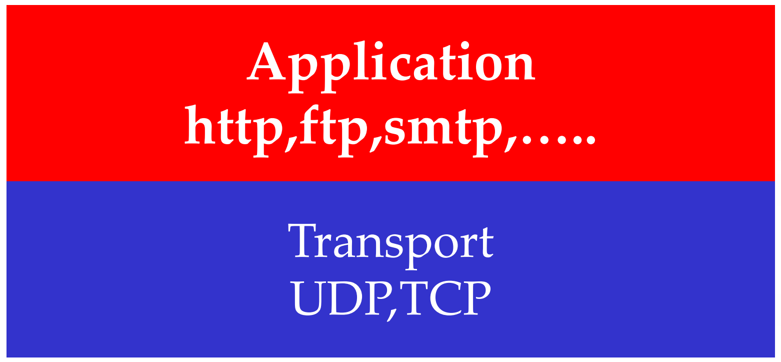
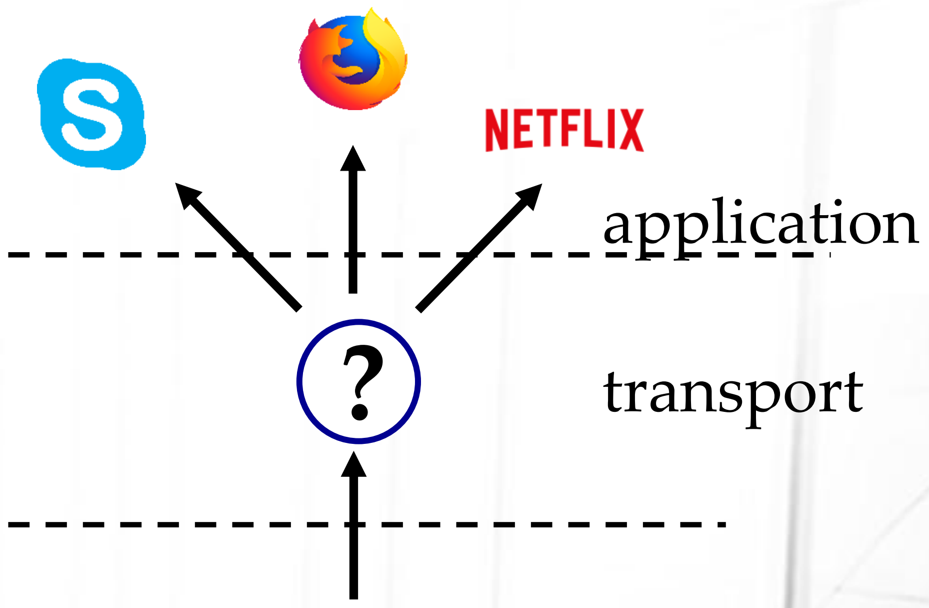
Chapter 3
Transport Layer



Application vs Transport

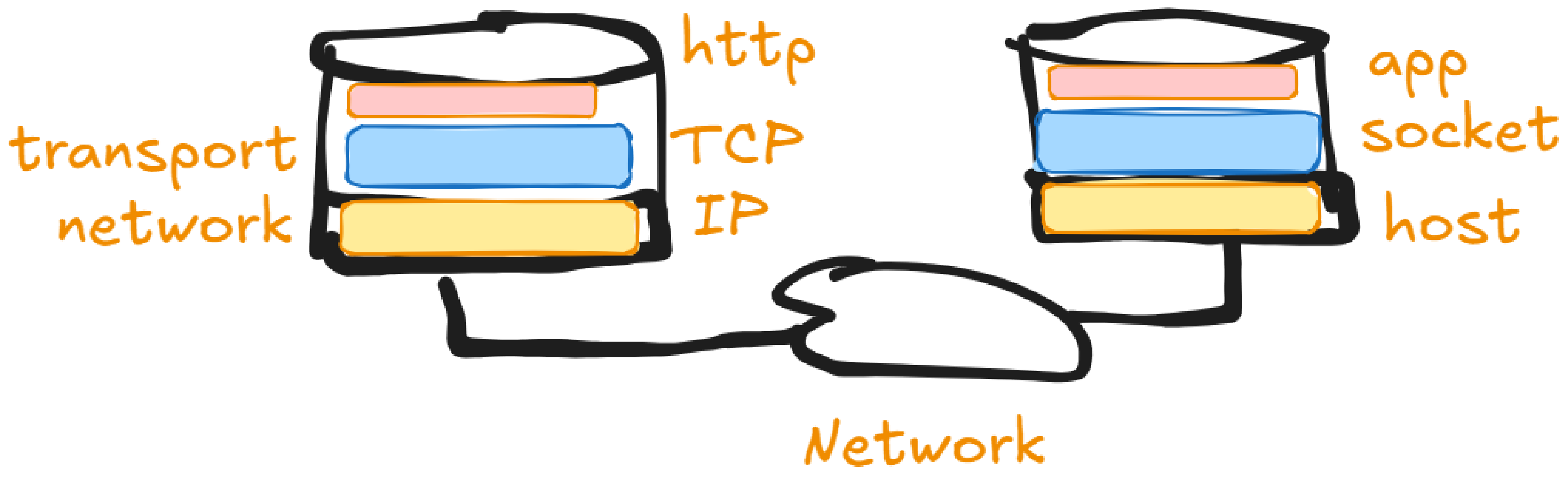
- *Application layer:* end user communication of content (email, webpages)
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services



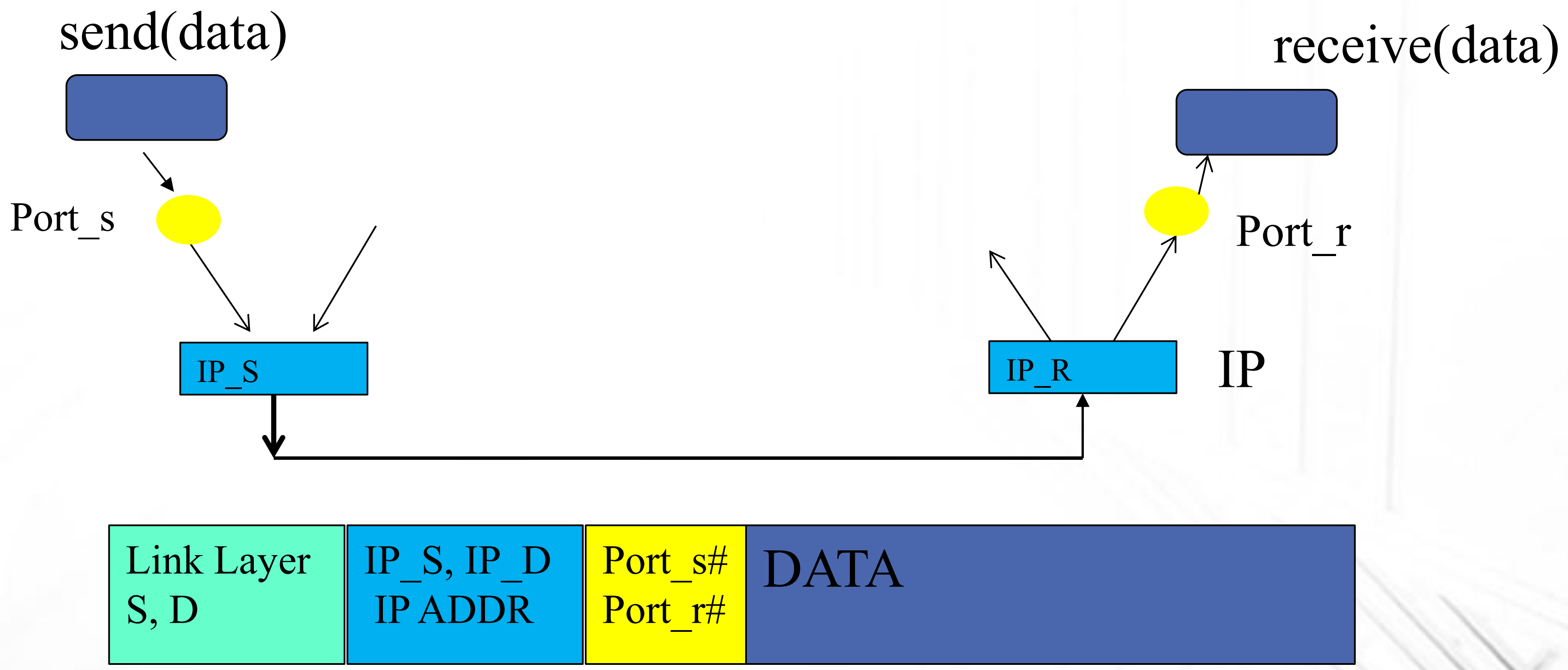
de-multiplexing

Transport vs. network layer

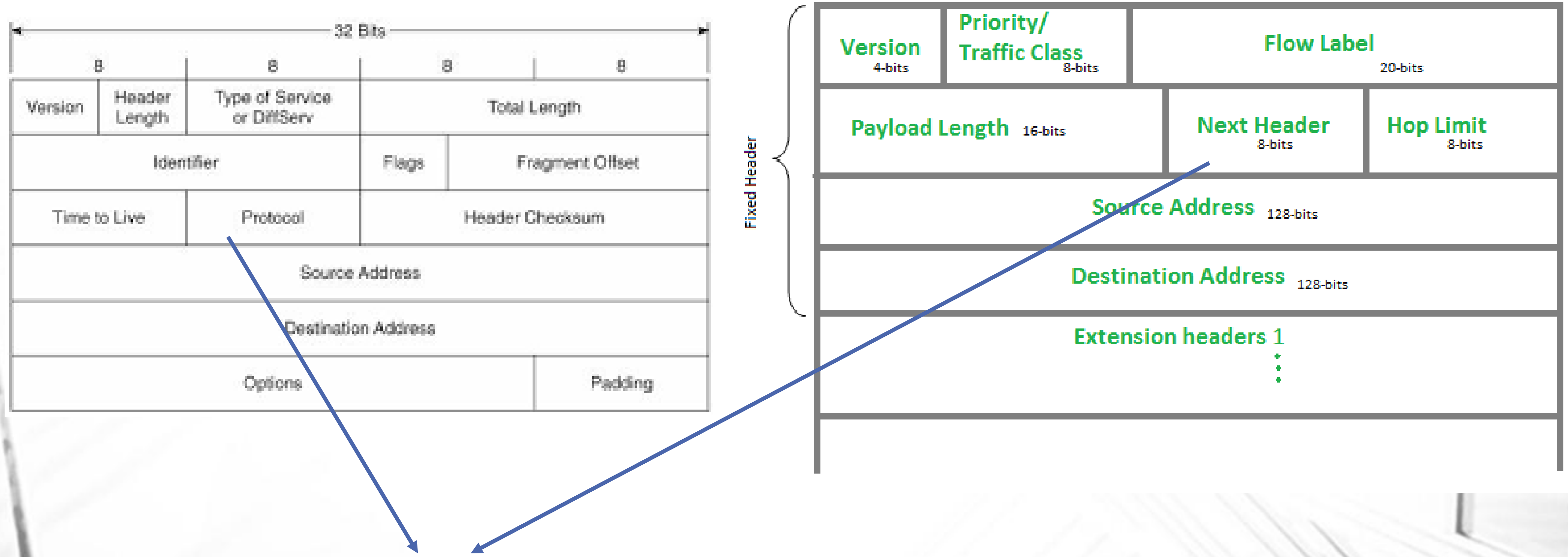
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services
- *network layer:* logical communication between hosts



Layering/demultiplexing

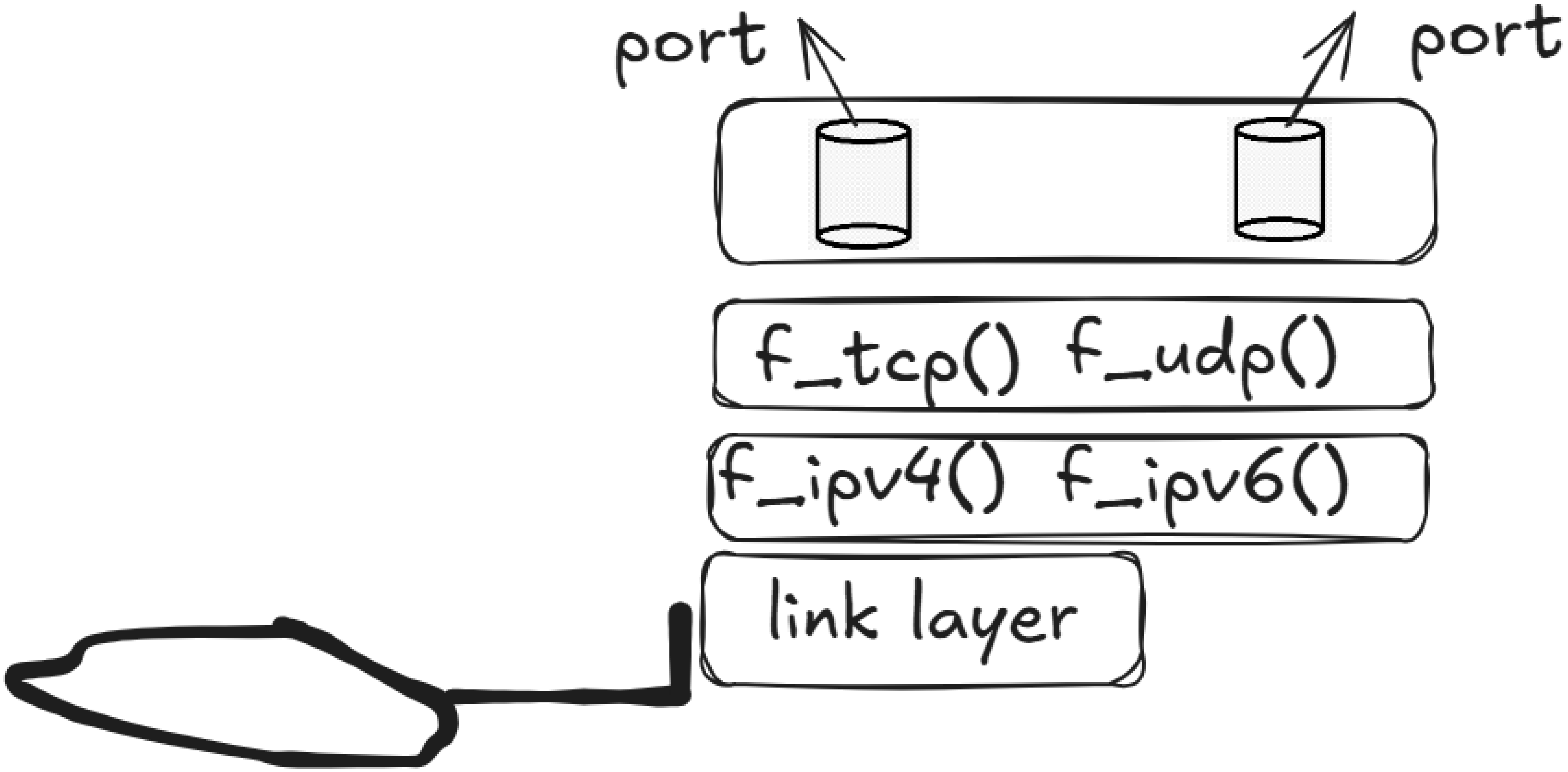


Network Layer Header: IPv4 and IPv6 header



6=TCP
17=UDP

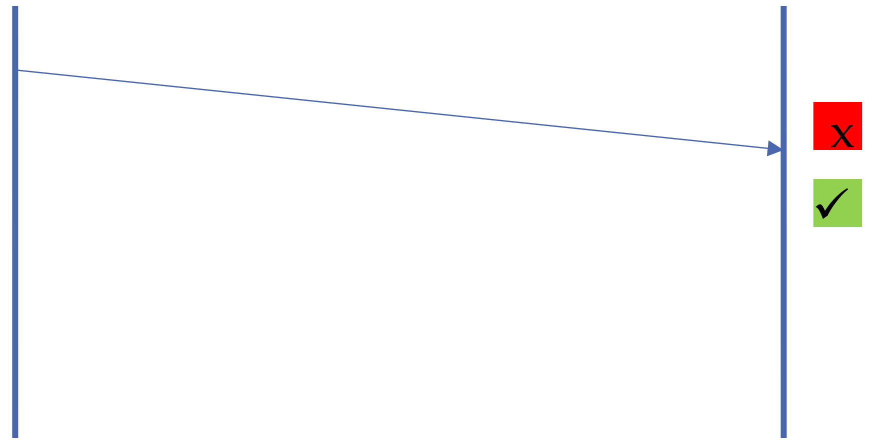
Layering and protocol processing



What can go wrong?

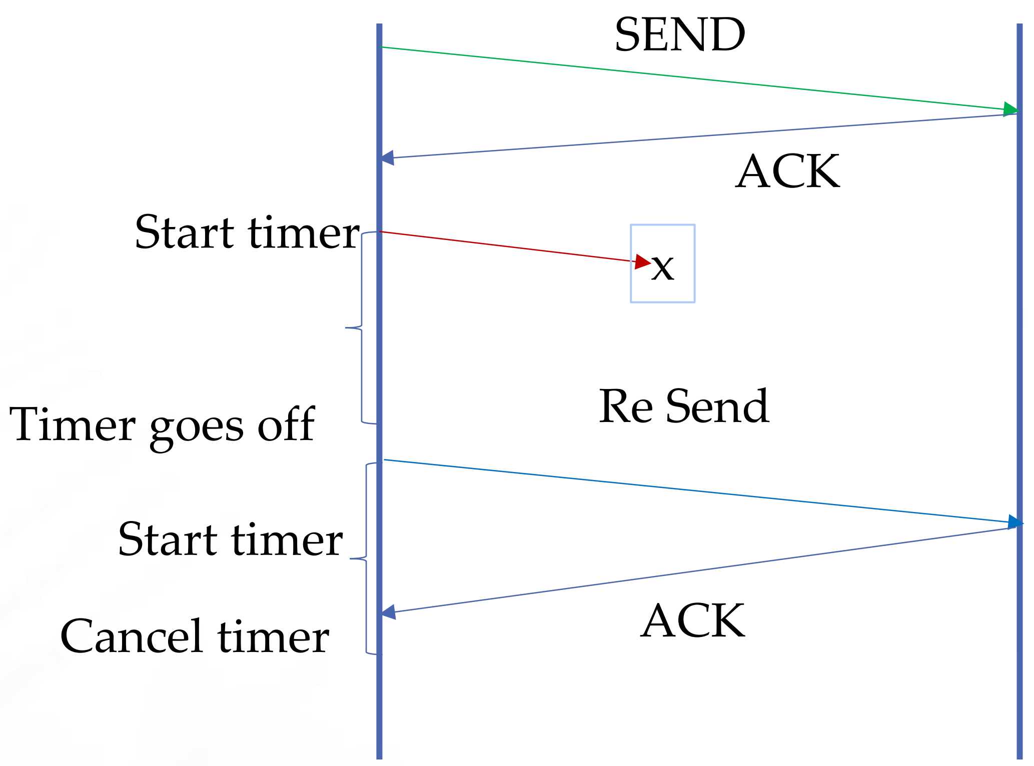
- Packets can be corrupted
  - Read and write of bits can cause bit flips
  - CHECKSUM
- Packets can be lost
  - At the host, or somewhere in the network
- Packets can be duplicated
  - Sender may send same packets

