Internet Transport Protocols UDP / TCP

Prof. Anja Feldmann, Ph.D.

What do you know already?

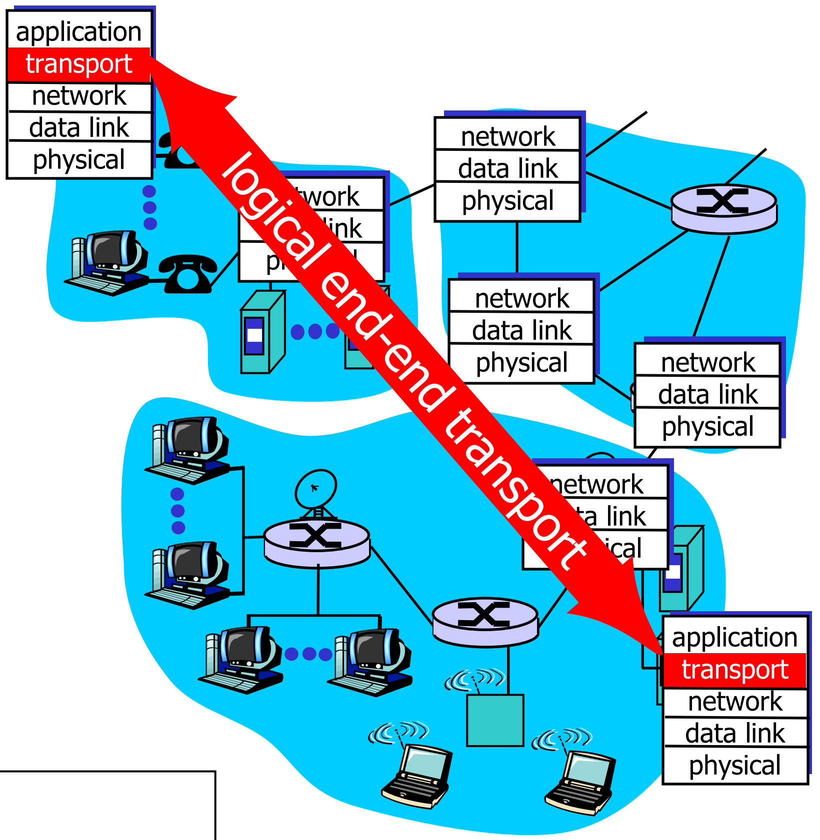
- Transport layer functionality?
  - Transport data between applications
  - Multiplexing to apps
  - Transport with and without connection
- Difference to network layer?
  - Connection between processes (rather than hosts)
- Transport layer protocols?
  - UDP, TCP
- UDP functionality (good for?)
  - Simple but unrealiable
  - good for fast&short, stateless transmissions
  - e.g., live streaming, DNS, ...
- TCP functionality
  - Reliable byte stream
  - Flow control, congestion control, ...
  - but not, e.g., bandwidth guarantees, etc.
  - e.g., HTTP

Transport Layer: Outline

- Transport-layer services
- Multiplexing and demultiplexing
  (data stream to correct app via headers)
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control (be nice to sender)
  - Connection management
- Principles of congestion control
  (be nice to network)
- 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



What concepts are needed?

- Sockets identified by ports to multiplex to apps at host
- According „identifiers" in packet headers: src ID = source multiplexor (also needed at desitination), dst ID = service selector

Sockets: interface to applications

Socket API

- Introduced in BSD4.1 UNIX, 1981
- Explicitly created, used, released by…?
- … applications
- Client/server paradigm
- Two types of transport service via socket API?
  - Unreliable datagram ("packet")
  - Reliable, byte stream-oriented

