

The background of the slide features a light gray network diagram. It consists of numerous circular nodes of varying sizes connected by thin lines, forming a complex, interconnected web. The nodes are distributed across the entire slide, with some appearing larger and more prominent than others.

# **LECTURE 4 - TRANSPORT LAYER (2)**

# GOALS (1)

- Understand principles behind transport layer services:
  - Flow control
  - Congestion control

# GOALS (2)

- Learn about Internet transport layer protocols:
  - TCP: connection-oriented reliable transport
  - TCP congestion control

# CONNECTION-ORIENTED TRANSPORT: TCP

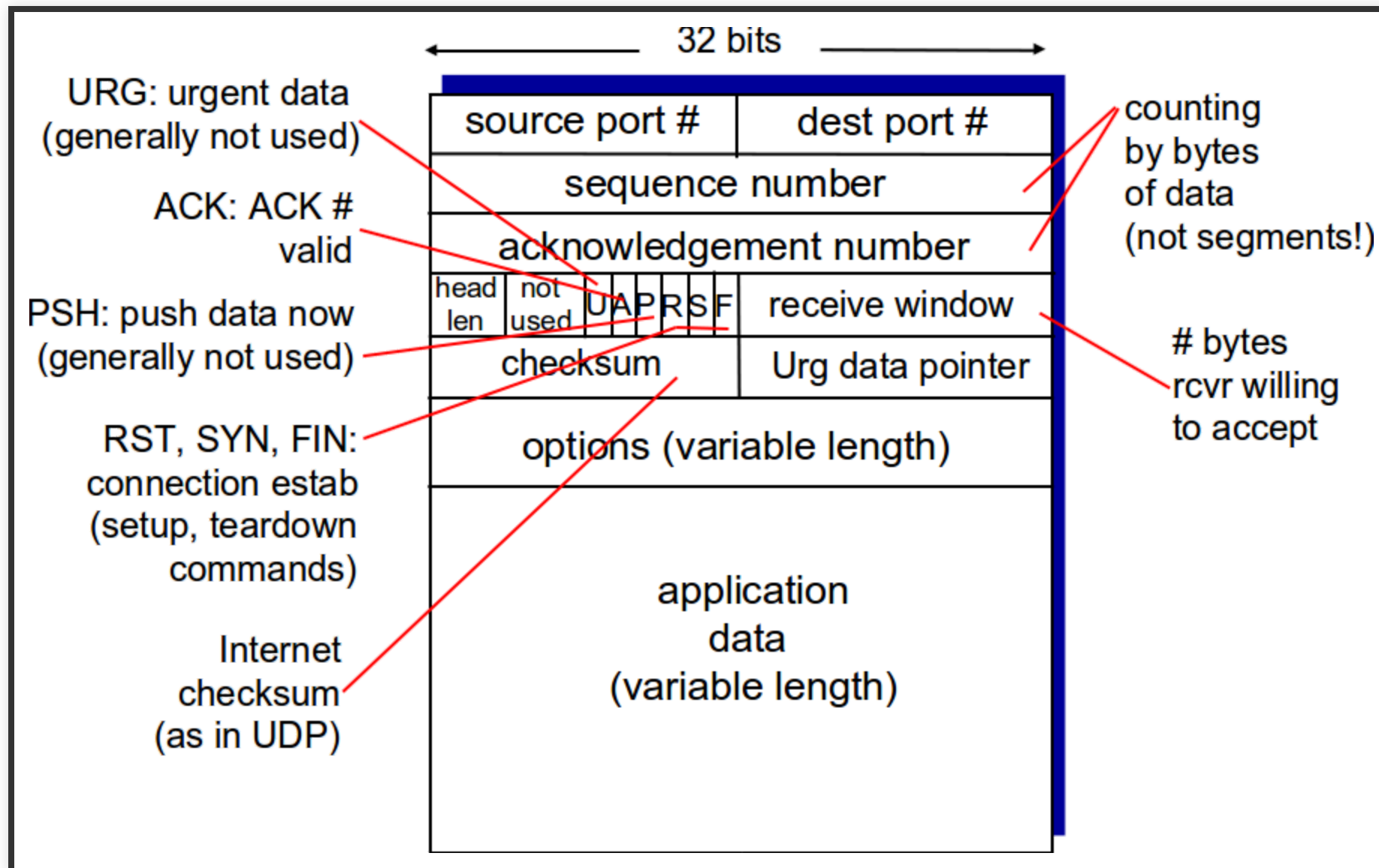
# TCP: OVERVIEW

- RFCs: 793,1122,1323, 2018, 2581
- **Point-to-point:**
  - one sender, one receiver
- **Reliable, in-order byte stream:**
  - no "message boundaries"
- **Pipelined:**
  - TCP congestion and flow control set window size

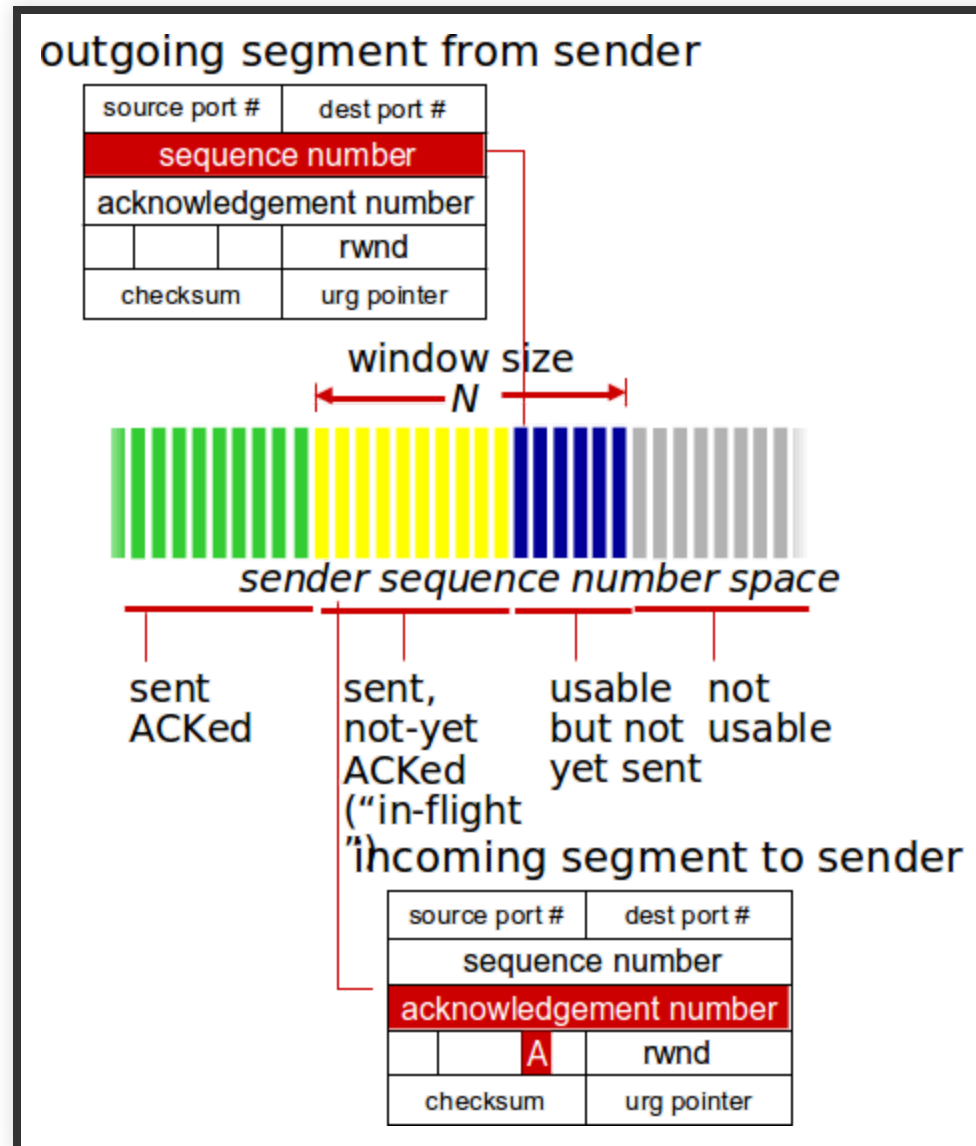
# TCP: OVERVIEW

- **Full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- **Connection-oriented:**
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- **Flow controlled:**
  - sender will not overwhelm receiver

# TCP SEGMENT STRUCTURE



# TCP SEQ. NUMBERS, ACKS





# TCP SEQ. NUMBERS, ACKS

## Sequence numbers:

- Byte stream “number” of first byte in segment’s data

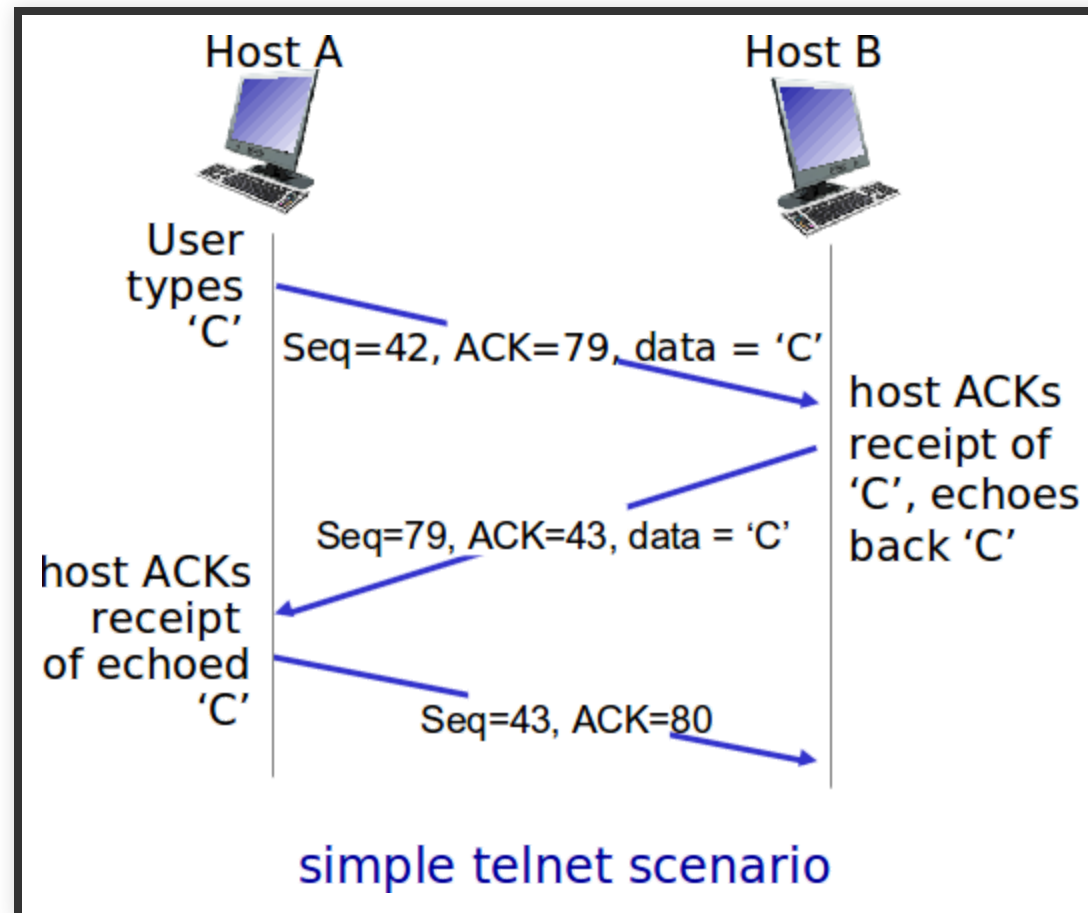
## Acknowledgements:

- 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. NUMBERS, ACKS



# TCP ROUND TRIP TIME, TIMEOUT

❗ Q: how to set TCP timeout value?

- Longer than RTT
  - But RTT varies
- *too short*: premature timeout, unnecessary retransmissions
- *too long*: slow reaction to segment loss

# TCP ROUND TRIP TIME, TIMEOUT

❗ Q:how to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

# TCP ROUND TRIP TIME, TIMEOUT

$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value:  $\alpha = 0.125$

# TCP RTT - TIMEOUT INTERVAL


- EstimatedRTT plus “safety margin”
- large variation in EstimatedRTT → larger safety margin

Estimate SampleRTT deviation from EstimatedRTT:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

Typically,  $\beta = 0.25$

**TimeoutInterval** = **EstimatedRTT** + **4\*DevRTT**

 ↑ ↑

estimated RTT      “safety margin”

# TCP RELIABLE DATA TRANSFER

- TCP creates rdt service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Single retransmission timer
- Retransmissions triggered by:
  - Timeout events
  - Duplicate acks
- Let's initially consider simplified TCP sender:
  - ignore duplicate acks, flow control, congestion control

# TCP SENDER EVENTS:

Data received from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeoutInterval`