Packet reaches the host but packet is corrupted



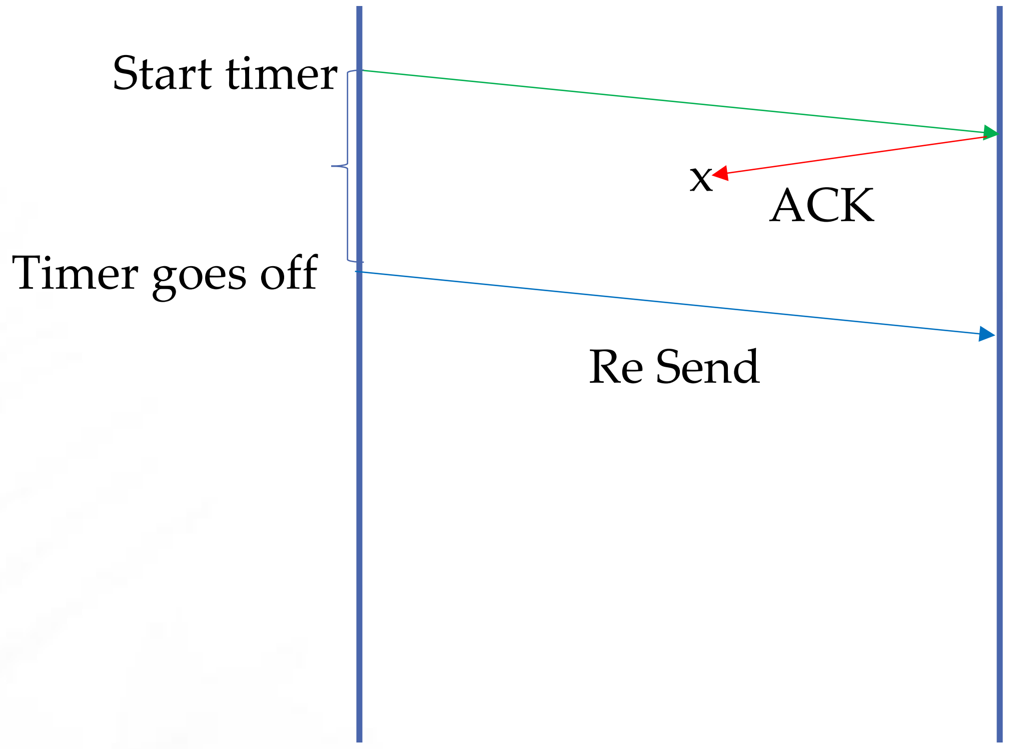
- How to detect if a packet is corrupted?
  - Checksum
- If corrupted, discard, else deliver to process

Lost packet



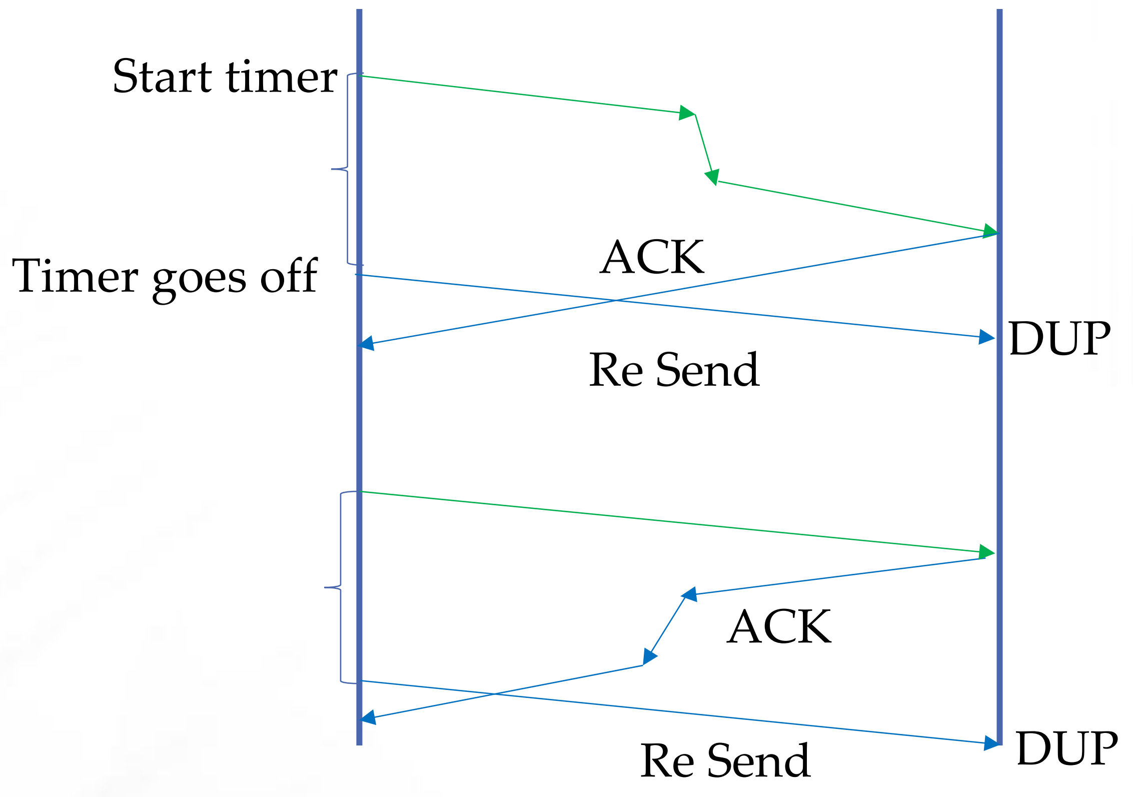
- Use ACKs
- Use timers to know when a packet is possible lost
- What should be the value of the timer? How long to wait?
- Too small?
- Too long?

Lost ack



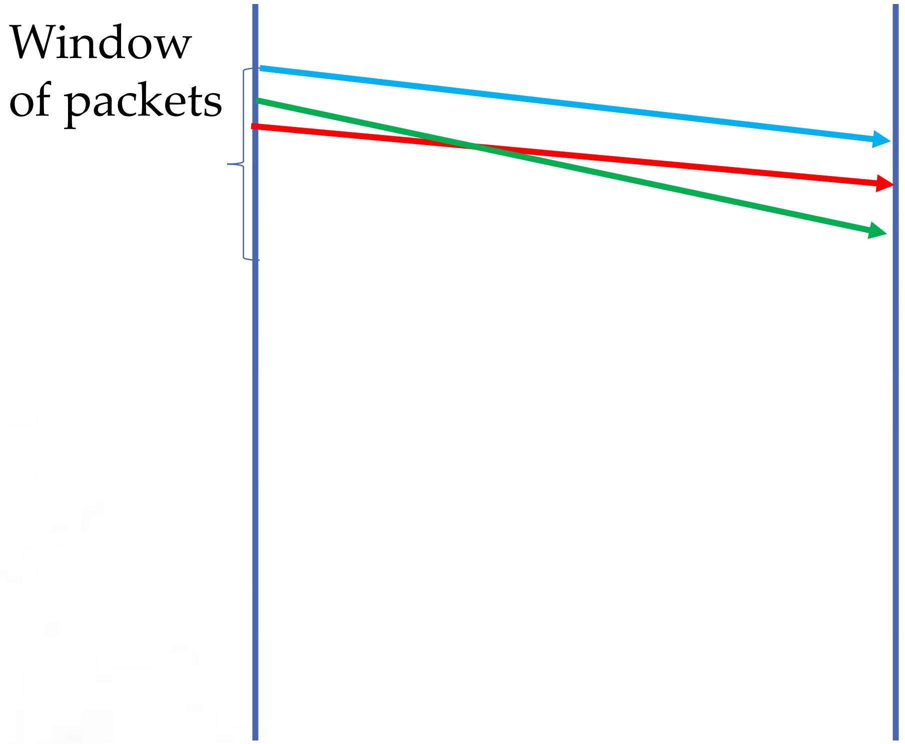
- Duplicate packets
- ACK lost, sender retransmits
- Receiver has duplicate packets
- How to detect and remove duplicates?

Delayed packet



- Delayed packets
- ACK arrives late, sender retransmits
- Receiver has duplicate packets
- How to detect and remove duplicates?

Out of order packets



- Out of order packets
- Earlier sent packet arrives later
- Receiver needs to reorder?
- How? Sequence#

Transport level issues

- Too many packets being sent to the receiver
- Receiver cannot keep up with the sender
- Too many packets being sent into the network
- Network cannot keep up with aggregate traffic
- How to build reliable delivery mechanism between two hosts?

Transport layer protocol: UDP

- User datagram protocol
- Minimal functionality-Best effort
- Provides a mechanism to demultiplex packets (port#)
- Provides checksum(detect corrupt packets)
- For many applications, a simple end to end protocol is good enough
- DNS, control packets (keep alive messages)

UDP: User Datagram Protocol [RFC 768]

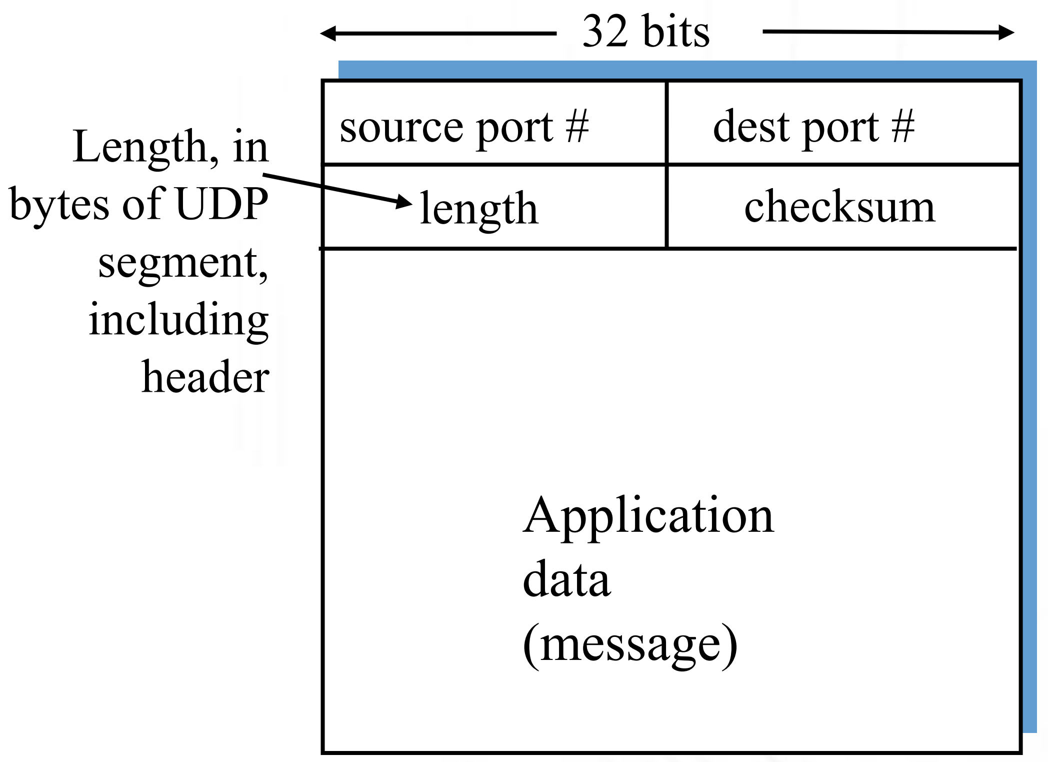
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

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

UDP

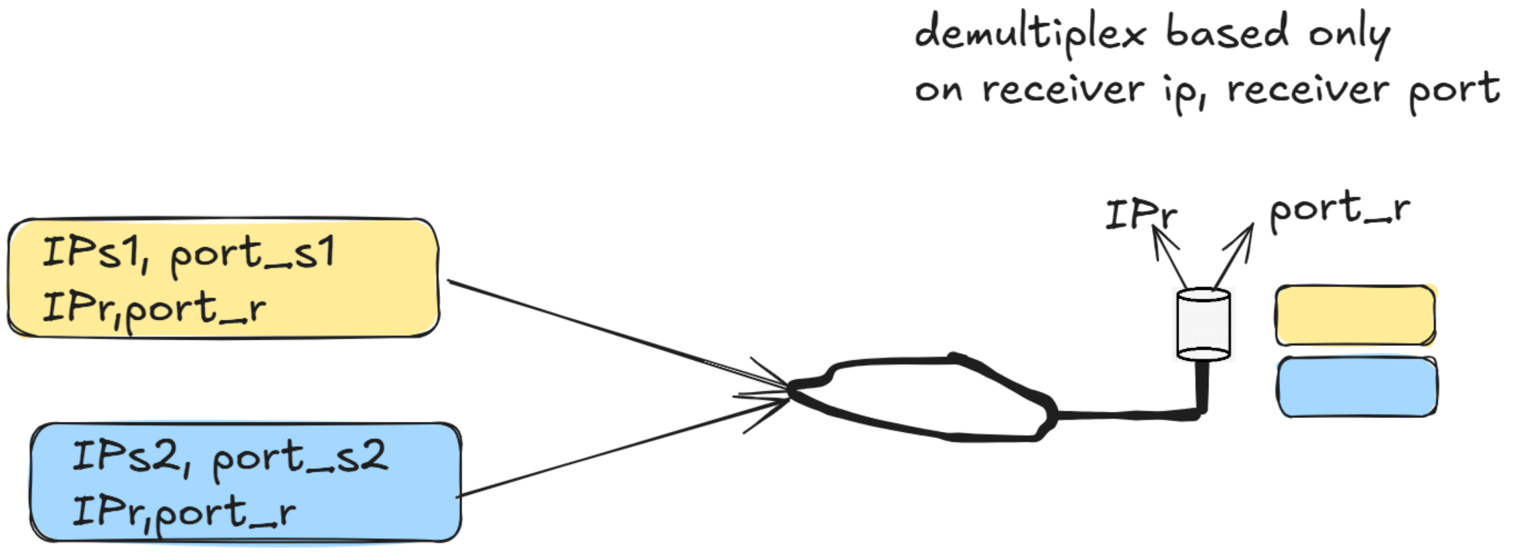
- ❑Apps send one or two packets
- ❑DNS
- ❑Other UDP uses
- ❑Build reliable protocol over UDP
  - QUIC (http/3)
- often used for streaming multimedia apps
  - loss tolerant


32 bits
Length, in bytes of UDP segment, including header
source port #
dest port #
length
checksum
Application data (message)

UDP segment format

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



UDP socket

```
 import socket

Server configuration
SERVER_HOST = "128.6.13.2"
SERVER_PORT = 12352

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Send message to server
message = "Hello, UDP Server!"
sock.sendto(message.encode(), (SERVER_HOST, SERVER_PORT))

# Receive response
data, server = sock.recvfrom(1024)
print(f"Received from server: {data.decode()}")

sock.close()
```

```
import socket

# Server configuration
HOST = "128.6.13.2"   # Remote server
PORT = 12345          # Port to listen on

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((HOST, PORT))

print(f"UDP server up and listening on {HOST}:{PORT}")

while True:
    # Receive message
    data, addr = sock.recvfrom(1024)  # buffer size = 1024 bytes
    print(f"Received message from {addr}: {data.decode()}")

    # Send response back to client
    reply = f"Message '{data.decode()}' received"
    sock.sendto(reply.encode(), addr)
```

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

$$\underbrace{4 \qquad 6}_{\text{receiver-computed checksum}} \neq \underbrace{11}_{\text{sender-computed checksum (as received)}}$$

UDP Checksum Example(cont)

- Complement of the sum is stored in the checksum field
- At the receiver, all the byte fields are added along with the checksum
- Sum whole packet + checksum must be all 1s (0xffff)
  - All 1s, No error else discard packet
- UDP checksum is optional in IPv4
- UPD checksum is mandatory in IPv6

Internet checksum: an example

example: add two 16-bit integers

$$\begin{array}{rcccccccccccccccccc}
 & & 1&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0 \\
 & & 1&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\
\hline
\text{wraparound} & \textcircled{1} & 1&0&1&1&1&0&1&1&1&0&1&1&1&0&1&1 \\
\hline
\text{sum} & & 1&0&1&1&1&0&1&1&1&0&1&1&1&1&0&0 \\
\text{checksum} & & 0&1&0&0&0&1&0&0&0&1&0&0&0&0&1&1
\end{array}$$

