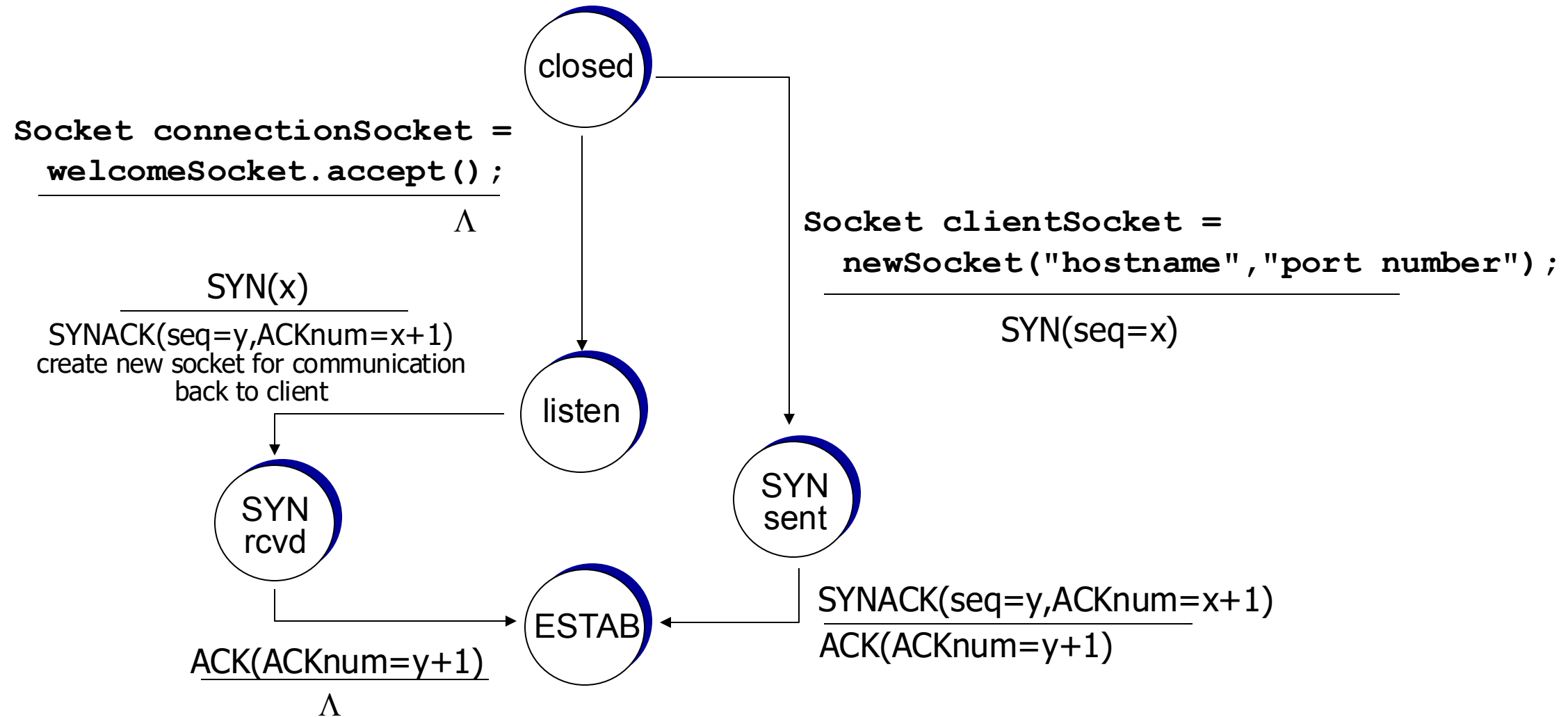
```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(('', serverPort))  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

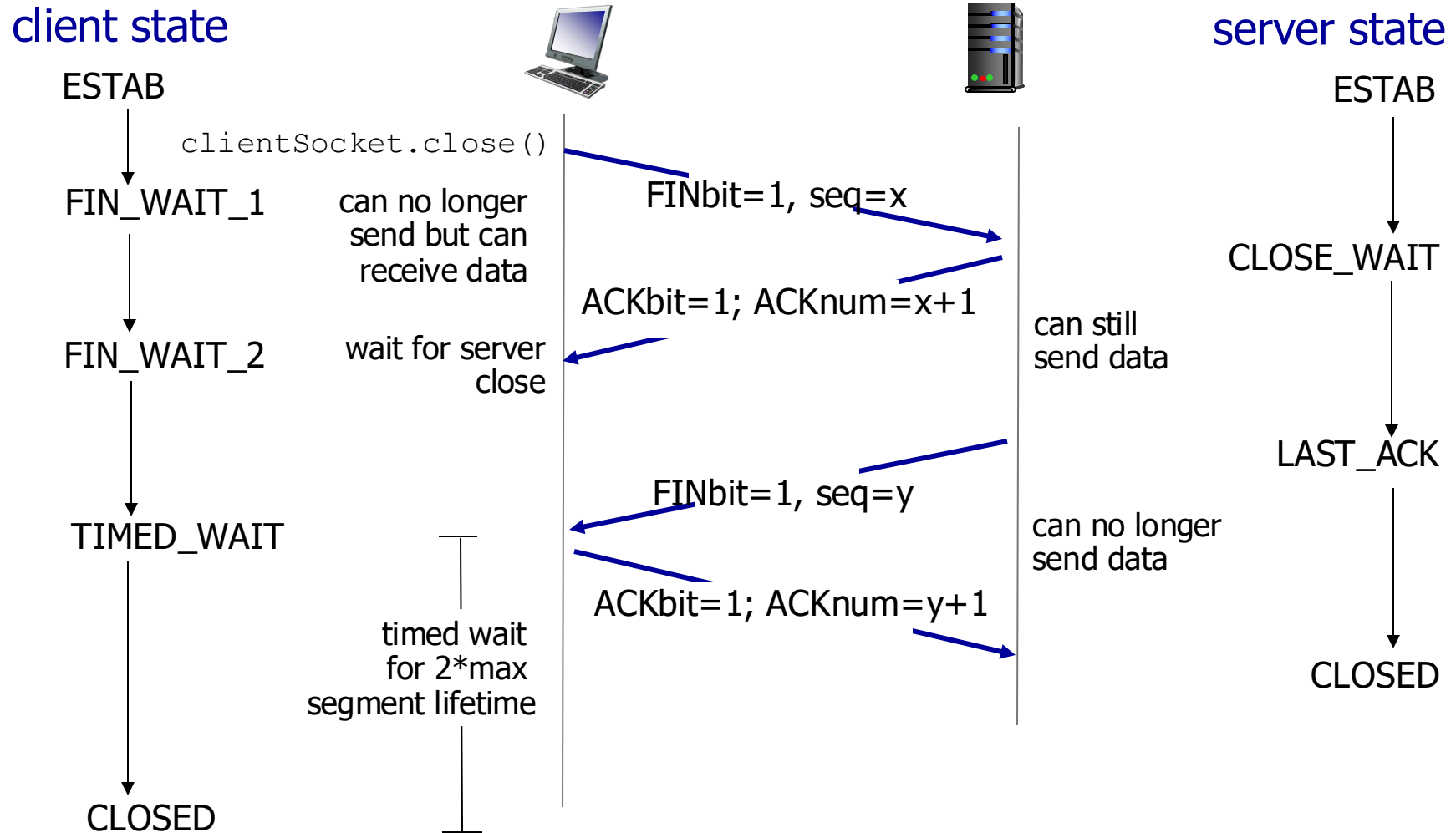
TCP 3-way handshake FSM



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

Closing a TCP connection



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

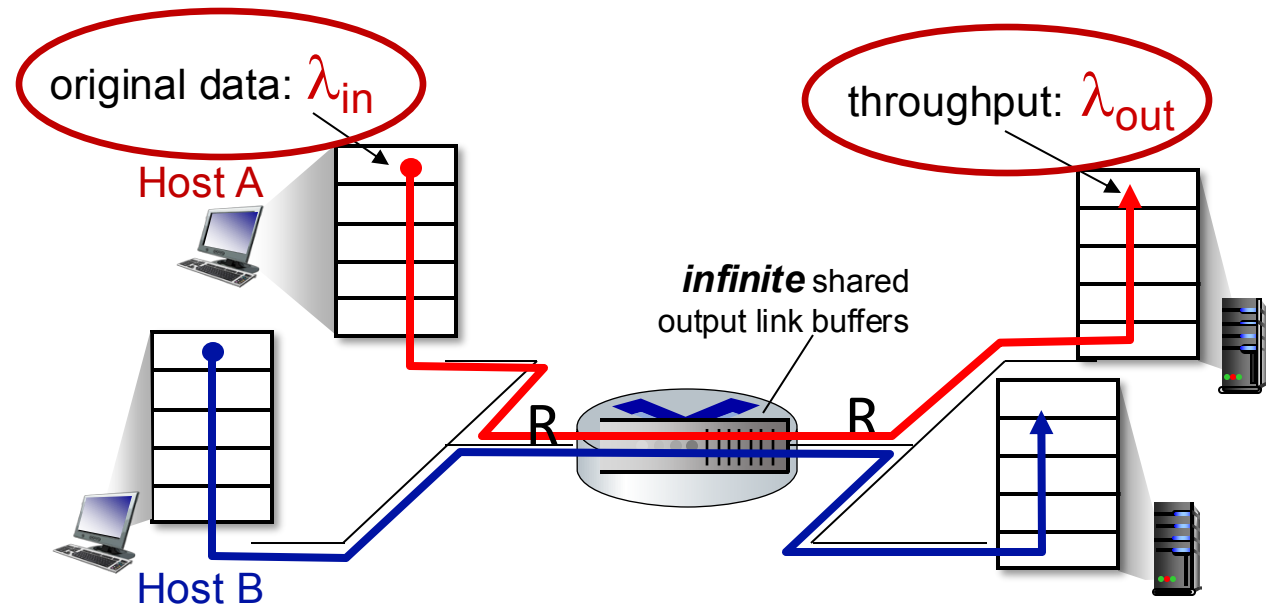


flow control: one sender
too fast for one receiver

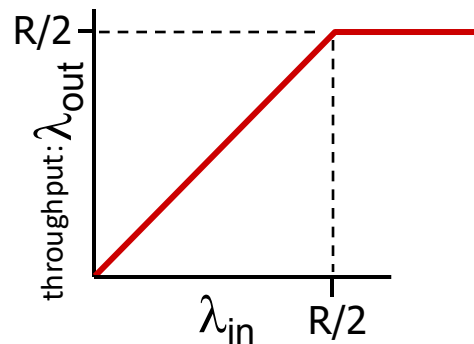
Causes/costs of congestion: scenario 1

Simplest scenario:

