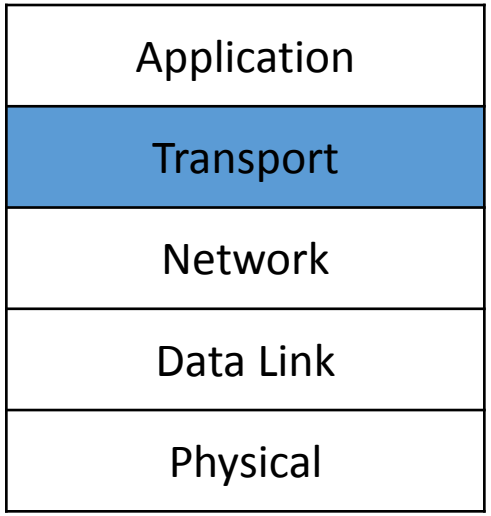
# TCP
## TRANSMISSION CONTROL PROTOCOL

Valerio Luconi
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
valerio.luconi@iit.cnr.it

Based on original slides from Computer Networking: a top-down approach. Copyright J. F. Kurose, K. W. Ross.

# What is TCP?

Transport Layer protocol for reliable communication.
Provides logical communication between processes.



# Why is TCP important?

# Why is TCP important?

More than 65% of the entire Internet traffic: 62.4 exobytes (62,400,000,000,000,000,000 B).

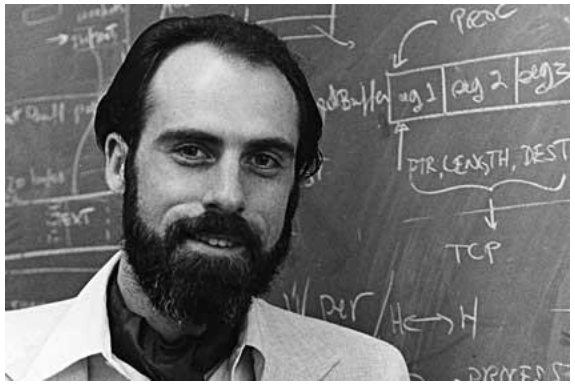
All on TCP!

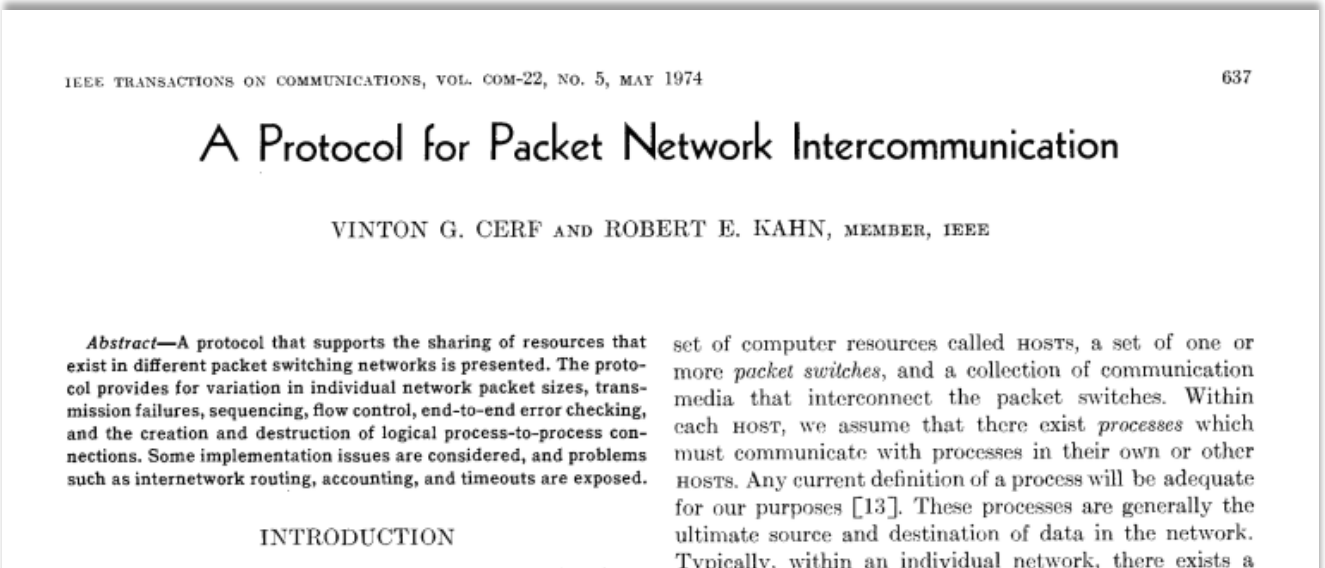
# A bit of history

1974 – Vint Cerf & Bob Kahn publish the first paper introducing TCP/IP concepts

1981 – TCP becames IETF standard RFC 793 / STD 7

1981-2015 – Many other RFCs with optimizations (1122, 1323, 2018, 2581)





IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. COM-22, NO. 5, MAY 1974 637

## A Protocol for Packet Network Intercommunication

VINTON G. CERF AND ROBERT E. KAHN, MEMBER, IEEE

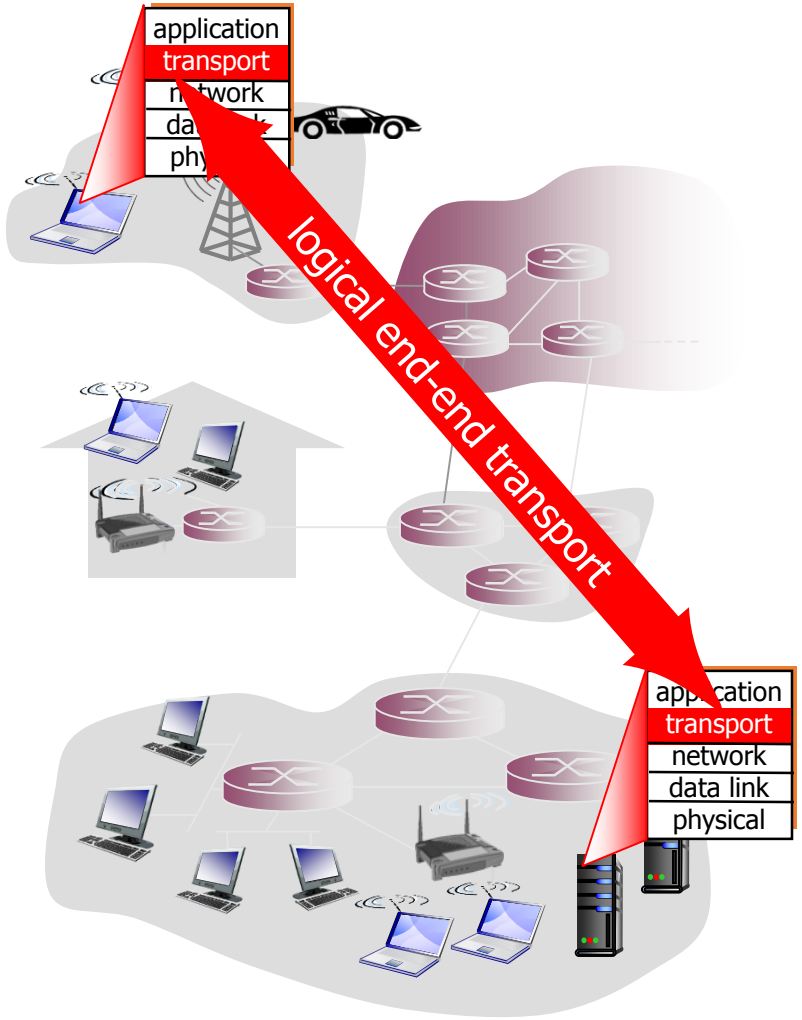
*Abstract*—A protocol that supports the sharing of resources that exist in different packet switching networks is presented. The protocol provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, end-to-end error checking, and the creation and destruction of logical process-to-process connections. Some implementation issues are considered, and problems such as internetwork routing, accounting, and timeouts are exposed.

### INTRODUCTION

set of computer resources called HOSTS, a set of one or more *packet switches*, and a collection of communication media that interconnect the packet switches. Within each HOST, we assume that there exist *processes* which must communicate with processes in their own or other HOSTS. Any current definition of a process will be adequate for our purposes [13]. These processes are generally the ultimate source and destination of data in the network. Typically, within an individual network, there exists a

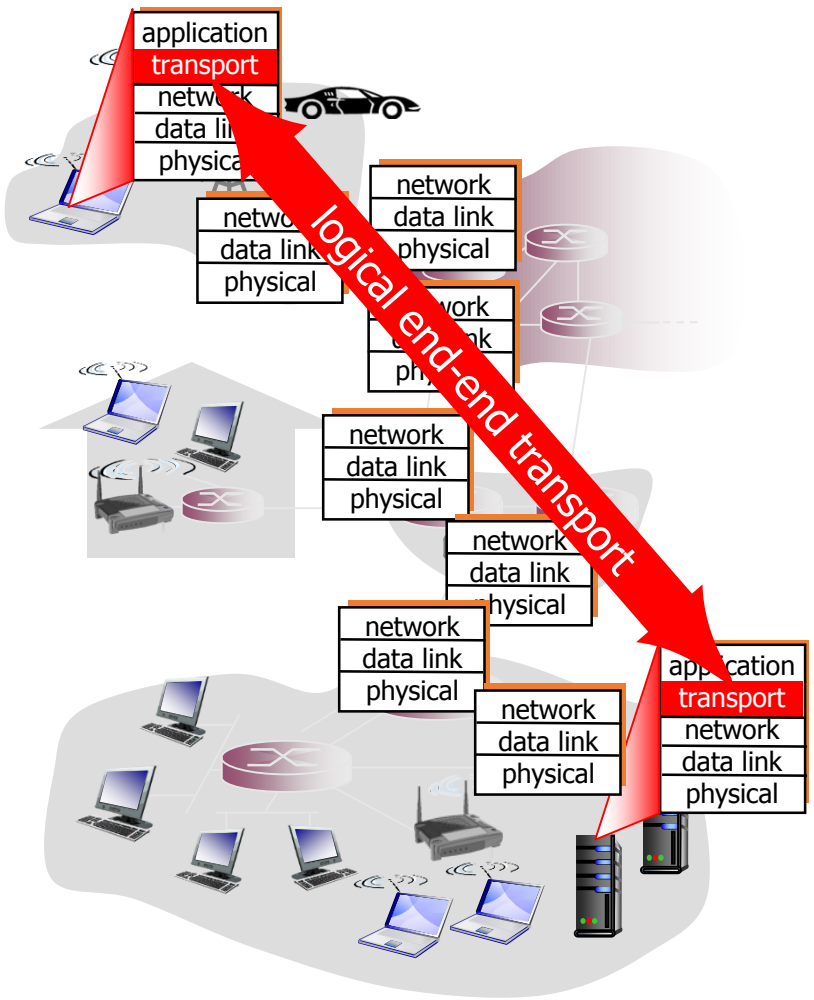
# Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



# Internet transport-layer protocols

- TCP: reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- UDP: unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



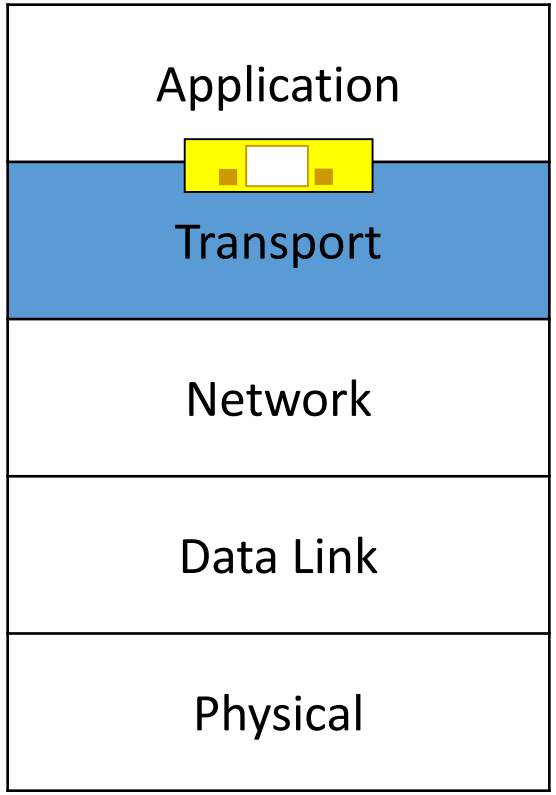
# TCP features

- Multiplexing/demultiplexing
- Reliable communication
- Ordered delivery
- Connection oriented
- Full-duplex communication
- Flow control
- Congestion control

RFCs: 793, 1122, 1323, 2018, 2581

# Multiplexing/demultiplexing

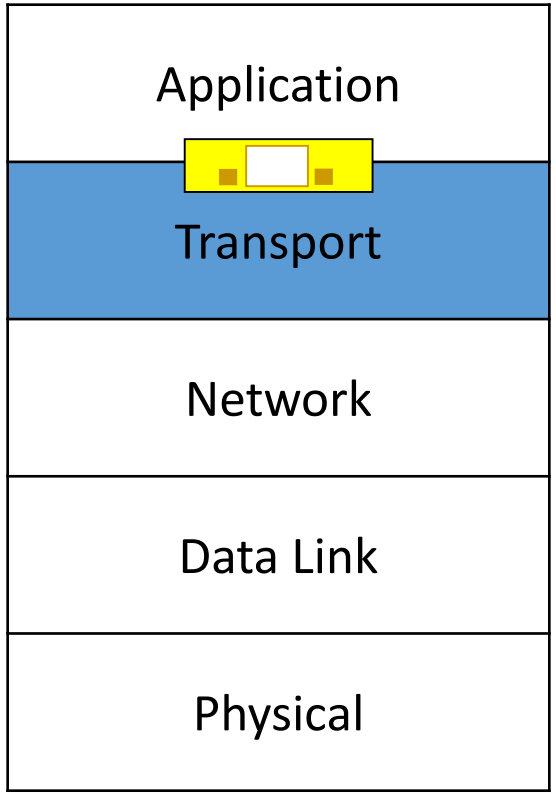
# Multiplexing/demultiplexing

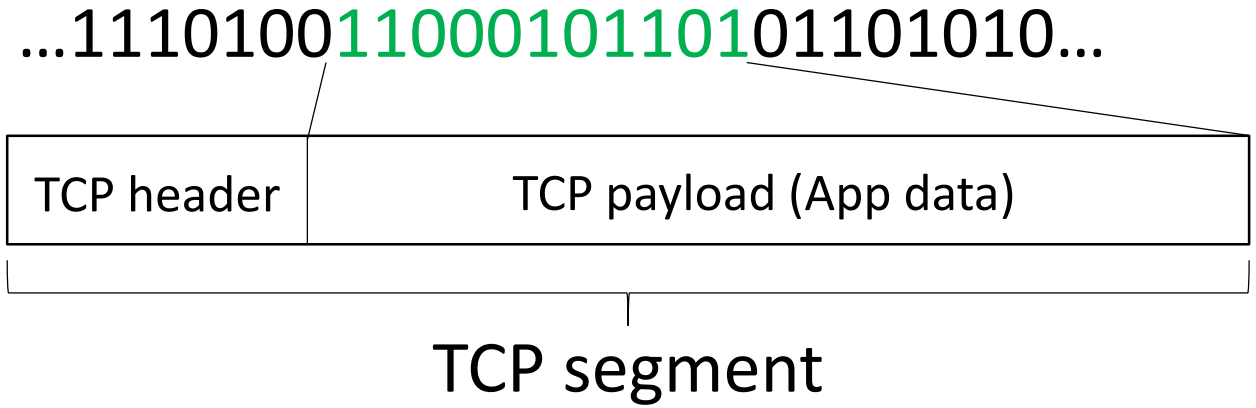



Socket

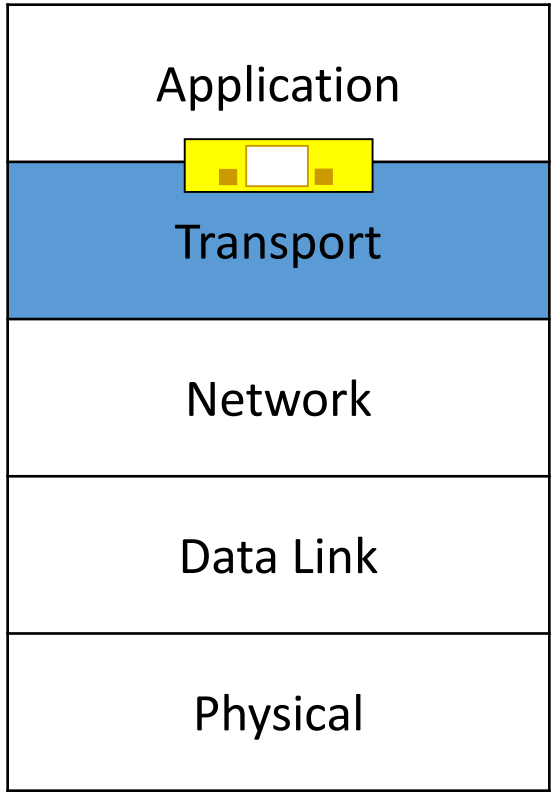
Interface between Application Layer and Transport Layer