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

example: add two 16-bit integers

```
                 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0     0 1
                 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1     1 0
               ---------------------------------
wraparound     1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
               ---------------------------------
sum              1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum         0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

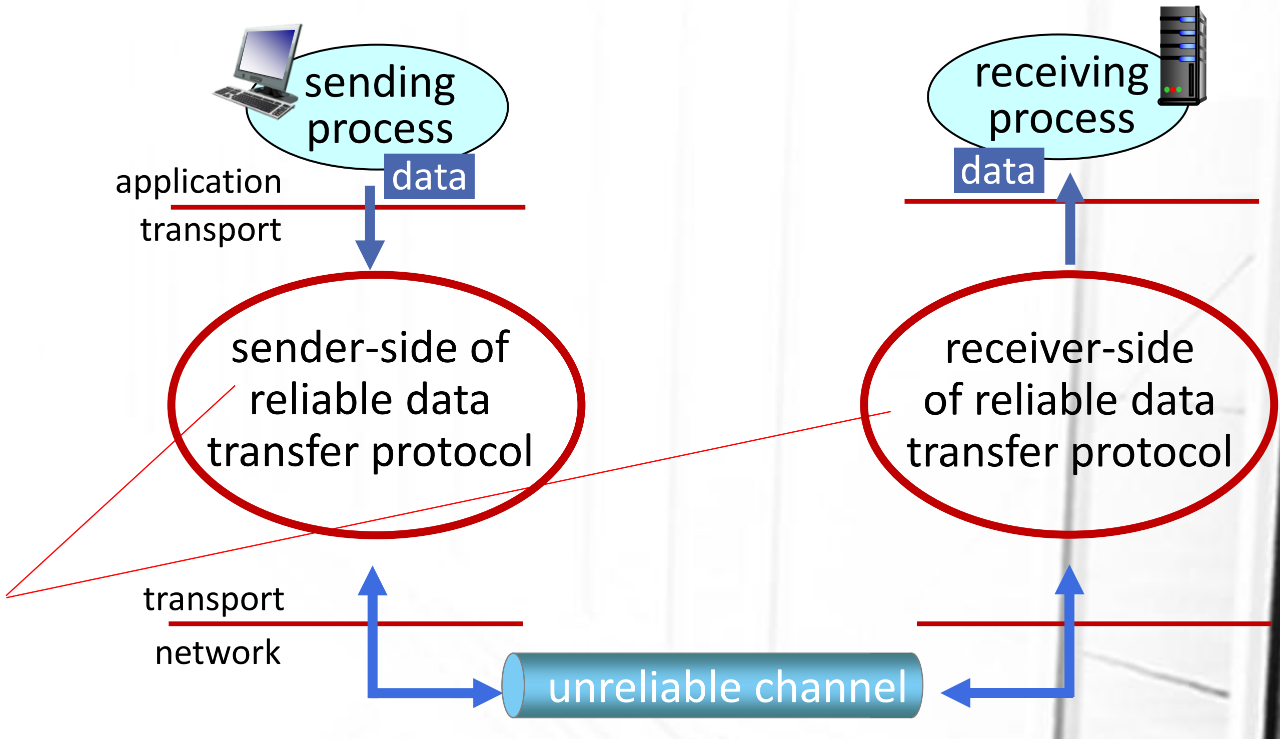
Even though numbers have changed (bit flips), *no* change in checksum!

Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

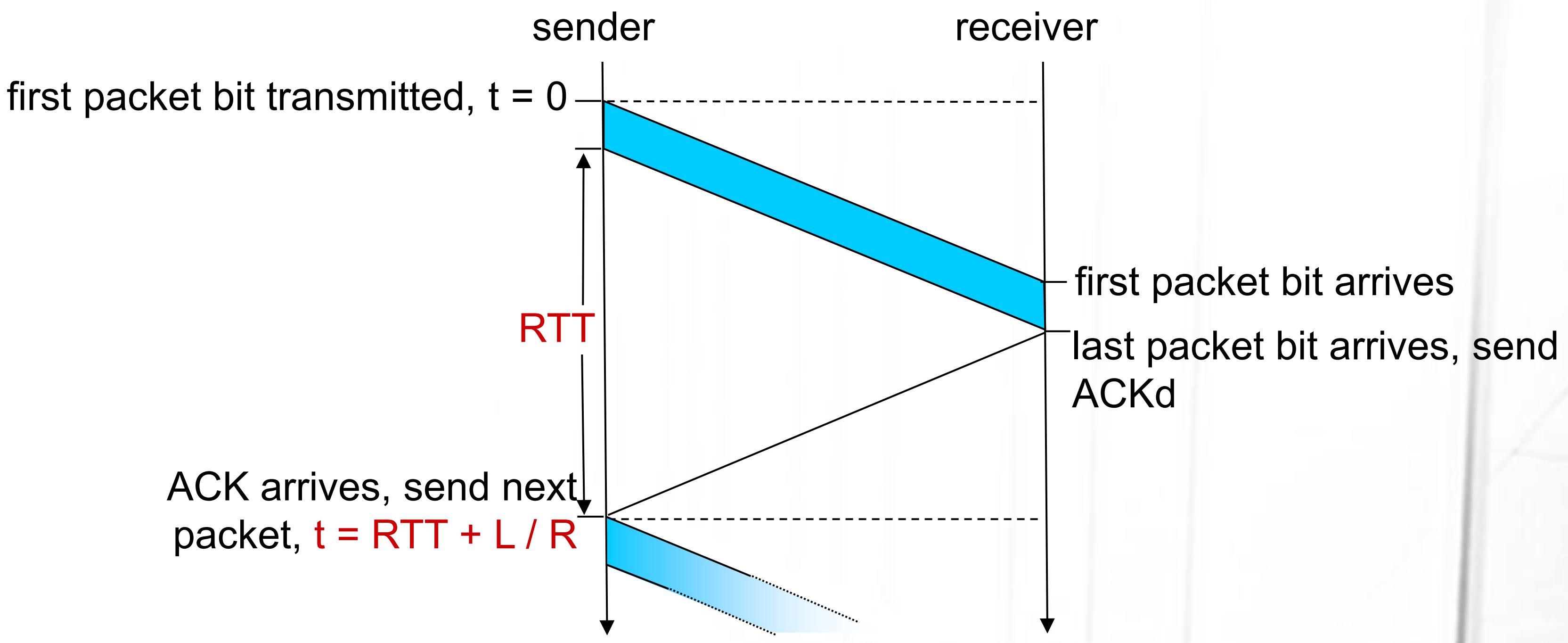
Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)


sending process
data
application
transport
sender-side of reliable data transfer protocol
transport
network
unreliable channel
receiving process
data
receiver-side of reliable data transfer protocol

reliable service *implementation*

stop-and-wait operation


sender
receiver
first packet bit transmitted, t = 0
first packet bit arrives
last packet bit arrives, send ACKd
RTT
ACK arrives, send next packet, t = RTT + L / R

1 Mbytes/sec link, RTT is 2 msec, 1000 Byte packet

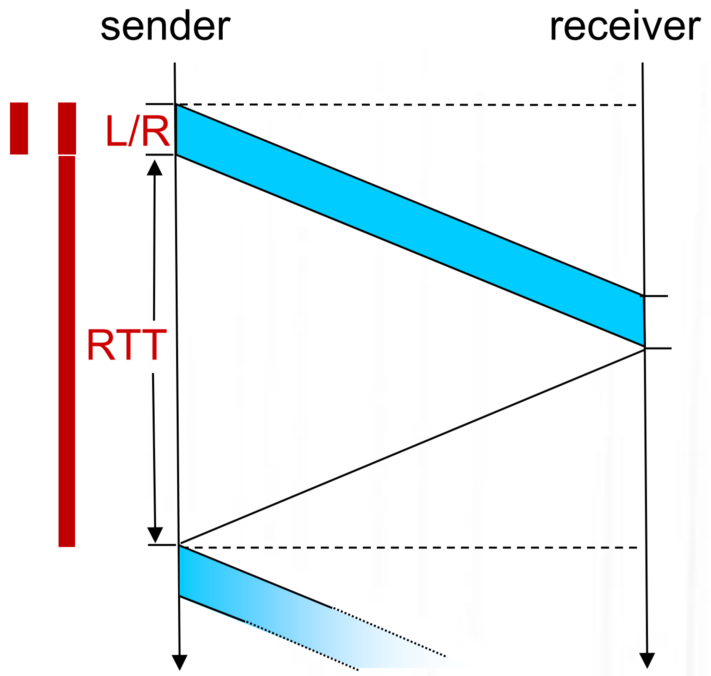
Total time:

stop-and-wait operation

$$U_{sender} = \frac{L / BW}{RTT + L / BW}$$

$$= \frac{1 \text{ msec}}{(2+1) \text{ msec}}$$

$$= 0.3$$

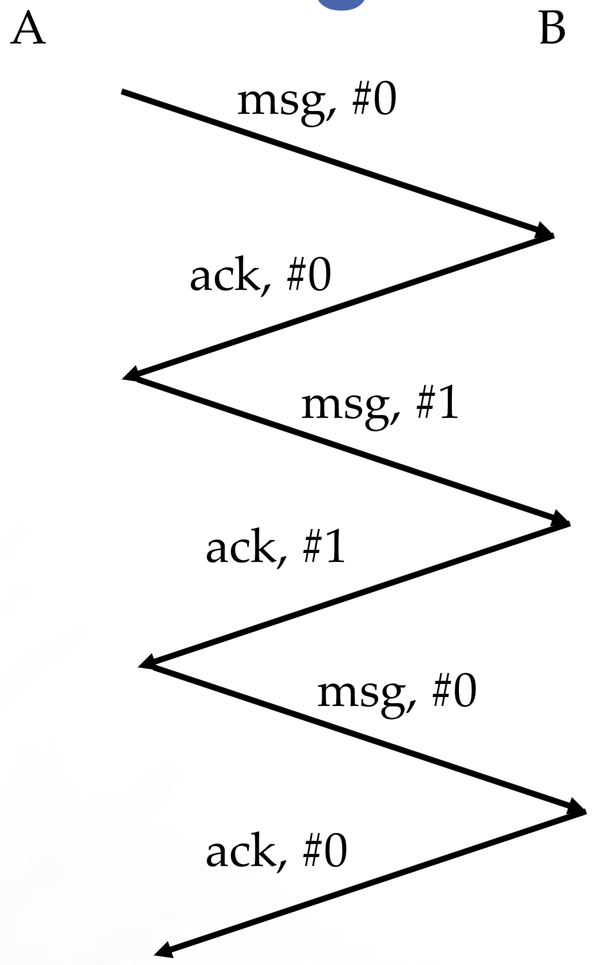


- If BW=10 MB/sec? U_{sender} =?
- Stop & Wait protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

What if ACK is Dropped?

- Receiver might assemble duplicate frames
- Solve problem with sequence number
- How many bits?
- Alternate bit protocol, sender alternates between 0 and 1 sequence #
- Send packet (with seq#=0), set timer, get ack before timer expires, send seq#1, if ack received, send next packet seq#0
- Receiver, on receiving seq#0, ack 0, wait for 1 and so on
- Lost packets, handled by timer
- Duplicate packets handled by sequence number
- Assumption FIFO link

Alternating Bit Protocol

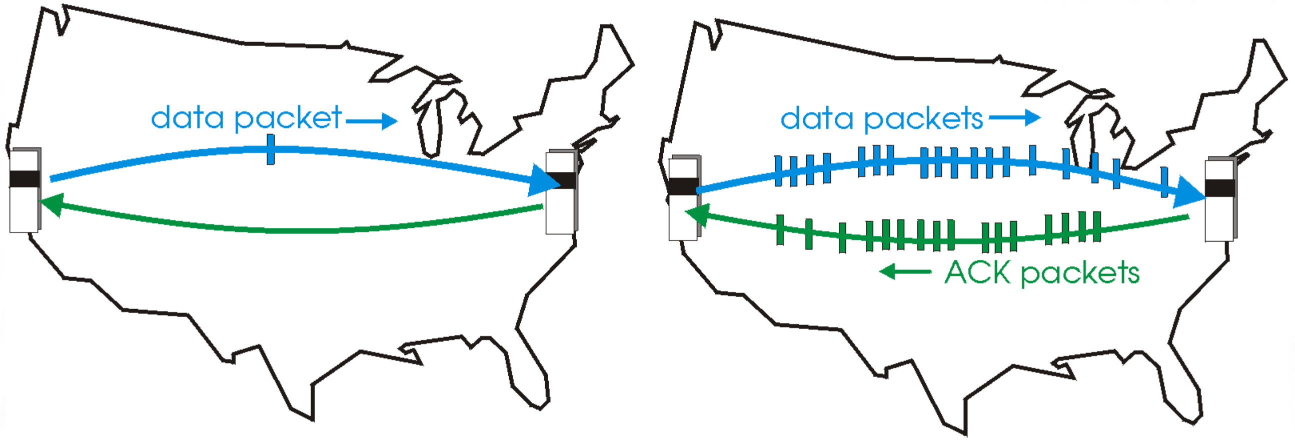


- 1-bit sequence number
- Why is this inefficient?
- Consider 1Mb/s link, 100ms path delay, 1000 byte packets
- Transfer time=8000bits/10^6 b/s =8msec
- 8000bits every 208 msec= 38.4 Kbps
- Send rate is only ~40kb/s << 1Mb/s!