# TCP SENDER EVENTS:

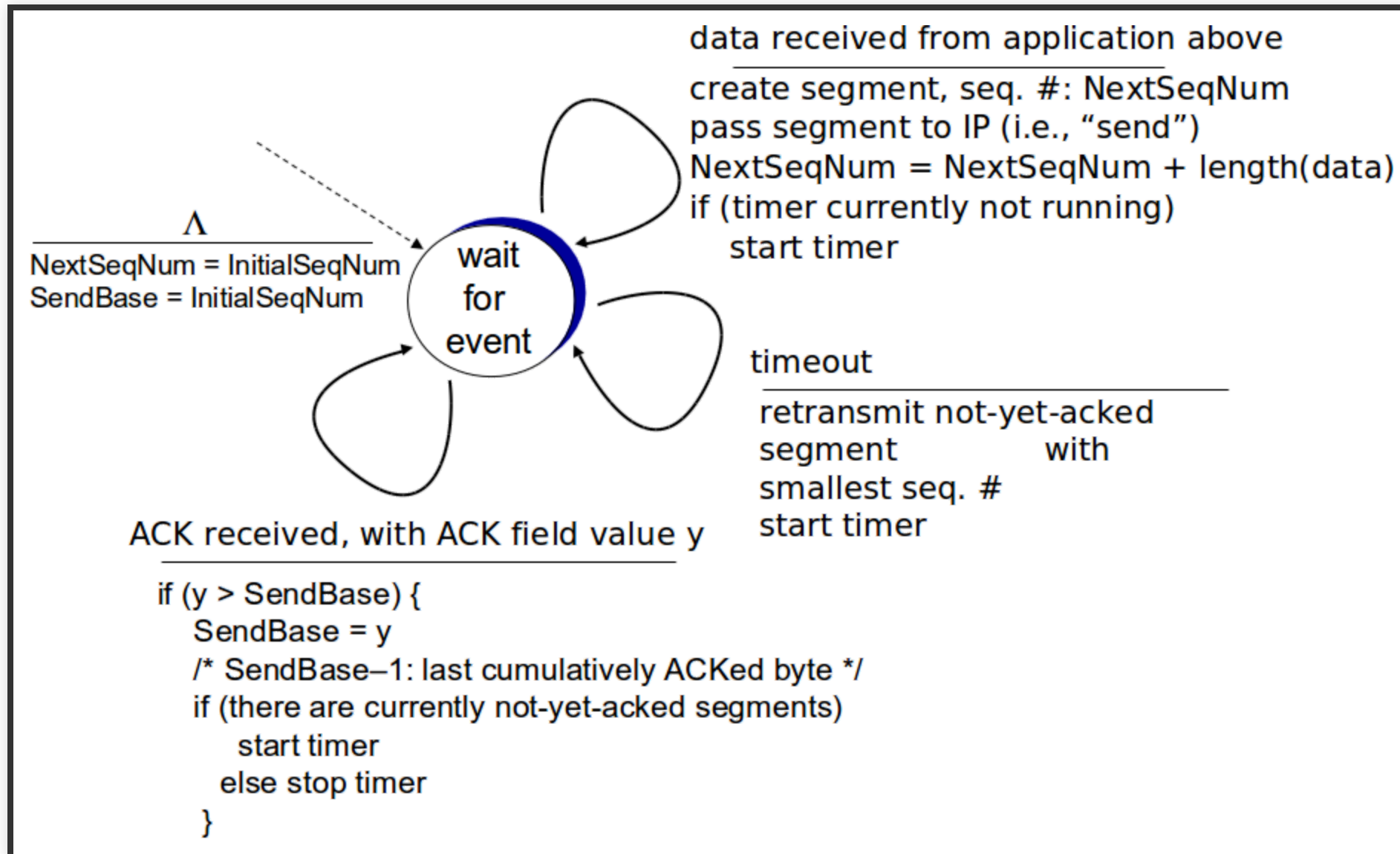
## Timeout:

- Retransmit segment that caused timeout
- Restart timer

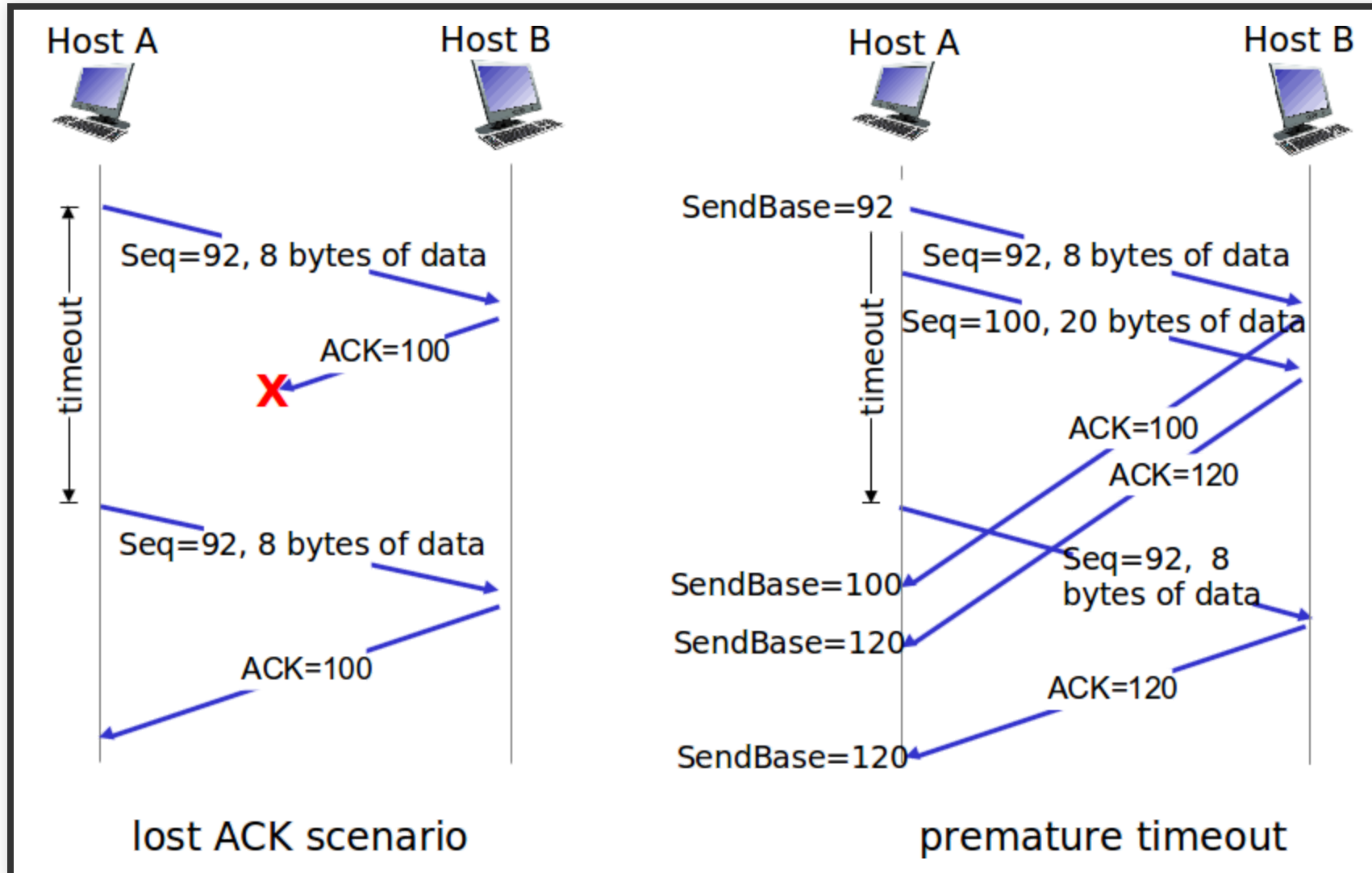
## Ack received:

- If ack acknowledges previously unacked segments
  - Update what is known to be ACKed
  - Start timer if there are still unacked segments

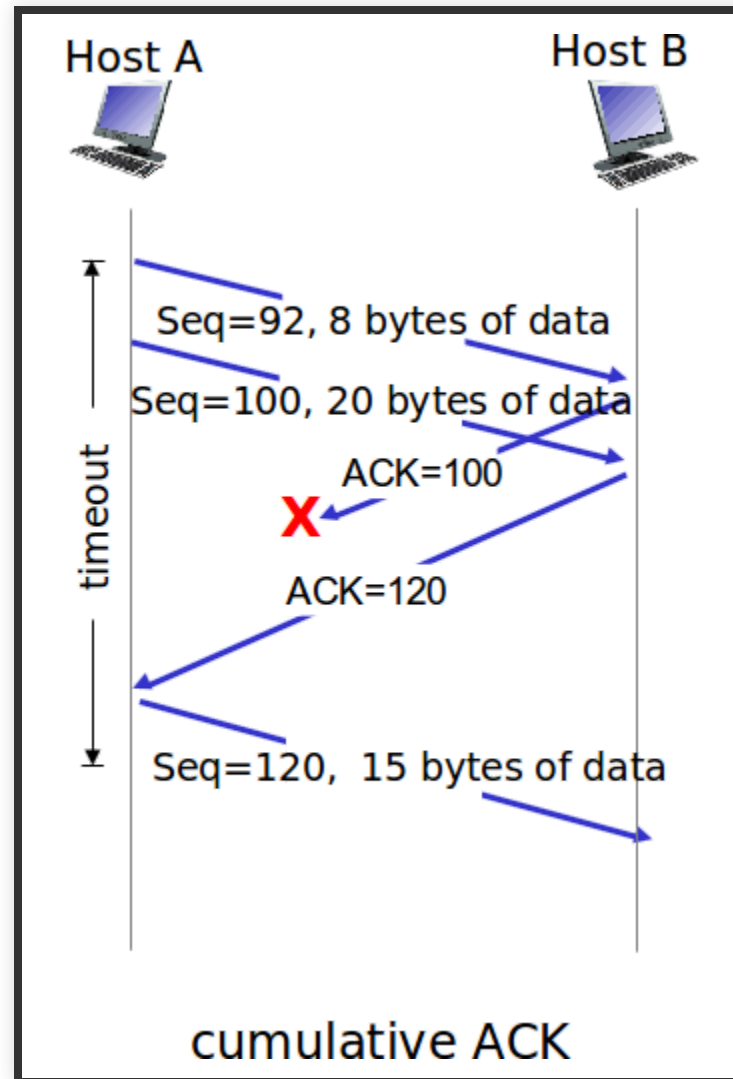
# TCP SENDER (SIMPLIFIED)



# TCP: RETRANSMISSION SCENARIOS



# TCP: RETRANSMISSION SCENARIOS



# TCP ACK GENERATION

RFC 1122, RFC 2581

## Event at receiver

## TCP receiver action

---

arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed

delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK

---

arrival of in-order segment with expected seq #. One other segment has ACK pending

immediately send single cumulative ACK, ACKing both in-order segments

# TCP ACK GENERATION

RFC 1122, RFC 2581

## Event at receiver

## TCP receiver action

---

arrival of out-of-order segment  
higher-than-expected seq. # →  
Gap detected

immediately send **duplicate ACK**, indicating seq. # of next expected byte

---

arrival of segment that partially or completely fills gap

immediate send ACK, provided that segment starts at lower end of gap

# TCP FAST RETRANSMIT

- Time-out period often relatively long:
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

# TCP FAST RETRANSMIT

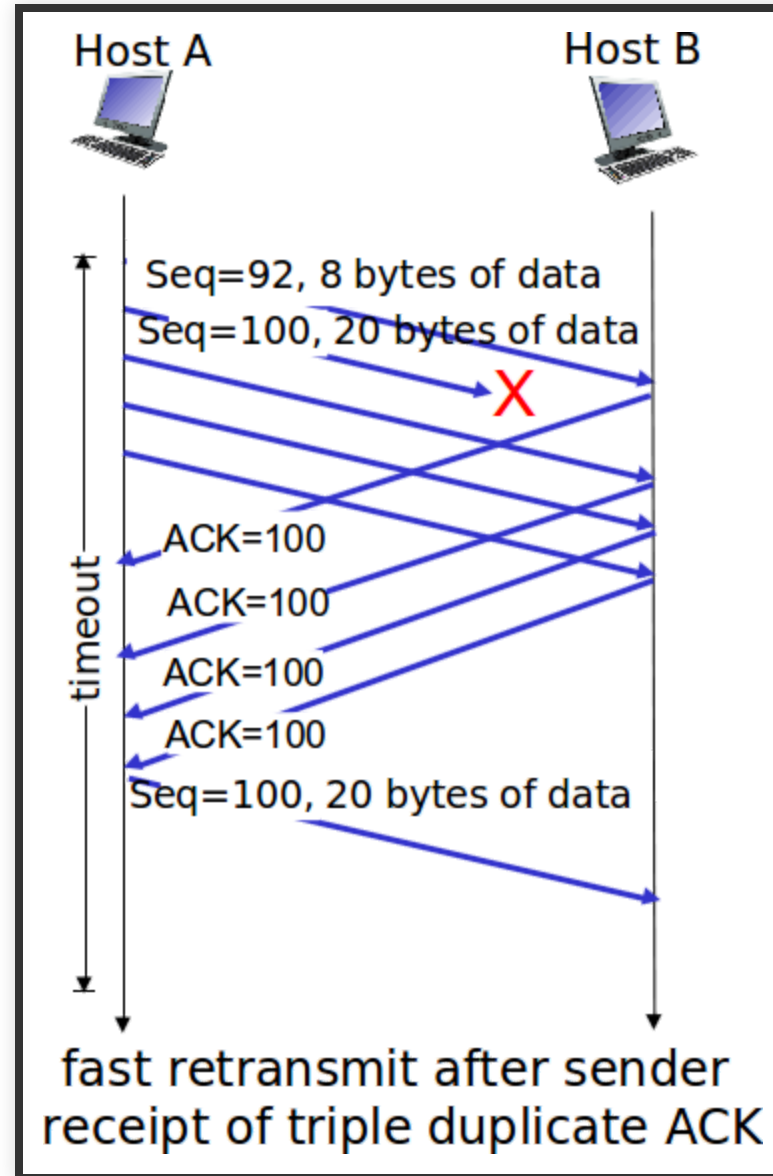
## ! TCP fast retransmit

if sender receives 3 ACKs for same data (*“triple duplicate ACKs”*), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout