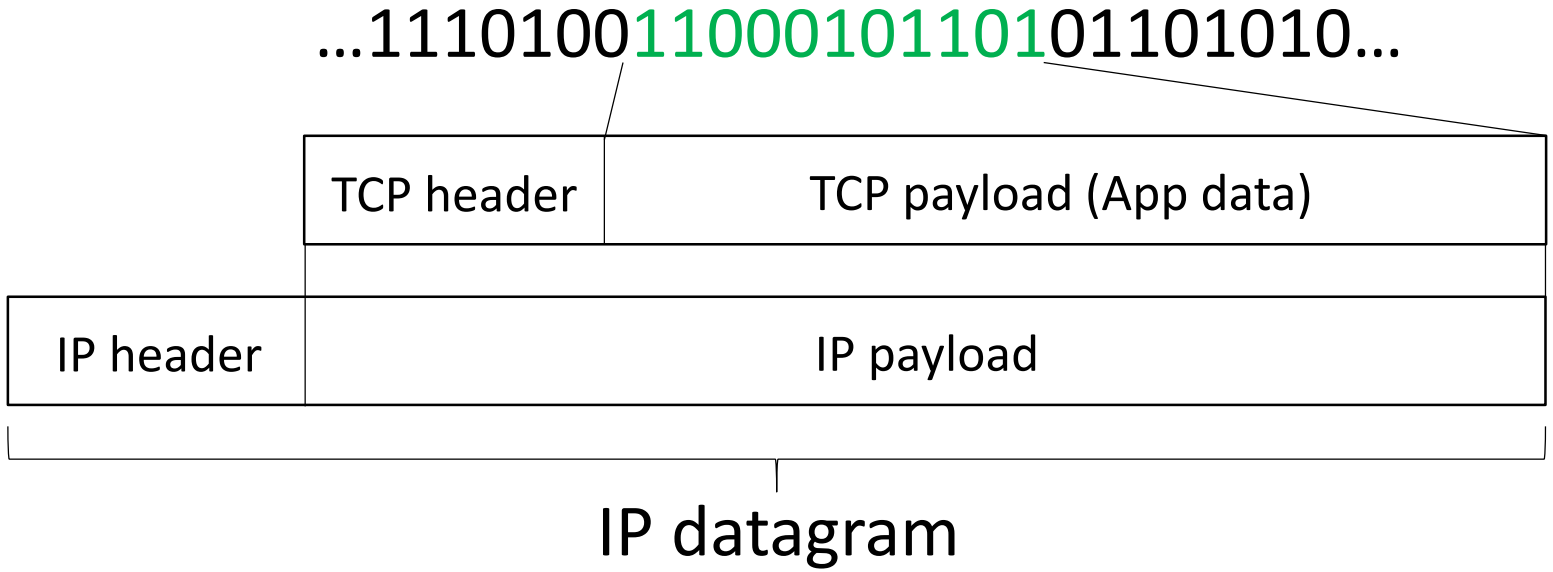
# Multiplexing/demultiplexing



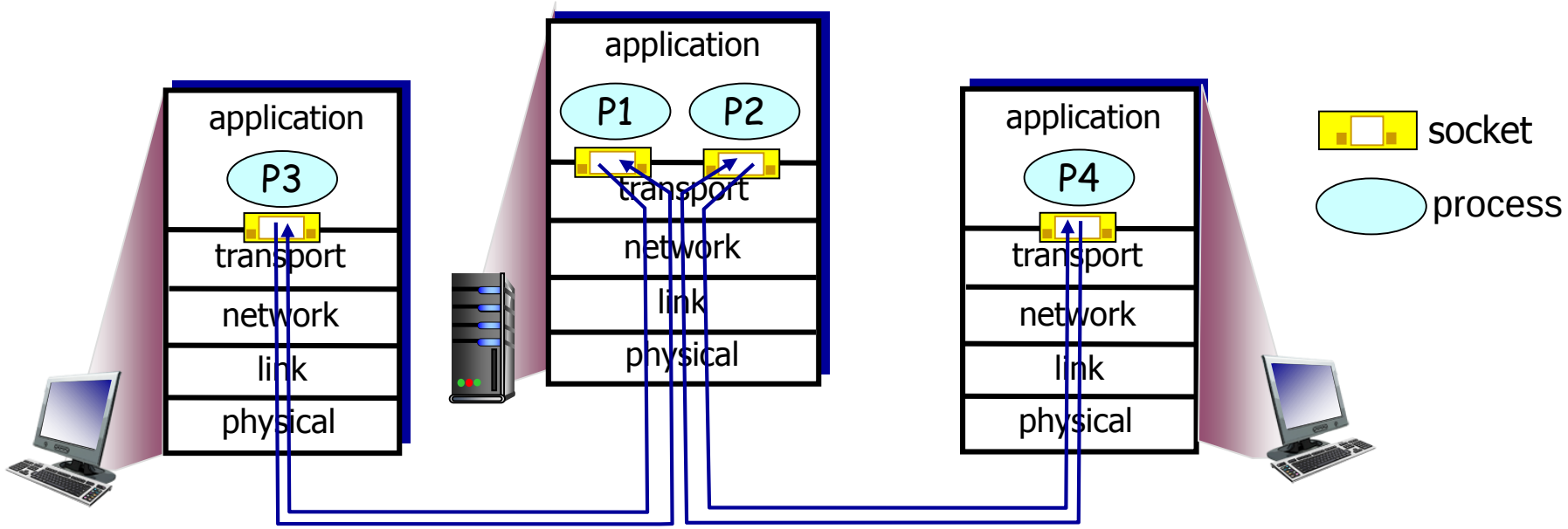



# Multiplexing/demultiplexing



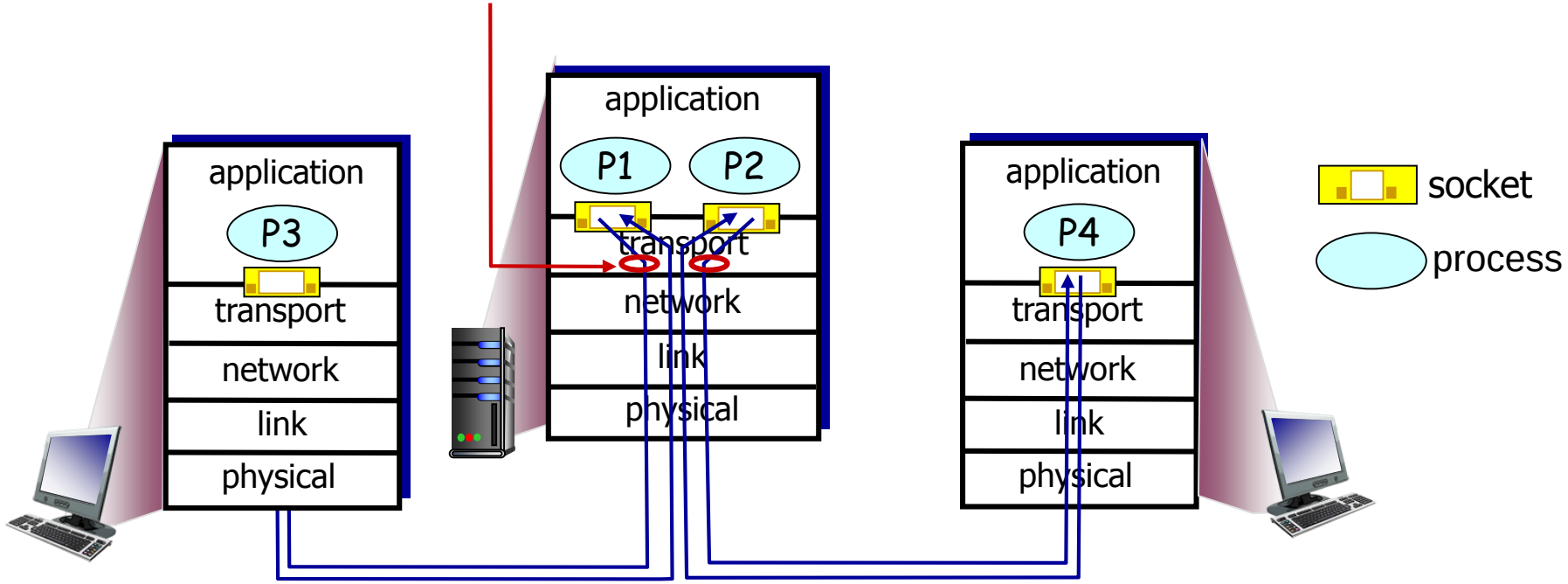



# Multiplexing/demultiplexing



# Multiplexing/demultiplexing

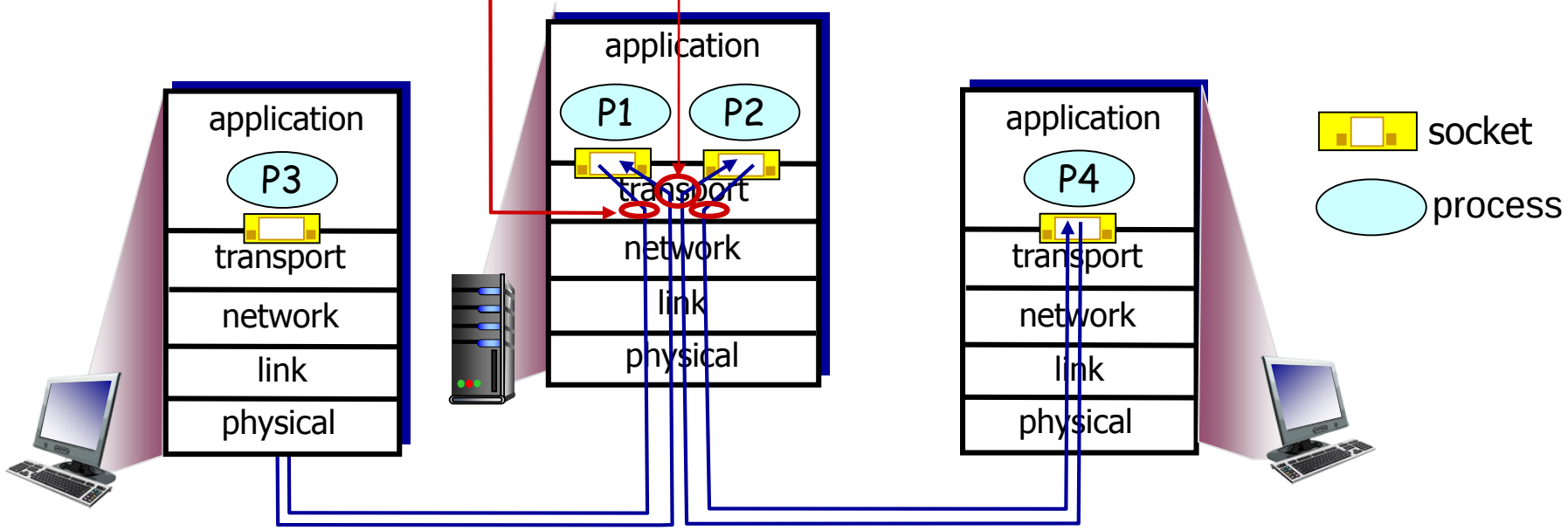
*multiplexing at sender:*
handle data from multiple sockets, add transport header (later used for demultiplexing)



# Multiplexing/demultiplexing

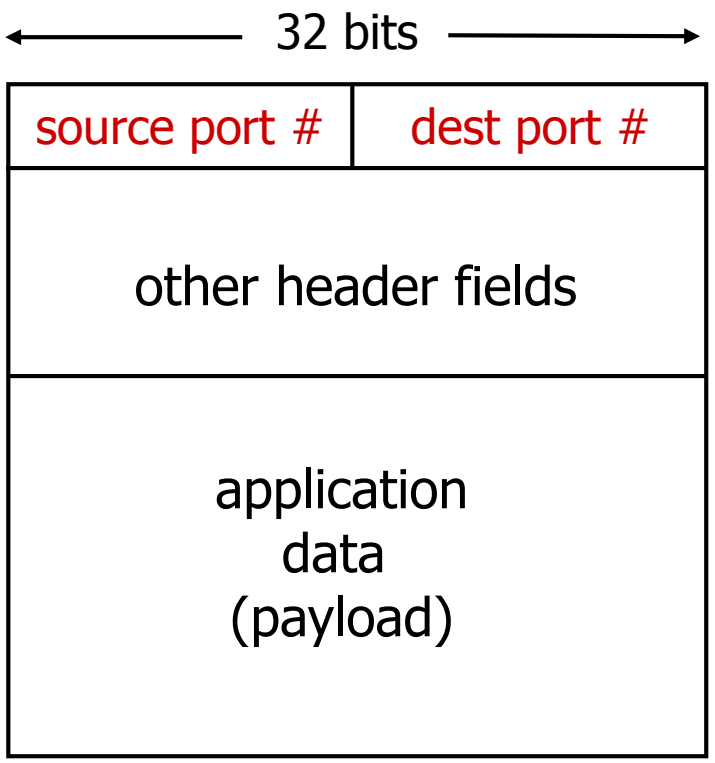
*multiplexing at sender:*
handle data from multiple sockets, add transport header (later used for demultiplexing)

*demultiplexing at receiver:*
use header info to deliver received segments to correct socket



# How demultiplexing works

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

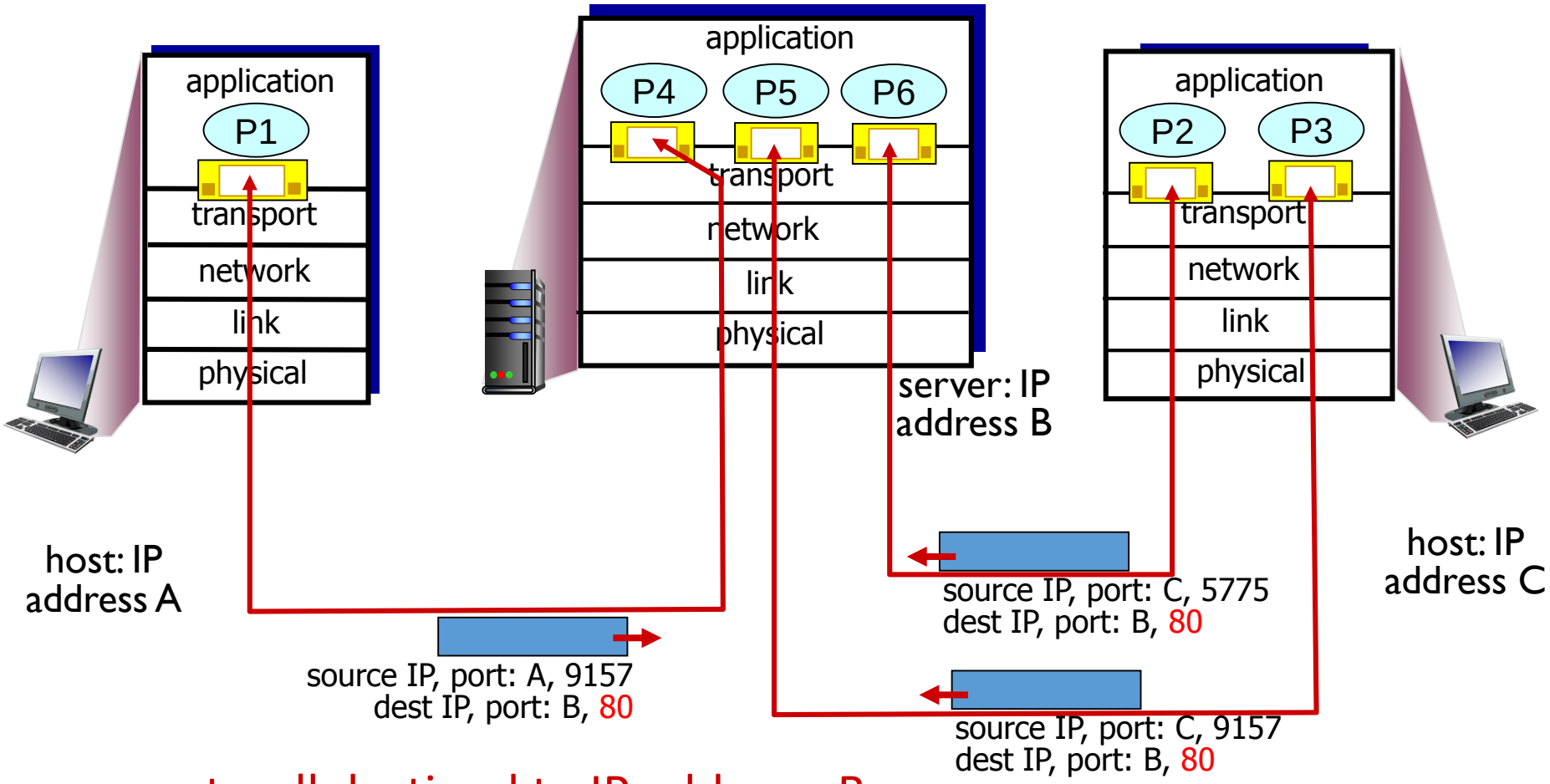



TCP segment format

# TCP demux

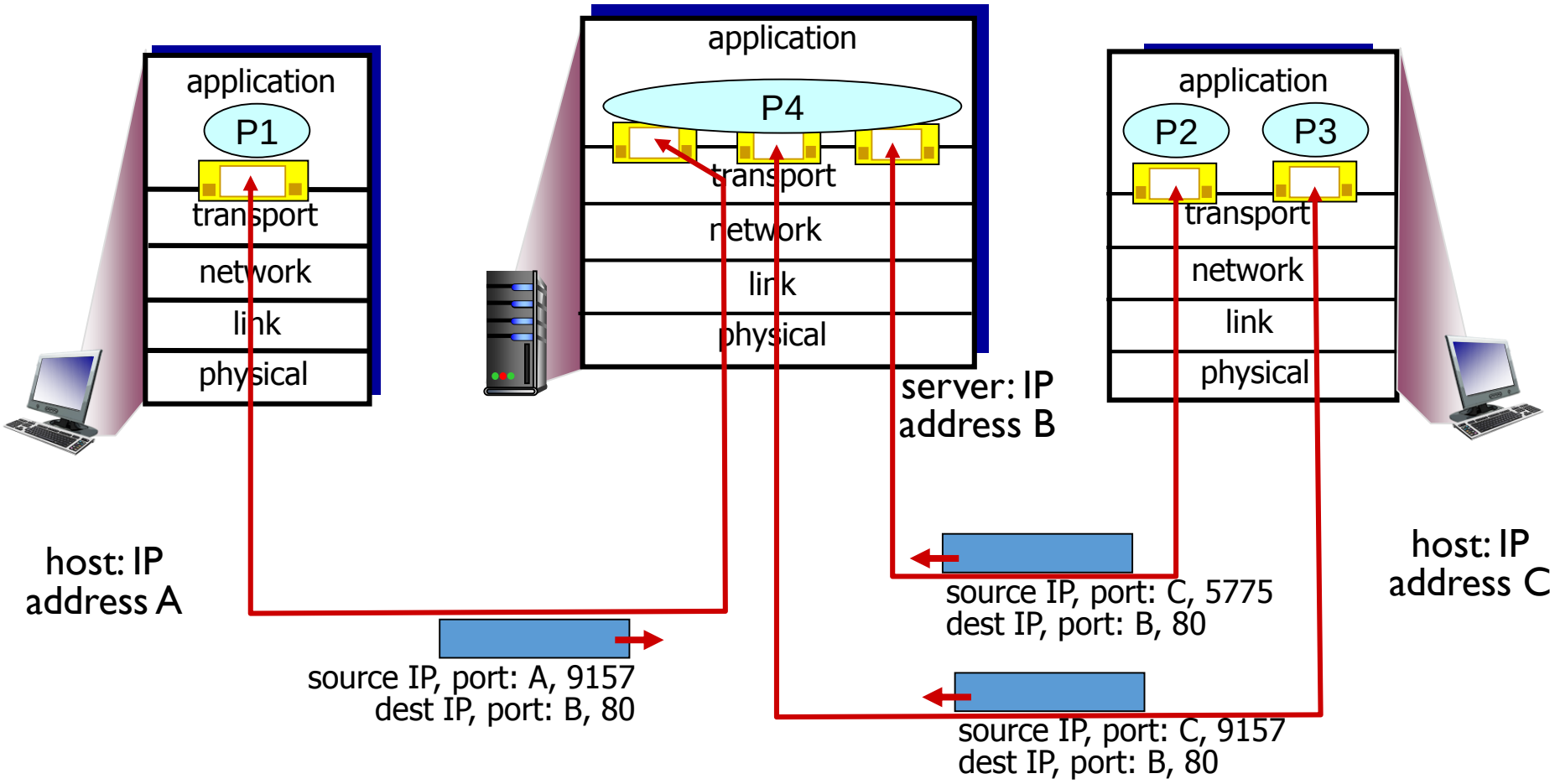
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Demux: receiver uses all four values to direct segment to appropriate socket