socket

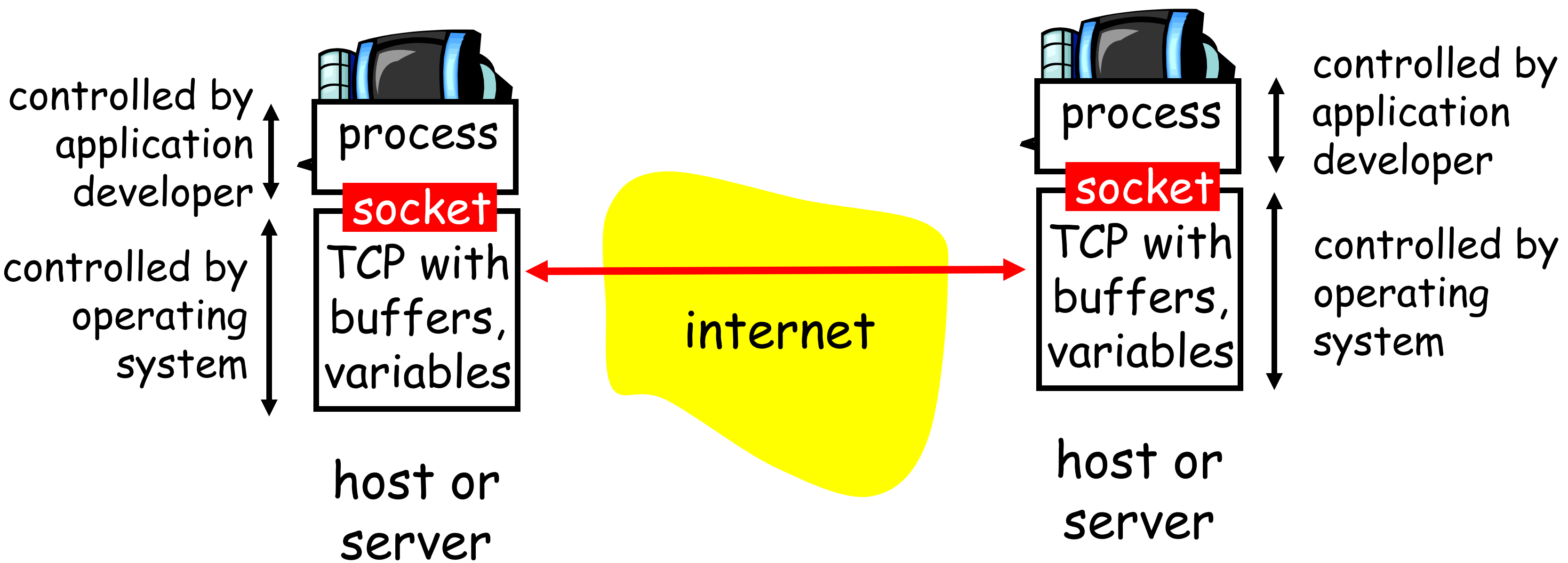
A *host-local*, *application-created/owned*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

E.g. Java?

- *DatagramSocket mySocket = new DatagramSocket();*
- Opens UDP socket, and transport layer automatically assigns a port number > 1023 (why needed at all on client side? why random okay for client side?)
- For TCP connection: *Socket clientSocket = new Socket („hostname", „dst port")*
- TCP server process then opens new socket upon request: *Socket conSocket = welcomeSocket.accept();*

