

# ECSE-4670: Computer Communication Networks (CCN)

## Chapter 3: TCP

Shivkumar Kalyanaraman: [shivkuma@ecse.rpi.edu](mailto:shivkuma@ecse.rpi.edu)

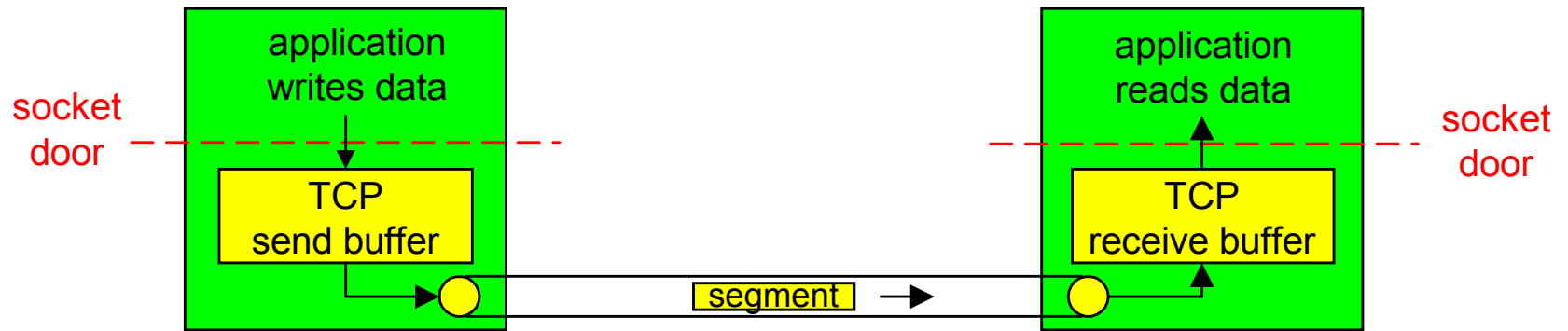
Biplab Sikdar: [sikdab@rpi.edu](mailto:sikdab@rpi.edu)



# TCP: Overview

- **Point-to-point:**
  - one sender, one receiver
- **Reliable, in-order *byte stream*:**
  - no “message boundaries”
  - But TCP chops it up into segments for transmission internally
- **Pipelined (window) flow control:**
  - Window size decided by receiver and network
- ***Send & receive buffers***

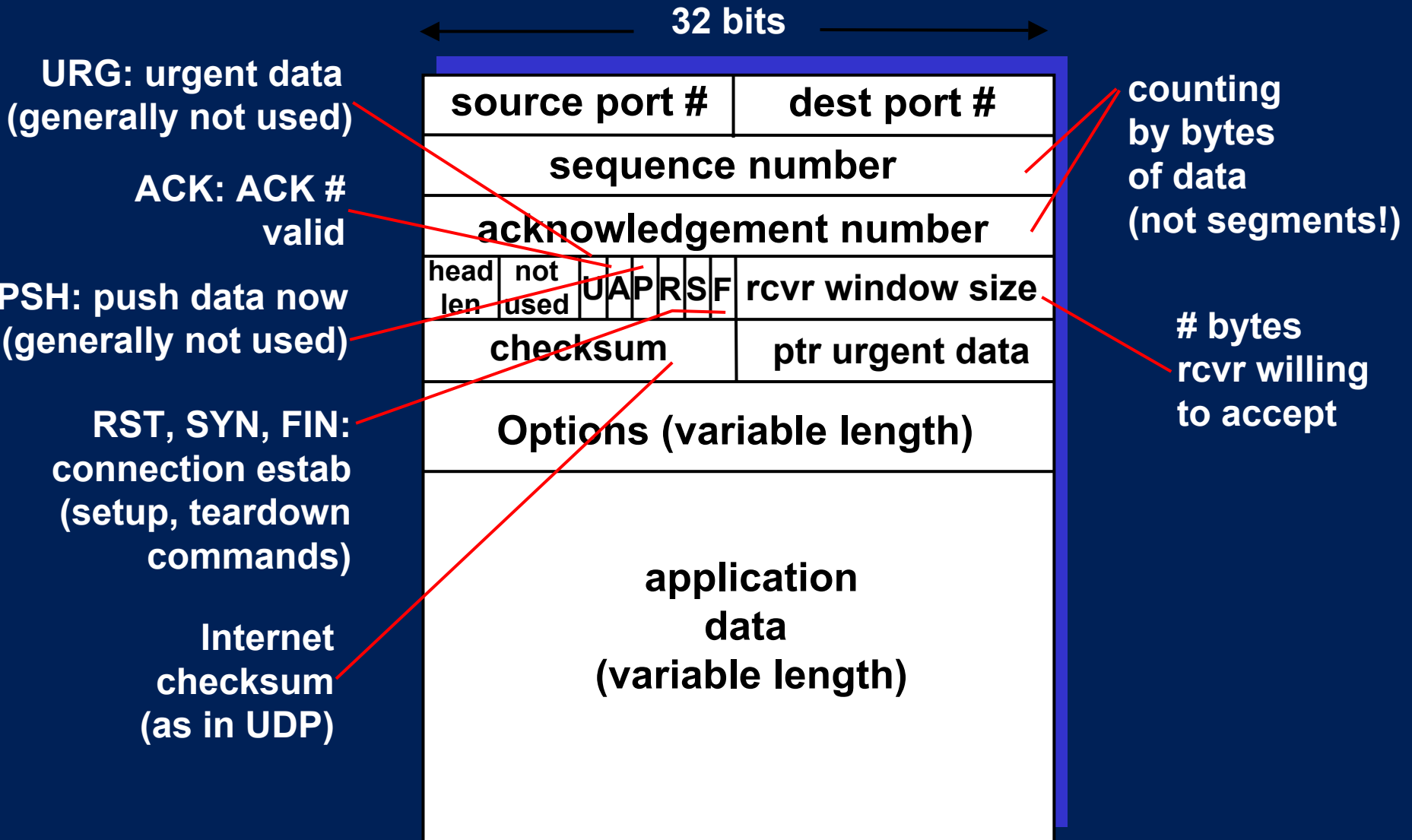
# TCP: Overview



# TCP: Overview

- **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 & Congestion Control:**
  - sender will not overwhelm receiver or the network

# TCP segment structure



# TCP seq. #'s and ACKs (I)

## Sequence Numbers:

- byte stream “number” of **first byte in segment's data**

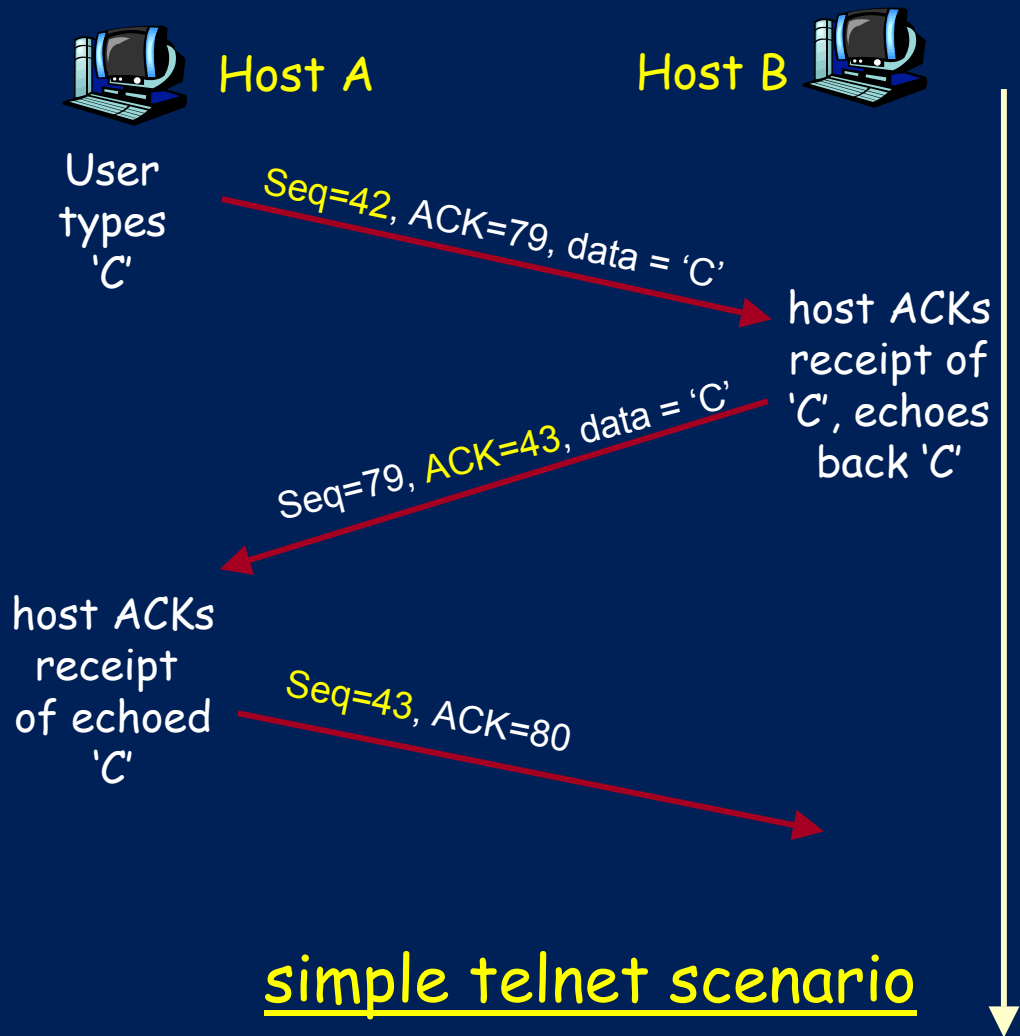
## ACKs:

- seq # of **next byte expected** from other side
- **cumulative** ACK

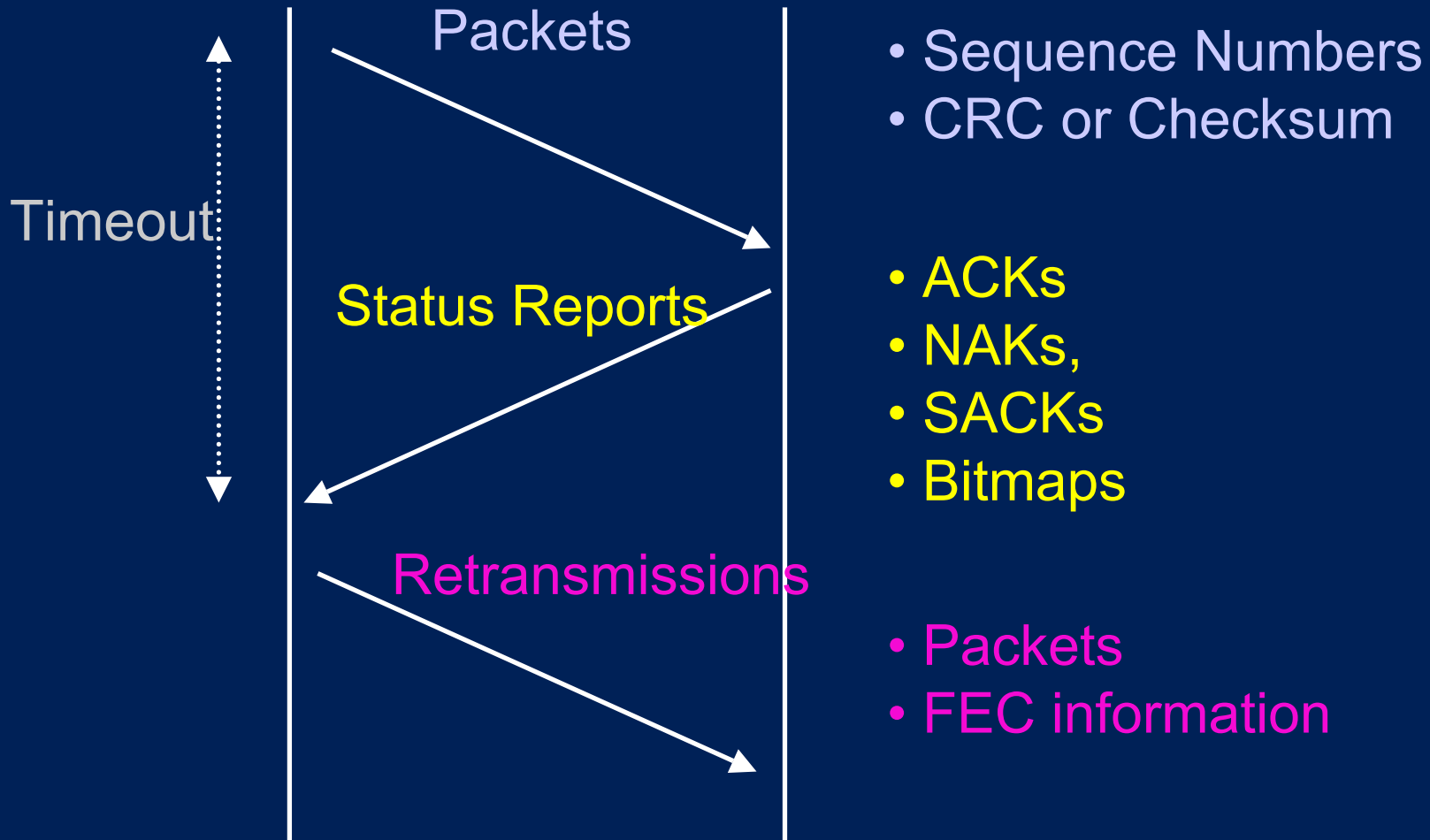
**Q: how receiver handles out-of-order segments**

