



# Computer Networks

Lecturer: ZHANG Ying

Fall semester 2022

# 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

# 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



# Principles of congestion control

## Congestion:

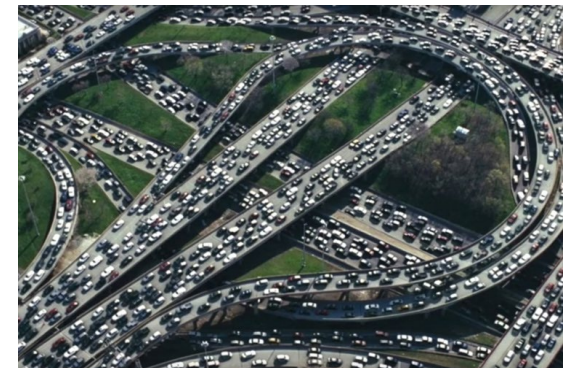
- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)



# 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

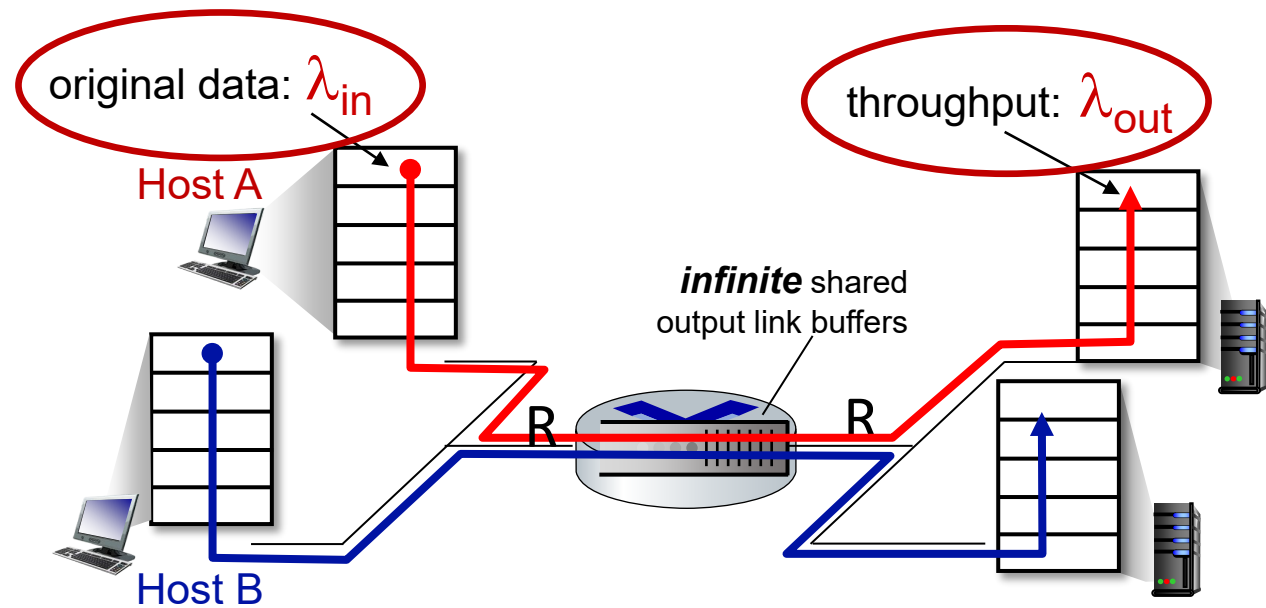


**flow control:** one sender  
too fast for one receiver

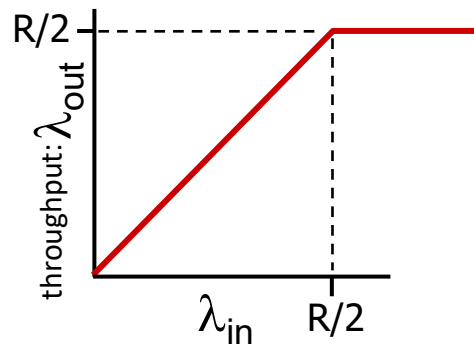
# Causes/costs of congestion: scenario 1

Simplest scenario:

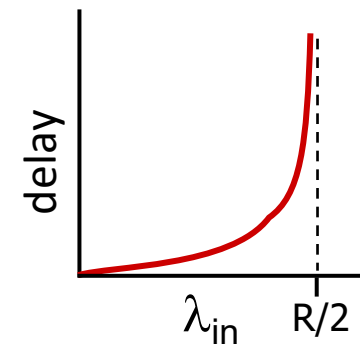
- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed



