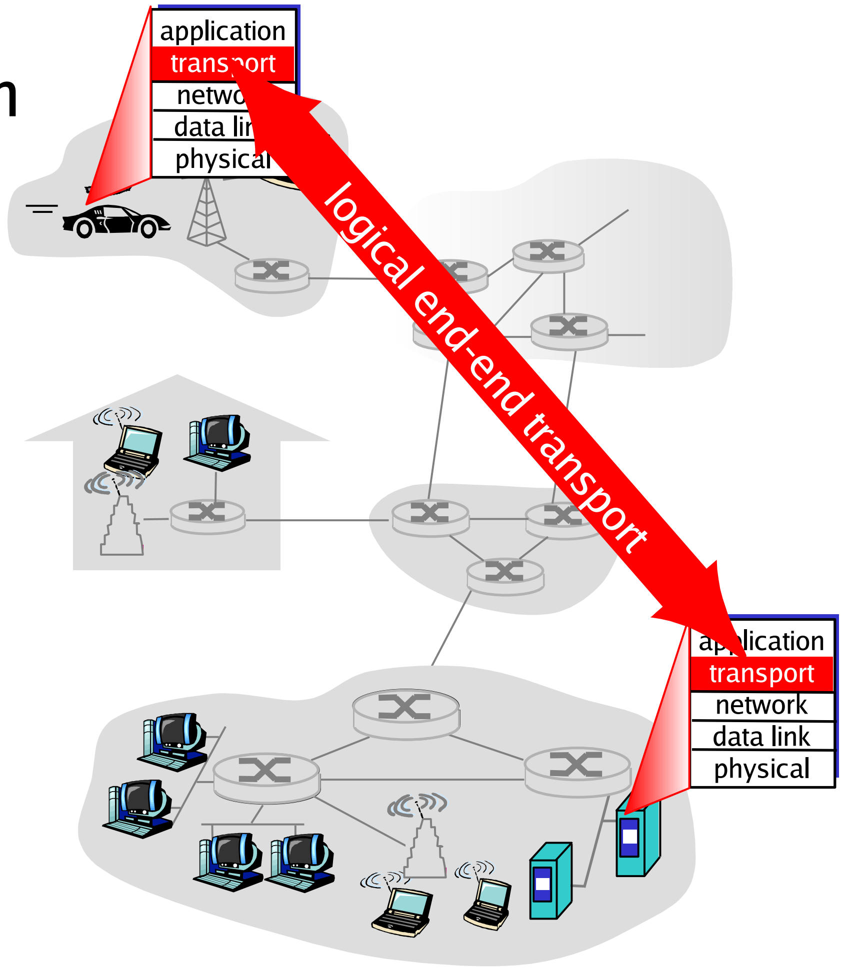
Chapter 3: Transport Layer

Objectives:

- To understand principles behind Transport Layer services:
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Understand the function and operation of Transport Layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

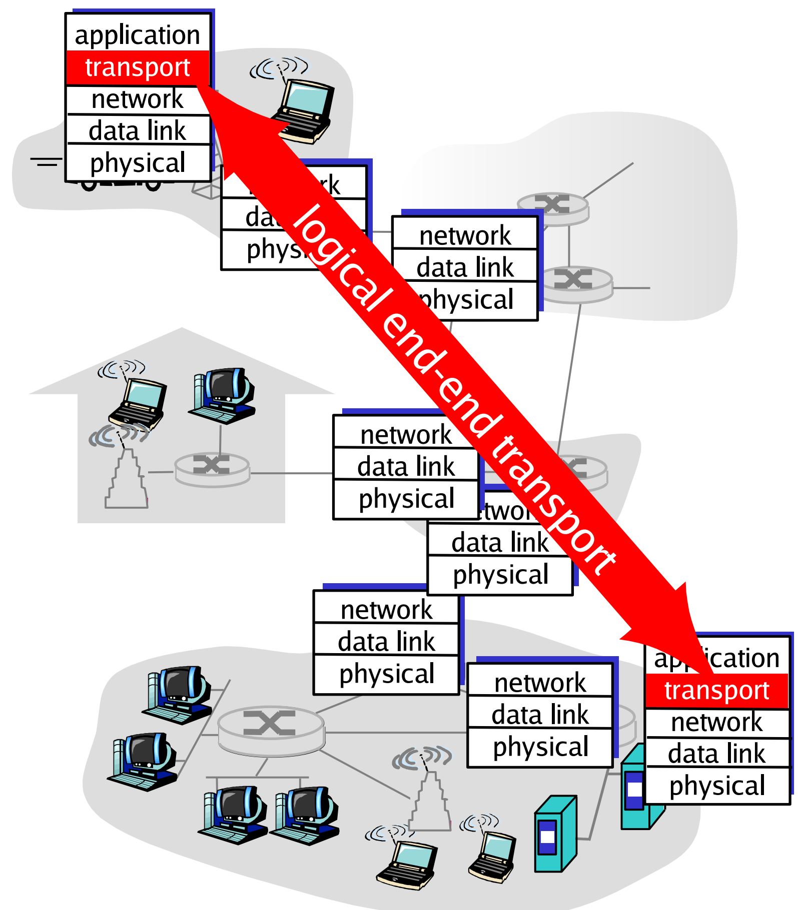
Transport Services and Protocols

- Provides a logical connection between **processes** running on different hosts
- On end systems
  - **Tx side**: breaks app messages into segments, passes to Network Layer
  - **Rx side**: reassembles segments into messages, passes to Application Layer
- Transport protocols in TCP/IP protocol suite: TCP and UDP



Internet Transport-Layer Protocols

- Reliable, in-order delivery: TCP
  - Congestion control
  - Flow control
  - Connection setup
- Unreliable, unordered delivery: UDP
  - No-frills extension of “best-effort” IP
- Services not implemented:
  - Delay guarantees
  - Bandwidth guarantees



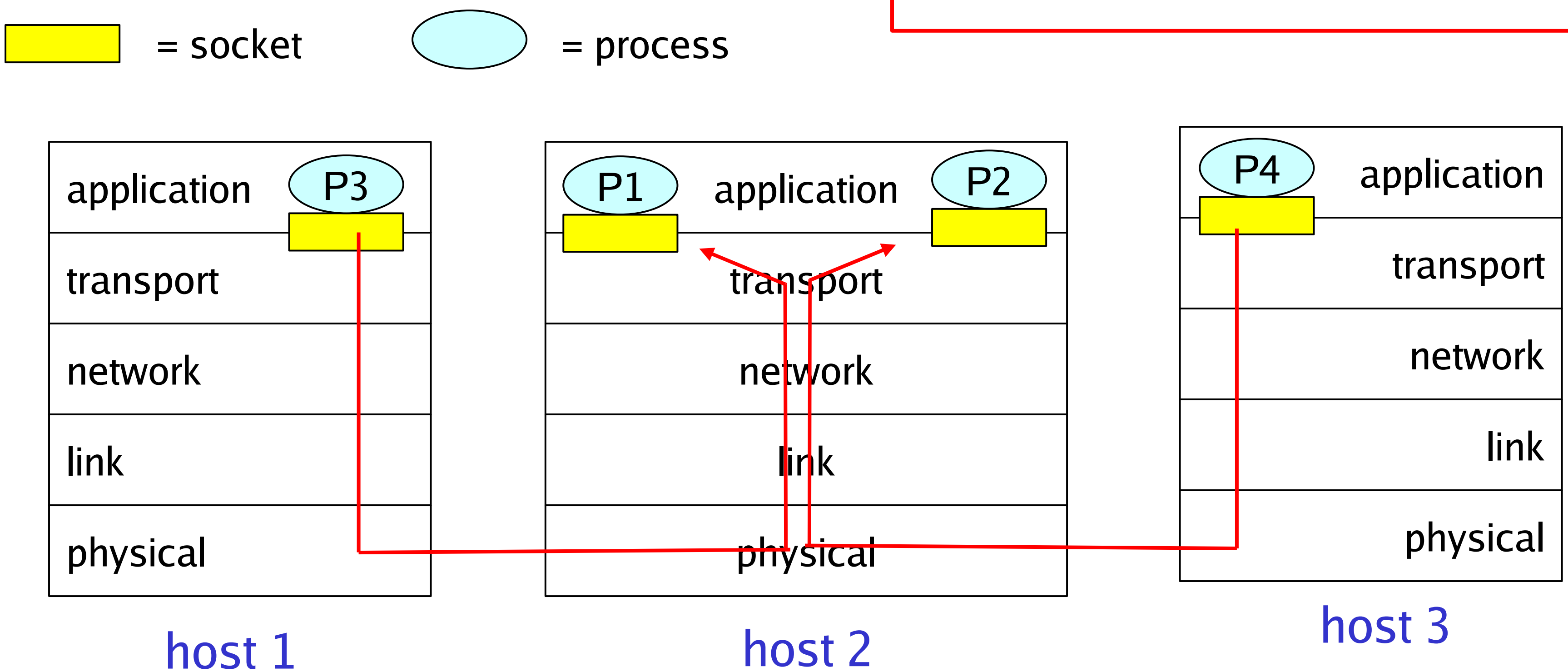
Multiplexing/Demultiplexing

Demultiplexing at rcv host:

Delivering received segments to correct socket

Multiplexing at send host:

Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

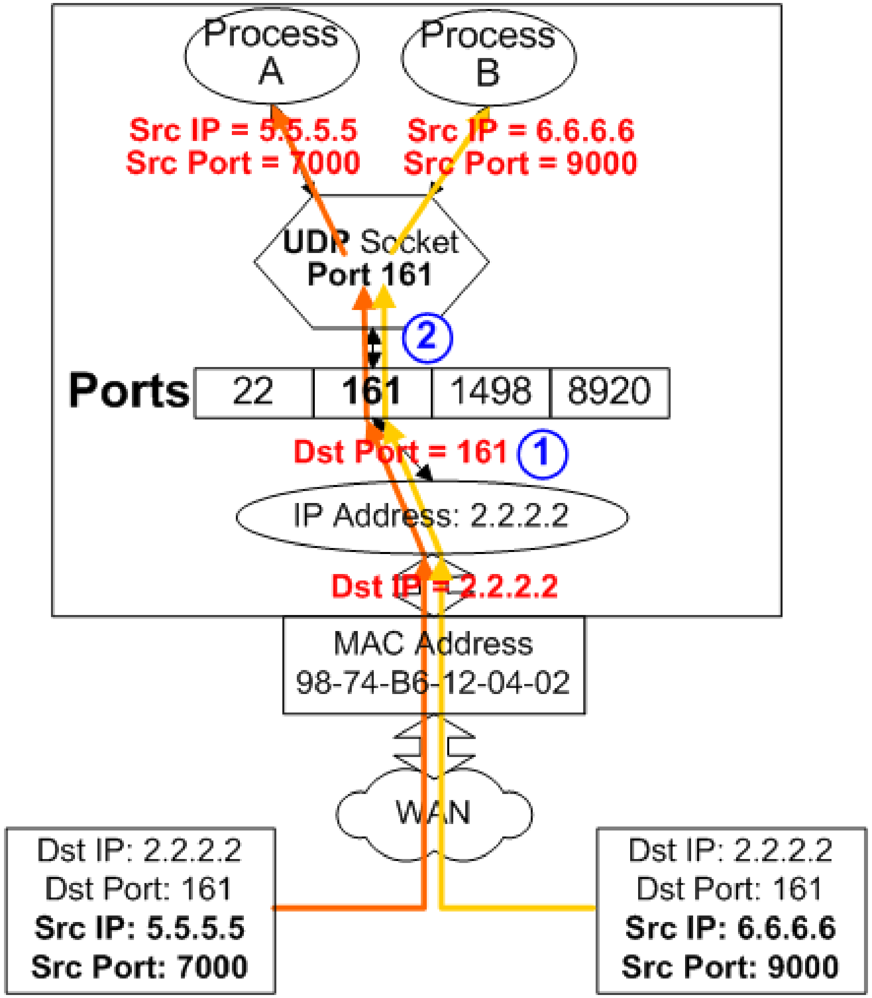


Connectionless Demultiplexing

- Create sockets and bind them to ports/IP Address
- UDP socket identified by two-tuple:
  - Dest IP address
  - Dest port number
- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

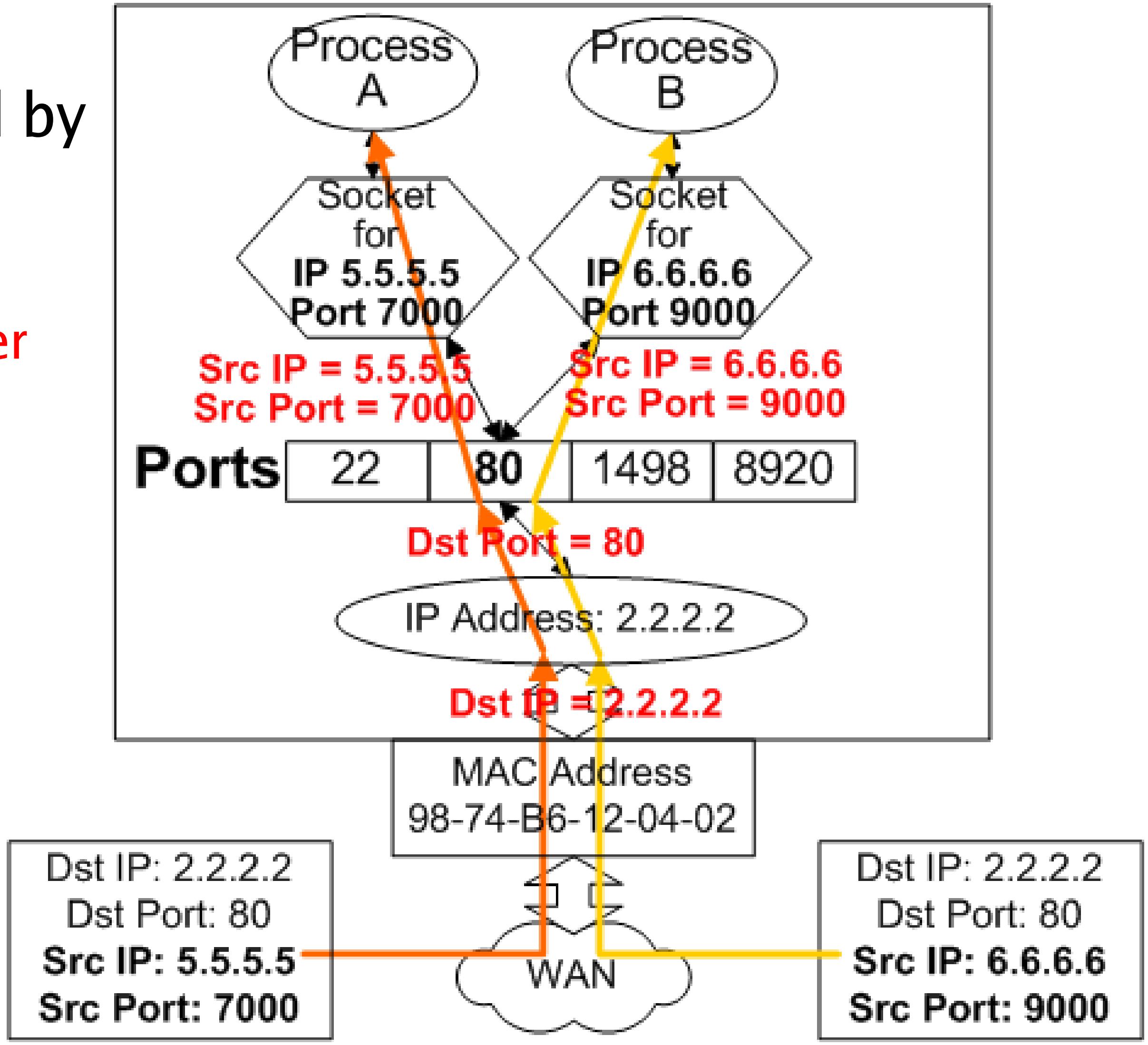
Connectionless Demux

- UDP socket is identified by two-tuple:
  - Dest IP address
  - Dest port number
- When host receives UDP segment:
  1. checks destination port number in segment
  2. directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket



Connection-Oriented Demux

- Socket is identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Server host may support many simultaneous TCP sockets:



UDP: User Datagram Protocol

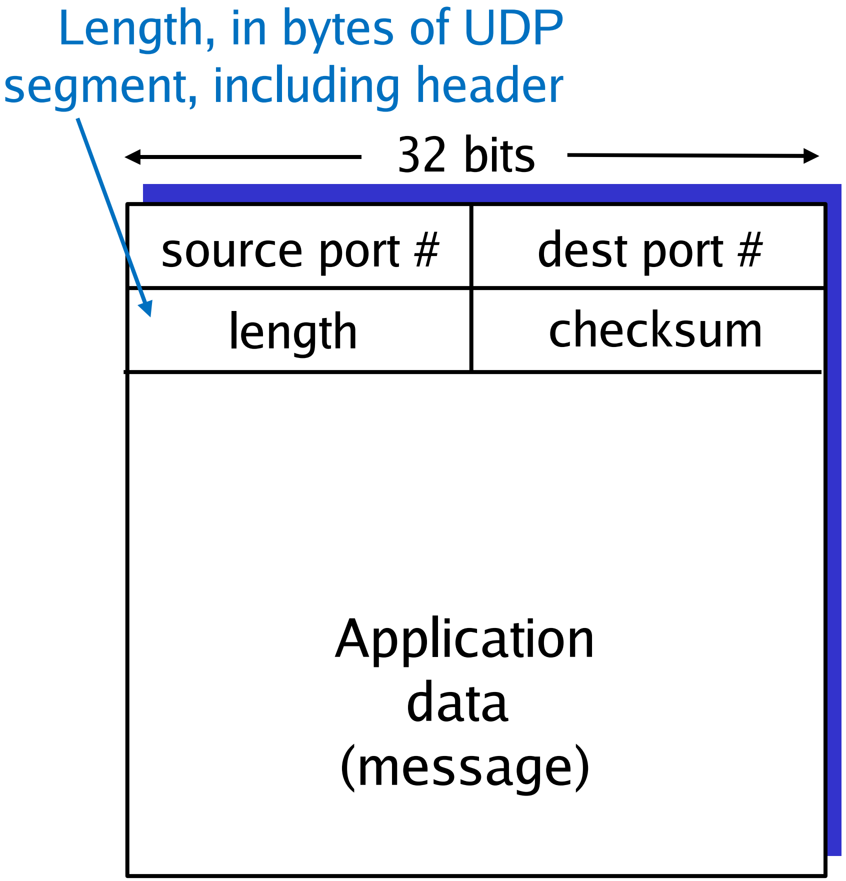
- "Best Effort" service; UDP segments may be:
  - Lost
  - Delivered out of sequence to application
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state maintenance at sender, receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired

UDP (cont'd)

- Typically used for streaming multimedia applications
  - Loss tolerant
  - Rate sensitive
- Other UDP uses
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery.



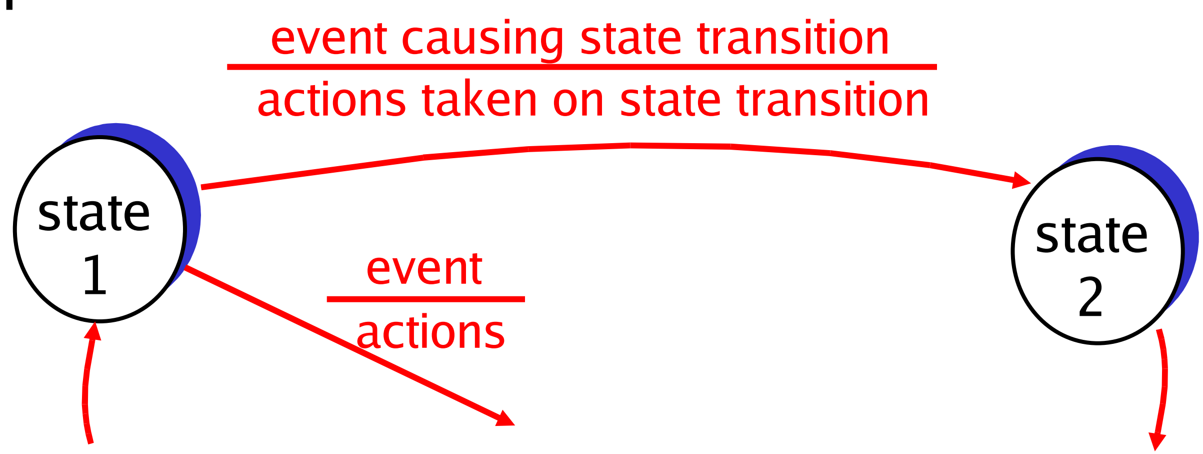
UDP segment format

Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event



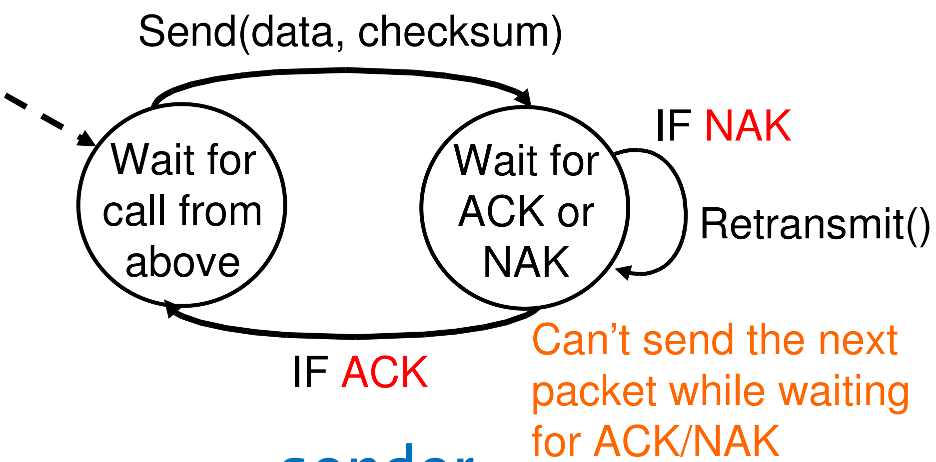
Reliable Transfer over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver read data from underlying channel

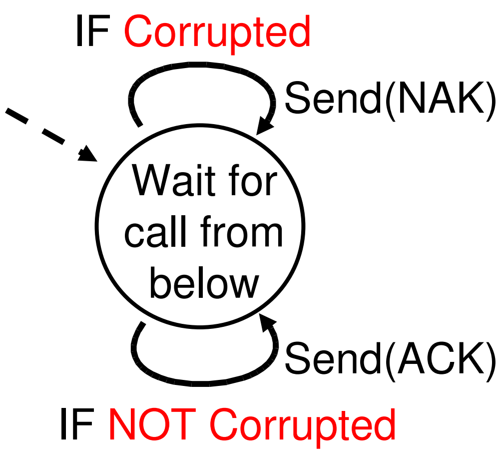
Channel with Bit Errors

- Underlying channel may flip bits in packet
  - **Checksum** to detect bit errors
- Question: how to recover from errors?
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - Sender ***retransmits*** pkt on receipt of NAK
- New mechanisms
  - **Error detection**
  - **Receiver feedback**: control msgs (ACK,NAK) rcvr to sender

Channel with Bit Errors: FSM specification



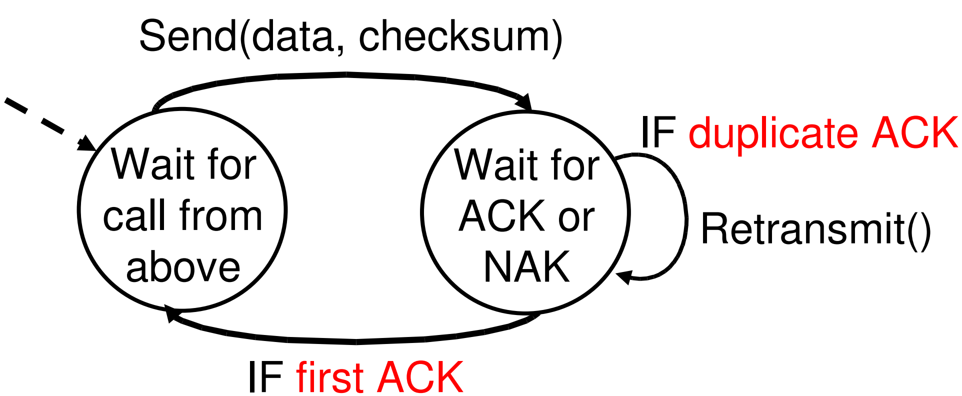
sender



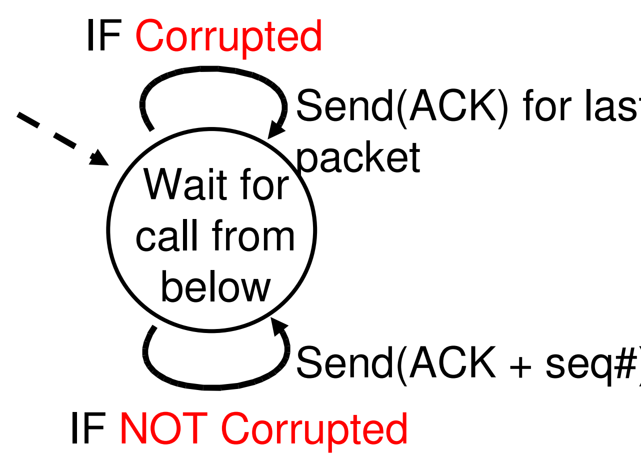
receiver

NAK-free Protocol

- Same functionality as previous, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*



sender



receiver

Channels with Errors and Loss

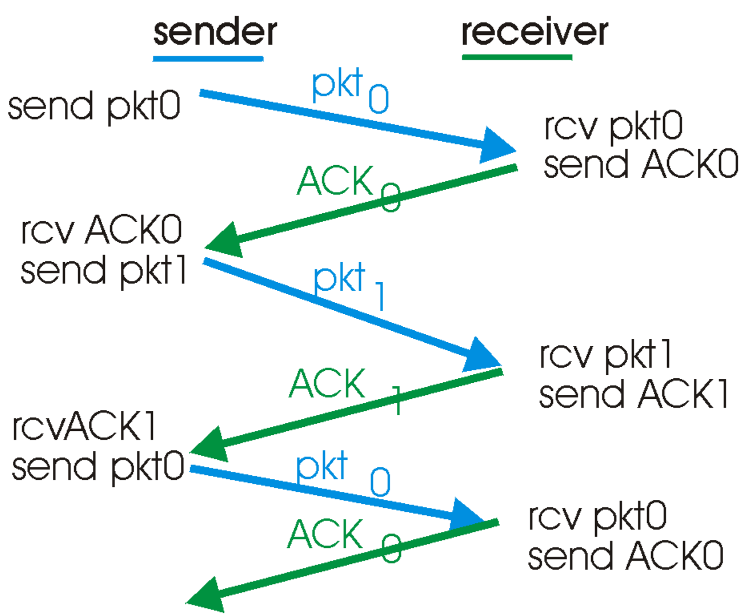
New assumption: underlying channel can also lose packets (data or ACKs)

- Checksum, seq. #, ACKs, retransmissions will be of help, but not enough

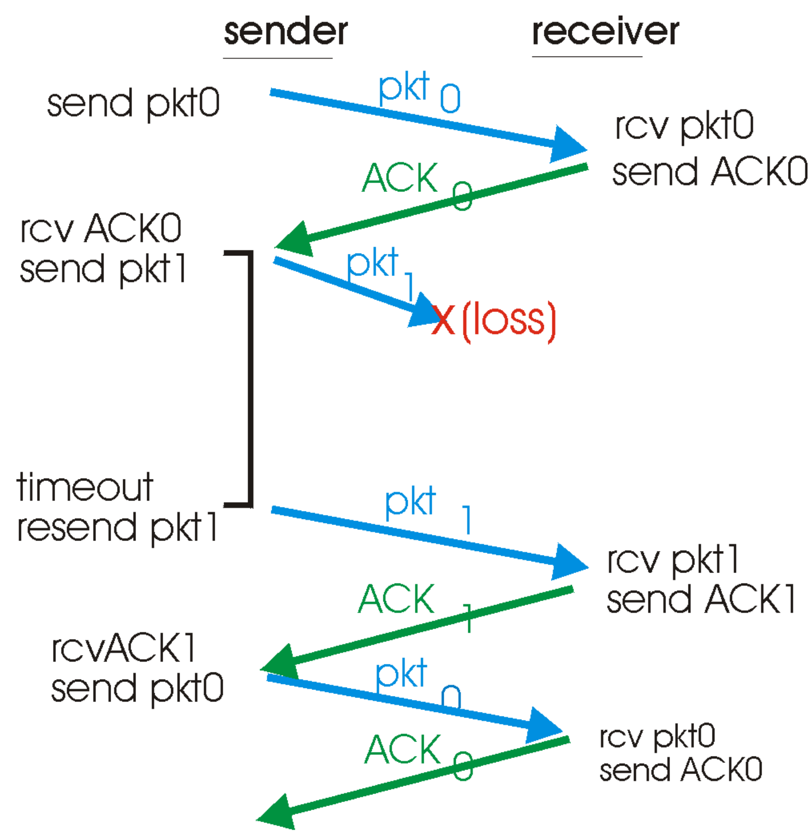
Approach: sender waits "reasonable" amount of time for ACK (Stop-and-Wait)

- Retransmits if no ACK received in this time
- If pkt (or ACK) just delayed (not lost):
  - Retransmission will be duplicate, but use of seq. #s already handles this
  - Receiver must specify seq # of pkt being ACKed
- Requires countdown timer