# TCP demux example




three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

# TCP demux example



# Reliable communication

# Reliable communication

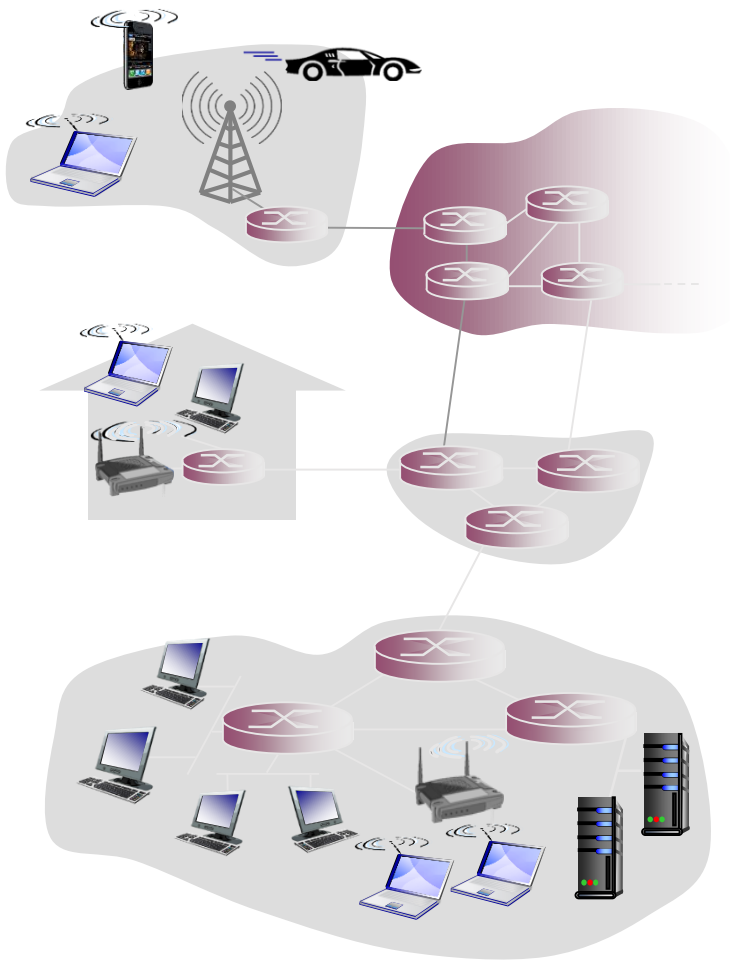
PROBLEM: unreliable channel

Packet corruption

Losses

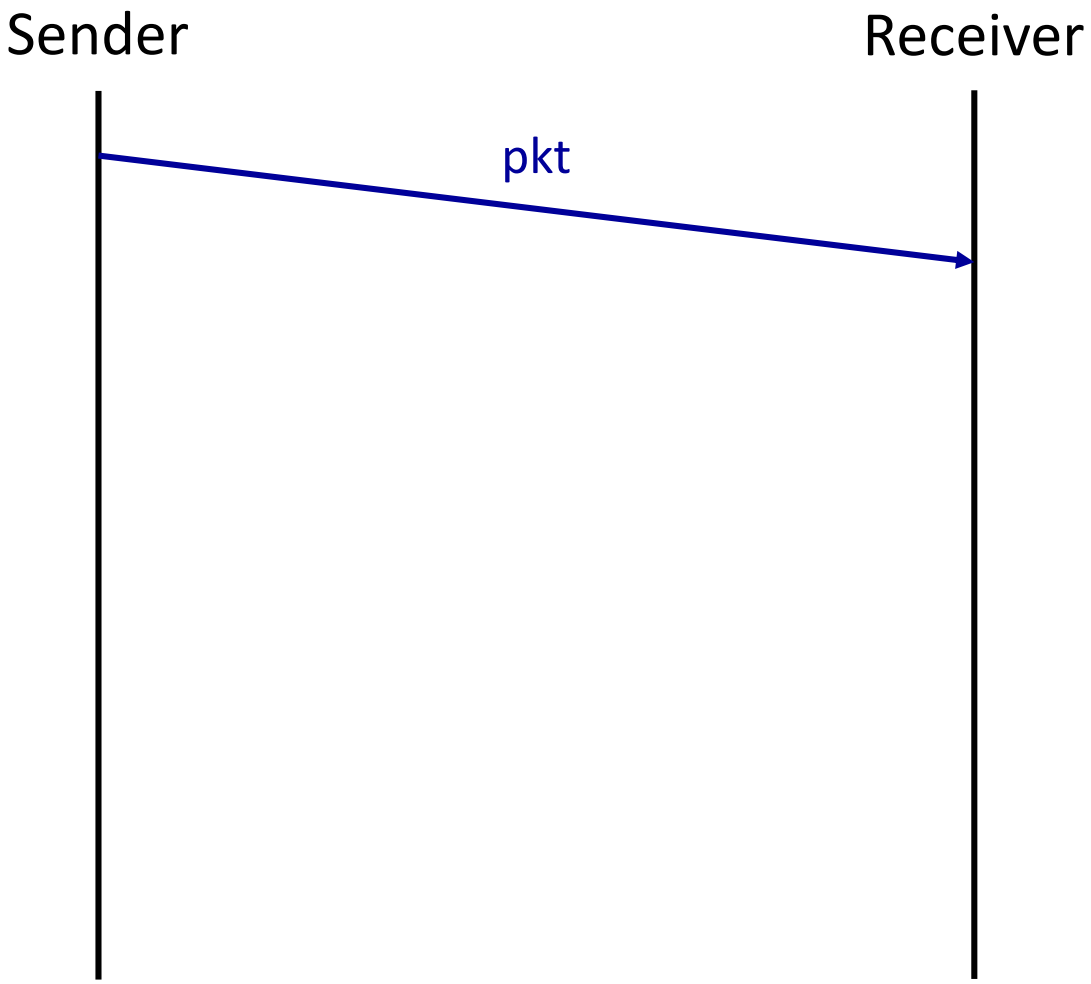
Unordered delivery

How to establish a reliable communication?



# Reliable communication

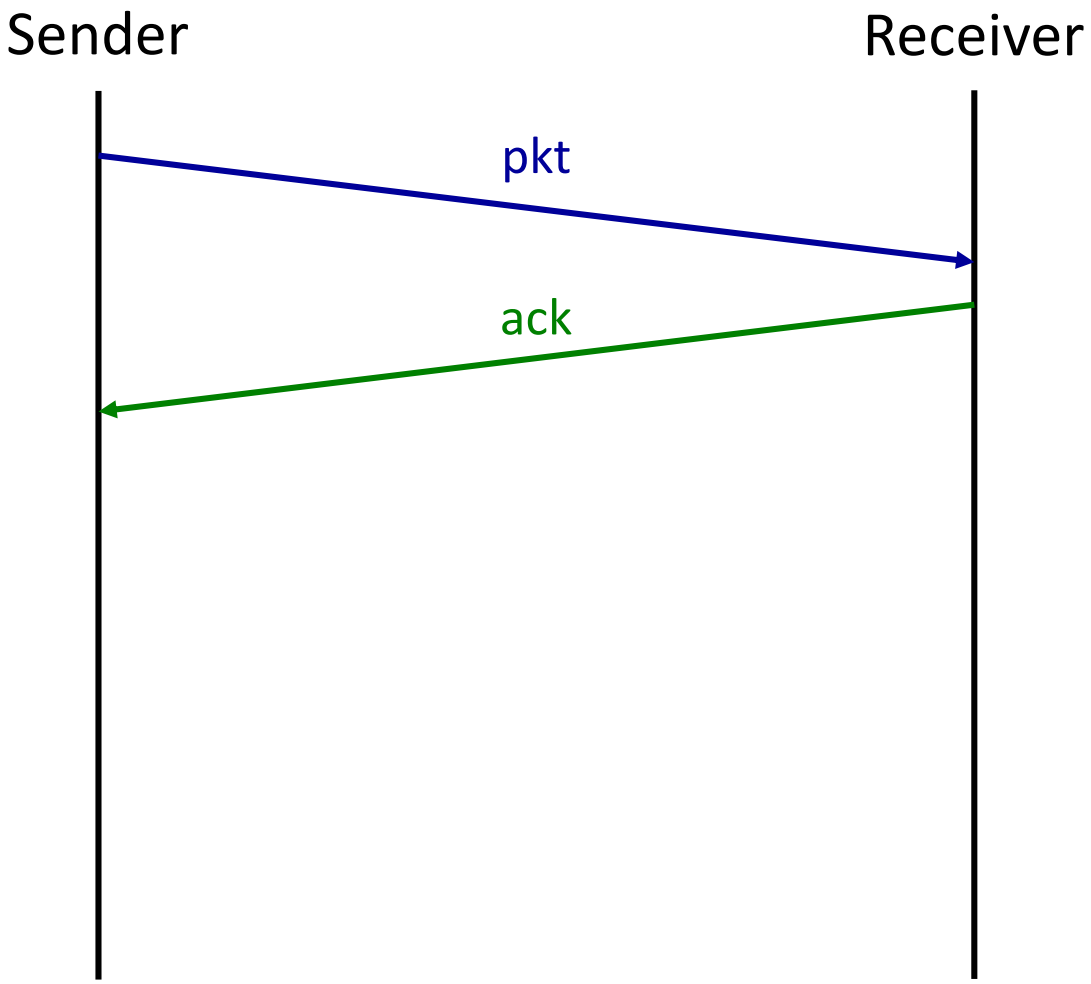
Q: How does the sender know if the message has been received?



# Reliable communication

Q: How does the sender know if the message has been received?

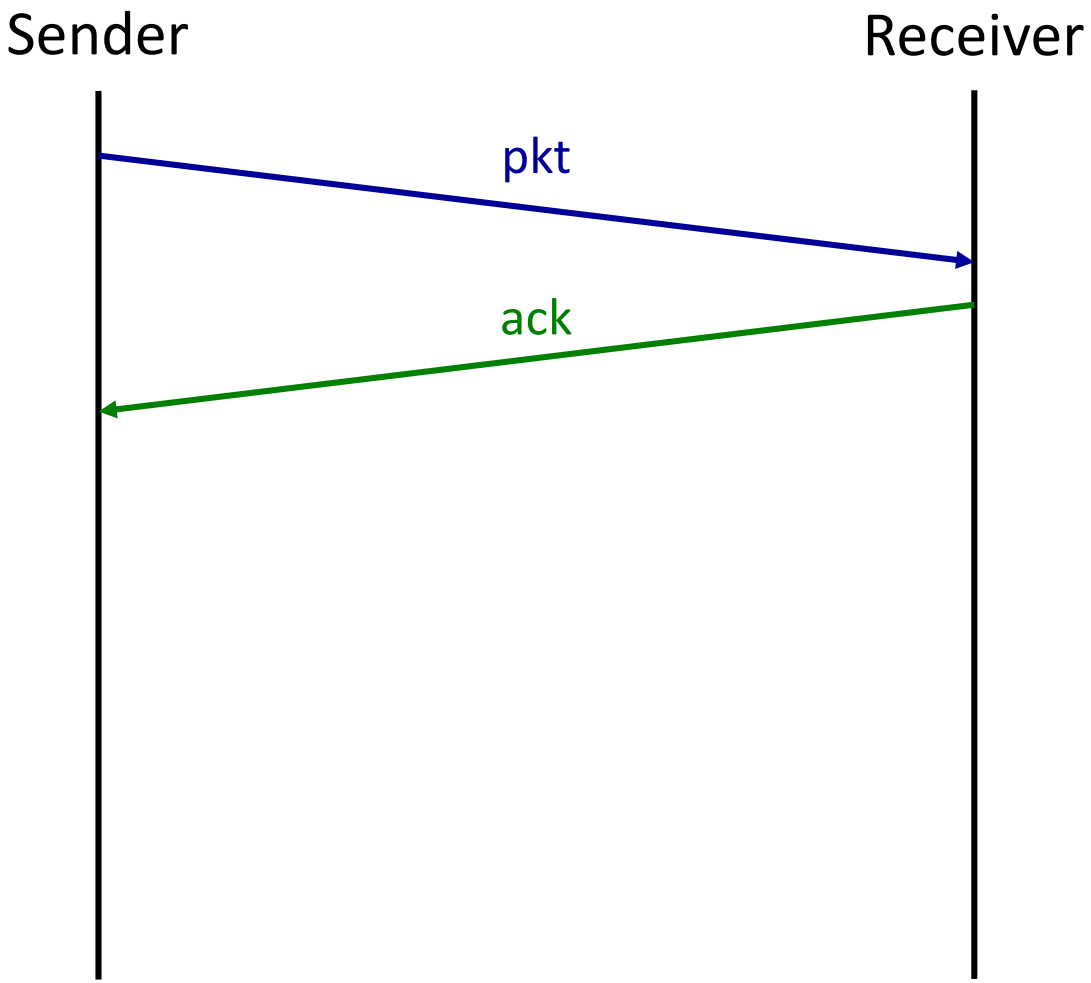
A: Feedback from receiver, ACK



# Reliable communication

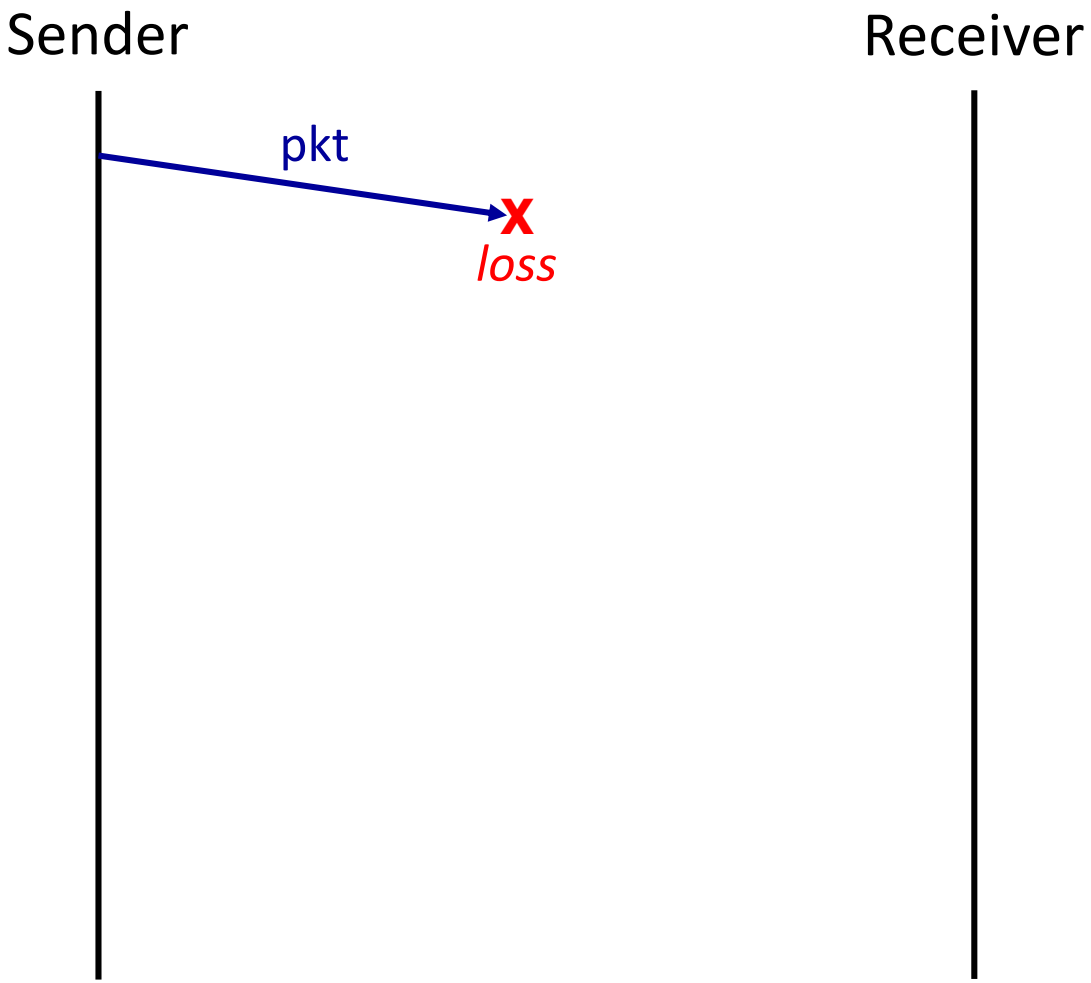
**stop and wait**

sender sends one packet, then waits for receiver response



# Reliable communication

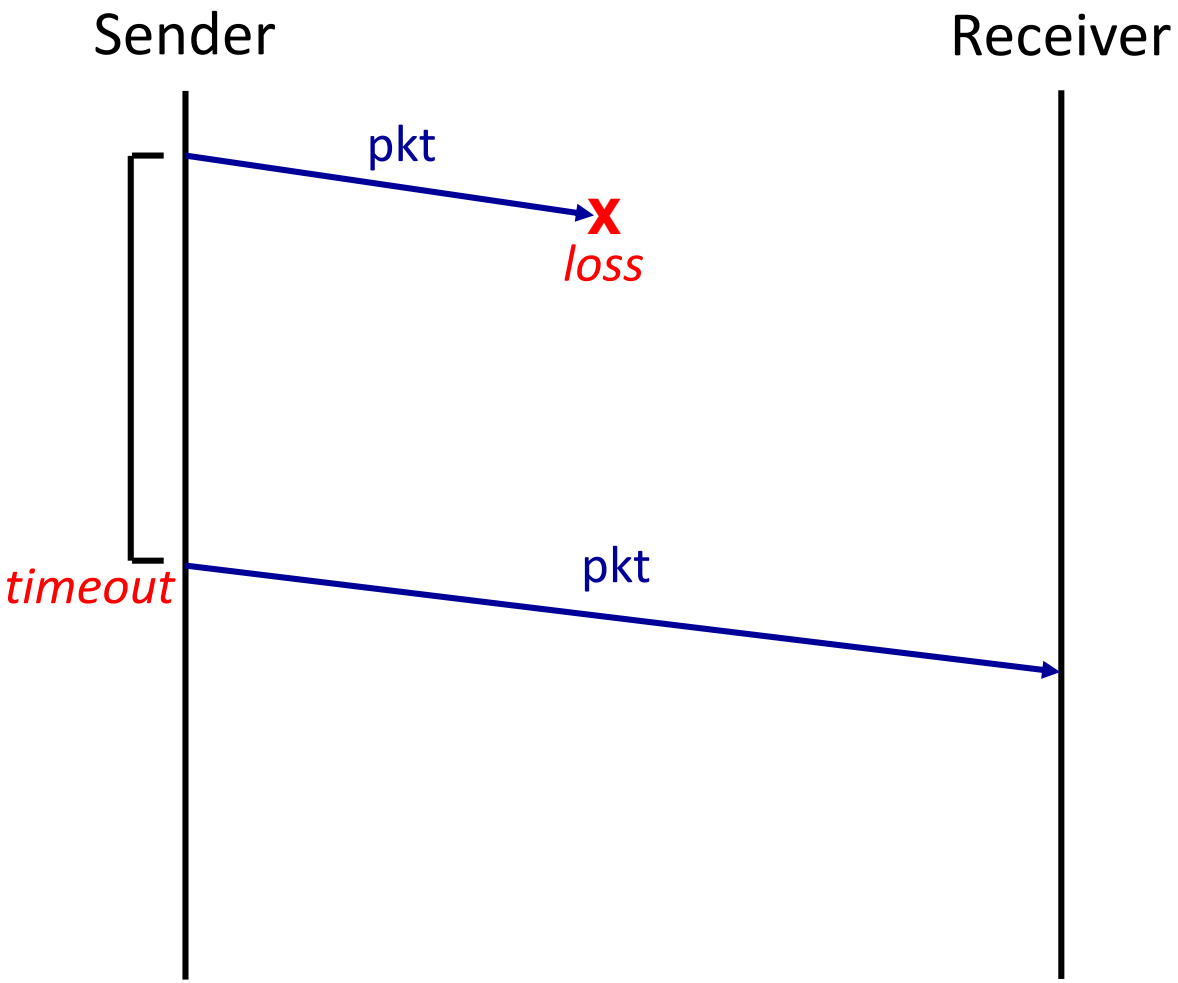
Q: What if the message is lost?