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



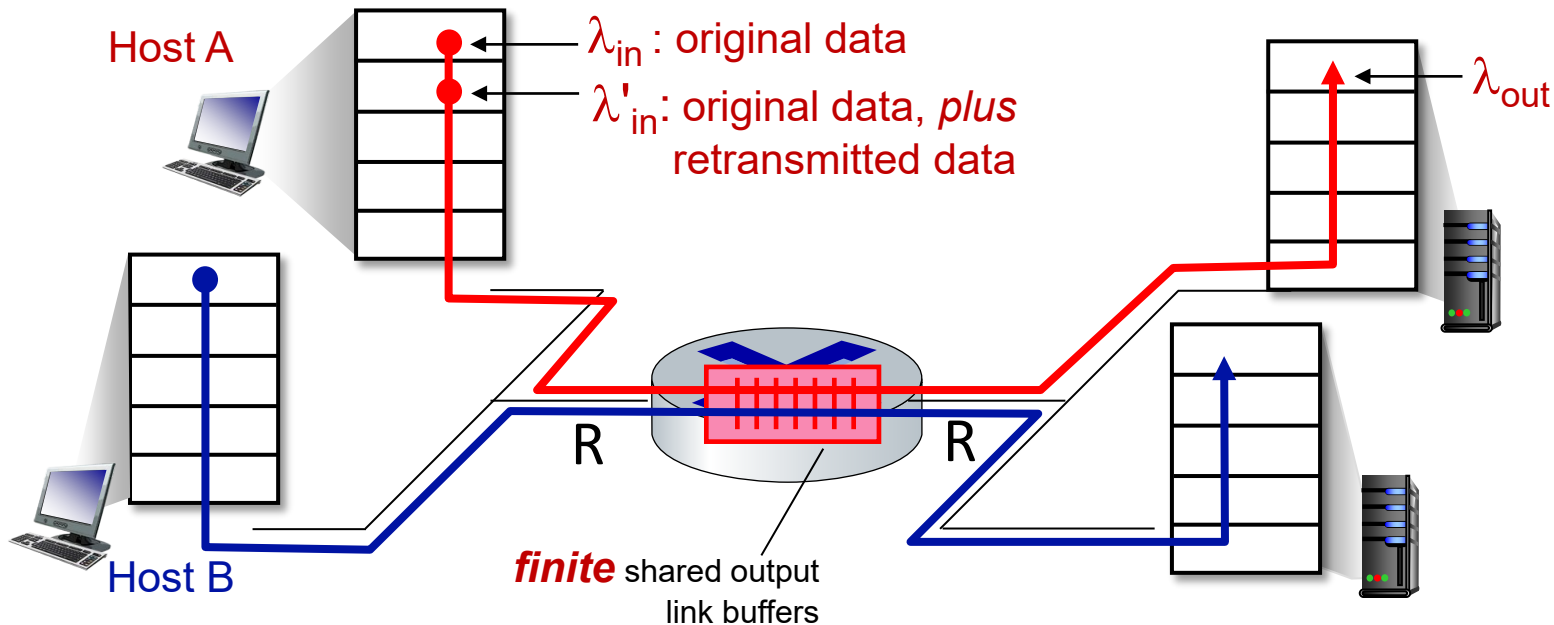
maximum per-connection throughput:  $R/2$



large delays as arrival rate  $\lambda_{in}$  approaches capacity

# Causes/costs of congestion: scenario 2

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

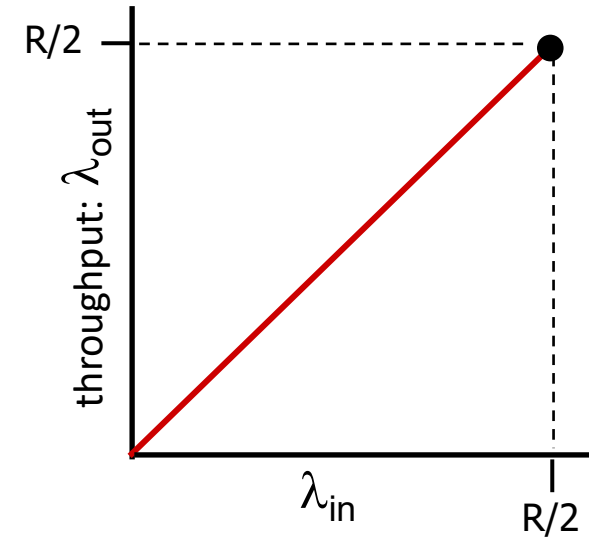
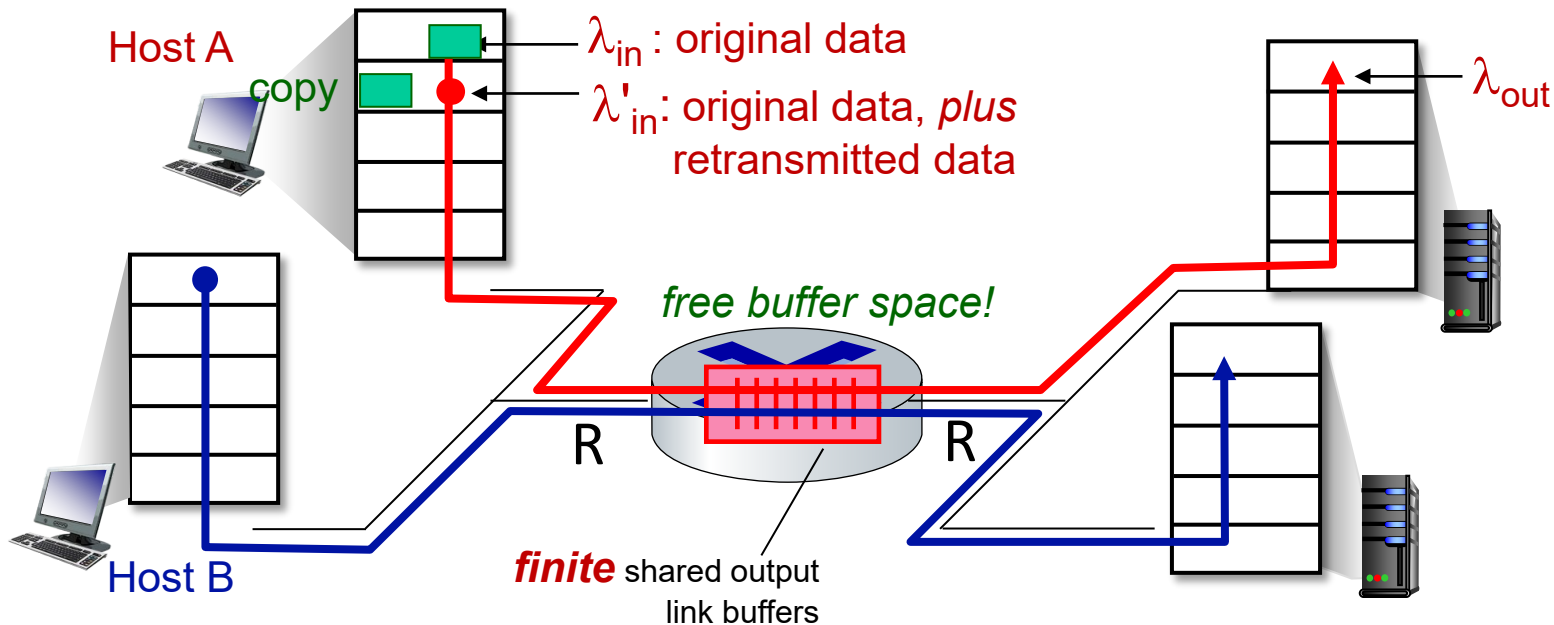