pipelined protocols operation

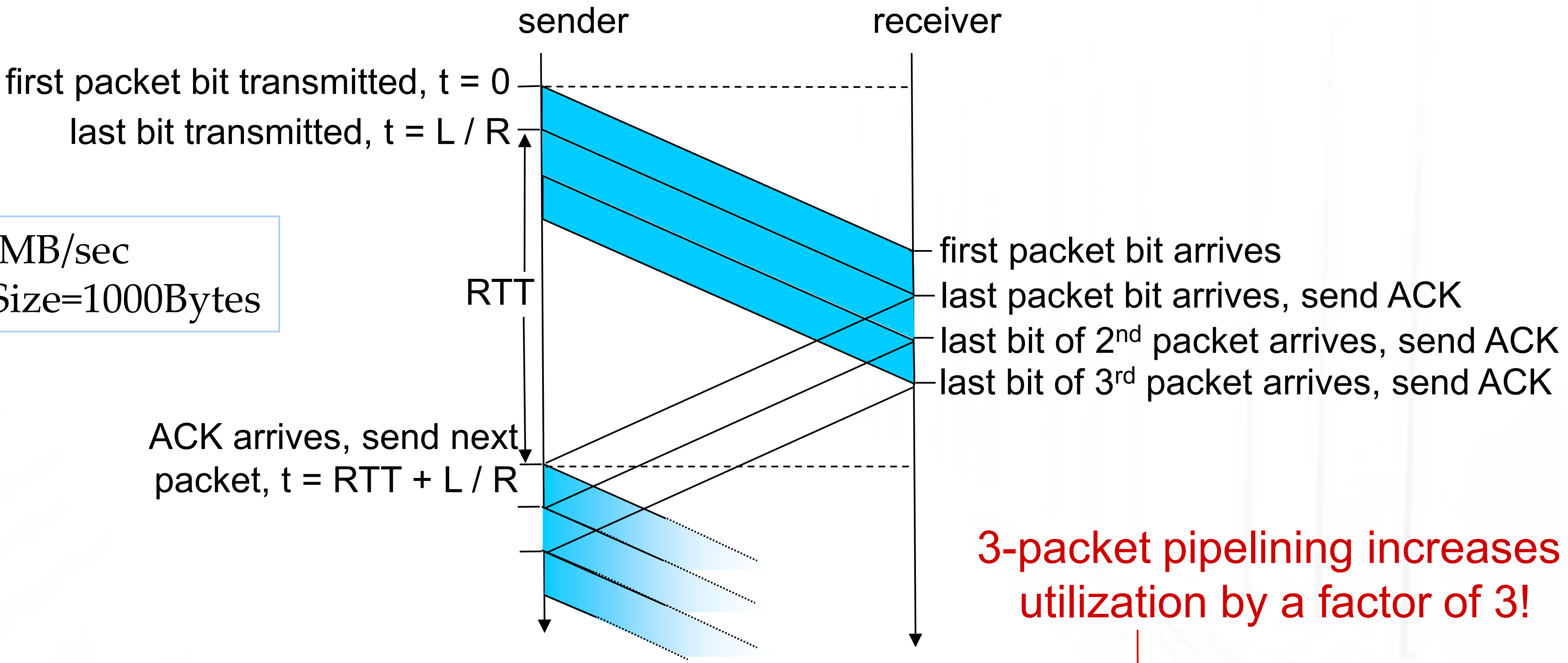
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- 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

Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

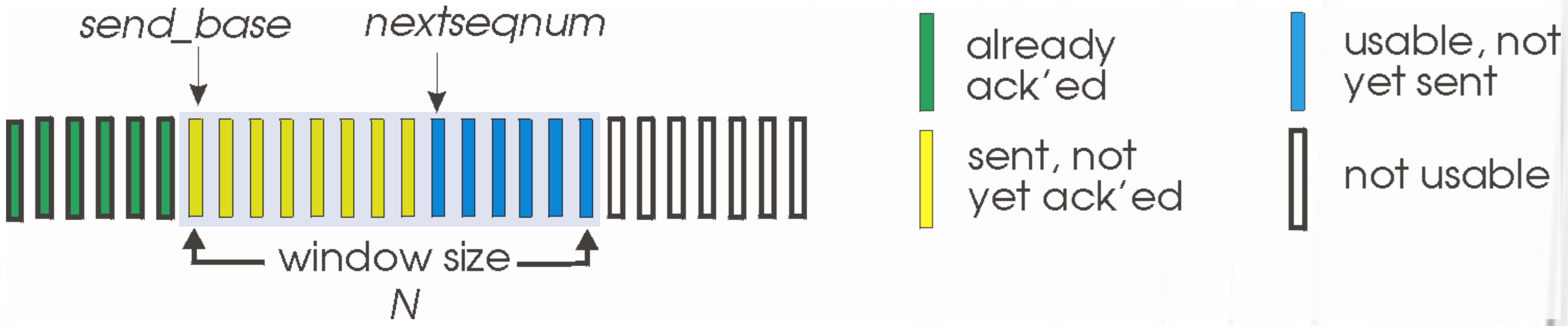
BW=10MB/sec
PacketSize=1000Bytes

3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L/BW}{RTT + L/BW} = \frac{0.3}{2.3} = 0.13$$

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

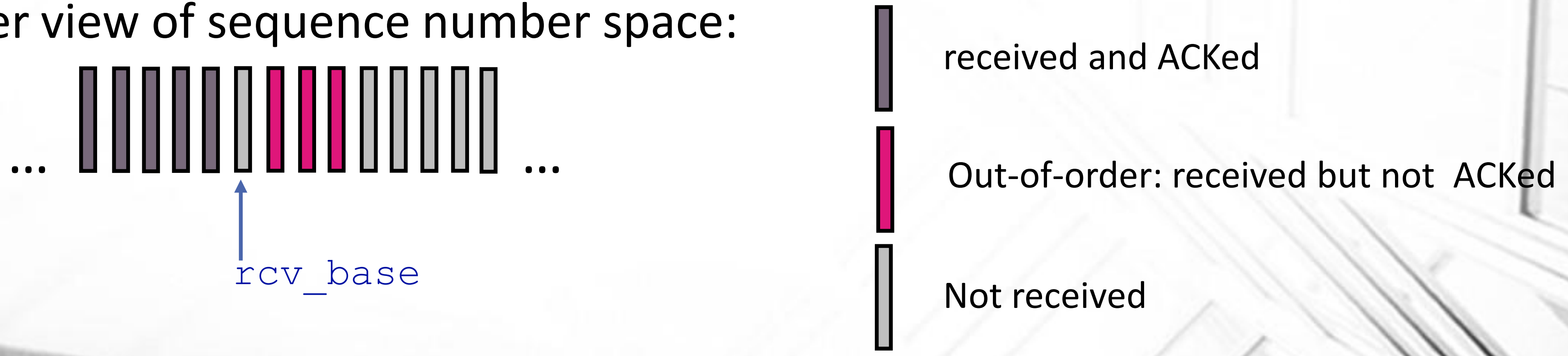


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

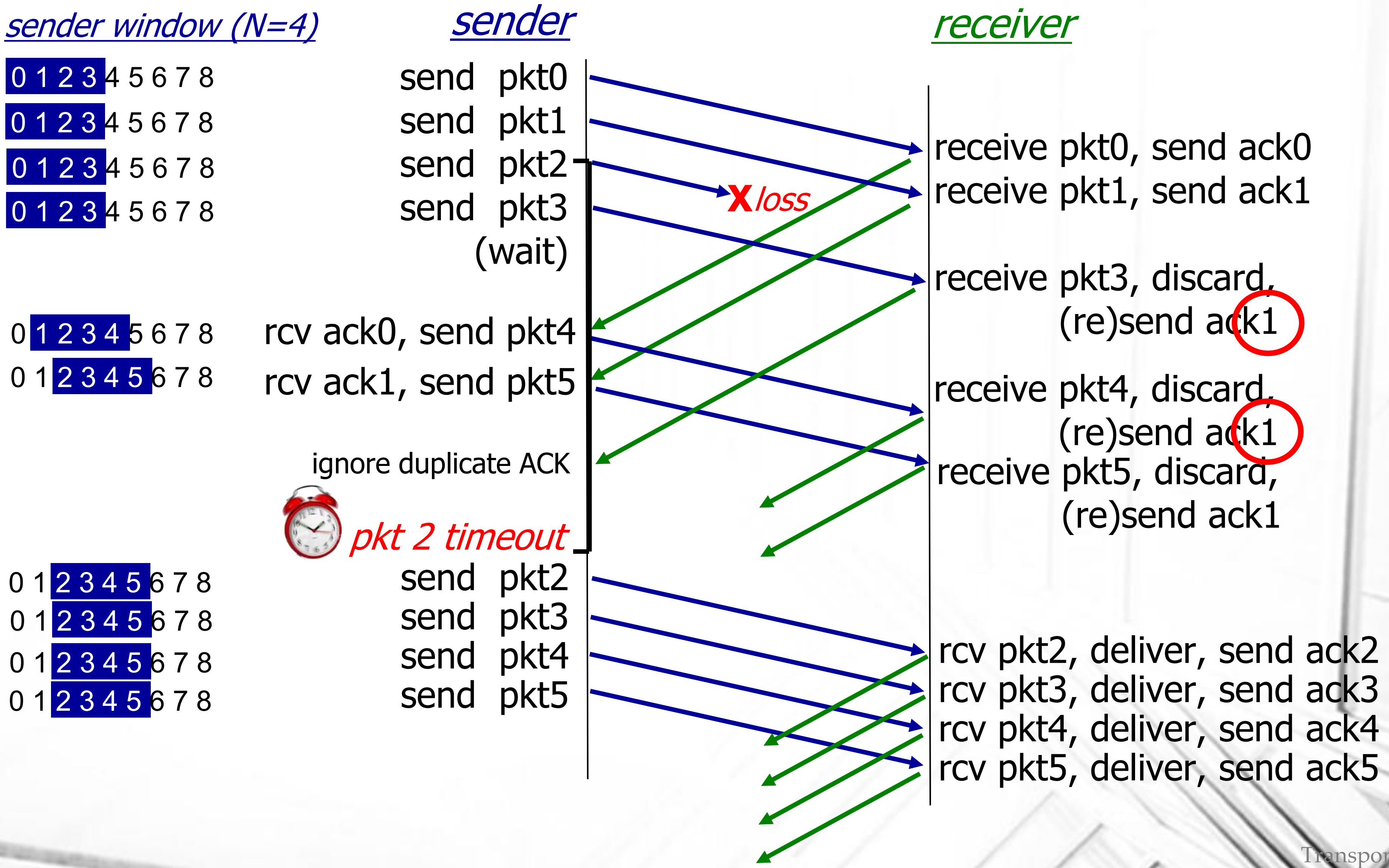
Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



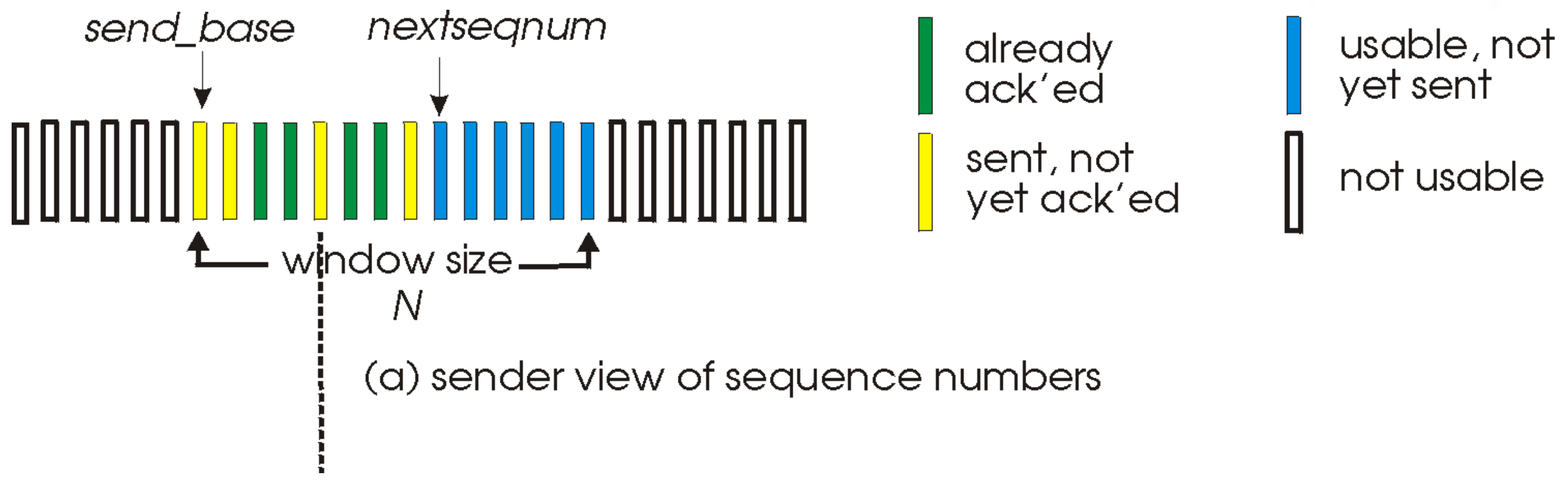
Go-Back-N in action



Selective repeat: the approach

- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) “window” over *N* consecutive seq #s
    - limits pipelined, “in flight” packets to be within this window

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N-1]:

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

receiver

packet *n* in [rcvbase, rcvbase+N-1]

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

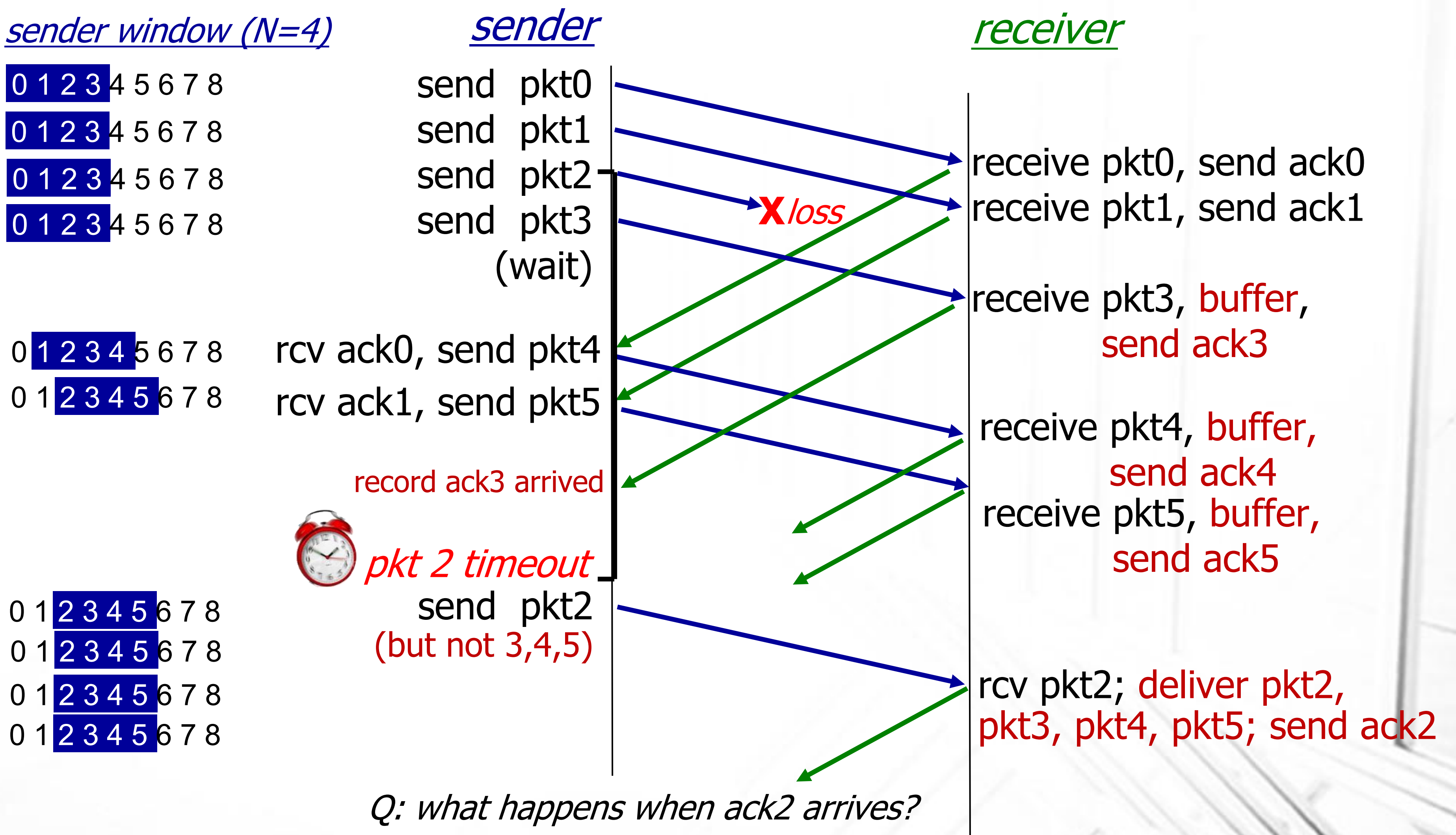
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action

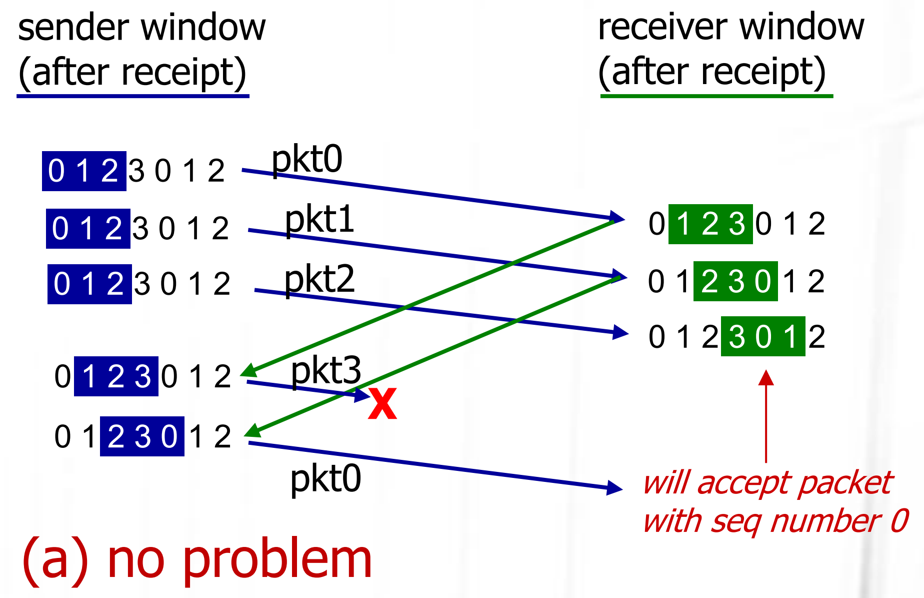


Q: what happens when ack2 arrives?

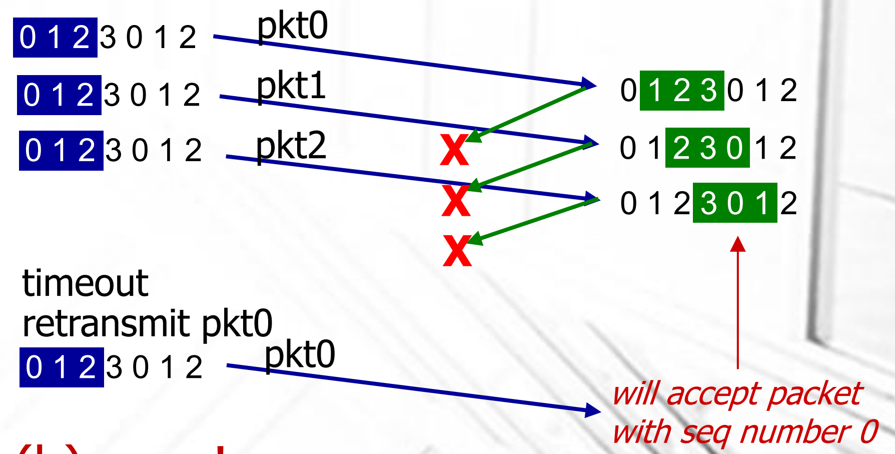
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



(a) no problem

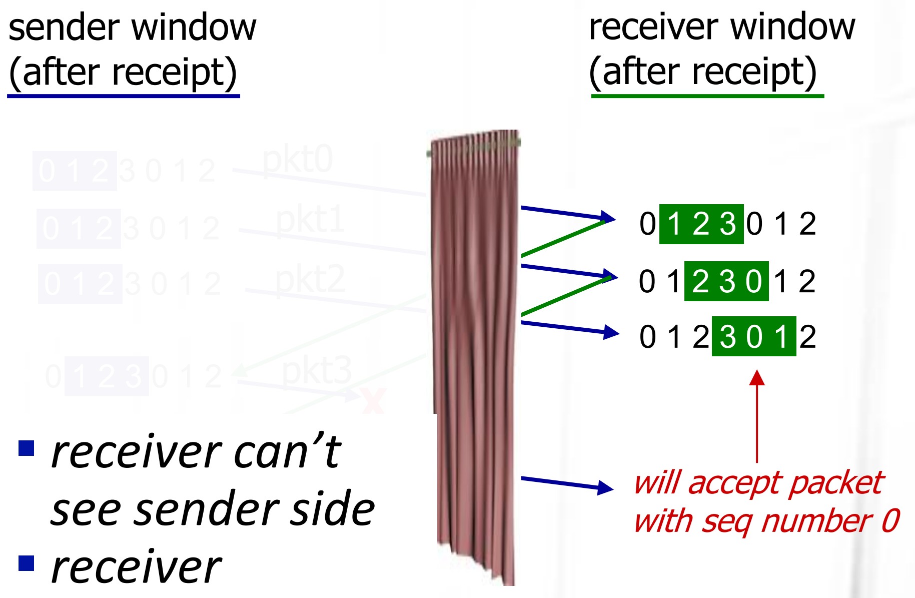


(b) oops!

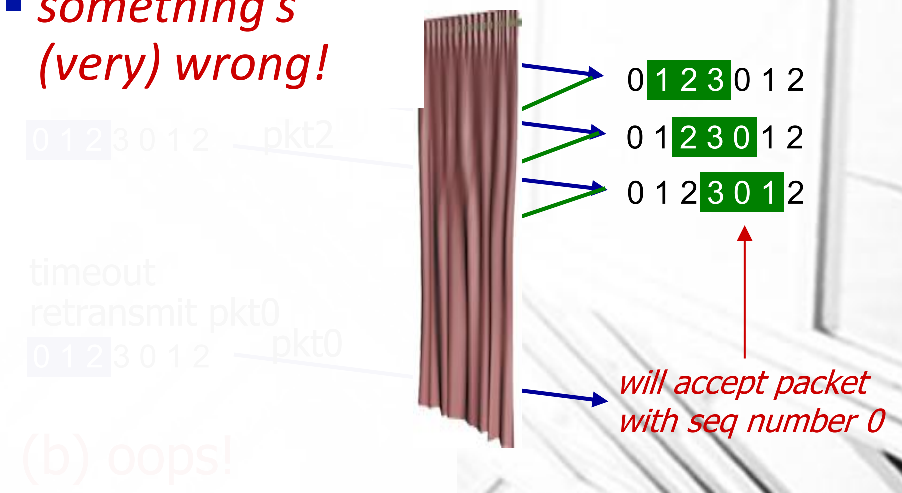
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



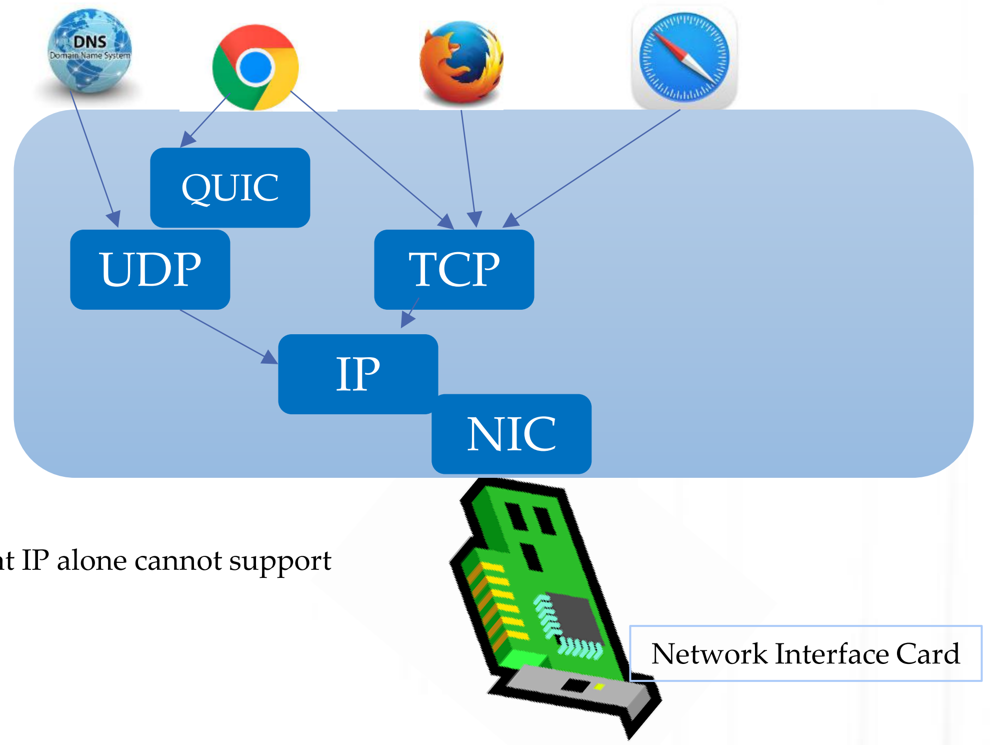
- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*



Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

Chapter 3
Transport Layer

TCP



- TCP provides the end-to-end reliable connection that IP alone cannot support

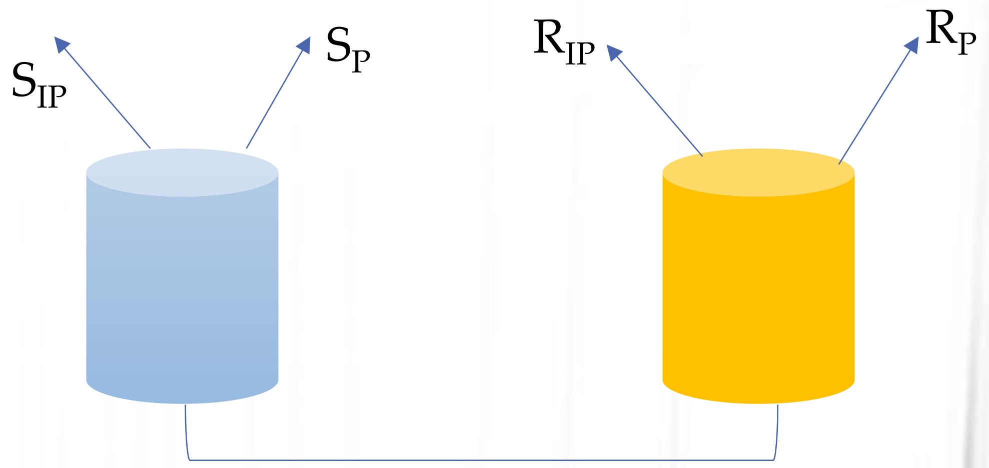
- The protocol
    - Connection management
    - Retransmission
    - Flow control
    - Congestion control
    - Frame format

TCP is connection oriented

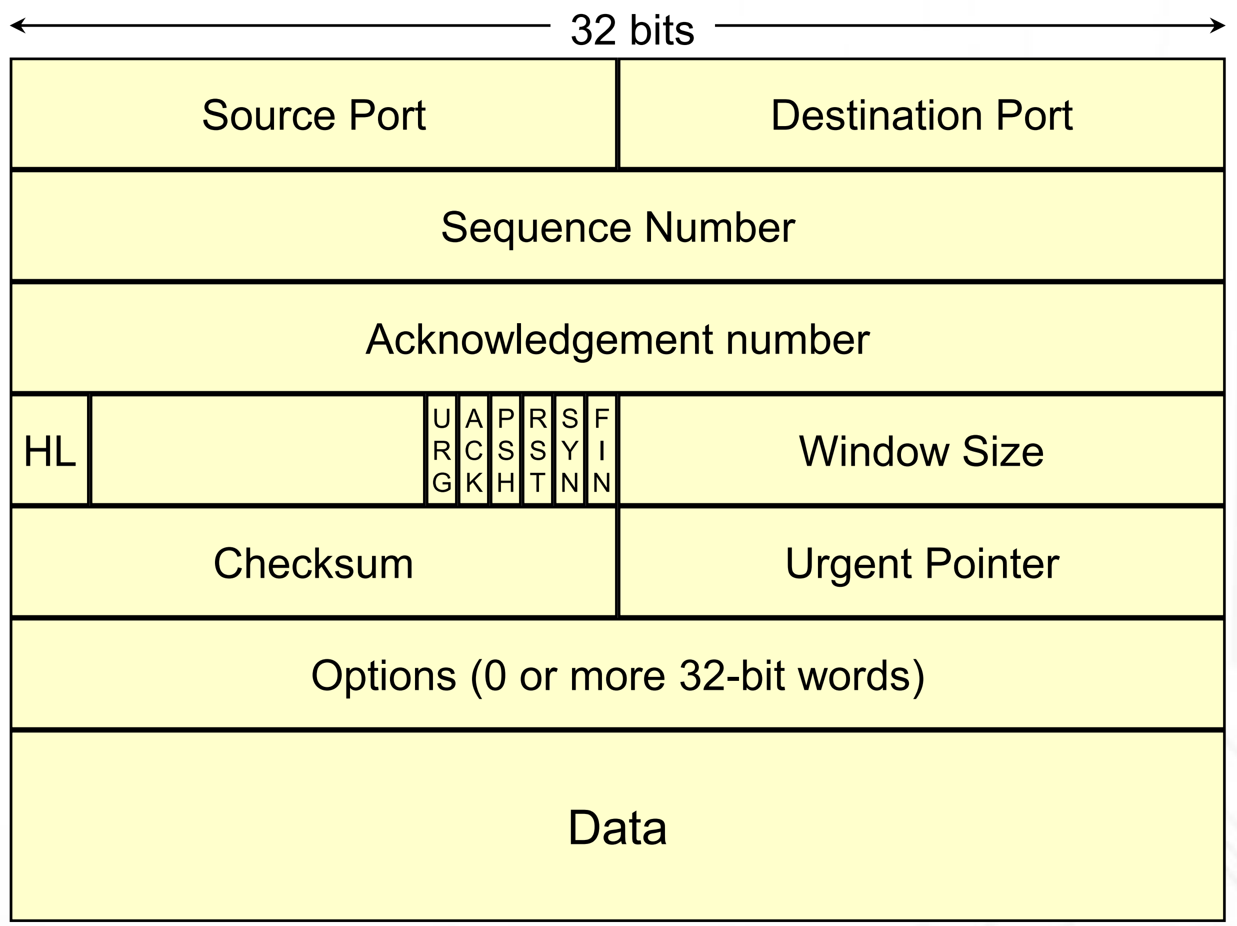
Full duplex

Data sent from sender to receiver and at the same time from receiver to sender

- A TCP socket is bound to both sender and receiver



TCP Header Format

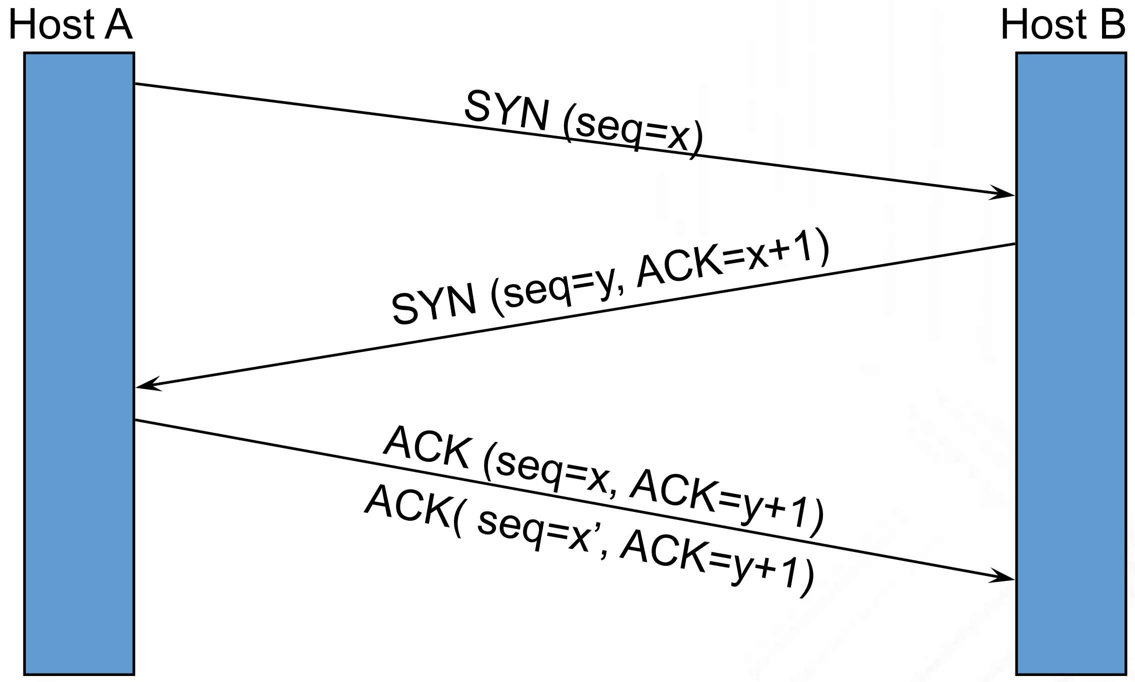


TCP Header Fields

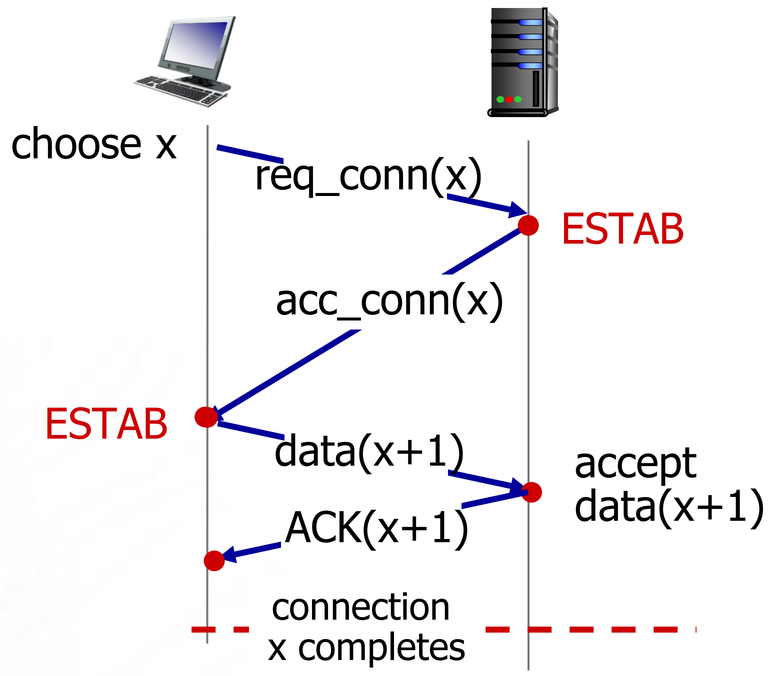
- Source & Destination Ports
  - 16 bit port identifiers for each packet