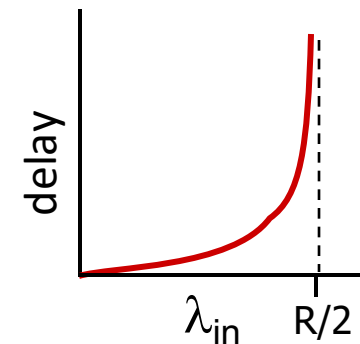
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



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



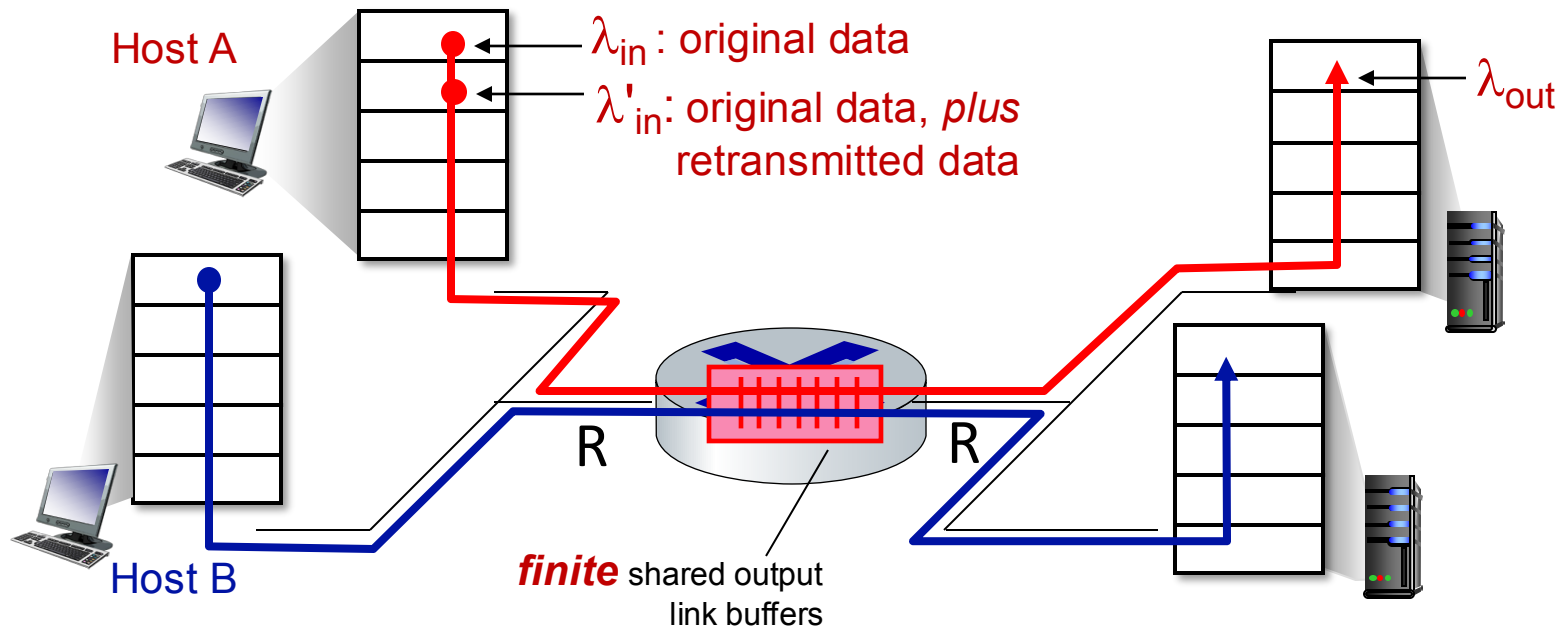
maximum per-connection throughput: $R/2$



large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

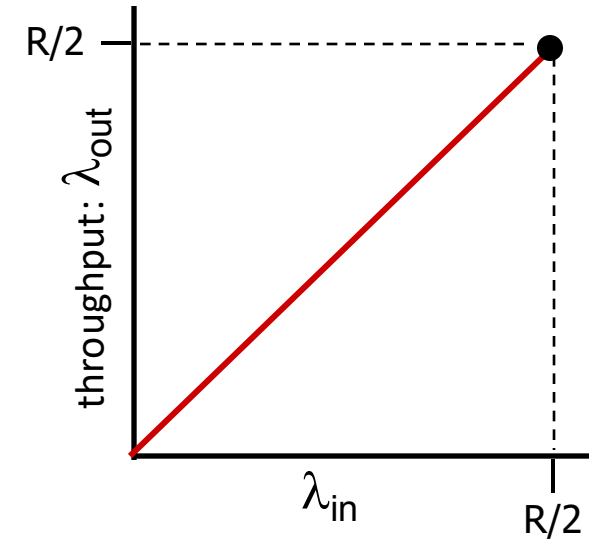
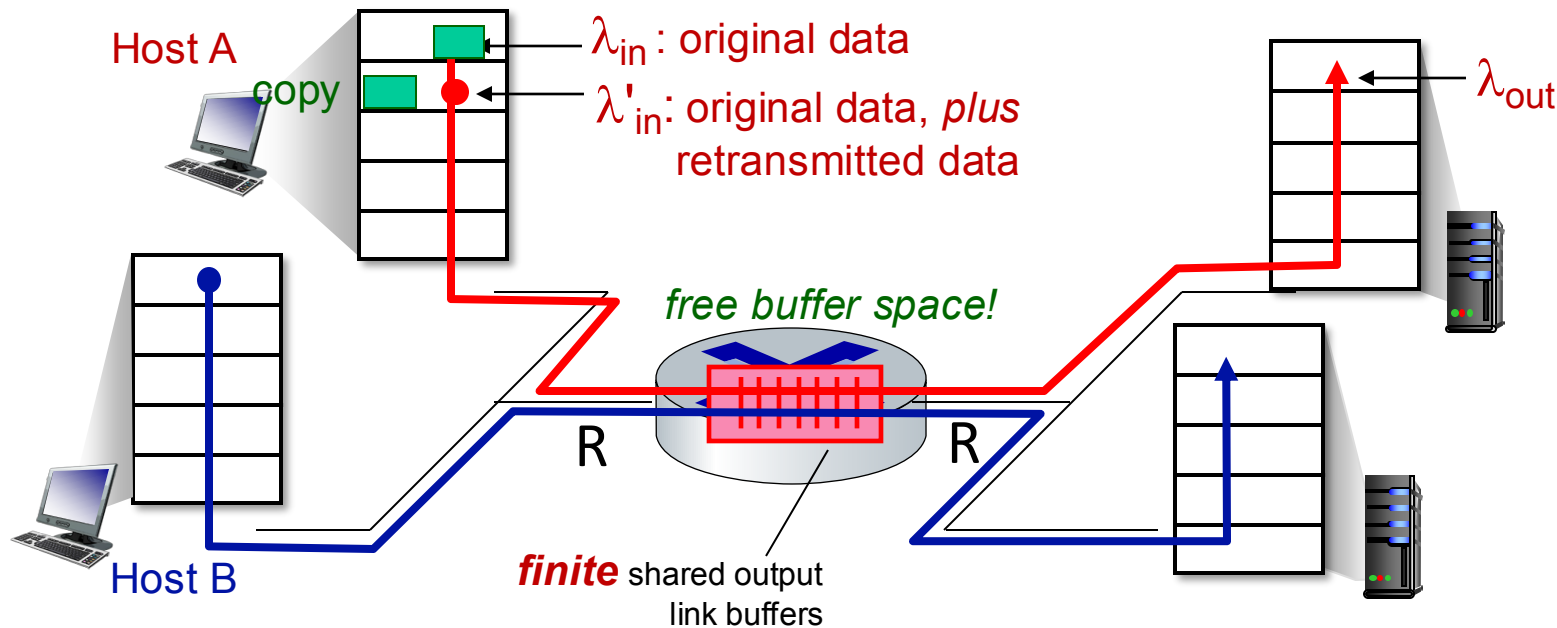
- one router, *finite* buffers
- sender retransmits lost, timed-out packets
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

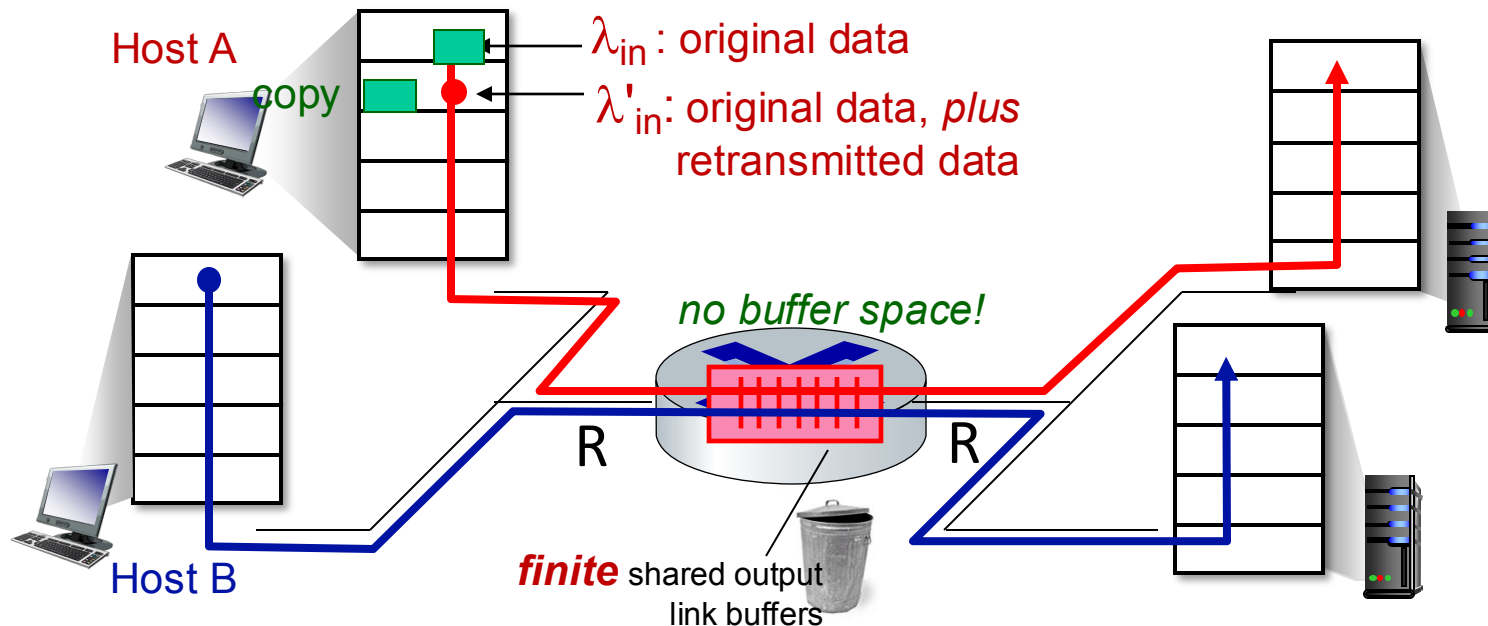
- sender sends only when router buffers available



Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

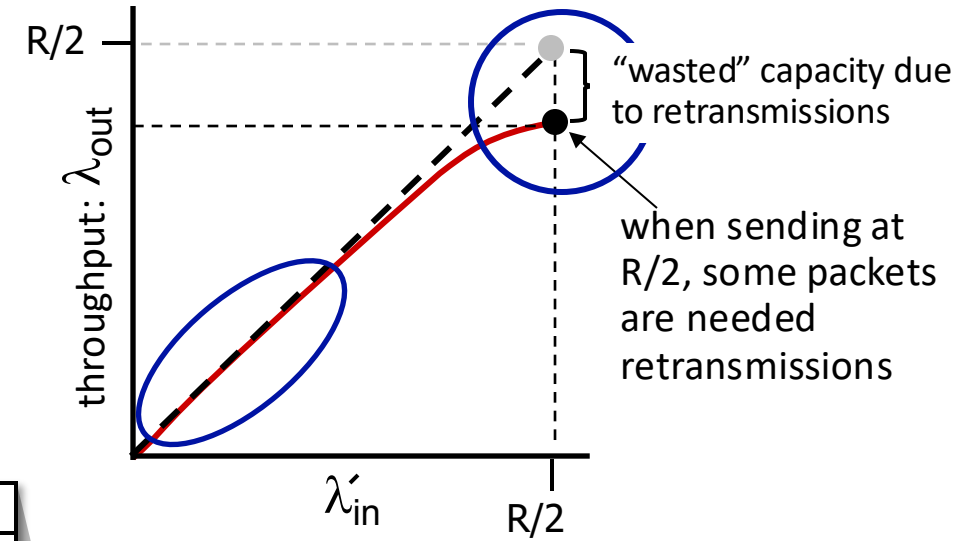
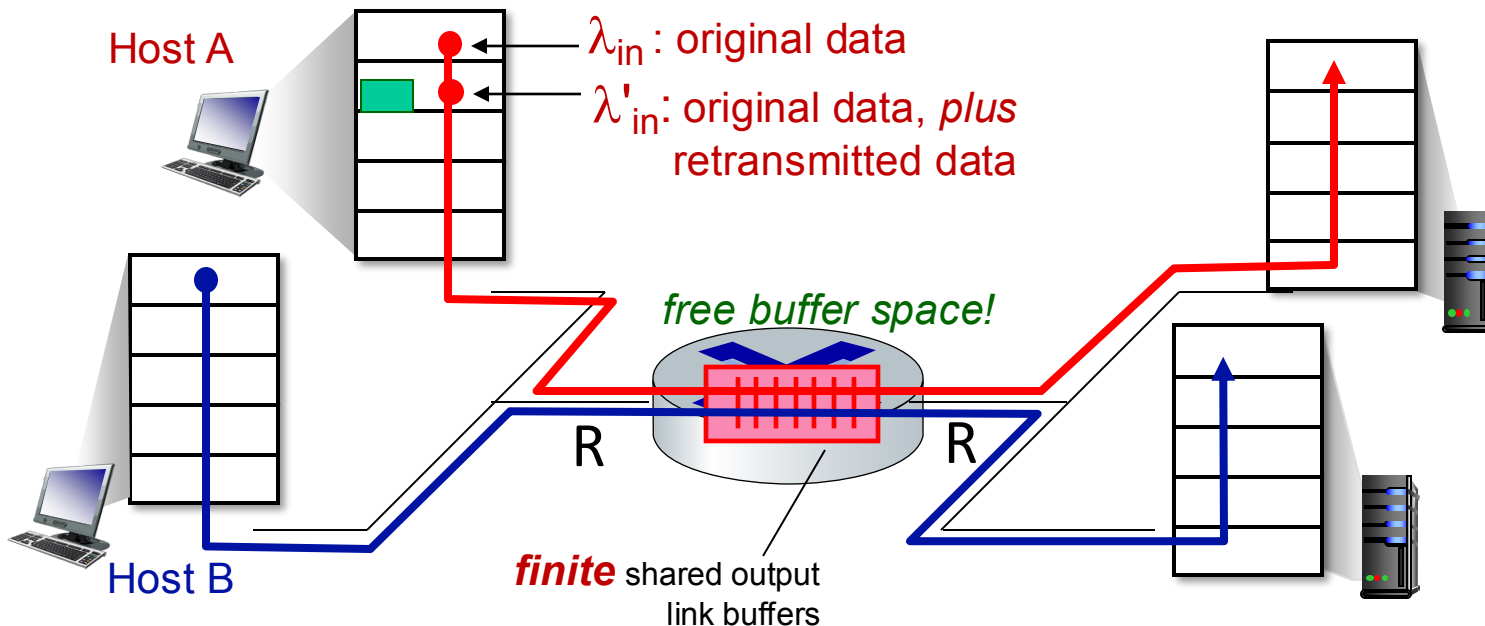
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

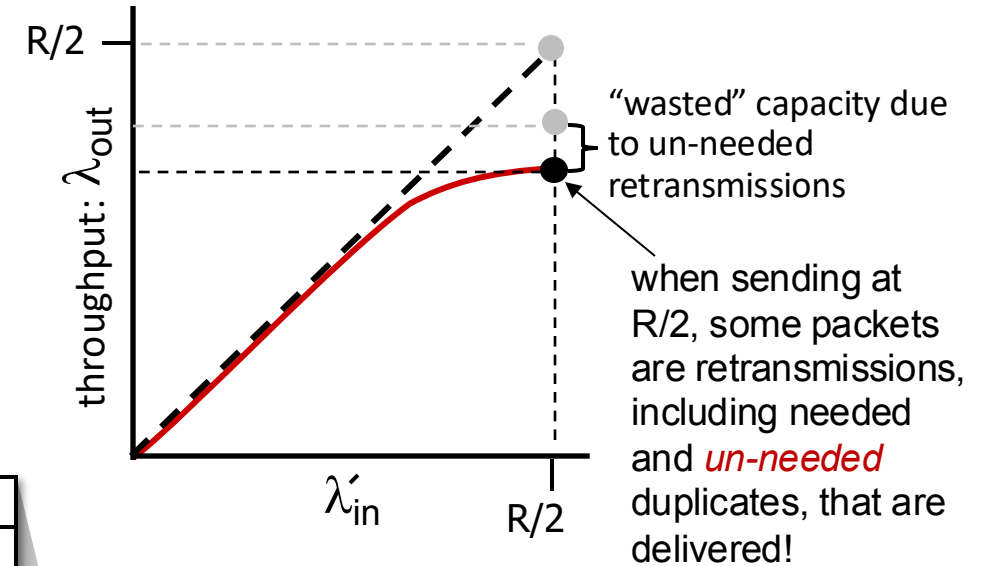
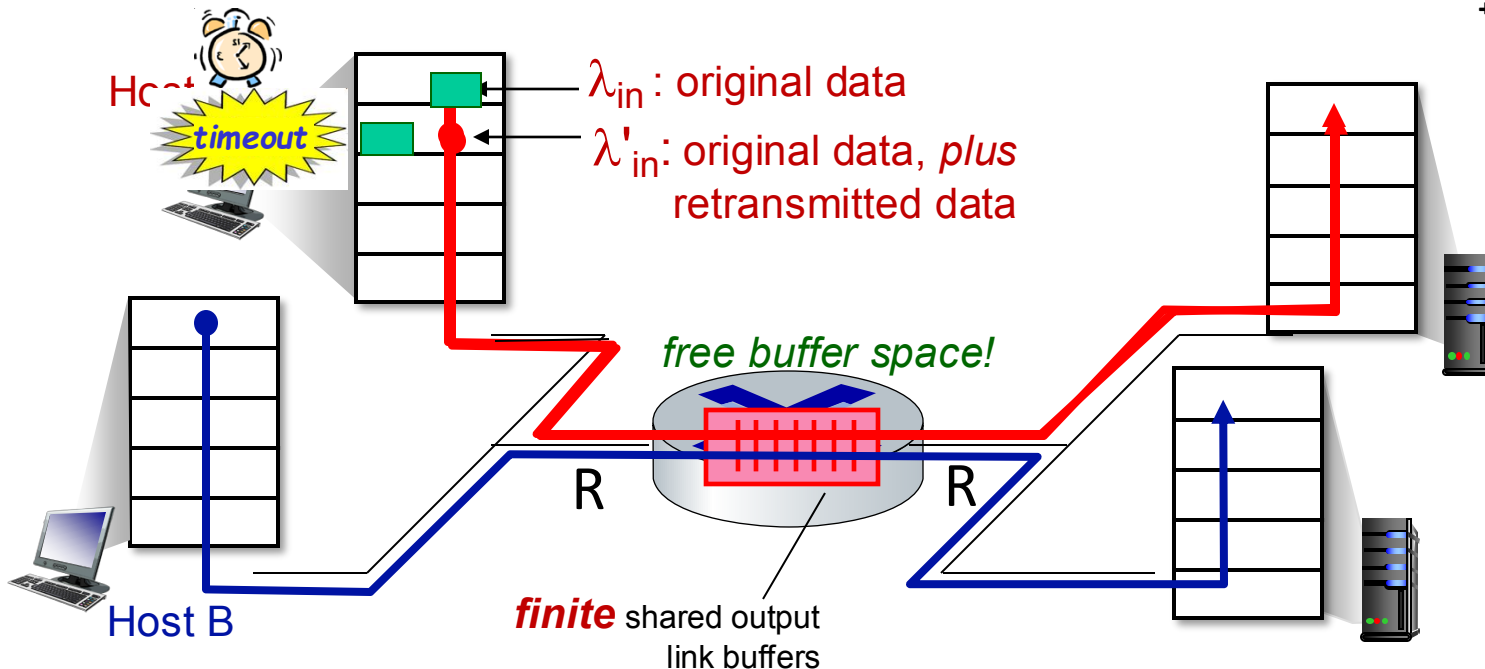
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

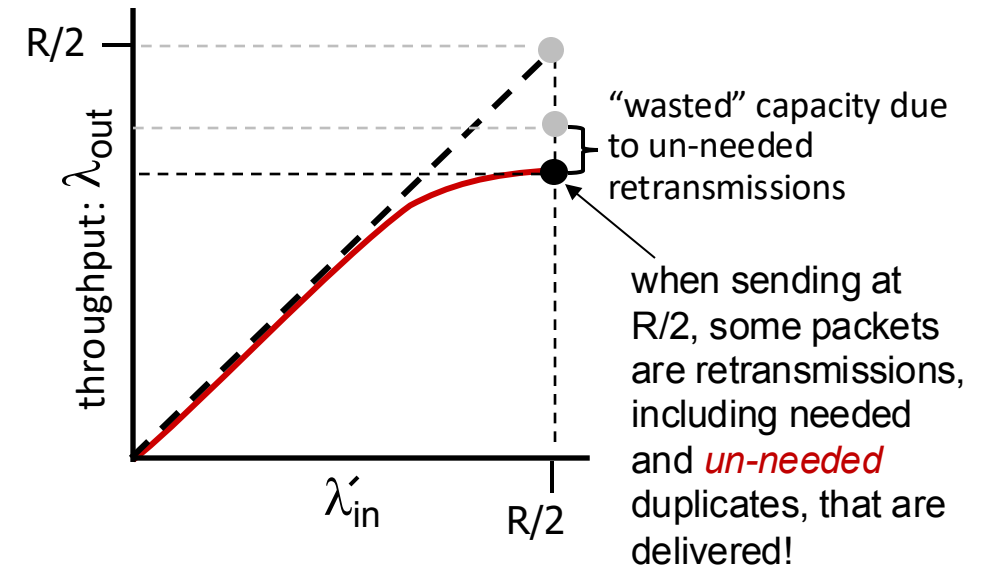
- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"costs" of congestion:

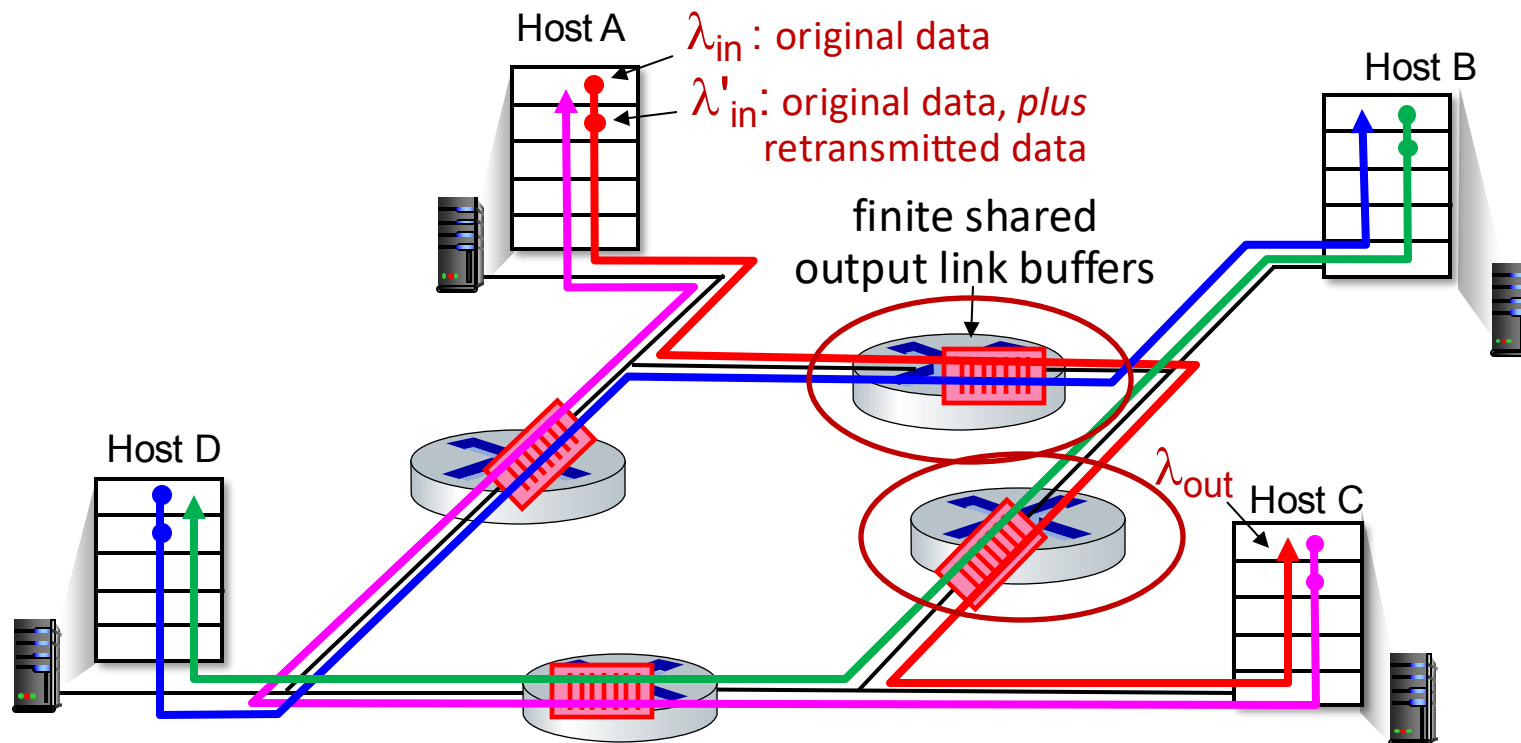
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
 - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