# TCP FAST RETRANSMIT



# TCP RDT

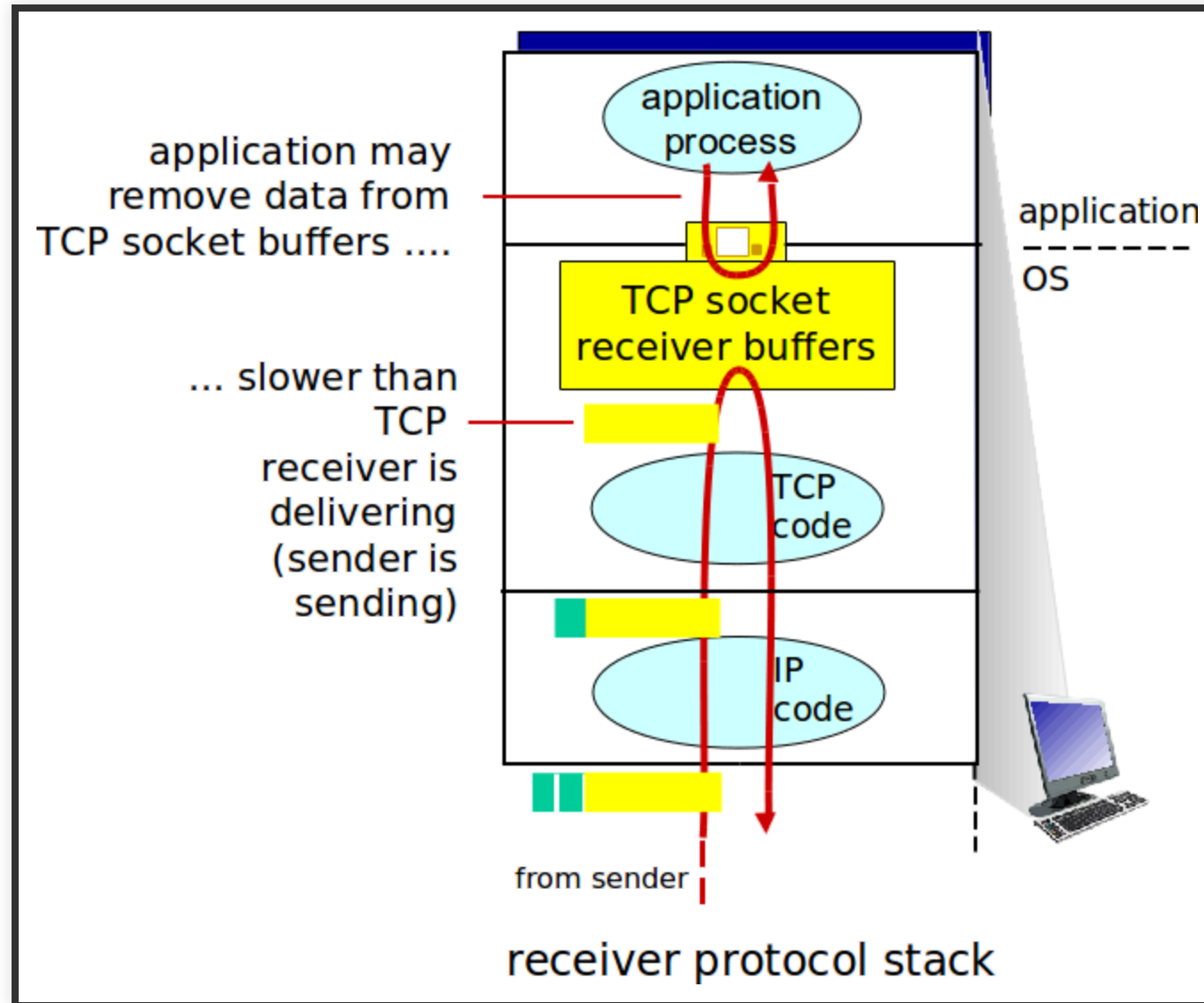
 Is TCP a GBN or SR protocol?

# TCP FLOW CONTROL

## ! Flow control

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

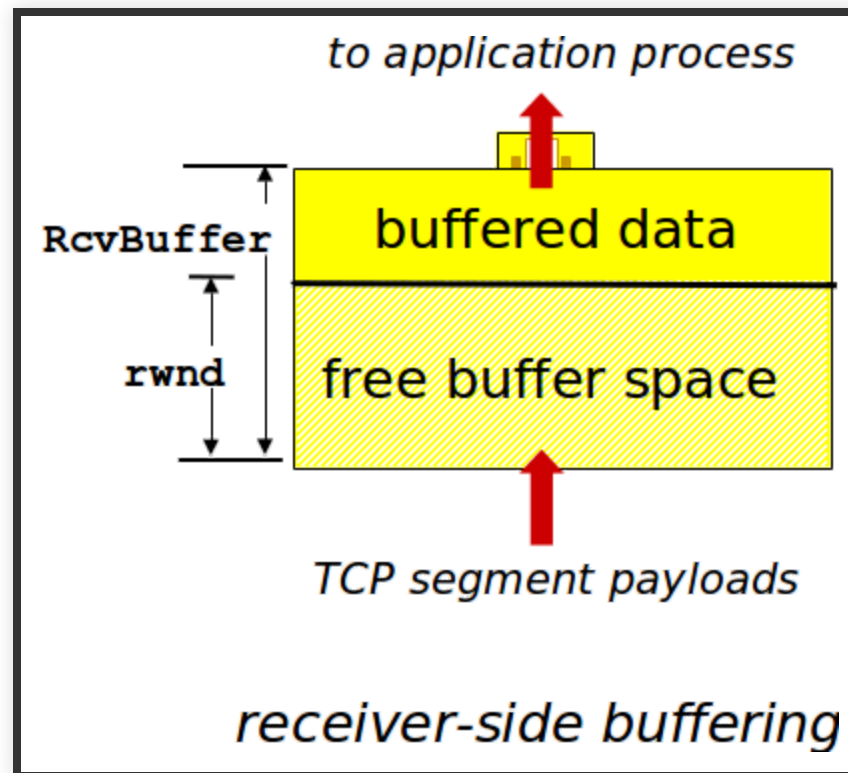
# TCP FLOW CONTROL



# TCP FLOW CONTROL

- Receiver “advertises” free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments
  - `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 receiver’s `rwnd` value
- Guarantees receive buffer will not overflow

# TCP FLOW CONTROL



# SILLY WINDOW SYNDROME

💡 The silly-window syndrome is a term for a scenario in which TCP transfers only small amounts of data at a time.

- TCP/IP packets have a minimum fixed header size of 40 bytes, sending small packets uses the network inefficiently.
- The silly-window syndrome can occur when either by the receiving application consuming data slowly or when the sending application generating data slowly.

# SILLY WINDOW SYNDROME

1. Suppose a TCP connection has a window size of 1000 bytes
2. Receiving application consumes data only 10 bytes at a time
3. At intervals about equal to the RTT



# SILLY WINDOW SYNDROME

- The sender sends bytes 1-1000.
- The receiving application consumes 10 bytes, numbered 1-10.
- The receiving TCP buffers the remaining 990 bytes and sends an ACK reducing the window size to 10
- Upon receipt of the ACK, the sender sends 10 bytes numbered 1001-1010, the most it is permitted.
- In the meantime, the receiving app has consumed bytes 11-20.
- Window size therefore remains at 10 in the next ACK.
- Sender sends bytes 1011-1020 while the application consumes bytes 21-30.

# SILLY WINDOW SYNDROME

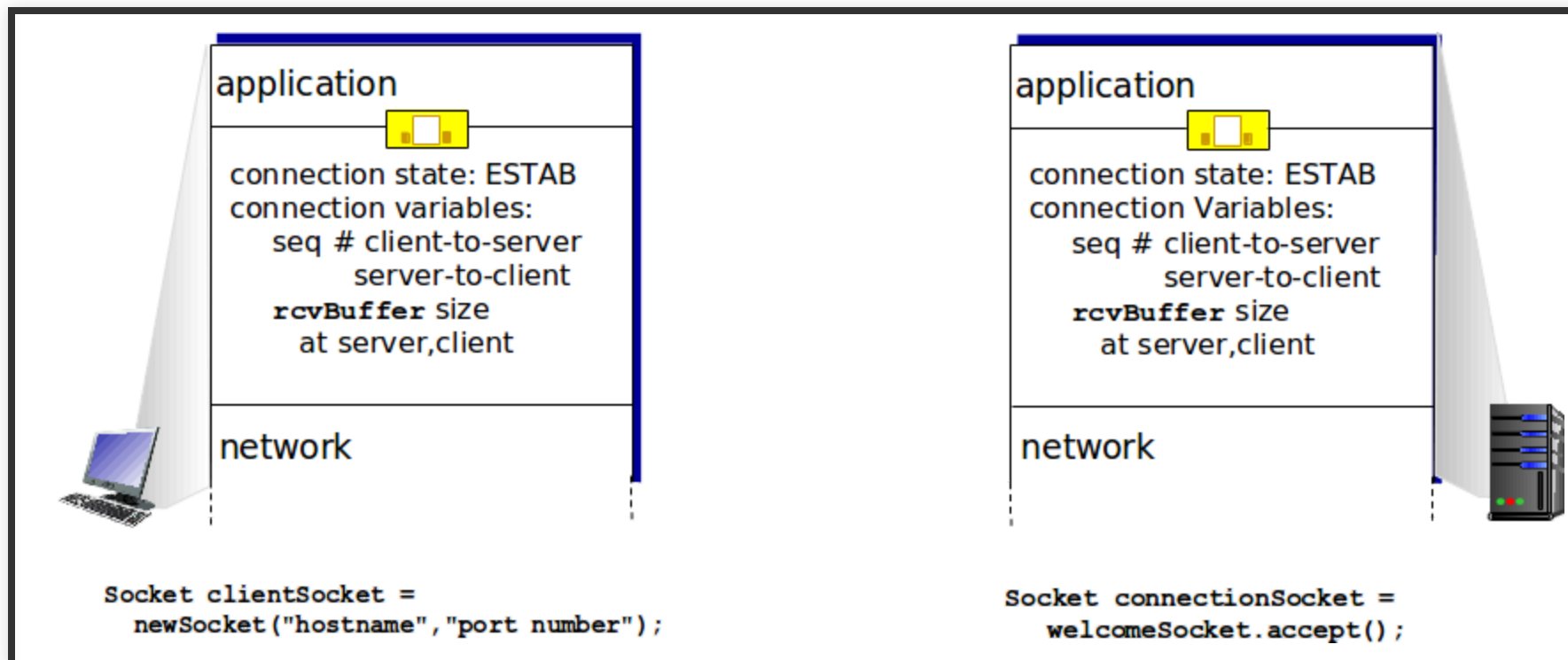
Standard fix: RFC 1122

- The receiver to use its ACKs to keep the window at 0 until it has consumed one full packet's worth
  - or half the window, for small window sizes.
- Then a full packet

# CONNECTION MANAGEMENT

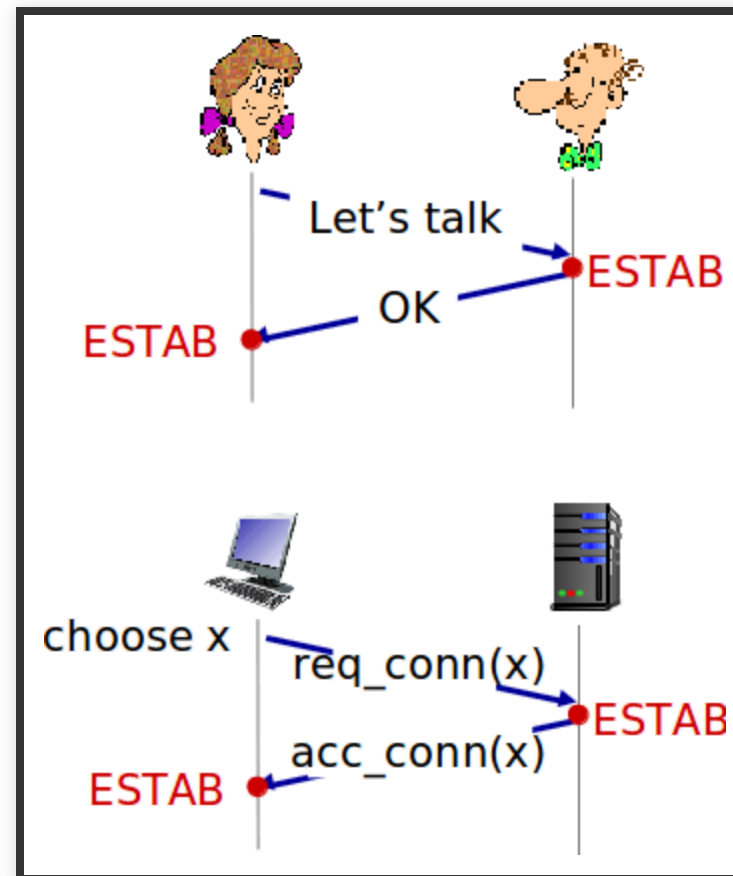
Before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



# AGREEING TO ESTABLISH A CONNECTION

## 2-way-handshake



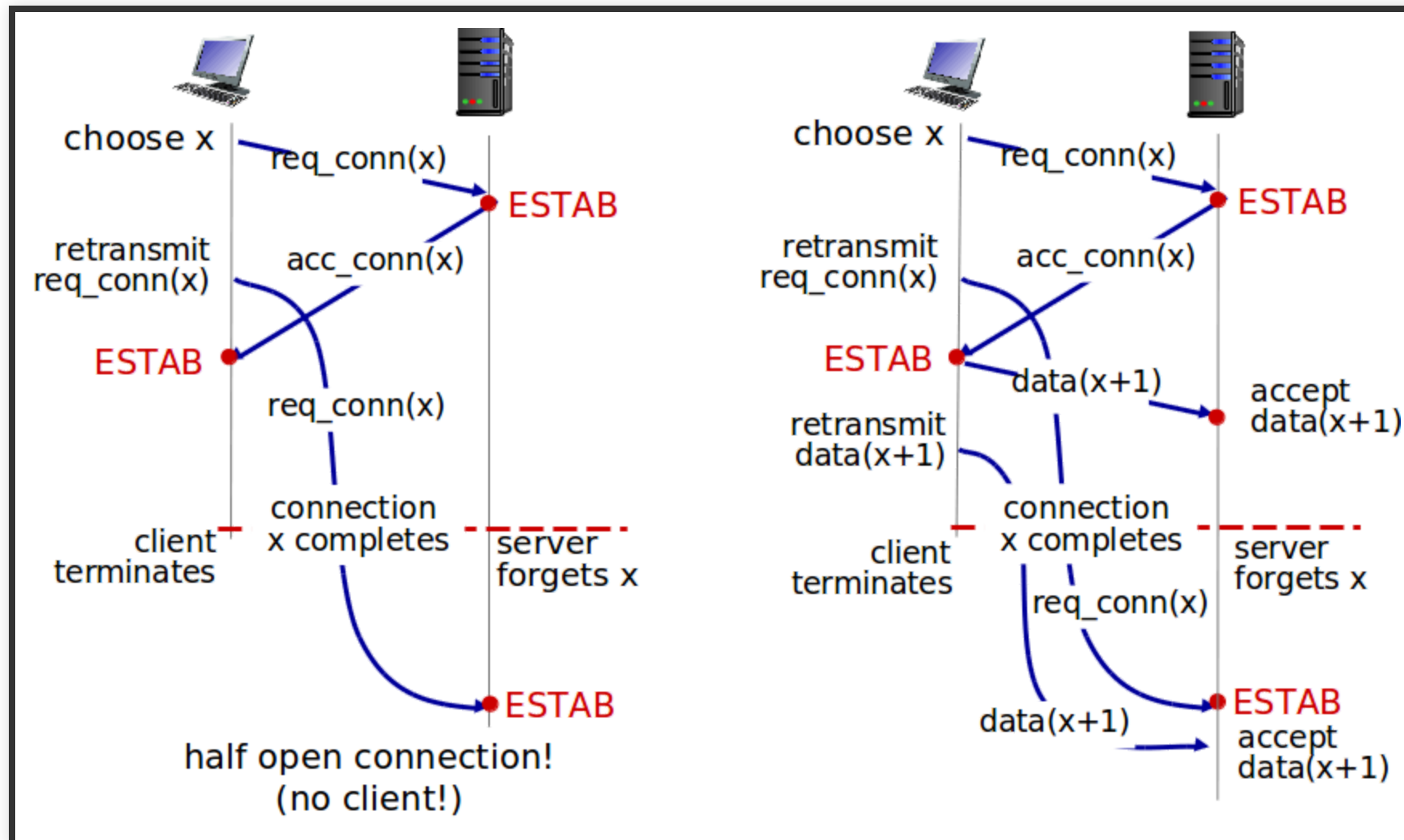
Q: will 2-way handshake always work in network?