Sockets and OS

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



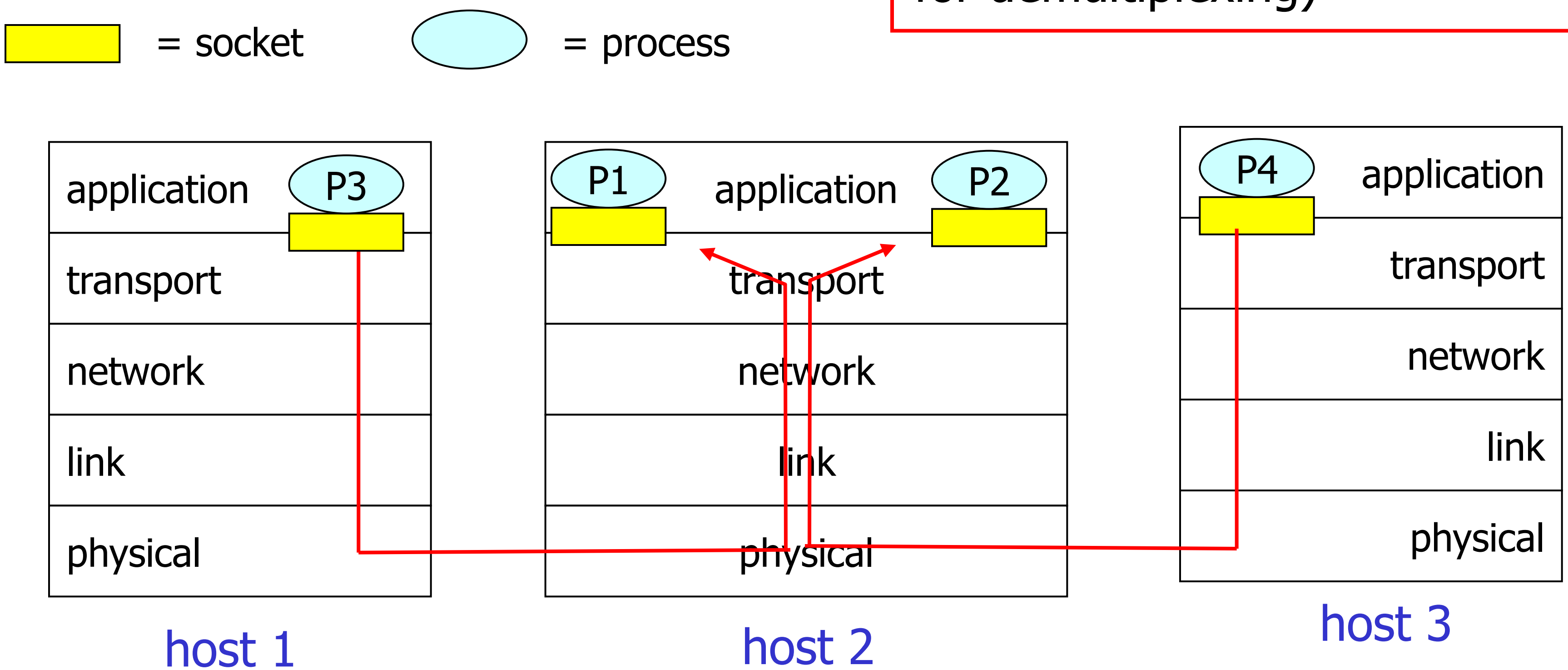
Multiplexing/Demultiplexing

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)

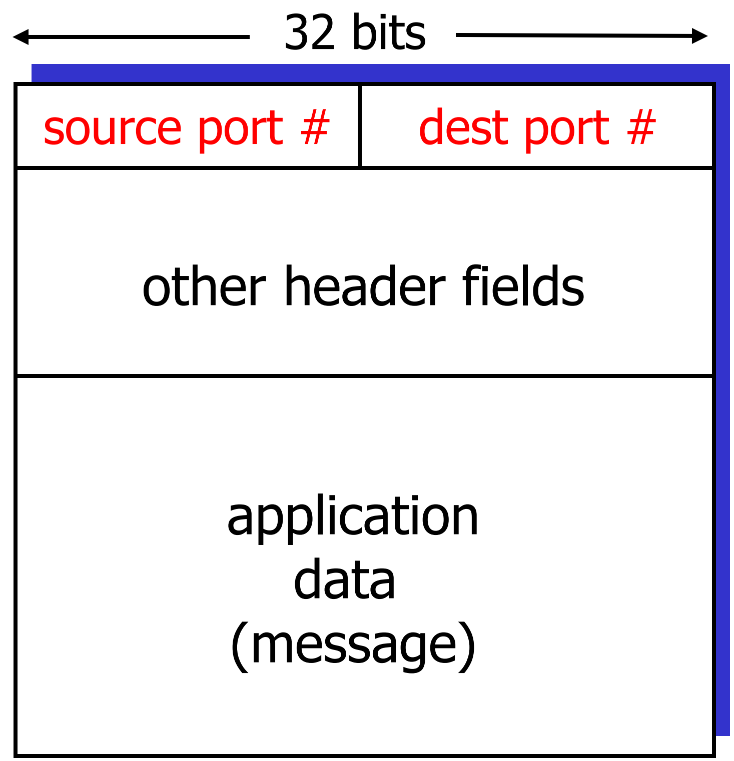


How should packet header look like?

