Internet Transport Protocols UDP / TCP

Prof. Anja Feldmann, Ph.D.

anja@net.t-labs.tu-berlin.de

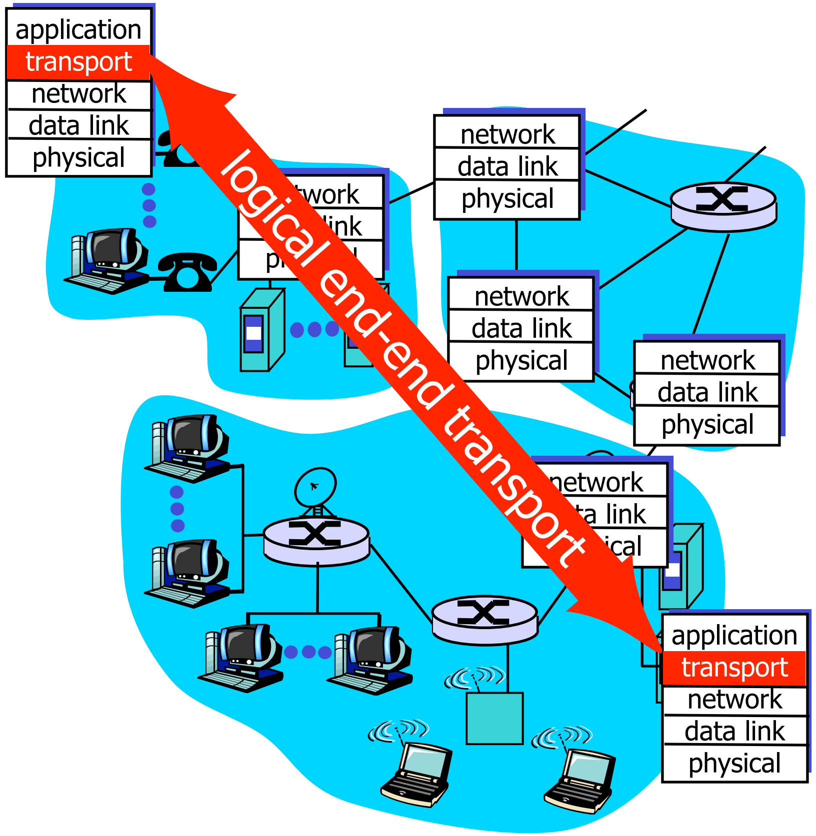
TCP/IP Illustrated, Volume 1, W. Richard Stevens
http://www.kohala.com/start

Transport Layer: Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

Internet Transport-Layer Protocols

- *Network layer:* Logical communication between hosts
- *Transport layer:* Logical communication between processes
  - Relies on, enhances, network layer services
- More than one transport protocol available to apps
  - Internet:
    - TCP
    - UDP



Sockets: interface to applications

Socket API

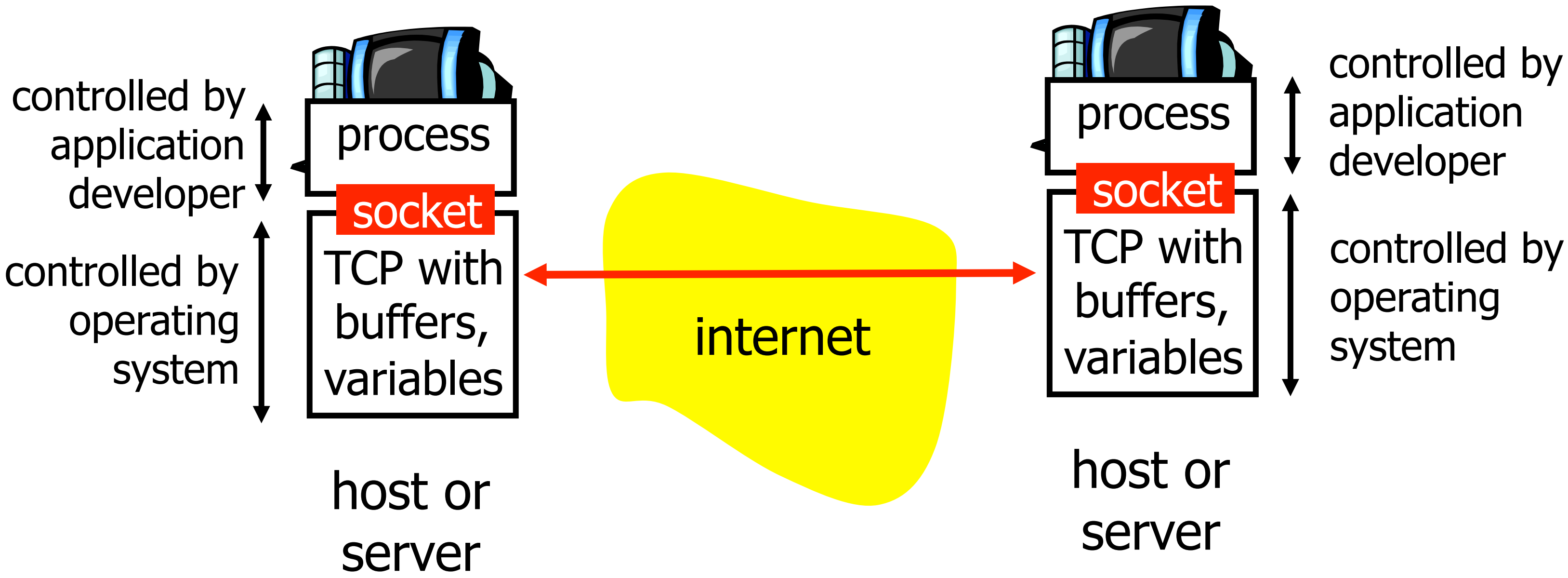
- Introduced in BSD4.1 UNIX, 1981
- Explicitly created, used, released by apps
- Client/server paradigm
- Two types of transport service via socket API:
  - Unreliable datagram
  - Reliable, byte stream-oriented

socket

A *host-local*, *application-created/owned*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Sockets and OS

Socket: a door between application process and end-end-transport protocol (UDP or TCP)



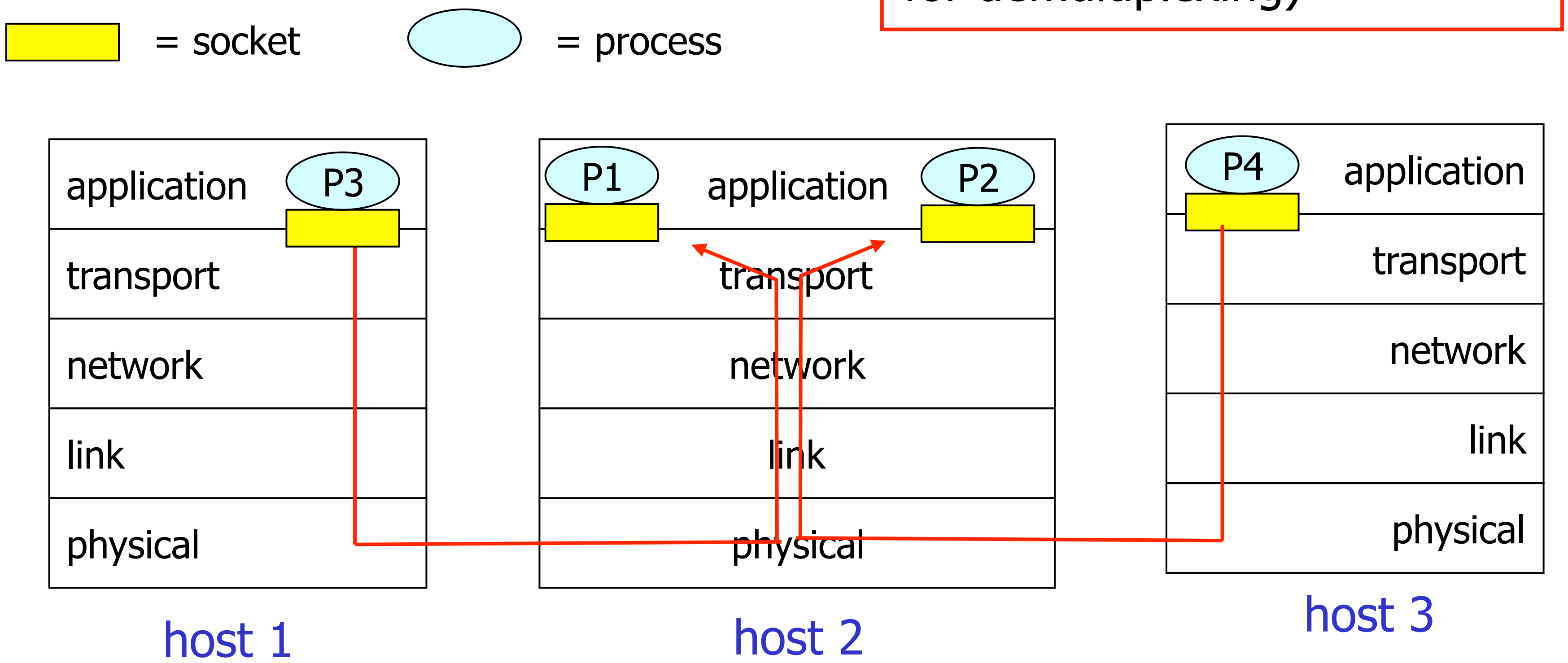
Multiplexing/Demultiplexing

Demultiplexing at rcv host:

Delivering received segments to correct application (socket)

Multiplexing at send host:

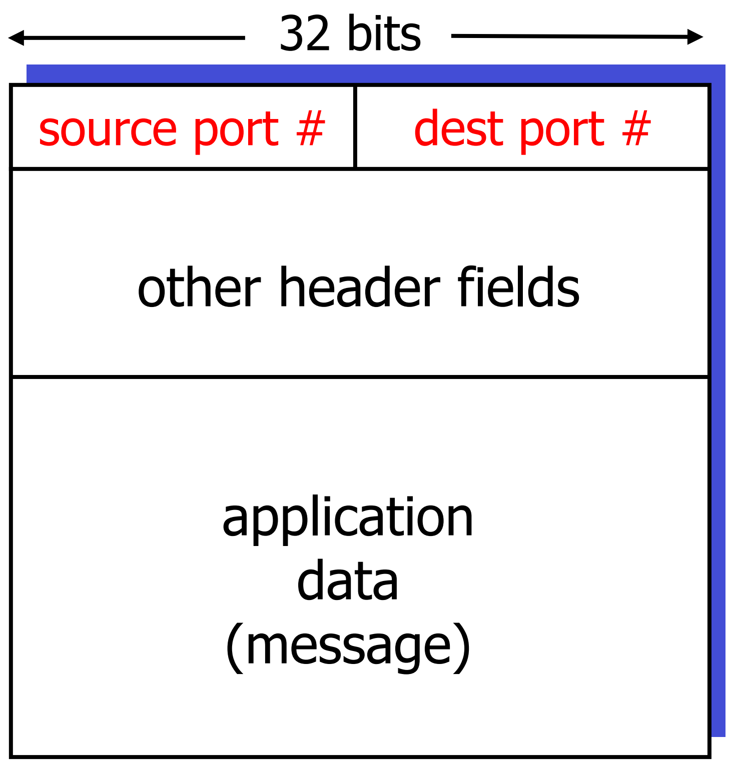
Gathering data from multiple appl. (sockets), enveloping data with header (later used for demultiplexing)