- **A: TCP spec doesn't say, - up to implementor**

# TCP Seq. #'s and ACKs (II)



# Temporal Redundancy Model





# Status Report Design

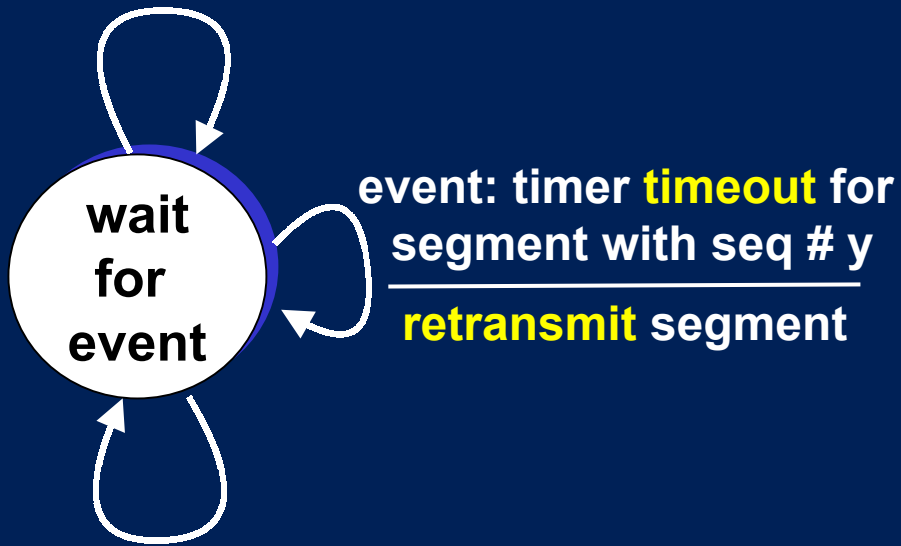
- **Cumulative** acks:
  - **Robust** to losses on the reverse channel
  - Can work with **go-back-N** retransmission
  - Cannot pinpoint **blocks** of data which are lost
    - The first lost packet can be pinpointed because the receiver would generate duplicate acks

# TCP: reliable data transfer (I)

event: data received  
from application above  

---

create, send segment



- one way data transfer
- no flow, congestion control

event: ACK received,  
with ACK # y  

---

ACK processing

# TCP: reliable data transfer (II)

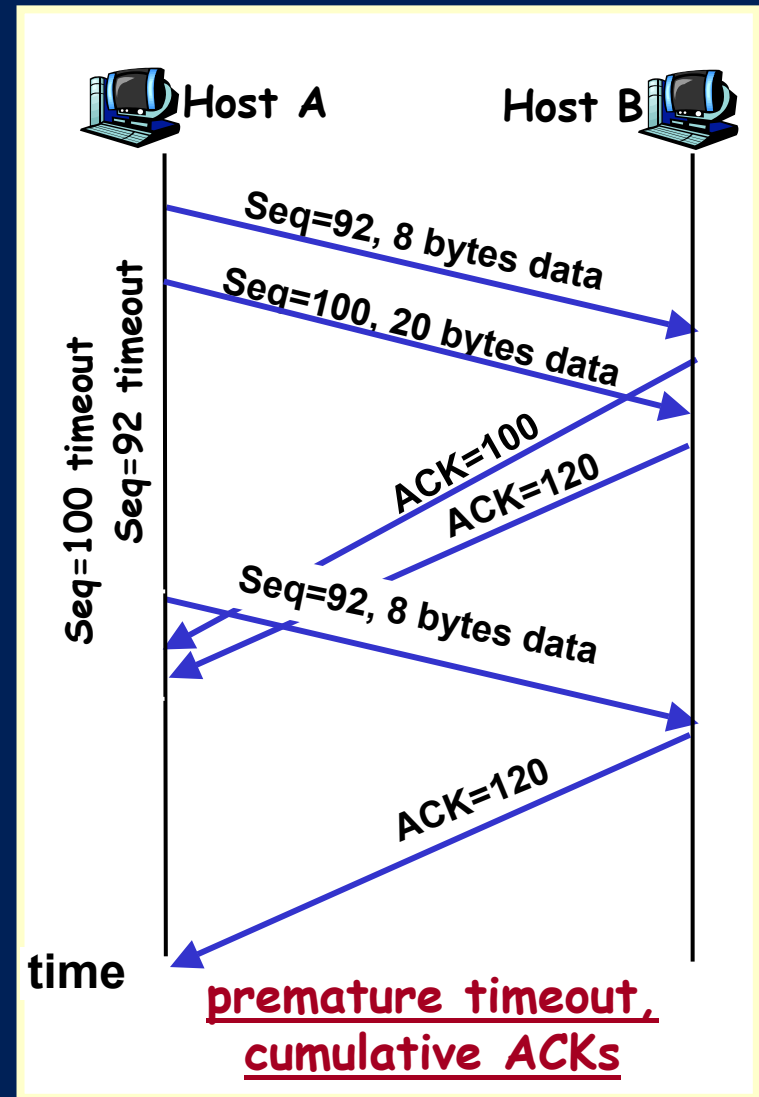
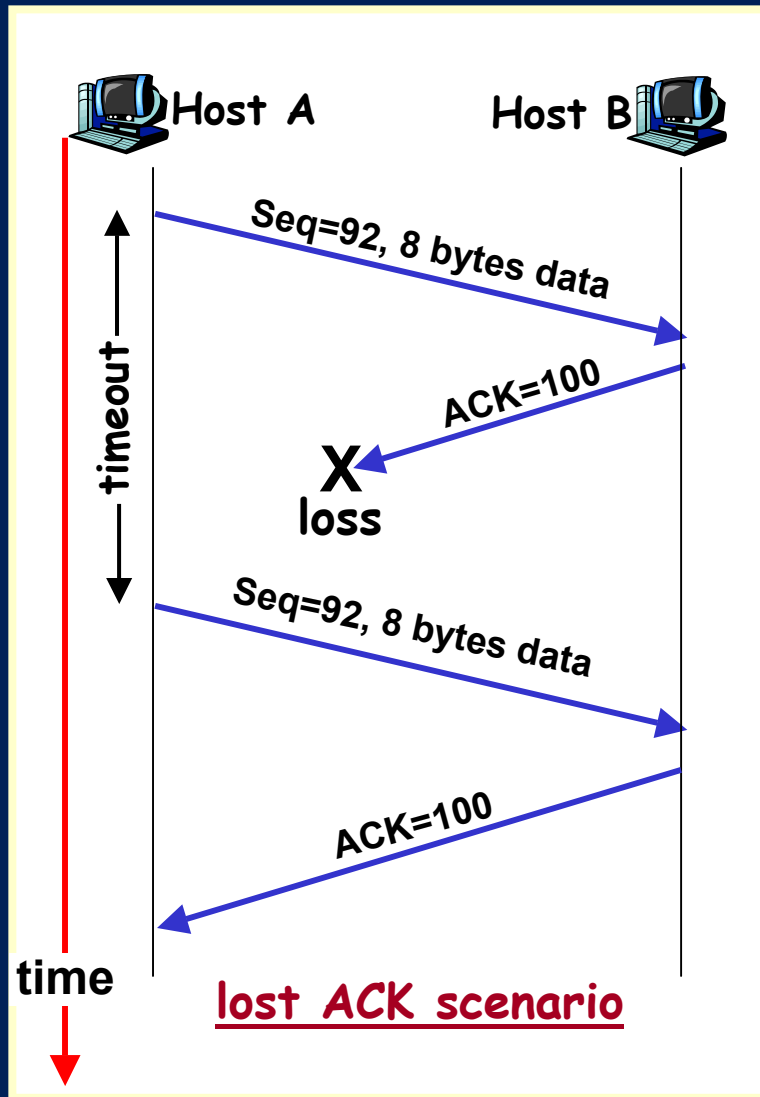
## Simplified TCP sender

```
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

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

# TCP: retransmission scenarios



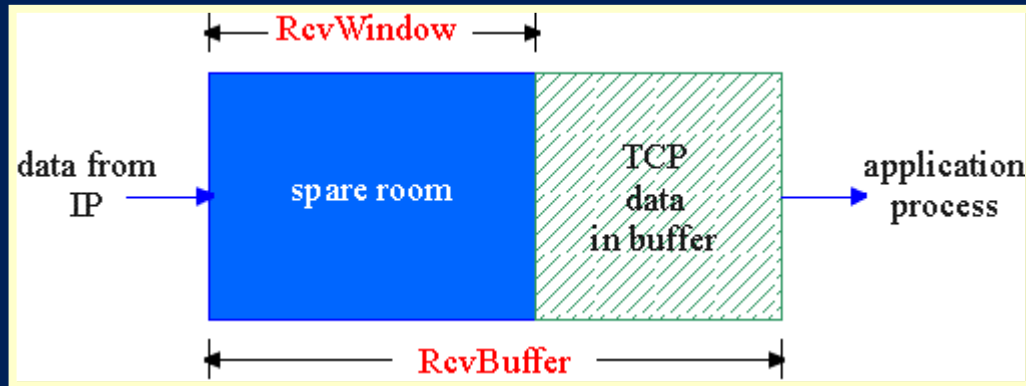
# TCP Flow Control

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

receiver: explicitly informs sender of free buffer space  
– **RcvWindow** field in TCP segment

RcvBuffer = size of TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

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

# Timeout and RTT Estimation

- Timeout: for robust detection of packet loss
- Problem: How long should timeout be ?
  - Too long => underutilization
  - Too short => wasteful retransmissions
  - Solution: adaptive timeout: based on estimate of **max RTT**

# How to estimate max RTT?

- RTT = prop + queuing delay
  - **Queuing delay** highly **variable**
  - So, different samples of RTTs will give different random values of queuing delay
- Chebyshev's Theorem:
  - **MaxRTT = Avg RTT + k\*Deviation**
  - **Error** probability is less than  **$1/(k^2)$**
  - Result true for **ANY** distribution of samples



# Round Trip Time and Timeout (II)

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- SampleRTT will vary wildly => want estimated RTT “smoother”
  - use several recent measurements, not just current SampleRTT to calculate “**AverageRTT**”

# TCP Round Trip Time and Timeout (III)

$$\text{AverageRTT} = (1-x)*\text{AverageRTT} + x*\text{SampleRTT}$$

- Exponential weighted moving average (EWMA)
- influence of given sample decreases exponentially fast;  $x = 0.1$

## Setting the timeout

- AverageRTT plus “safety margin” proportional to variation

$$\text{Timeout} = \text{AverageRTT} + 4*\text{Deviation}$$

$$\text{Deviation} = (1-x)*\text{Deviation} + x*|\text{SampleRTT} - \text{AverageRTT}|$$

# TCP Connection Management - 1

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();
```