Multiplexing/Demultiplexing

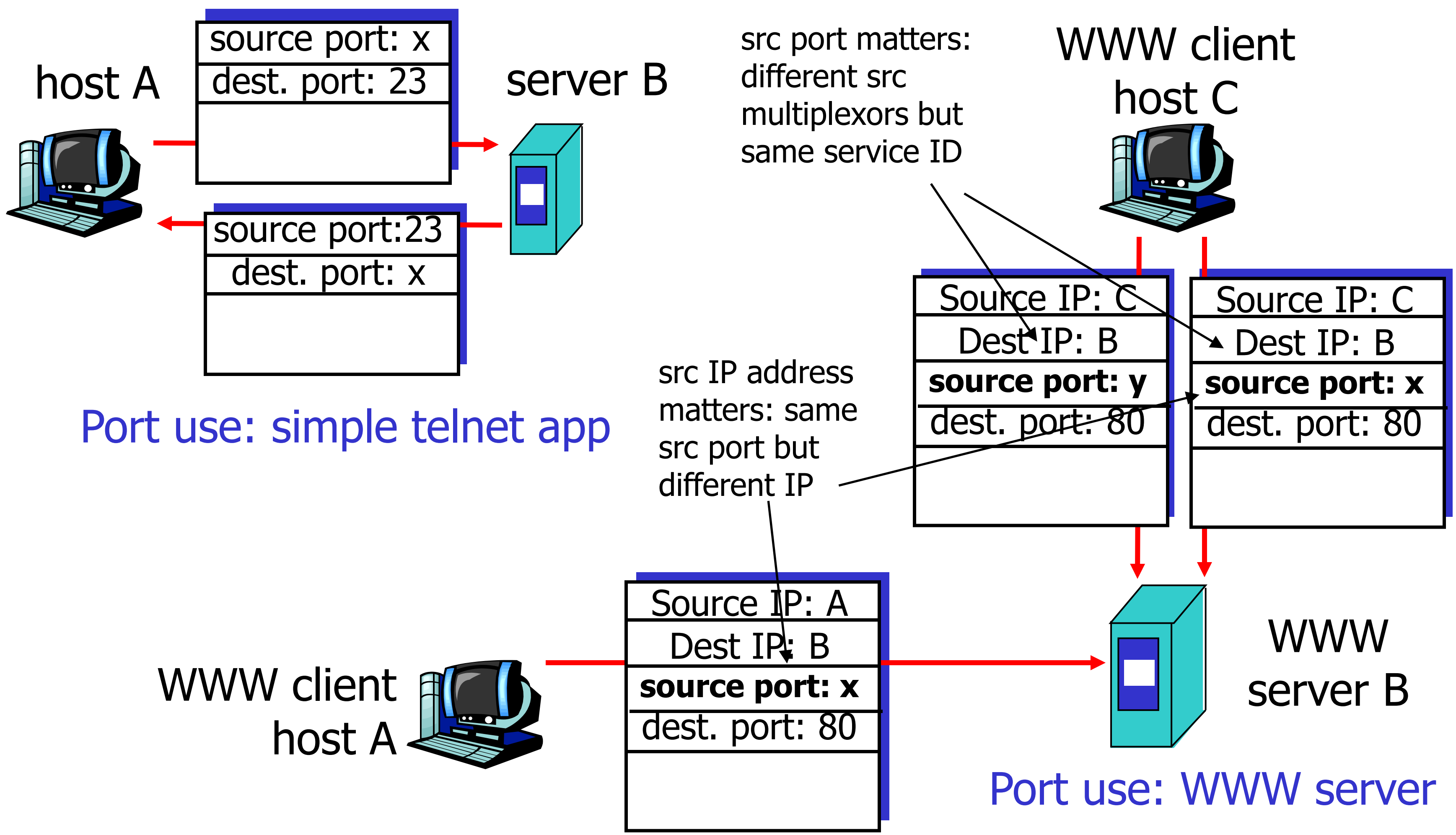
Multiplexing/demultiplexing: how to achieve? (e.g., infos needed?)

- Based on sender, receiver port numbers, IP addresses
  - Source, dest port #s in each segment (= packet in transport layer)
  - 1024 well-known port numbers for specific applications: clear where to obtain service! (check on Linux how many are open: /etc/services)
  - Example ports?
  - ftp = 21, telnet = 23, http = 80, ...
  - Why do IP addresses matter?
  - Different requesting hosts can have same ports...! (but UDP and TCP differ on how dest processes are shared)



TCP/UDP segment format

Multiplexing/Demultiplexing: Examples



Remark 1: Sockets do not always constitute an own process, but can be managed by a thread.
Remark 2: In non-persistent HTTP, each request/response pair is a new socket/TCP connection.

UDP: User Datagram Protocol [RFC 768]

- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost (no ACKs…)
  - 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
- I can implement my own extensions (TCP?) with it...

Other disadvantages of UDP? E.g., sometimes filtered at firewalls...

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 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
- I can implement my own extensions (TCP?) with it...

Examples for UDP? Youtube, live streaming...