- Sequence number
  - The packet's unique sequence ID
  - Sequence number is the number of the first byte in the packet + ISN
  - ISN=K ; byte 10 to 1000 is sent; Seq no=K+10
  - Next packet is 1001 to 2000 ; seq no=K+1001
- Acknowledgement number
  - The sequence number of the next expected byte at the receiver

TCP Connection Establishment

- Three-way Handshake
- Sequence numbers



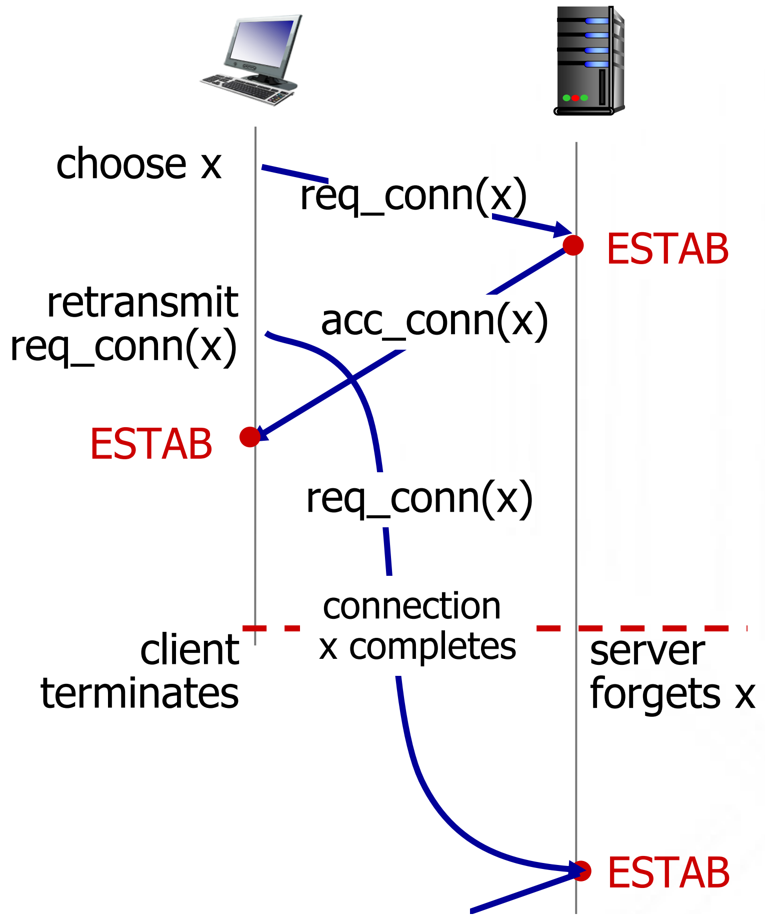
2-way handshake scenarios



No problem!

✓

2-way handshake scenarios



❌ Problem: half open connection! (no client)

TCP 3-way handshake

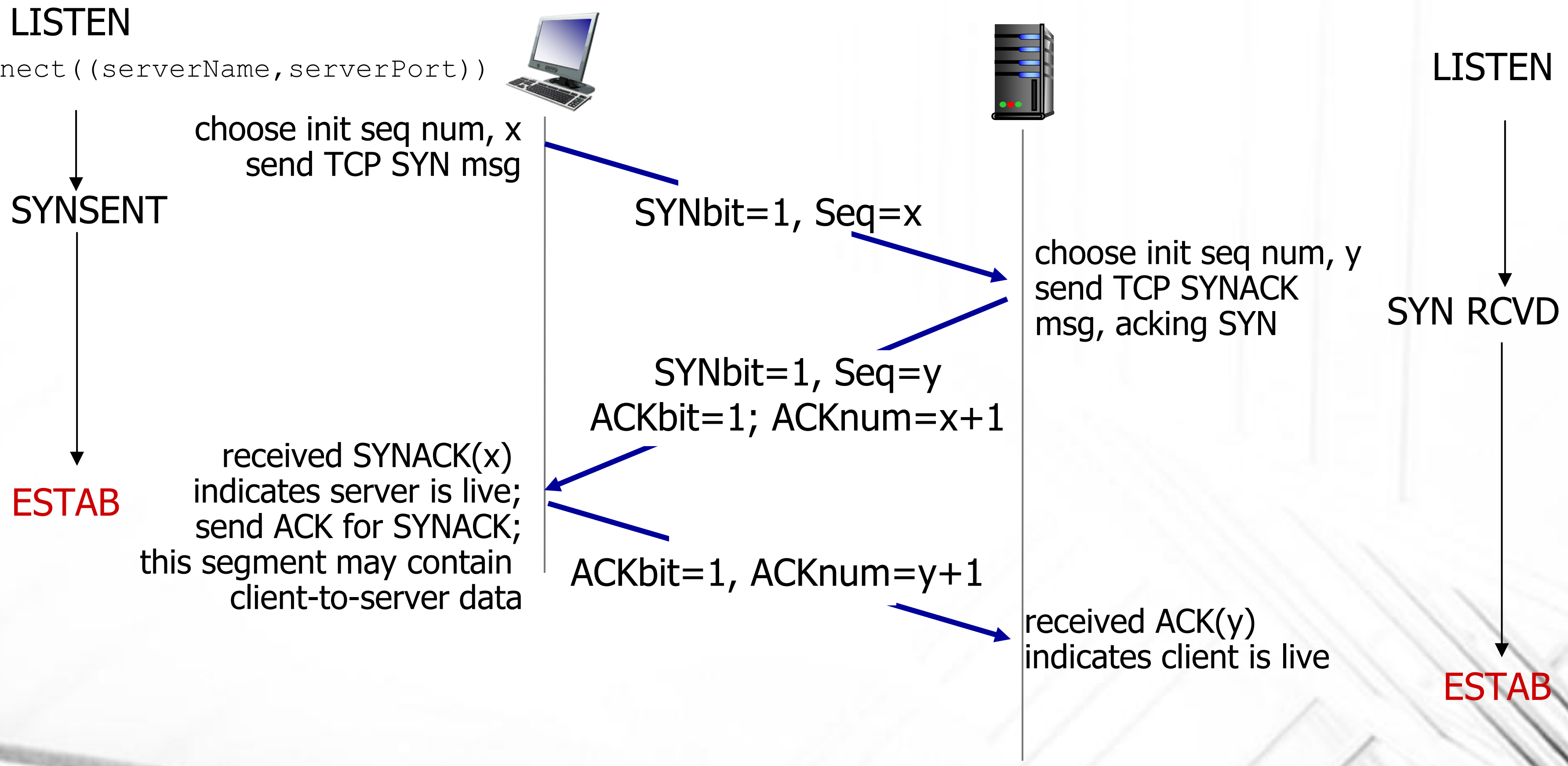
Client state

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

Server state

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

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



TCP Header Fields *(cont'd)*

- Header bits
  - URG = Urgent pointer field in use
  - ACK = Indicates whether frame contains acknowledgement
  - PSH = Data has been "pushed". It should be delivered to higher layers right away.
  - RST = Indicates that the connection should be reset
  - SYN = Used to establish connections
  - FIN = Used to release a connection

TCP is connection oriented

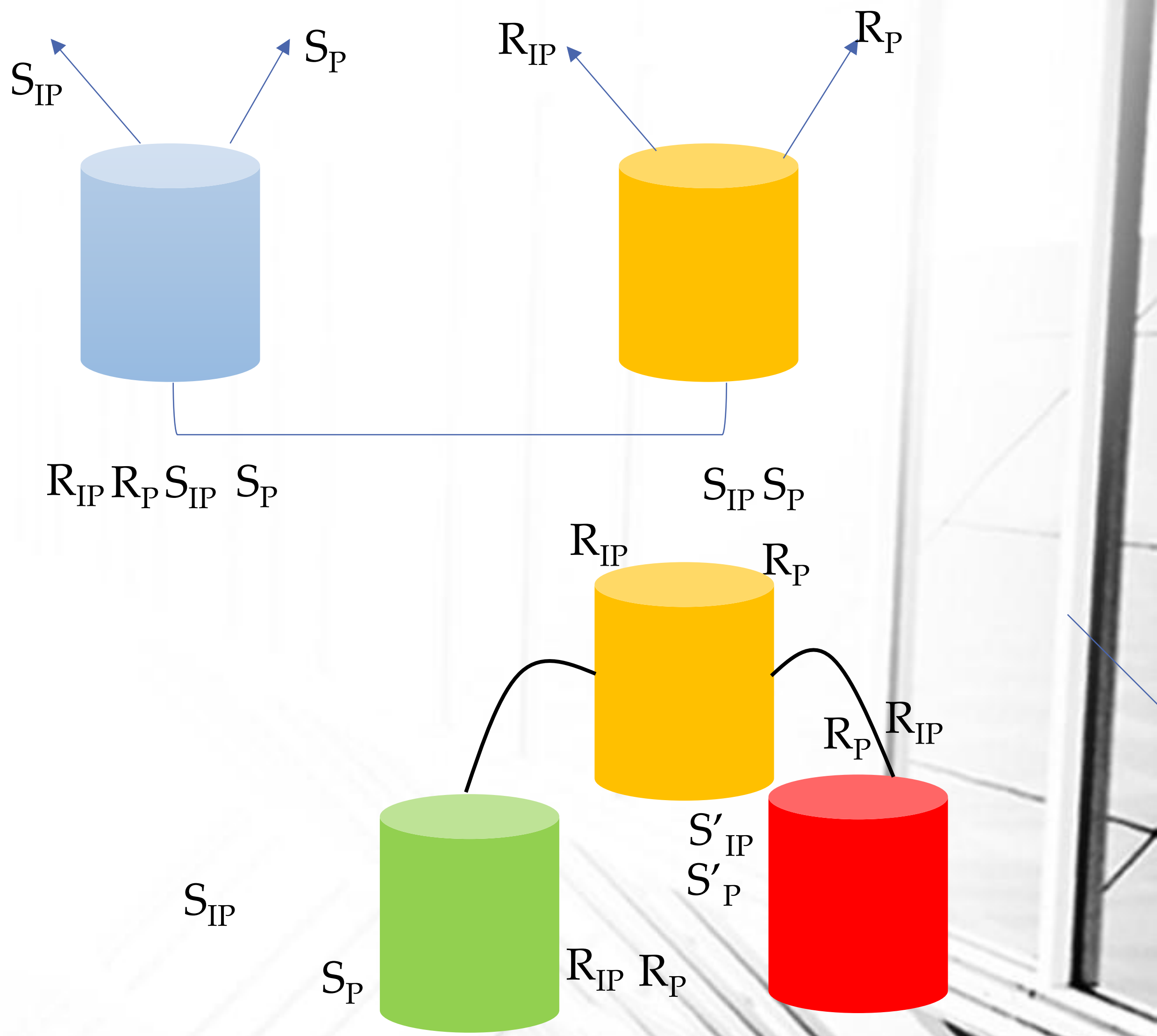
Full duplex

Data sent from sender to receiver and at the same time from receiver to sender

For each connection request, the server accepts , creates another socket with all 4 parameters.

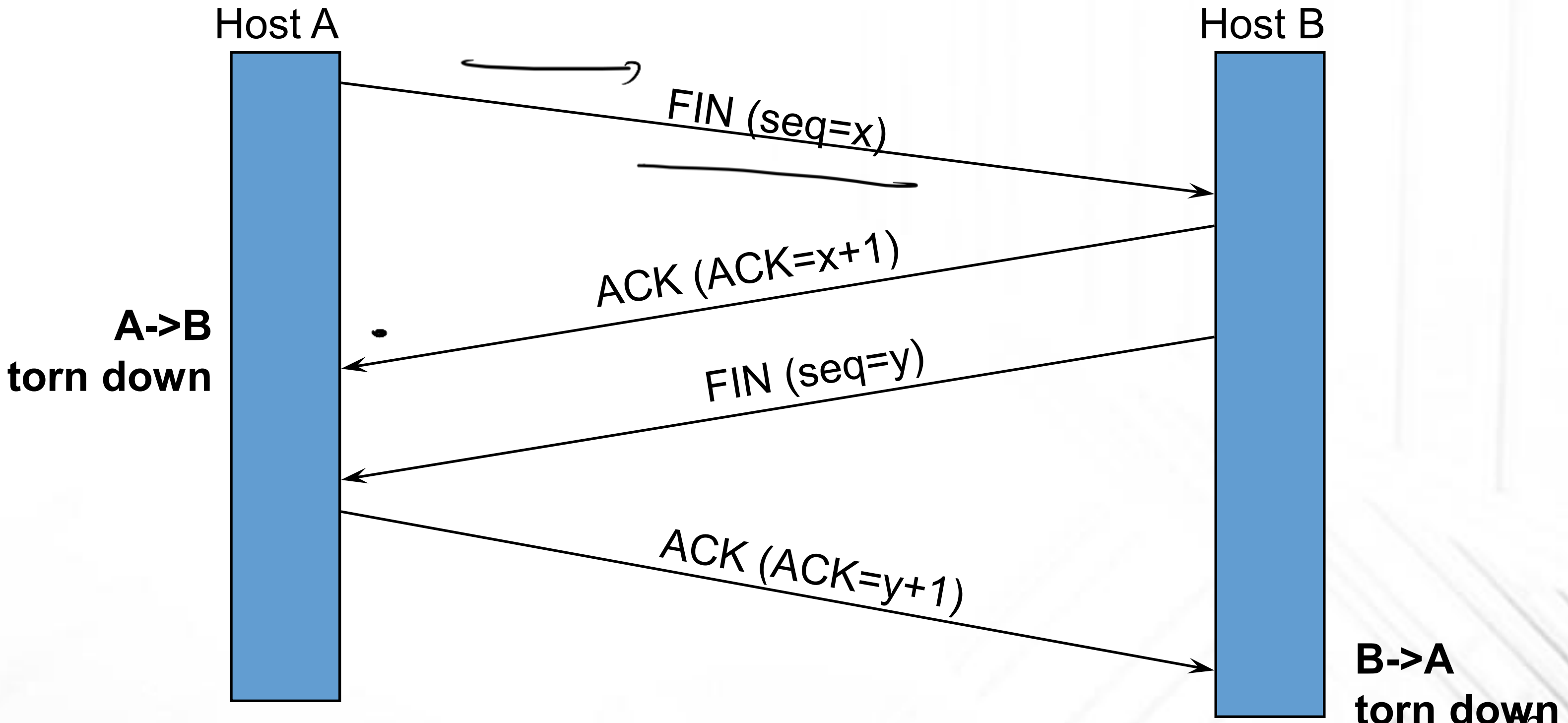
Server can handle multiple connections

- A TCP socket is bound to both sender and receiver



TCP Connection Tear-down

- Two double handshakes:



Retransmission

TCP sequence numbers, ACKs

Sequence numbers:

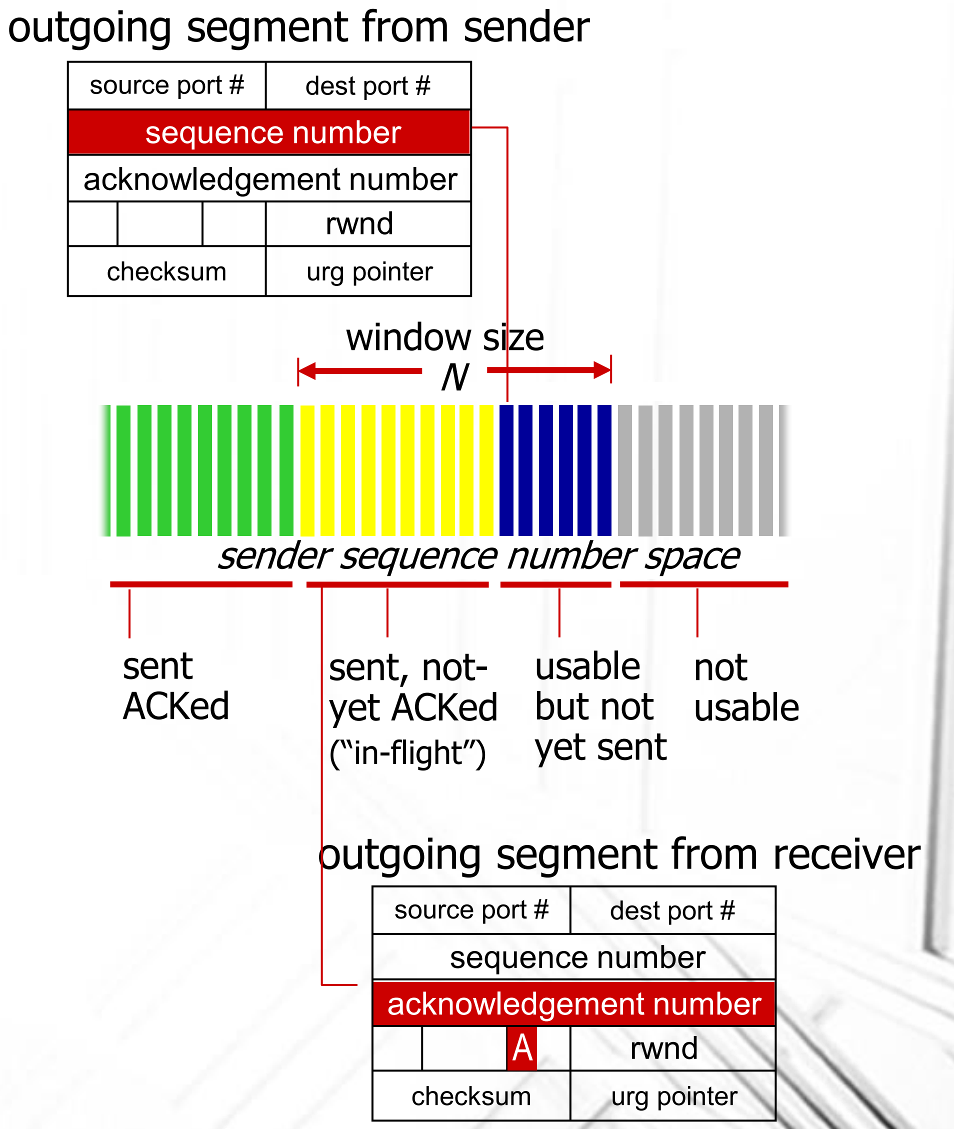
- byte stream "number" of first byte in segment's data

Acknowledgements:

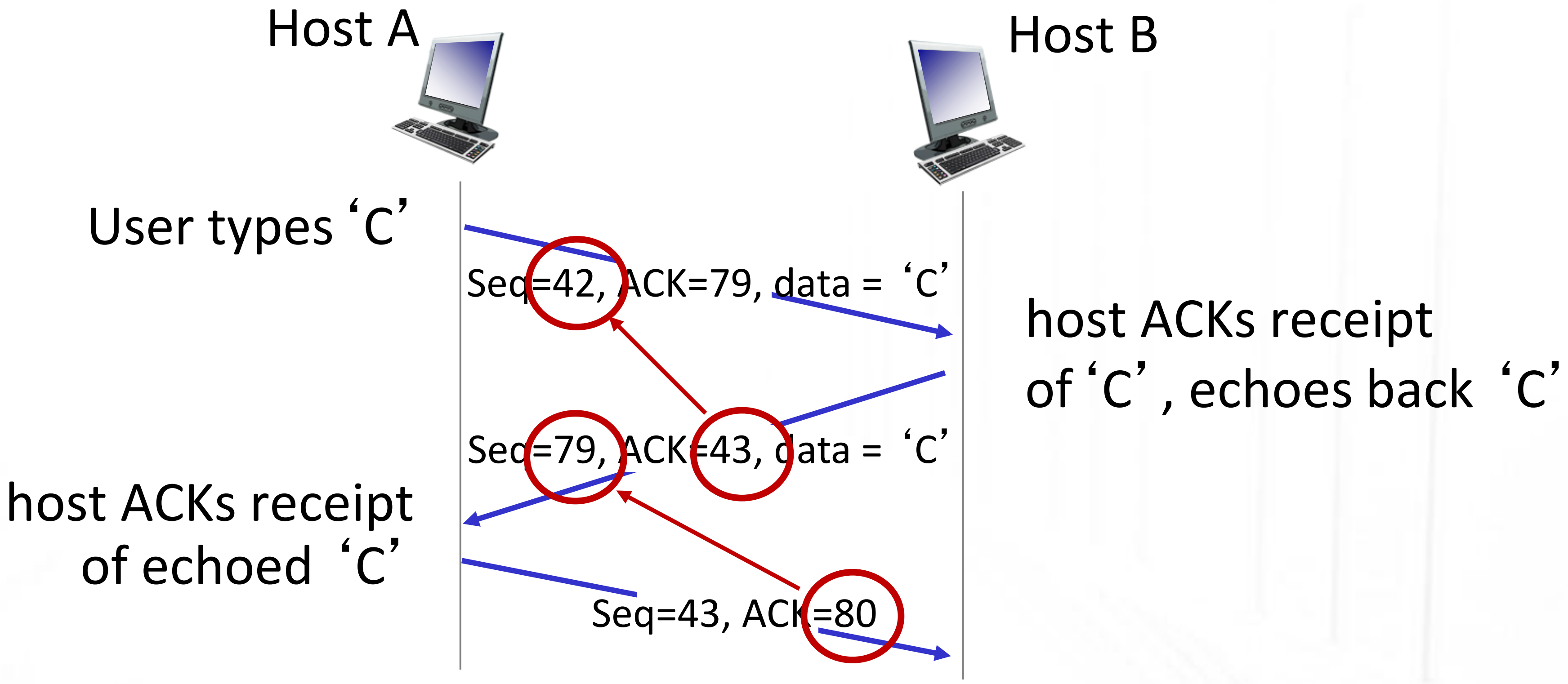
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

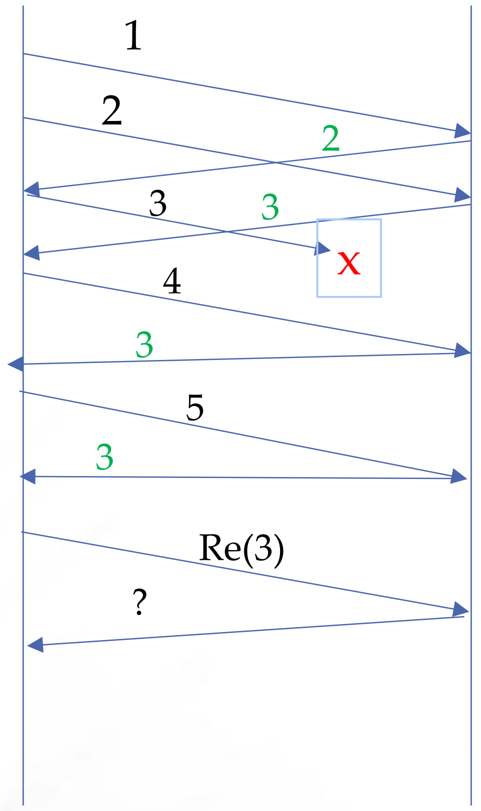


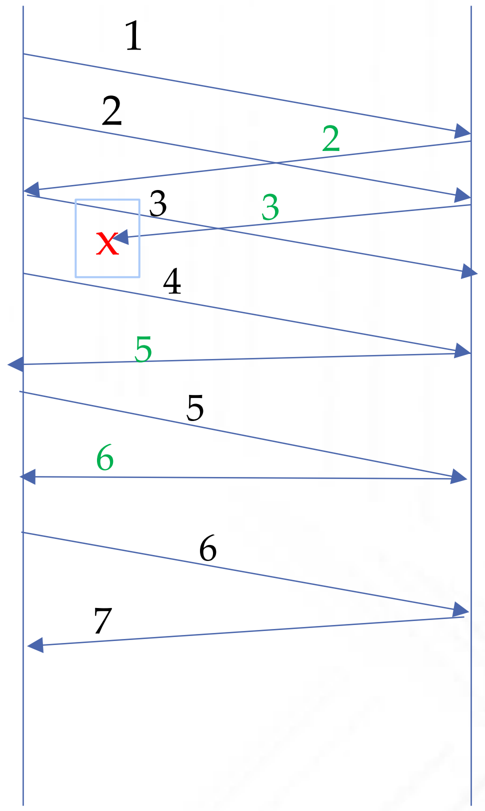
TCP sequence numbers, ACKs


Host A
Host B
User types ‘C’
Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’, echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
host ACKs receipt of echoed ‘C’
Seq=43, ACK=80

simple telnet scenario

Cumulative ack





TCP CUMULATIVE ACK

- ACK the highest seq# received so far or the next expected packet in sequence
- More resilient to lost acks
- Can afford to maintain only one timer