# TCP Connection Management - 2

## 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 - 3

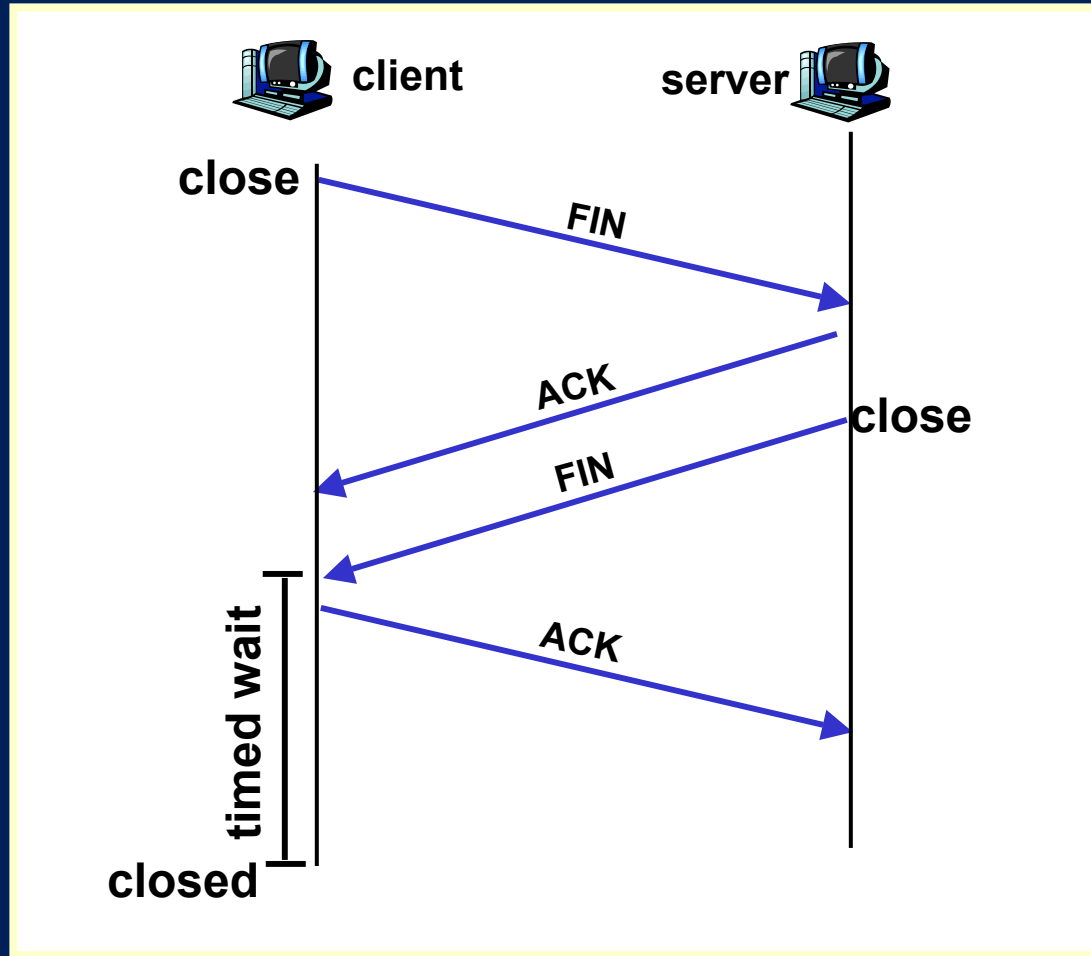
## Closing a connection:

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 - 4



# TCP Connection Management - 5

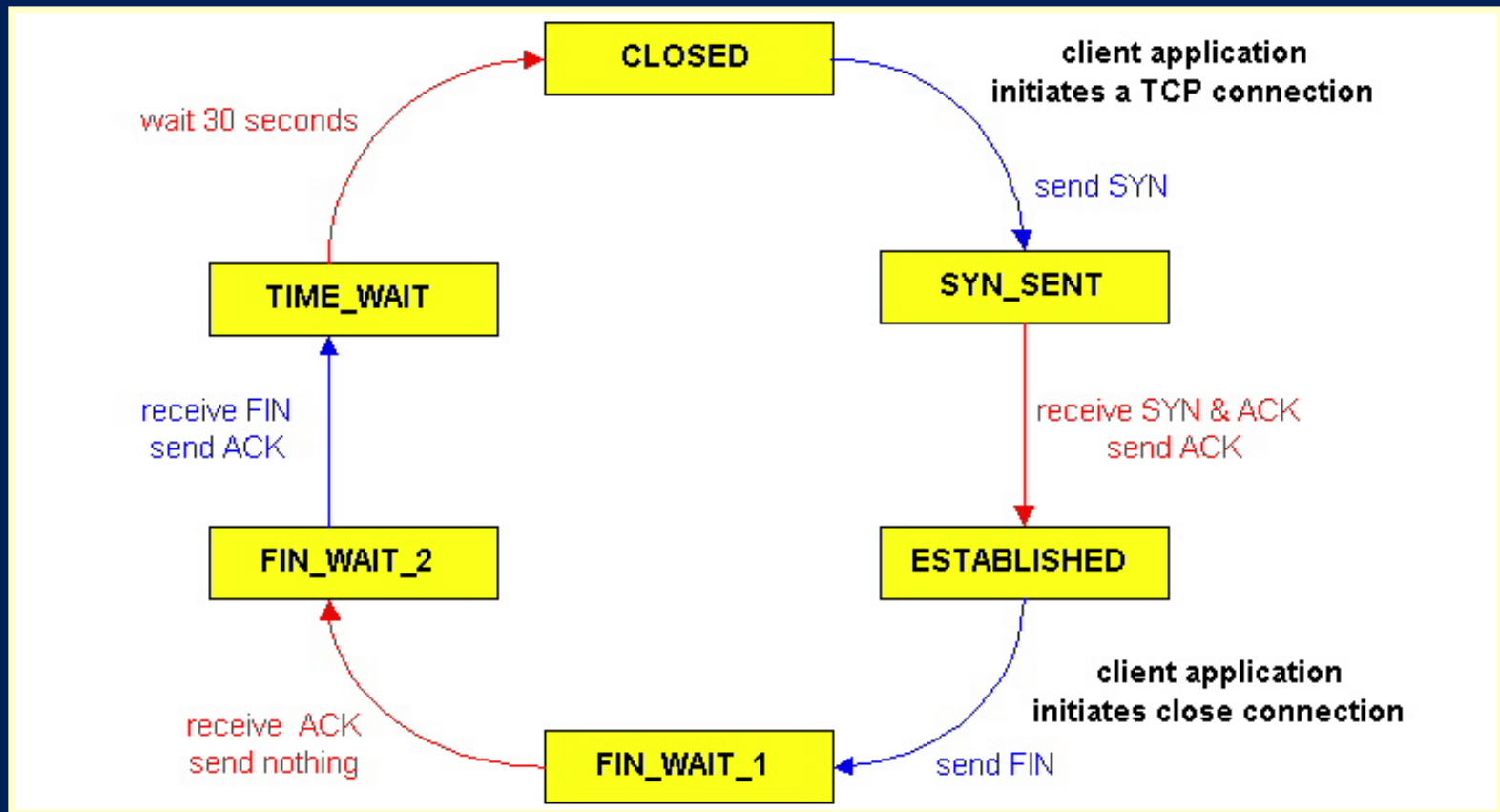
**Step 3:** client receives FIN, replies with ACK.

- 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.