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 application
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

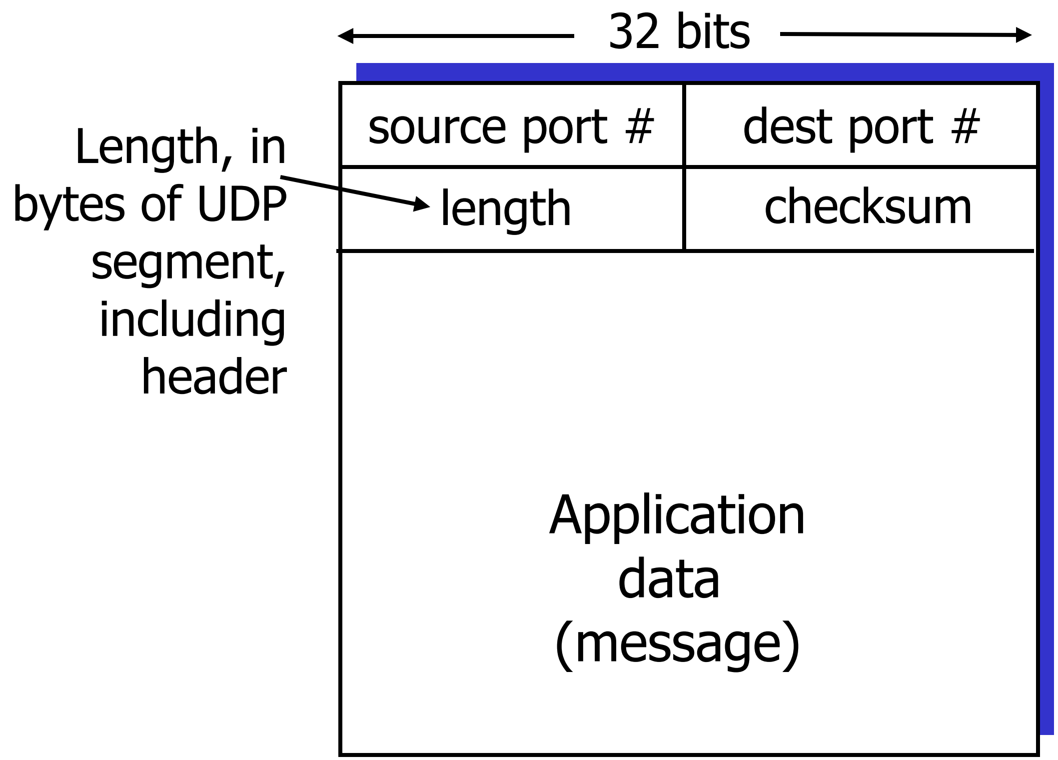
Why is there a UDP?

- 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
- I can implement my own extensions (TCP?) with it...

What about HTTP? **Spec requires reliable transport (e.g., many objects do not fit into one packet)**

UDP: More

- Each user request transferred in a single datagram
- UDP has a receive buffer but no sender buffer: app packets given to UDP are immediately sent (no delay to set up connection, flow/congestion control, fill packet, ... like in TCP)
- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS (fast!), SNMP (network management packets need to get through even in "troublesome" times), NFS
  - Routing updates (loss no problem, periodic anyway)
  - Faster and robuster? (HTTP slow because not UDP?)
- Reliable transfer over UDP? Add reliability at application layer



UDP segment format

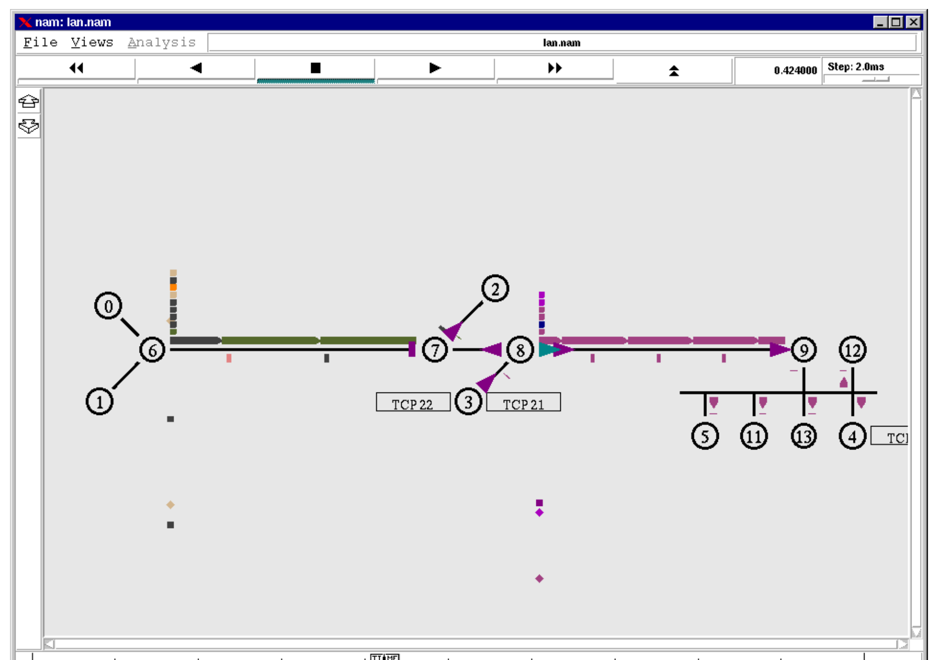
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order *byte stream:*
  - No "message boundaries"
  - Flush! (Why?)
- Pipelined:
  - TCP congestion and flow control set window size
- 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
- Flow controlled:
  - Sender will not overwhelm receiver
- Congestion controlled:
  - Sender will not overwhelm the network
- Implication for header: new fields?