# Causes/costs of congestion: scenario 2-1

Idealization: perfect knowledge

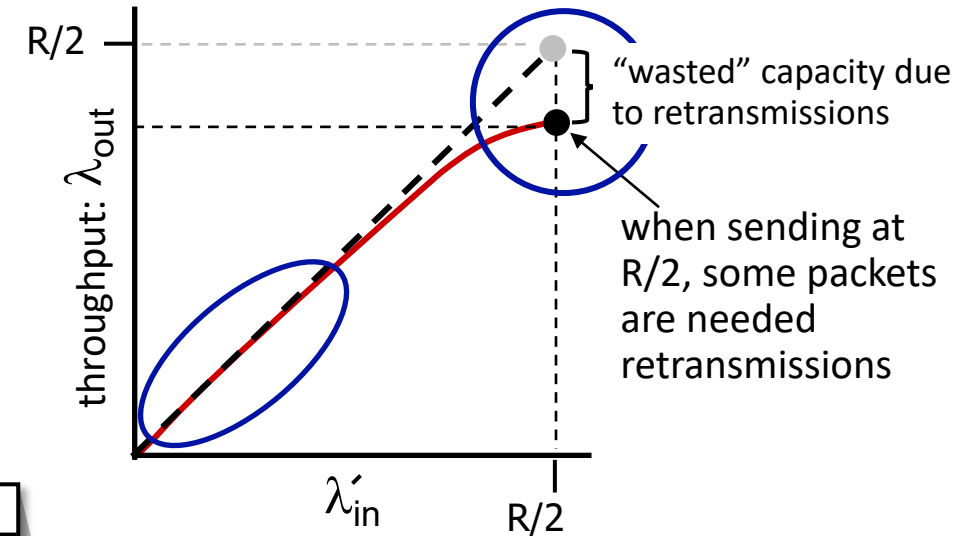
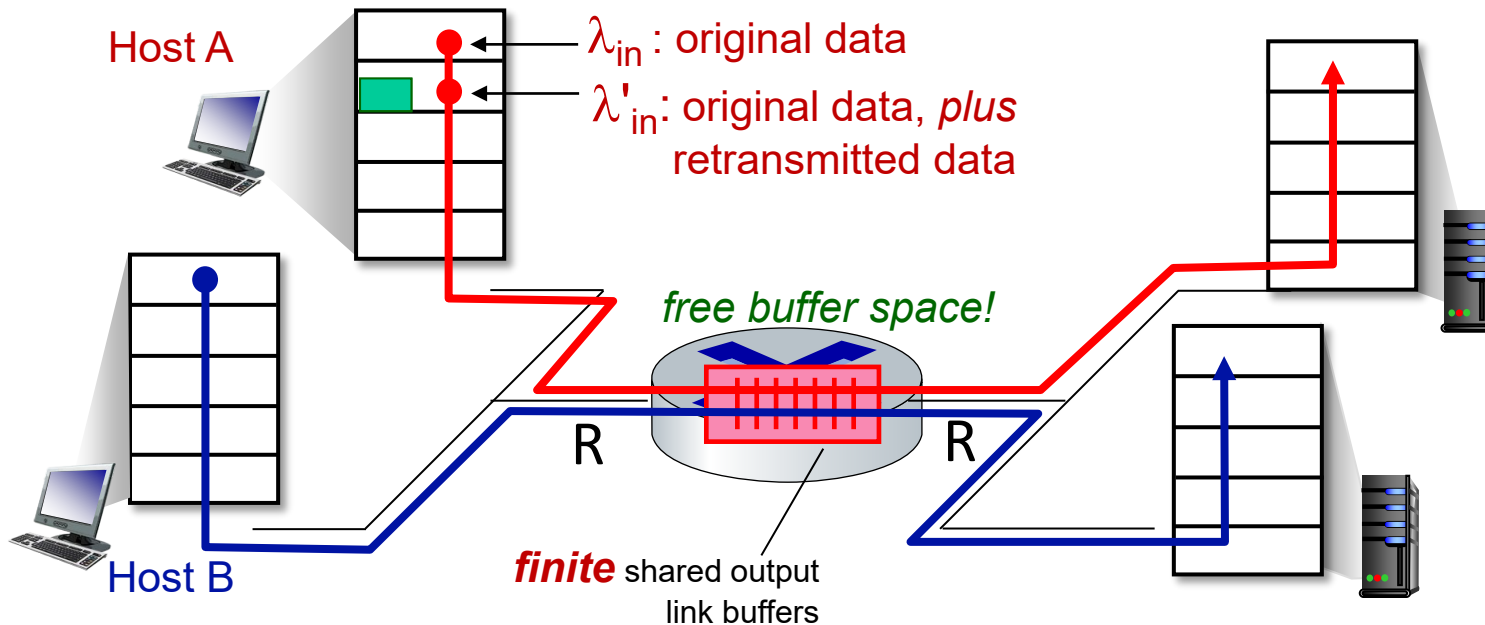
- sender sends only when router buffers available