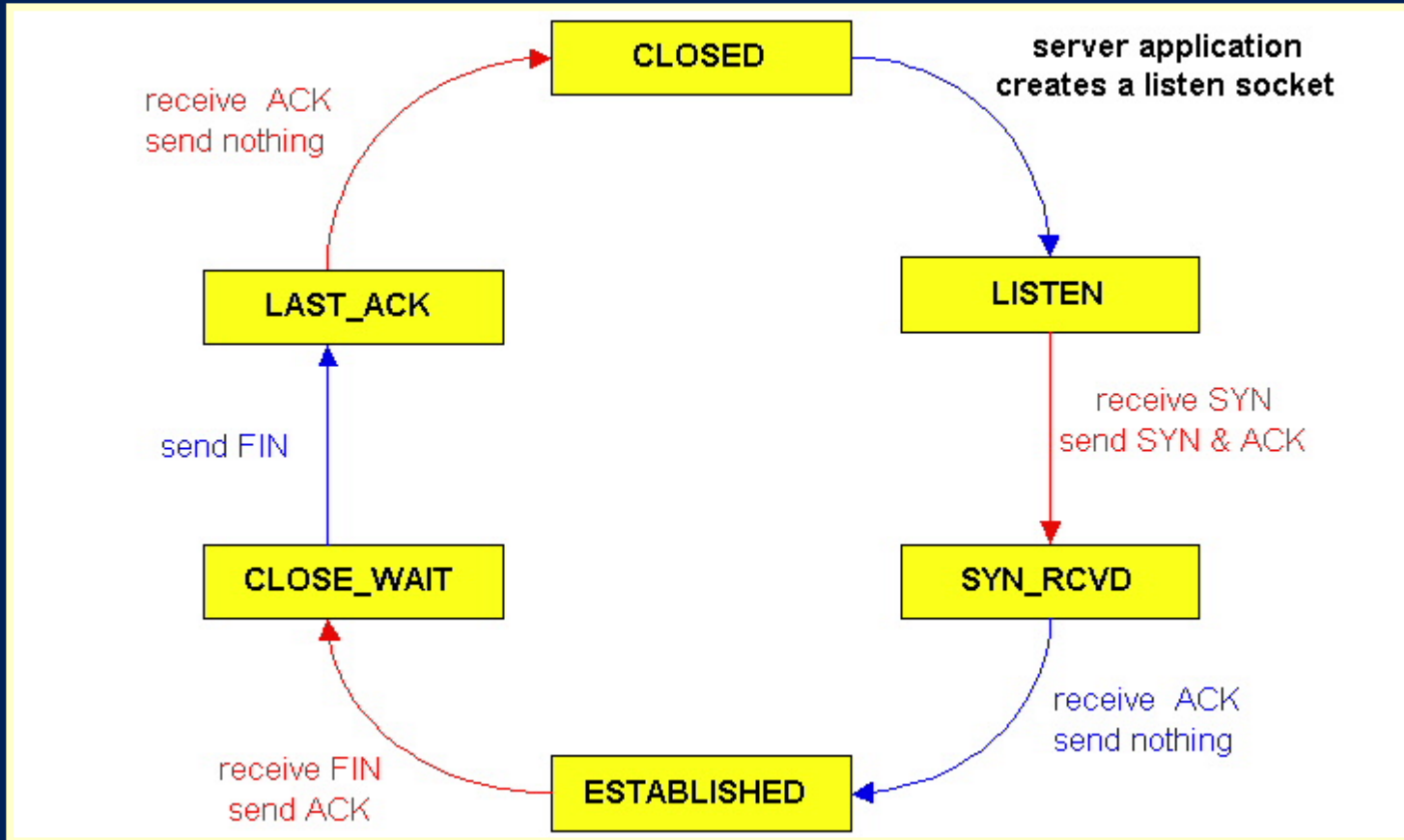
# TCP Connection Management - 6



## TCP client lifecycle



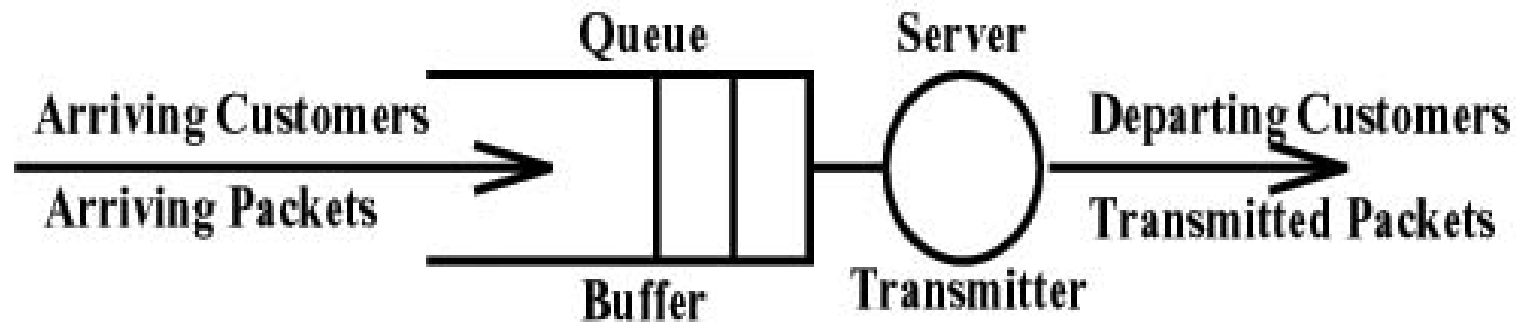
# TCP Connection Management - 7



## TCP server lifecycle

# Recap: Stability of a Multiplexed System

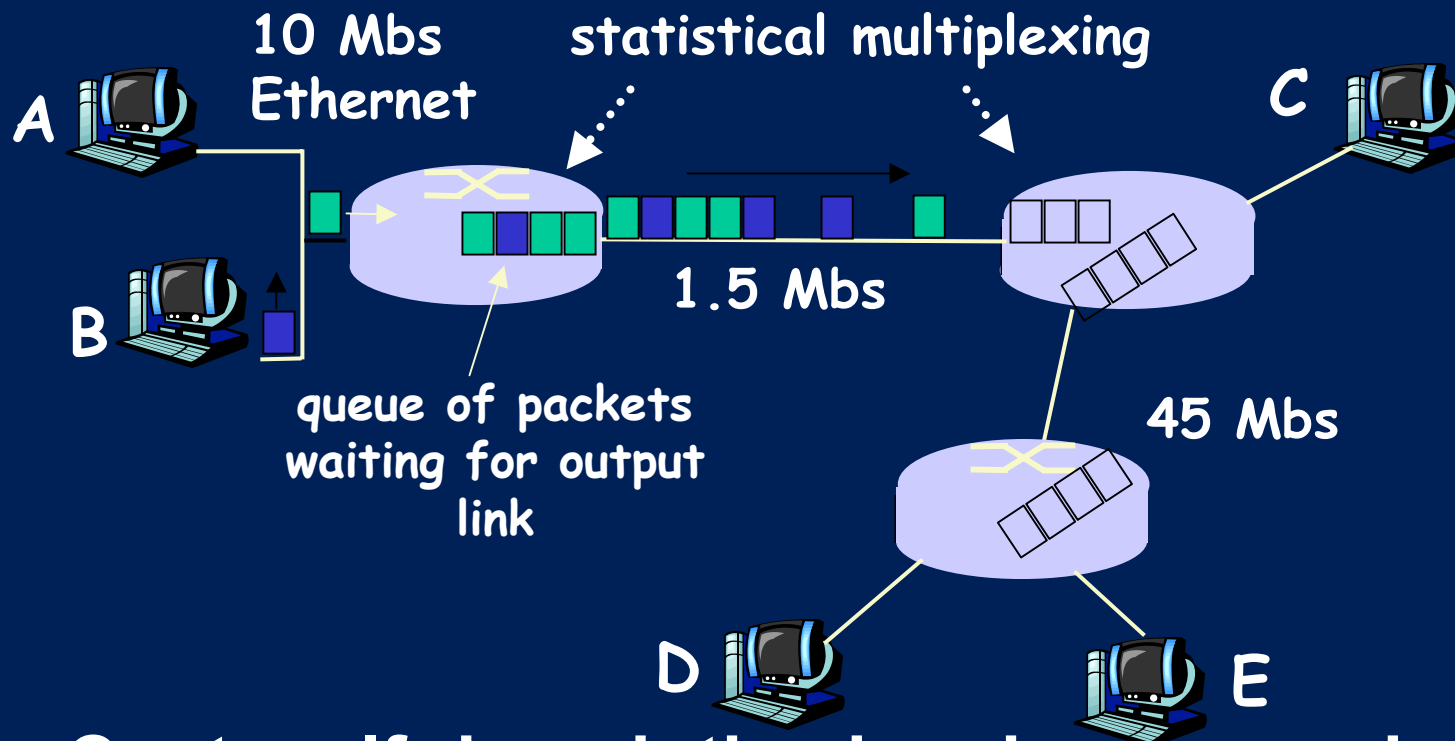
Average Input Rate  $>$  Average Output Rate  
 $\Rightarrow$  system is unstable!



How to ensure stability ?

1. Reserve enough capacity so that demand is less than reserved capacity
2. Dynamically detect overload and adapt either the demand or capacity to resolve **overload**

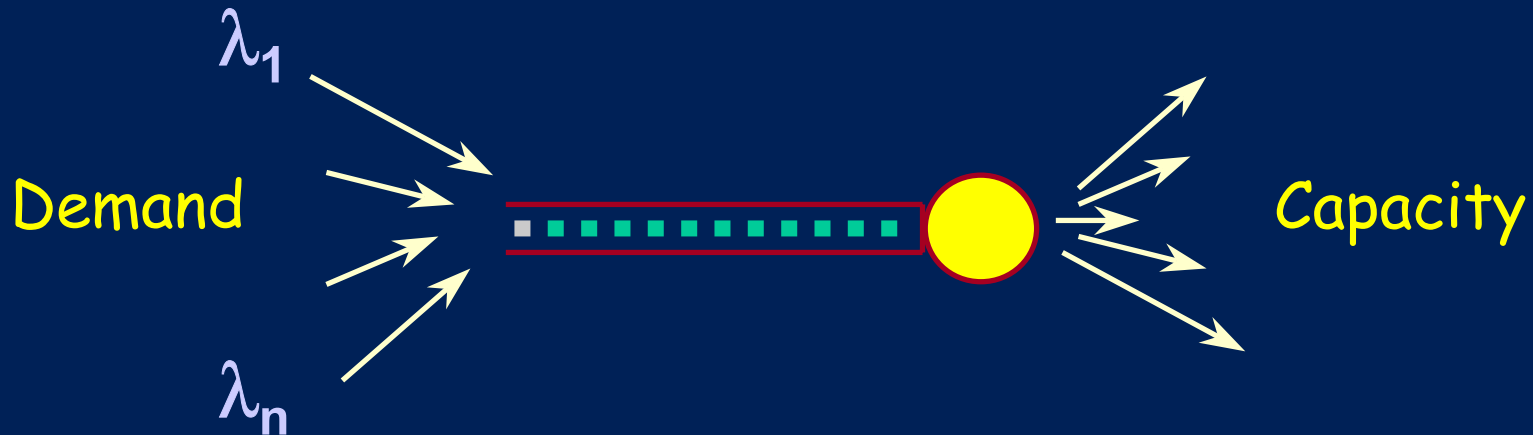
# Congestion Problem in Packet Switching



- ❑ Cost: self-descriptive header per-packet, buffering and delays for applications.
- ❑ Need to *either reserve resources or dynamically detect/adapt to overload for stability*

# The Congestion Problem

- **Problem:** demand outstrips available capacity



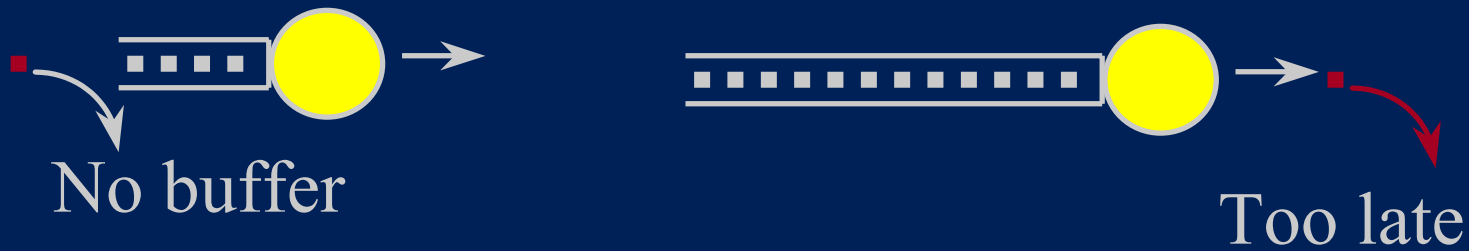
- If information about  $\lambda_i$ ,  $\lambda$  and  $\mu$  is known in a central location where control of  $\lambda_i$  or  $\mu$  can be effected with zero time delays,
  - the congestion problem is solved!

# The Congestion Problem (Continued)

- **Problems:**
  - **Incomplete information (eg: loss indications)**
  - **Distributed solution required**
  - **Congestion and control/measurement locations different**
  - **Time-varying, heterogeneous time-delay**

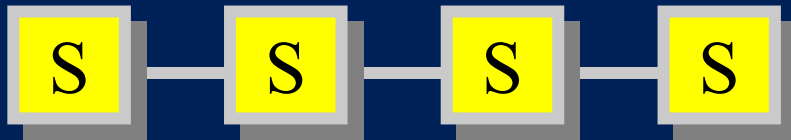
# The Congestion Problem

- Static fixes may not solve congestion
  - a) Memory becomes cheap (infinite memory)



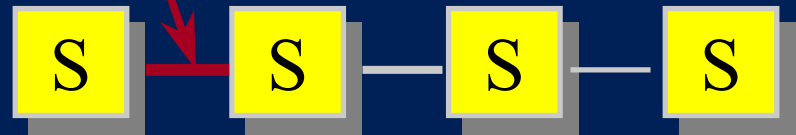
- b) Links become cheap (high speed links)?

All links 19.2 kb/s



*File Transfer time = 5 mins*

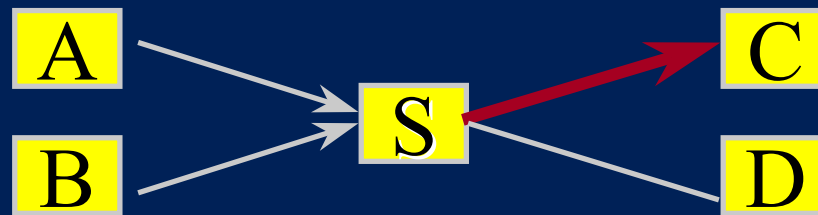
Replace with 1 Mb/s



*File Transfer Time = 7 hours*

# The Congestion Problem (Continued)

- c) Processors become cheap  
(fast routers & switches)



*Scenario: All links 1 Gb/s.*

*A & B send to C*

*=> “high-speed” congestion!!  
(lose more packets faster!)*

# Principles of Congestion Control

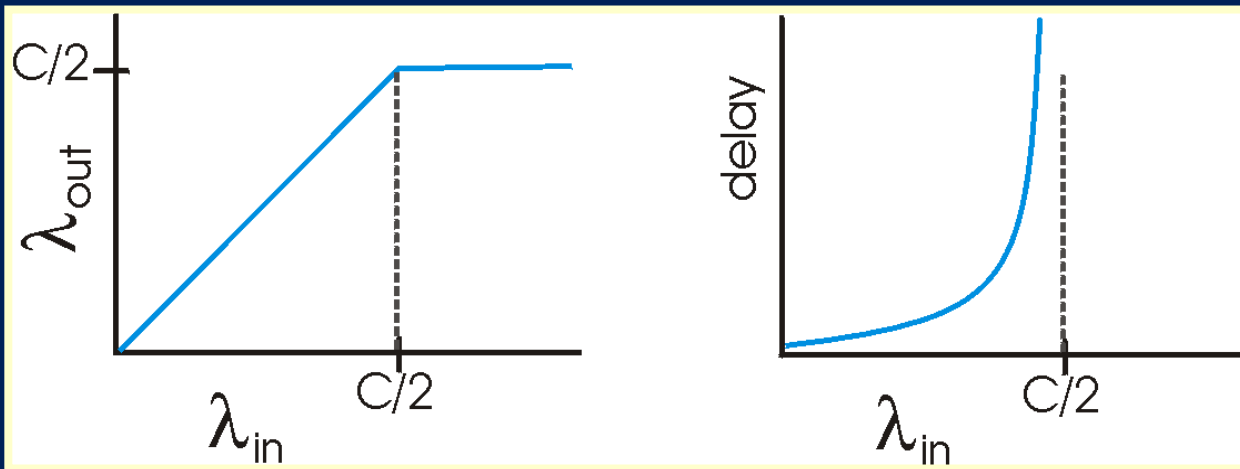
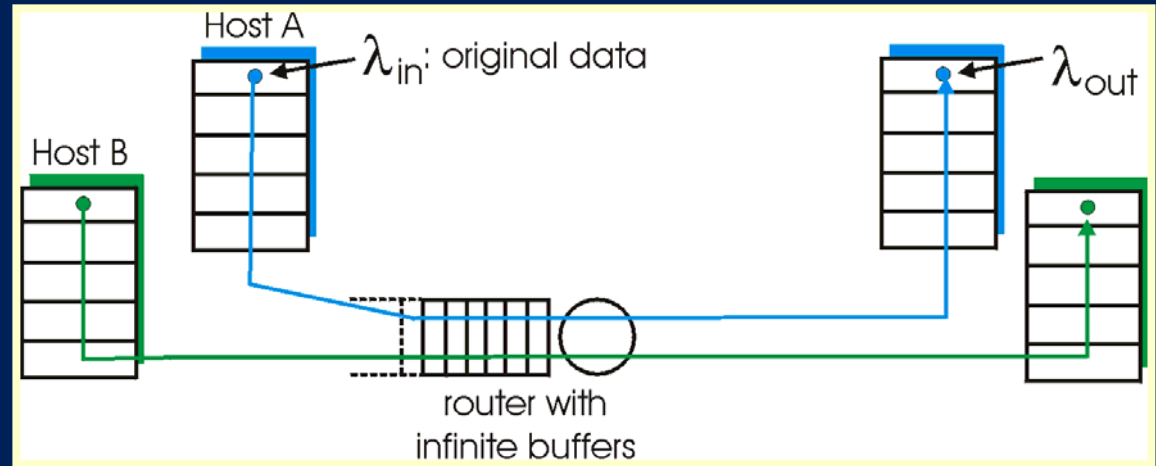
## Congestion:

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



# Causes/costs of congestion: scenario 1

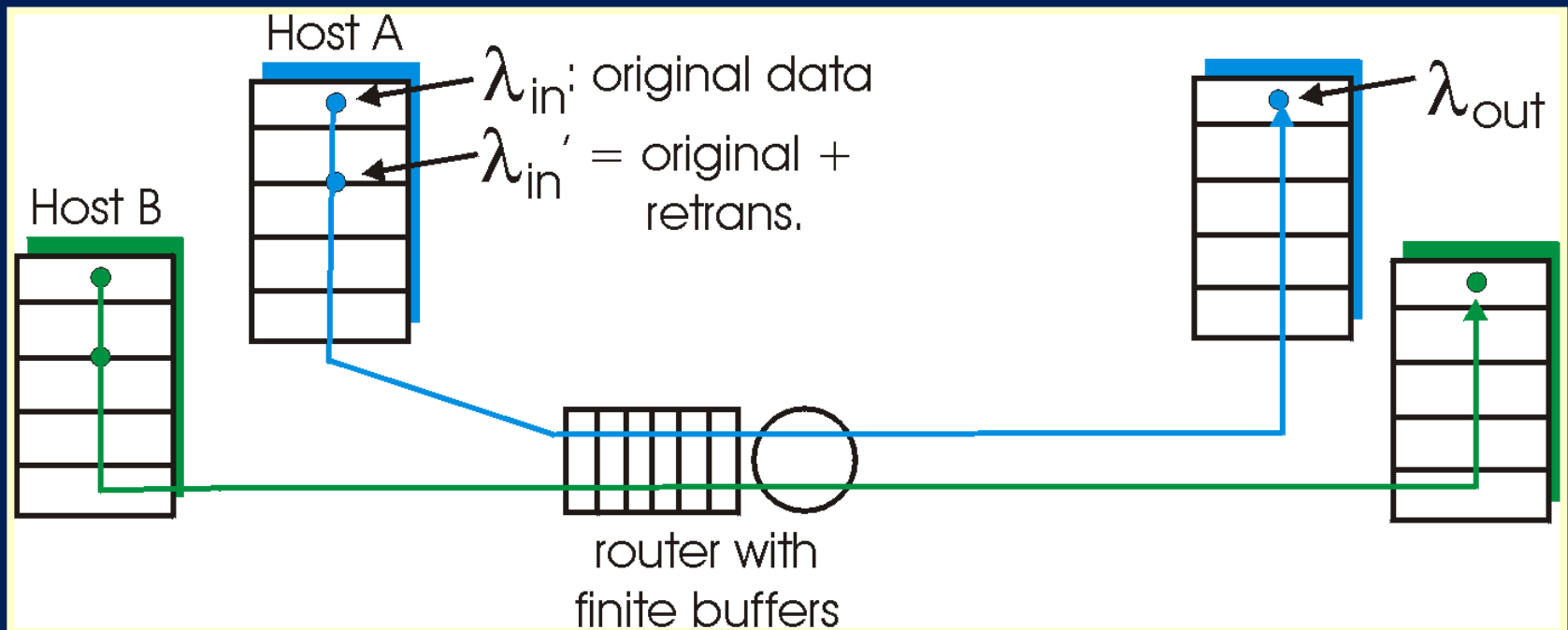
- two senders, two receivers
- one router, infinite buffers
- no retransmission



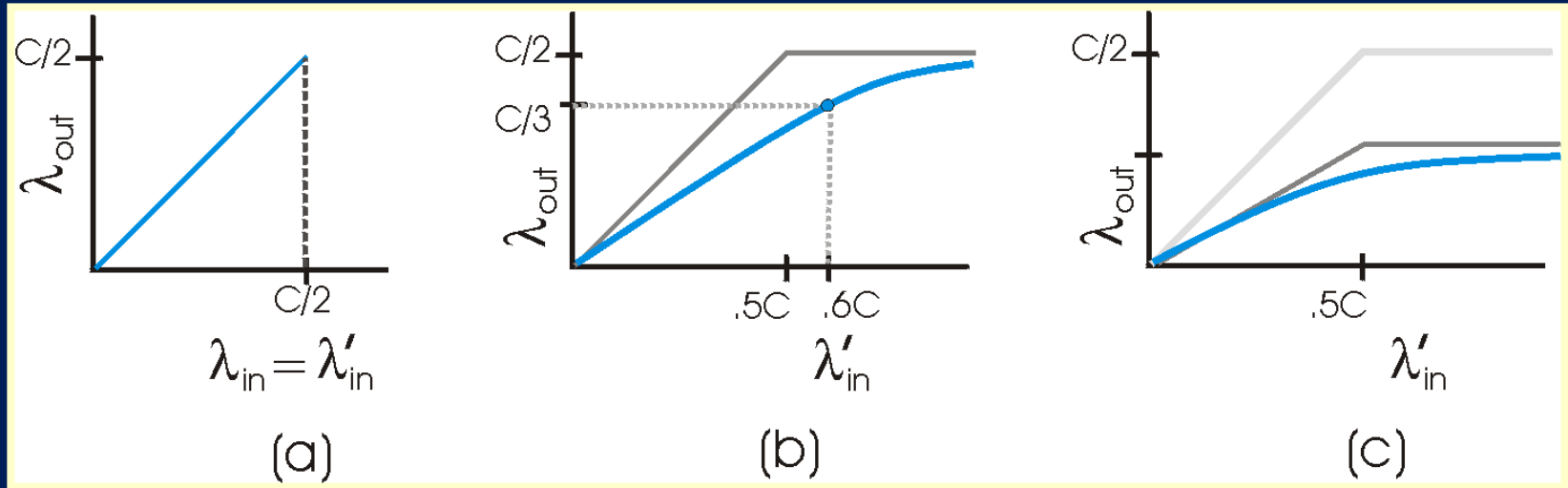
- large delays when congested
- maximum achievable throughput

# Causes/costs of congestion: scenario 2

- one router, **finite** buffers
- sender retransmission of lost packet



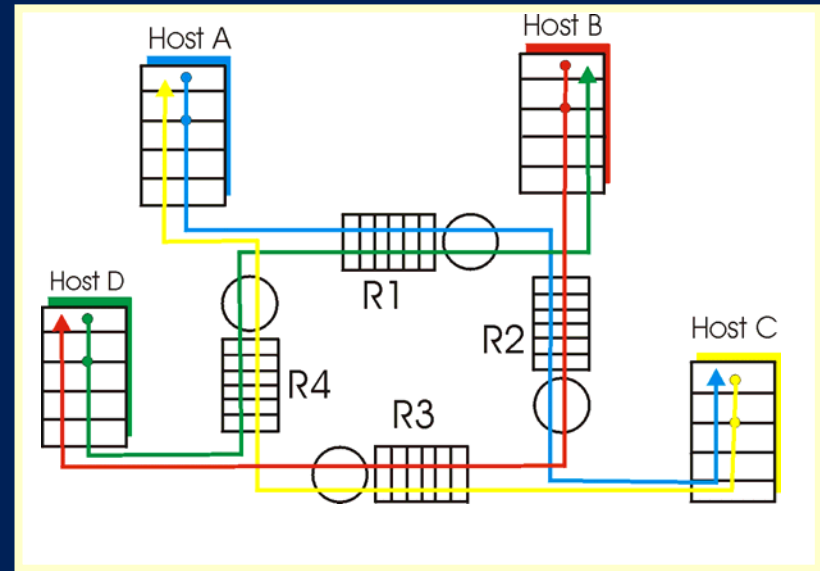
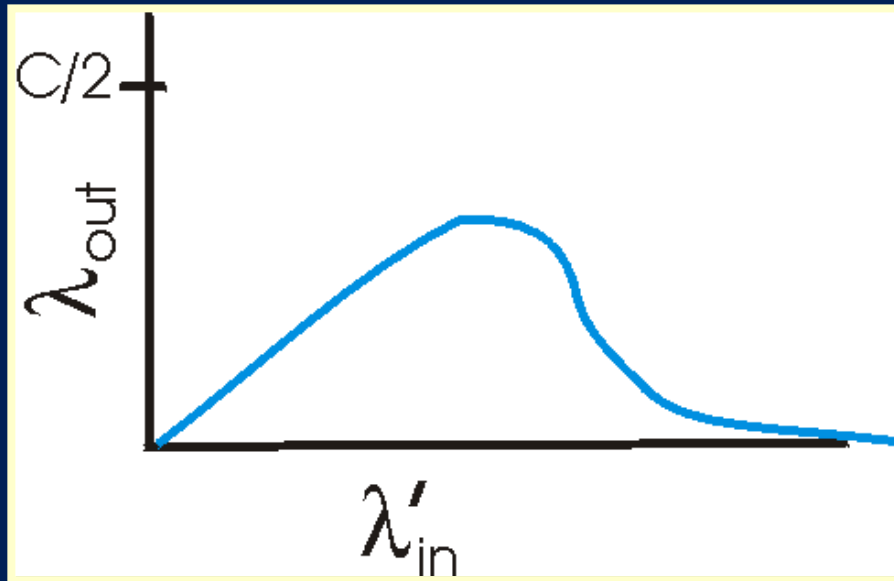
# Causes/costs of congestion: scenario 2 (continued)



## “Costs” of congestion:

- More work (*retrans*) for given “goodput”
- *Unneeded retransmissions*: link carries multiple copies of pkt due to *spurious timeouts*

# Causes/costs of congestion: scenario 3



## Another “cost” of congestion:

- when packet dropped, any “*upstream transmission capacity used for that packet was wasted!*”

# Approaches towards congestion control - 1

- 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

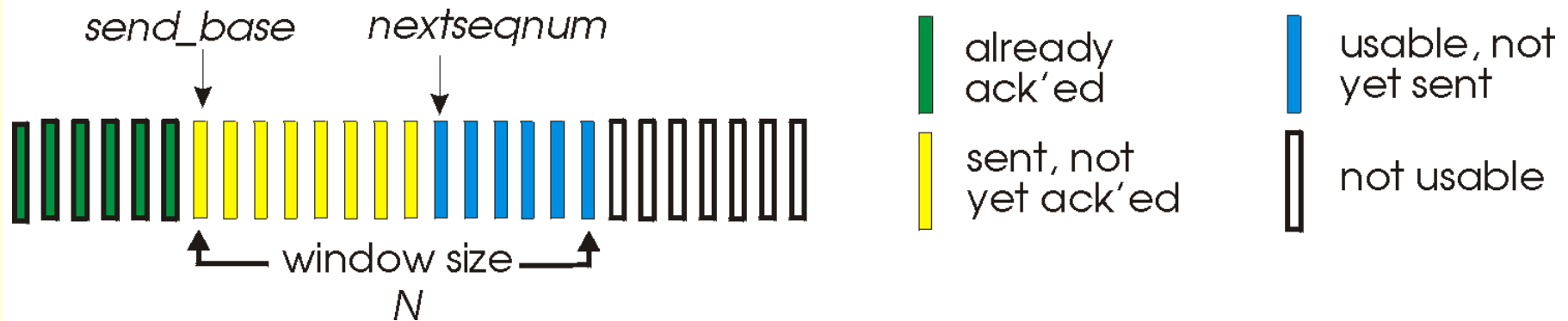
# Approaches towards congestion control - 2