Multiplexing/Demultiplexing

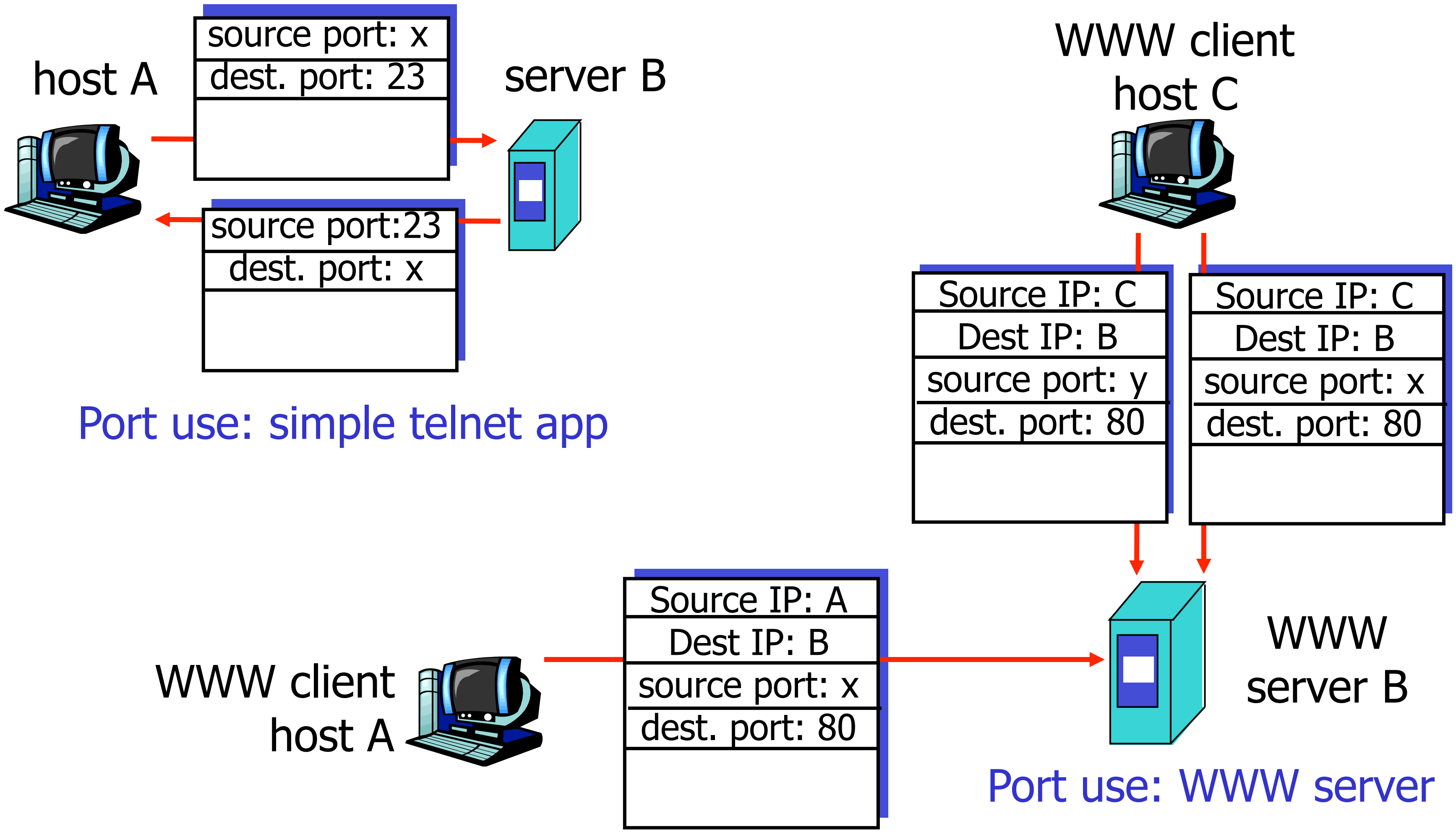
Multiplexing/demultiplexing:

- Based on sender, receiver port numbers, IP addresses
  - Source, dest port #s in each segment
  - Well-known port numbers for specific applications (see /etc/services)



TCP/UDP segment format

Multiplexing/Demultiplexing: Examples



Port use: simple telnet app

Port use: WWW server

UDP: User Datagram Protocol [RFC 768]

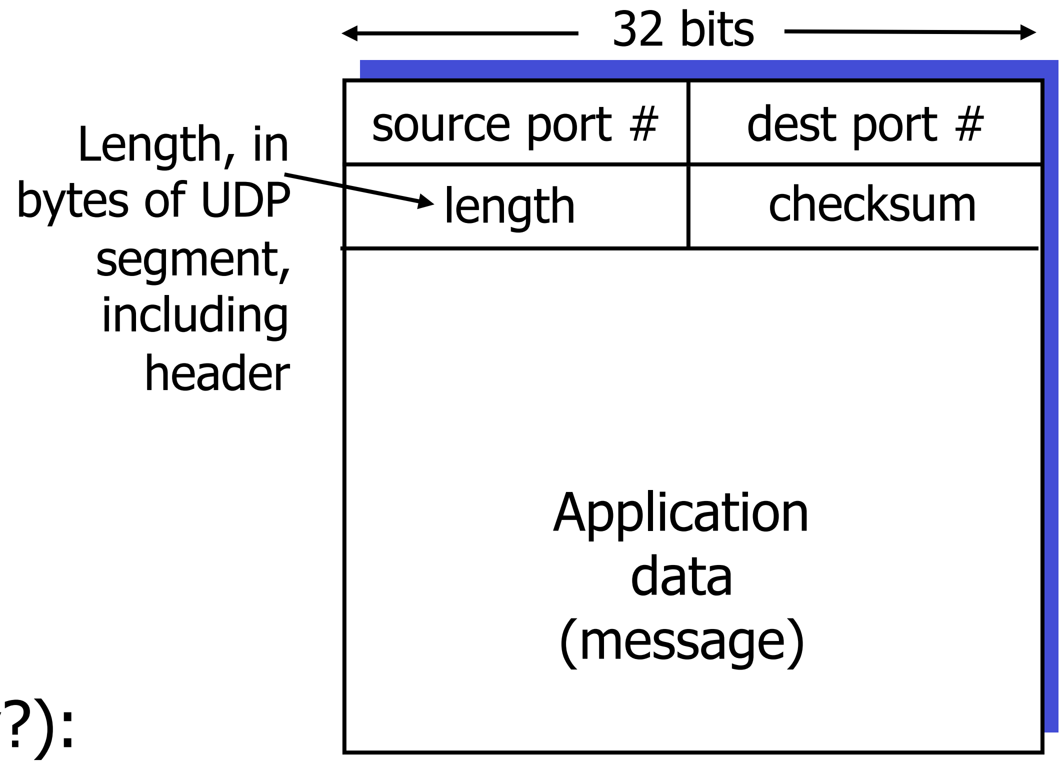
- "Bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to application
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

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

UDP: More

- Each user request transferred in a single datagram
- UDP has a receive buffer but no sender buffer
- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS, SNMP, NFS
- Reliable transfer over UDP: add reliability at application layer



UDP segment format

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order *byte stream:*
  - No “message boundaries”
- Pipelined:
  - TCP congestion and flow control set window size
- Full duplex data:
  - Bi-directional data flow in one connection
  - MSS: maximum segment size
- Connection-oriented:
  - Handshaking (exchange of control msgs) init’s sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver
- Congestion controlled:
  - Sender will not overwhelm the network

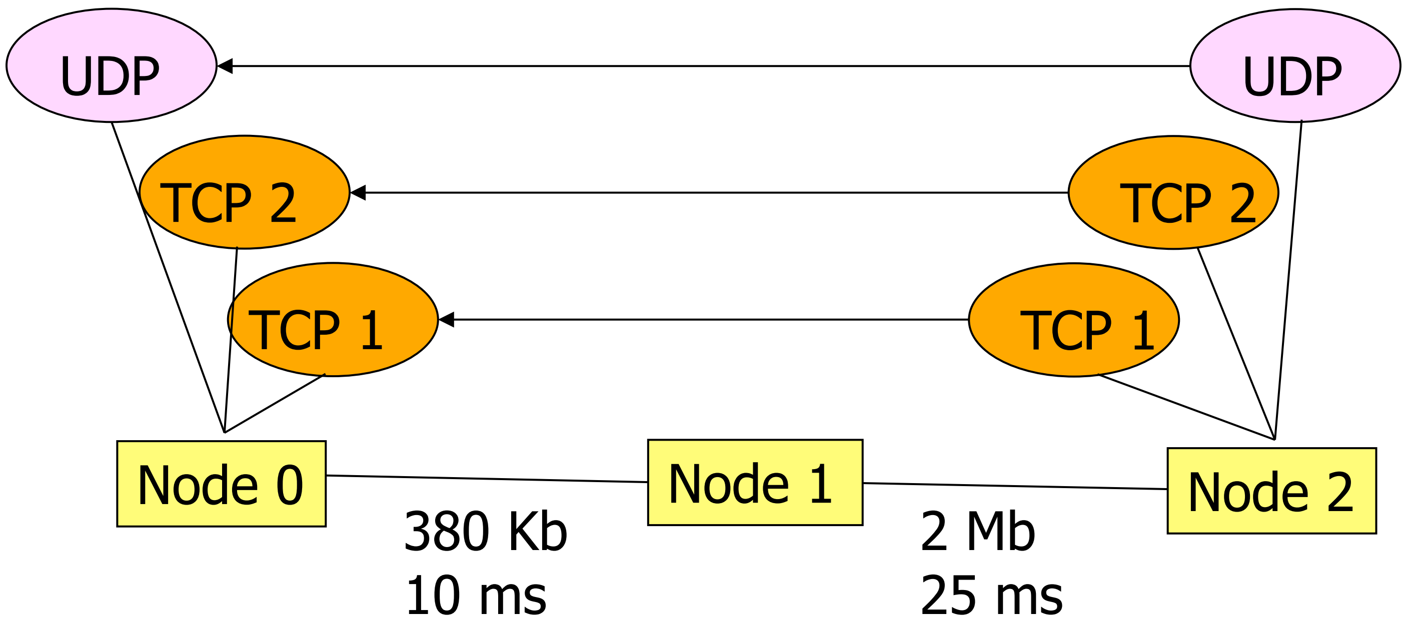
Simulating Transport Protocols

- Network simulator
- Examples:
  - Network Simulator (NS), SSFNet, ...

- Animation of NS traces via NAM (Network Animator)
- Try it!