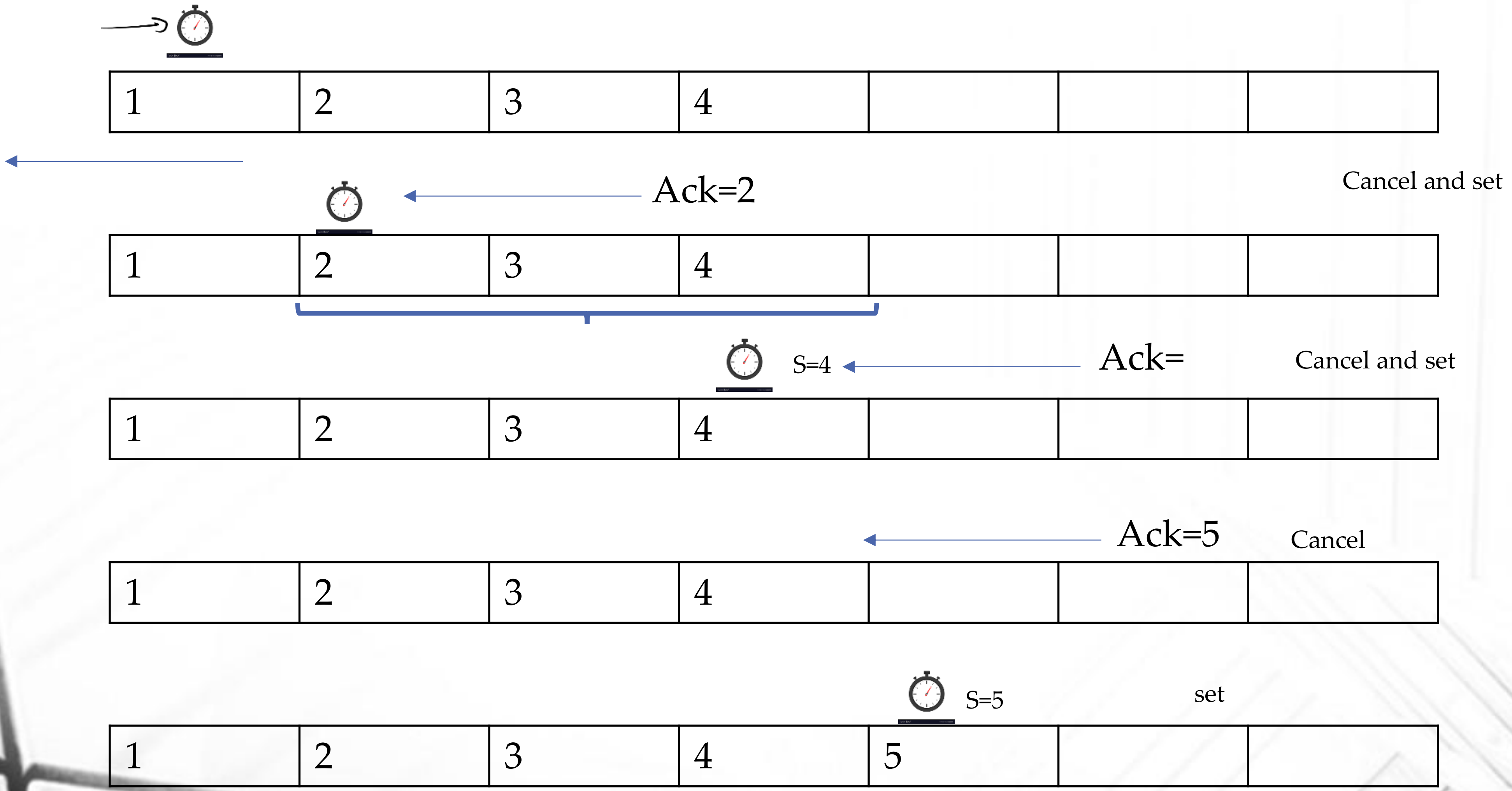
TCP Retransmission

- When a packet remains unacknowledged for a period of time, TCP assumes it is lost and retransmits it
- TCP tries to calculate the round trip time (RTT) for a packet and its acknowledgement
- From the RTT, TCP can guess how long it should wait before timing out
- TCP can send a window of packets Window = K
- Naïve Solution, Set one timer for each packet sent, if expired, retransmit

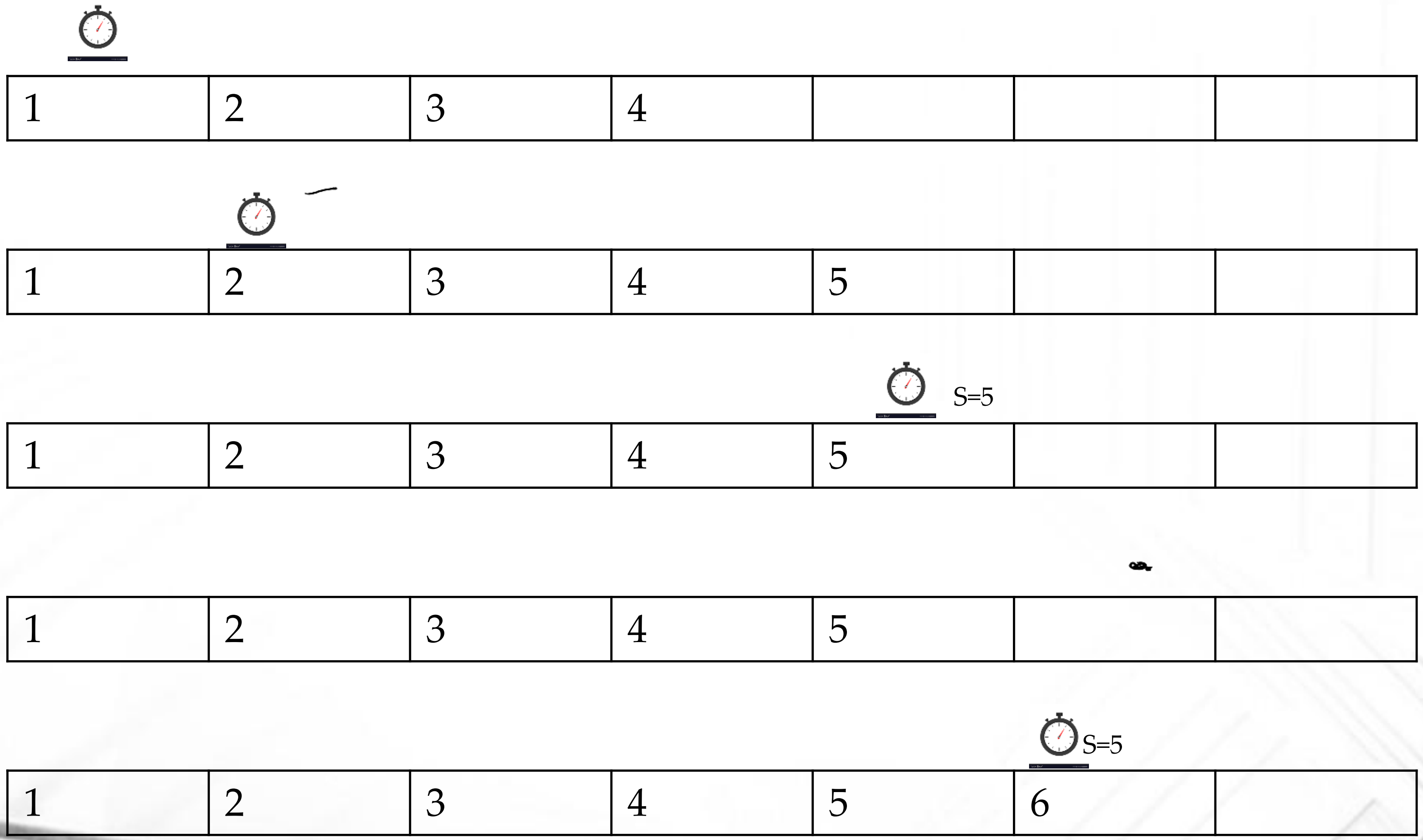
TCP retransmission timer

- Because TCP receiver, sends a cumulative ack, TCP sender needs to maintain only one timer
- Whenever TCP sender has to send a packet, if a timer is already running, send packet
- Else set timer=timeout value, and store timer seq# of sent packet
- When an ack > timer seq# , cancel timer and start new timer with timer seq#= head of retransmission Queue (sent but not acked)
- If timer expires, retransmit packet
- Basically one timer: set, cancel and set, cancel operations on timer

TCP timer: set or cancel



TCP timer: set or cancel



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

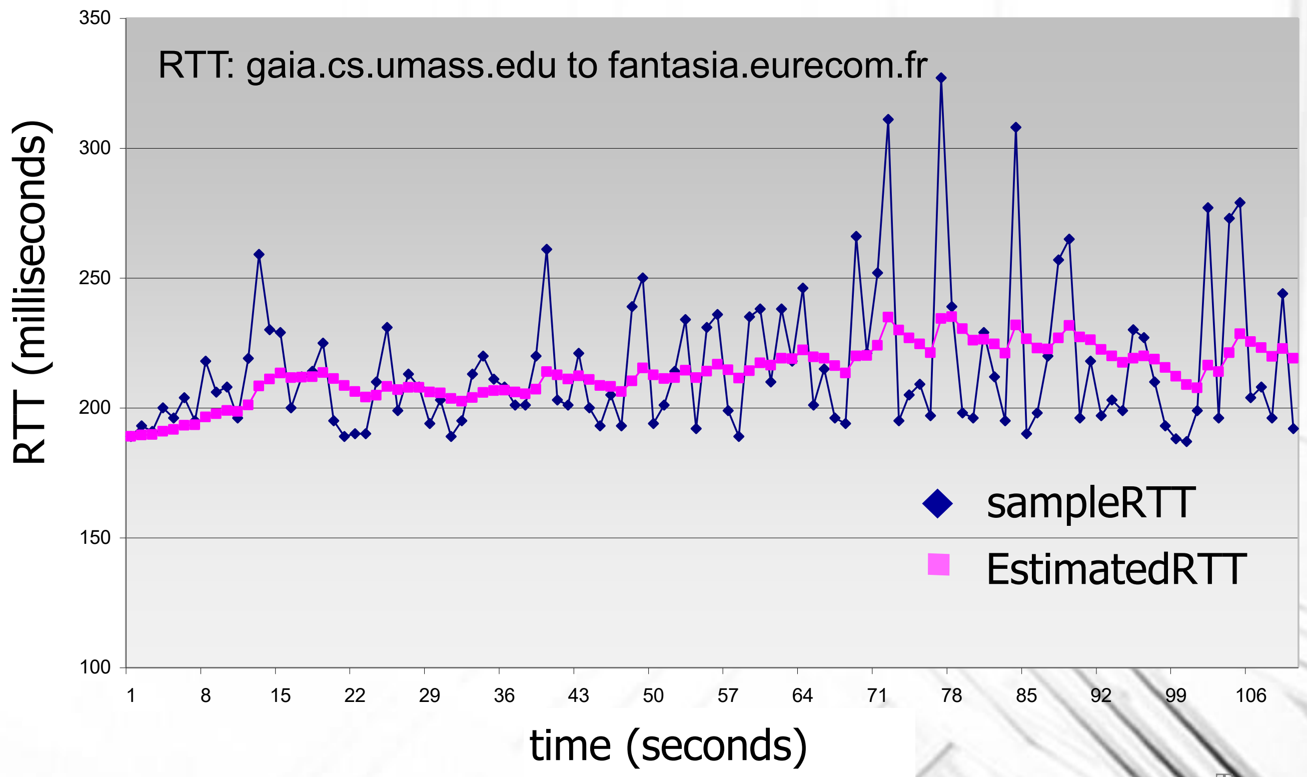
Q: how to estimate RTT?

- `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

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

- <u>e</u>xponential <u>w</u>eighted <u>m</u>oving <u>a</u>verage (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus “safety margin”
  - large variation in **`EstimatedRTT:`** want a larger safety margin

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

estimated RTT (↑ EstimatedRTT) “safety margin” (↑ 4*DevRTT)

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

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

(typically, β = 0.25)

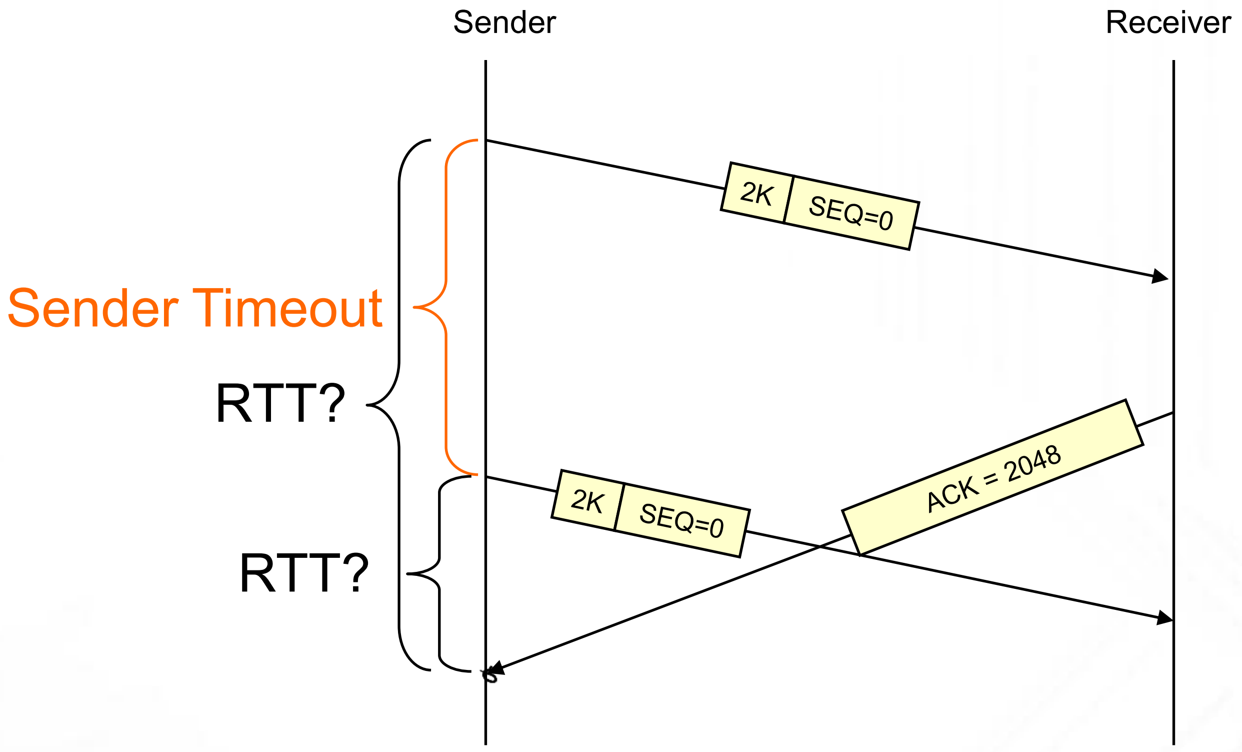
Retransmission Timeout Interval (RTO)

- The timeout value is then calculated by multiplying the smoothed RTT by some factor (greater than 1) called β

$$\text{Timeout} = \beta * \text{SRTT}$$

- This coefficient of β is included to allow for some variation in the round trip times.

Problem with RTT Calculation



Karn’s Algorithm

- Retransmission ambiguity
  - Measure RTT from original data segment
  - Measure RTT from most recent segment
- Either way there is a problem in RTT estimate
- One solution
  - Never update RTT measurements based on acknowledgements from retransmitted packets
- Problem: Sudden change in RTT can cause system never to update RTT