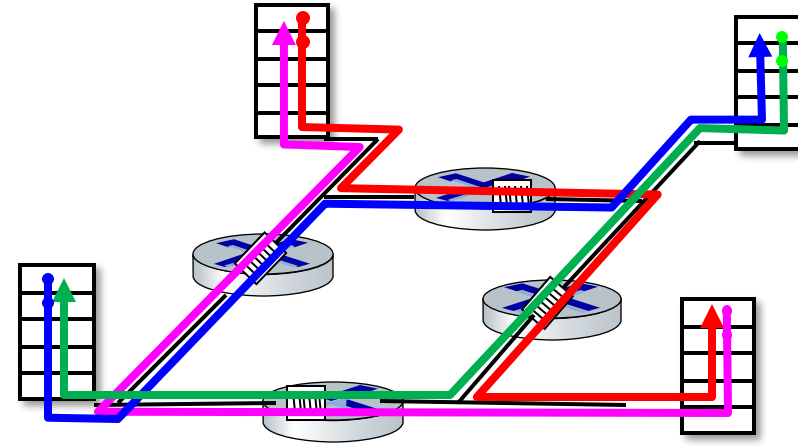
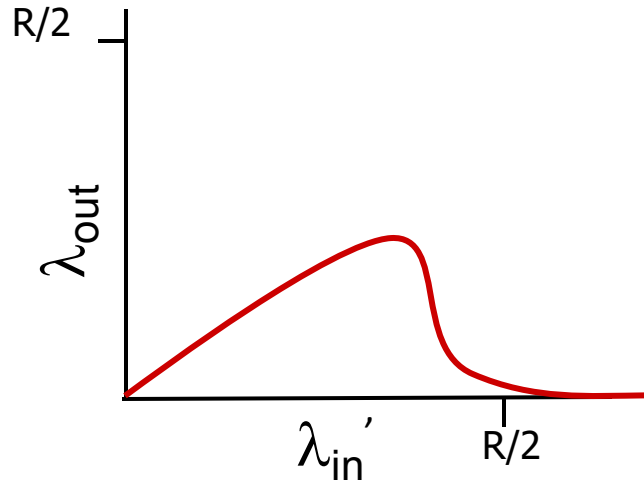
- *four* senders
- *multi-hop* paths
- timeout/retransmit

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

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



Causes/costs of congestion: scenario 3

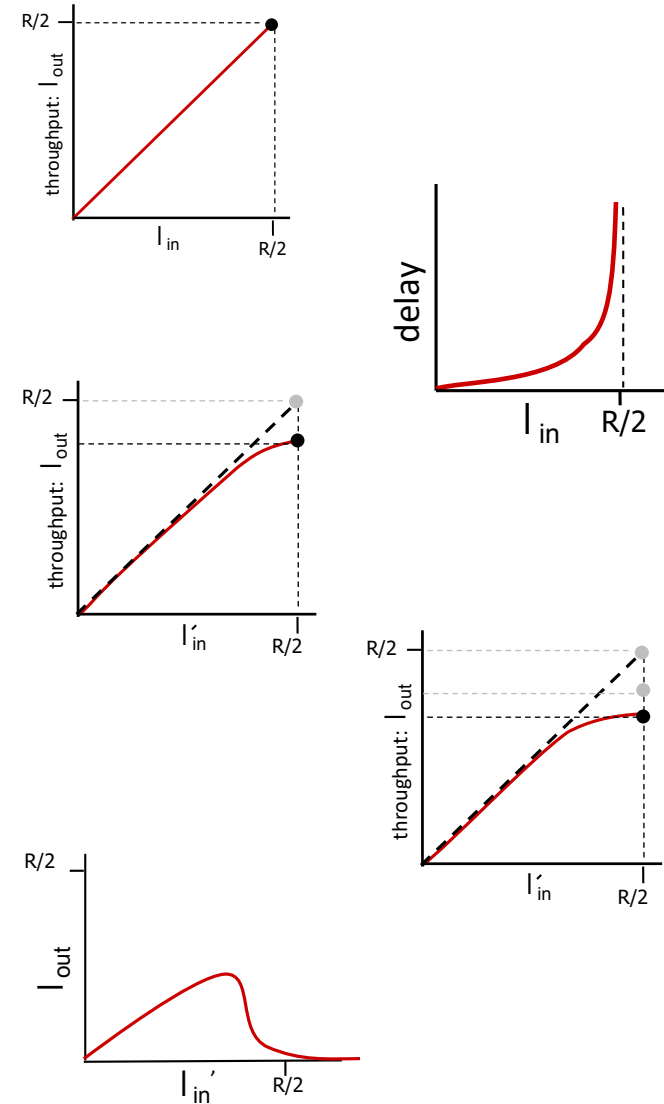


another “cost” of congestion:

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

Causes/costs of congestion: insights

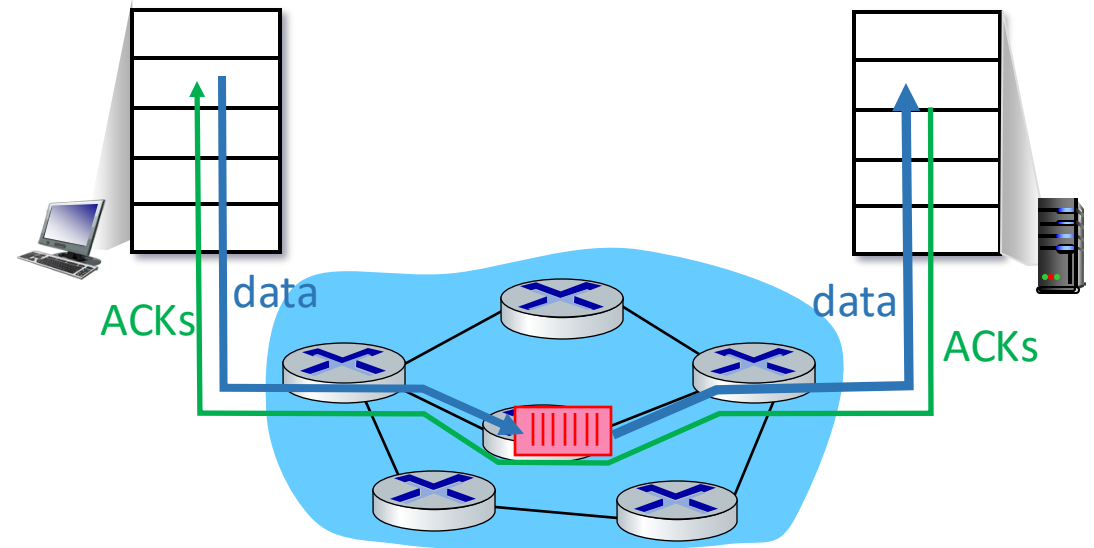
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

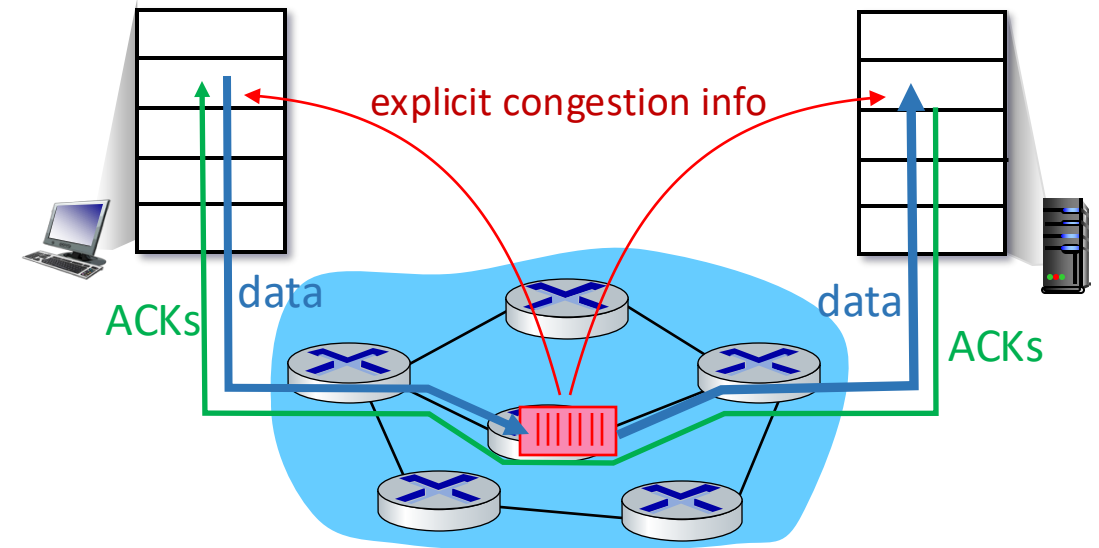