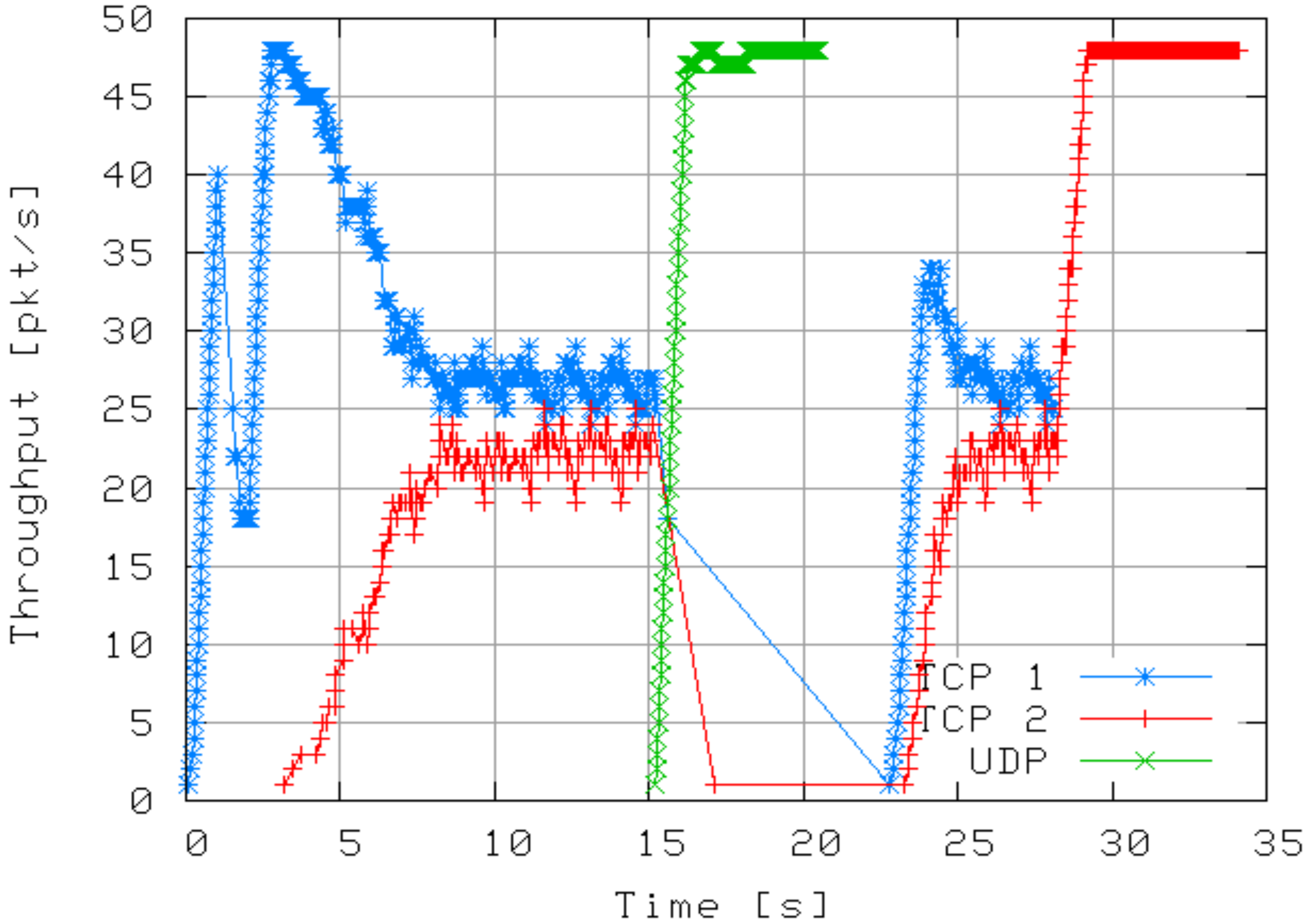
Simulating Transport Protocols

- Example: 2 TCP connections + 1 UDP flow
- Topology:

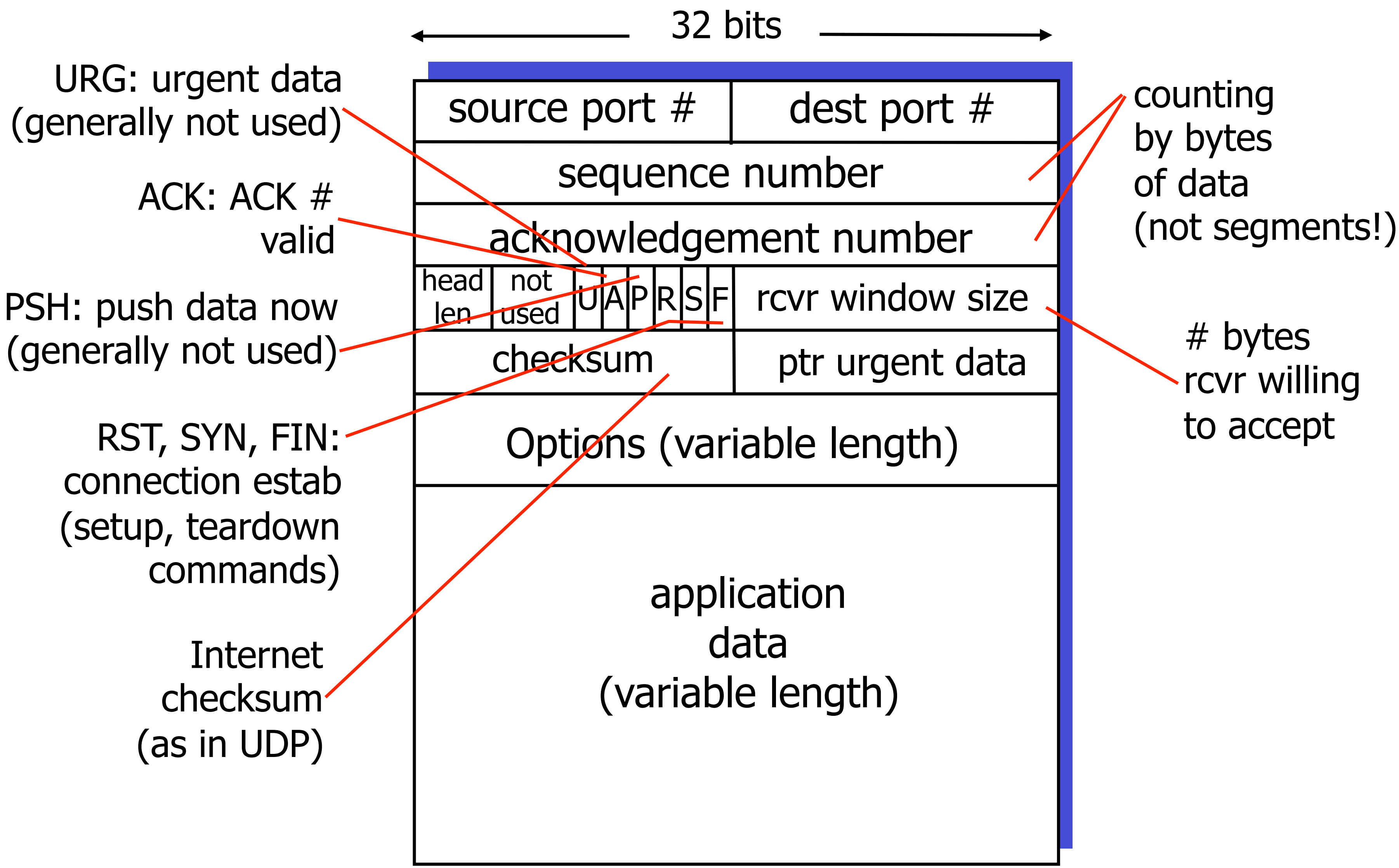


- TCP1 starts at time 0 seconds, TCP2 at time 3 seconds
- UDP starts at time 15 seconds

Simulation Results



TCP Segment Structure



TCP Reliability: Seq. #'s and ACKs

Seq. #'s:

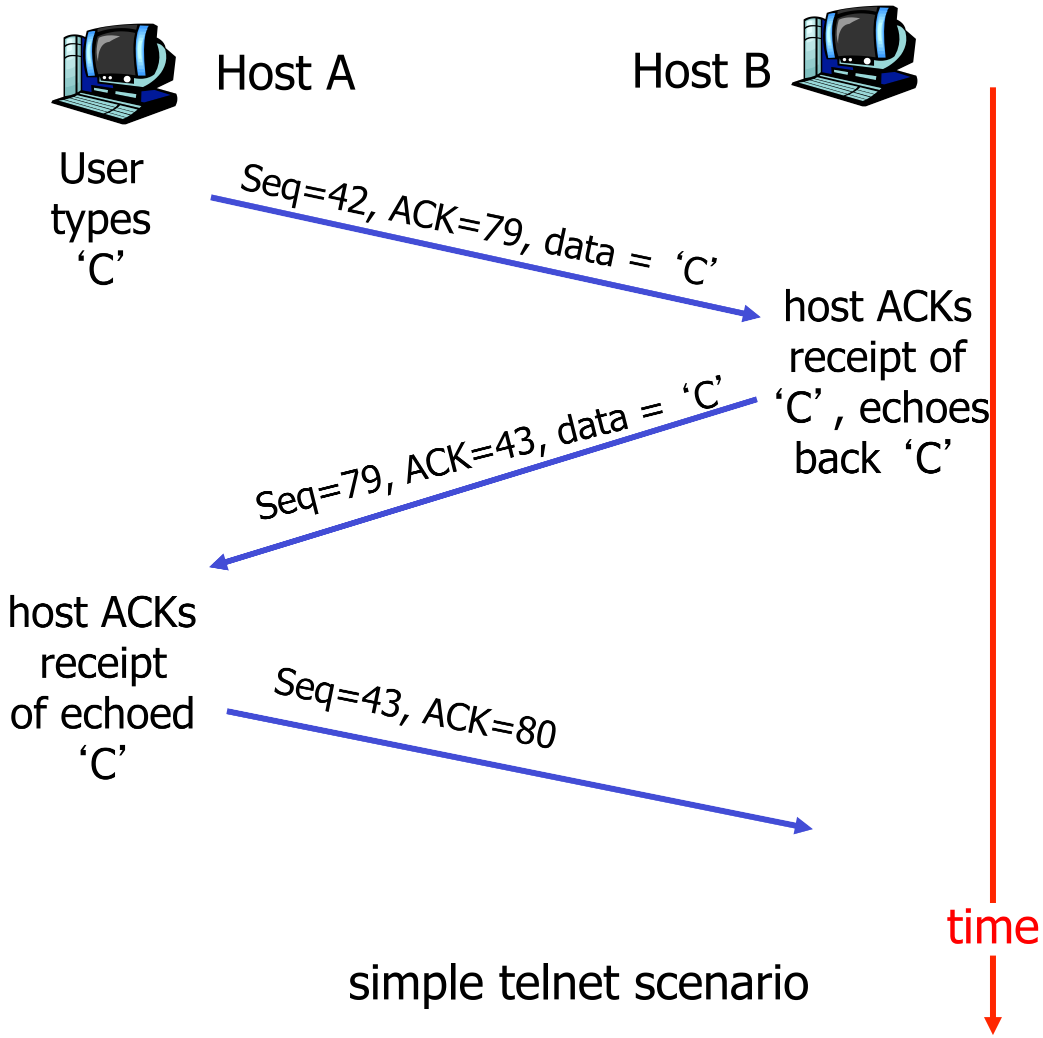
- **Byte stream**
  "Number" of first byte in segment's data

ACKs:

- Seq # of **next byte** expected from other side
- Cumulative ACK

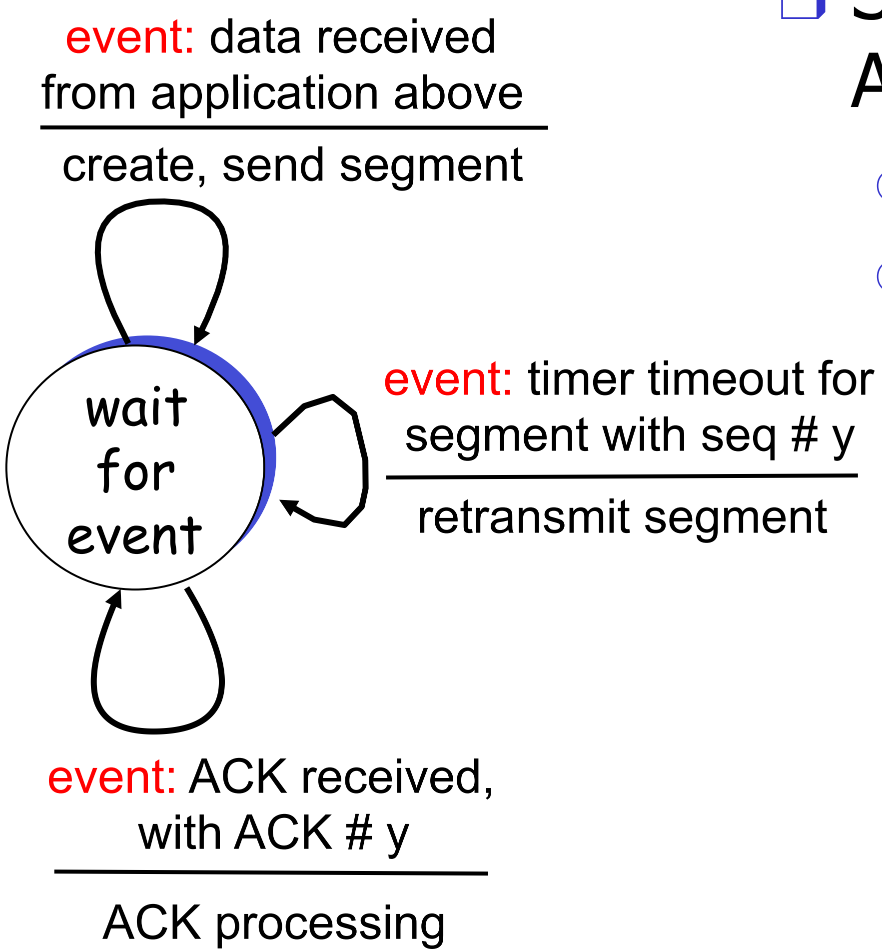
Q: How receiver handles out-of-order segments?

- TCP spec doesn't say, – up to implementer



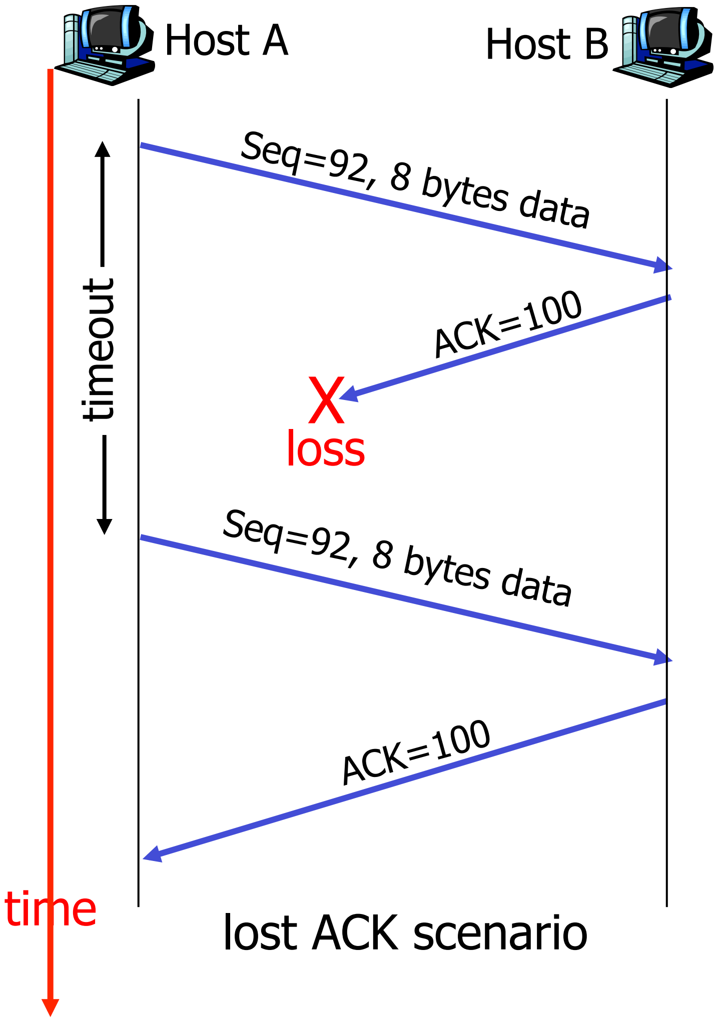
simple telnet scenario

TCP: Reliable Data Transfer

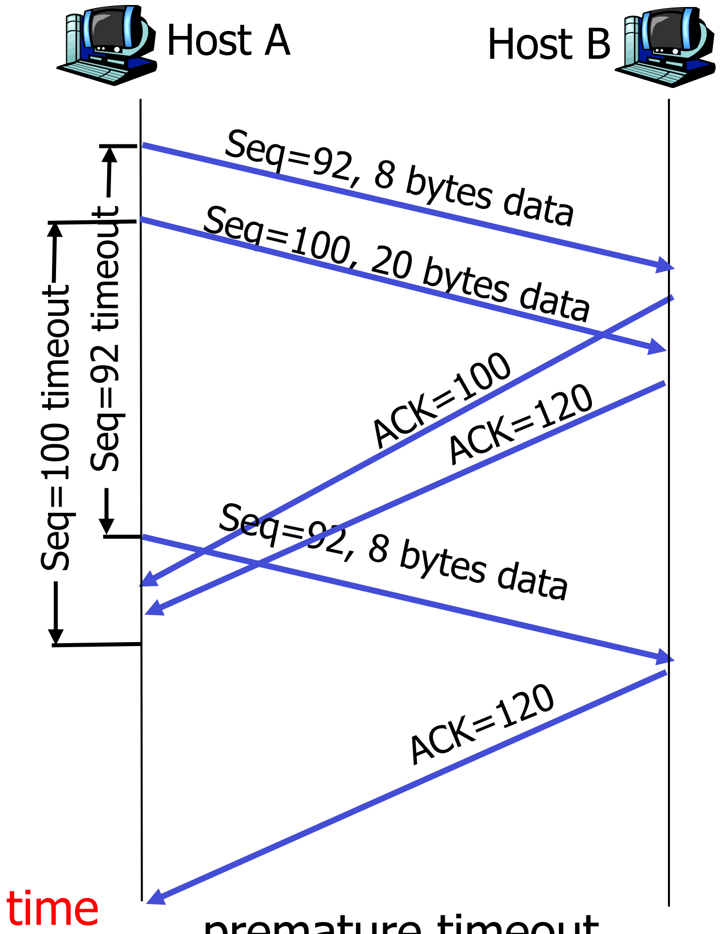


- Simplified sender Assumption
  - One way data transfer
  - No flow, congestion control
- Packet loss detection:
  - Retransmission timeout
  - Fast retransmit
    - Three duplicate ACKs
- Retransmission mechanisms
  - ARQ: Go-Back-N, selected retransmissions

TCP: Retransmission Scenarios



lost ACK scenario



premature timeout,
cumulative ACKs

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 (reduces ACK traffic)
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 (trigger fast retransmit)
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

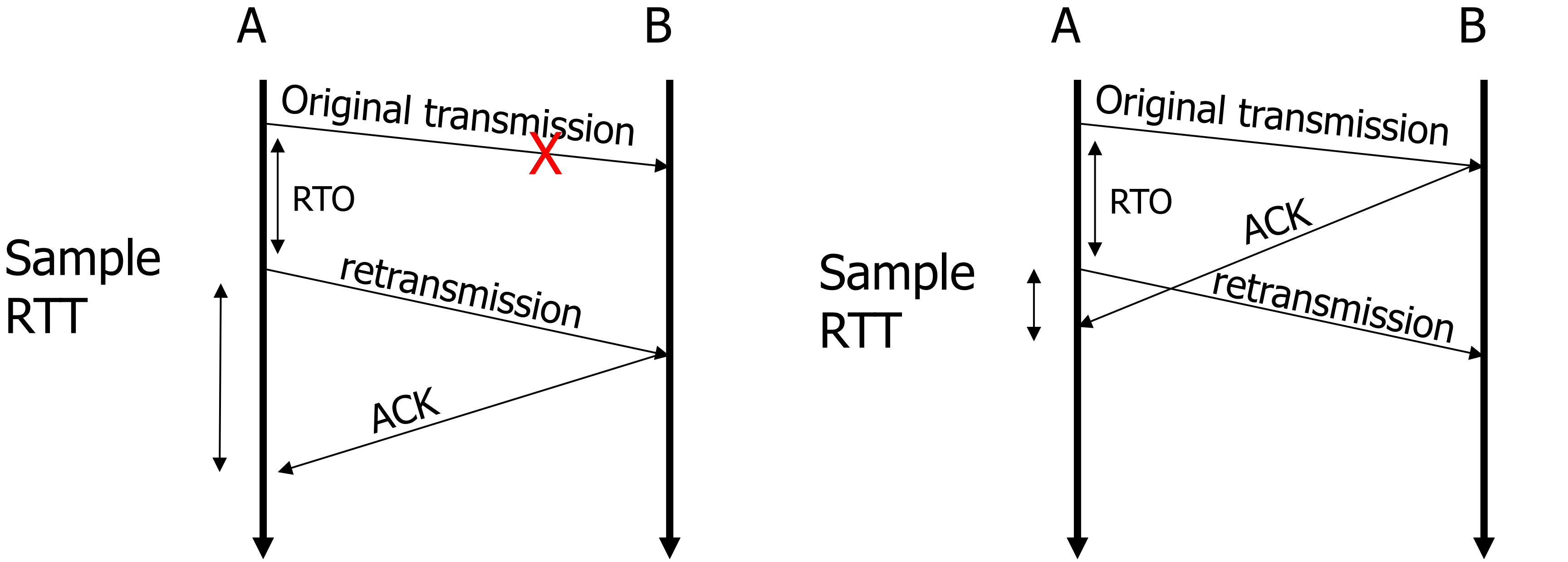
TCP Retransmission Timeout

- TCP uses one timer for one pkt only
- Retransmission Timeout (RTO) calculated dynamically
  - Based on Round Trip Time estimation (RTT)
  - Wait at least one RTT before retransmitting
  - Importance of accurate RTT estimators:
    - Low RTT → unneeded retransmissions
    - High RTT → poor throughput
  - RTT estimator must adapt to change in RTT
    - But not too fast, or too slow!
  - Spurious timeouts
    - “Conservation of packets” principle – more than a window worth of packets in flight

Retransmission Timeout Estimator

- Round trip times exponentially averaged:
  - New RTT = α (old RTT) + $(1 - \alpha)$ (new sample)
  - $a = 0.875$ for most TCPs
- Retransmit timer set to β RTT, where $b = 2$
  - Every time timer expires, RTO exponentially backed-off
- Key observation: At high loads round trip variance is high
- Solution (currently in use):
  - Base RTO on RTT and standard deviation of RTT: RTT + 4 * rttvar
  - rttvar = α * dev + $(1-\alpha)$ * rttvar
    - dev = linear deviation (also referred to as mean deviation)
    - Inappropriately named – actually smoothed linear deviation
  - RTO is discretized into ticks of 500ms (RTO ≥ 2 ticks)