Channels with Errors and Loss

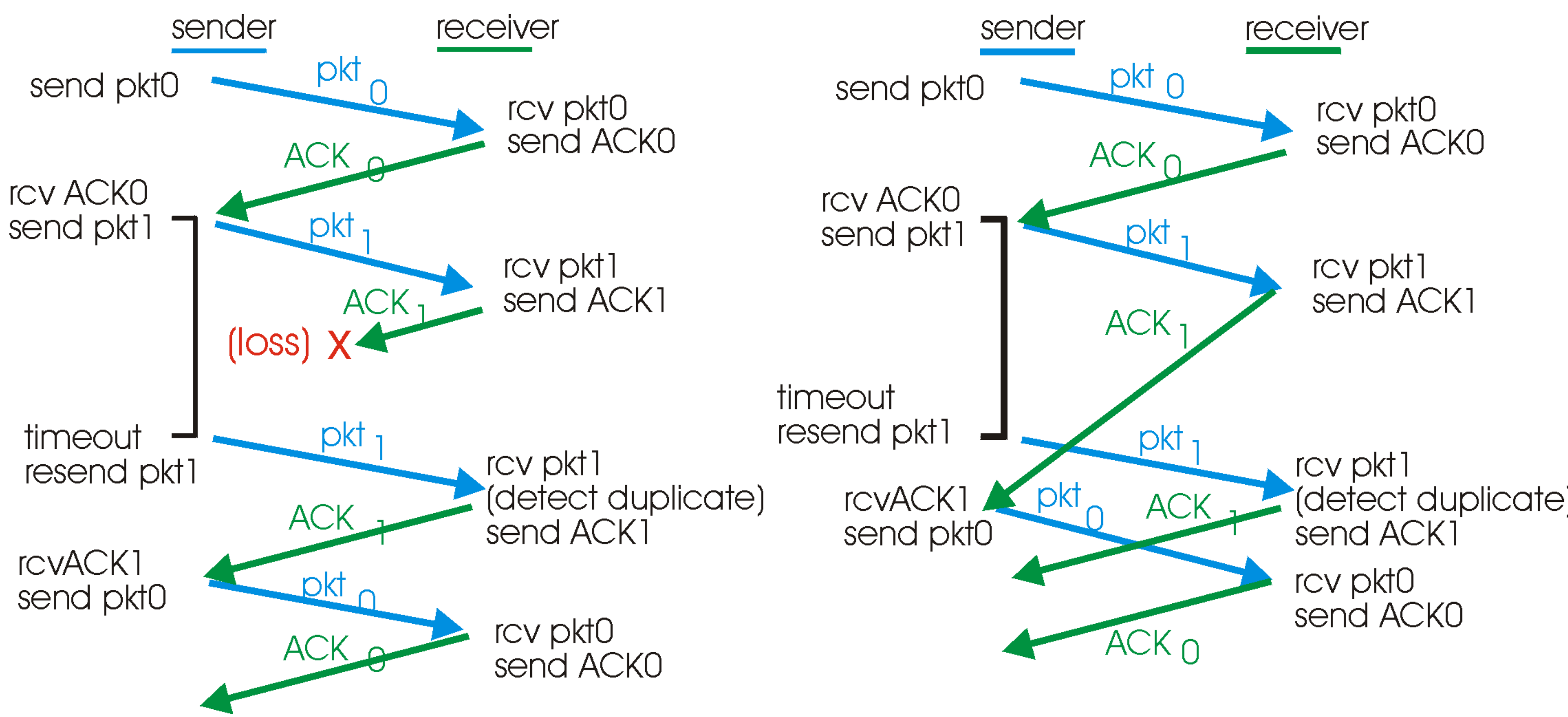


(a) operation with no loss



(b) lost packet

Channels with Errors and Loss

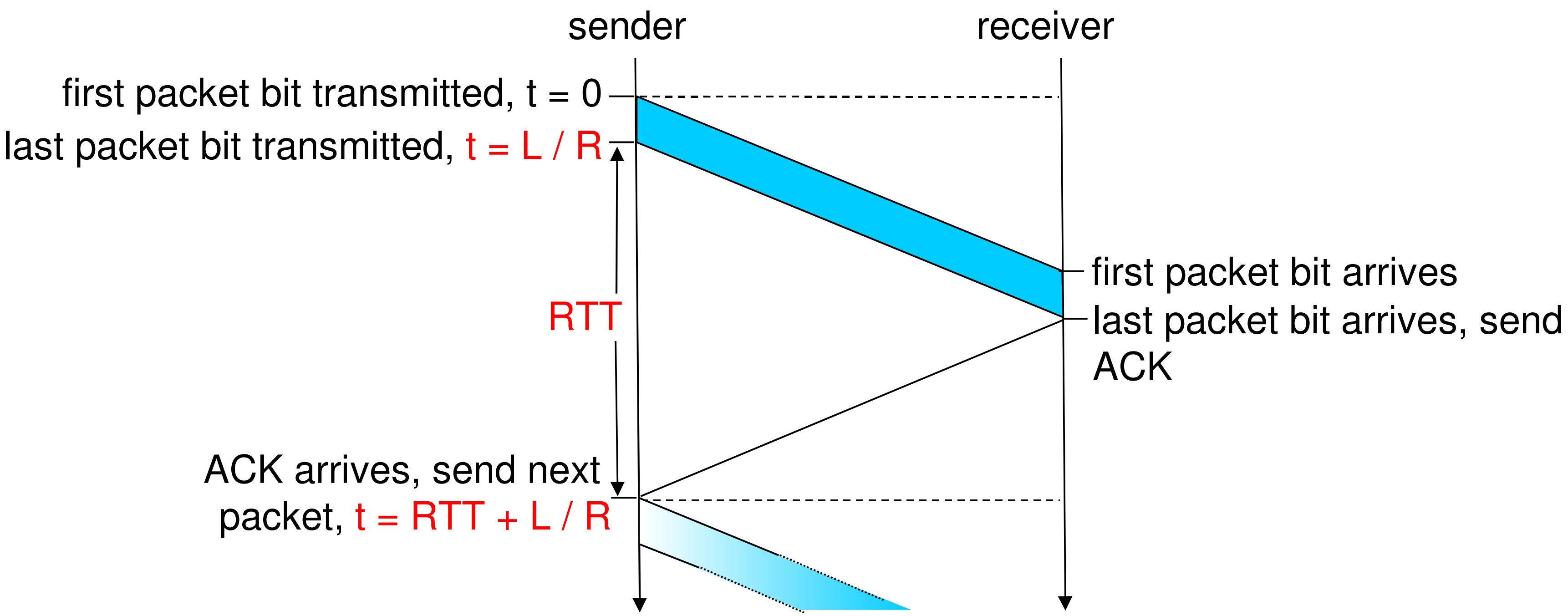


(c) lost ACK

(d) premature timeout

Performance of Stop-and-Wait operation

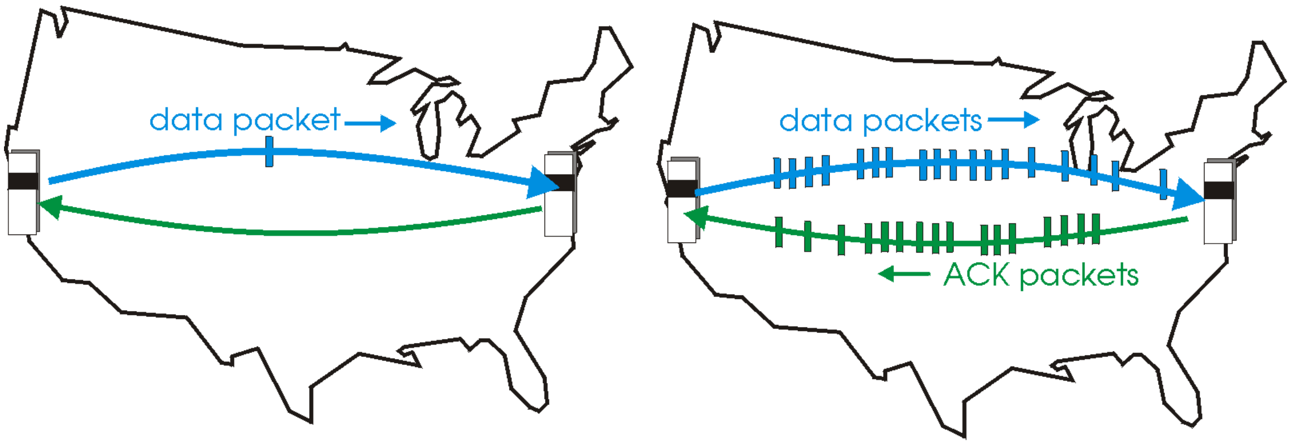
Utilization of the channel is very low!



Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

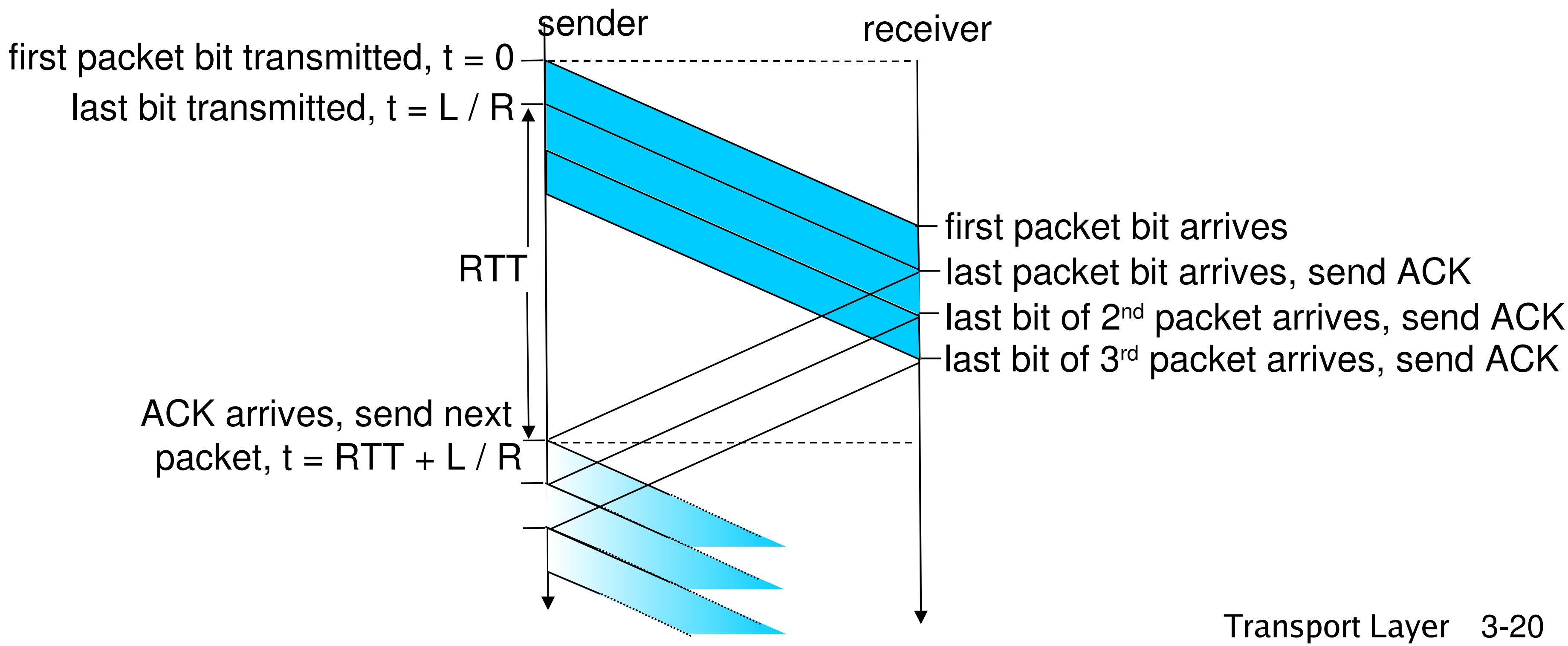
- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining Protocol: Increased Utilization

Sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

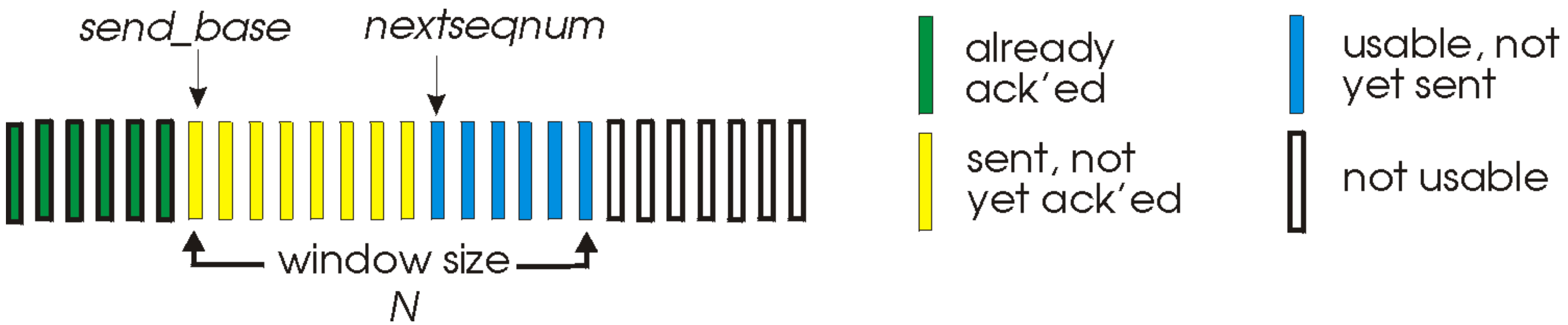
- Protocols: *go-Back-N, selective repeat*



Go-Back-N (GBN)