# AGREEING TO ESTABLISH A CONNECTION

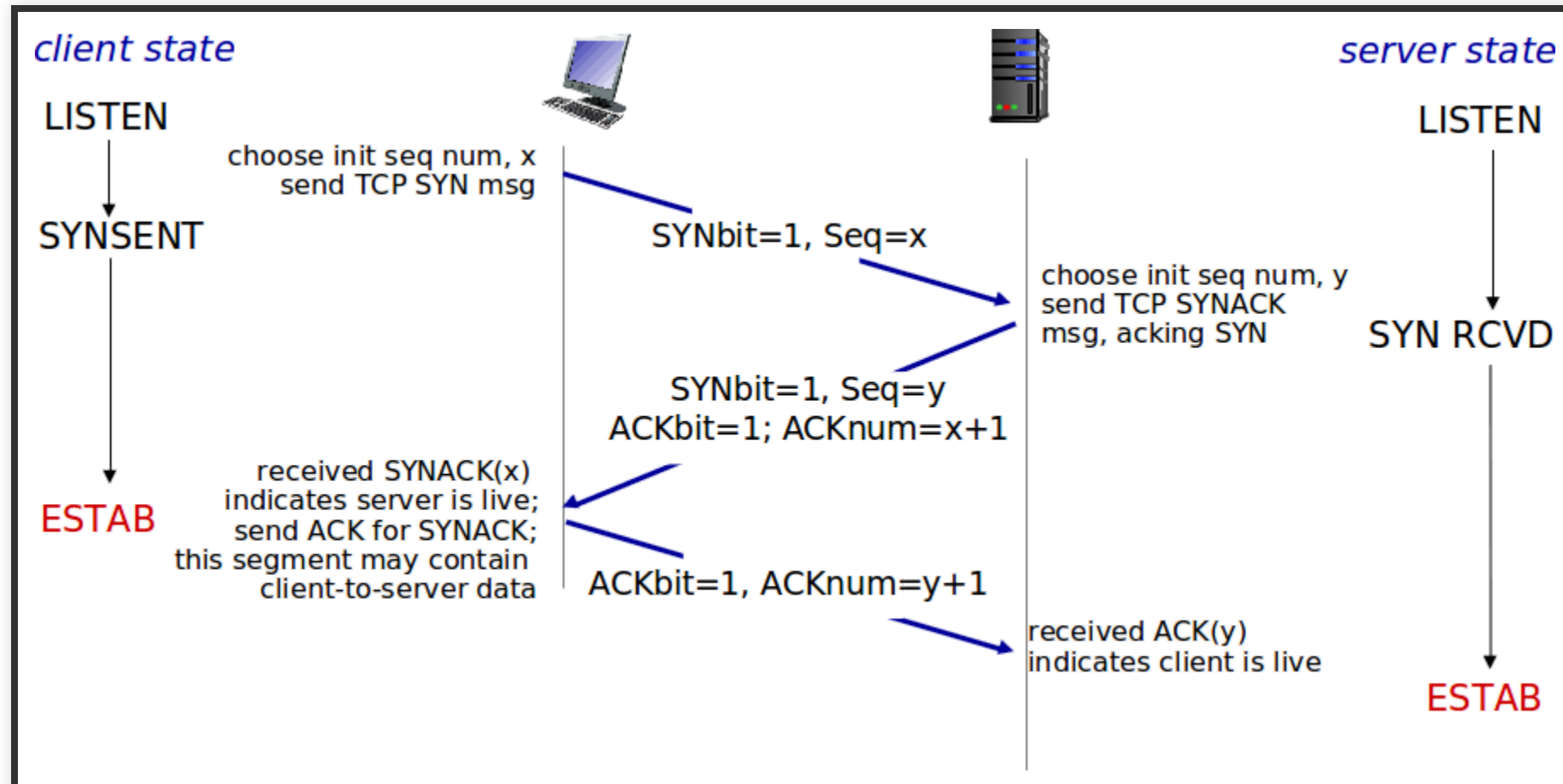
- variable delays
- retransmitted messages (e.g. `req_conn(x)`) due to message loss
- message reordering
- can't "see" other side

# AGREEING TO ESTABLISH A CONNECTION

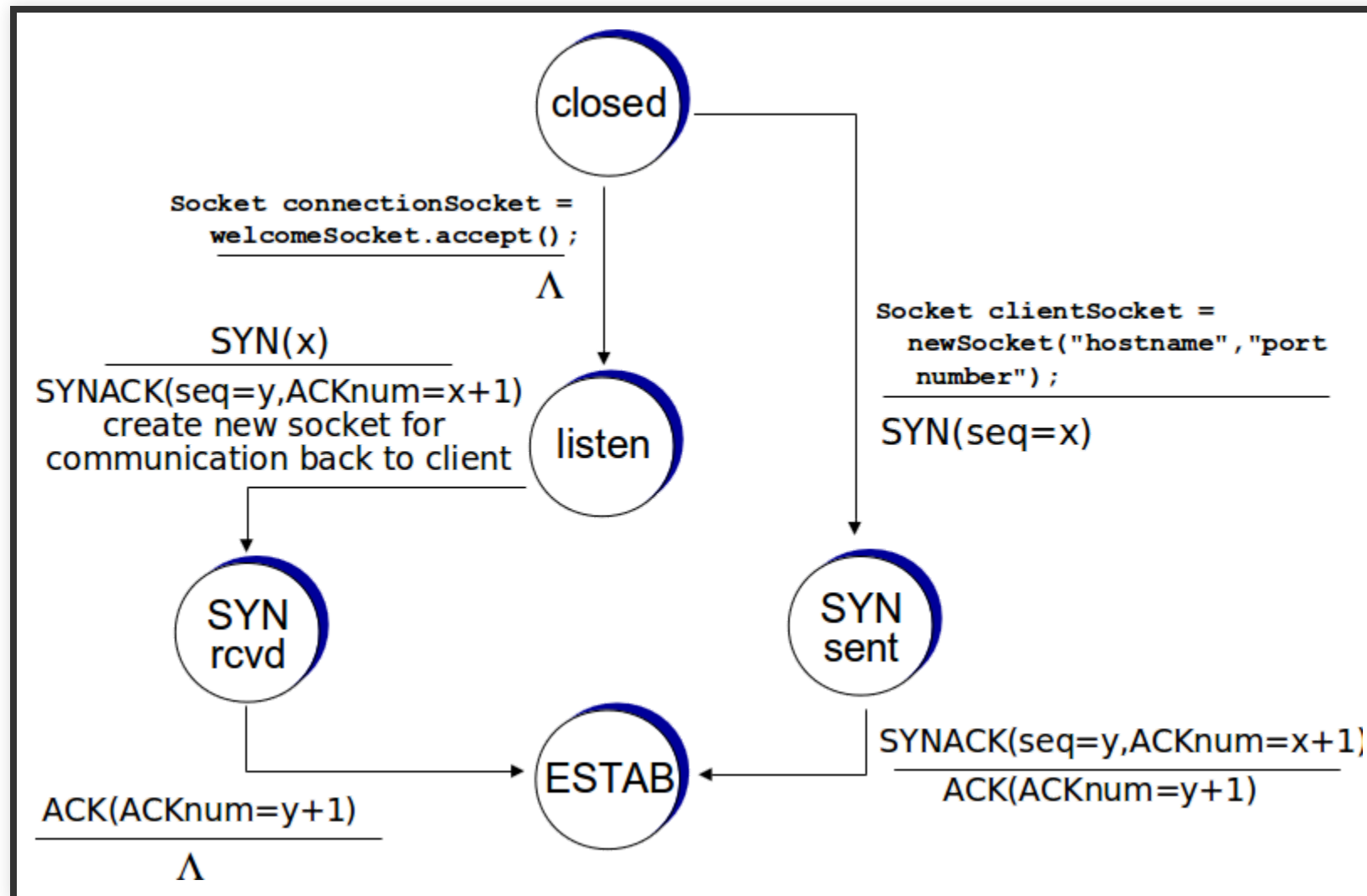
2-way handshake failure scenarios:



# TCP 3-WAY HANDSHAKE



# TCP 3-WAY HANDSHAKE: FSM

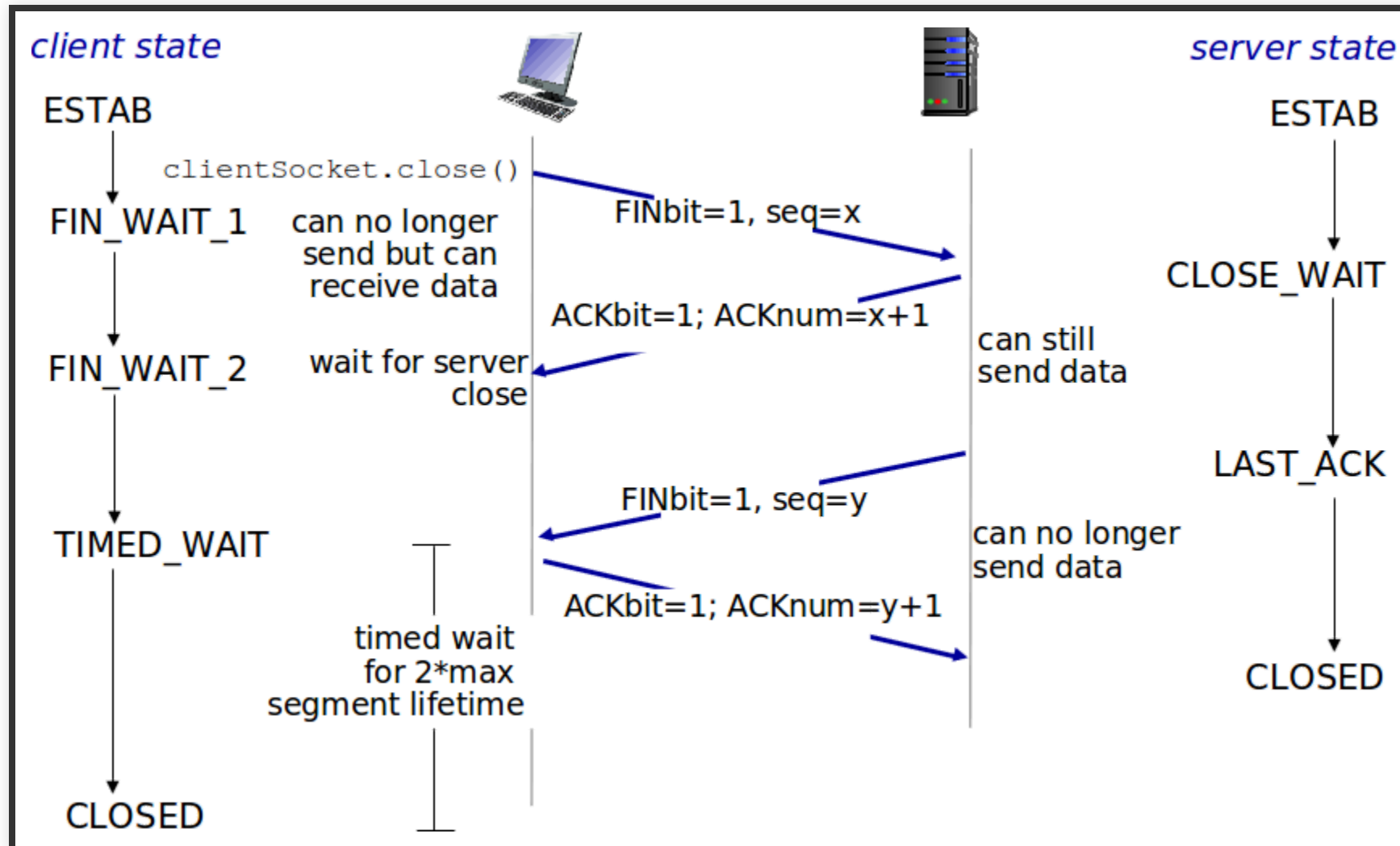




# TCP: CLOSING A CONNECTION

- 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

# TCP: CLOSING A CONNECTION



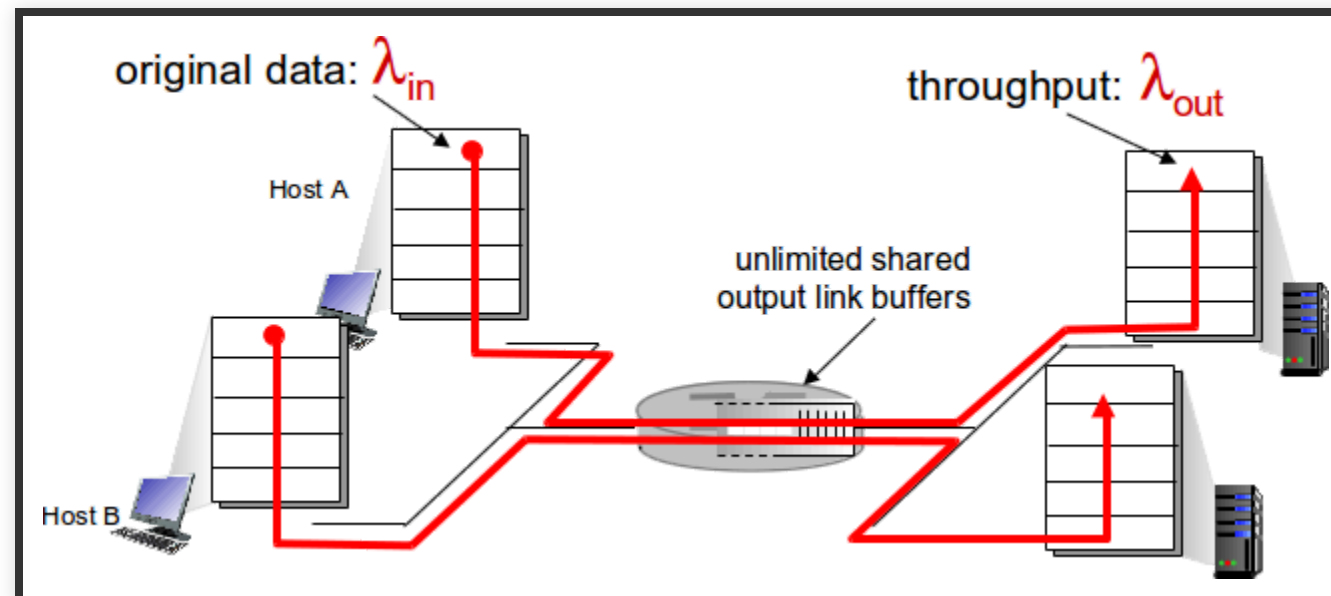
# PRINCIPLES OF CONGESTION CONTROL

# PRINCIPLES OF CONGESTION CONTROL

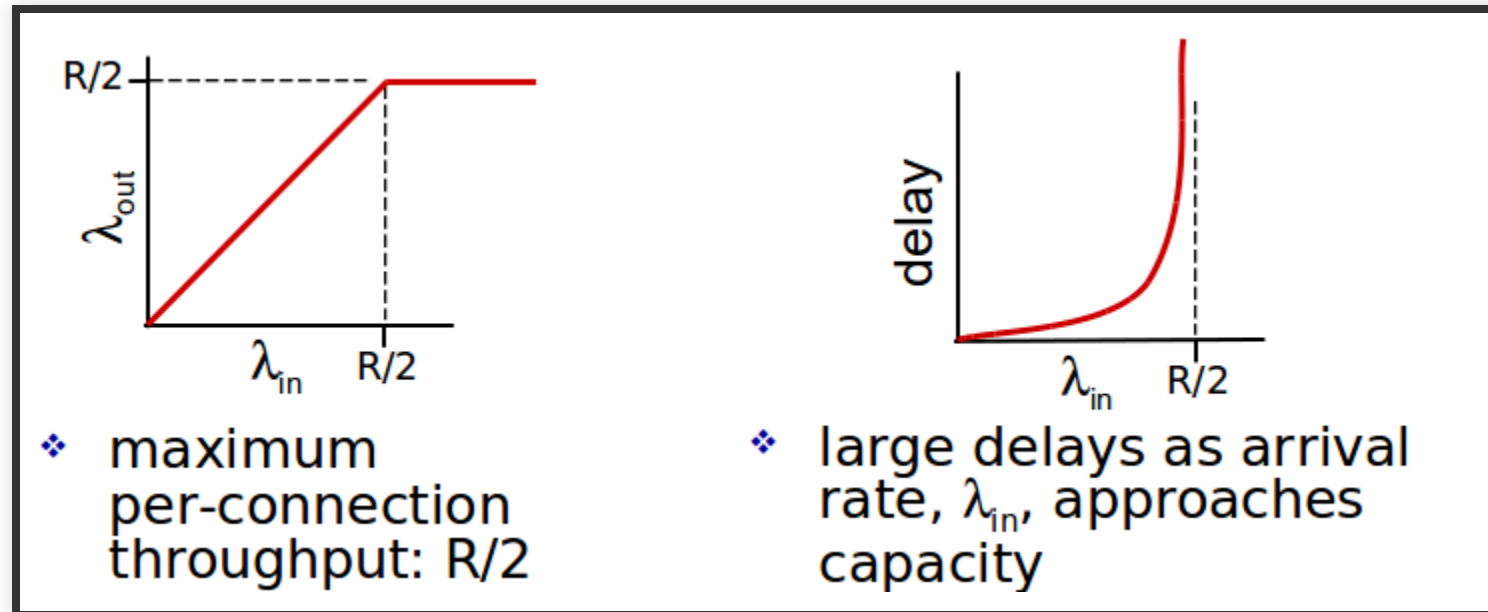
- ❗ congestion: informally: “too many sources sending too much data too fast for network to handle”
  - different from flow control!
  - manifestations:
    - lost packets (buffer overflow at routers)
    - long delays (queueing in router buffers)
  - a top-10 problem!

# CAUSES/COSTS OF CONGESTION: SCENARIO 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity:  $R$
- no retransmission

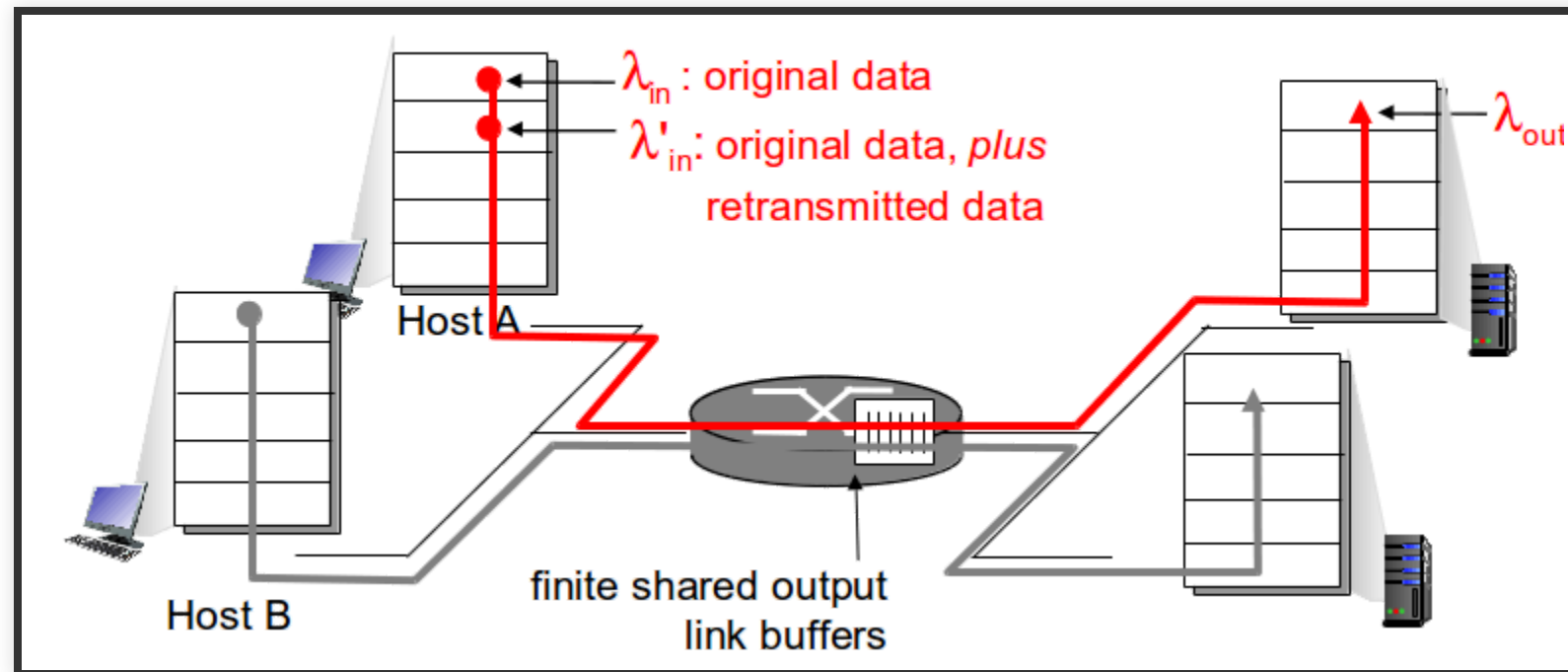


# CAUSES/COSTS OF CONGESTION: SCENARIO 1



# CAUSES/COSTS OF CONGESTION: SCENARIO 2

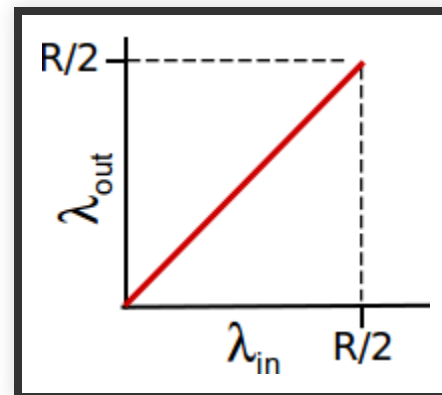
- one router, **finite buffers**
- sender retransmission of timed-out packet
  - Application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes retransmissions:  $\lambda_{in} \geq \lambda_{out}$



# CAUSES/COSTS OF CONGESTION: SCENARIO 2

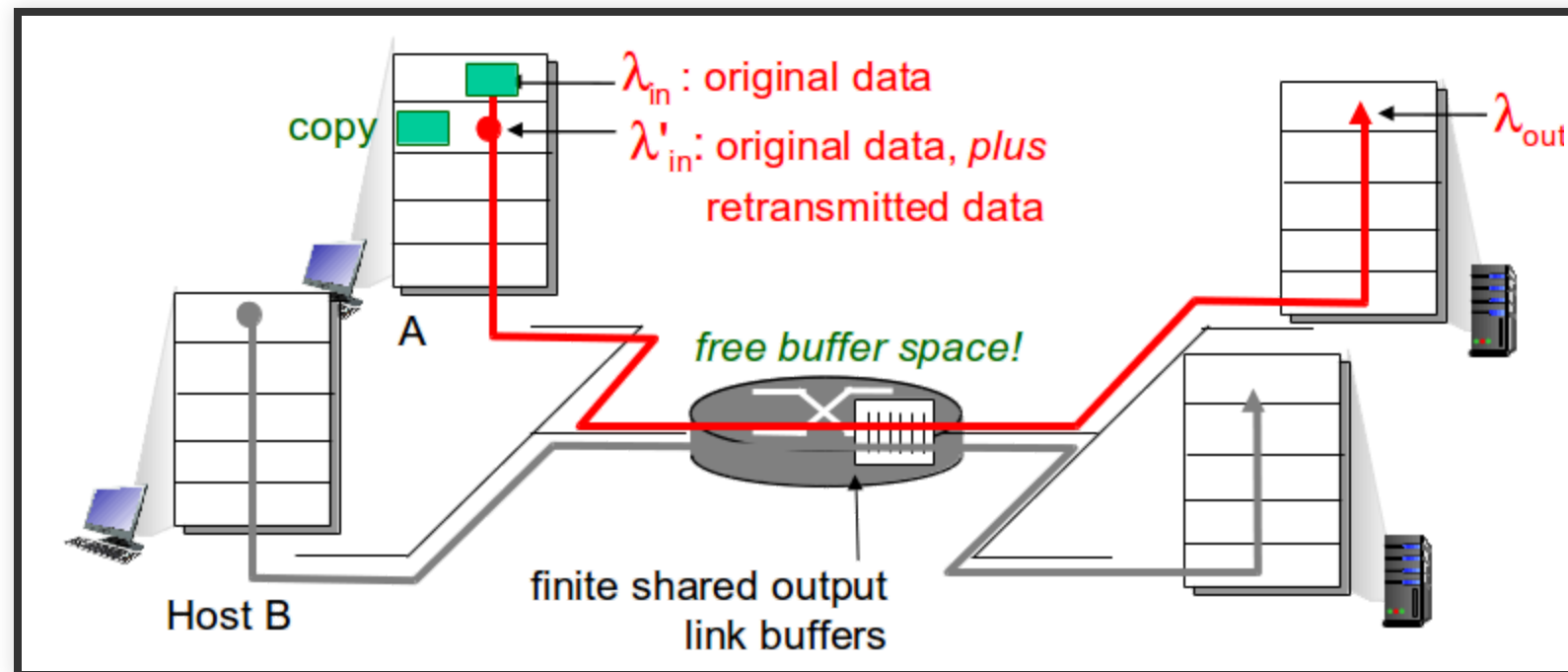
idealization: perfect knowledge

- sender sends only when router buffers available





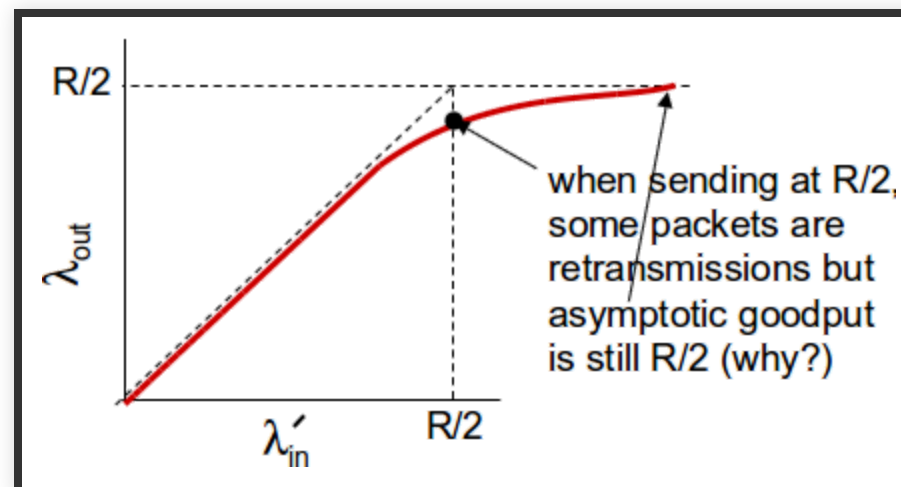
# CAUSES/COSTS OF CONGESTION: SCENARIO 2