## *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

# TCP congestion control - 1

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



# TCP congestion control - 2

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

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



# TCP congestion control - 3

- “Probing” for usable bandwidth:
  - **Window flow control:** avoid receiver overrun
  - **Dynamic window congestion control:** avoid/control network overrun
- **Policy:**
  - **Increase** Congwin until loss (congestion)
  - Loss => **decrease** Congwin, then begin probing (increasing) again

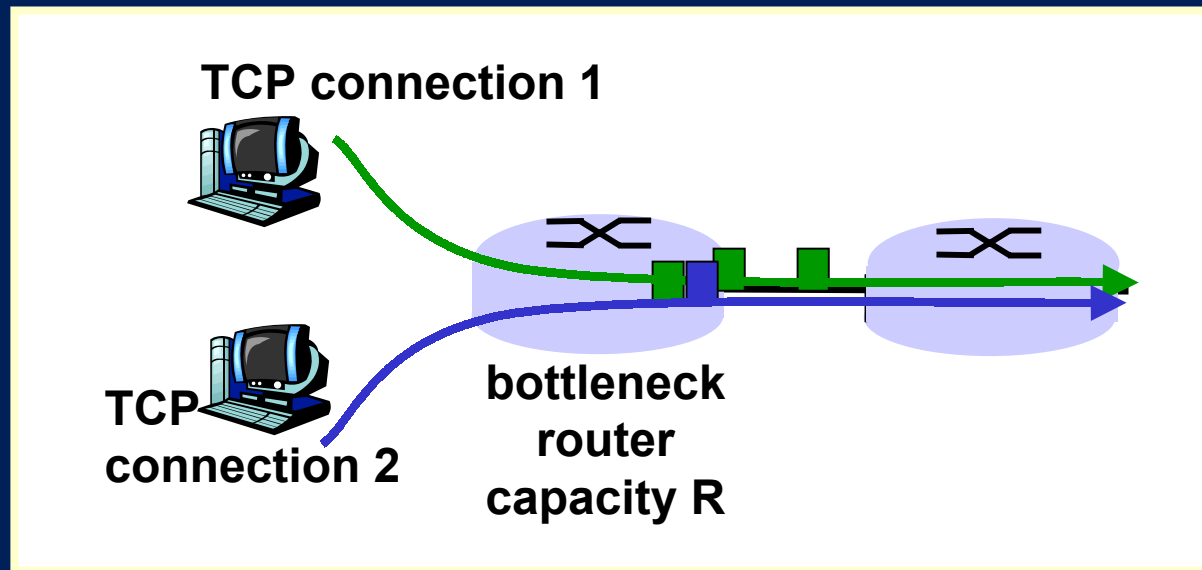
# Additive Increase/Multiplicative Decrease (AIMD) Policy

- For stability:
  - rate-of-**decrease**  $>$  rate-of-**increase**
  - Decrease performed “**enough**” times as long as congestion exists
- AIMD policy satisfies this condition, provided packet loss is congestion indicator

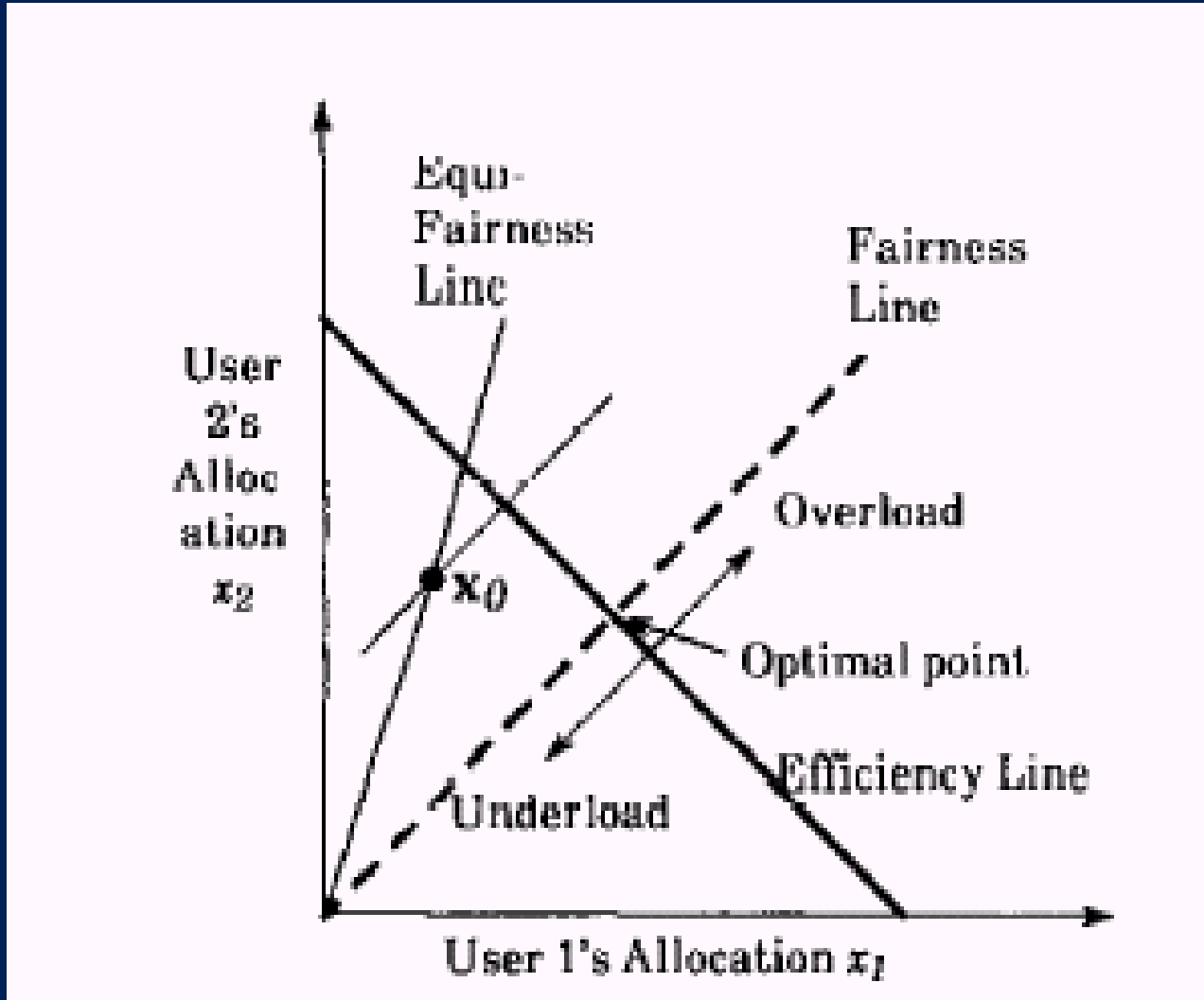


# Fairness

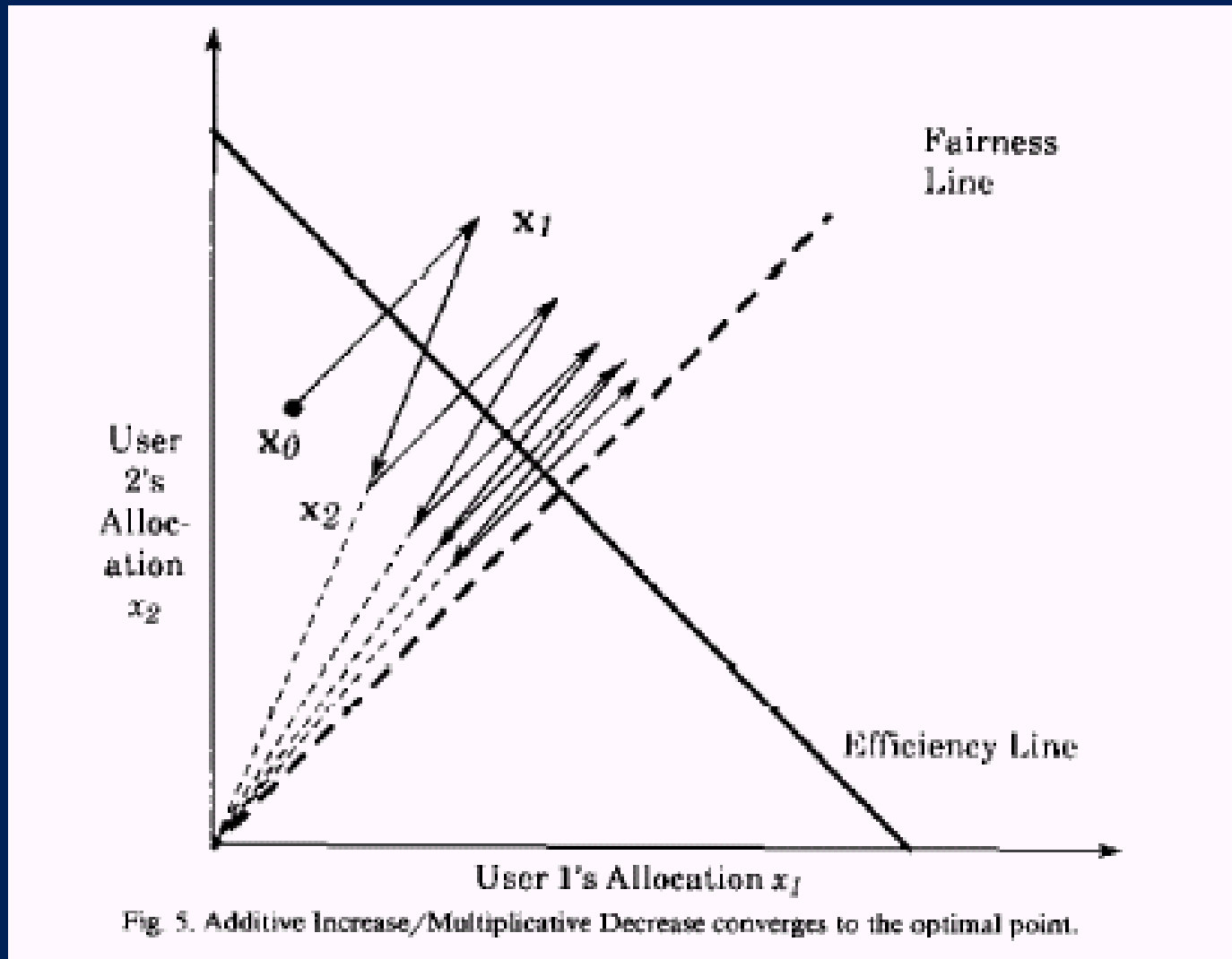
**Fairness goal: if  $N$  TCP sessions share same bottleneck link, each should get  $1/N$  of link capacity**



# Fairness Analysis



# AIMD Converges to Fairness



# TCP congestion control - 4

- TCP uses AIMD policy in “steady state”
- Two “phases”
  - Transient phase: aka “**Slow start**”
  - Steady State: aka “**Congestion avoidance**”
- Important variables:
  - **Congwin**
  - **threshold**: defines threshold between two slow start phase, congestion avoidance phase