# Causes/costs of congestion: scenario 2-2

Idealization: *some* perfect knowledge

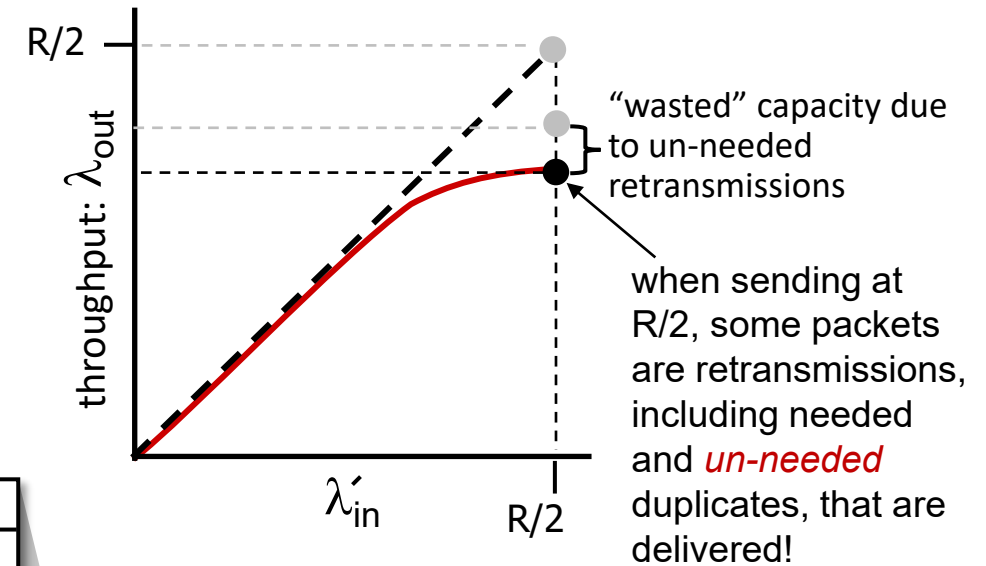
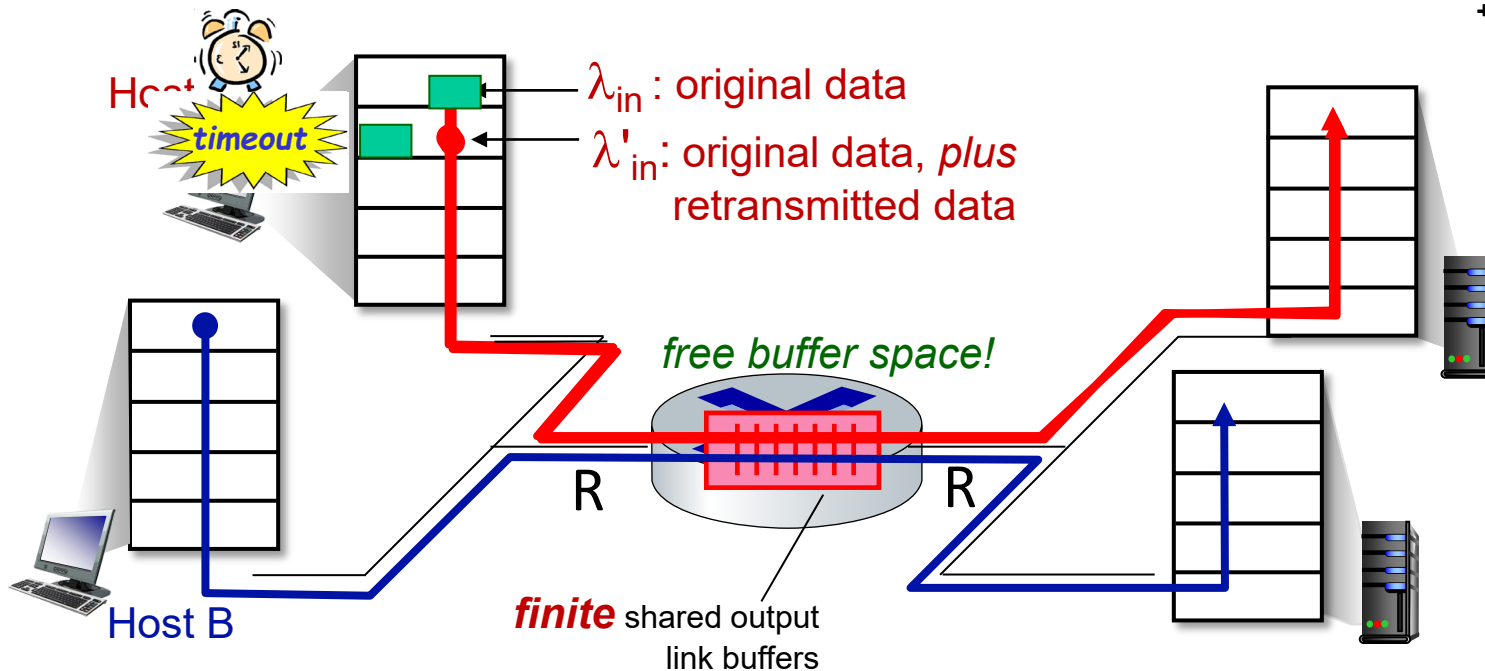
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2-3