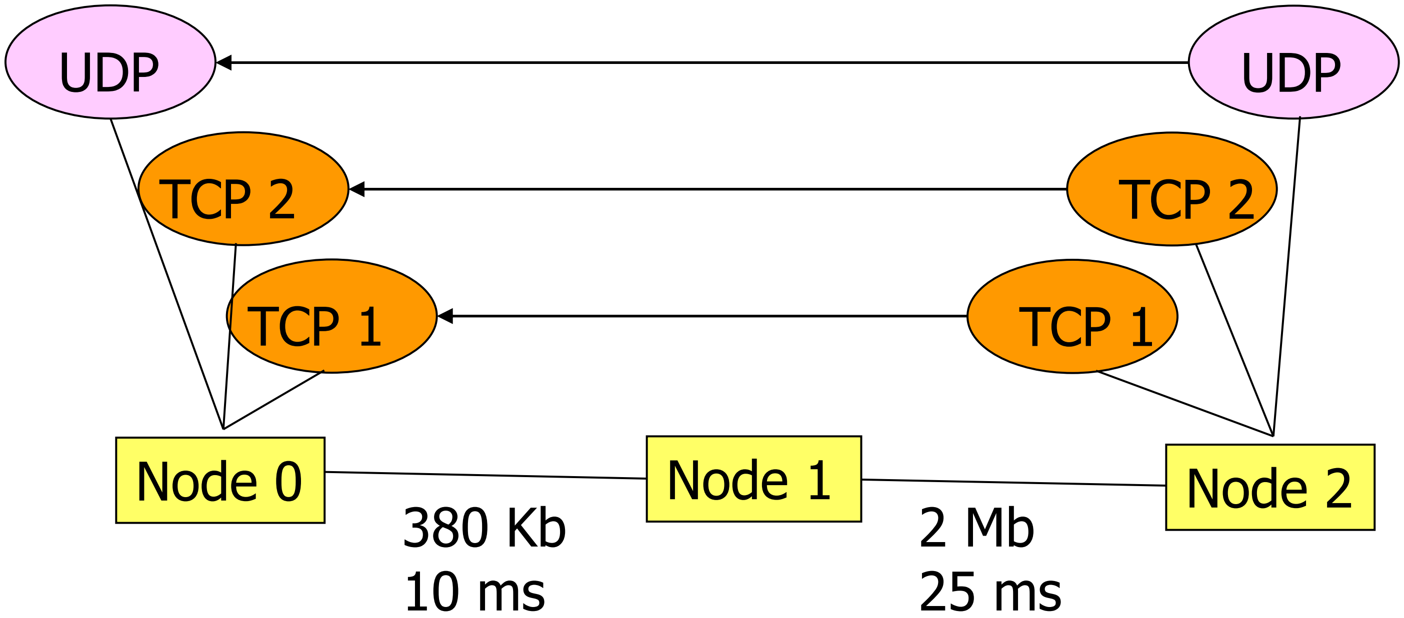
Simulating Transport Protocols

- TCP dynamics are complex! (and interesting ☺)
- Help? Network simulator!
- Examples:
  - Network Simulator (NS), SSFNet, …
- Animation of NS traces via NAM (Network Animator)
- Try it!
  - Queues, packet drops, bit-durations, transmission times, ...