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



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

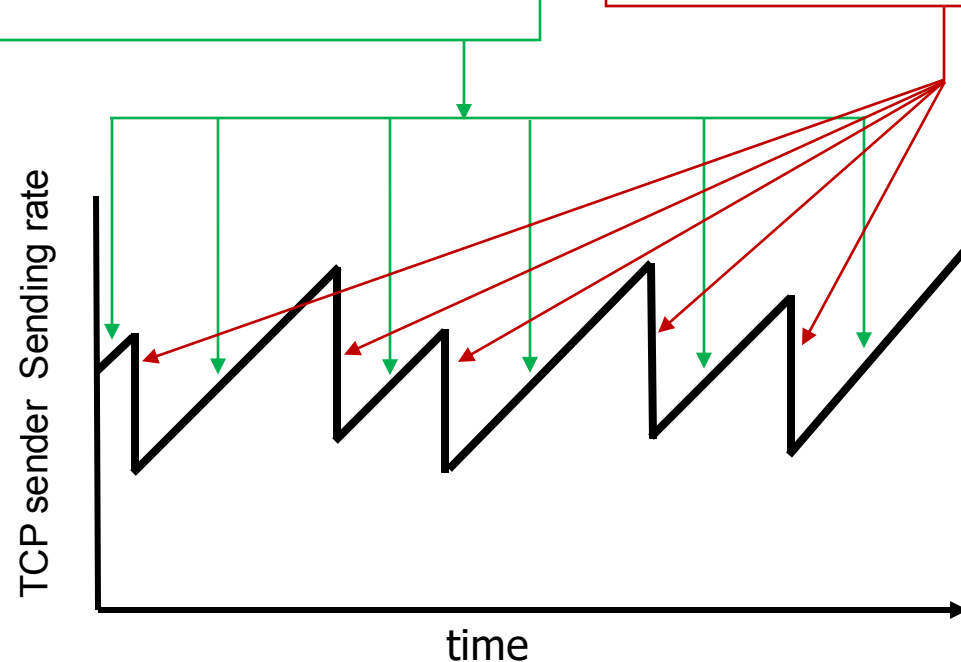
- *approach*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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

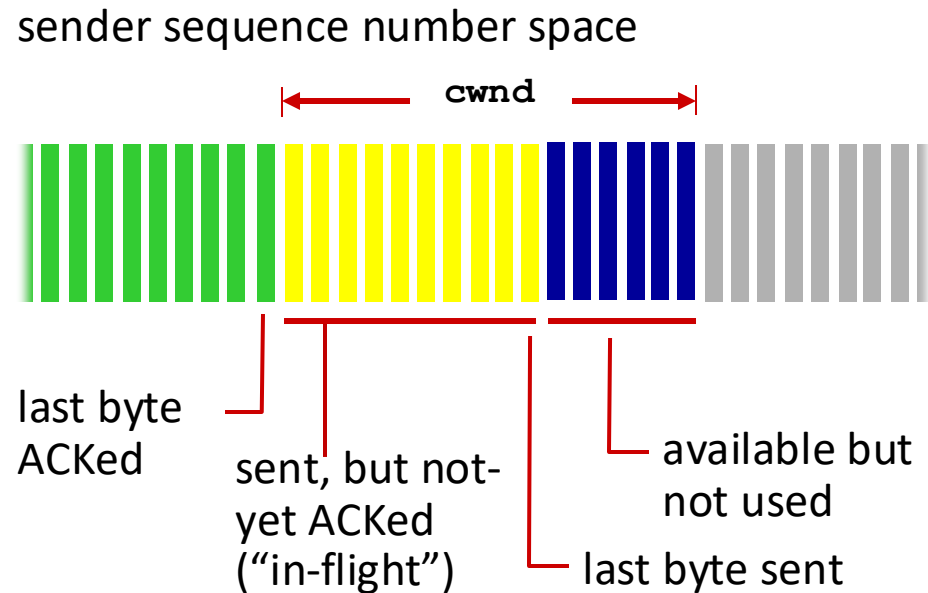
Multiplicative decrease detail: sending rate is

- 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:

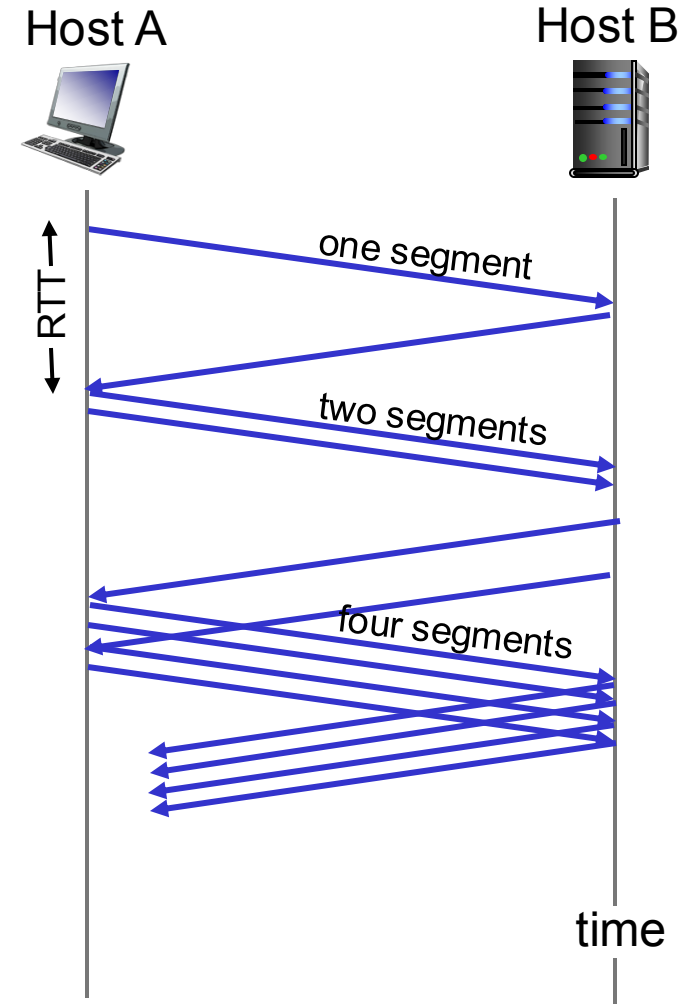
- *roughly*: send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

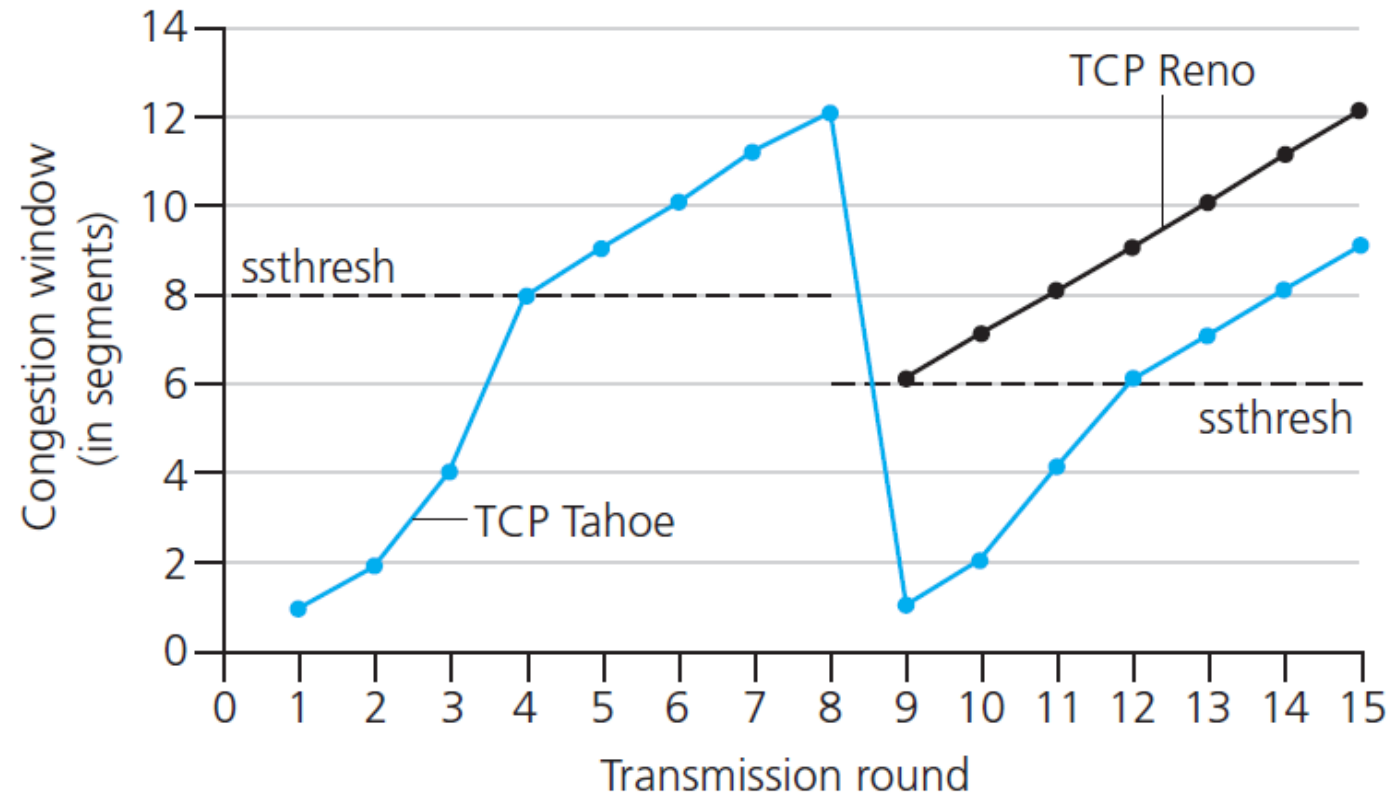