## Realistic scenario: *un-needed duplicates*

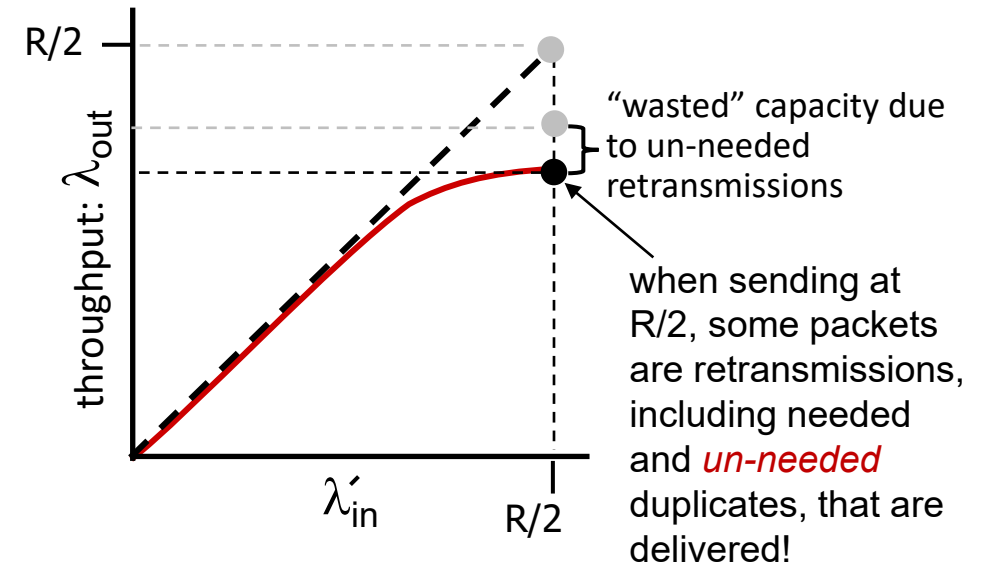
- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



# Causes/costs of congestion: scenario 2-3

## 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:

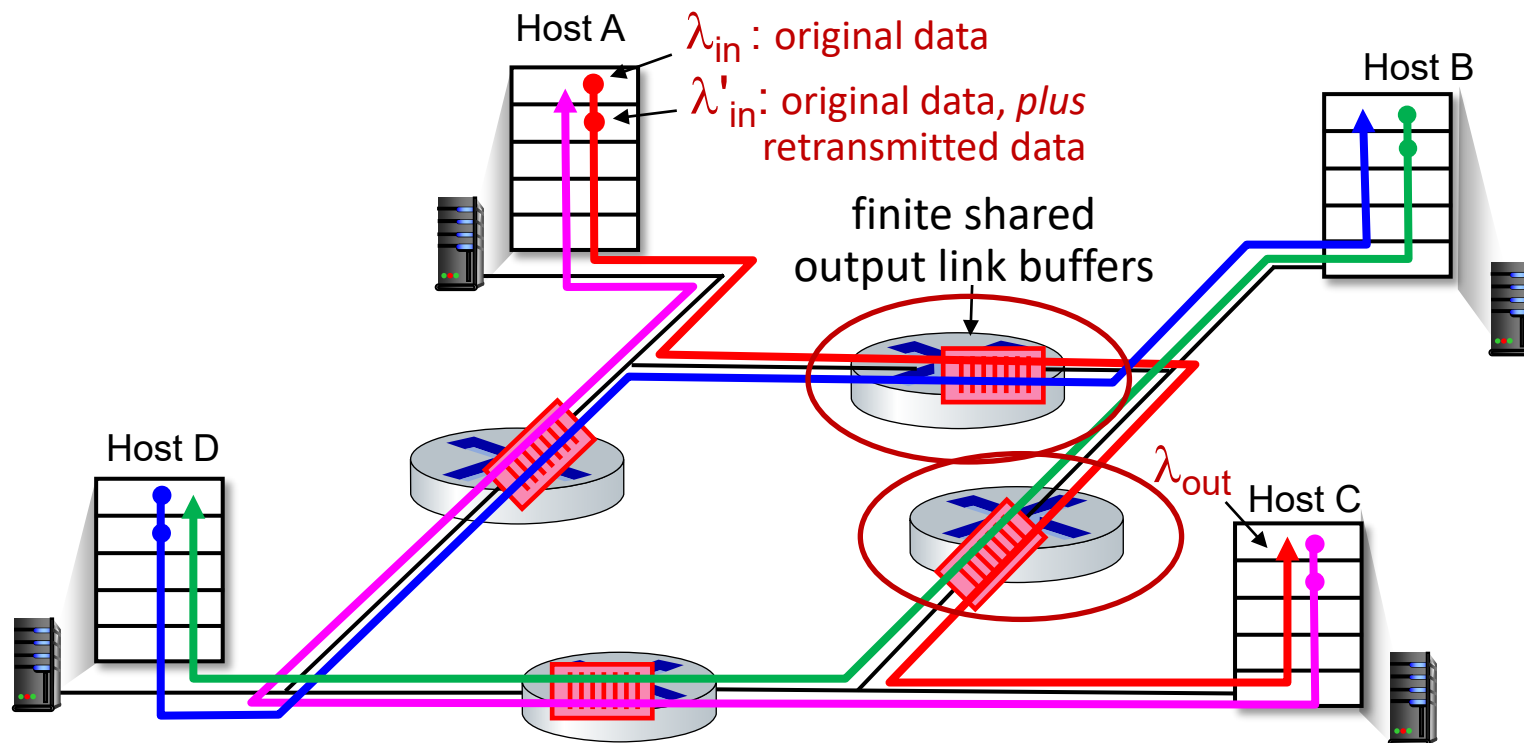
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