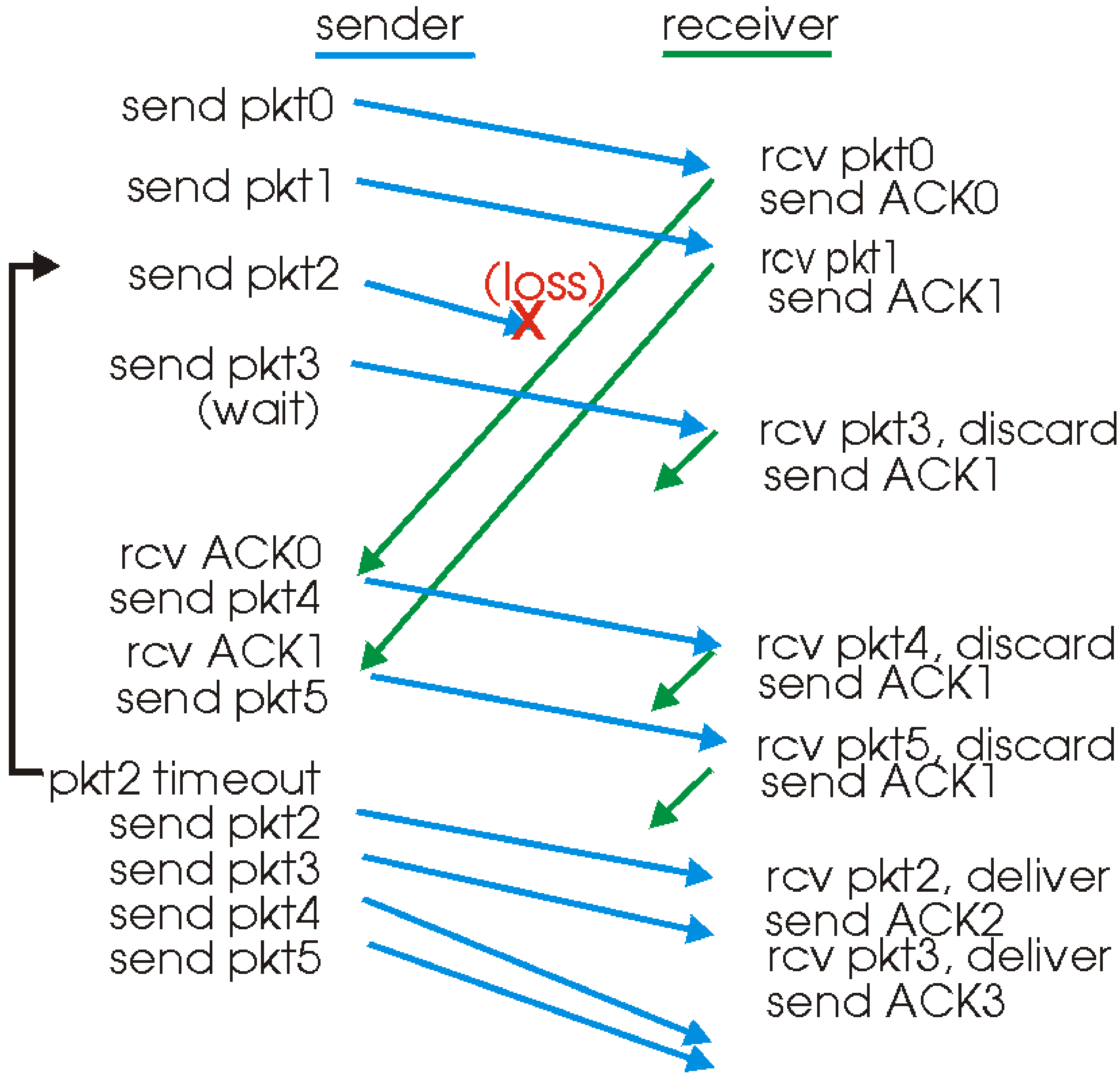
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unACK'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- *Timeout(n):* retransmit pkt n and all higher seq # pkts in window
- **Receiver** sends ACK for correctly-received pkt with highest *in-order* seq #

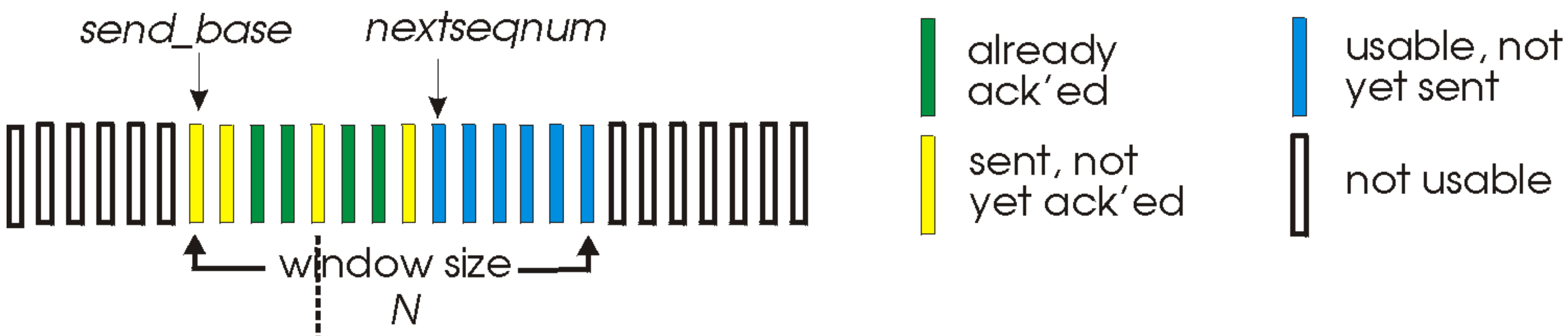
GBN in action



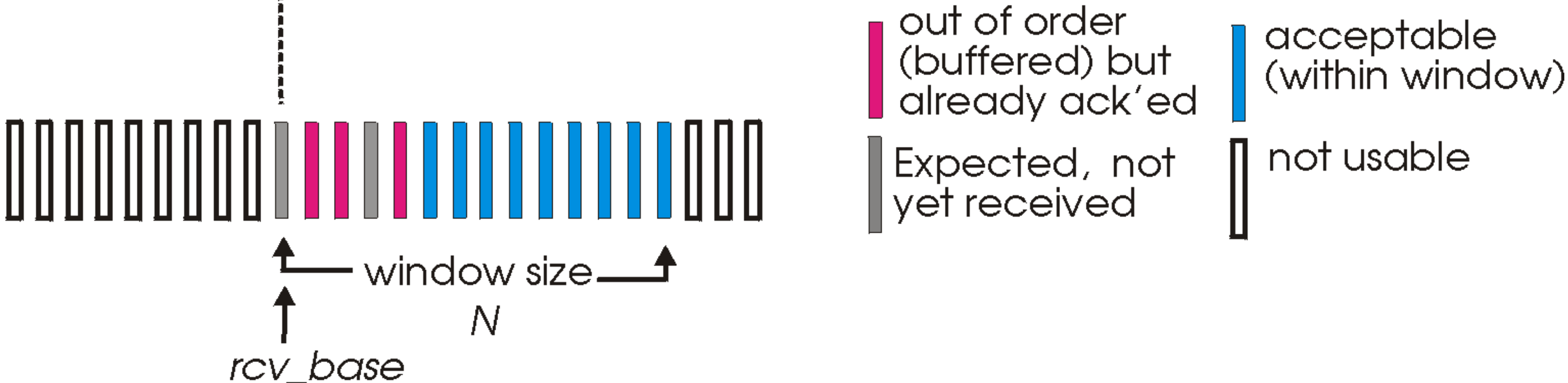
Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers packets, as needed, arranges the sequence for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unACK'ed pkt
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACK'ed pkts

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective Repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

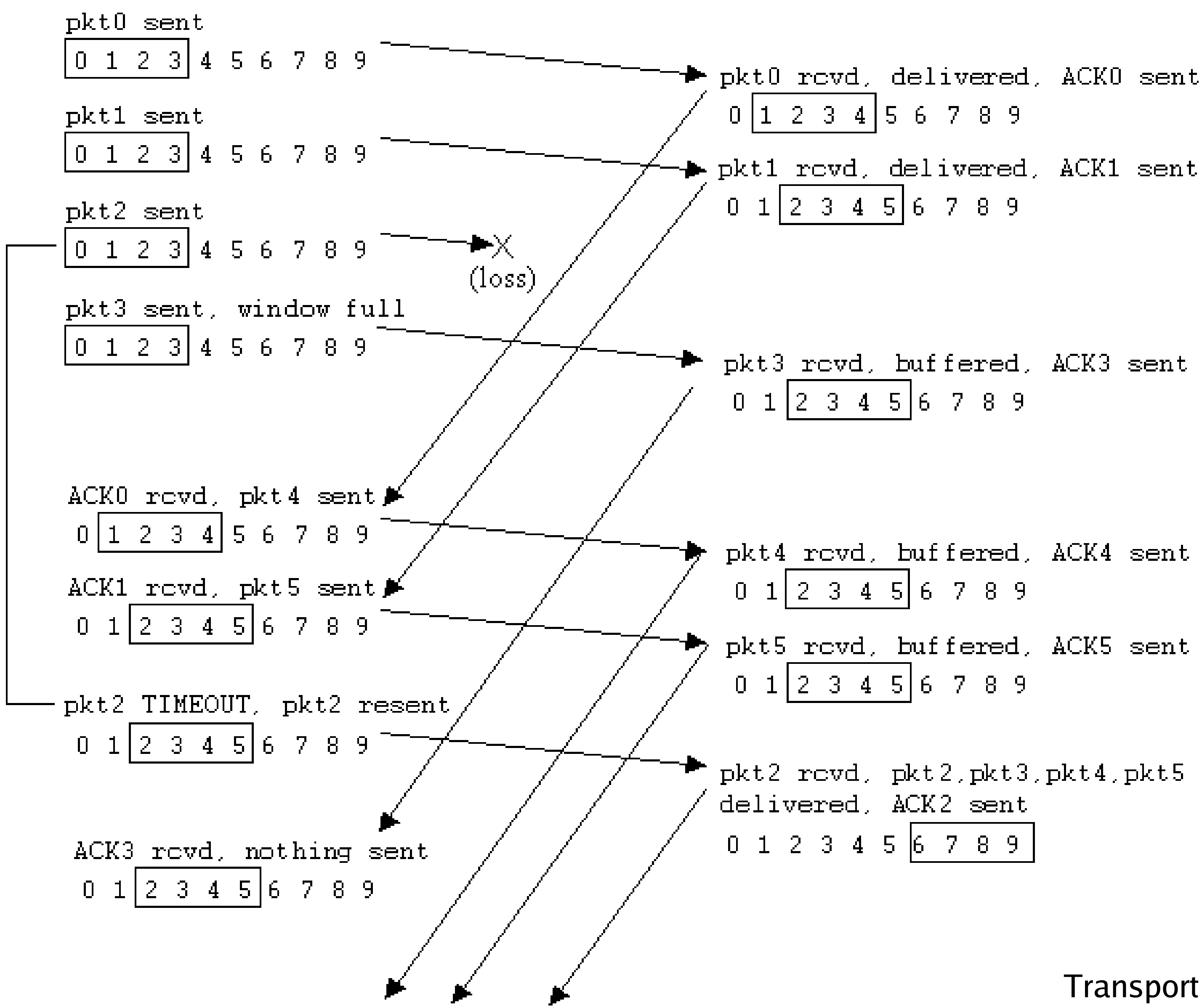
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