# Reliable communication

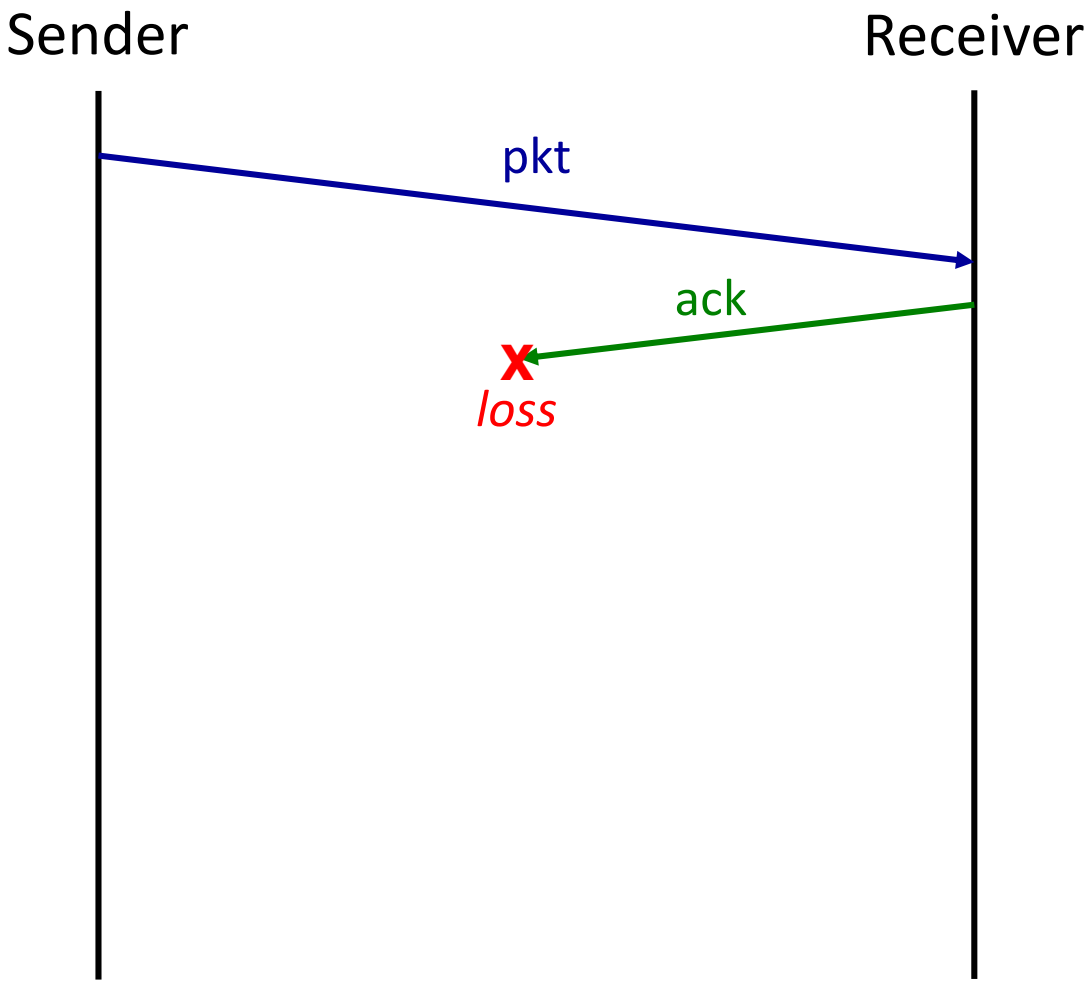
Q: What if the message is lost?

A: Timeout and retransmission



# Reliable communication

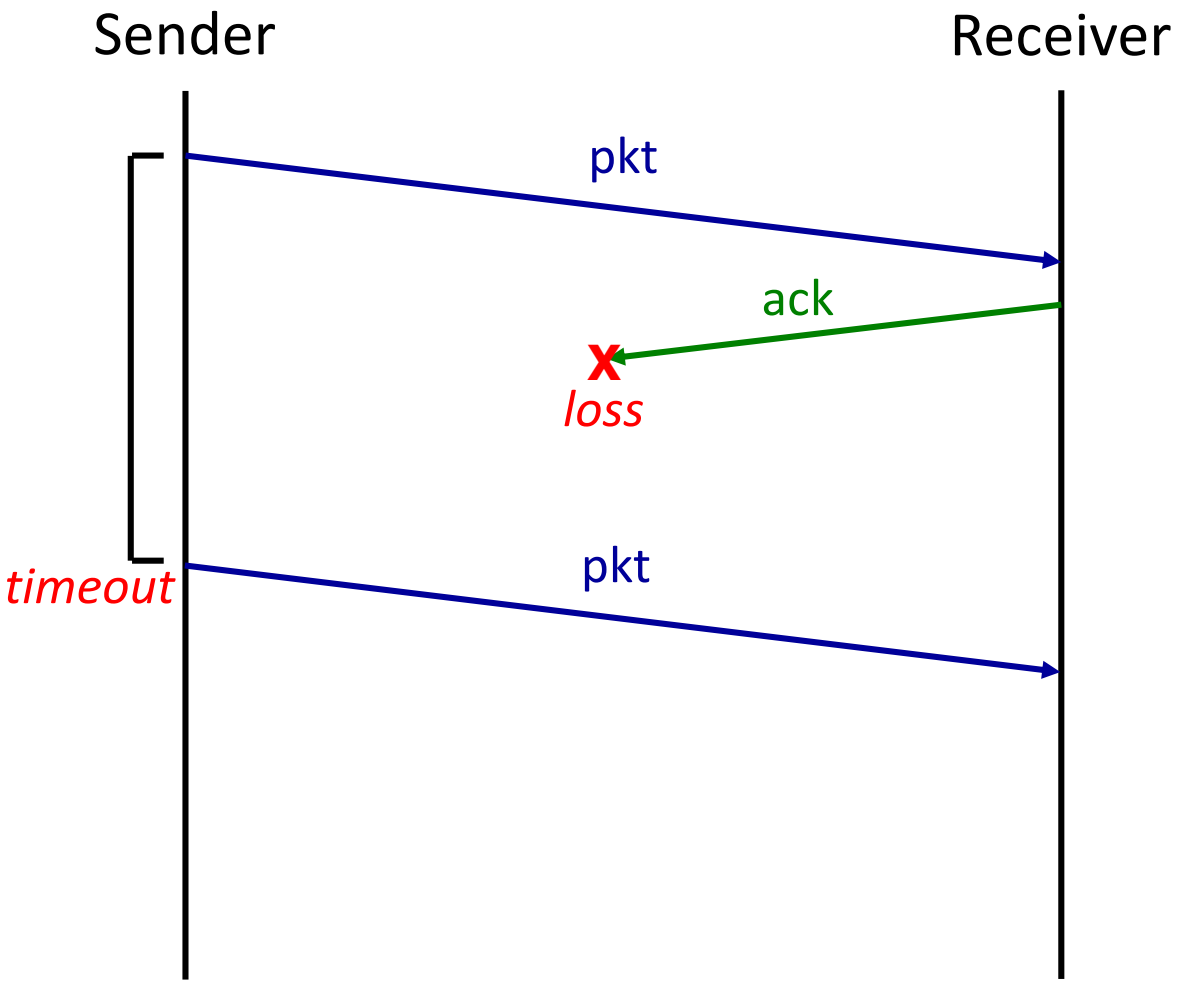
Q: What if ACK is lost?



# Reliable communication

Q: What if ACK is lost?

PROBLEM: how does the receiver know that it is a duplicate?

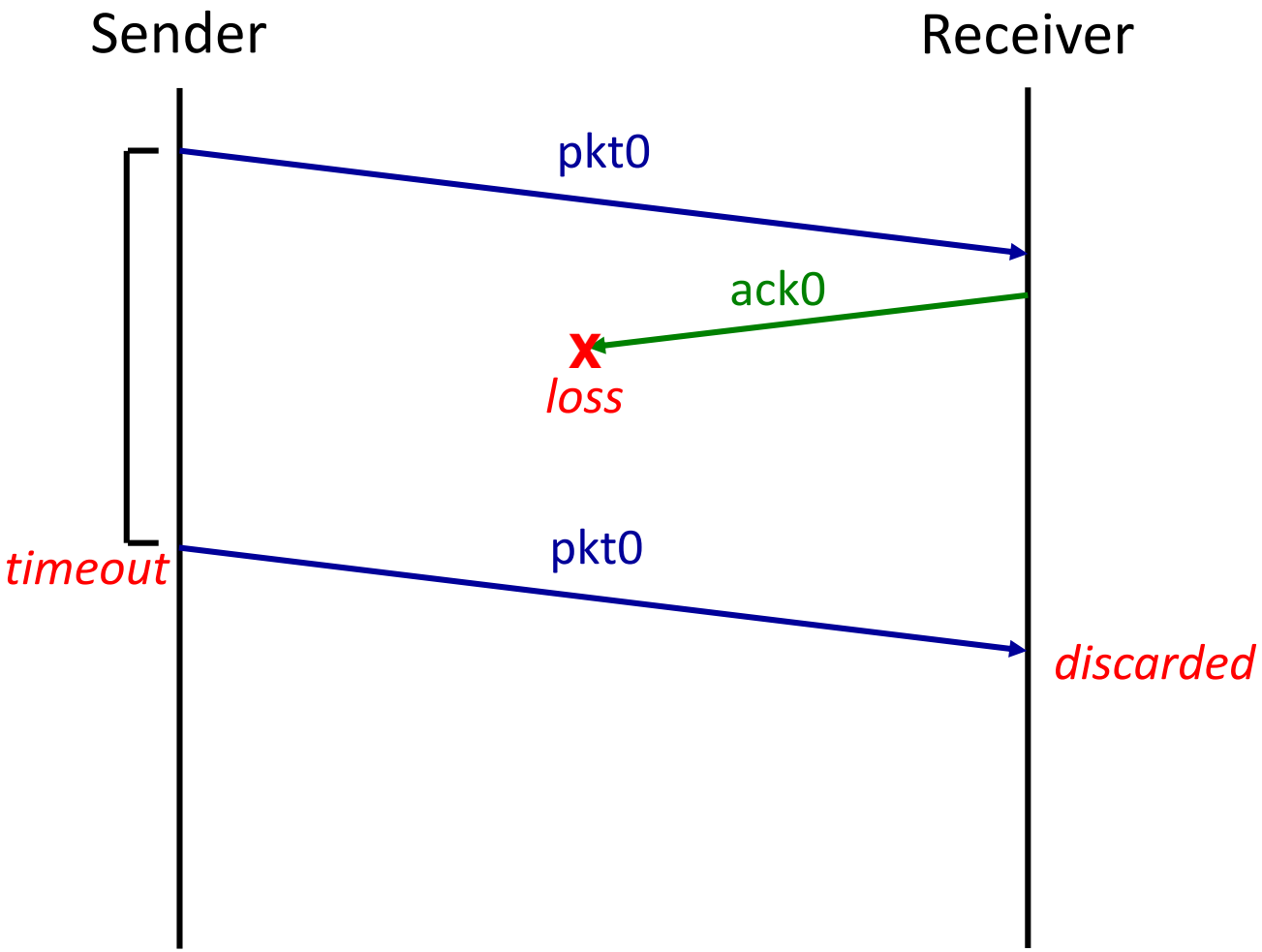


# Reliable communication

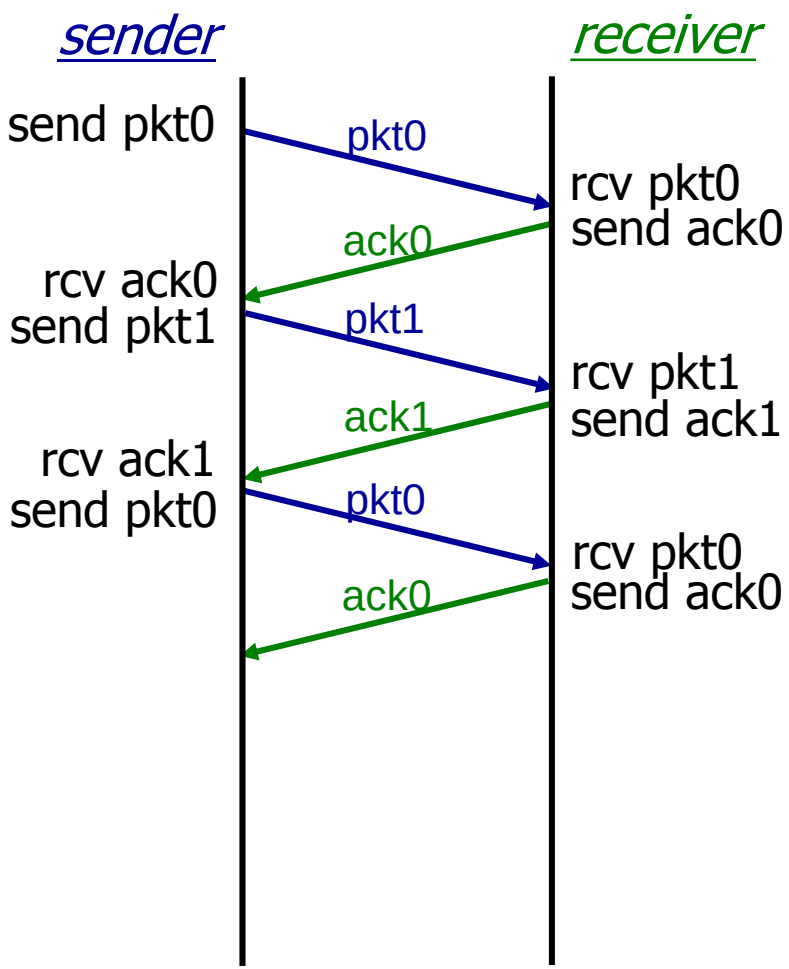
Q: What if ACK is lost?

PROBLEM: how does the receiver know that it is a duplicate?