# Causes/costs of congestion: scenario 3

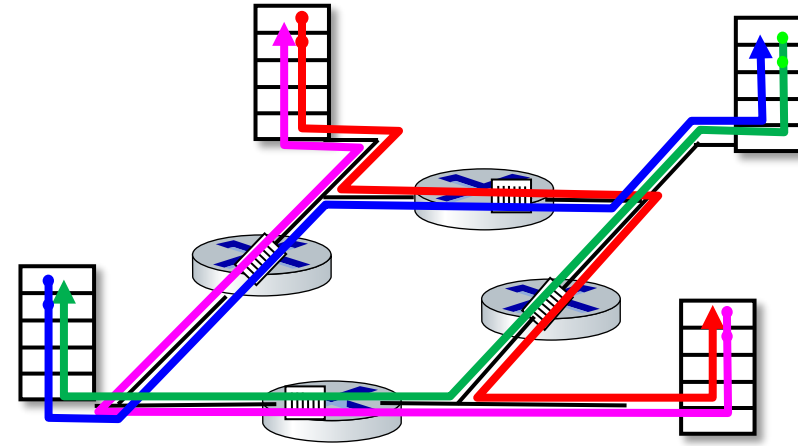
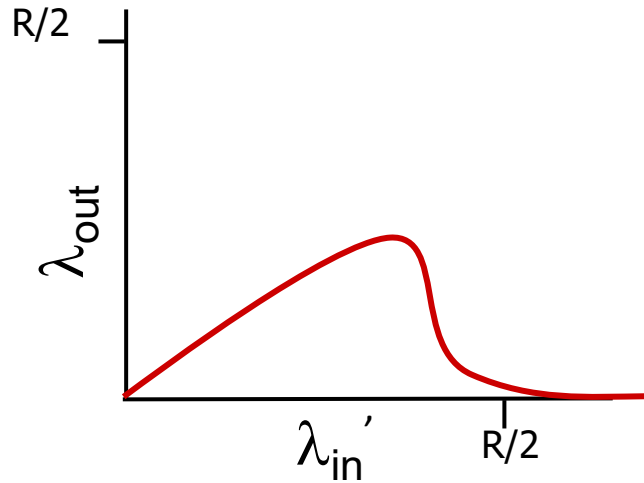
- *four* senders
- *multi-hop* paths
- timeout/retransmit

Q: what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

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



# Causes/costs of congestion: scenario 3



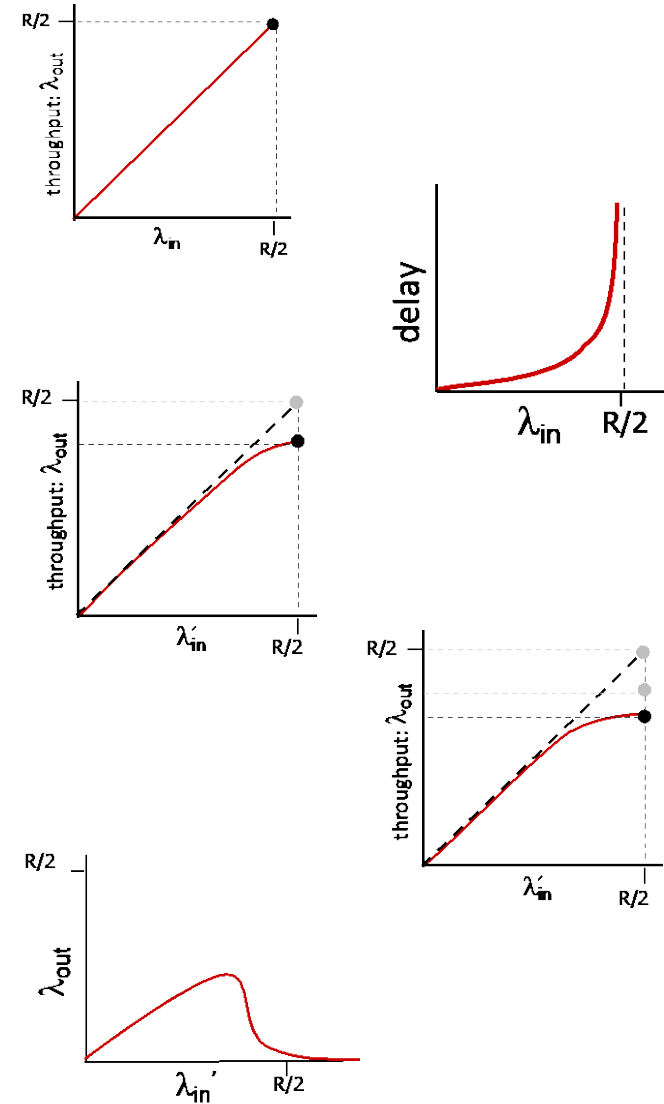
another “cost” of congestion:

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



# Causes/costs of congestion: insights

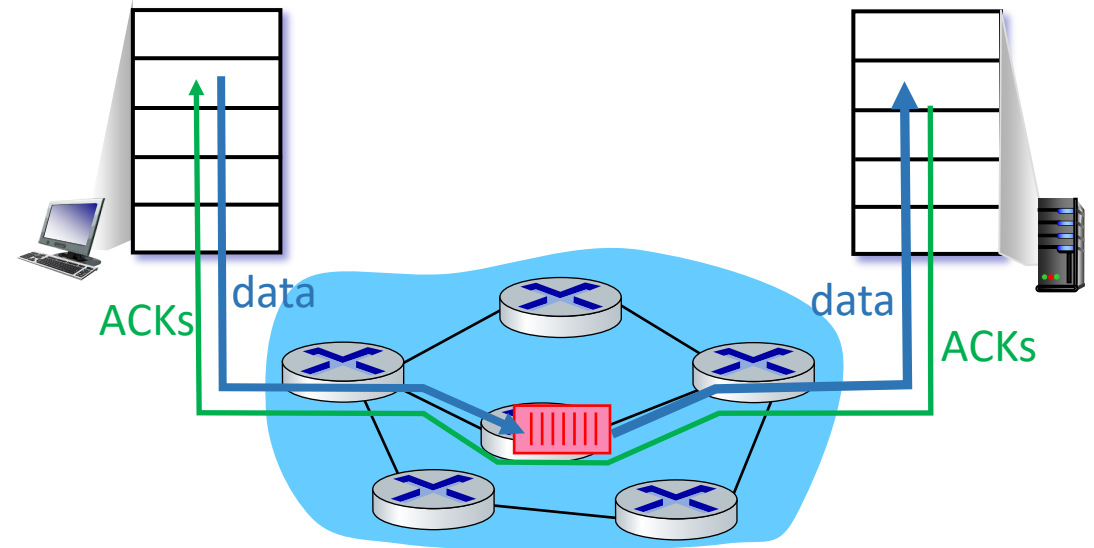
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