  - Primary path failure leads to a slower secondary path

Karn’s algorithm

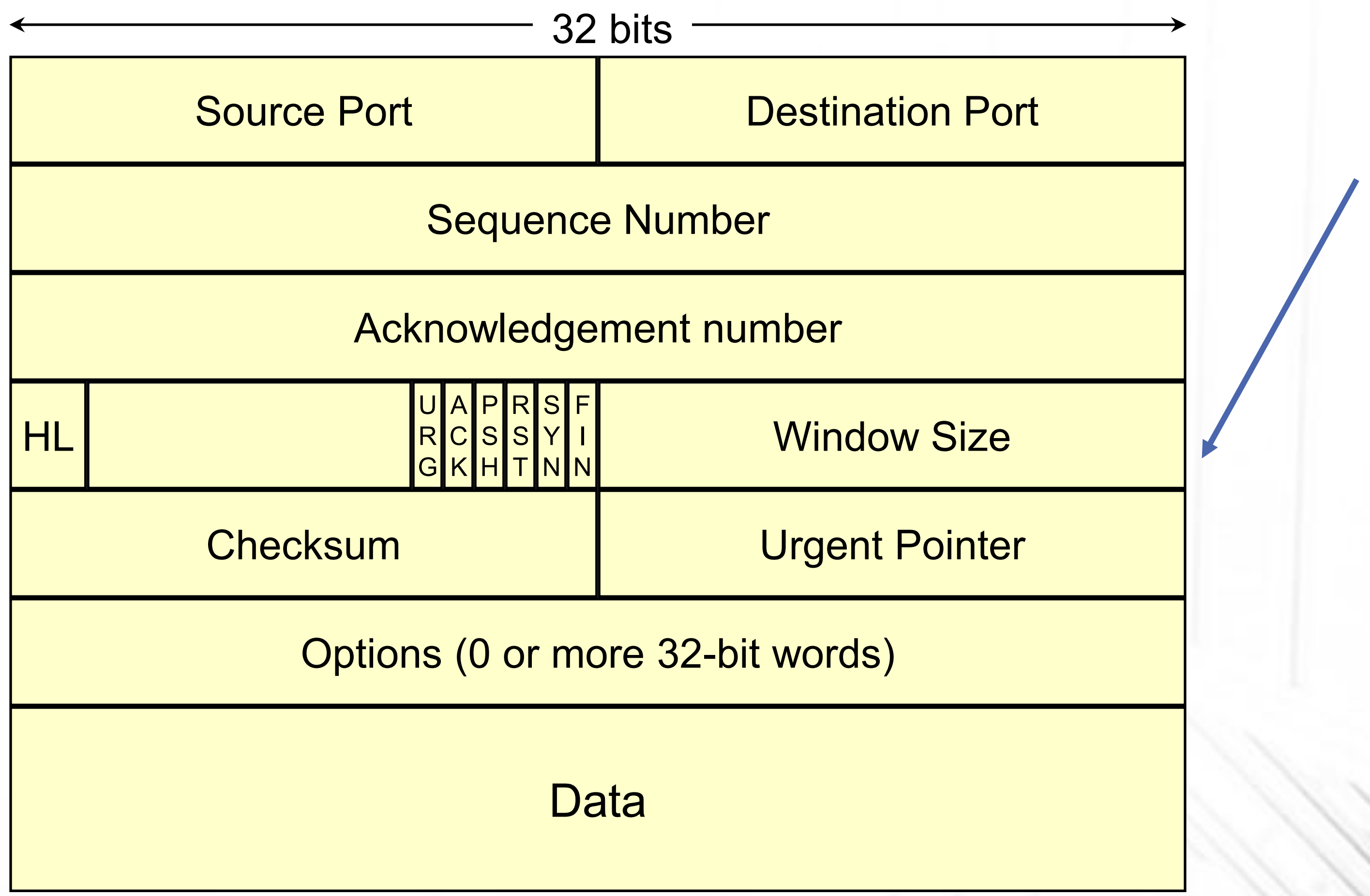
- Use back-off as part of RTT computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from SRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from SRTT

Flow Control

TCP flow control

- Receiver buffers out of order packets
- How big a buffer?
- Receiver tells sender how much space is left
- During connection establishment
  - Tell MAX flow control window size
- Send current buffer availability : advertised window
- Window advertisement carried in ACKS

TCP Header Format



TCP Flow Control

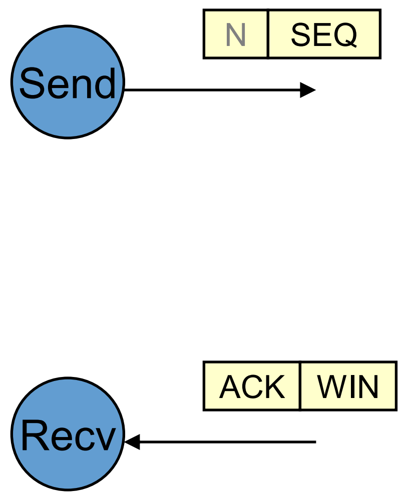
- TCP uses a modified version of the sliding window
- In acknowledgements, TCP uses the "Window size" field to tell the sender how many bytes it may transmit
- TCP uses bytes, not packets, as sequence numbers

TCP Header Fields used for flow control

- Window size
  - Specifies how many bytes may be sent after the first acknowledged byte
- Checksum
  - Checksums the TCP header and IP address fields
- Urgent Pointer
  - Points to urgent data in the TCP data field

TCP Flow Control *(cont'd)*

Important information in TCP/IP packet headers



Number of bytes in packet (N)

Sequence number of first data byte in packet (SEQ)

ACK bit set

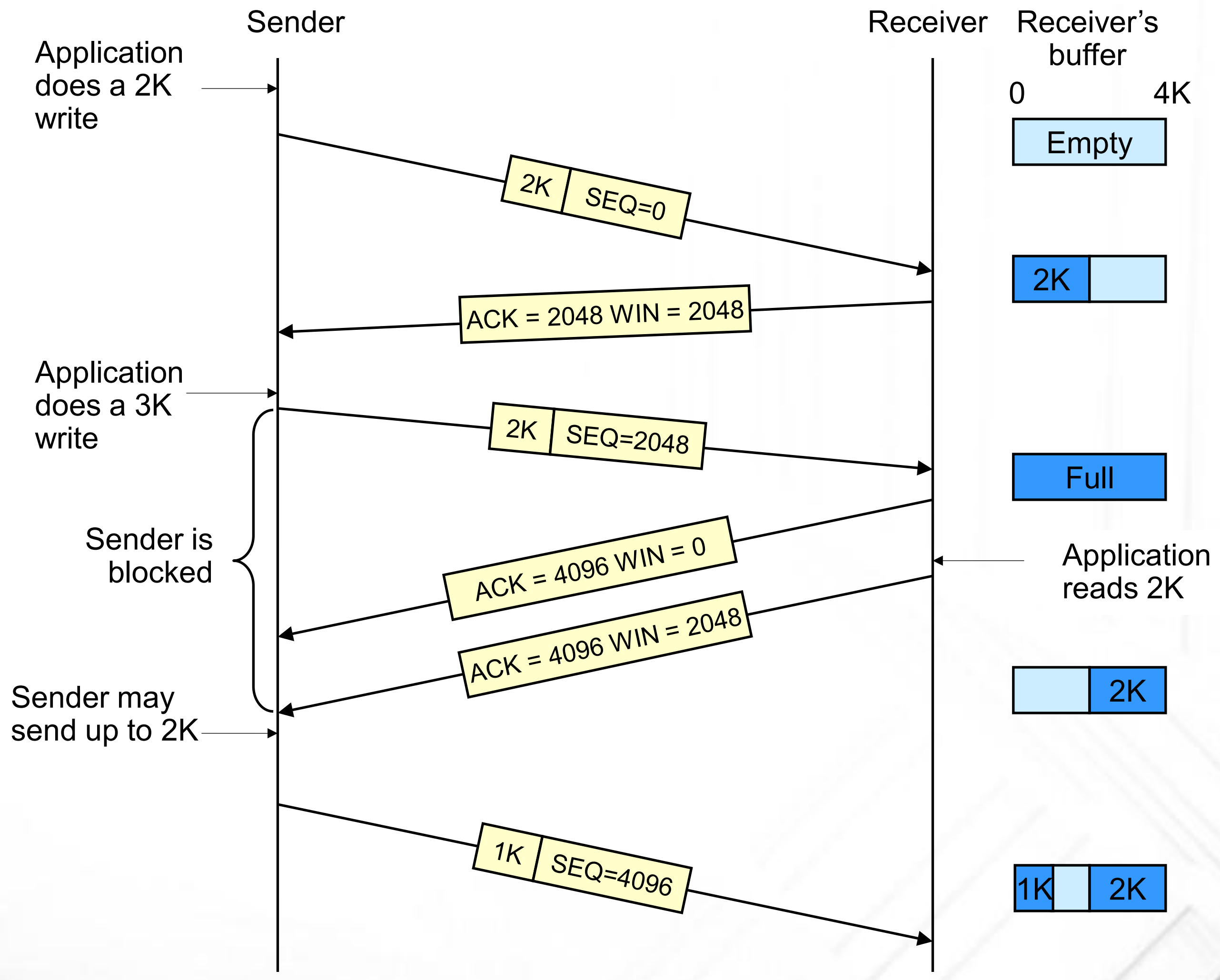
Sequence number of next expected byte (ACK)

Window size at the receiver (WIN)

Contained in IP header

Contained in TCP header

TCP Flow Control *(cont'd)*



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)
- 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

TCP Congestion Control

- Goal: fully (fairly) utilize the resource (bandwidth)
  - Don't over use - congestion
  - Don't under use - waste
- Goal: achieve self-clocking state
  - Even if don't know bandwidth of bottleneck
  - Bottleneck may change over time

TCP Congestion Window

- TCP introduces a second window, called the "congestion window"
- This window maintains TCP's best estimate of amount of outstanding data to allow in the network to achieve self-clocking
- Sending size = min(congestion control window, flow control window)

TCP Congestion Control

- Two phases to keep bottleneck busy (fully utilize the resource):
  - Increase the usage (window size) to keep probing the network
  - Decrease the usage when congestion is detected

TCP congestion control: AIMD