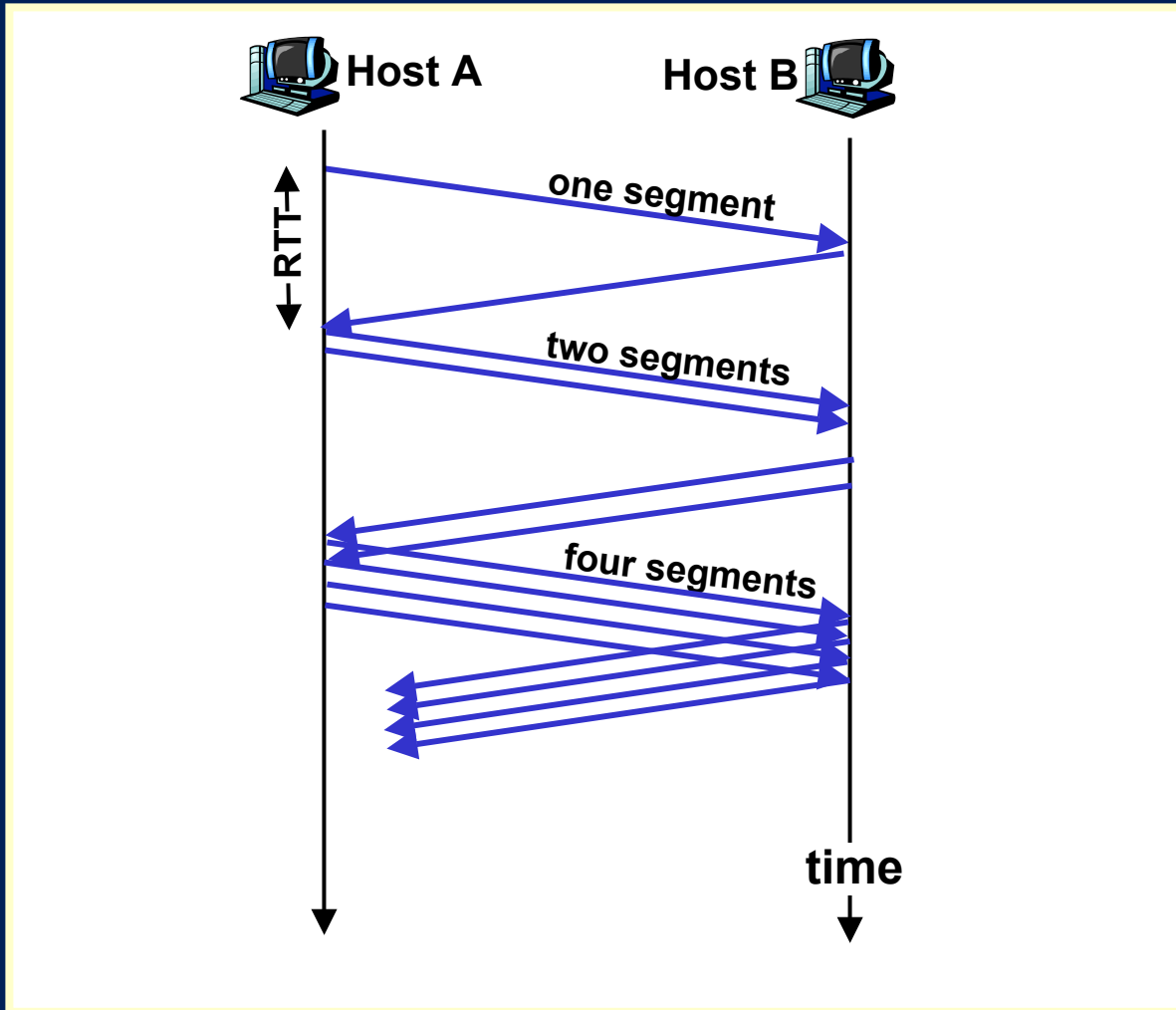
# TCP Slowstart - 1

## 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 three duplicate ACKs (Reno TCP)

# TCP Slowstart - 2

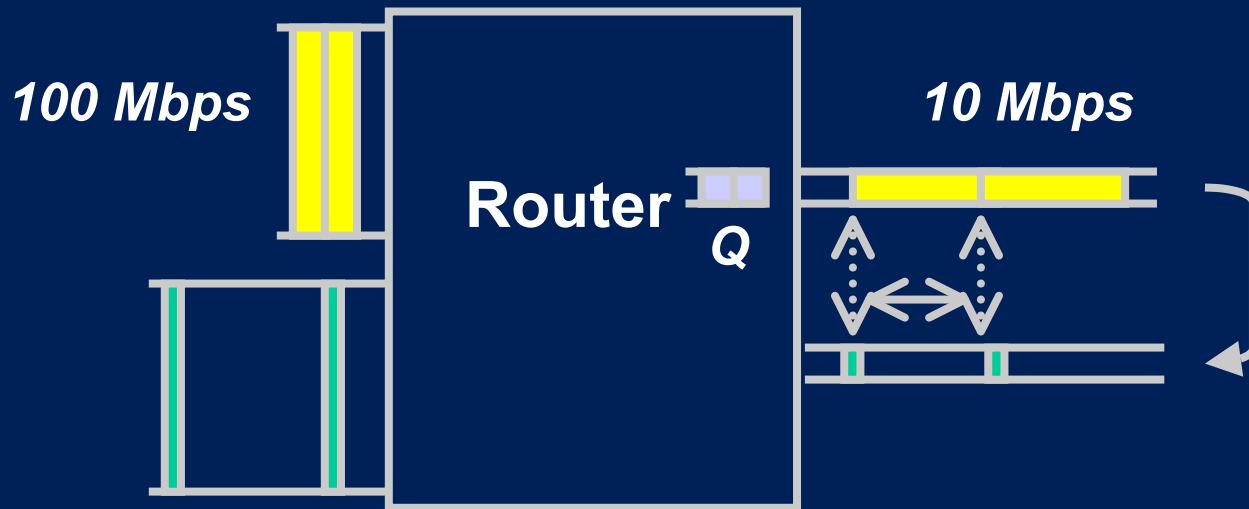




# TCP Dynamics



- Rate of acks determines rate of packets : “**Self-clocking**” property.



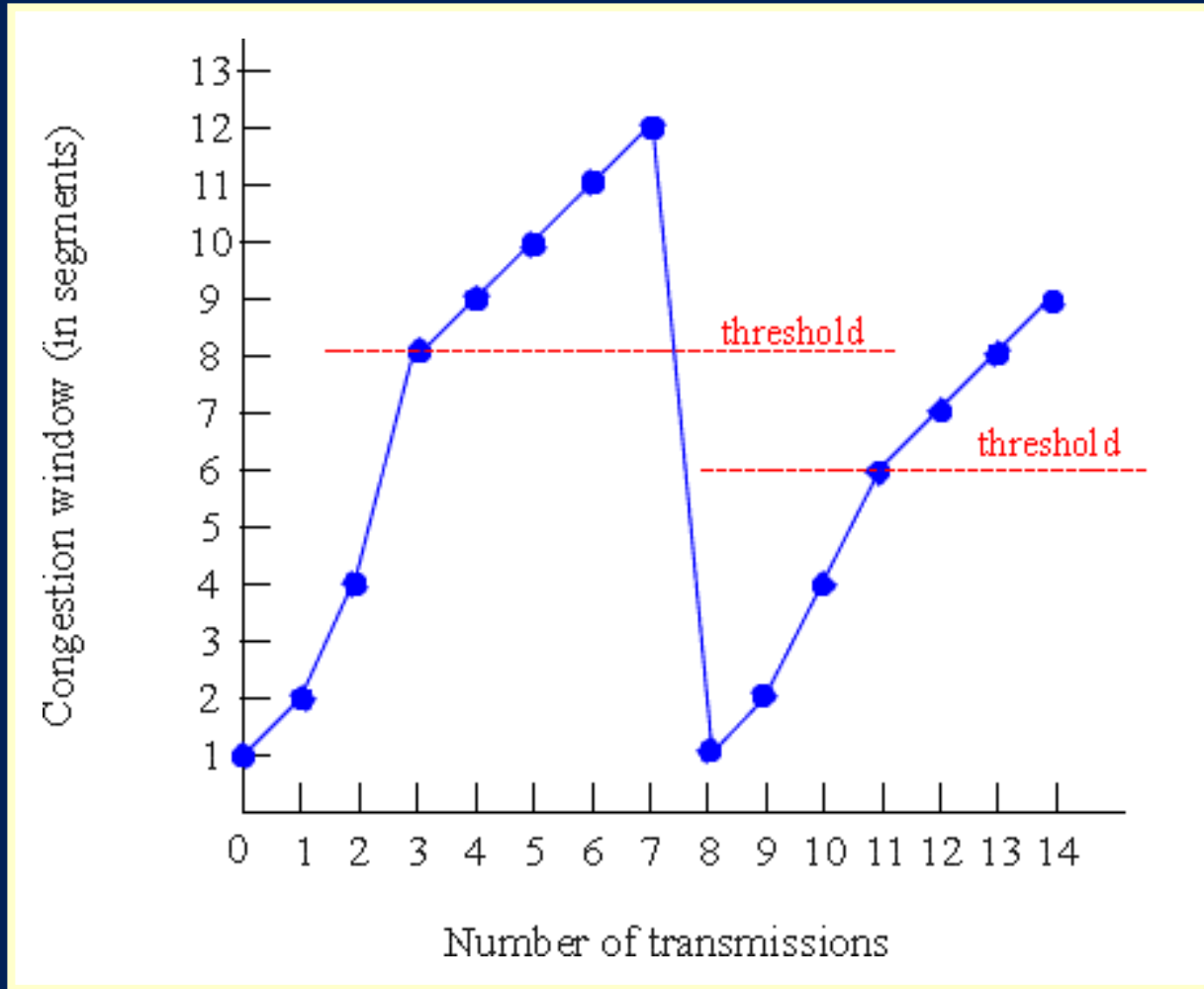
# TCP Congestion Avoidance - 1

## Congestion avoidance

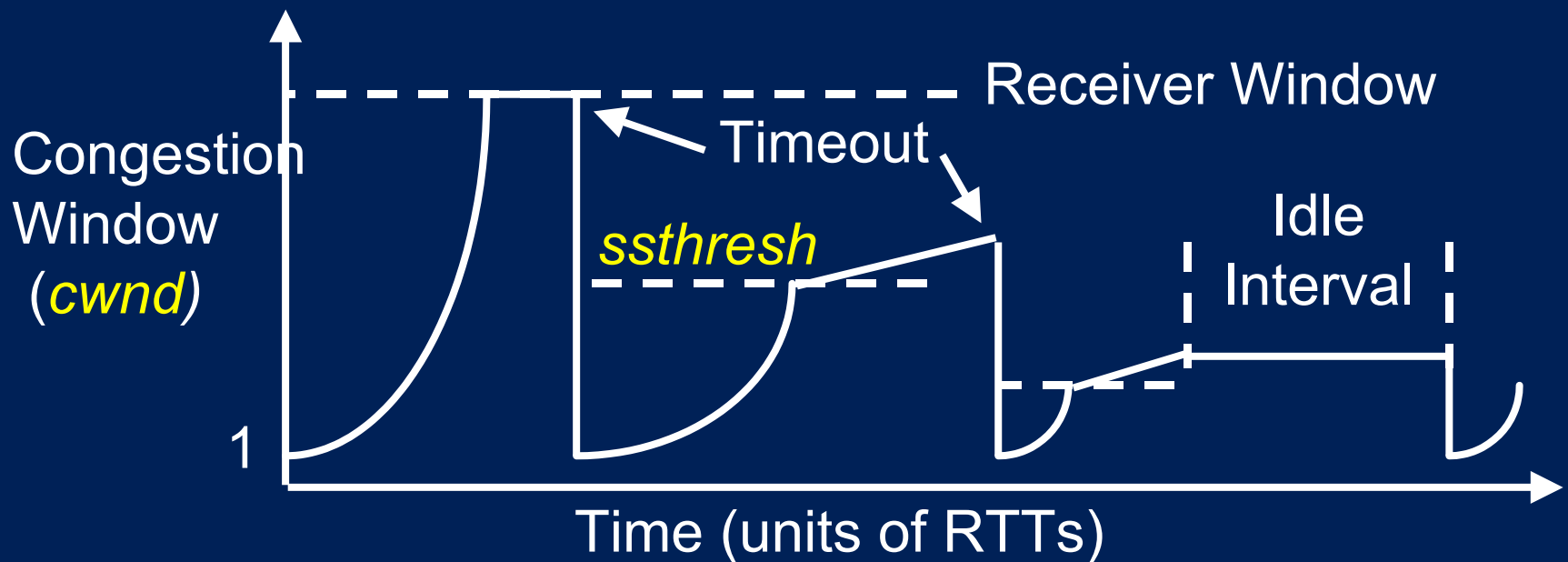
```
/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
    every w segments ACKed:
        Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
```

1: TCP Reno skips slowstart (aka *fast recovery*) after three duplicate ACKs and performs close to AIMD