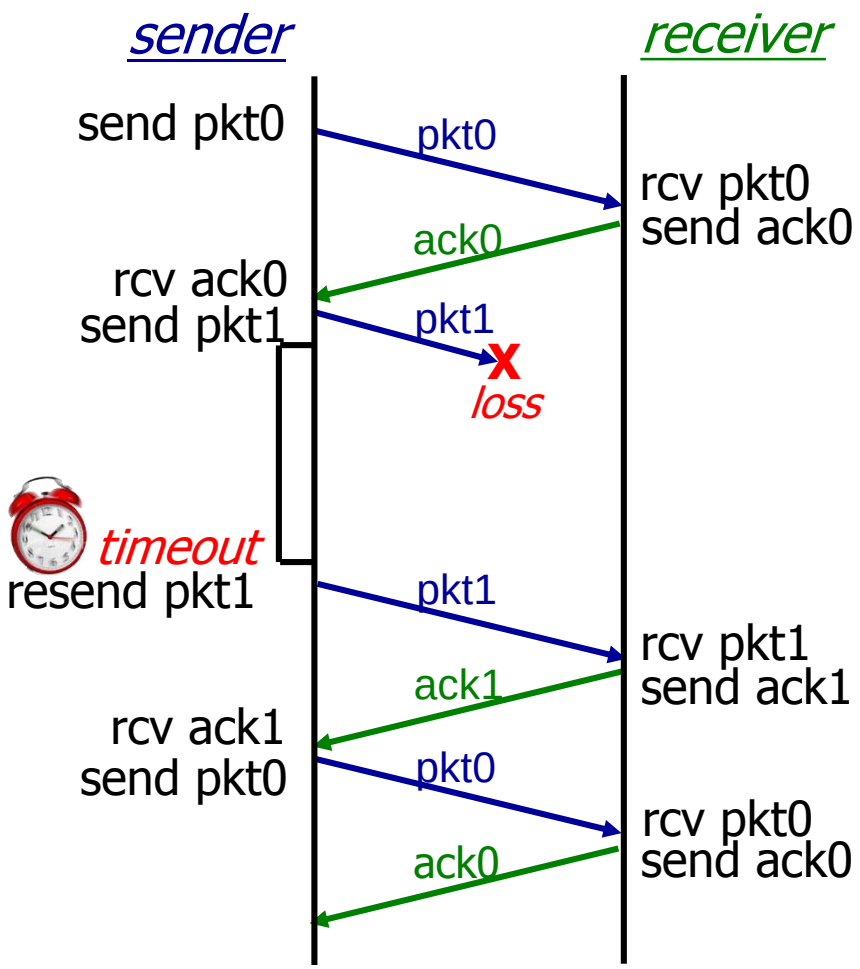
SOLUTION: Sequence number



# Stop-and-wait recap

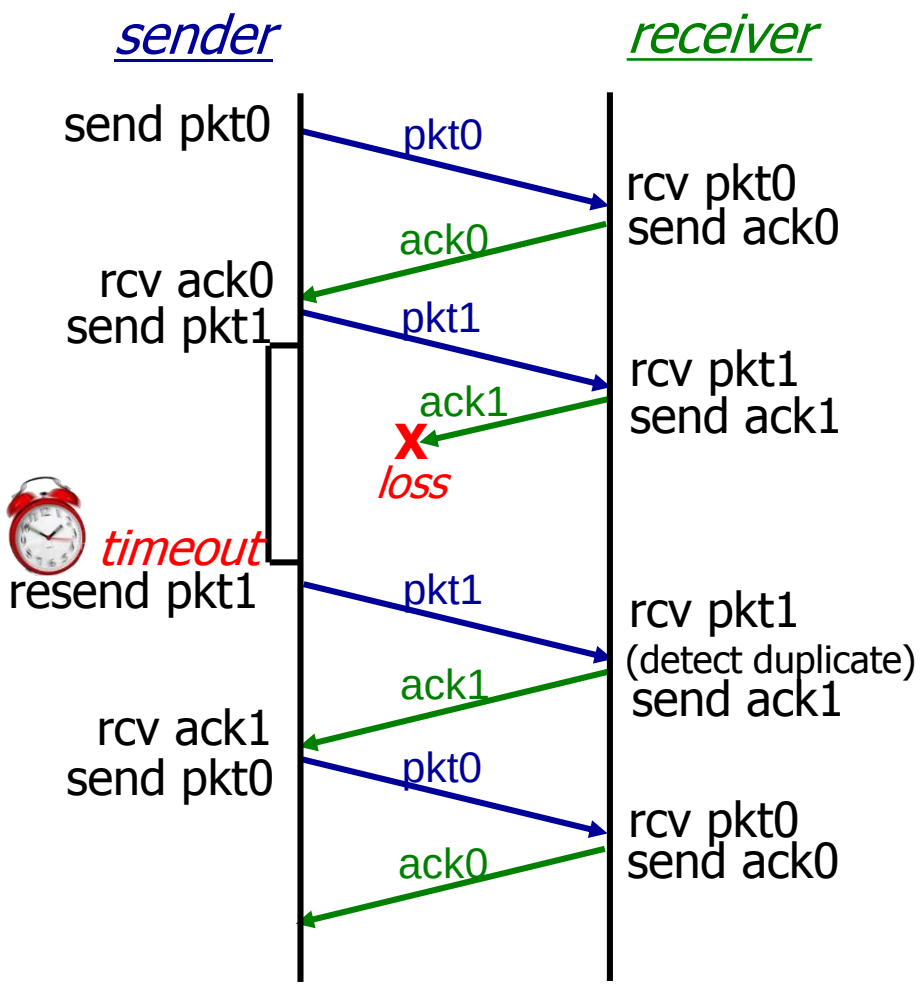



(a) no loss

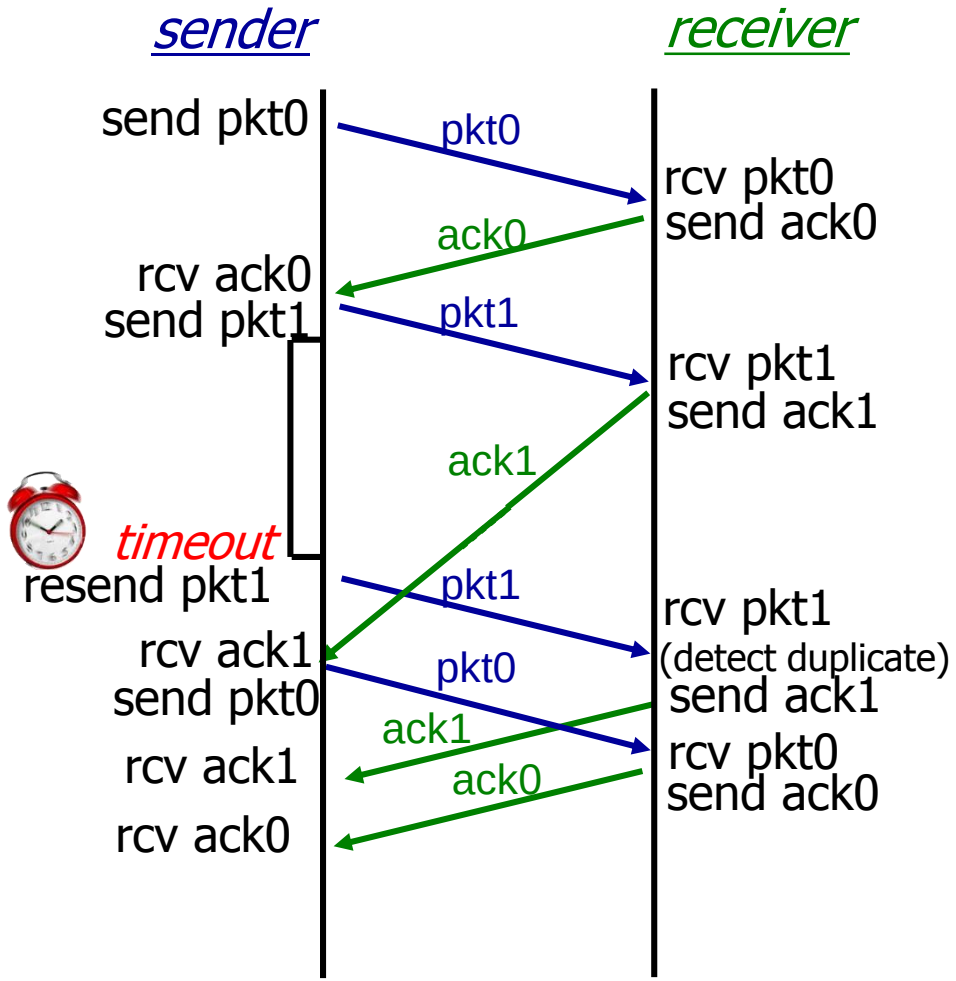



(b) packet loss

# Stop-and-wait recap




(c) ACK loss



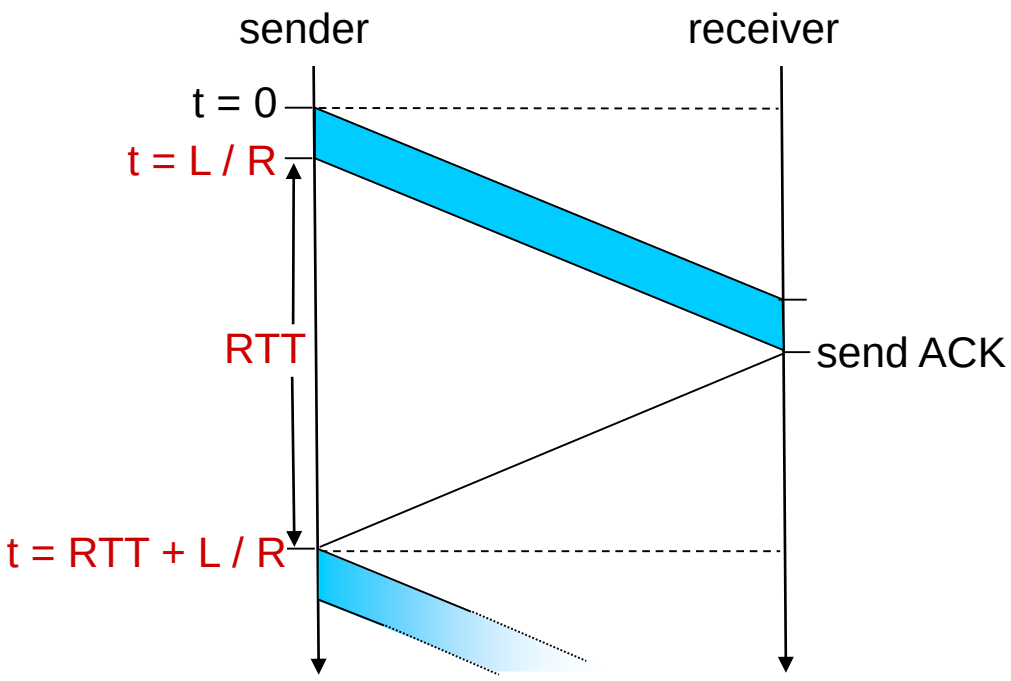

(d) premature timeout/ delayed ACK

# Stop-and- wait utilization

Example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

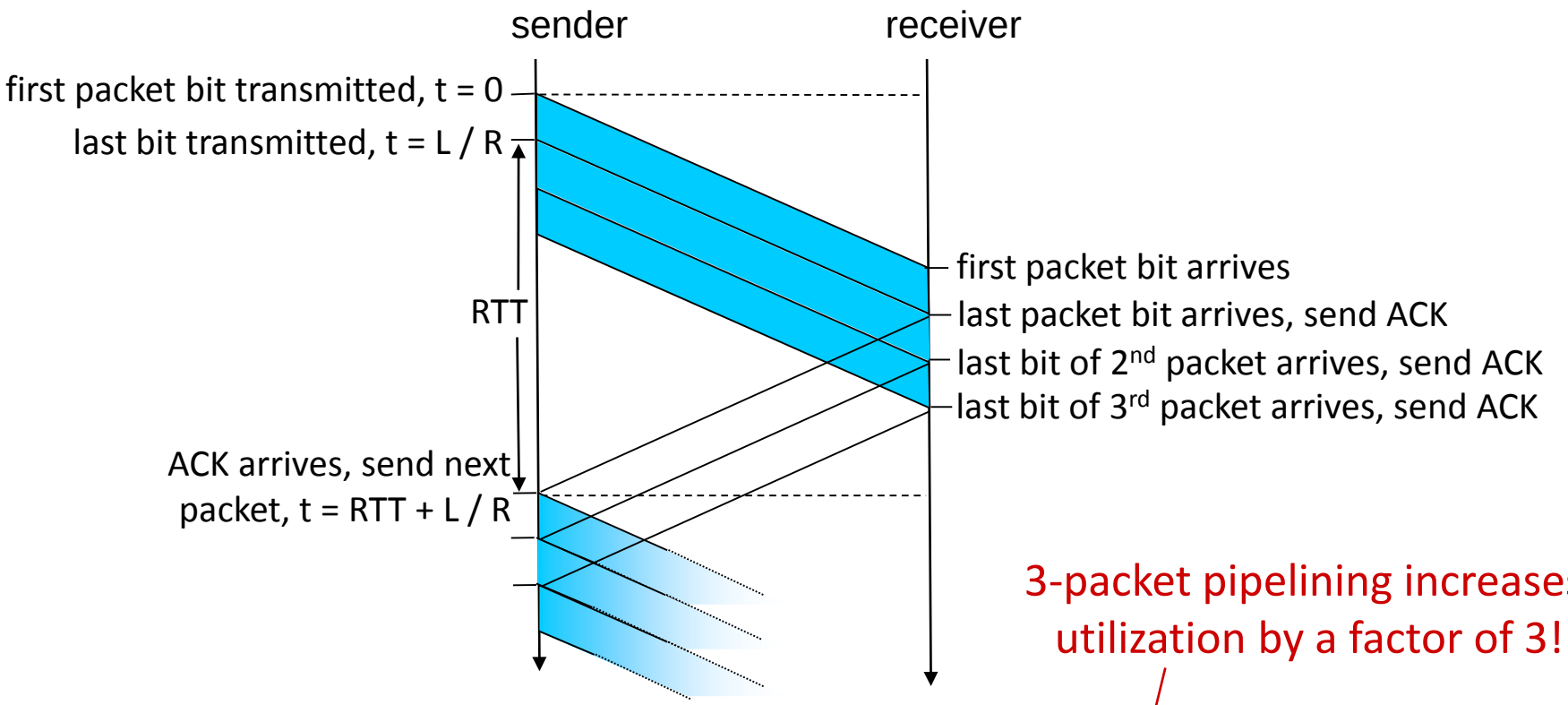
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

*Utilization* – fraction of time sender busy sending




If RTT=30 ms, 1kB pkt every 30 ms:
33kB/s throughput

# Pipelining: increased utilization

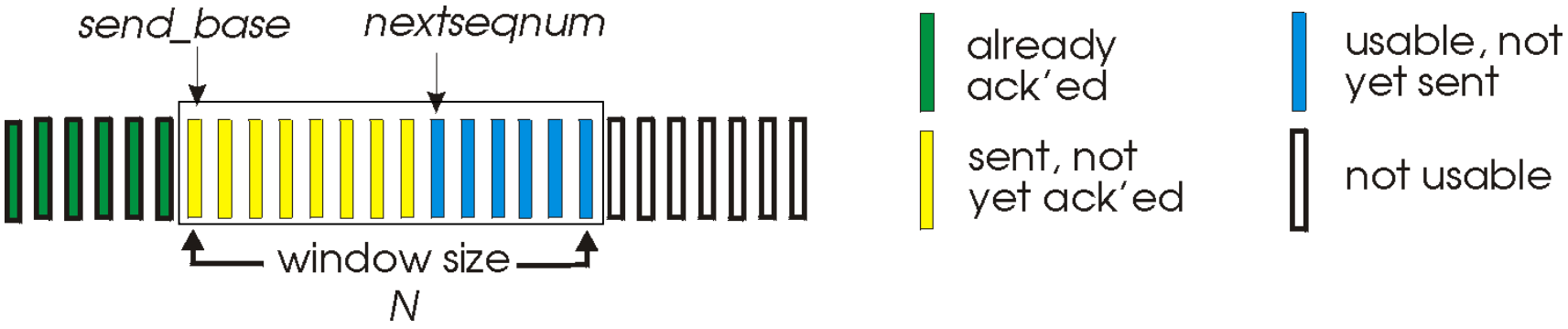



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.00081$$

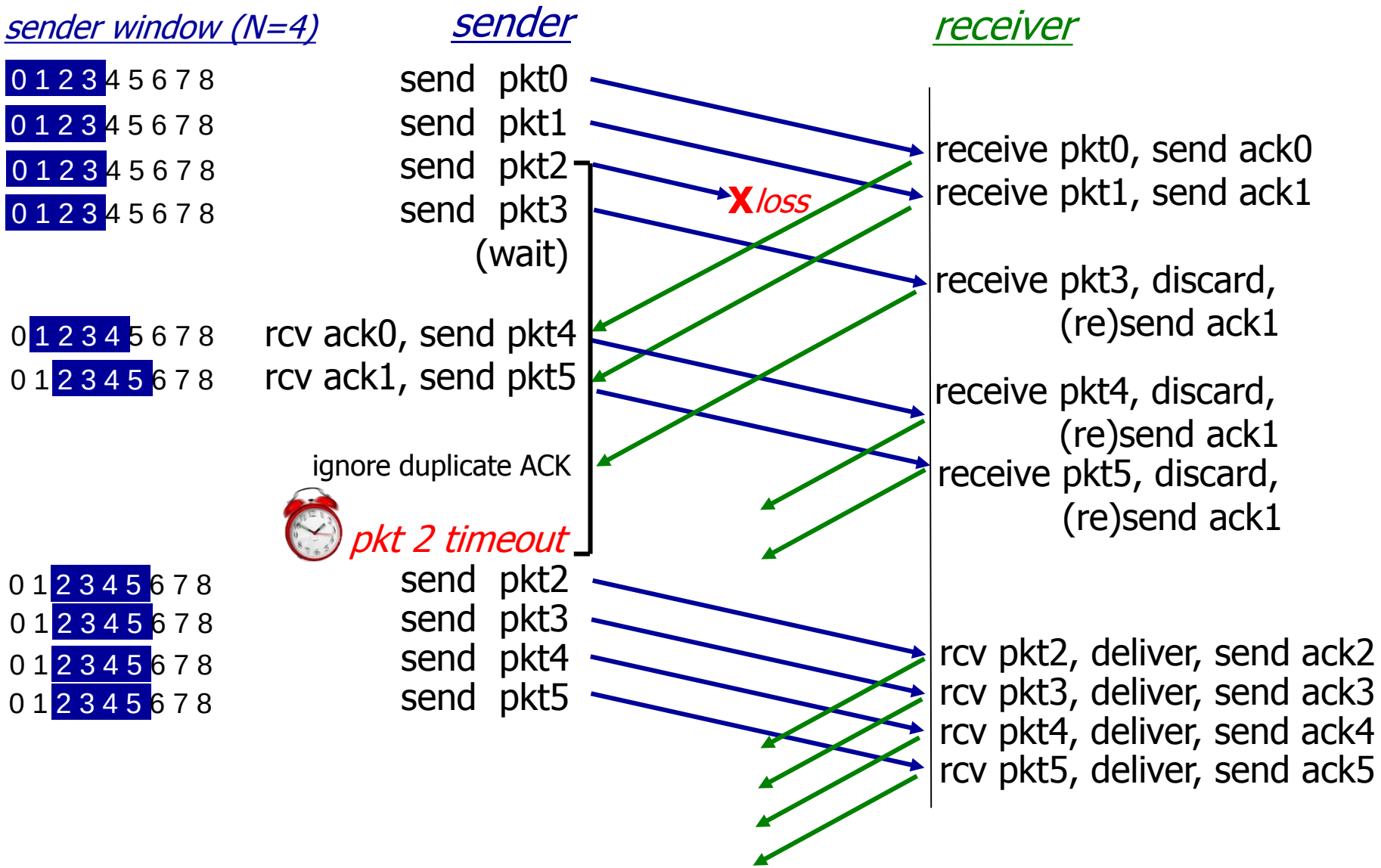
# Go-Back-N

Sliding window of up to N, consecutive unacked pkts allowed




- Cumulative ACK ACK(n): ACKs all pkts up to, including seq # n
  - Sender may receive duplicate ACKs – ignored
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window
- out-of-order pkts:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