# Approaches towards congestion control

## End-end congestion control:

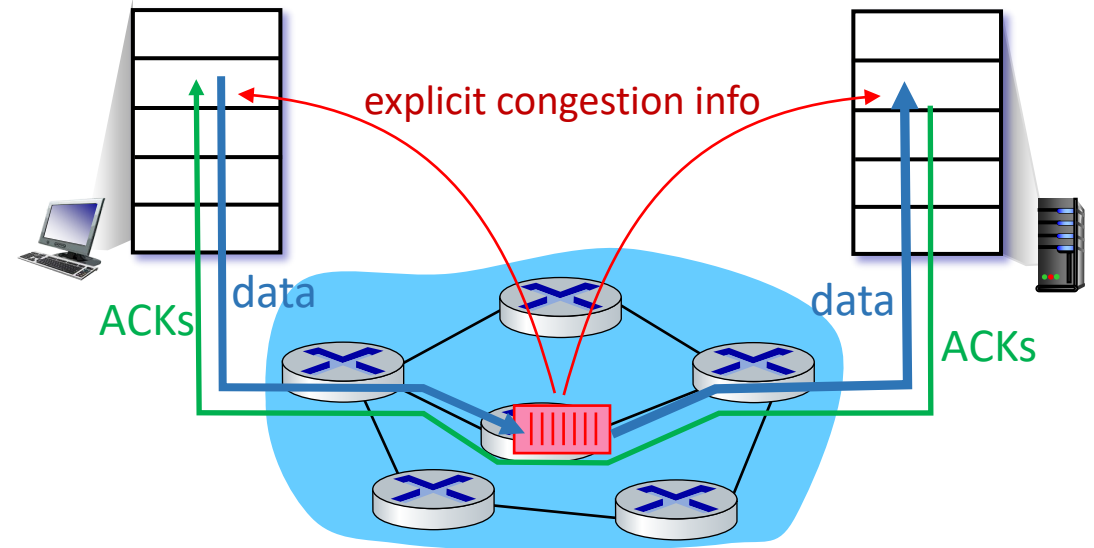
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



# 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

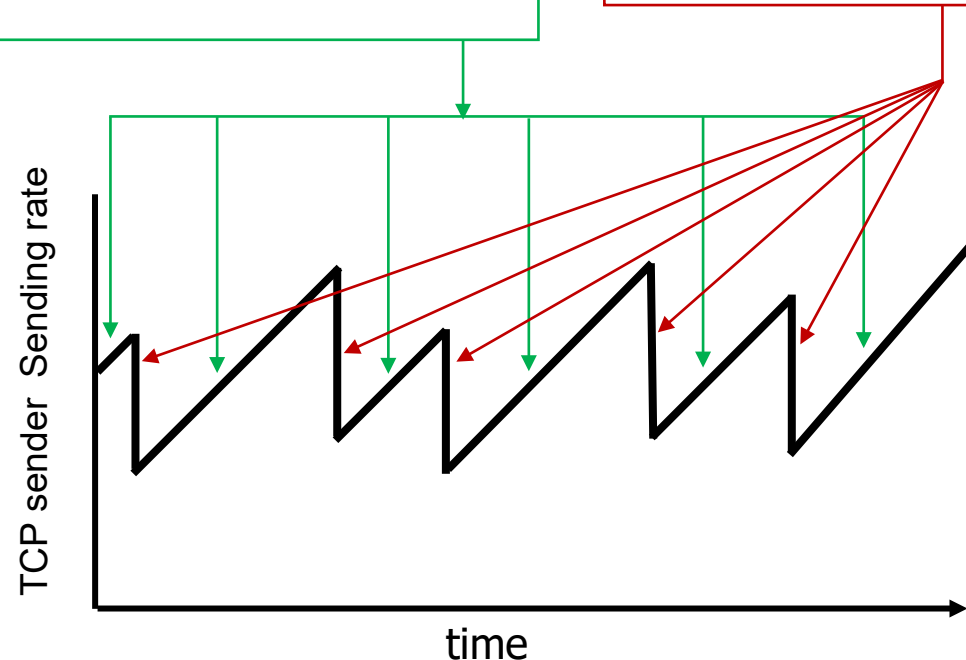
- *approach*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