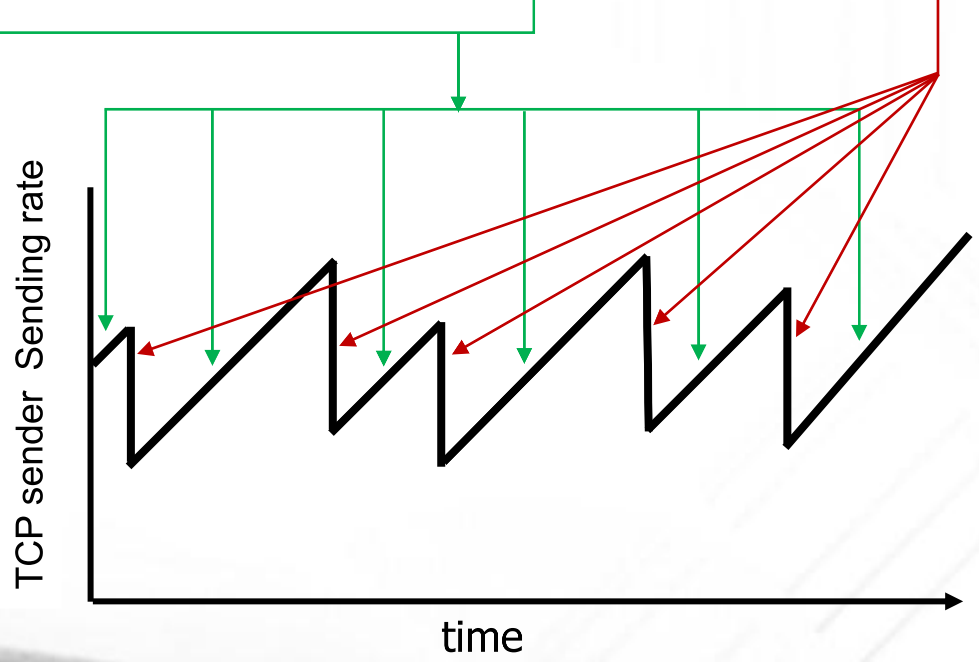
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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

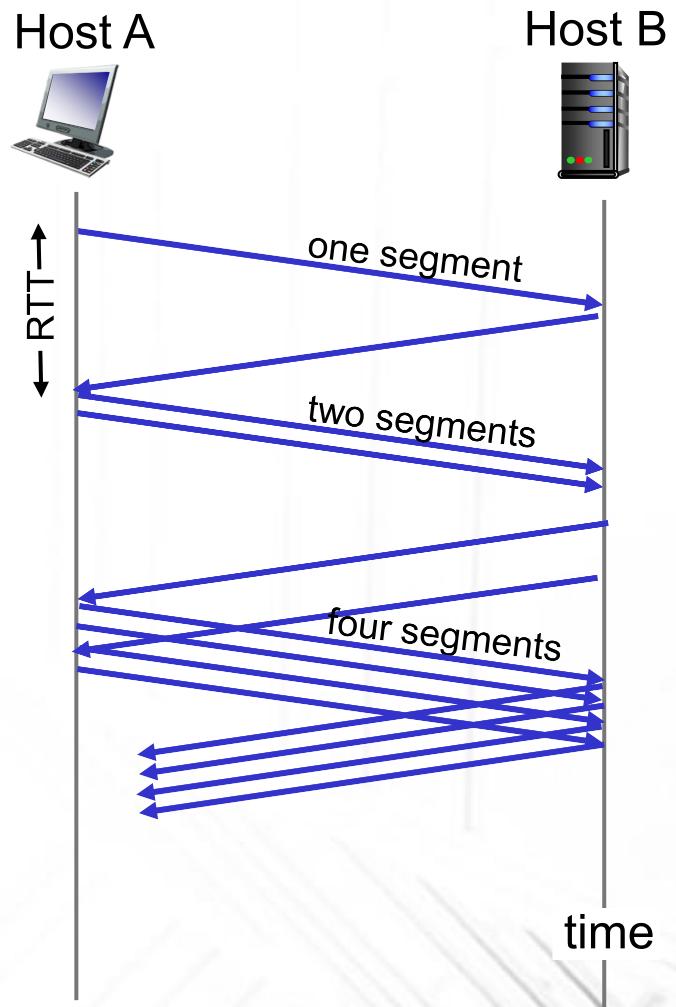
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

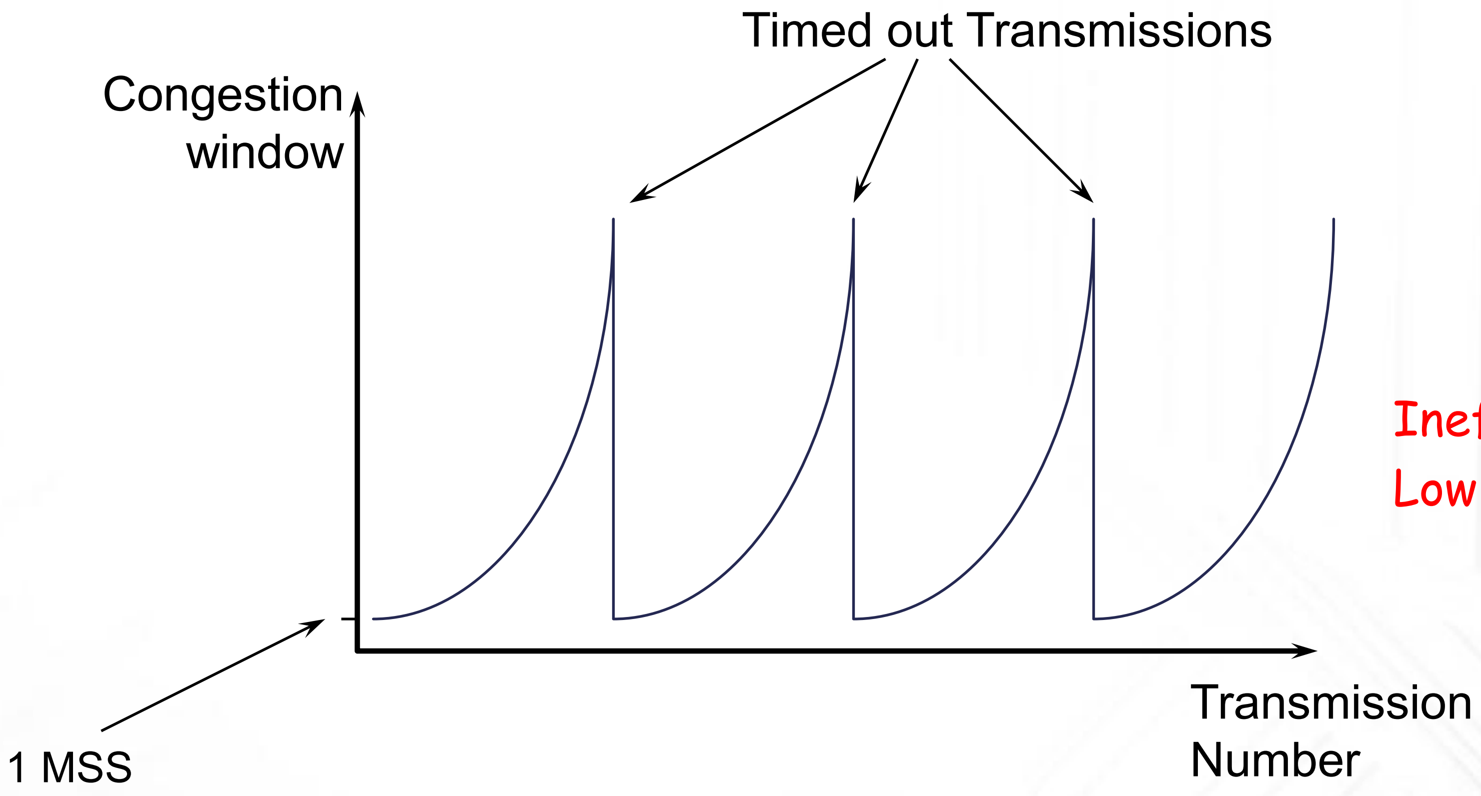
- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide [Kelly]!
  - have desirable stability properties

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 *(cont'd)*

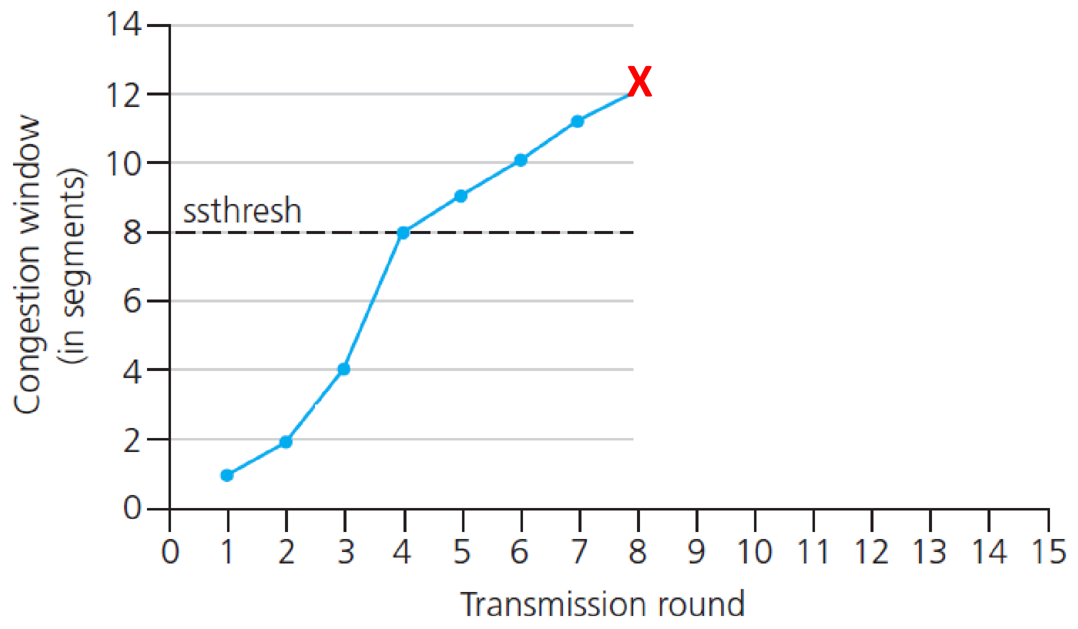


Inefficient,
Low throughput

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

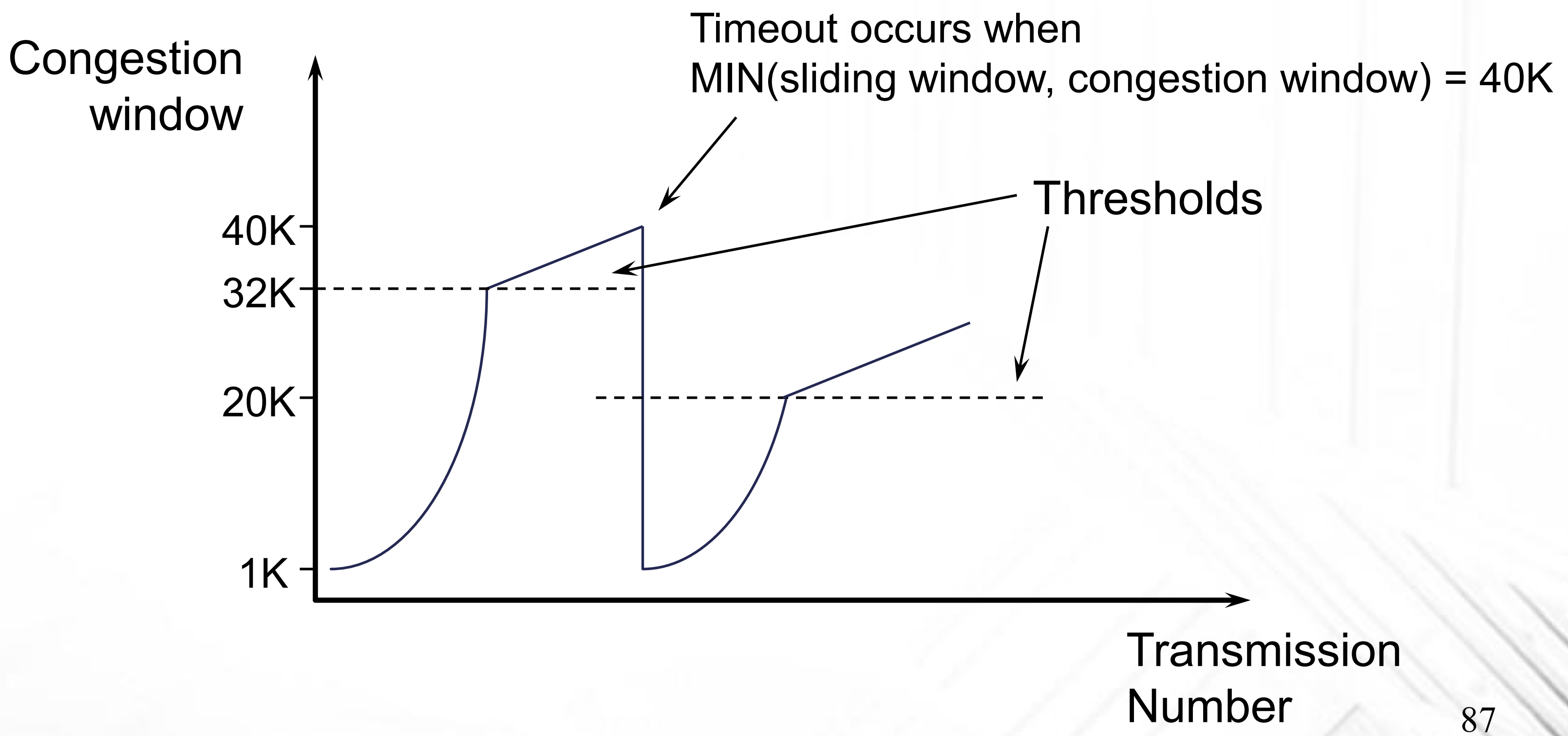
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP Linear Increase Algorithm

- Algorithm:
  - Start the threshold at 64K
  - Slow start
  - Once the threshold is passed
    - For each ack received,
      - 1 MSS for each congestion window of data transmitted
    - Timeout
      - reset the congestion window size to 1 MSS

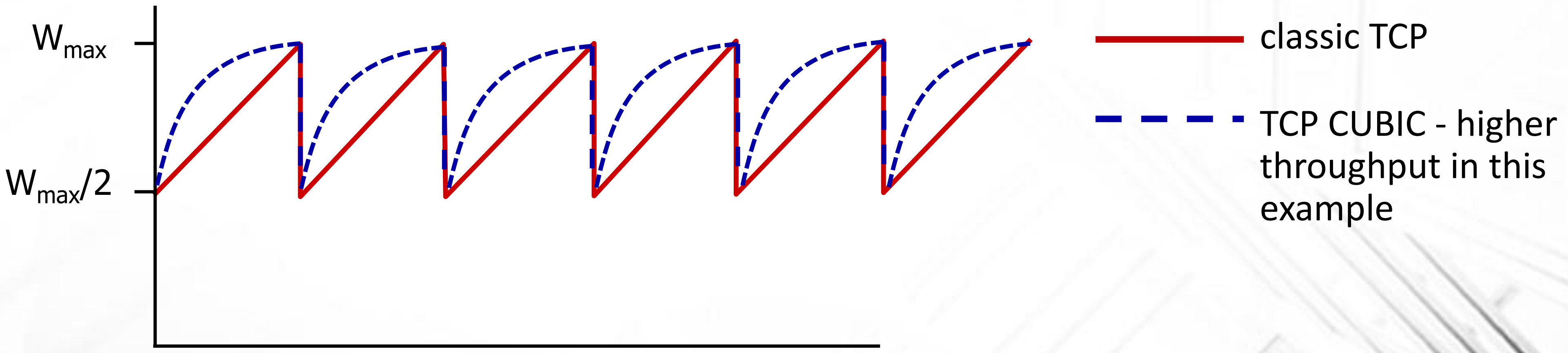
TCP Linear Increase Threshold Phase

Example: Maximum segment size = 1K
Assume thresh=32K



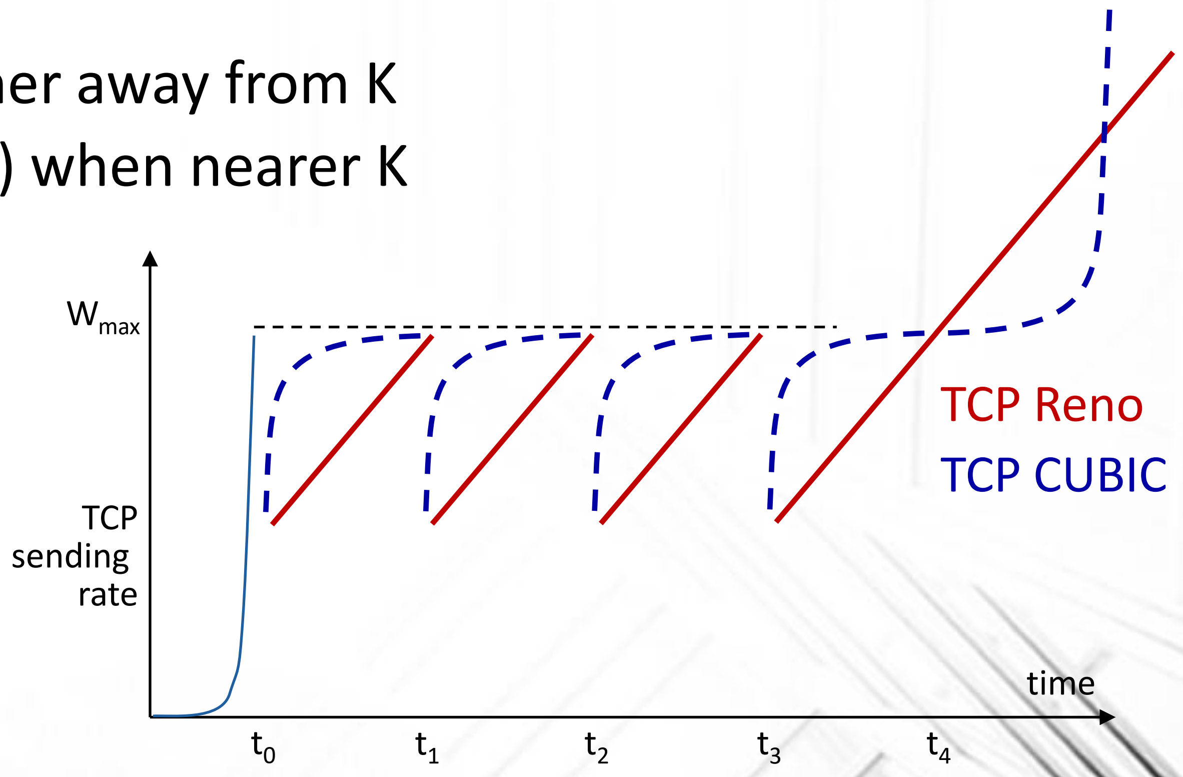
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tunable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers (until ~2024)



TCP Fast Retransmit

- Idea: When sender sees 3 duplicate ACKs, it assumes something went wrong
- The packet is immediately retransmitted instead of waiting for it to timeout

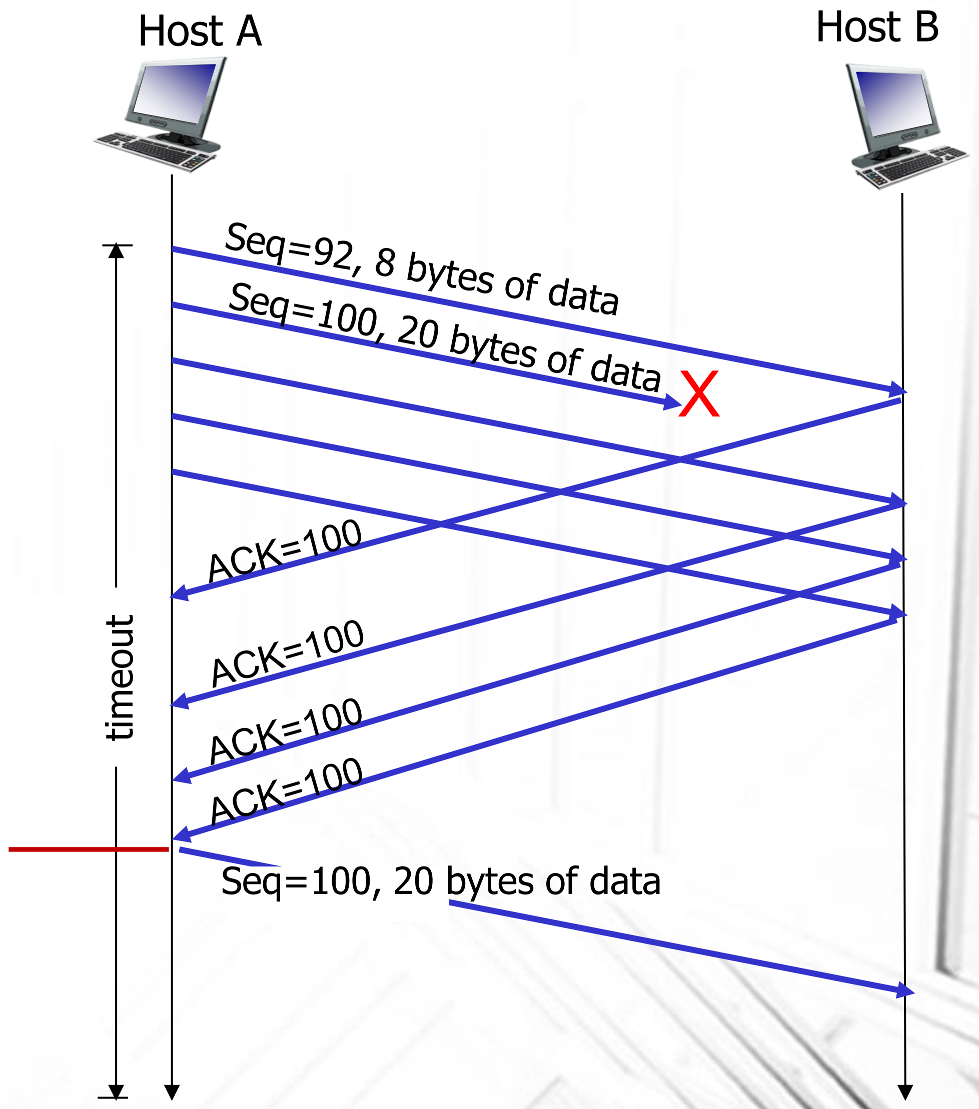
TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

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

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



TCP Recap

- Timeout Computation
  - Timeout is a function of 2 values
    - the weighted average of sampled RTTs
    - The sampled variance of each RTT
- Congestion control:
  - Goal: Keep the self-clocking pipe full in spite of changing network conditions
  - 3 key Variables:
    - Sliding window (Receiver flow control)
    - Congestion window (Sender flow control)
    - Threshold (Sender's slow start vs. linear mode line)

Algorithm 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 max(**`FlightSize/2,2*mss)`** and **`CongWin`** set to **`Threshold+3*mss`**. (Fast retransmit, Fast recovery)
- When timeout occurs, **`Threshold`** set to max(**`FlightSize/2,2*mss)`** and **`CongWin`** is set to 1 MSS.

FlightSize: The amount of data that has been sent but not yet acknowledged.