# Go-Back-N example



# TCP reliable communication

- TCP obtains reliable communication 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
- ignore flow control, congestion control

# TCP seq. numbers, ACKs

sequence numbers:

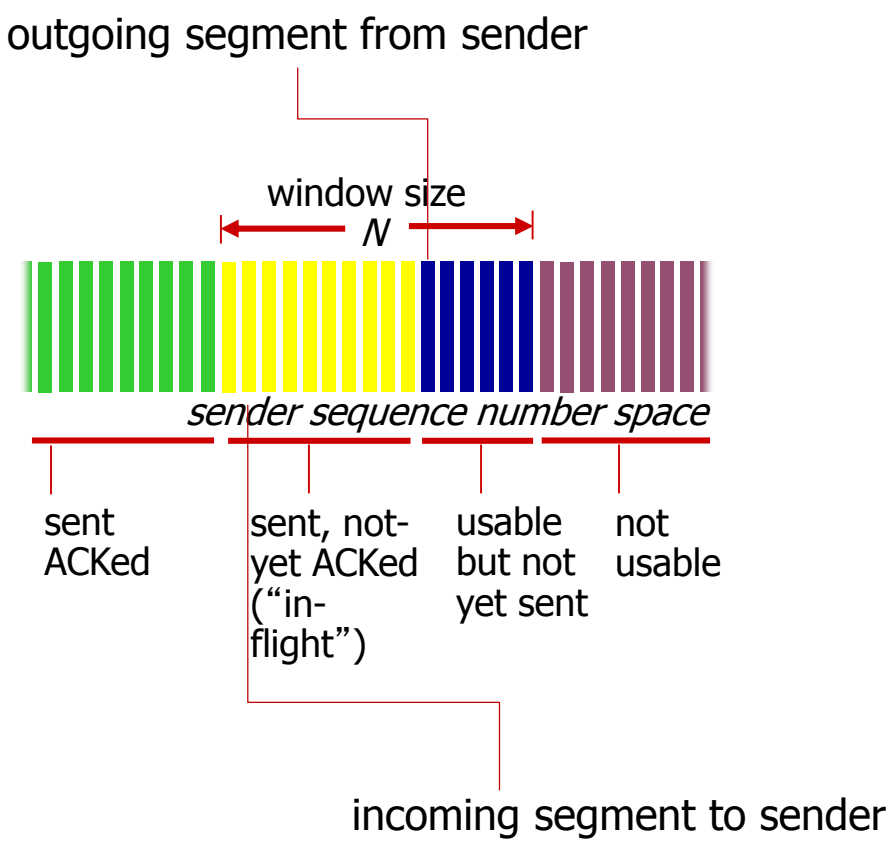
- byte stream "number" of first byte in segment's data

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 sender

*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

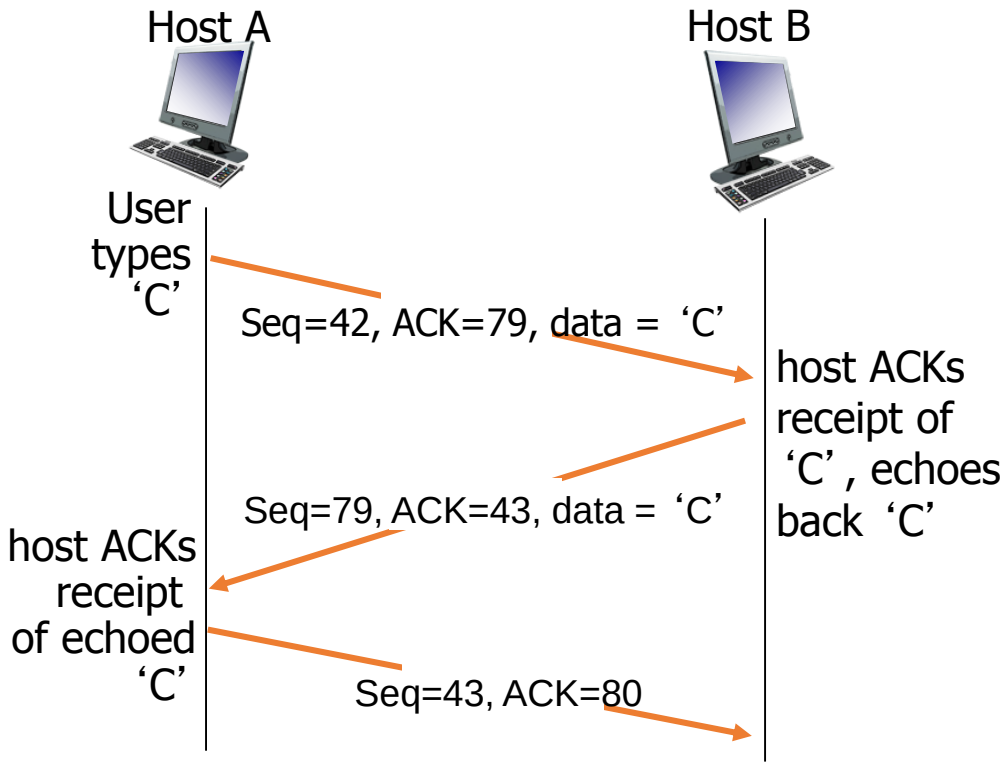
*timeout:*

- retransmit segment that caused timeout
- restart timer

*ack rcvd:*

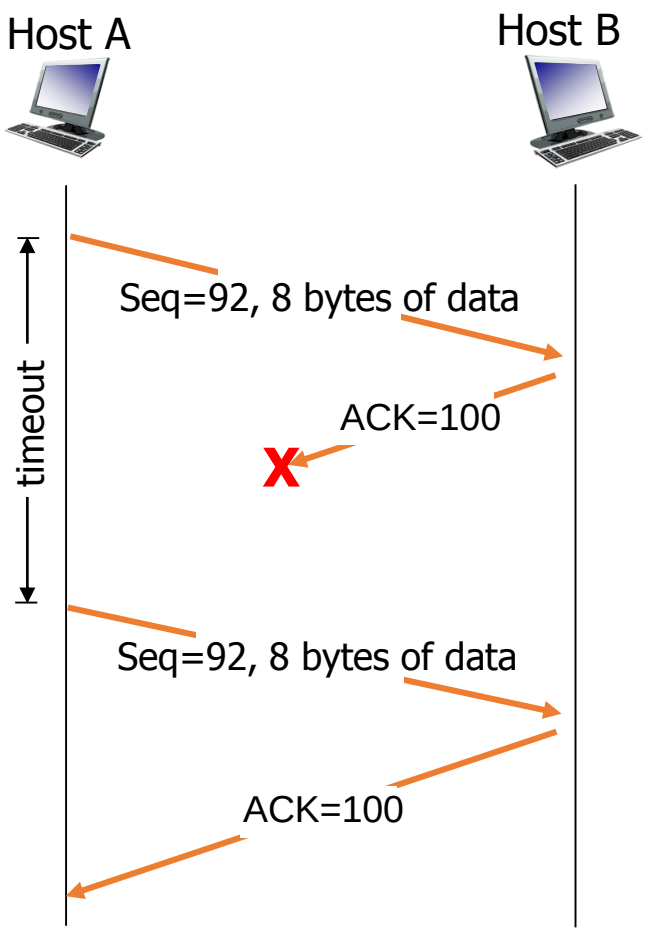
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

# TCP seq. numbers, ACKs

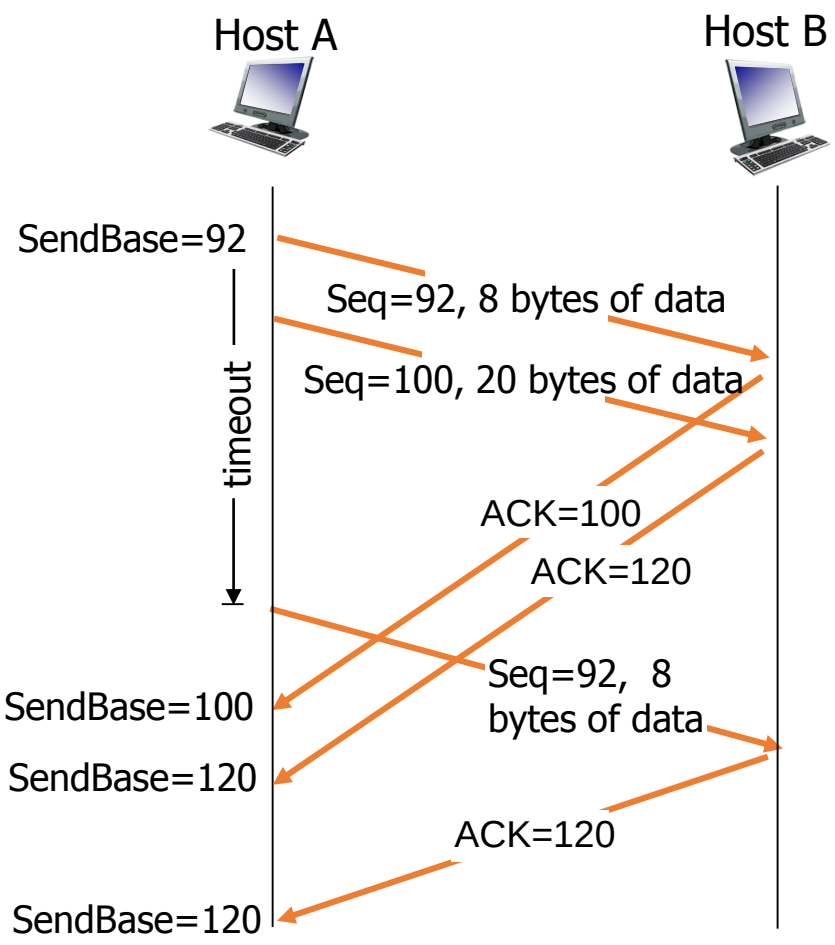



simple telnet scenario

# TCP: retransmission scenarios

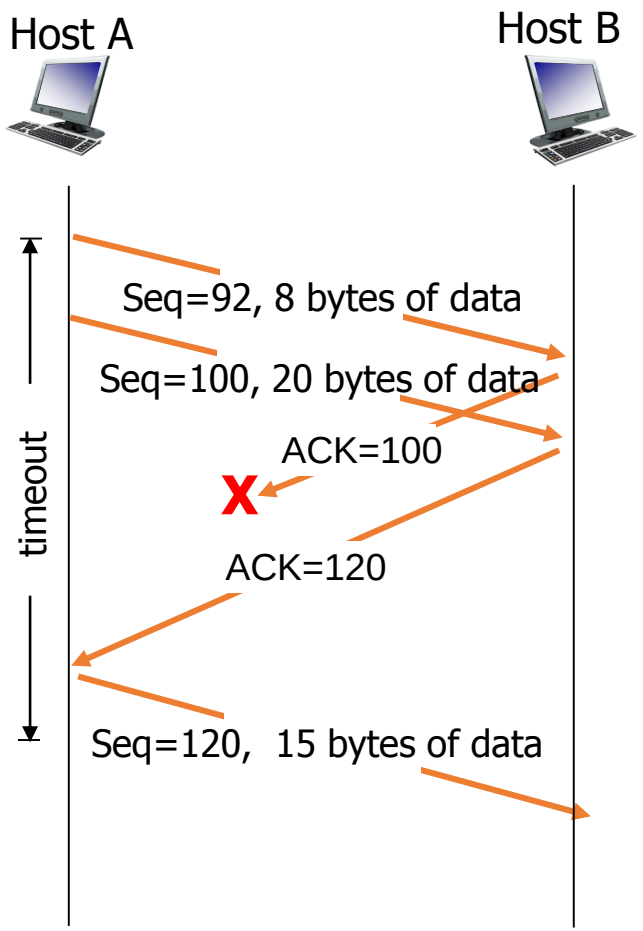



lost ACK scenario




premature timeout

# TCP: retransmission scenarios




cumulative ACK

# TCP ACK generation [RFC 1122, RFC 2581]

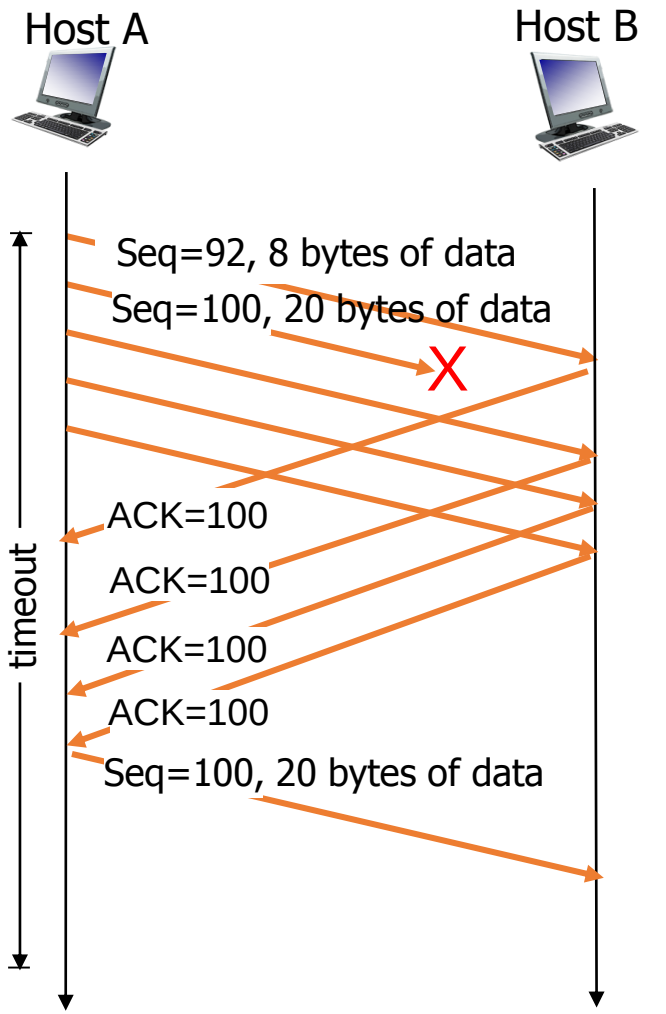
| *Event at receiver* | *TCP receiver action* |
|---|---|
| arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK |
| arrival of in-order segment with expected seq #. One other segment has ACK pending | immediately send single cumulative ACK, ACKing both in-order segments |
| 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 starts at lower end of gap |

# 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




fast retransmit after sender receipt of triple duplicate ACK

# 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

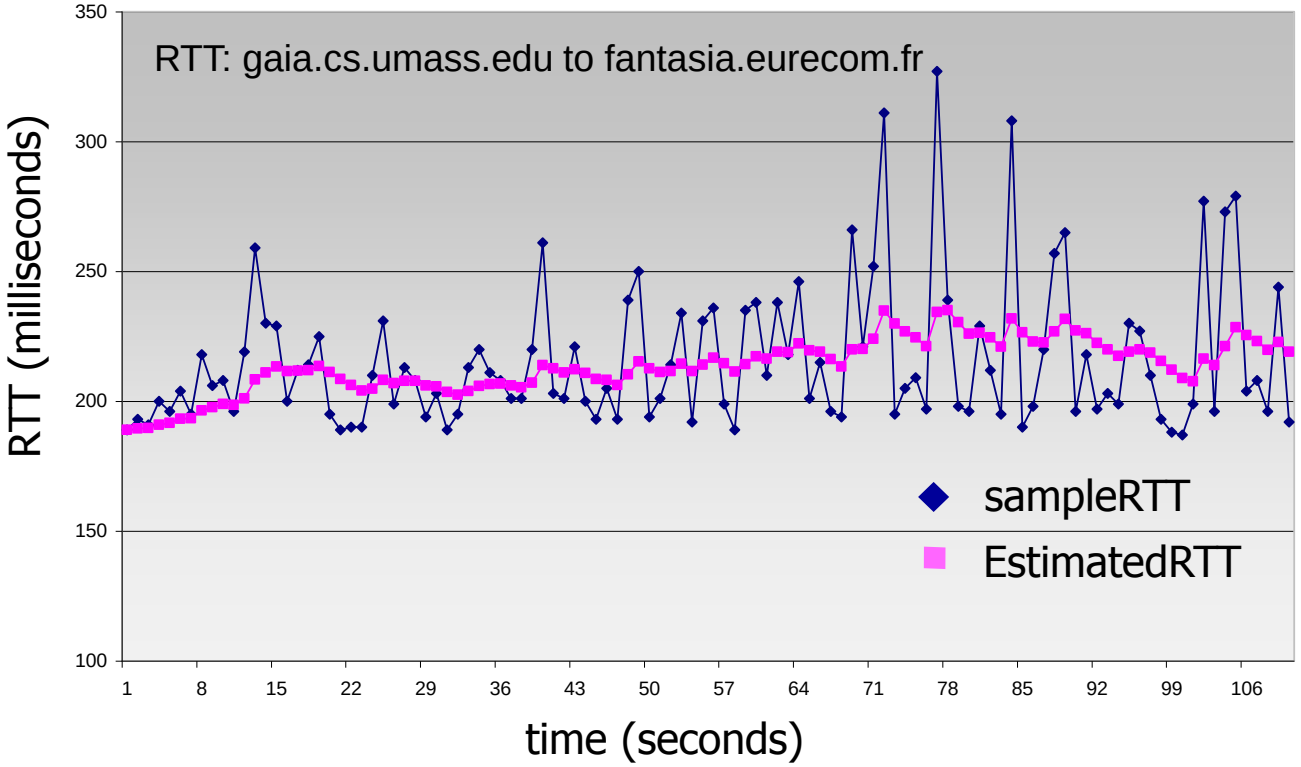
Q: how to estimate RTT?