Retransmission Ambiguity

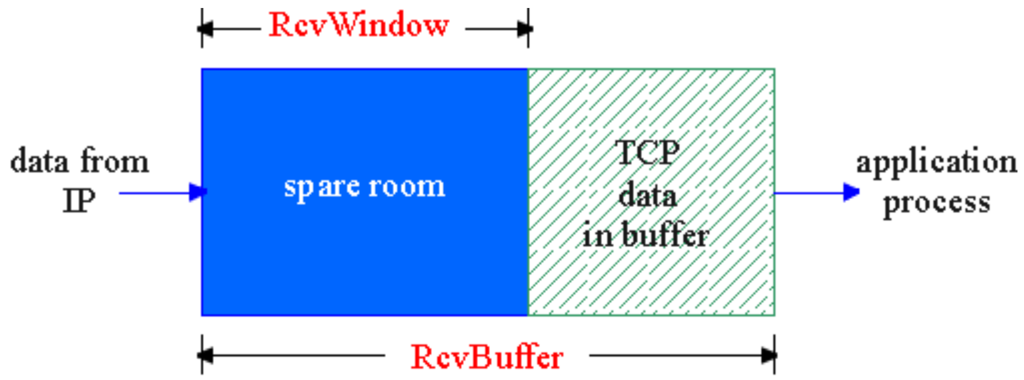


- Karn's RTT Estimator
  - If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

TCP Flow Control: Sliding Window Protocol

flow control

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



receiver buffering

Receiver: Explicitly informs sender of (dynamically changing) amount of free buffer space

- **`rcvr window size`** field in TCP segment

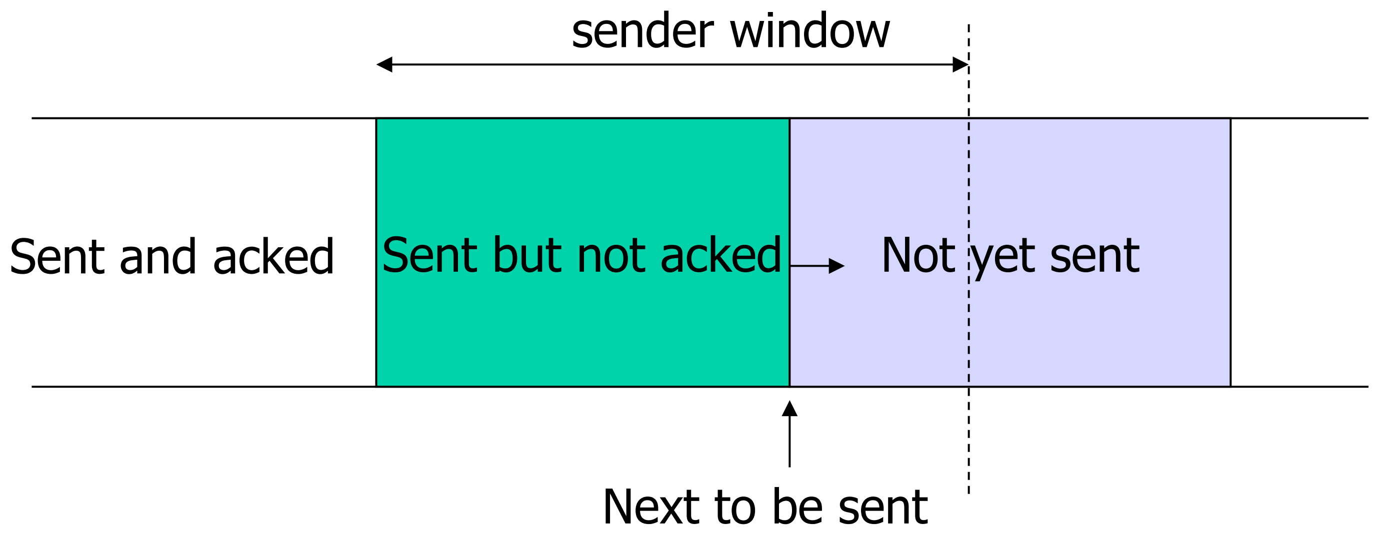
Sender: Amount of transmitted, unACKed data less than most recently-receiver **`rcvr window size`**

TCP Flow Control

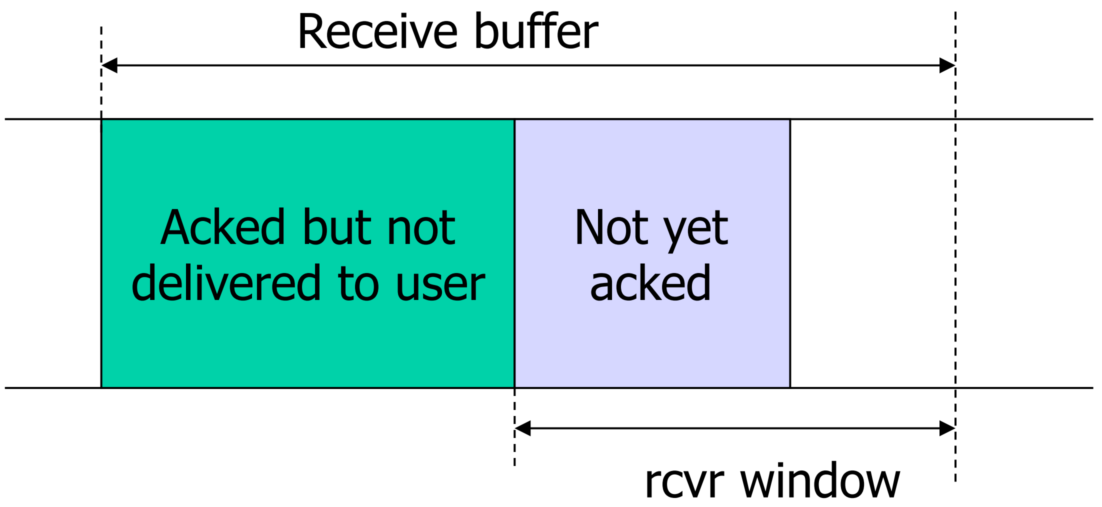
- TCP is a sliding window protocol
  - For window size n, can send up to n bytes without receiving an acknowledgement
  - When the data is acknowledged, the window slides forward
- Original TCP always sent entire window
  - Congestion control now limits this via congestion window determined by the sender! (network limited)
  - If not, data rate is receiver limited
- Silly window syndrome
  - Too many small packets in flight
  - Limit the # of smaller pkts than MSS to one per RTT

Window Flow Control:

Sender Side



Receiver Side



Ideal Window Size

- Ideal size = delay * bandwidth
  - Delay-bandwidth product (RTT * bottleneck bitrate)
- Window size < delay * bw ⇨ wasted bandwidth
- Window size > delay * bw ⇨
  - Queuing at intermediate routers ⇨ increased RTT
  - Eventually packet loss

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- Initialize TCP variables:
  - Seq. #s
  - Buffers, flow control info (e.g. **`RcvWindow`**)
  - MSS and other options
- *Client:* connection initiator, *server:* contacted by client

- Three-way handshake
  - Simultaneous open
- TCP Half-Close (four-way handshake)
- Connection aborts via RSTs

TCP Connection Management (2)

Three way handshake:

Step 1: Client end system sends TCP SYN control segment to server

- Specifies initial seq #
- Specifies initial window #

Step 2: Server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- Allocates buffers
- Specifies server → receiver initial seq. #
- Specifies initial window #

Step 3: Client system receives SYNACK

TCP Connection Management (3)

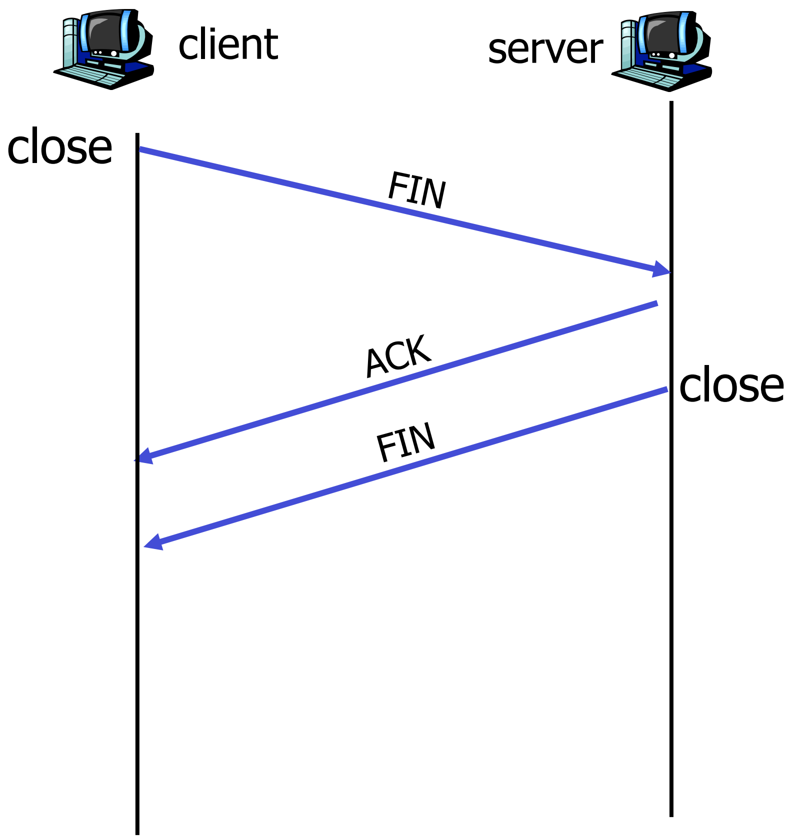
Closing a connection:

Client closes socket:

```
clientSocket.close();
```

Step 1: Client end system sends TCP FIN control segment to server

Step 2: Server receives FIN, replies with ACK. Closes connection, sends FIN.