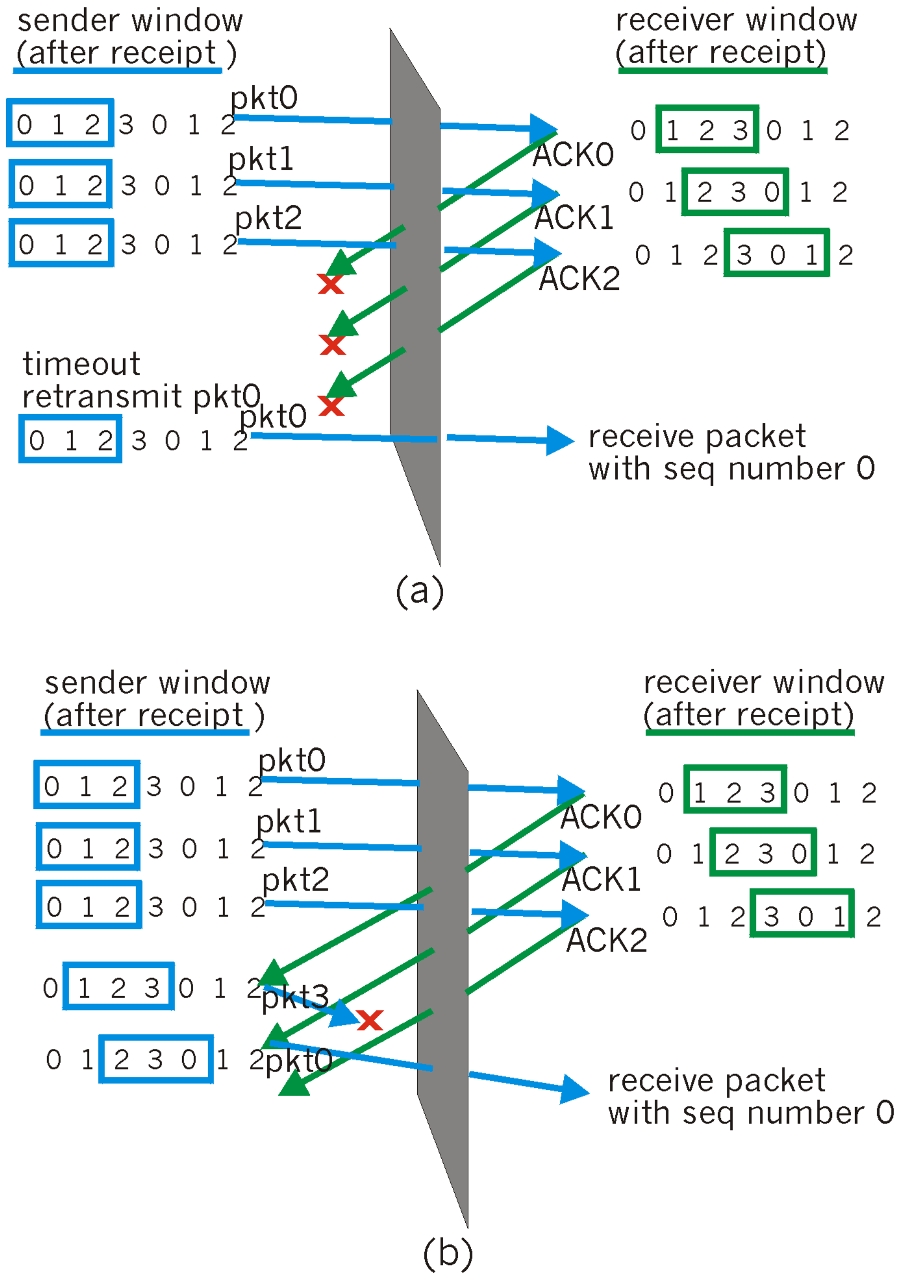
Selective Repeat in Action



Selective Repeat: Protocol Failure

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

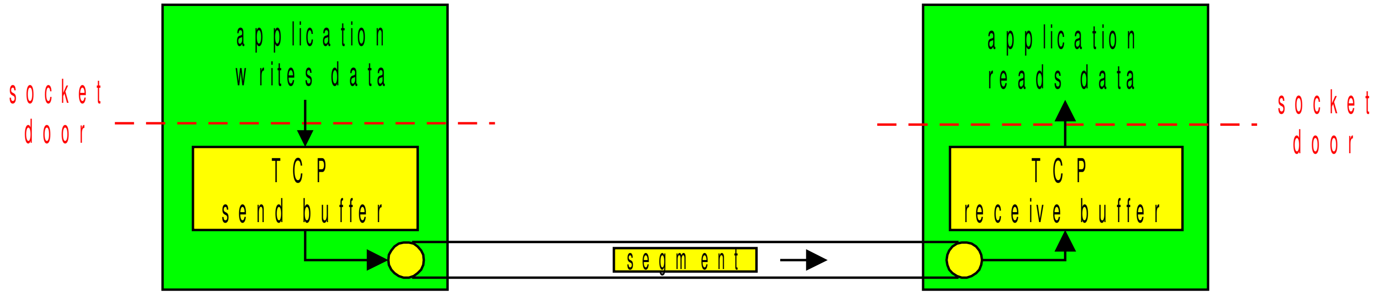


- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

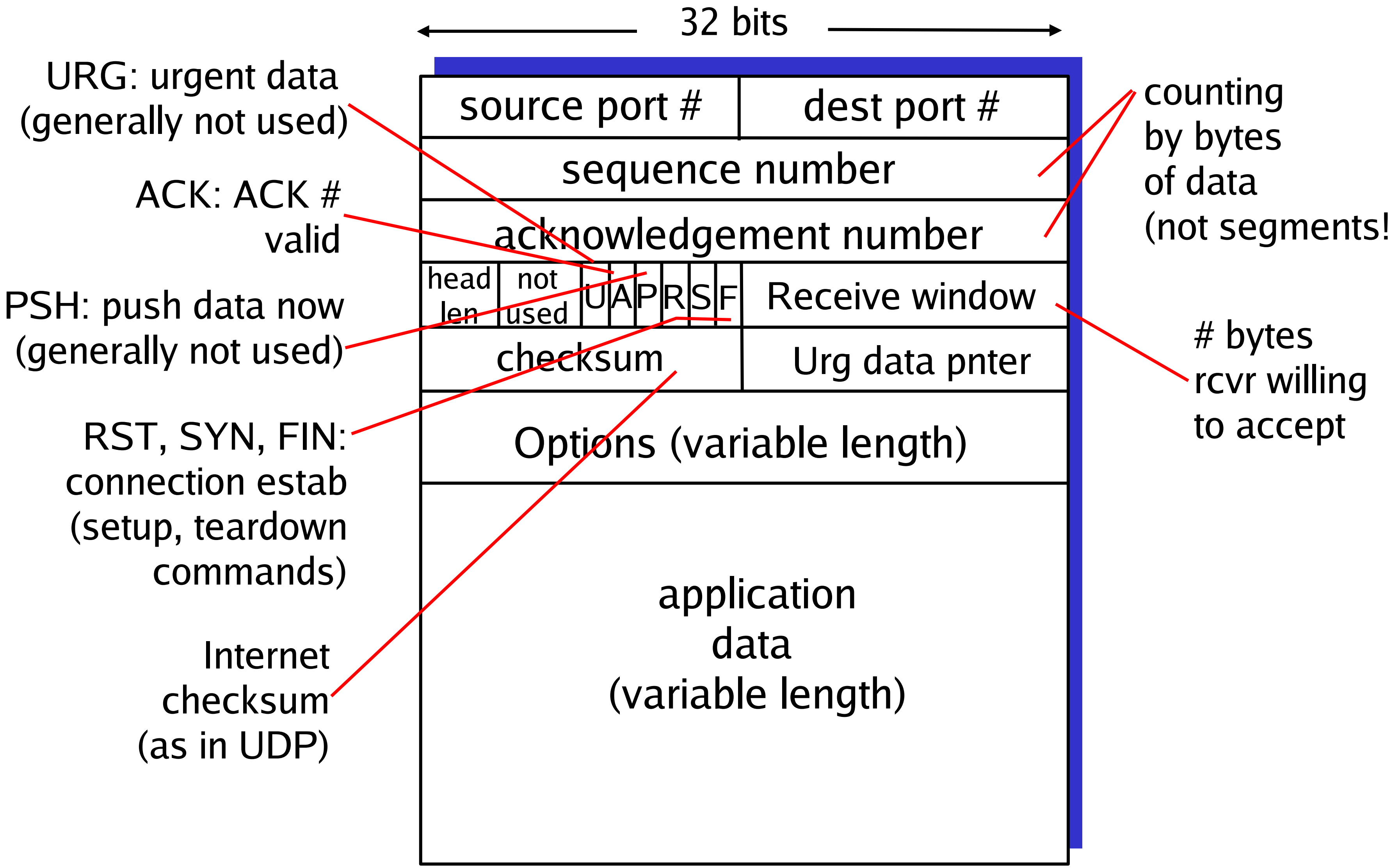
Q: what relationship between seq # size and window size?

TCP: Overview

- Point-to-point:
  - one sender, one receiver
- Reliable, in-order *byte steam:*
  - no "message boundaries"
- Pipelined:
  - TCP congestion and flow control set window size
- *Send & receive buffers*
- Flow controlled:
  - sender will not overwhelm receiver
- 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



TCP Segment Structure



TCP seq. #s and ACKs

Seq. #s:

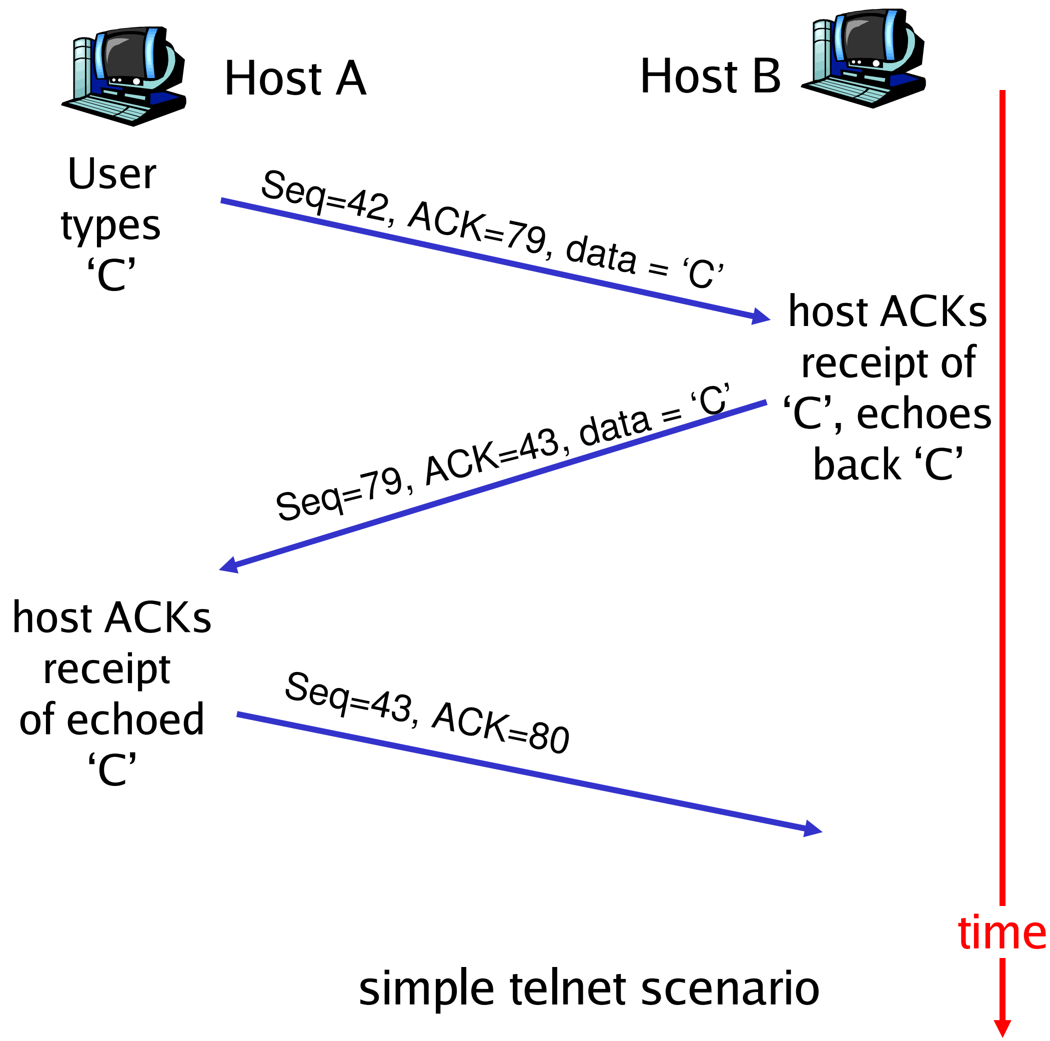
- Byte stream "number" of first byte in segment's data

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 implementation



simple telnet scenario

TCP Round Trip Time and 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

Q: How to estimate RTT?

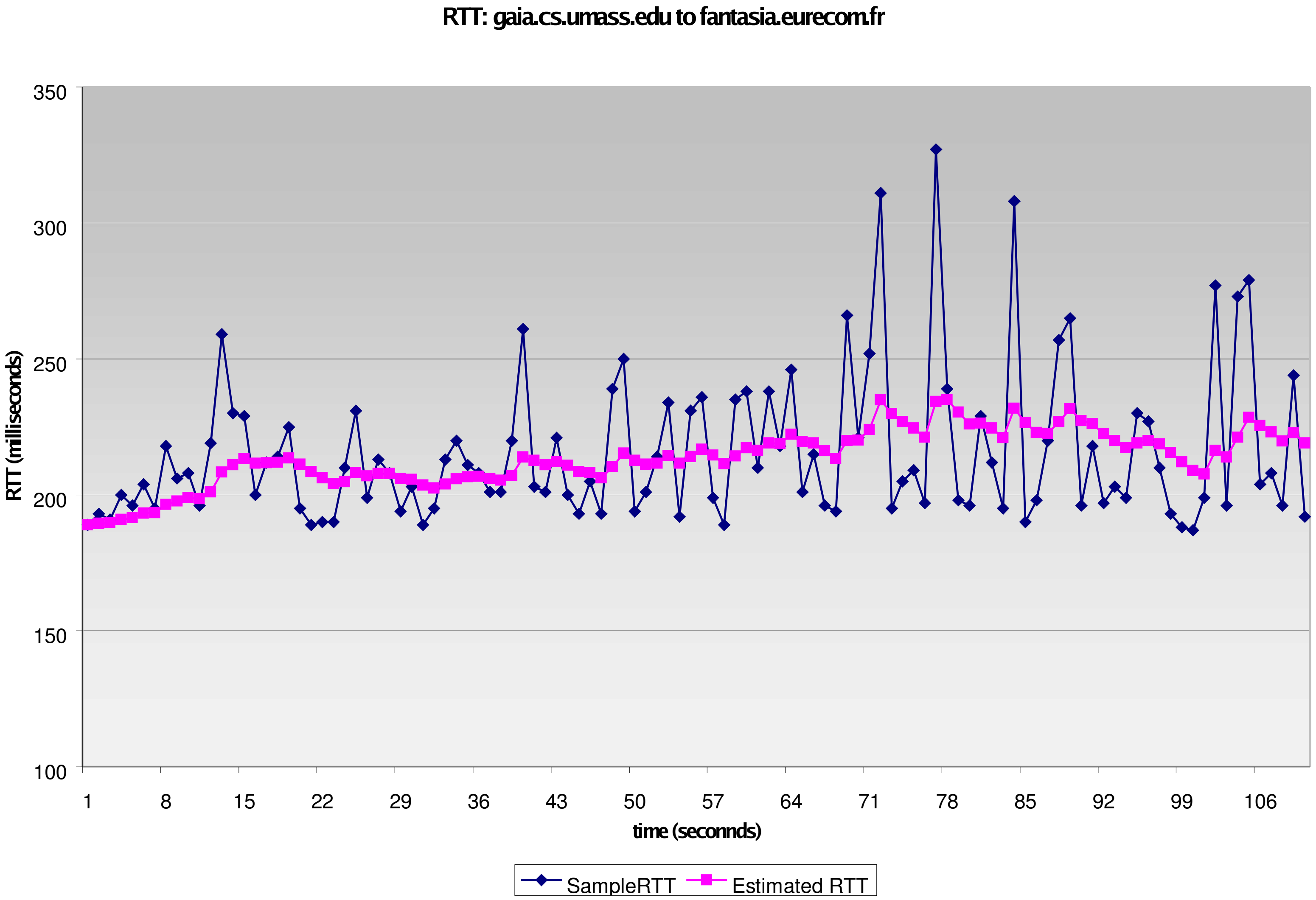
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - Ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

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

Example RTT Estimation



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimtedRTT** plus "safety margin" (proposed by Jacobsen)
  - Large variation in **EstimatedRTT** -> larger safety margin
- First estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate ACKs
- Initially consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP Sender Events

Data rcvd 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

Timeout:

- Retransmit segment that caused timeout
- Restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - Update what is known to be acked
  - Start timer if there are outstanding segments

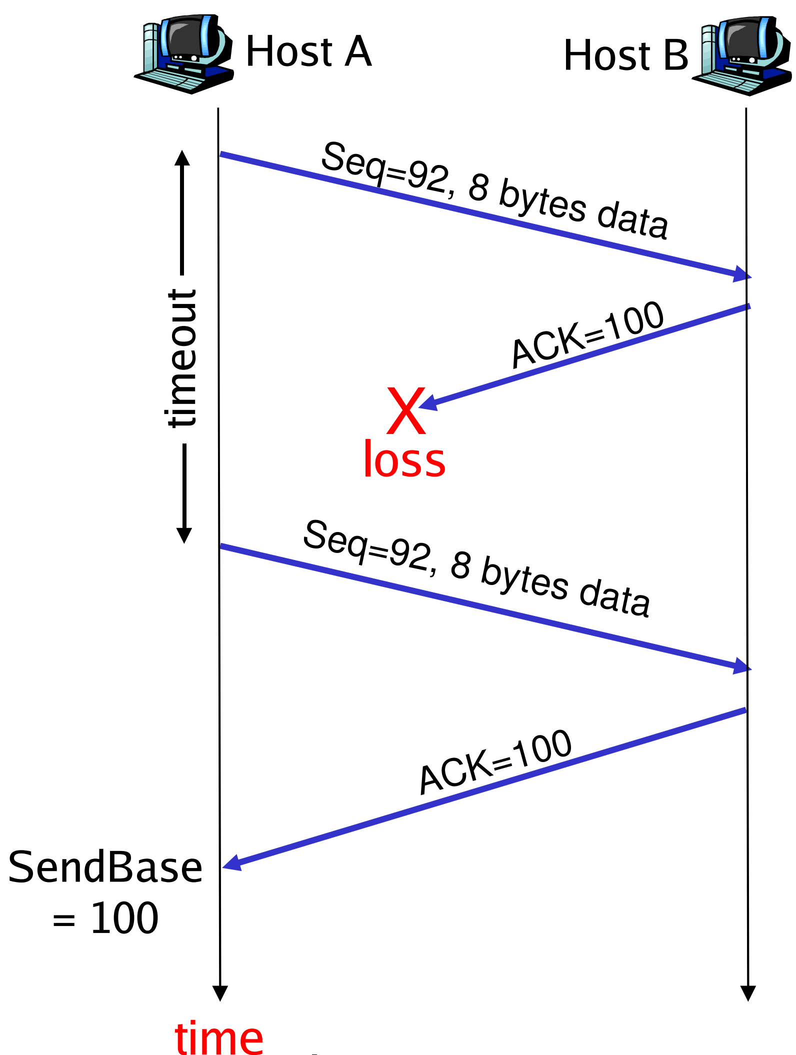
TCP Sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
            start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

    event: ACK received, with ACK field value of y
       if (y > SendBase) {
            SendBase = y
           if (there are currently not-yet-acknowledged segments)
                start timer
           }