- **`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`**

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



# TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** **`->`** larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

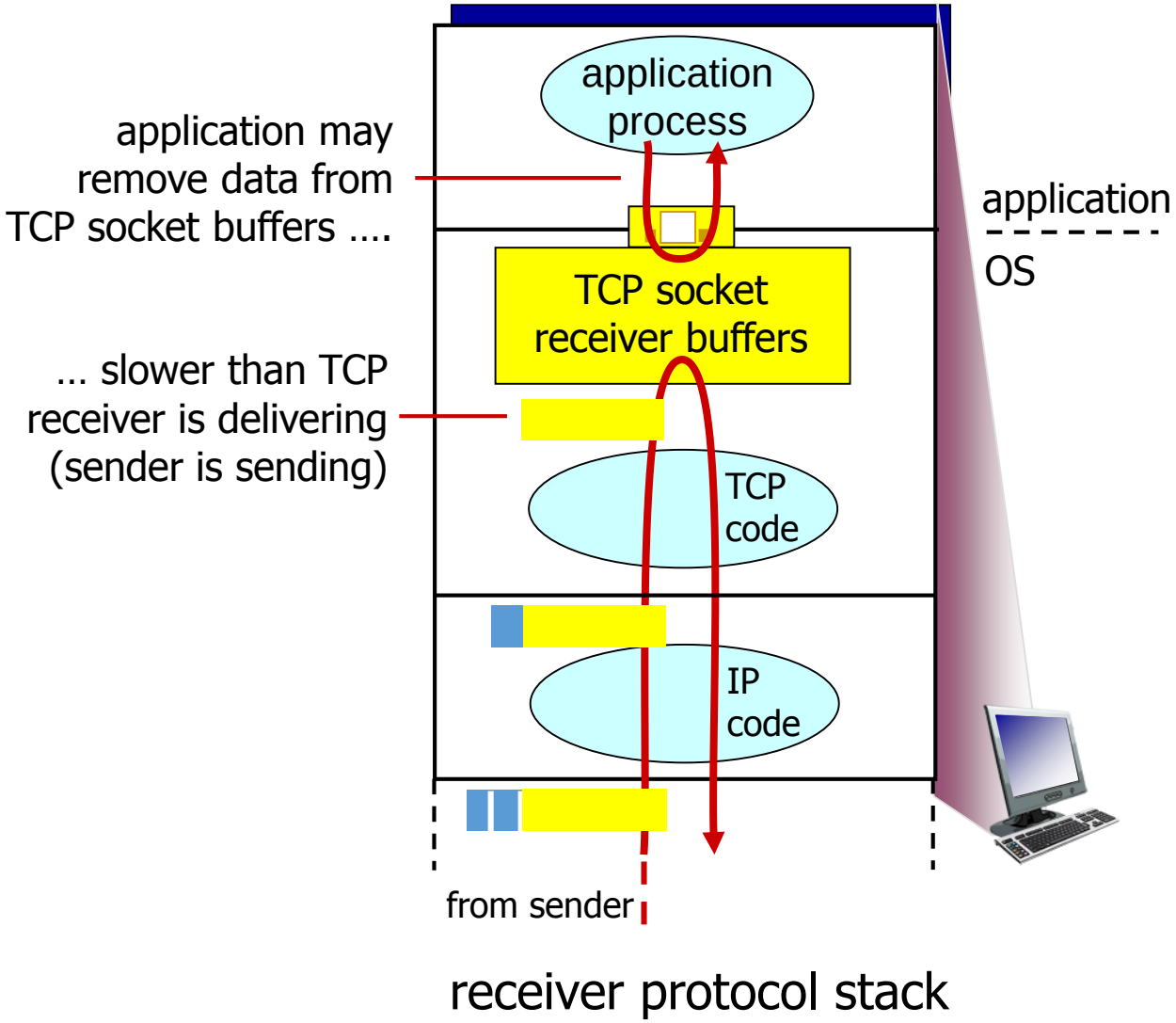
$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + 4 * \underbrace{\texttt{DevRTT}}_{\text{"safety margin"}}$$

# Flow control

# TCP flow control

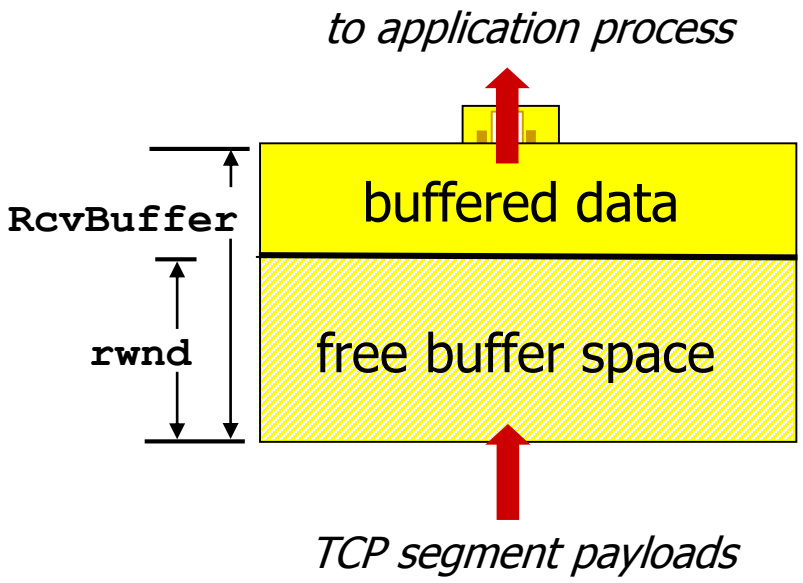
*flow control*

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



# TCP flow control

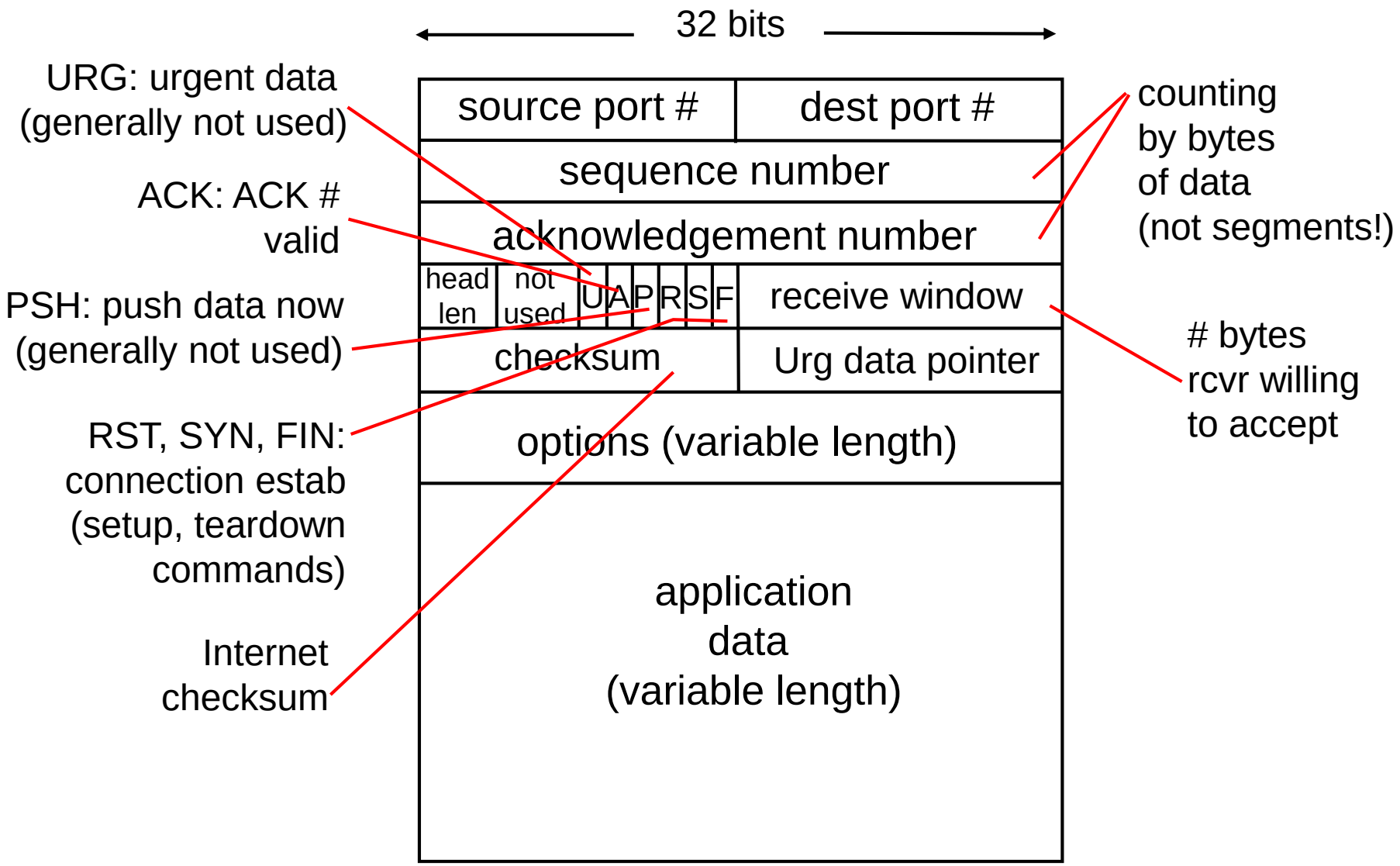
- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow




*receiver-side buffering*

# TCP segment structure

# TCP segment structure

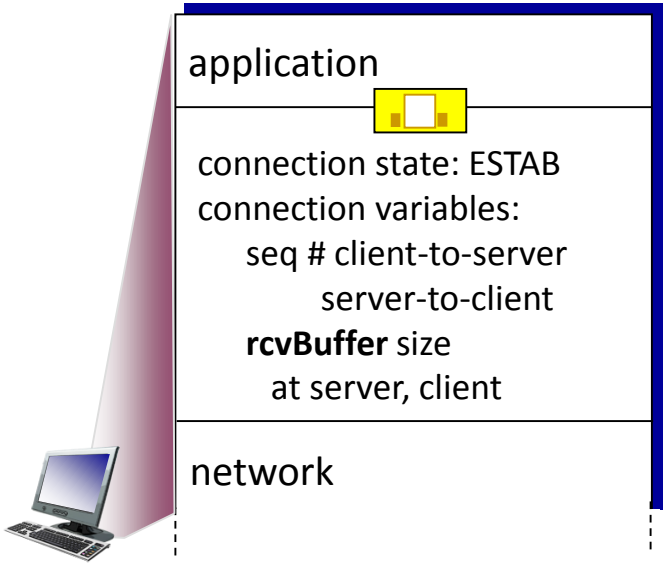


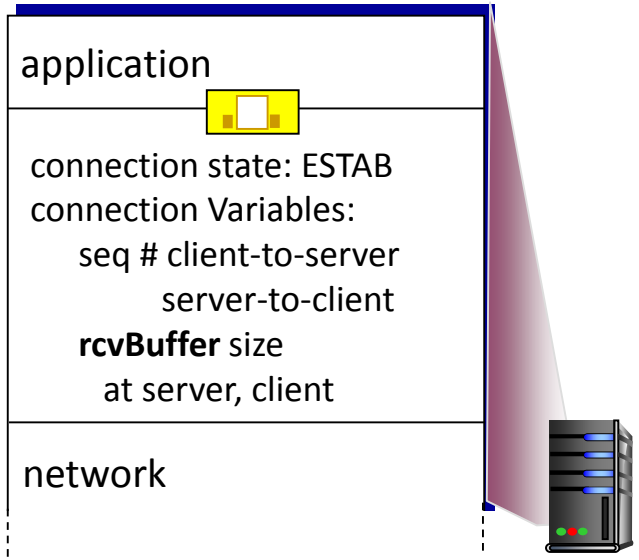
# Connection management

# Connection management

before exchanging data, sender/receiver "handshake":

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

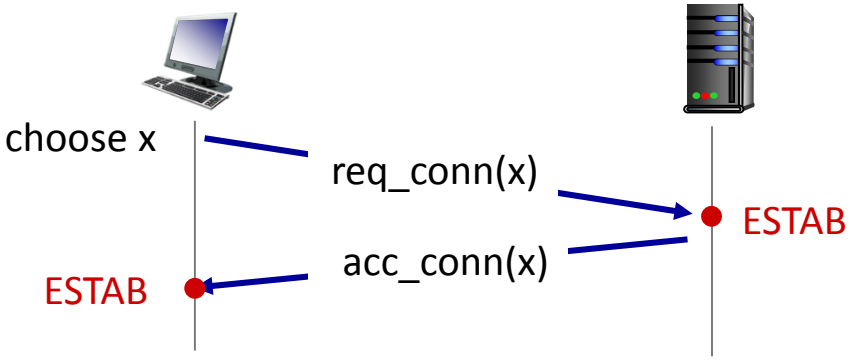



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

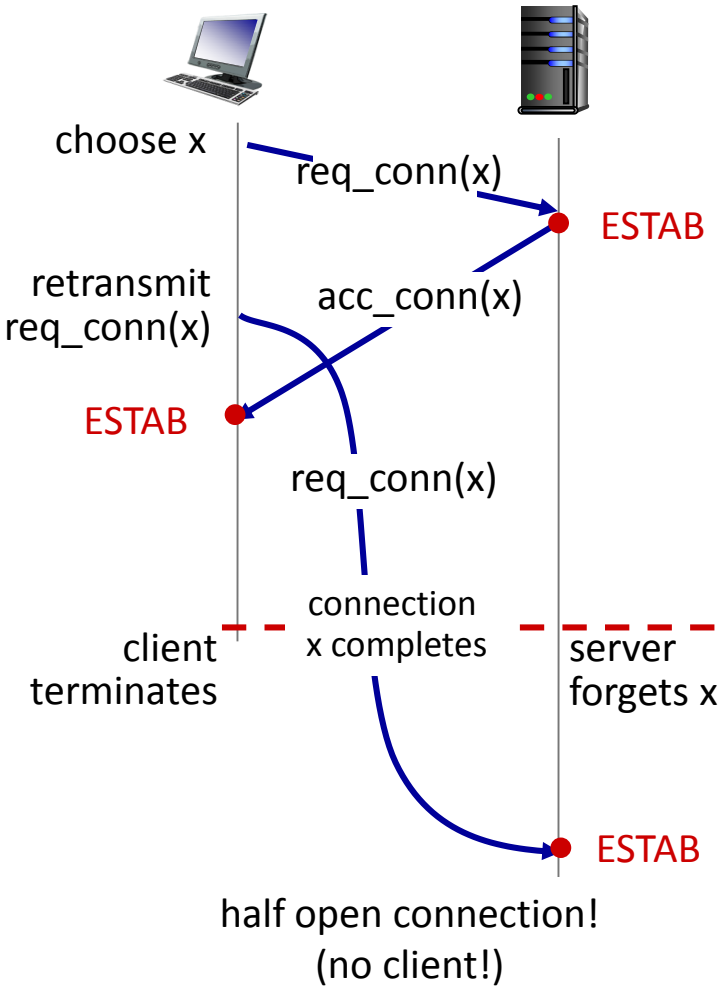
# 2-way handshake

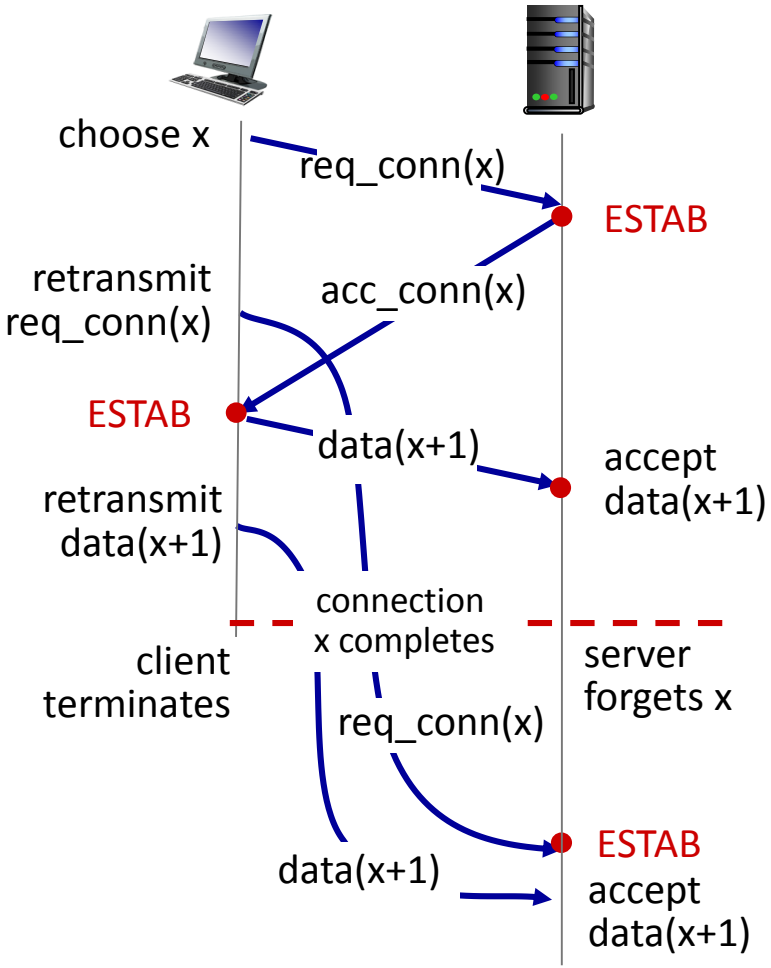



*Q:* will 2-way handshake always work in unreliable network?