TCP Connection Management (4)

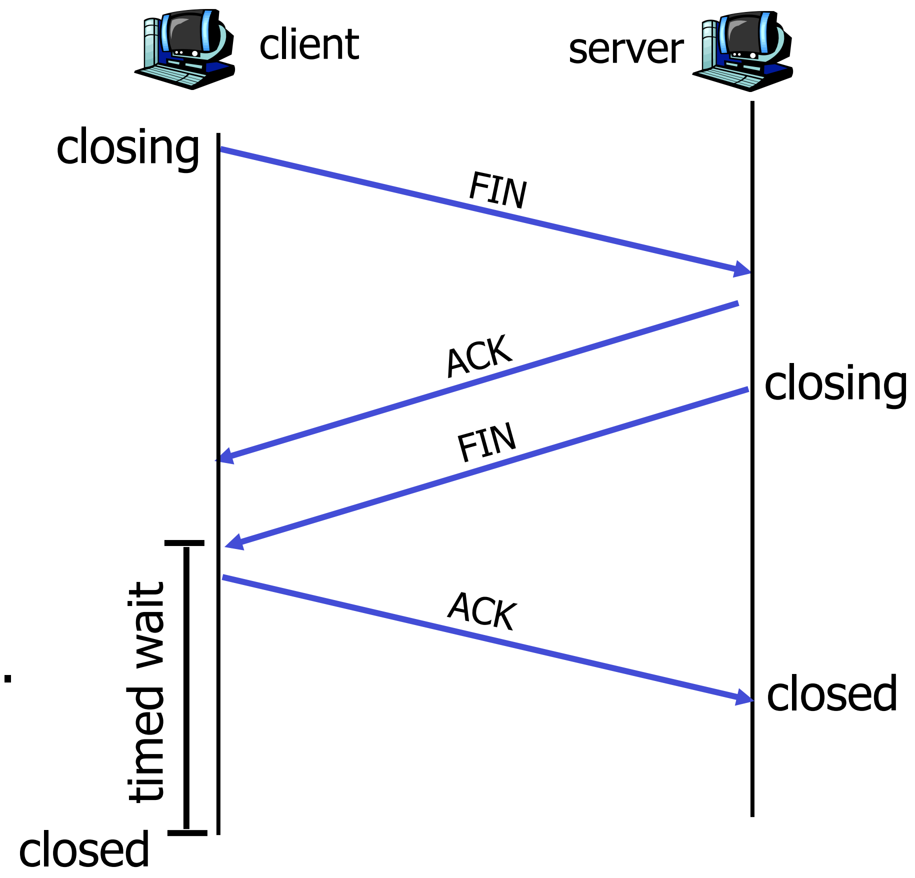
Step 3: Client receives FIN, replies with ACK.

- Enters "timed wait" – will respond with ACK to received FINs

Step 4: Server, receives ACK. Connection closed.

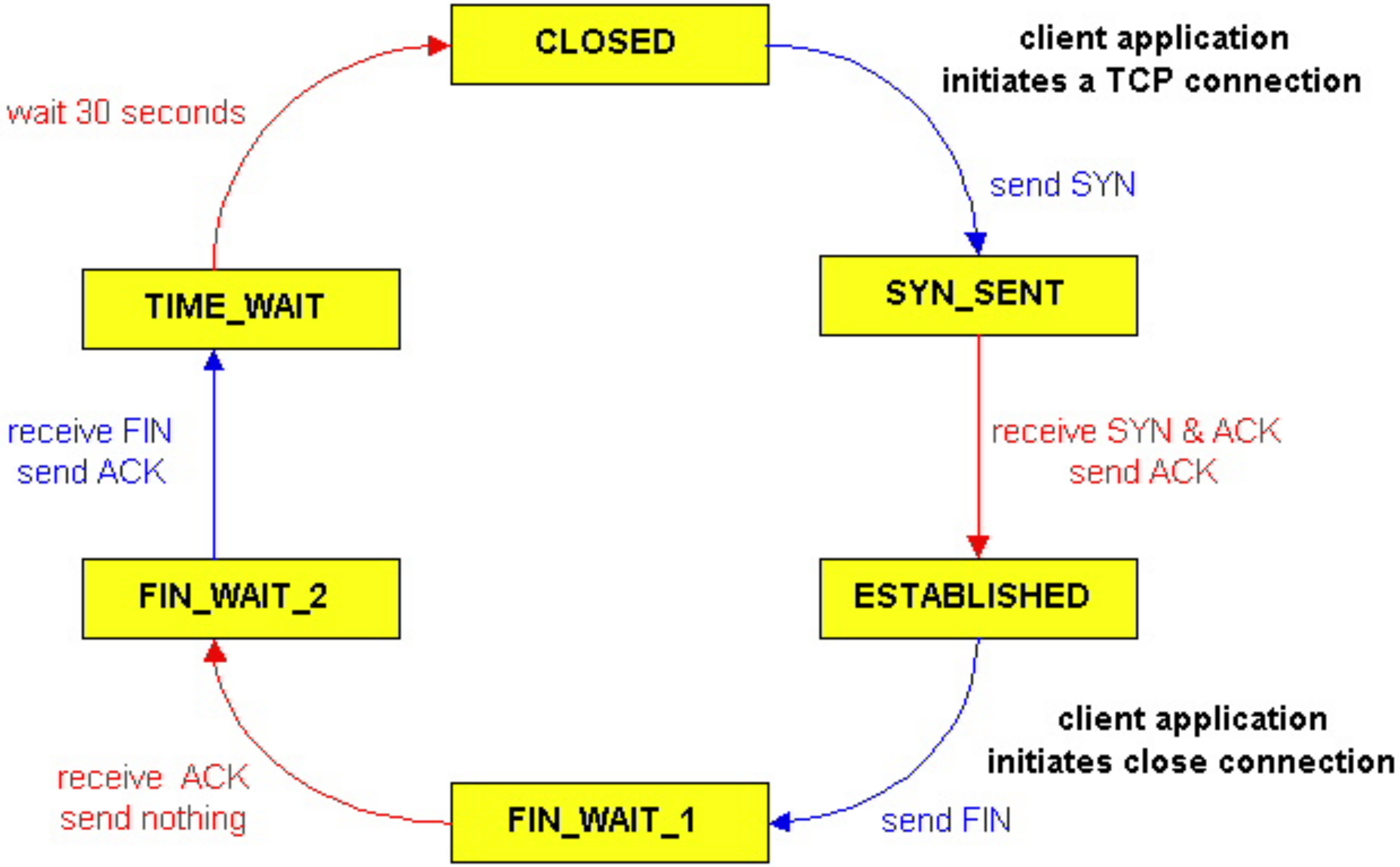
Note: With small modification, can handle simultaneous FINs.

Note: SYN, SYNACK, FIN each add one byte to seq #



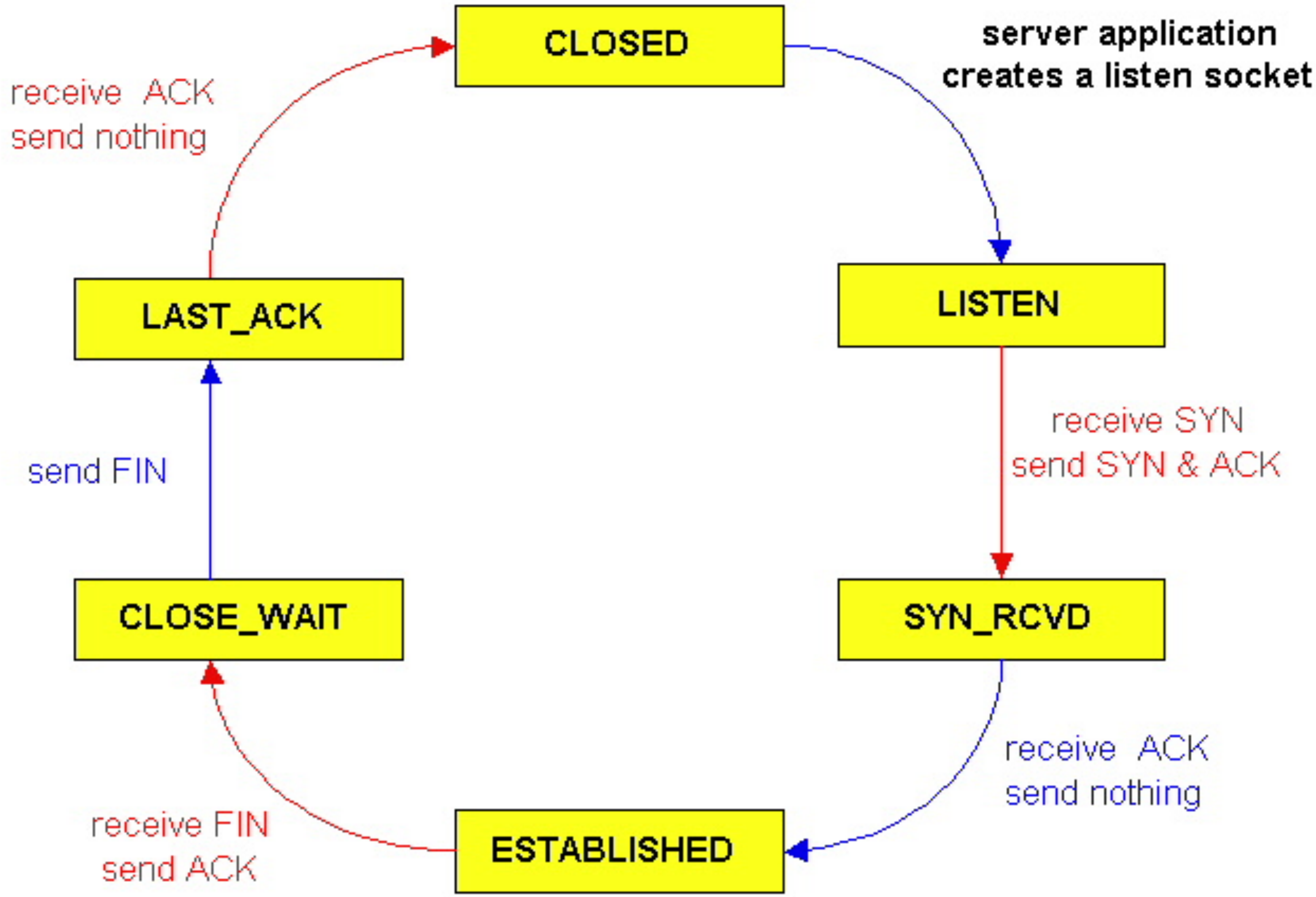
TCP Connection Management (5)