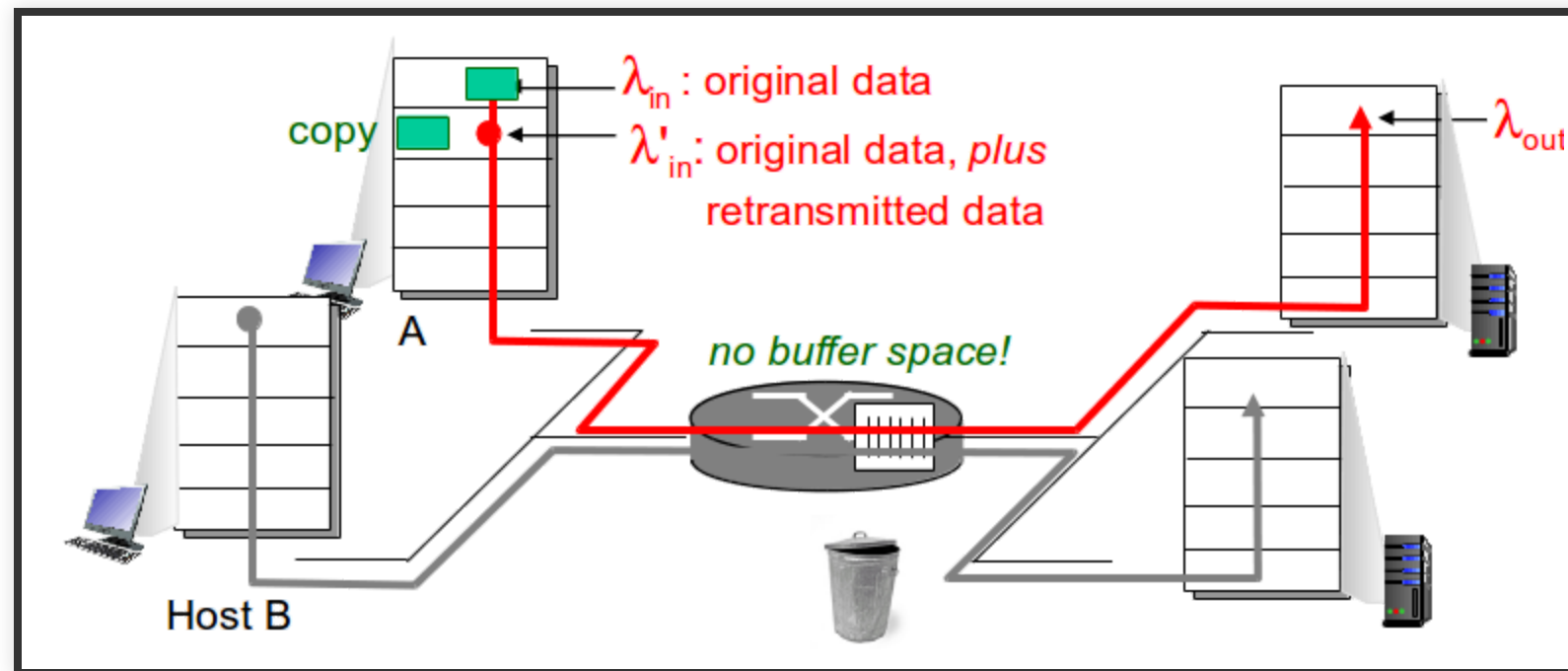
# CAUSES/COSTS OF CONGESTION: SCENARIO 2

Idealization: *known loss* packets can be lost, dropped at router due to full buffers

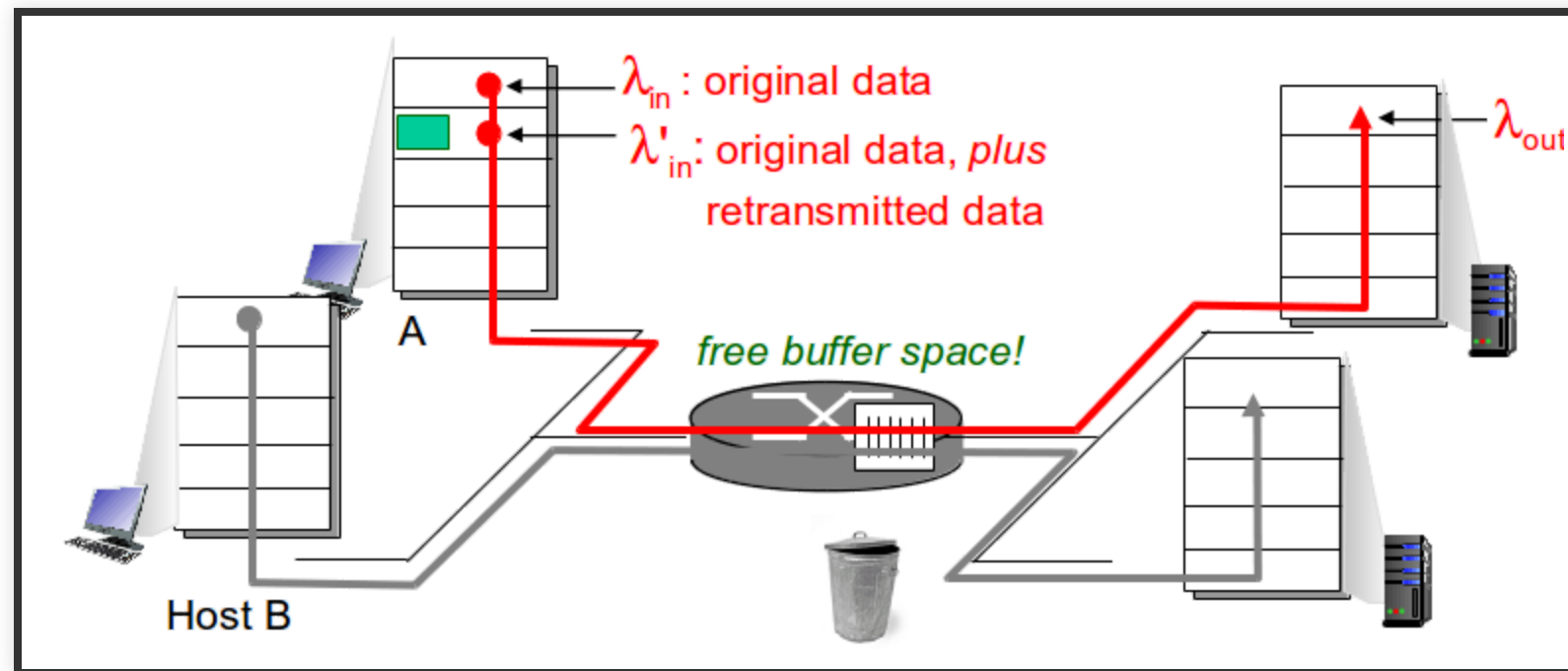
- sender only resends if packet *known* to be lost



# CAUSES/COSTS OF CONGESTION: SCENARIO 2



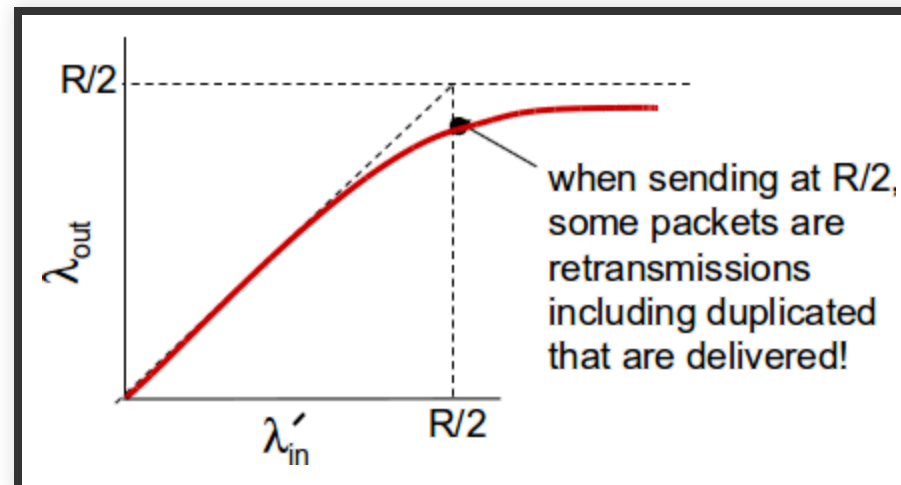
# CAUSES/COSTS OF CONGESTION: SCENARIO 2



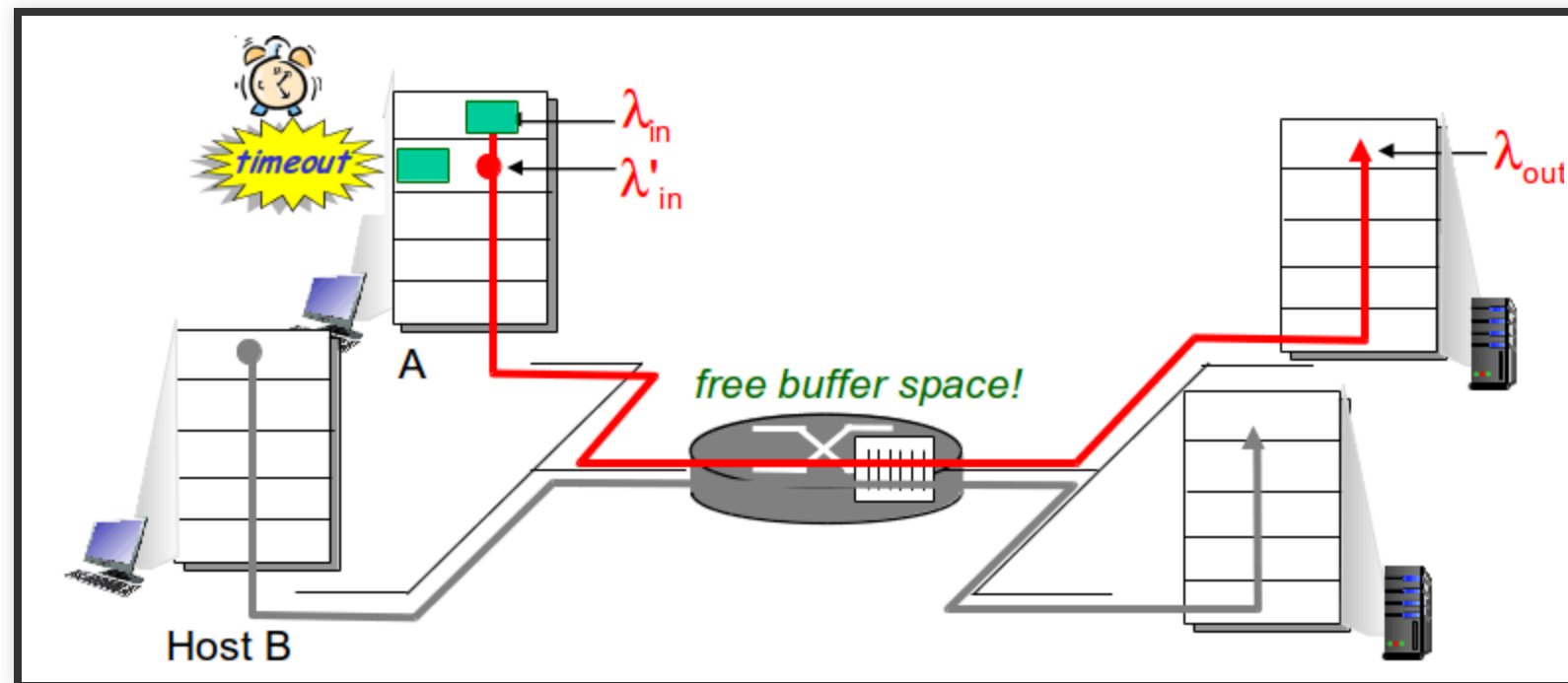
# CAUSES/COSTS OF CONGESTION: SCENARIO 2

Realistic: *duplicates*

- Packets can be lost, dropped at router due to full buffers
- Sender times out prematurely, sending **two** copies, both of which are delivered



# CAUSES/COSTS OF CONGESTION: SCENARIO 2



# CAUSES/COSTS OF CONGESTION: SCENARIO 2

“costs” of congestion:

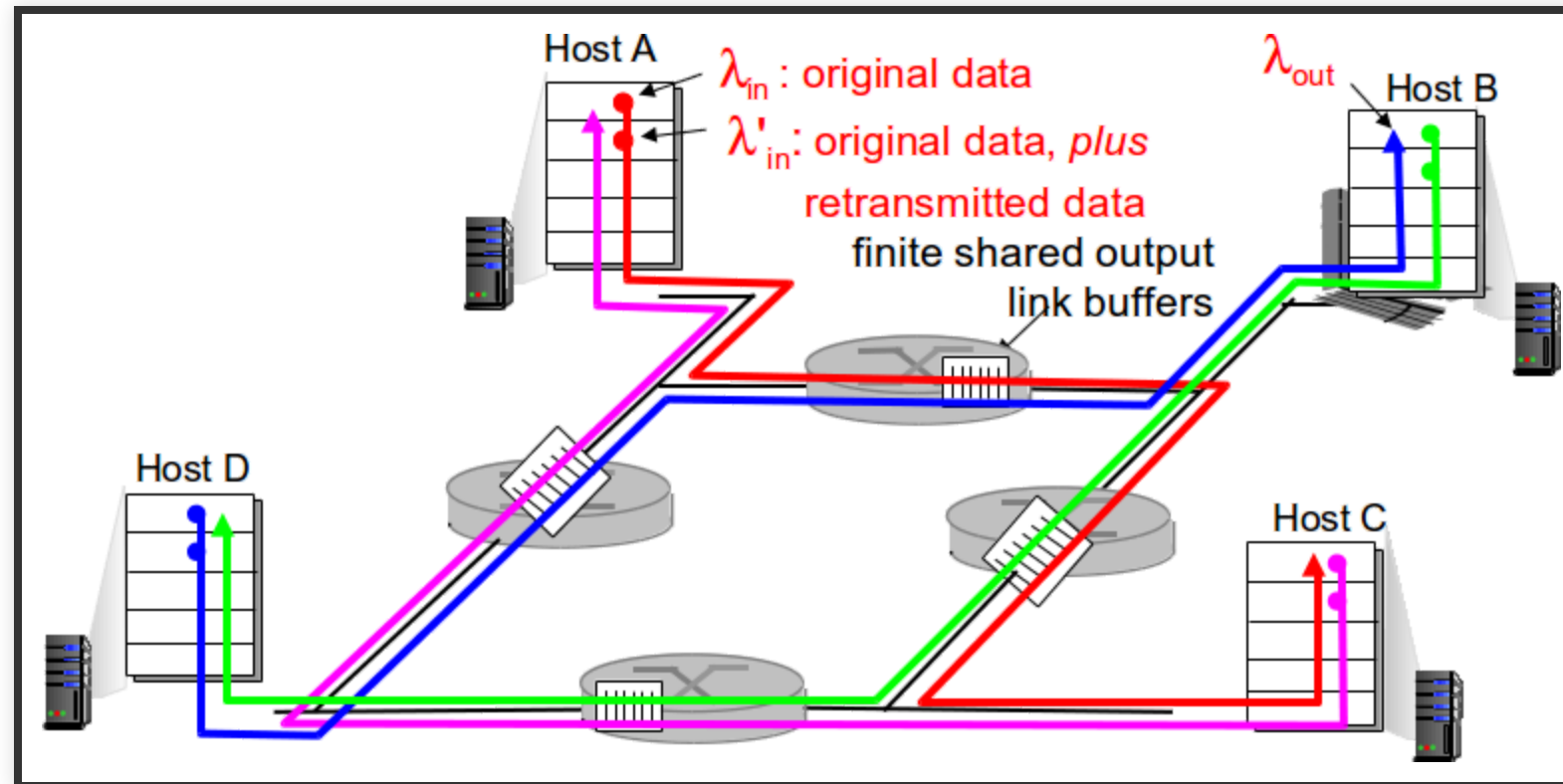
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