  }  /* end of loop forever */
```

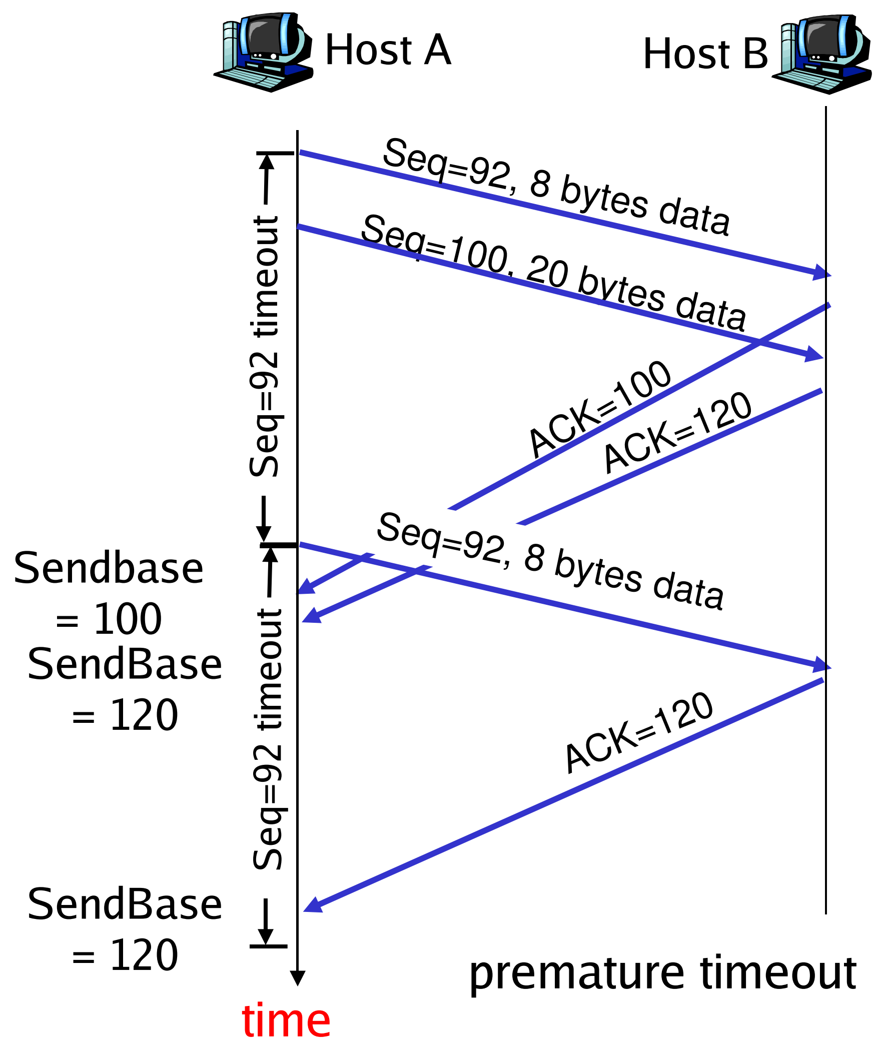
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: Retransmission Scenarios

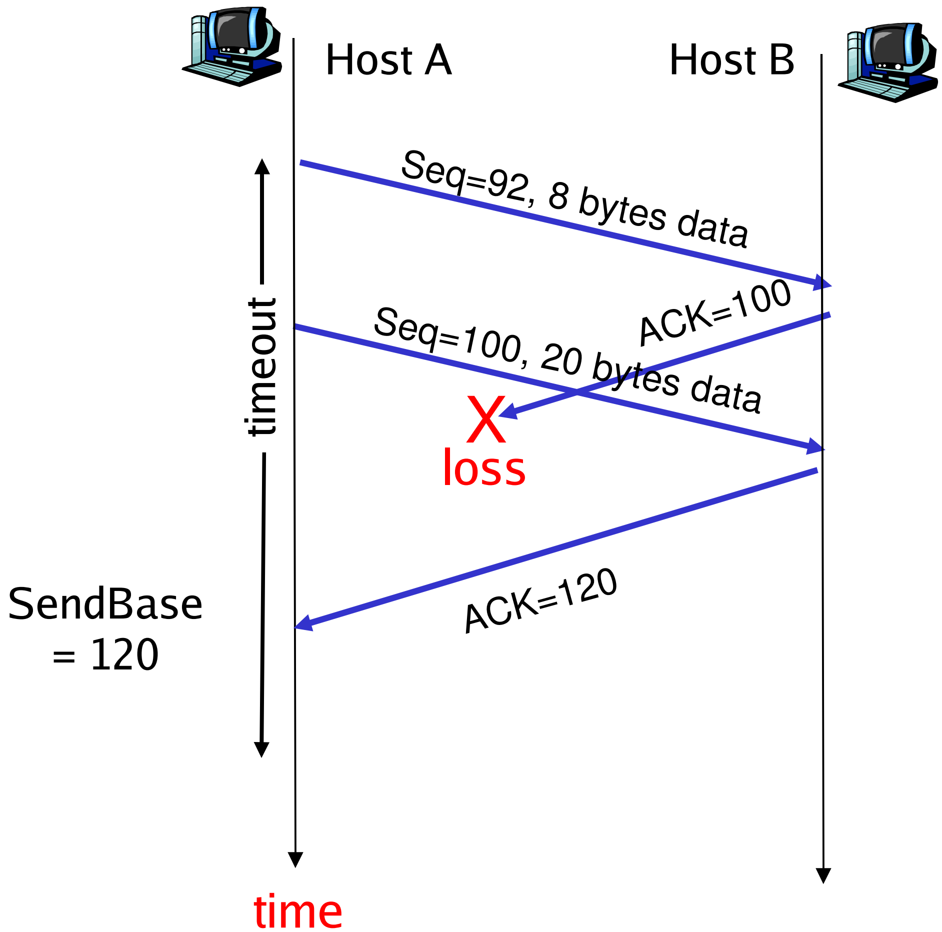


lost ACK scenario



premature timeout

TCP Retransmission Scenarios (more)



Cumulative ACK scenario

TCP ACK Generation

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
Arrival of out-of-order segment higher-than-expect 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 startsat lower end of gap

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.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires

Fast Retransmit Algorithm:

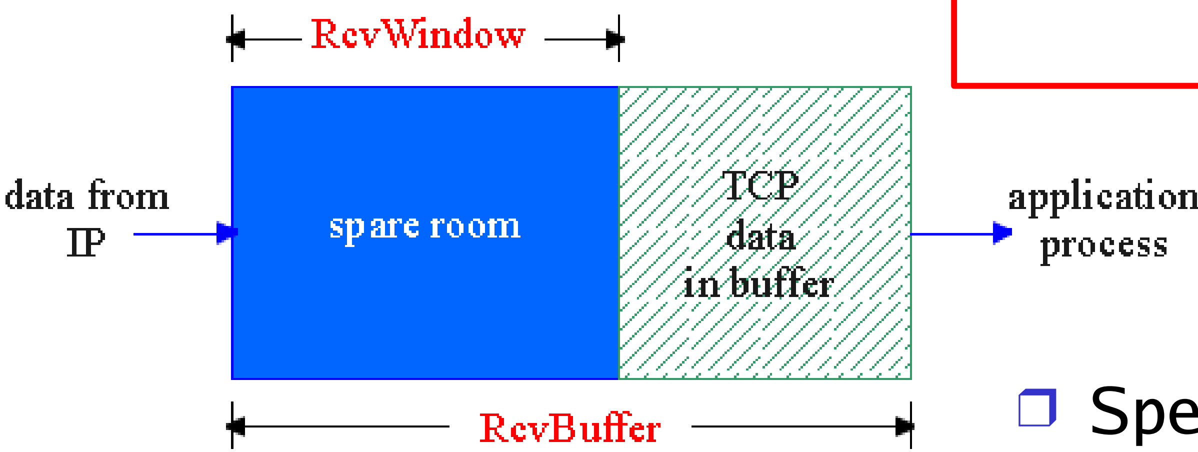
```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

fast retransmit

TCP Flow Control

- Receive side of TCP connection maintains a receive buffer:

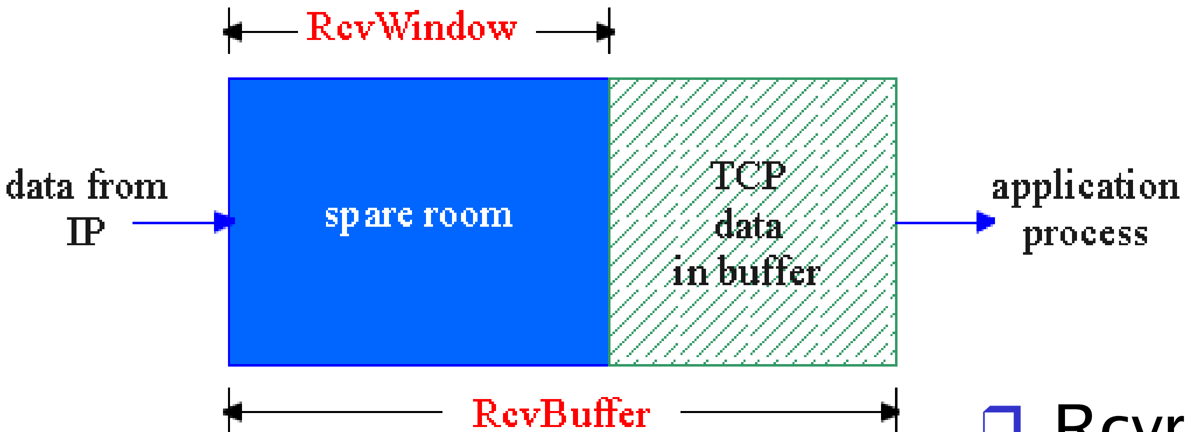


- Rx process may be slow at reading from buffer

Flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

- Speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

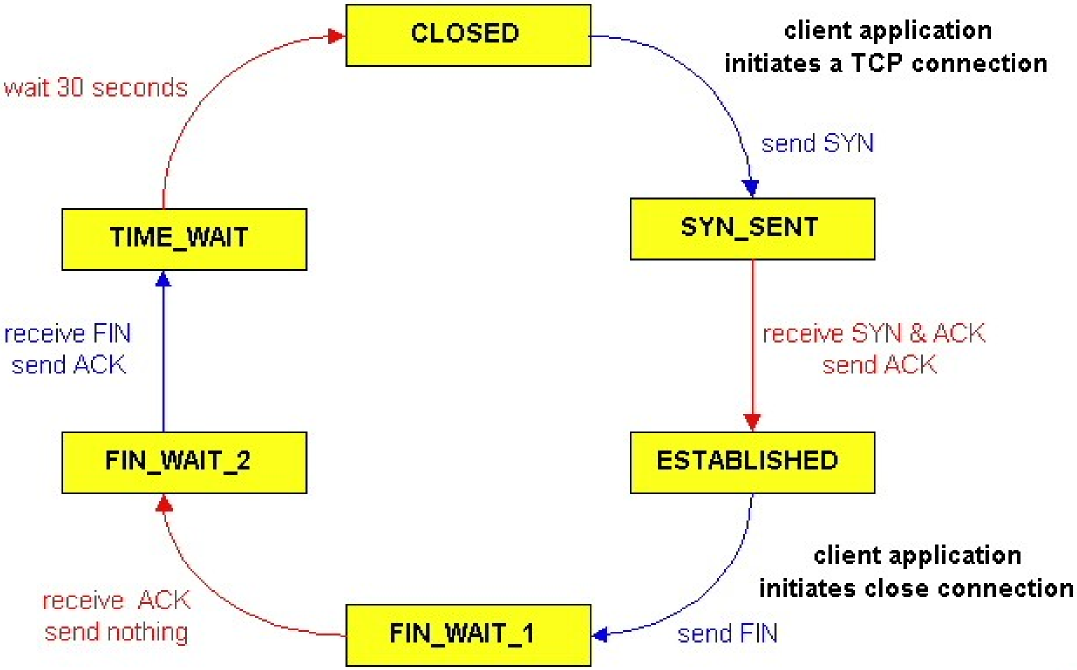
- Buffer space remaining

= **RcvWindow**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

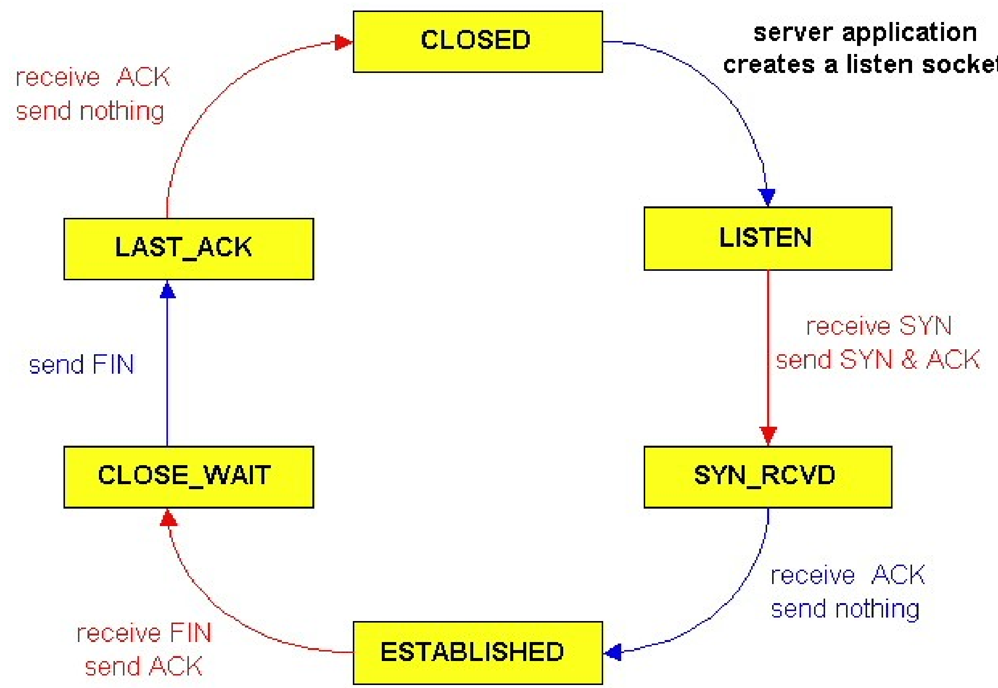
- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - Guarantees receive buffer doesn't overflow

TCP Connection Management



TCP client lifecycle

TCP server lifecycle



Principles of Congestion Control

Congestion:

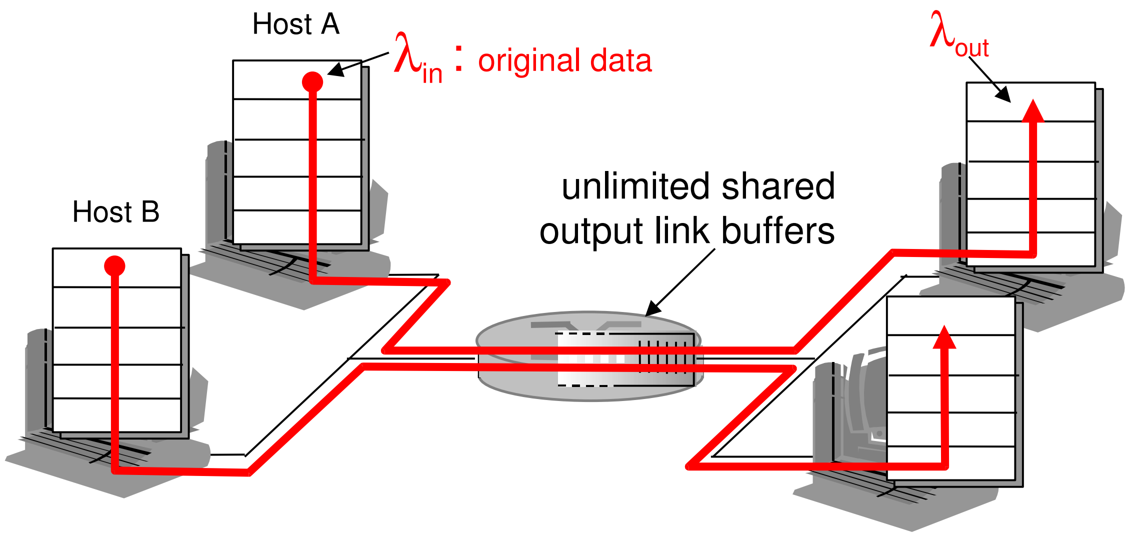
- A situation that occurs when too many packets are present a part of the subnet.
- Congestion control ensures that the subnet is able to carry the offered traffic.
- Directly related to the carrying capacity of the network.
- Flow control - primary function is to stop a fast sender from overwhelming a slow receiver.
- Manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

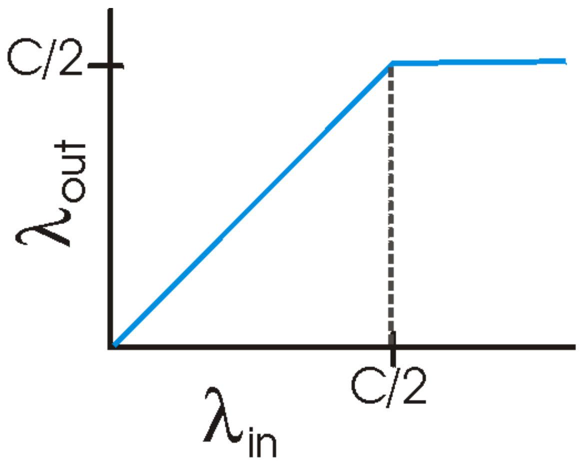
Principles of Congestion Control

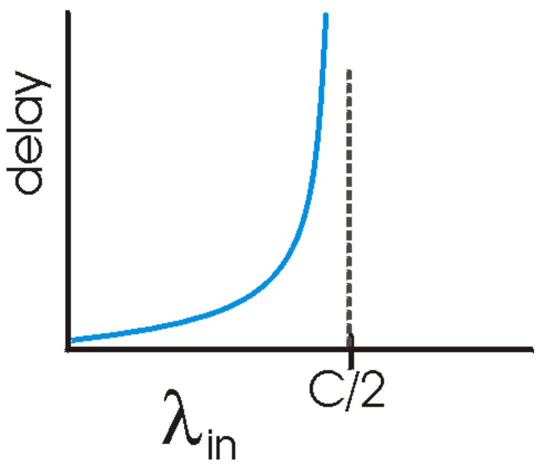
- Congestion is brought about by several factors:
- Slow Hosts resulting in a buildup of queues
- Input traffic rate exceeding the channel capacity also results in a buildup of queues.
- Poor channel conditions (noise, etc.) resulting in an inordinate amount of retransmissions.
- Congestion control techniques limit the queue lengths at the nodes so as to avoid throughput collapse.

Causes/costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- No retransmission



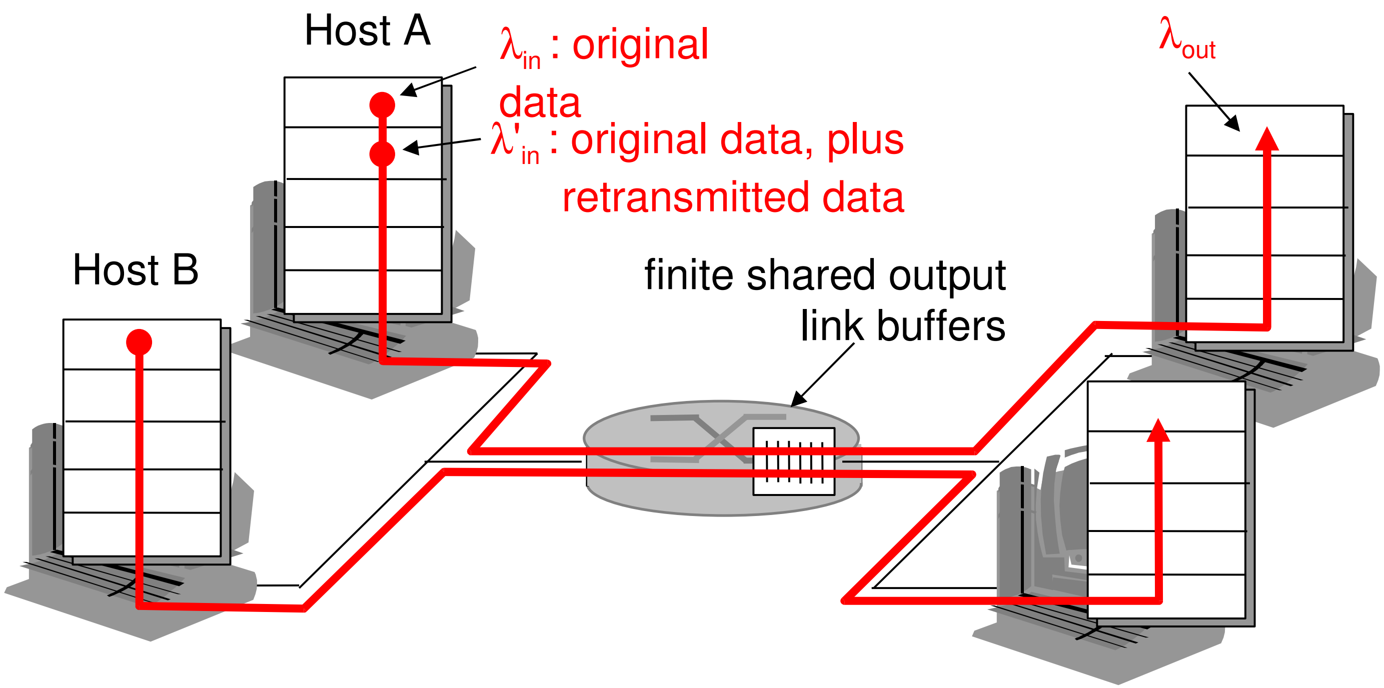




- Large delays when congested
- Maximum achievable throughput

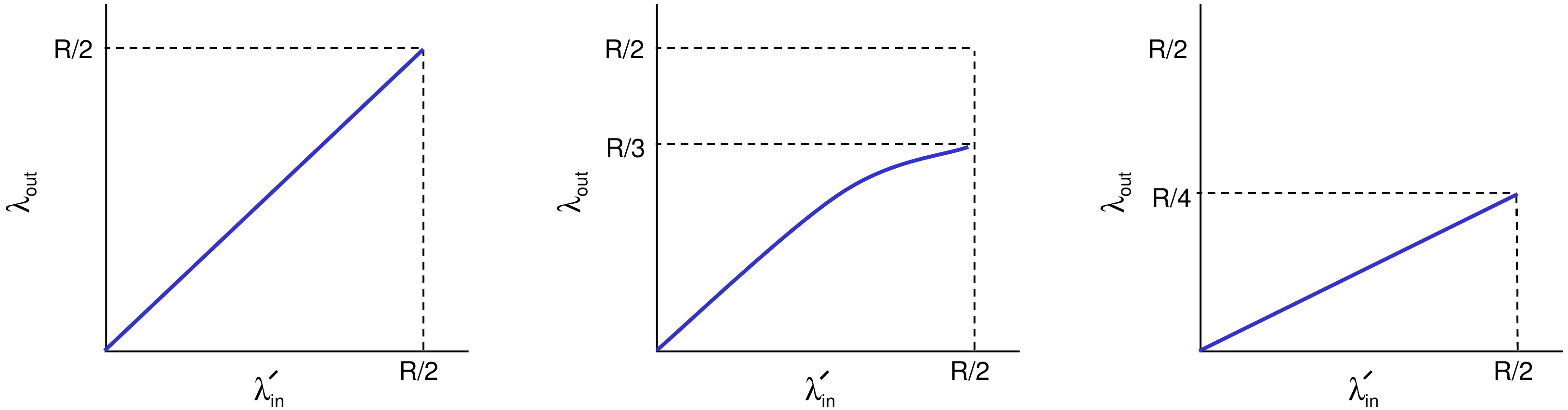
Causes/costs of Congestion: Scenario 2

- One router, *finite* buffers
- Sender retransmission of lost packet



Causes/costs of Congestion: Scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a. b. c.

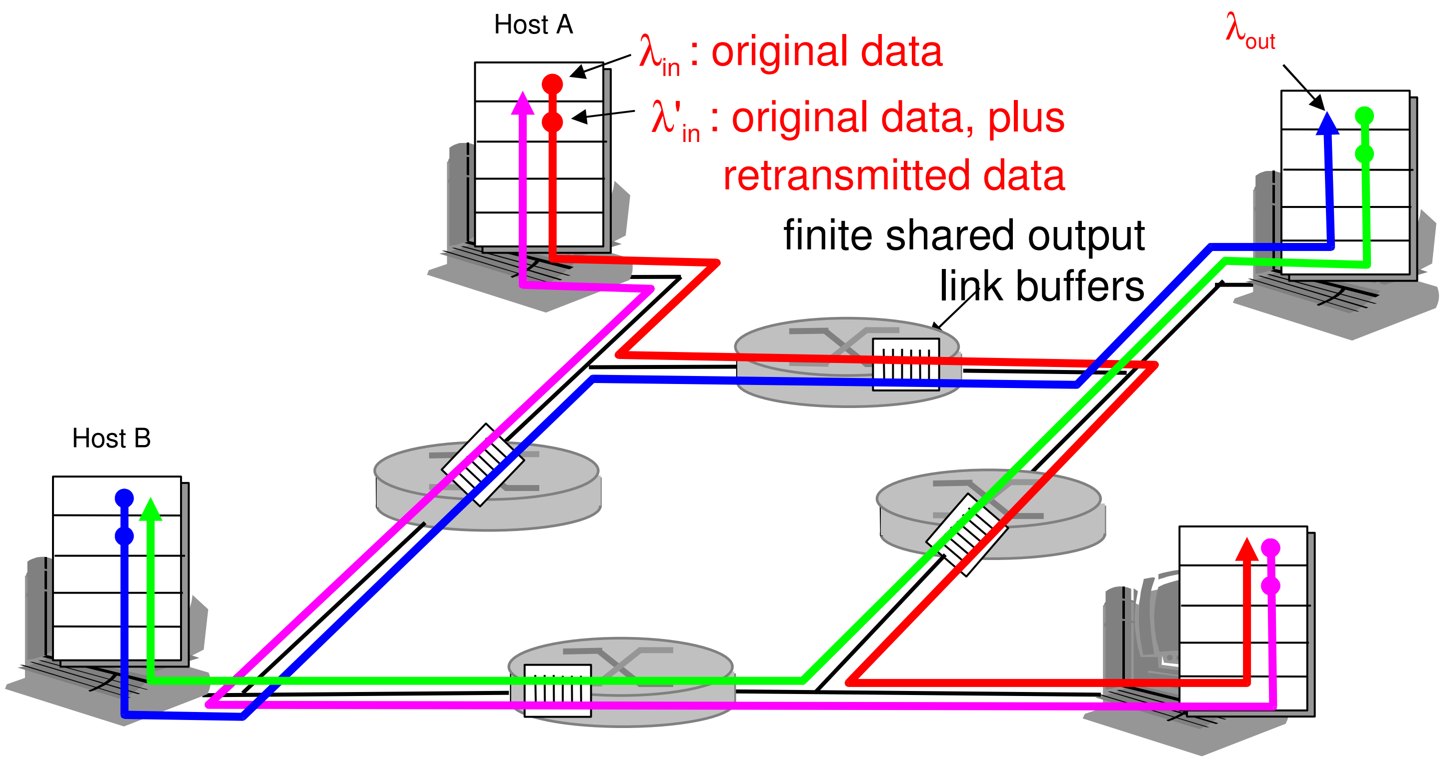
"costs" of congestion:

- More work (retrans) for given "goodput"
- Unneeded retransmissions: link carries multiple copies of pkt

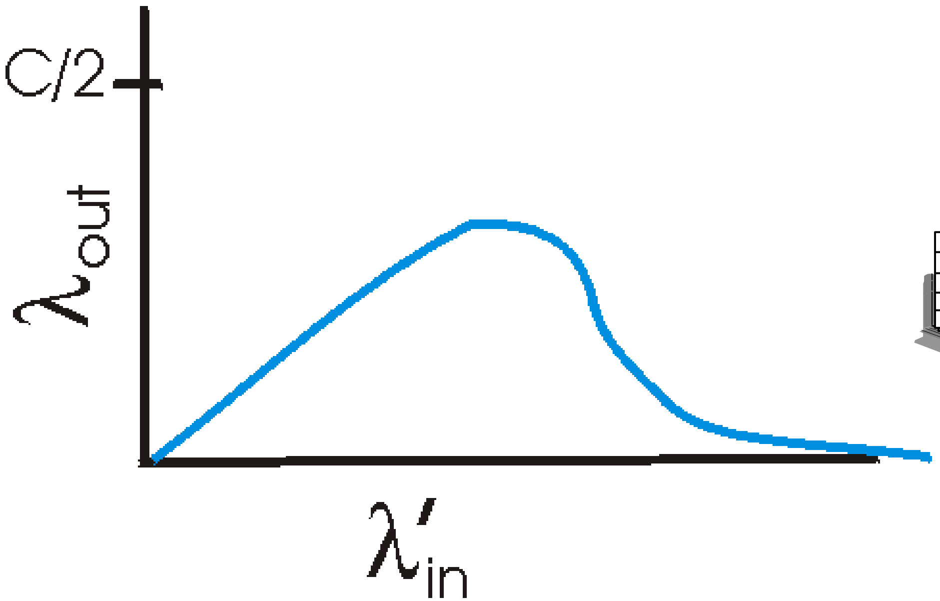
Causes/costs of Congestion: Scenario 3

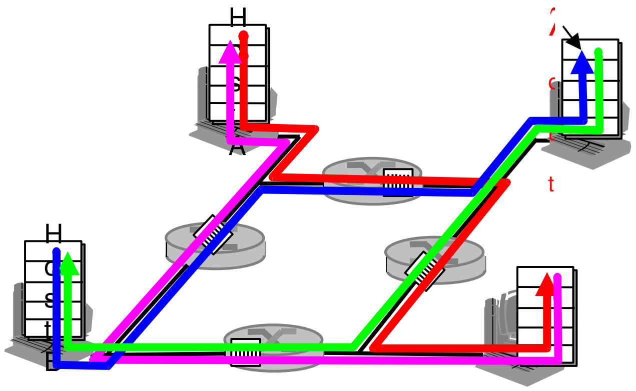
- Four senders
- Multihop paths
- Timeout/retransmit

Q: What happens as λ_{in} and λ'_{in} increases ?