TCP client lifecycle

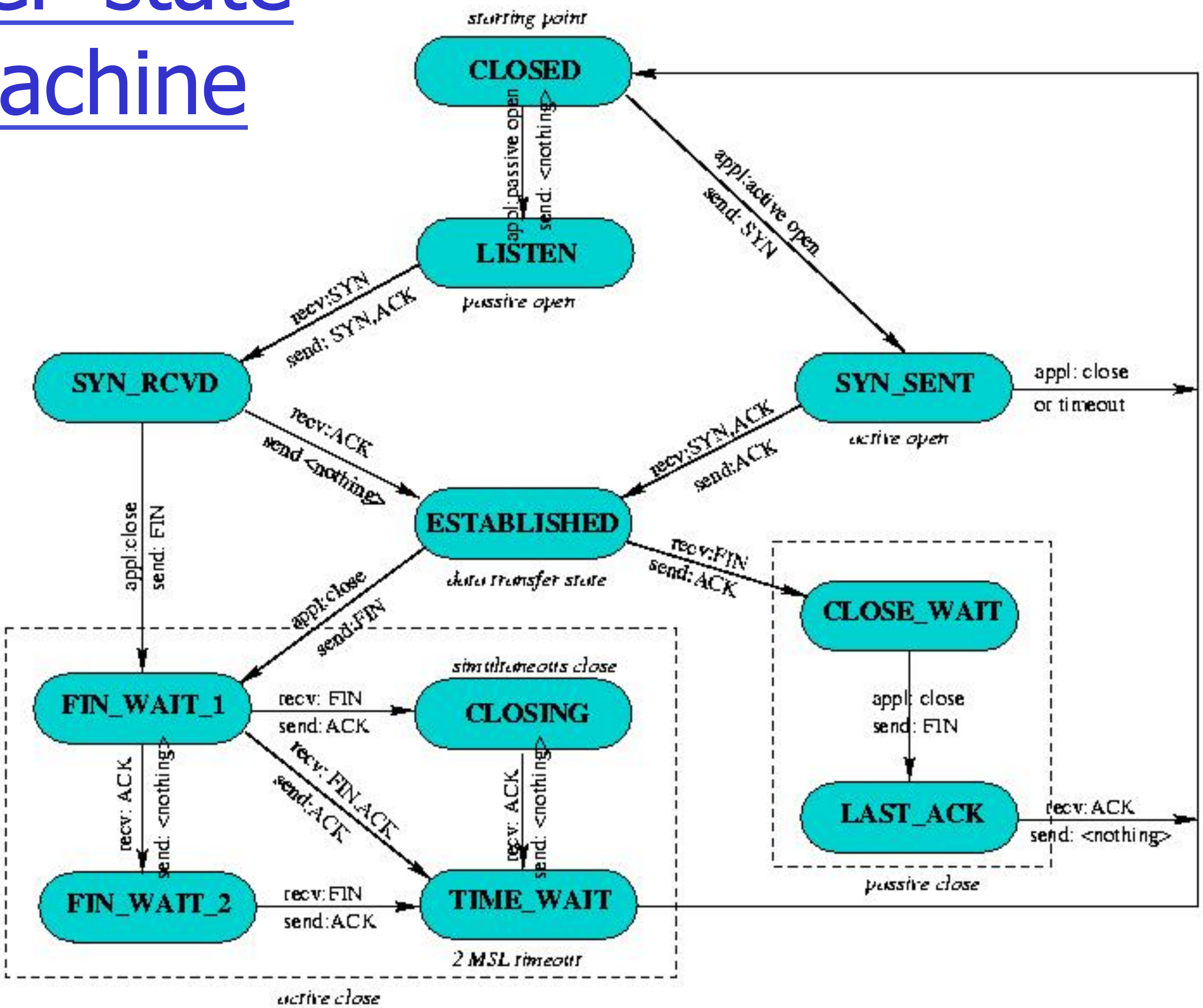


TCP Connection Management (cont)

TCP server lifecycle



TCP state machine



Excursion:
Congestion Control Principles

TCP Acknowledgement Clocking

- TCP is "self-clocking"
- New data sent when old data is acked
- Ensures an "equilibrium"
- But how to get started?
  - Slow Start
  - Congestion Avoidance
- Other TCP features
  - Fast Retransmission
  - Fast Recovery

TCP Congestion Control:

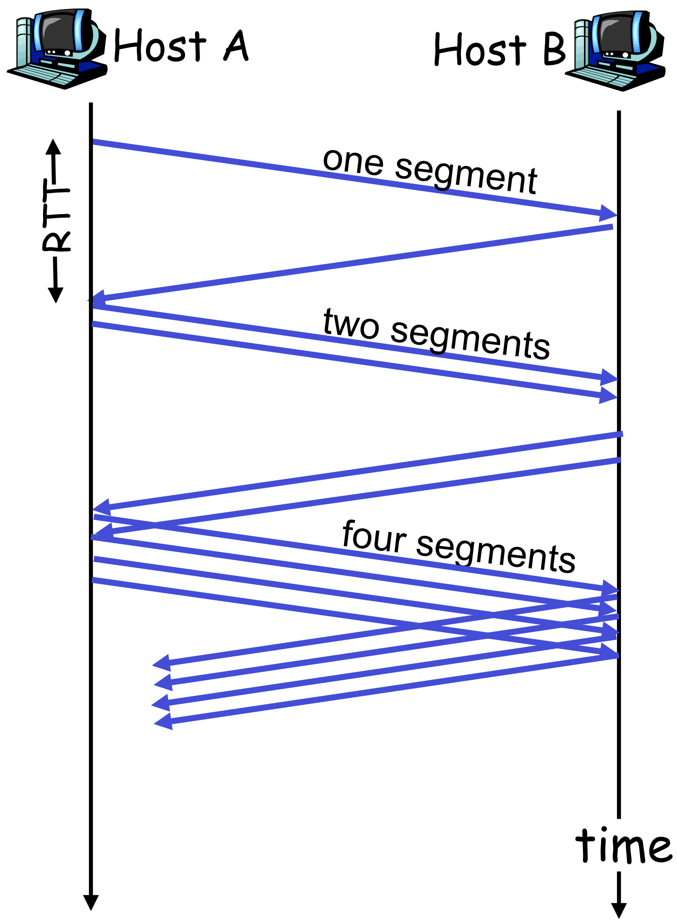
- "Probing" for usable bandwidth:
  - Ideally: Transmit as fast as possible (**`cwnd`** as large as possible) without loss
  - *Increase* **`cwnd`** until loss (congestion)
  - Loss: *Decrease* **`cwnd`**, then begin probing (increasing) again
- Two "phases"
  - Slow start
  - Congestion avoidance
- Important variables:
  - **`cwnd`**
  - **`threshold (ssthresh)`**: Defines threshold between two slow start phase, congestion control phase

TCP Slowstart

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event: Timeout or three duplicate ACKs (Tahoe TCP or Reno TCP)

Slowstart algorithm

```
initialize: cwnd = 1
for (each segment ACKed)
      cwnd++
until (loss event OR
        cwnd> threshold)
```



Congestion Avoidance

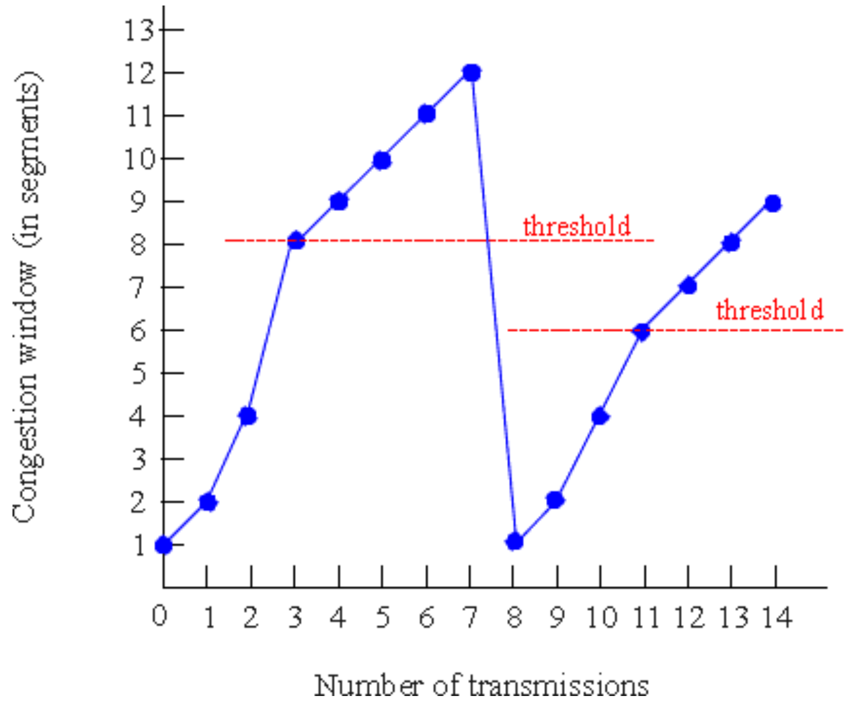
- Loss implies congestion – why?
  - Not necessarily true on all link types
- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving new ACK
  - Increase cwnd by 1/cwnd
  - Results in additive increase

TCP Congestion Avoidance

Congestion avoidance

```
/* slowstart is over       */
/* cwnd> threshold */
until (loss event) {
  every cwnd segments ACKed:
      cwnd++
  }
threshold = cwnd/2
cwnd= 1
perform slowstart[1]
```

1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

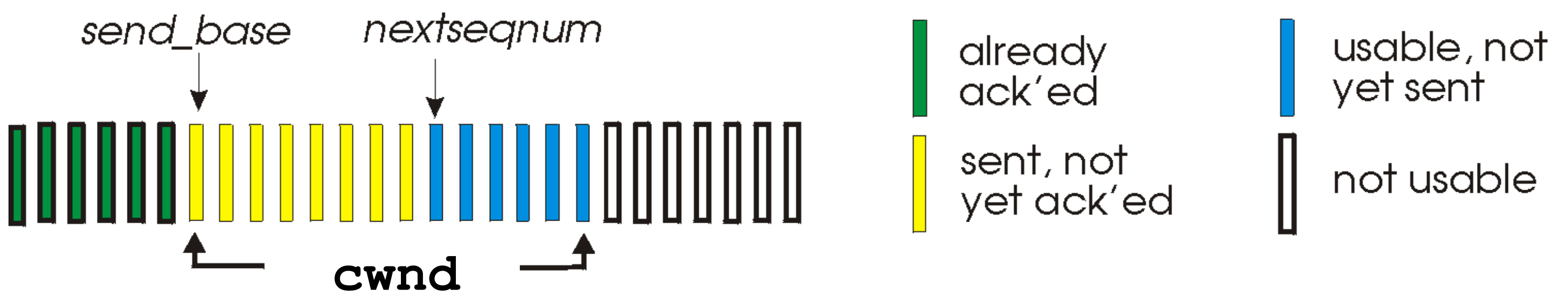


Return to Slow Start

- If packet is lost we lose self clocking
  - Need to implement slow-start and congestion avoidance together
- When timeout occurs
  - Set threshold to 0.5 cwnd
  - Set cwnd to one segment
- When three duplicate acks occur:
  - Set threshold to 0.5 cwnd
  - Retransmit missing segment == Fast Retransmit
  - cwnd = threshold + number of dupacks
  - Upon receiving acks cwnd = threshold (cut in half!)
  - Use congestion avoidance == Fast Recovery