# CAUSES/COSTS OF CONGESTION: SCENARIO 3

- four senders
- multihop paths
- timeout/retransmit



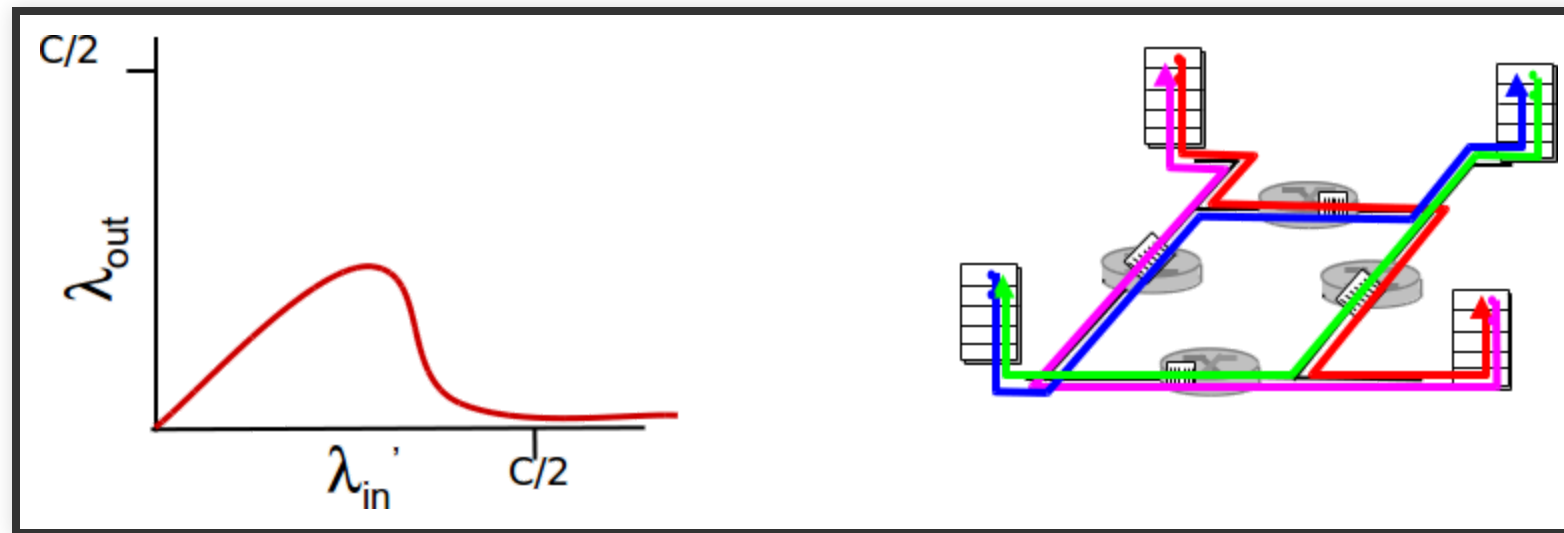
# CAUSES/COSTS OF CONGESTION: SCENARIO 3



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 used for that packet was wasted!

# APPROACHES TOWARDS CONGESTION CONTROL

Two broad approaches towards congestion control:

- End-end congestion control
- Network-assisted 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 for sender to send at

# TCP CONGESTION CONTROL

# TCP CONGESTION CONTROL

3 components

1. Slow start
2. Congestion avoidance
3. Fast recovery

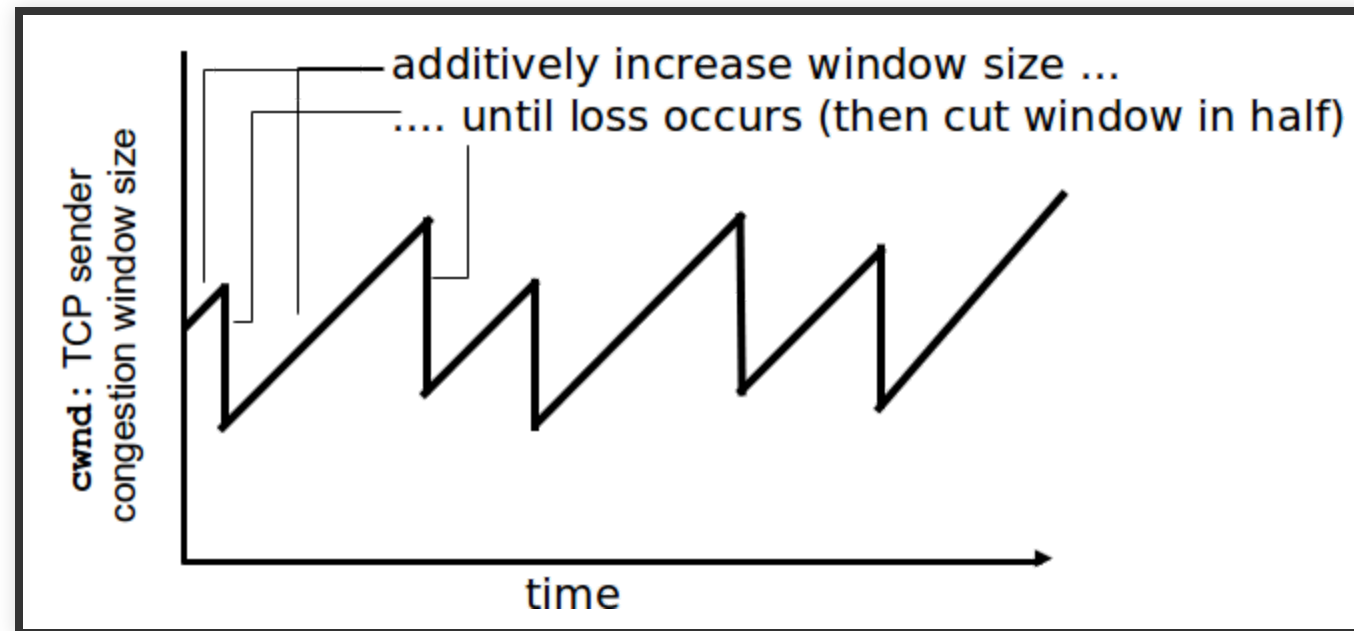
# TCP CONGESTION CONTROL

## Additive Increase Multiplicative Decrease

- **approach:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase:** increase cwnd by 1 MSS every RTT until loss detected
  - **multiplicative decrease:** cut cwnd in half after loss

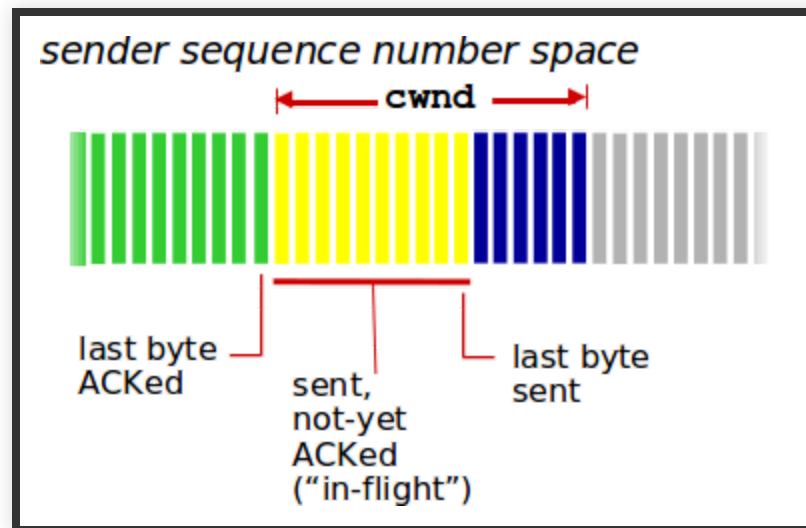


# TCP CONGESTION CONTROL



AIMD saw tooth behavior: probing for bandwidth

# TCP CONGESTION CONTROL: DETAILS



- sender limits transmission:  
 $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$
- cwnd is dynamic, function of perceived network congestion

# TCP CONGESTION CONTROL: DETAILS

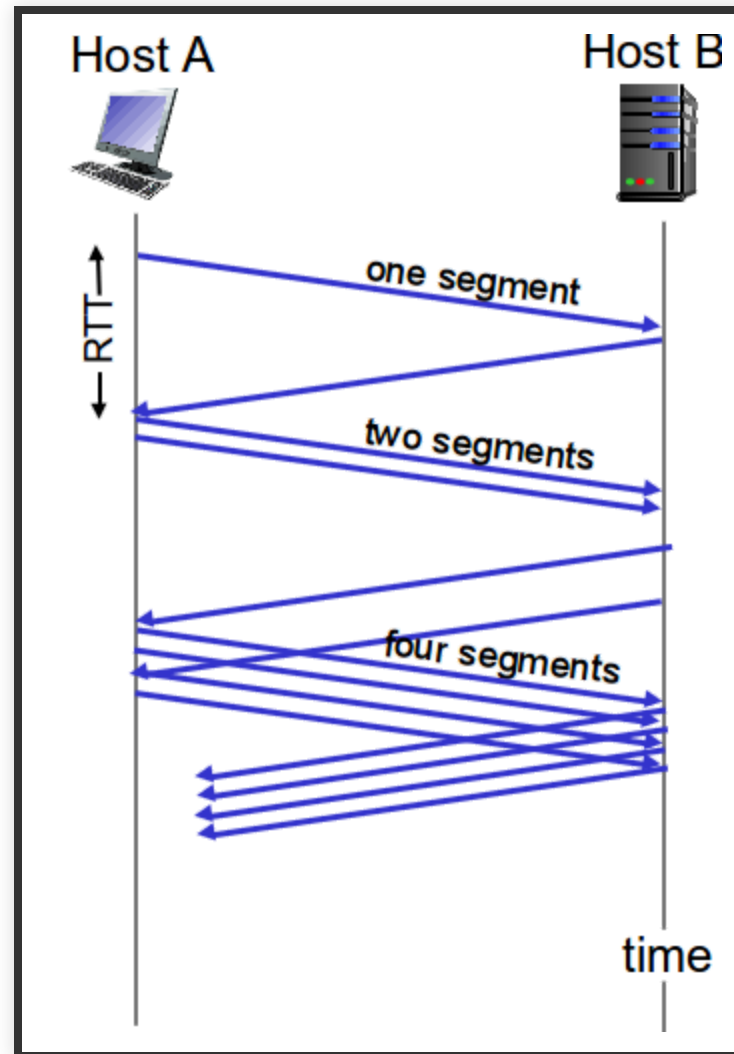
❗ TCP sending rate:

roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes rate  $\sim$  cwnd/RTT bytes/sec

# 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 SLOW START



# 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
- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)

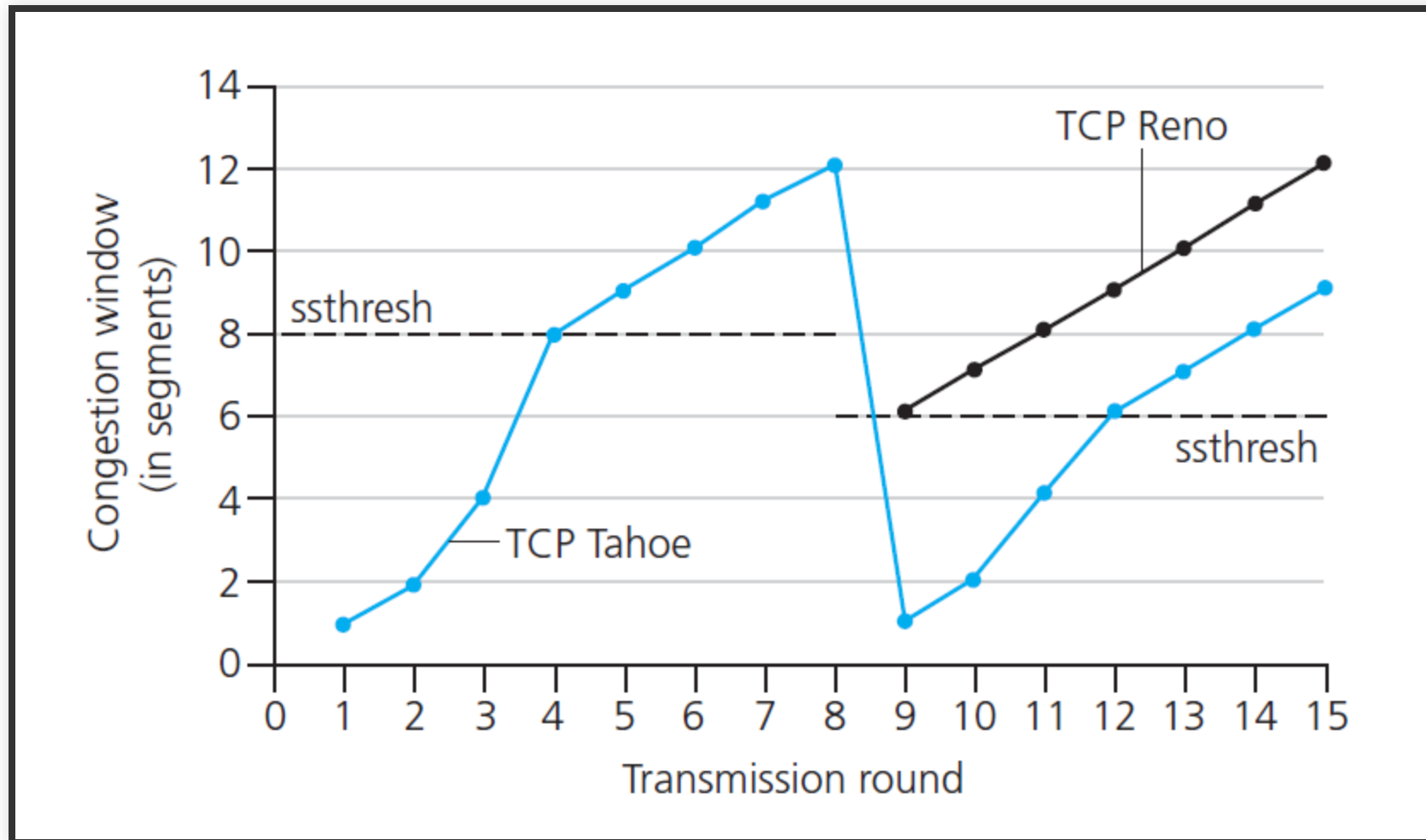
# TCP: SWITCHING FROM SLOW START TO CA

- 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: SWITCHING FROM SLOW START TO CA