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:

- 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 sender should send at

Case Study: ATM ABR Congestion Control

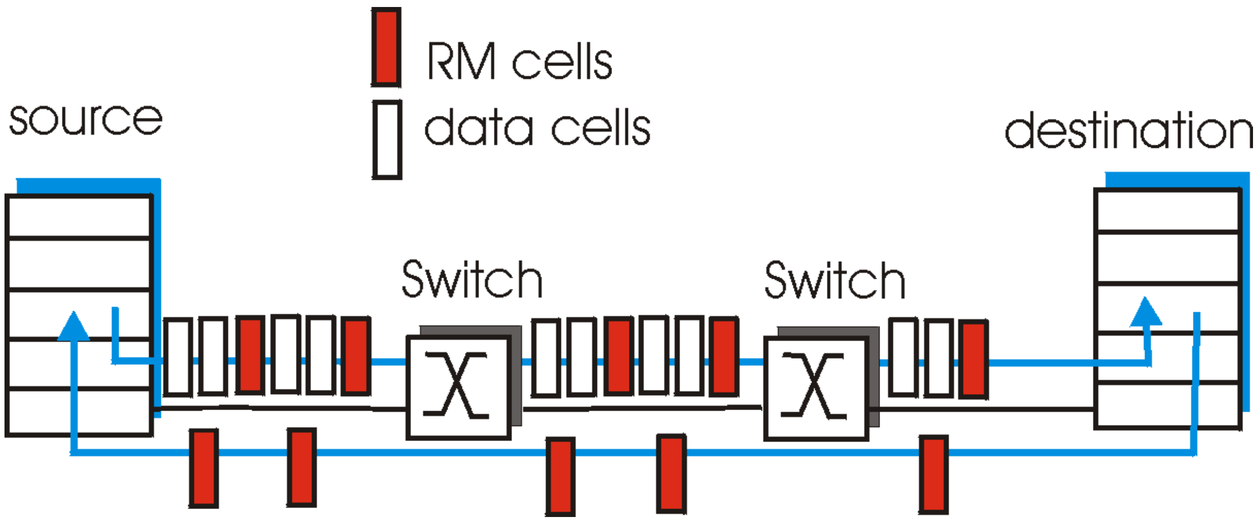
ABR: available bit rate:

- "elastic service"
- If sender's path "underloaded":
  - Sender should use available bandwidth
- If sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (Resource Management) cells:

- Sent by sender, interspersed with data cells
- Bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control

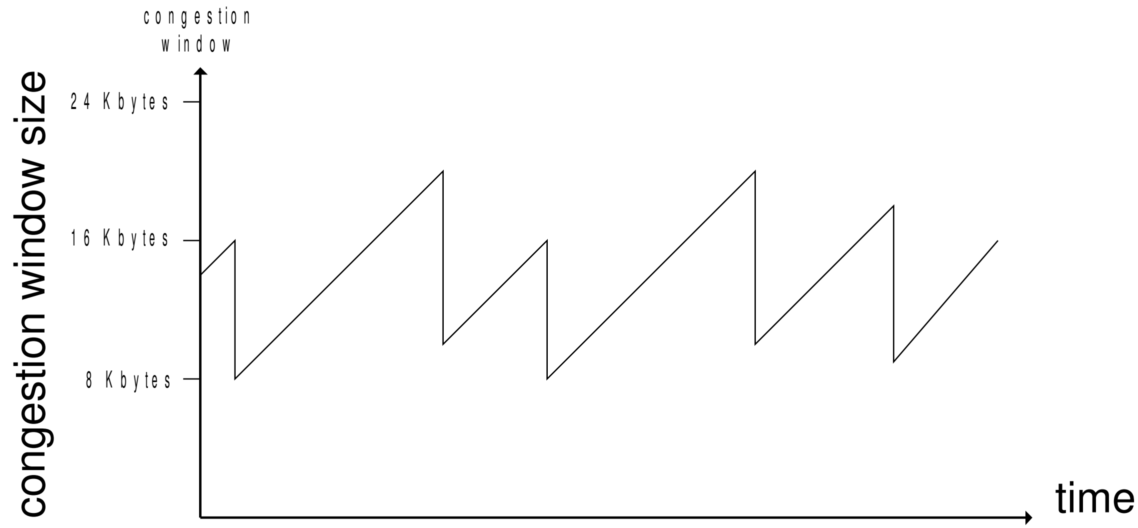


- Two-Byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

TCP Congestion Control: Additive Increase, Multiplicative Decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *Additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *Multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: Details

- Sender limits transmission:

  LastByteSent-LastByteAcked ≤ CongWin

- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** is dynamic; a function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

Three mechanisms:

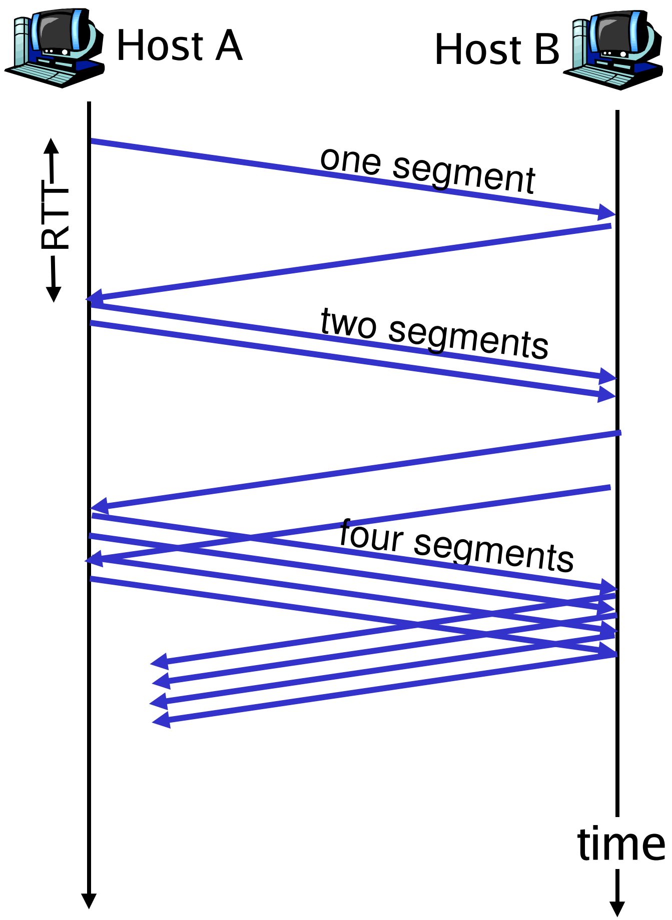
- AIMD
- slow start
- conservative after timeout events

TCP Slow Start

- When connection begins, **CongWin** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- Available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

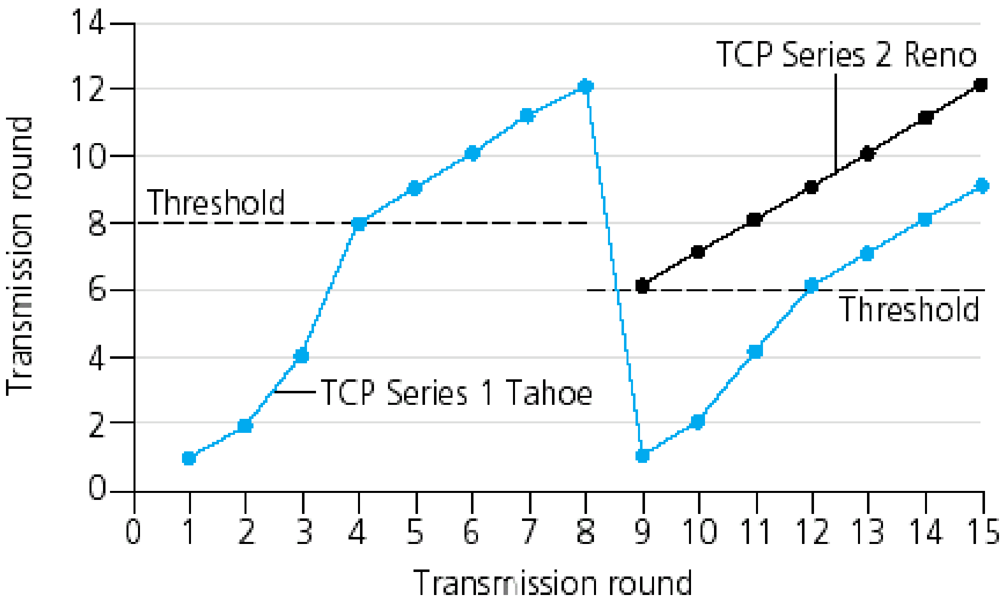
- When connection begins, increase rate exponentially until first loss event:
  - double **CongWin** every RTT
  - done by incrementing **CongWin** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Refinement

Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout.



Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

Refinement: Inferring Loss

- After 3 dup ACKs:
  - **CongWin** is cut in half
  - window then grows linearly
- But after timeout event:
  - **CongWin** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP Throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

TCP Futures: TCP over "long, fat pipes"

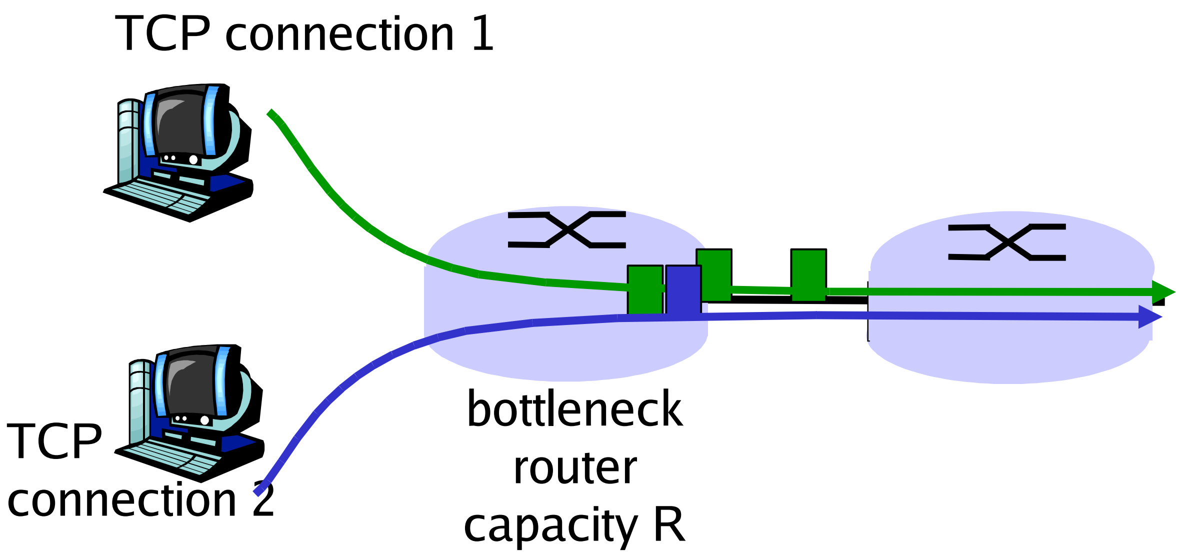
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- $L = 2 \cdot 10^{-10}$
- New versions of TCP for high-speed

TCP Fairness

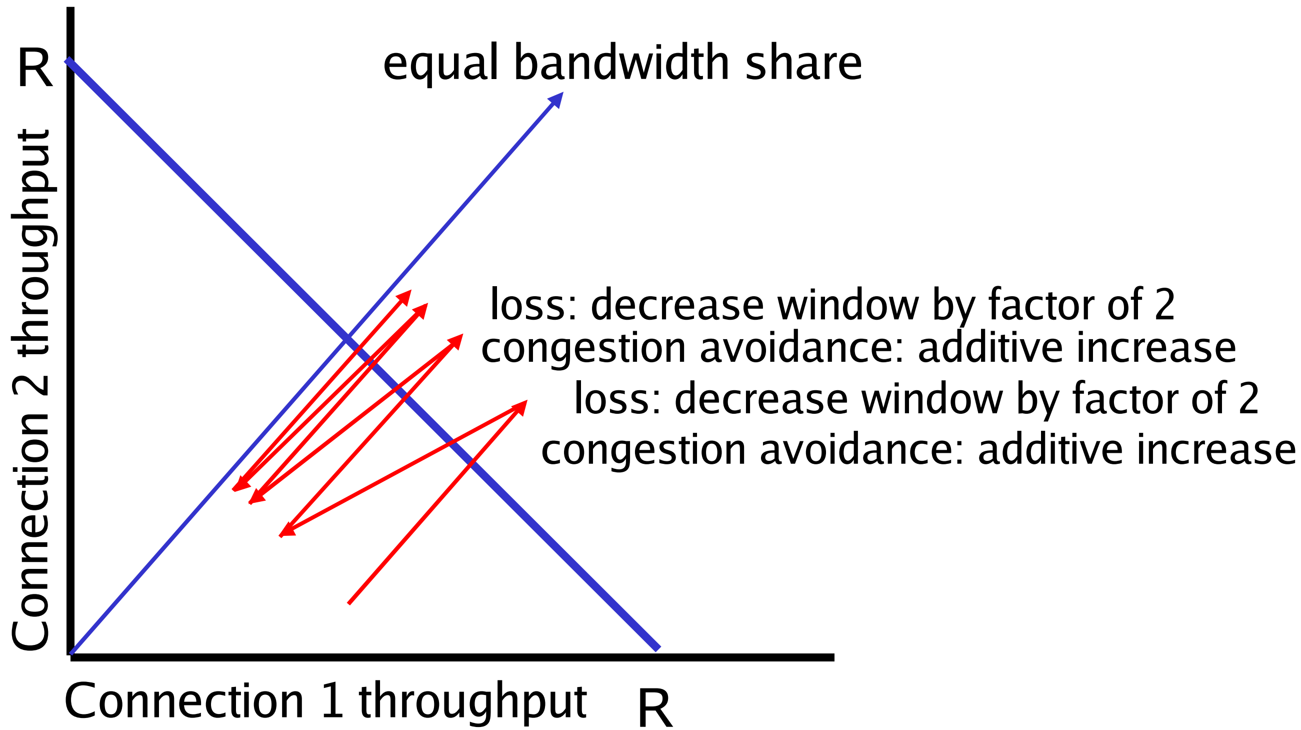
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



Fairness (cont'd)

Fairness and UDP

- Multimedia apps often do not use TCP
  - Do not want rate throttled by congestion control
- Instead use UDP:
  - Pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- Nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - New app asks for 1 TCP, gets rate R/10
  - New app asks for 11 TCPs, gets R/2 !