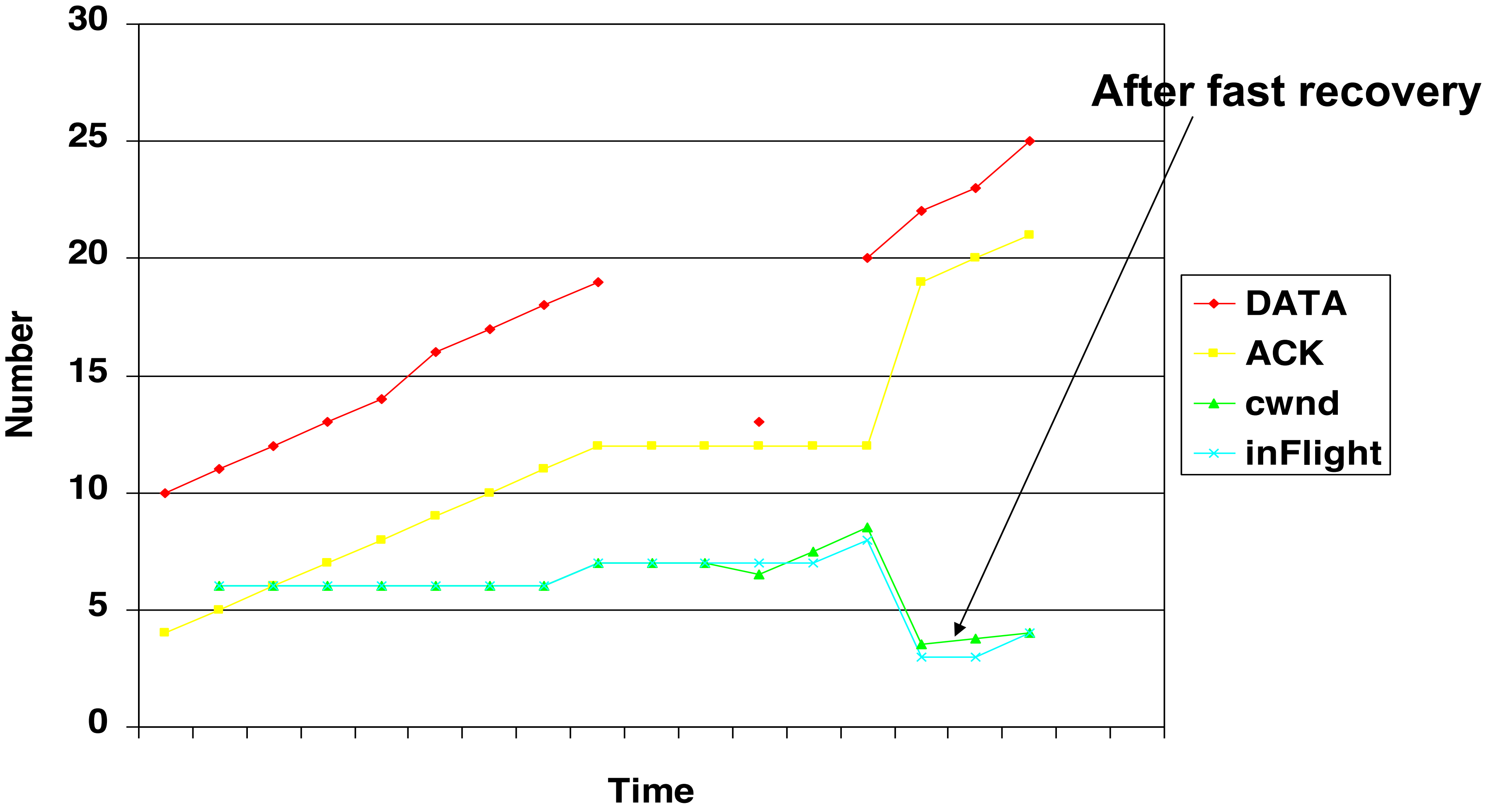
TCP Congestion Control

- End-end control (no network assistance)
- TCP throughput limited by rcvr window (flow control)
- Transmission rate limited by congestion window size, **cwnd**, over segments:



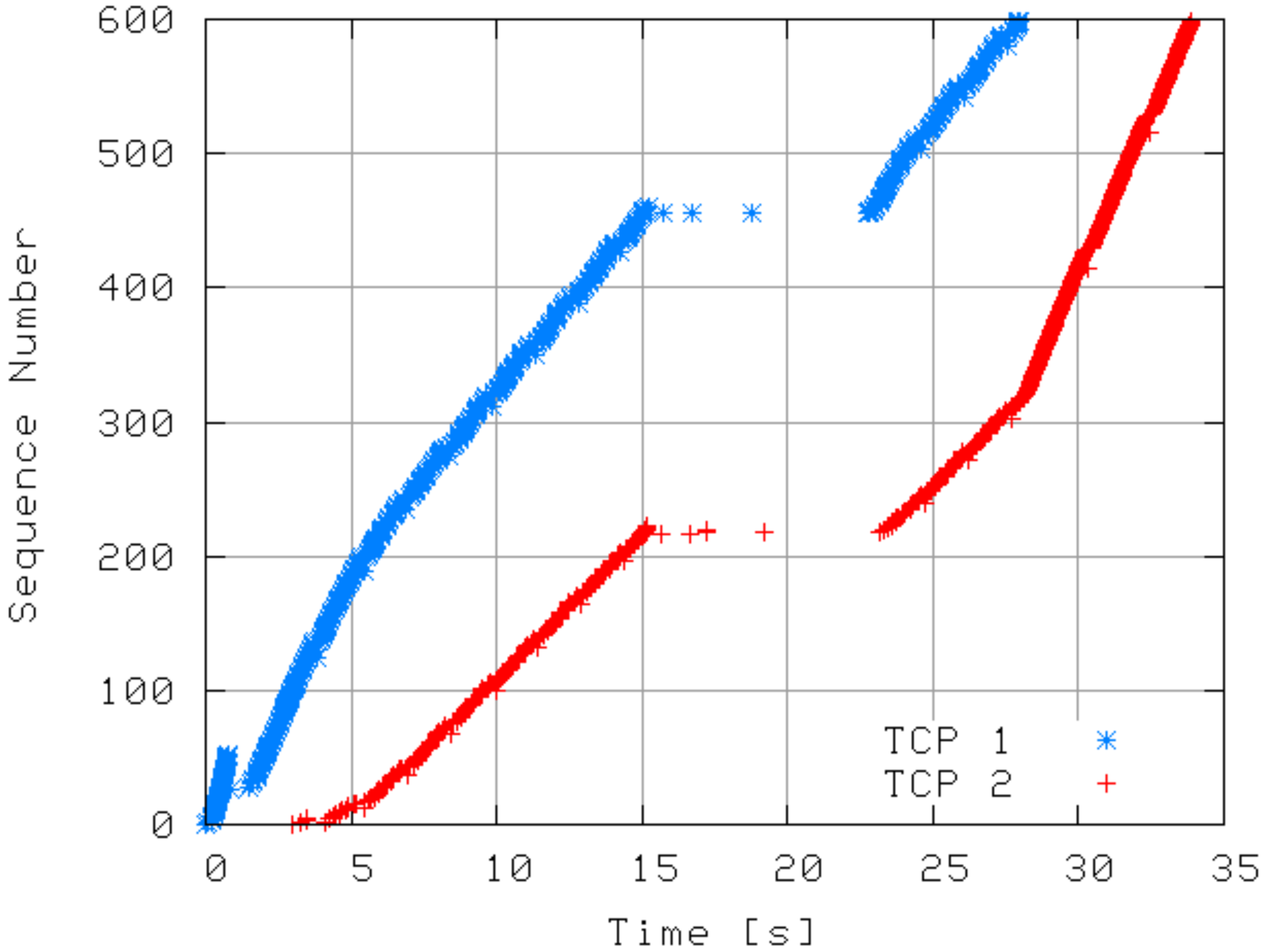
- cwnd segments, each with MSS bytes sent in one RTT

Fast Recovery Example

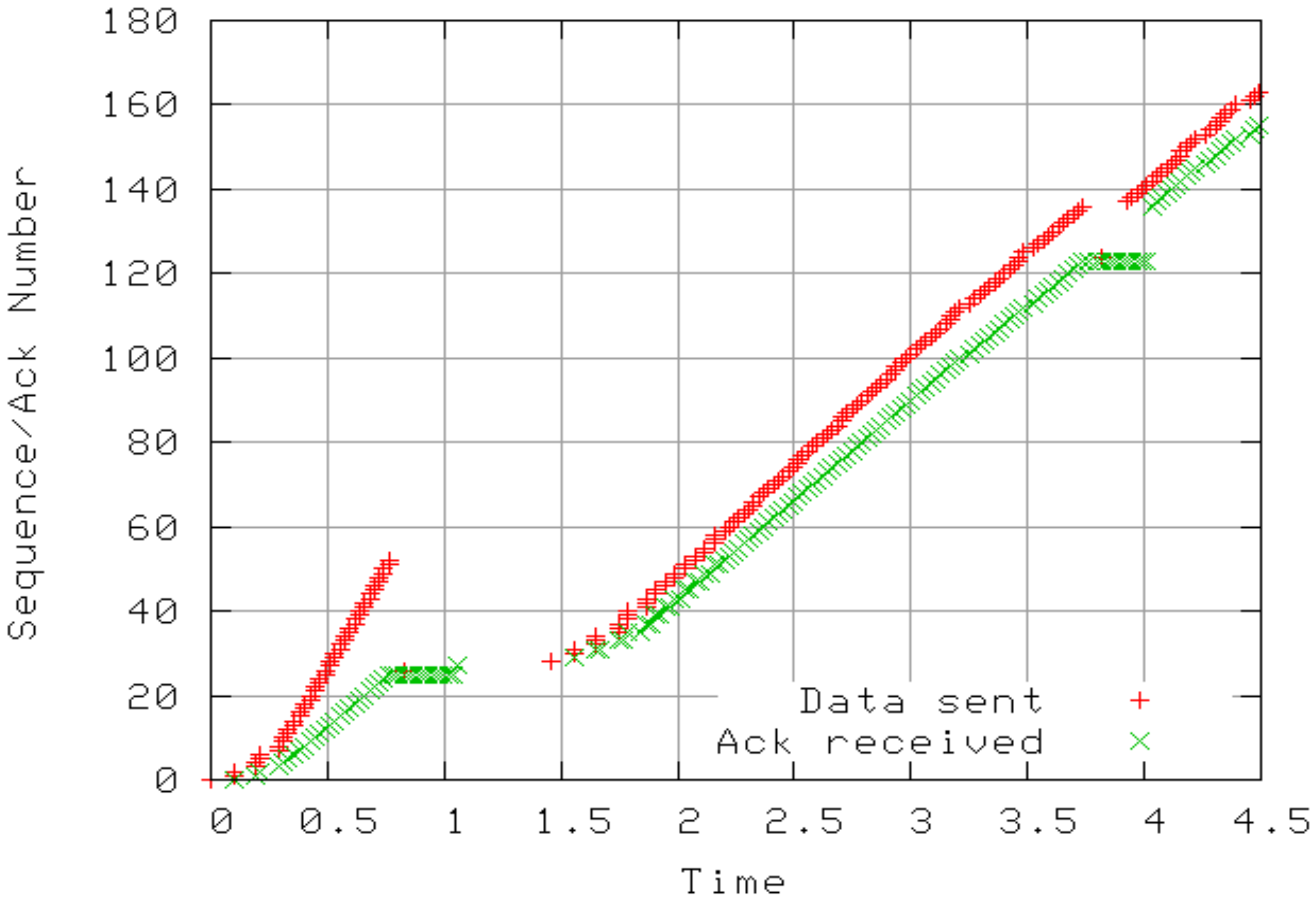


- cwnd = 6; in congestion avoidance

Sequence Number Plot (Simulation)



Seq. Number Plot (Simulation) zoom



TCP Flavors / Variants

- TCP Tahoe
  - Slow Start
  - Congestion Avoidance
  - Timeout, 3 duplicate acks → cwnd = 1 ⇨ slow start
- TCP Reno
  - Slow-start
  - Congestion avoidance
  - Fast retransmit, Fast recovery
  - Timeout → cwnd = 1 ⇨ slow start
  - Three duplicate acks → Fast Recovery,
Congestion Avoidance

Extensions

- Fast recovery, multiple losses per RTT ⇨ timeout
- TCP New-Reno
  - Stay in fast recovery until all packet losses in window are recovered
  - Can recover 1 packet loss per RTT without causing a timeout

- Selective Acknowledgements (SACK) [rfc2018]
  - Provides information about out-of-order packets received by receiver
  - Can recover multiple packet losses per RTT

Additional TCP Features

- Urgent Data
  - Nice for interactive applications
  - In-Band via urgent pointer

- Nagle algorithm
  - Avoidance of small segments
  - Needed for interactive applications
  - Methodology: only one outstanding packet can be small

Summary

- Reviewed principles of transport layer:
  - Reliable data transfer
  - Flow control
  - Congestion control
  - (Multiplexing)
- Instantiation in the Internet
  - UDP
  - TCP