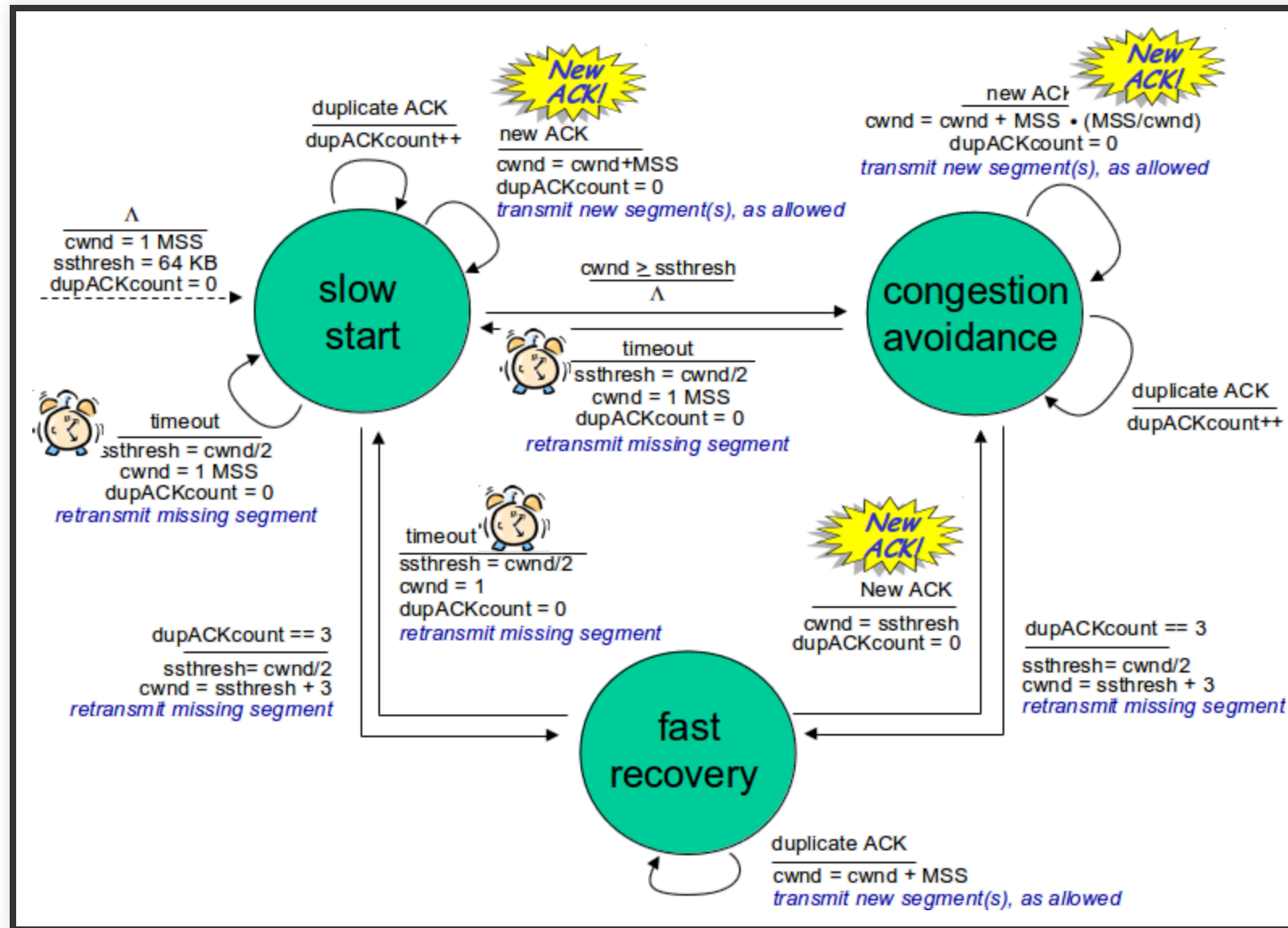




# FAST RECOVERY

- cwnd increased by 1 MSS for every duplicate ack.
  1. ack received  $\rightarrow$  cwnd = ssthresh and goto CA
  2. if timeout: goto slow start, cwnd = 1, ssthresh = cwnd/2

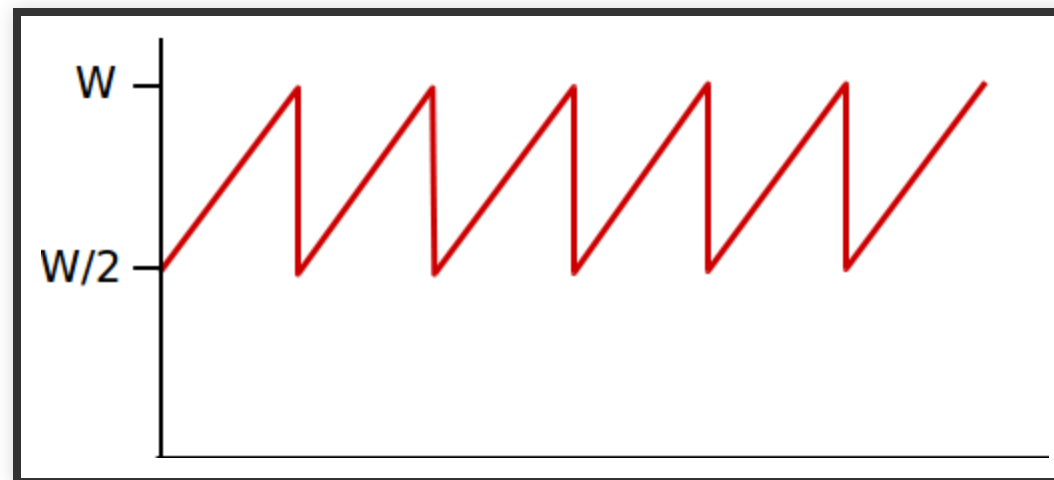
# SUMMARY: TCP CONGESTION CONTROL



# TCP THROUGHPUT

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

avg TCP throughput =  $(3 W) / (4 RTT)$  bytes/sec



# TCP VERSIONS

Multiple TCP Versions exists

```
cat /proc/sys/net/ipv4/tcp_available_congestion_control  
cd /lib/modules/$(uname -r)/kernel/net/ipv4
```

# TCP VERSIONS

In the C language, we can select the linux congestion control mechanism, after socket creation but before connection, by including the `setsockopt ( )`

```
#include <netinet/in.h>
#include <netinet/tcp.h>
...
char * cong_algorithm = "vegas";
int slen = strlen( cong_algorithm ) + 1;
int rc = setsockopt( sock, IPPROTO_TCP, TCP_CONGESTION, cong_algorithm, slen);
if (rc < 0) { /* error */ }
```

# TCP FUTURES: 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]:  
TCP throughput =  $(1.22 \text{ MSS}) / (\text{RTT} \sqrt{L})$  → to achieve 10 Gbps throughput, need a loss rate of  $L = 2 \cdot 10^{-10}$   
– a very small loss rate!
- new versions of TCP for high-speed

# TCP CUBIC

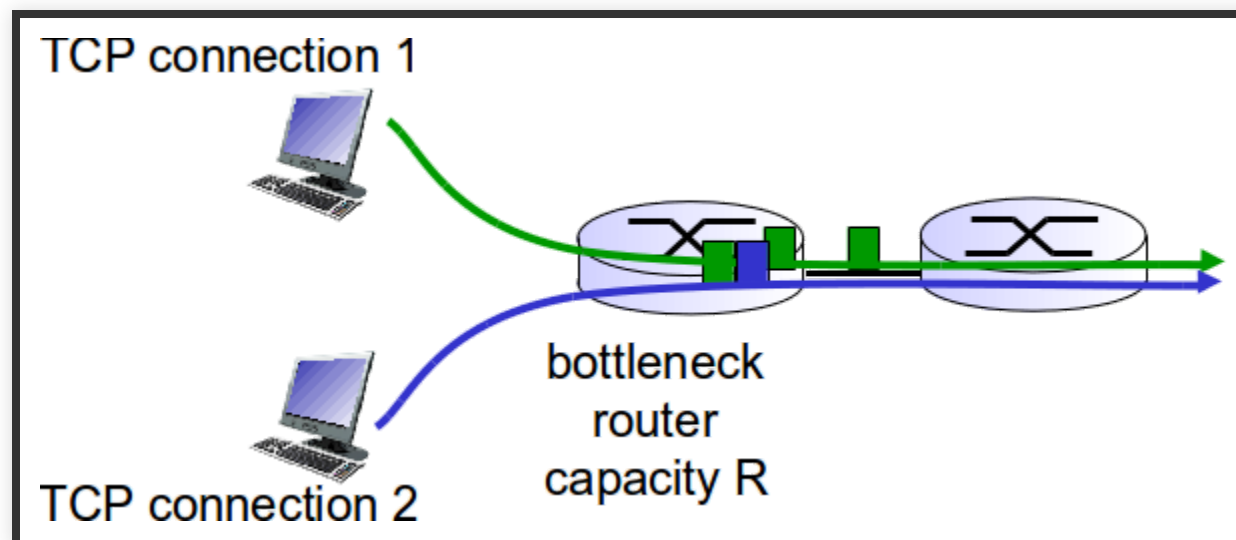
TCP Cubic is currently the default linux congestion-control implementation.

TCP Cubic has a number of interrelated features, in an attempt to address several TCP issues:

# TCP FAIRNESS

## ! Fairness goal:

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

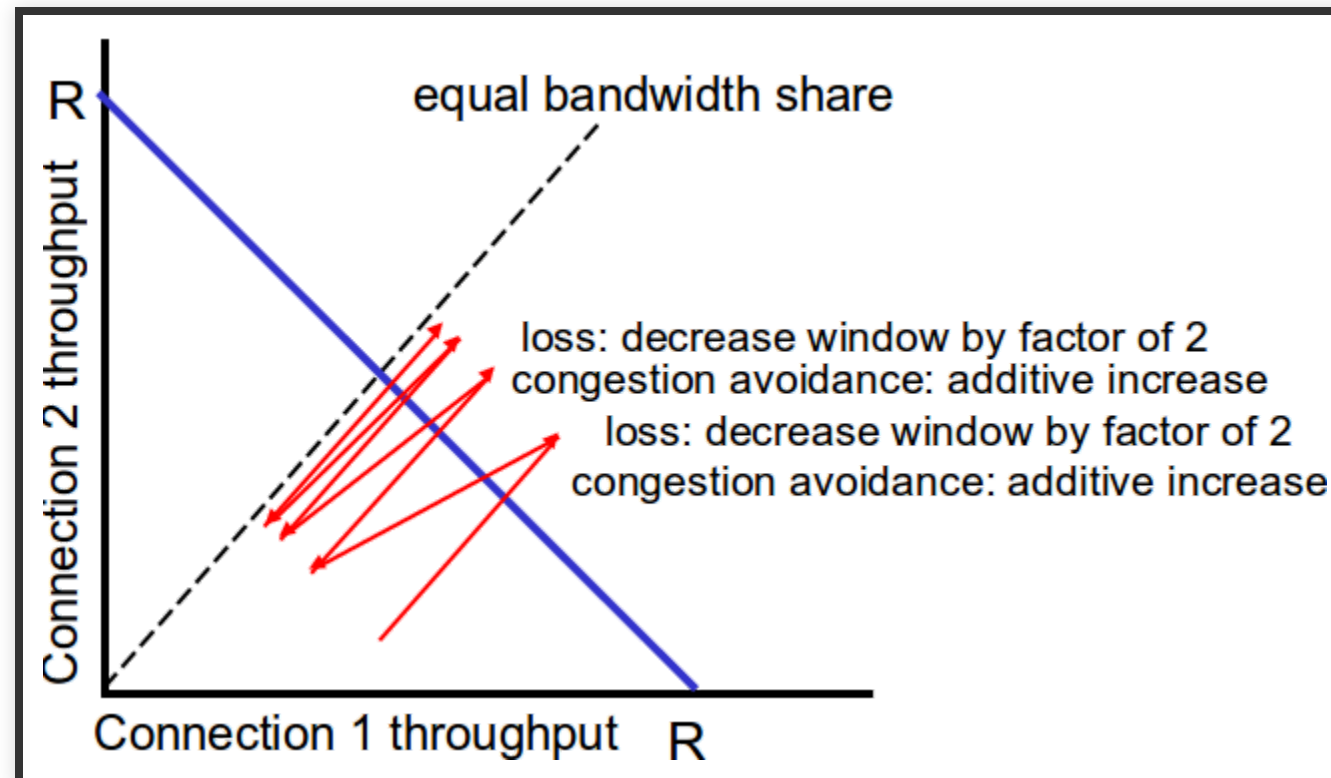




# WHY IS TCP FAIR?

Two competing sessions:

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



# FAIRNESS (MORE)

## ! 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

# FAIRNESS (MORE)

## ! 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:
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - TCP

## Next:

- leaving the network “edge” (application, transport layers)
- into the network “core”