Simulating Transport Protocols

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

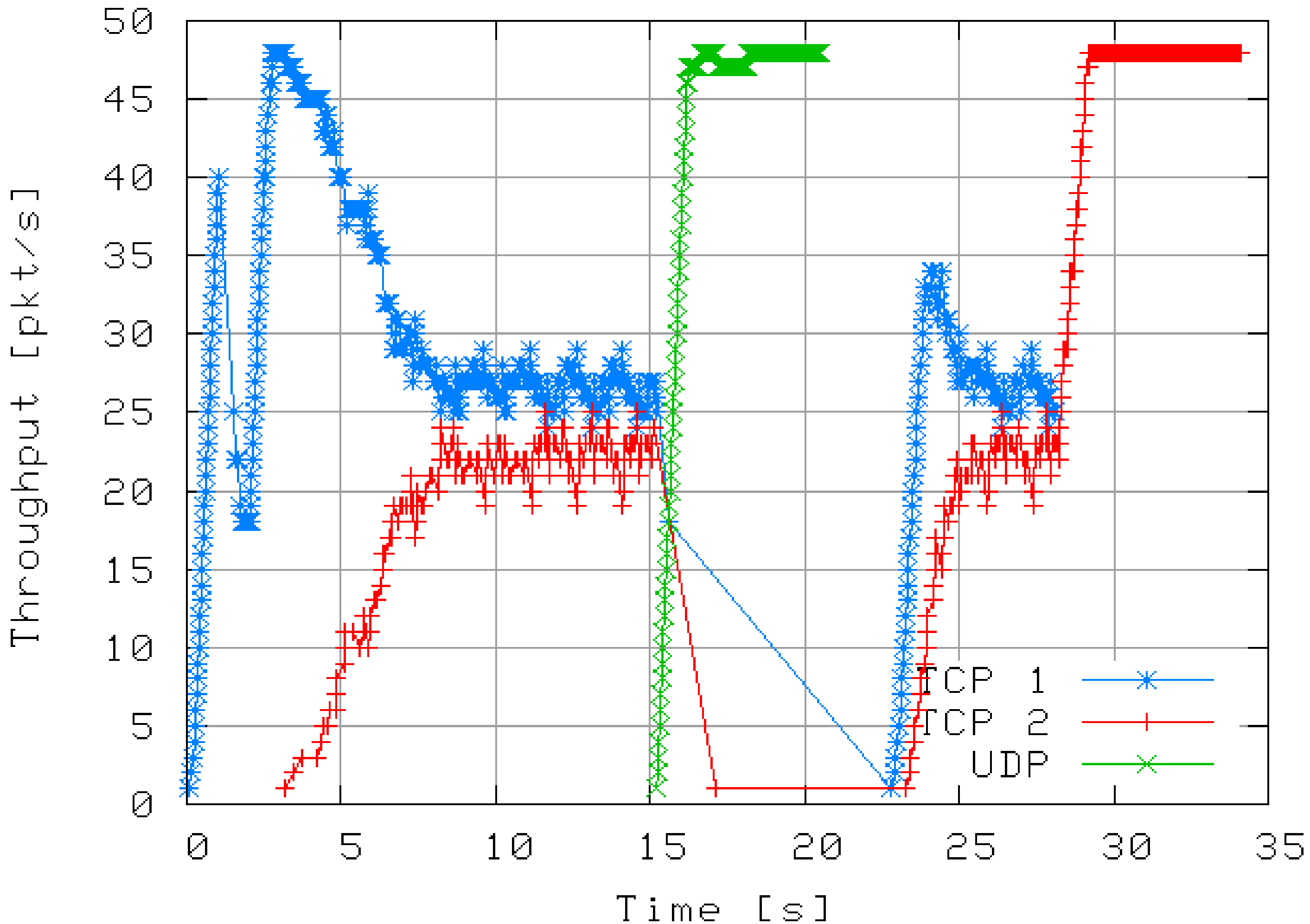


- TCP1 starts at time 0 seconds, TCP2 at time 3 seconds
- UDP starts at time 15 seconds
- Dynamic allocation of resource over time?!

Simulation Results

(Try other scenarios yourself with ns2!)

- Takeaways?
  - TCP allocates resource well (first whole, than half) and fair
  - UDP gets all...
  - ...



Question: Is TCP always fair…?!

Sometimes, but not always!