- when connection begins, increase rate exponentially until first loss event:
 - initially **cwnd** = 1 MSS
 - double **cwnd** every RTT
 - done by incrementing **cwnd** for every ACK received
- *summary*: initial rate is slow, but ramps up exponentially fast



TCP: detecting, reacting to loss

- loss indicated by timeout:
 - cwnd set to 1 MSS;
 - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP **RENO**
 - dup ACKs indicate network capable of delivering some segments
 - cwnd is cut in **half window** then grows linearly
(do not set cwnd to “half window + 3” as shown in textbook)
- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)

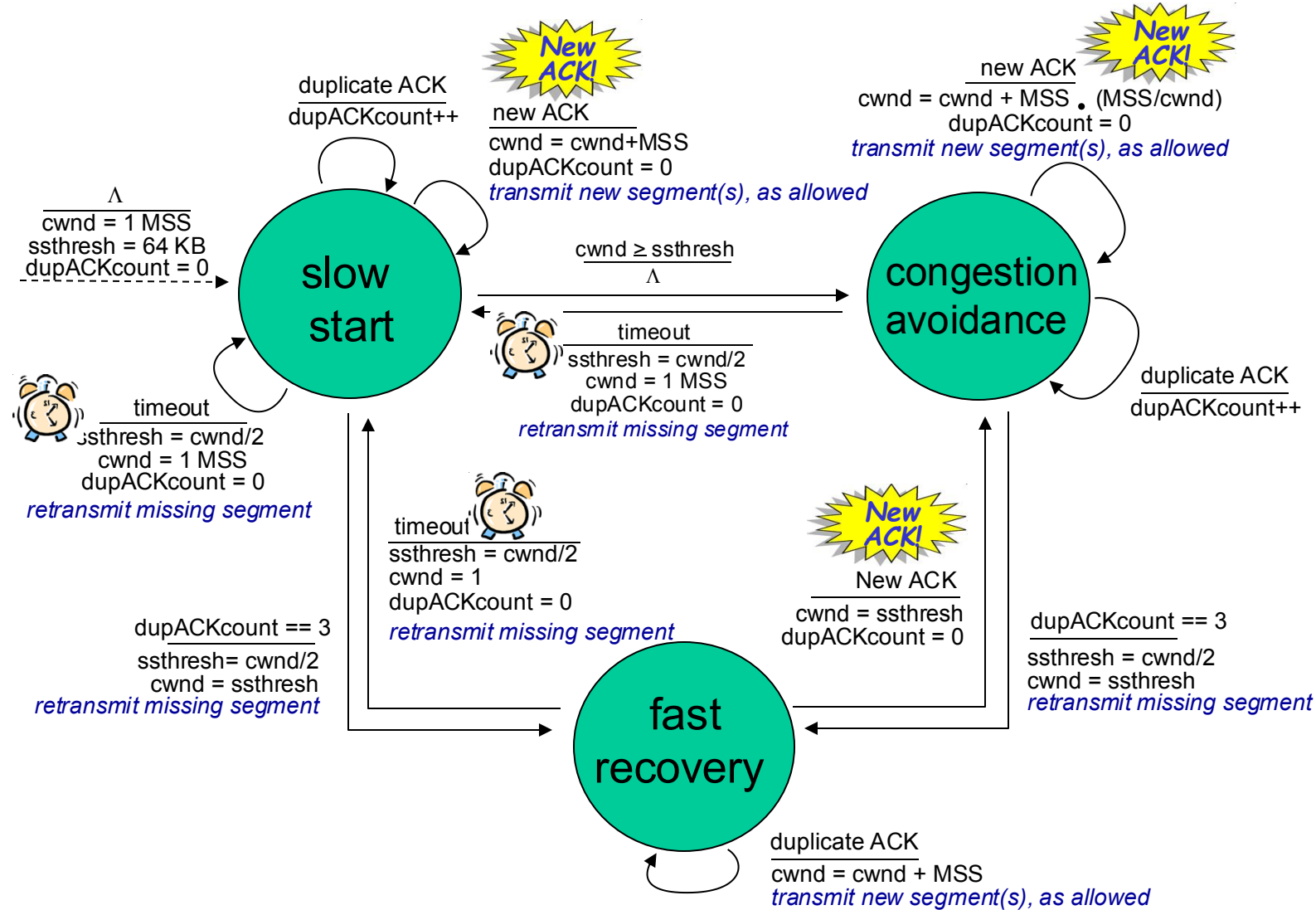
TCP: from slow start to congestion avoidance



Implementation:

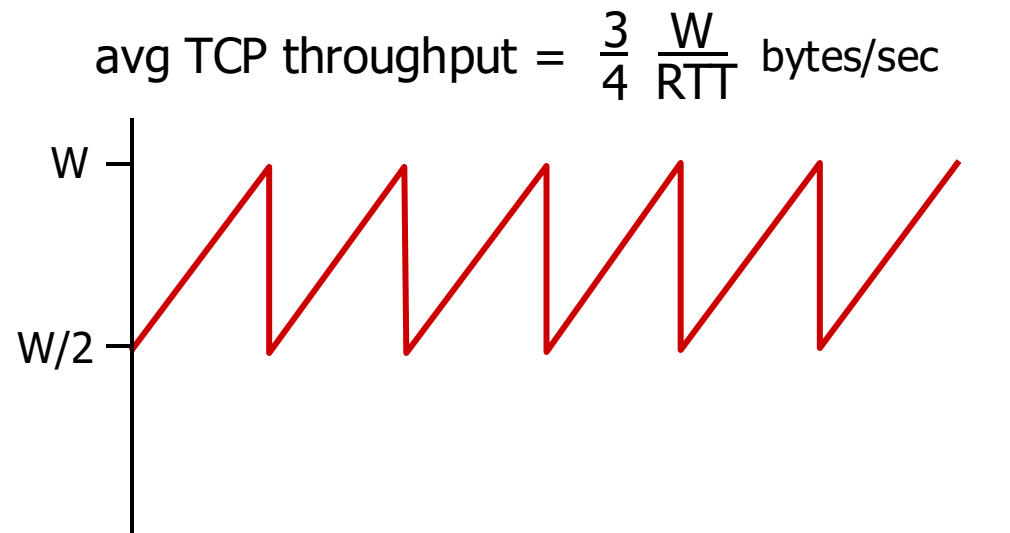
- variable **ssthresh** (slow start threshold)
- on loss event, **ssthresh** is set to $1/2$ of **cwnd** just before loss event

Summary: TCP congestion control



TCP throughput

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



TCP over “long, fat pipes”

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires $W = 83,333$ in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