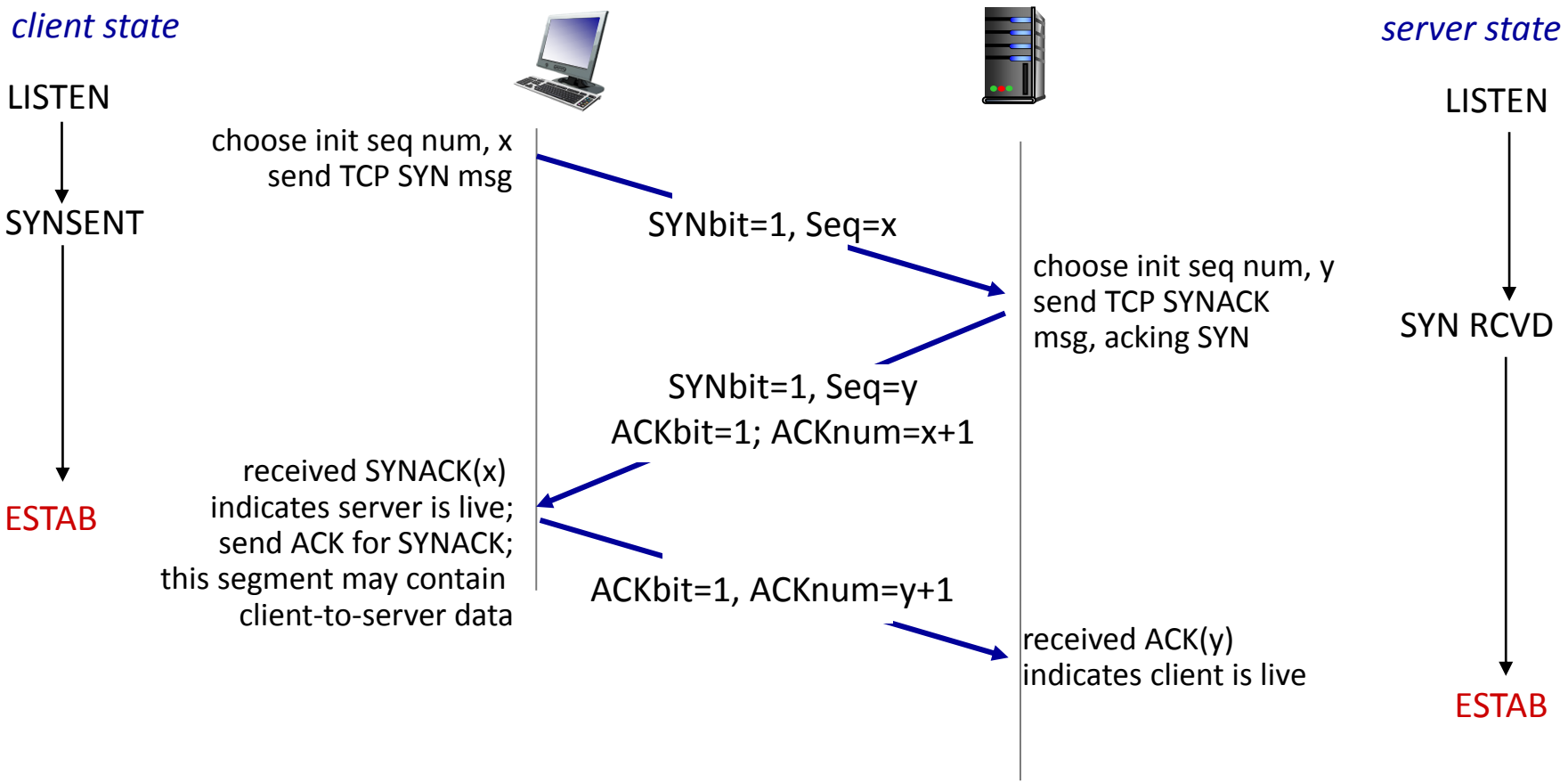
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

# 2-way handshake failure scenarios



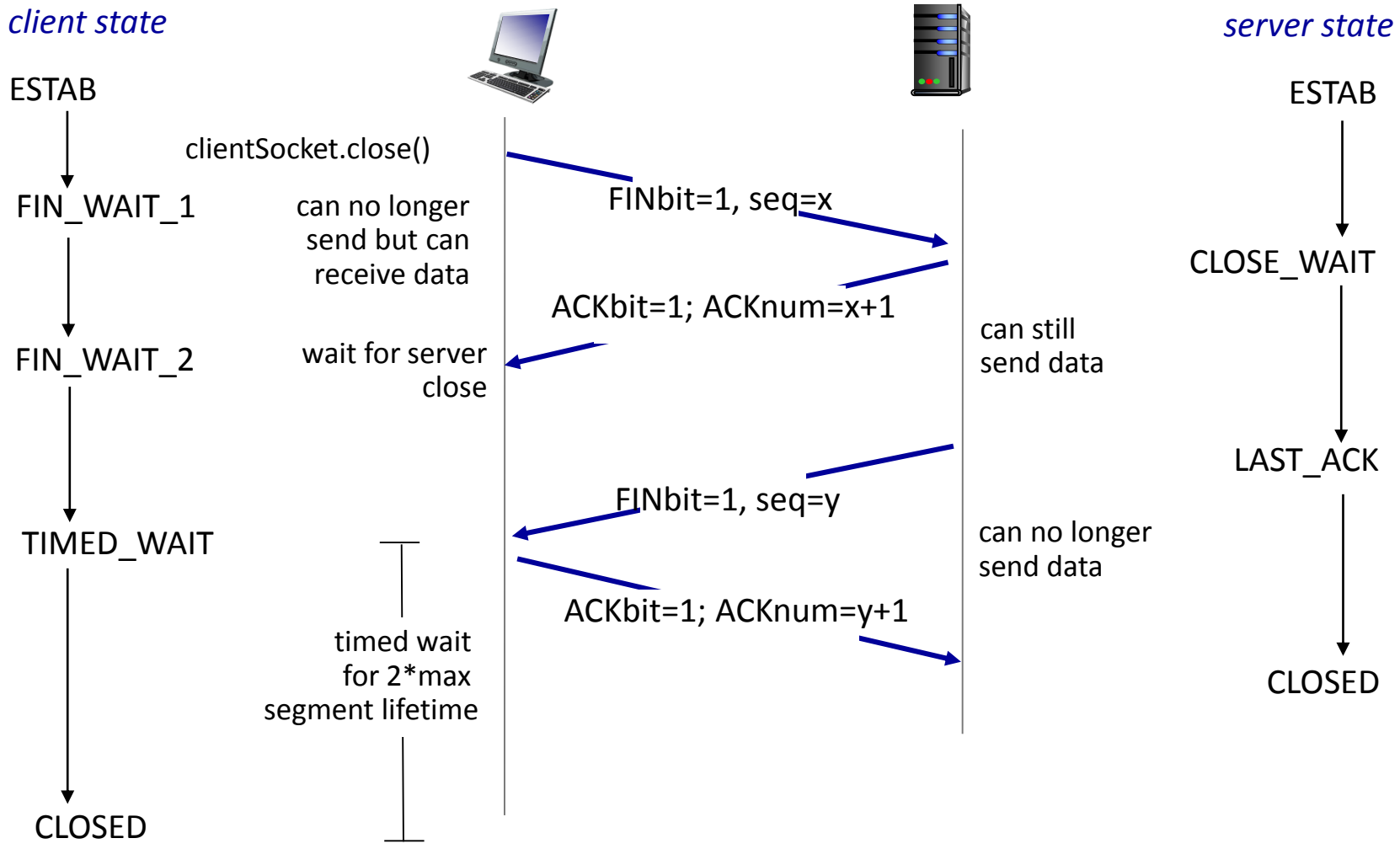
# TCP 3-way handshake



# TCP closing 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

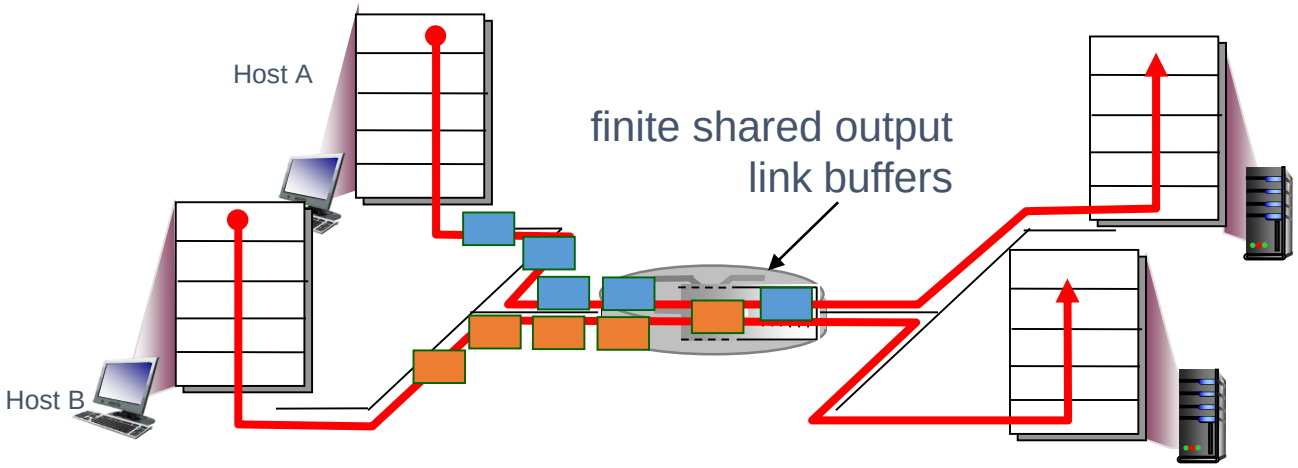
# TCP closing connection



# Congestion control

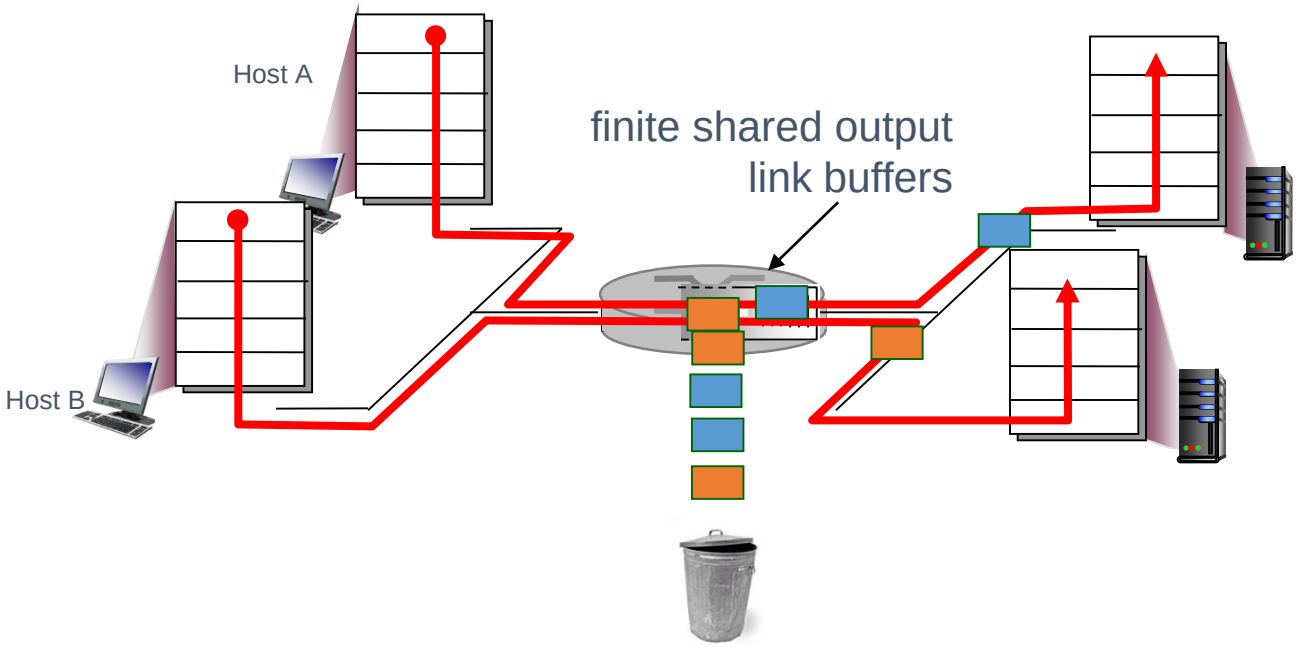
# Congestion

- 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!



# 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!



# Approaches to congestion control

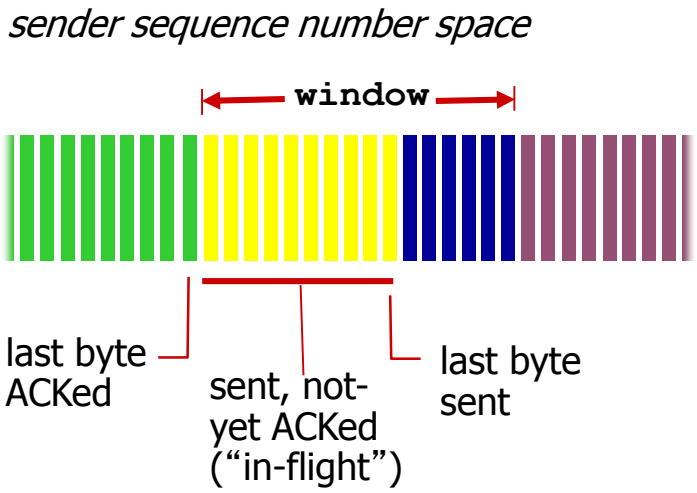
**end-end congestion control:**

- 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



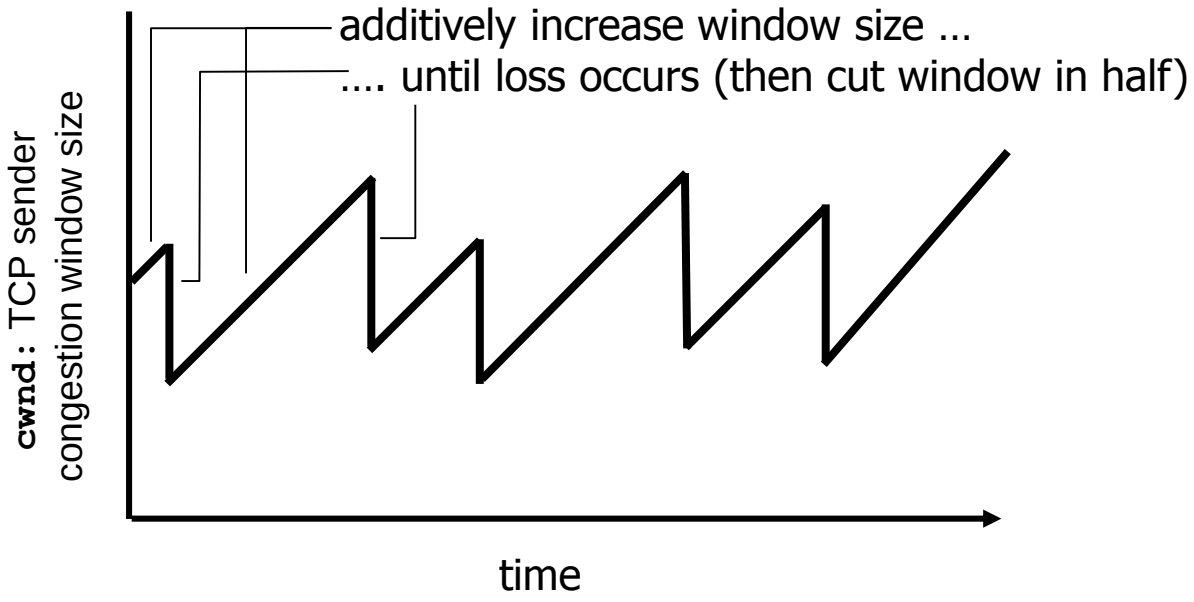

$$\texttt{LastByteSent - LastByteAcked} \leq \texttt{window}$$

- New state variable **`cwnd`**
- **`window = min(cwnd, rwnd)`**
- **`cwnd`** is dynamic, function of perceived network congestion

# Additive increase, multiplicative decrease

*additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected

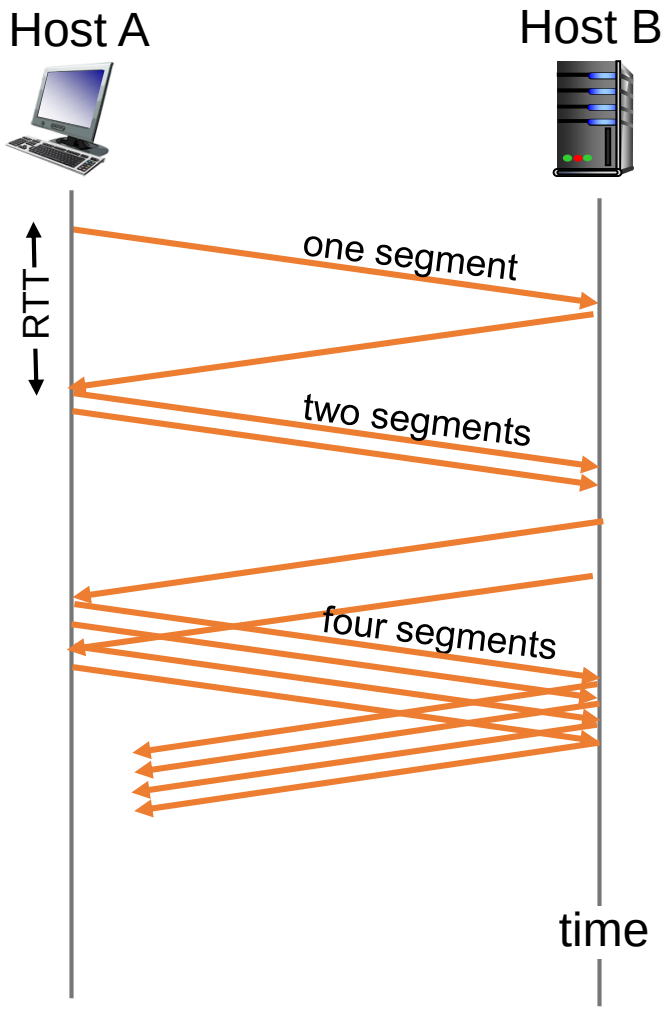
*multiplicative decrease*: cut **`cwnd`** in half after loss




AIMD saw tooth behavior: probing for bandwidth

# 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



# 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

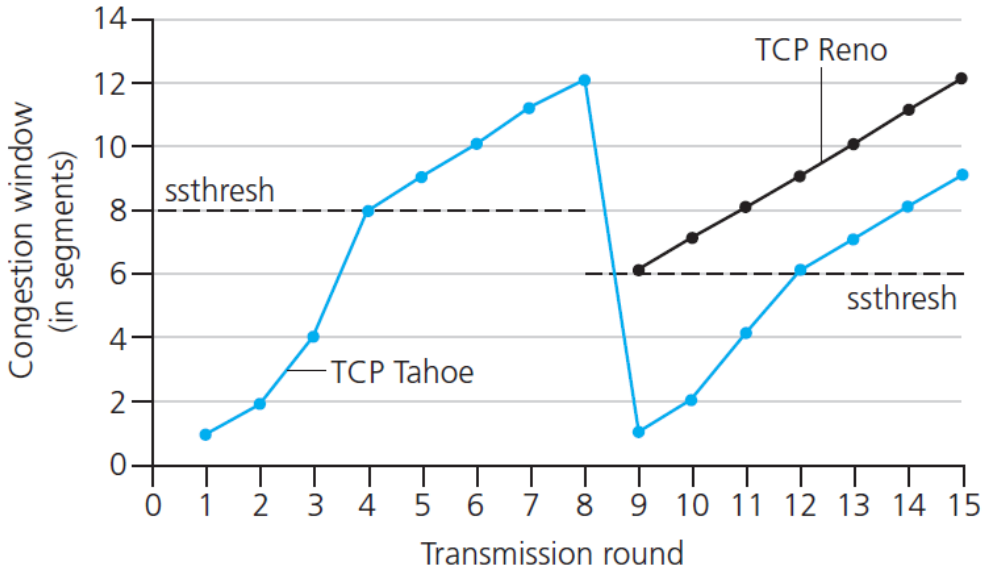
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window then grows linearly

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

# 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