# TCP Congestion Avoidance - 2



# TCP window dynamics (more)



# TCP latency modeling - 1

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

- TCP connection establishment
- data transfer delay

# TCP latency modeling - 2

## Notation, assumptions:

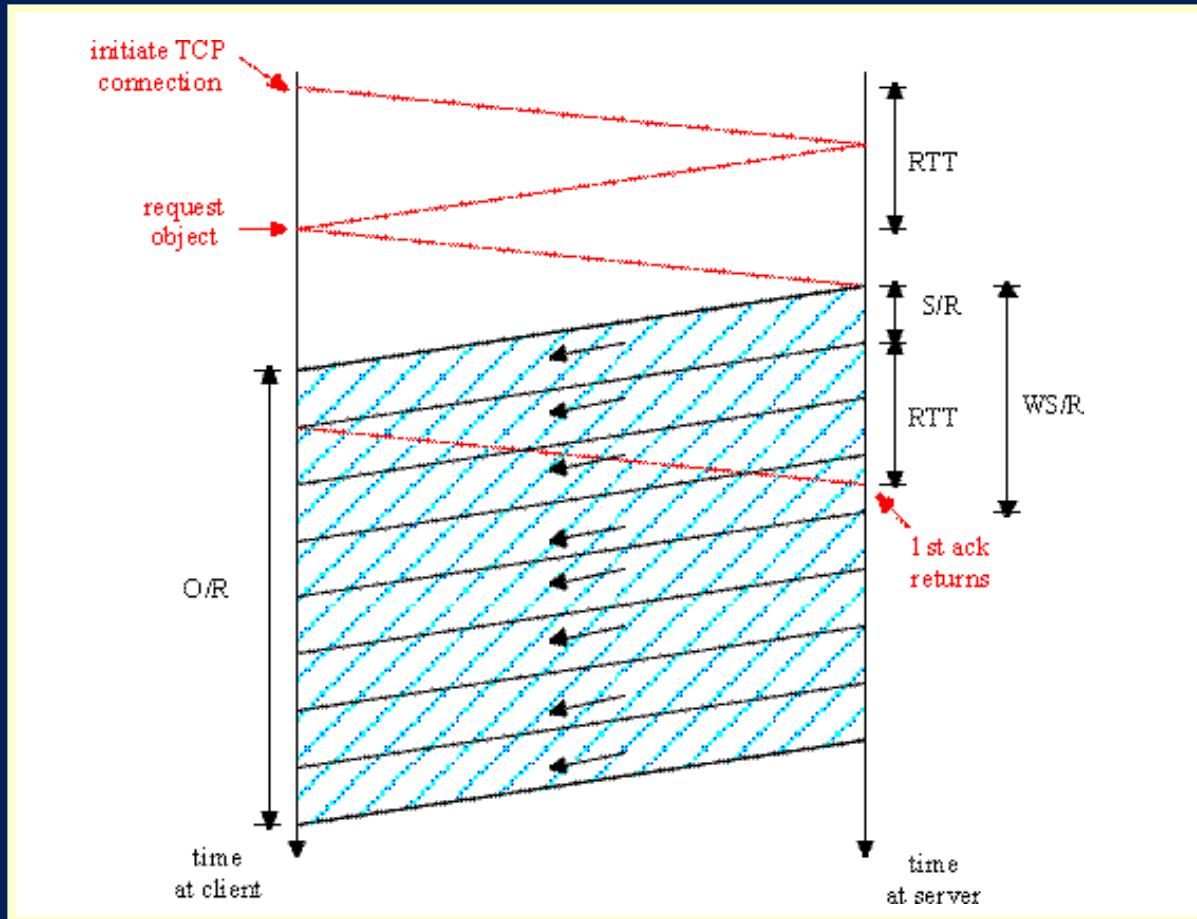
- Assume one link between client and server of rate **R**
- Assume: fixed congestion window, **W** segments
- **S**: MSS (bits)
- **O**: object size (bits)
- no retransmissions (no loss, no corruption)

# TCP latency modeling - 3

## Two cases to consider:

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

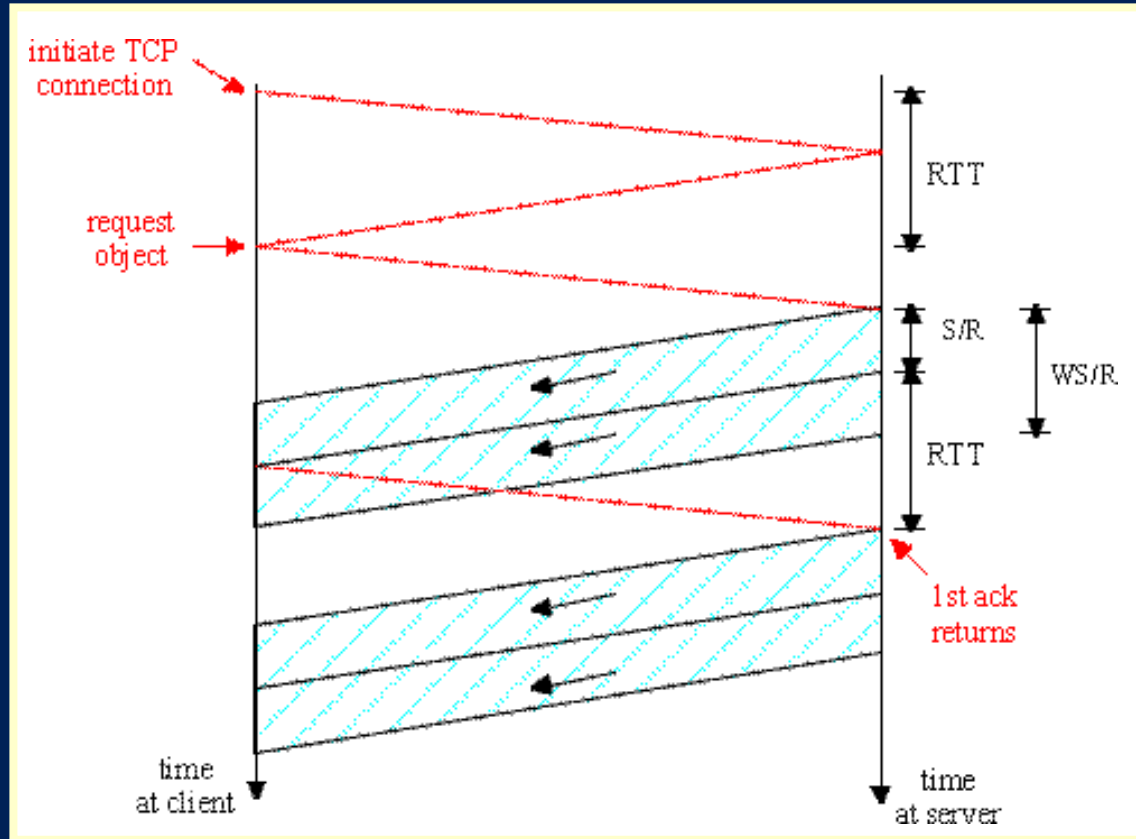
# TCP latency modeling - 4



**Case 1: latency =  $2RTT + O/R$**



# TCP latency modeling - 5



$$K = O/WS$$

$$\text{Case 2: latency} = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]$$

# TCP latency modeling: slow start - 1

- 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:

# TCP latency modeling: slow start - 2

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

- where **Q** is the **number of times the server would stall** if the object were of **infinite size**.
- and **K** is the number of windows that cover the object.

# TCP latency modeling: slow start - 3

Example:

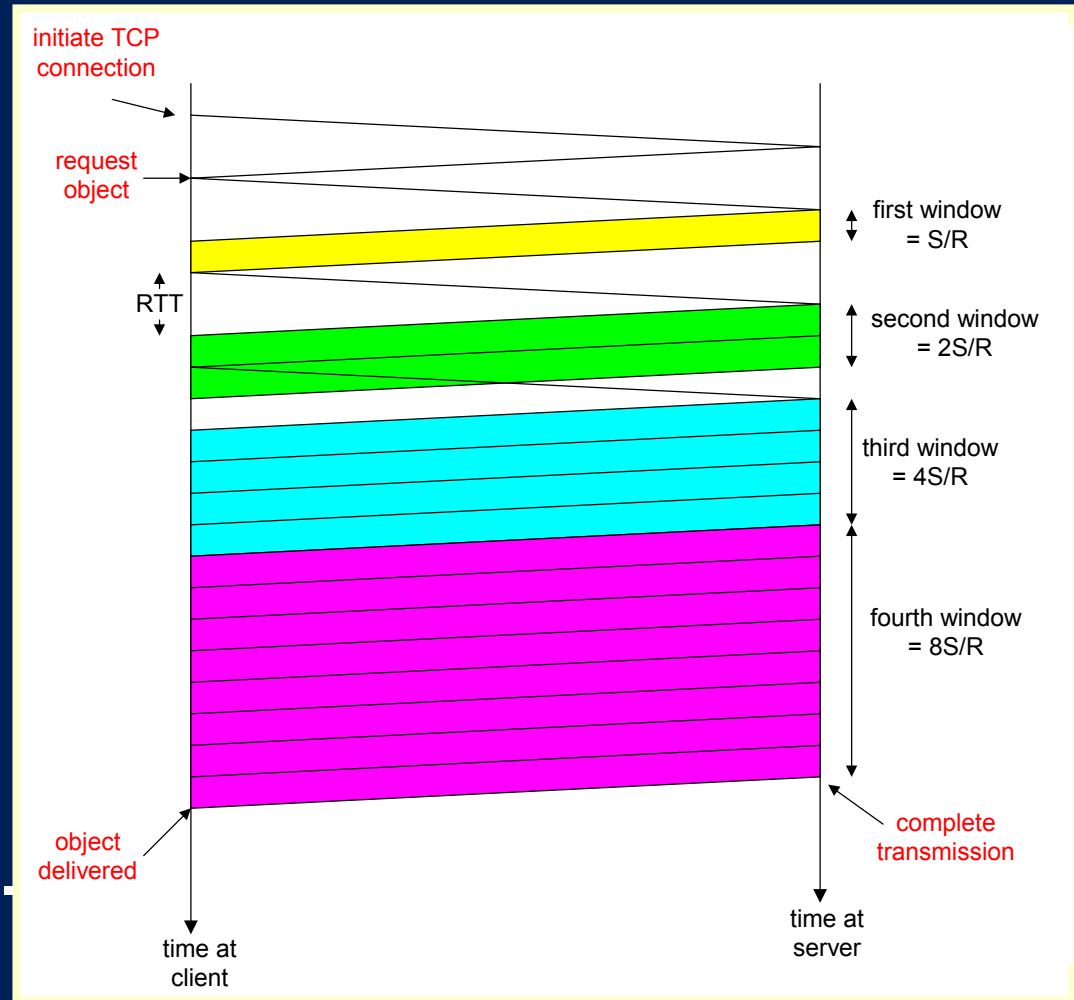
$O/S = 15$  segments

$K = 4$  windows

$Q = 2$

$P = \min\{K-1, Q\} = 2$

Server stalls  $P=2$  times.



# TCP latency modeling: slow start - 4

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

until server receives acknowledgement

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

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

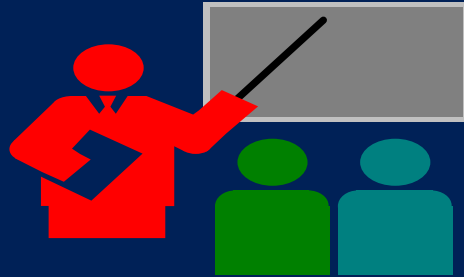
# TCP latency modeling: slow start - 5

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

# Sample Results

<b>R</b>	<b>O/R</b>	<b>p</b>	<b>Minimum Latency: O/R + 2 RTT</b>	<b>Latency with slow start</b>
<b>28 Kbps</b>	<b>28.6 sec</b>	<b>1</b>	<b>28.8 sec</b>	<b>28.9 sec</b>
<b>100 Kbps</b>	<b>8 sec</b>	<b>2</b>	<b>8.2 sec</b>	<b>8.4 sec</b>
<b>1 Mbps</b>	<b>800 msec</b>	<b>5</b>	<b>1 sec</b>	<b>1.5 sec</b>
<b>10 Mbps</b>	<b>80 msec</b>	<b>7</b>	<b>0.28 sec</b>	<b>0.98 sec</b>

# Summary: Chapter 3



- **Principles behind transport layer services:**
  - **multiplexing/demultiplexing**
  - **reliable data transfer**
  - **flow control**
  - **congestion control**
- **Instantiation and implementation in the Internet**
  - **UDP, TCP**