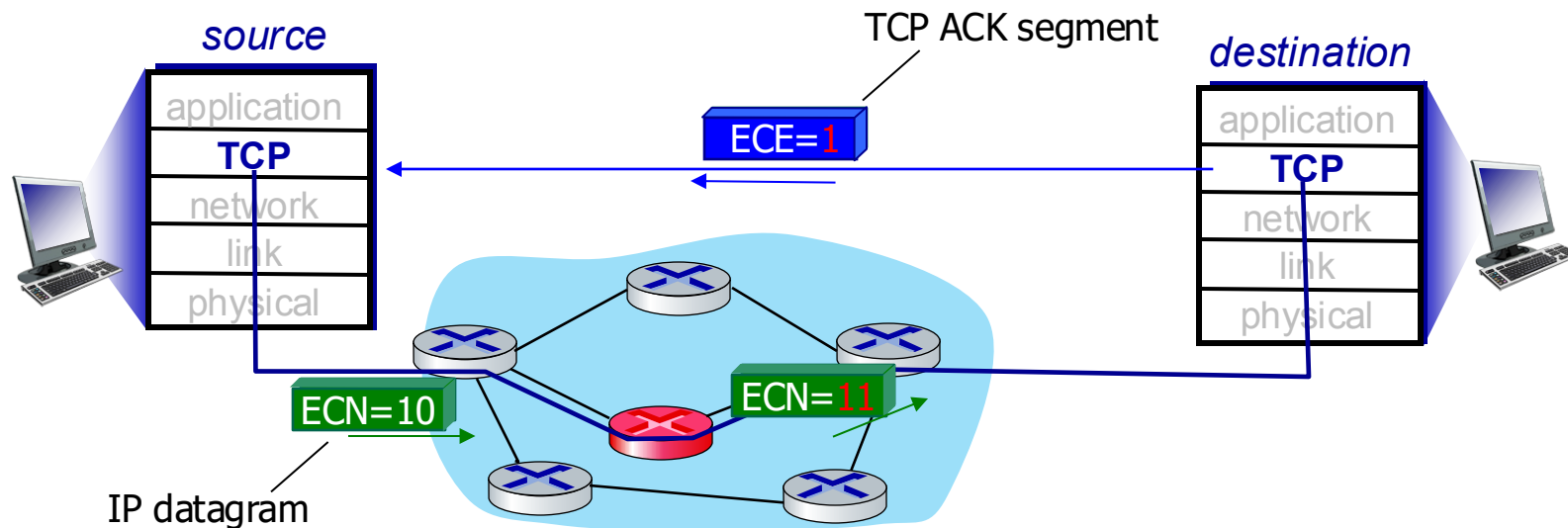
→ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$
– *a very small loss rate!*

- versions of TCP for long, high-speed scenarios

Explicit congestion notification (ECN)

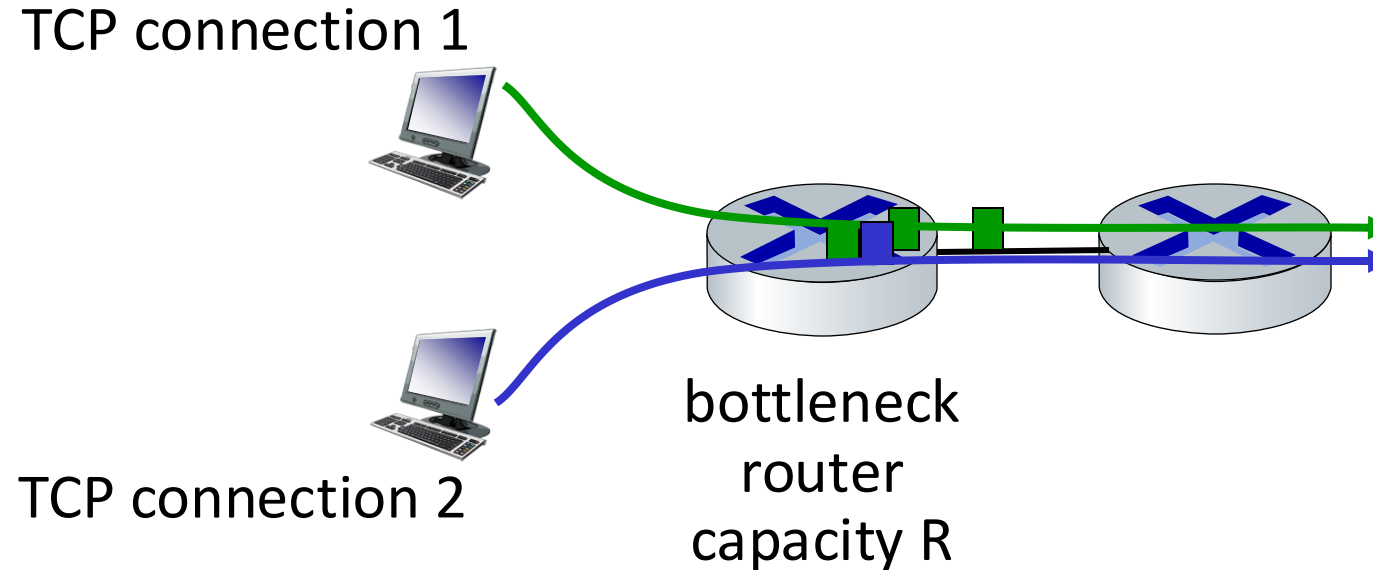
TCP deployments often implement *network-assisted* congestion control:

- 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

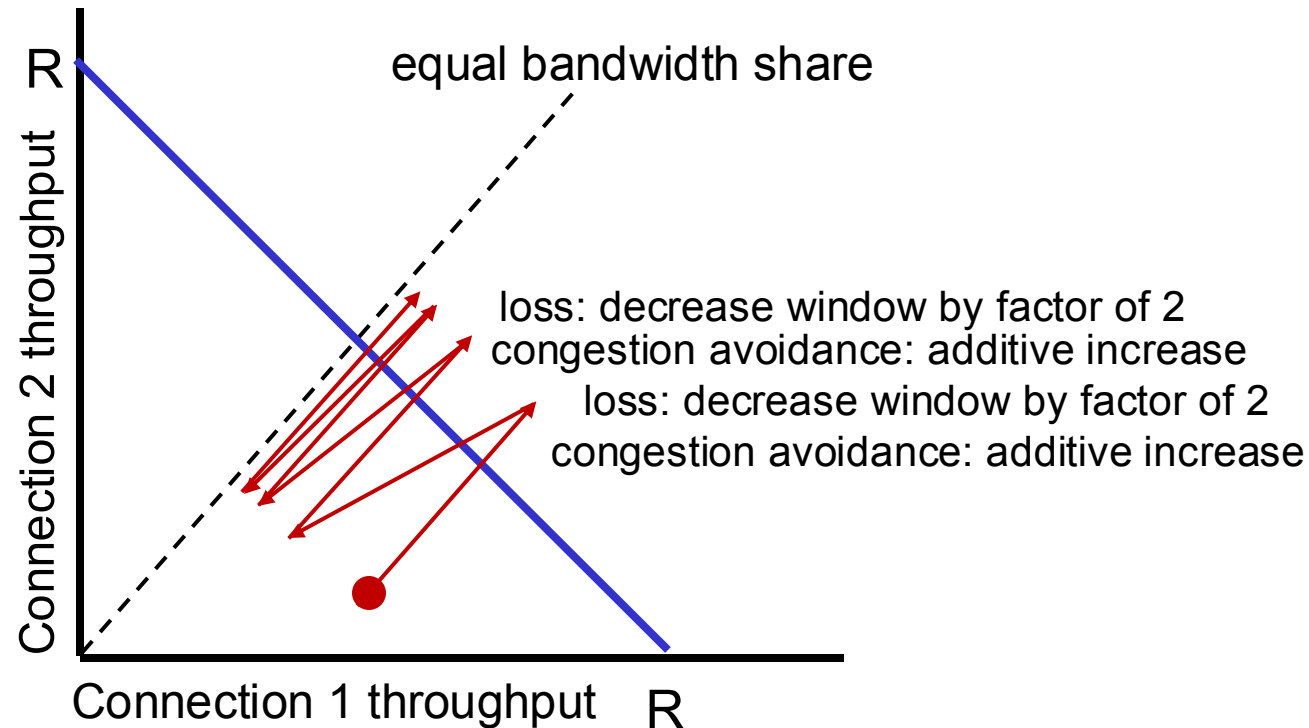
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughput 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 $R/2$

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