## 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

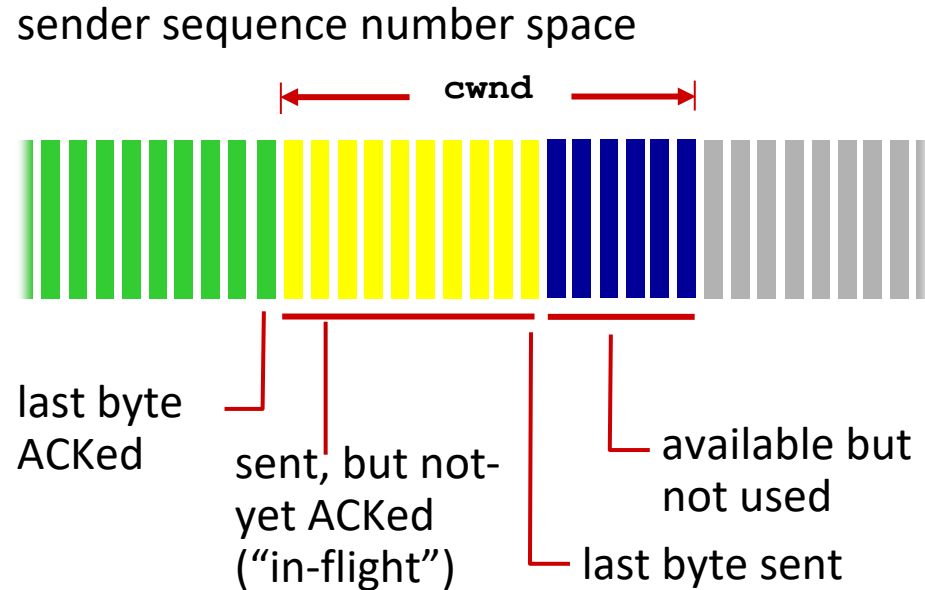
# TCP AIMD: more

*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)



# TCP congestion control: details



TCP sending behavior:

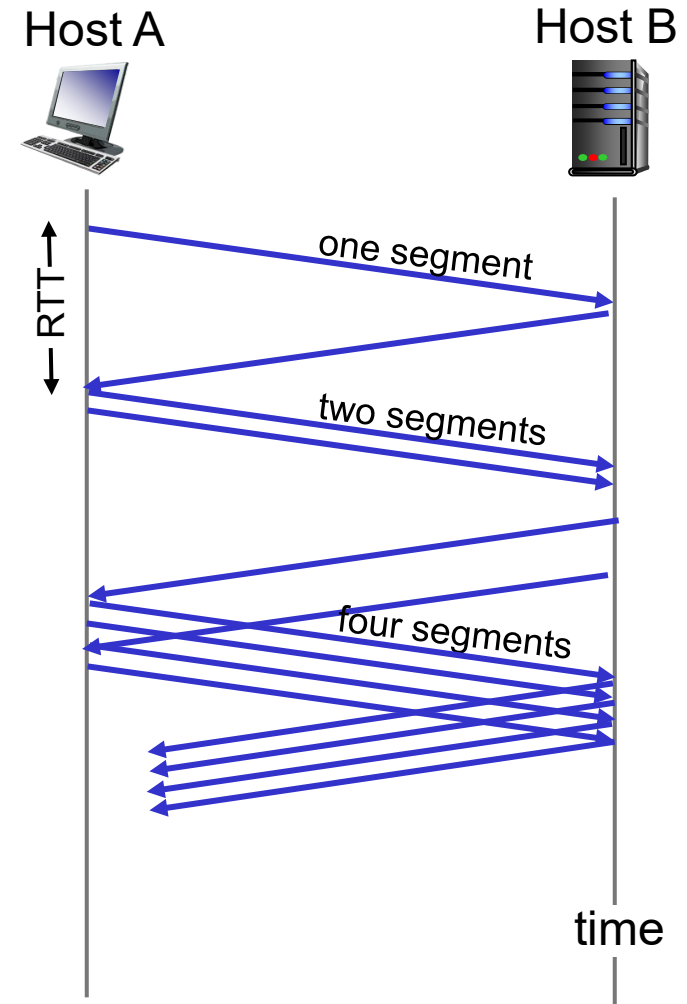
- *roughly*: send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- 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



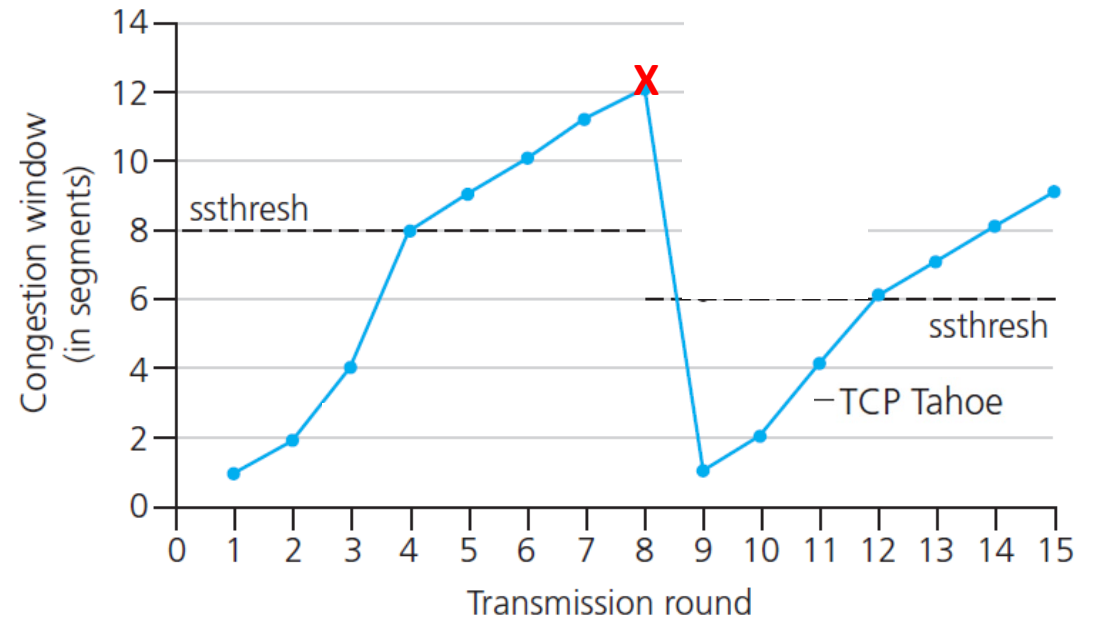
# TCP: from slow start to congestion avoidance

**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



# TCP Tahoe & Reno