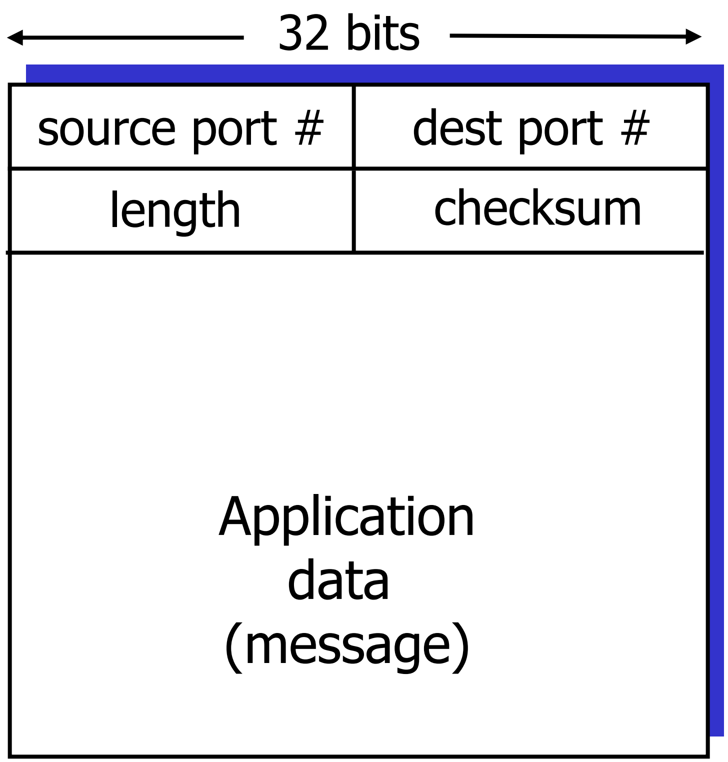
Depends on RTT, reaction time, ...:
e.g., faster reacting participants fill out free slots quicker!

For users: Depends on number of parallel connections...

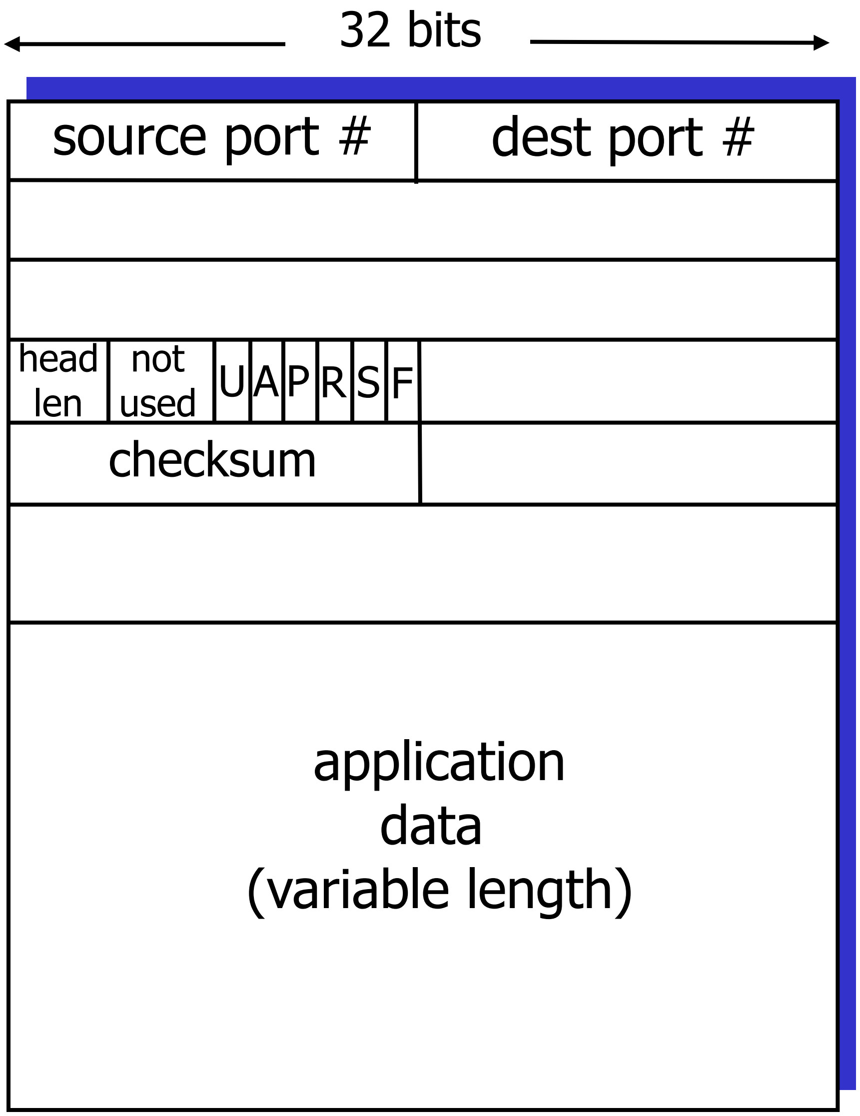
(Recall: UDP sometimes unfair share...)

Question: TCP Segment Structure?

What do we need compared to UDP?
Recall:



Becomes more complicated…



TCP Segment Structure

URG: urgent data
(generally not used)

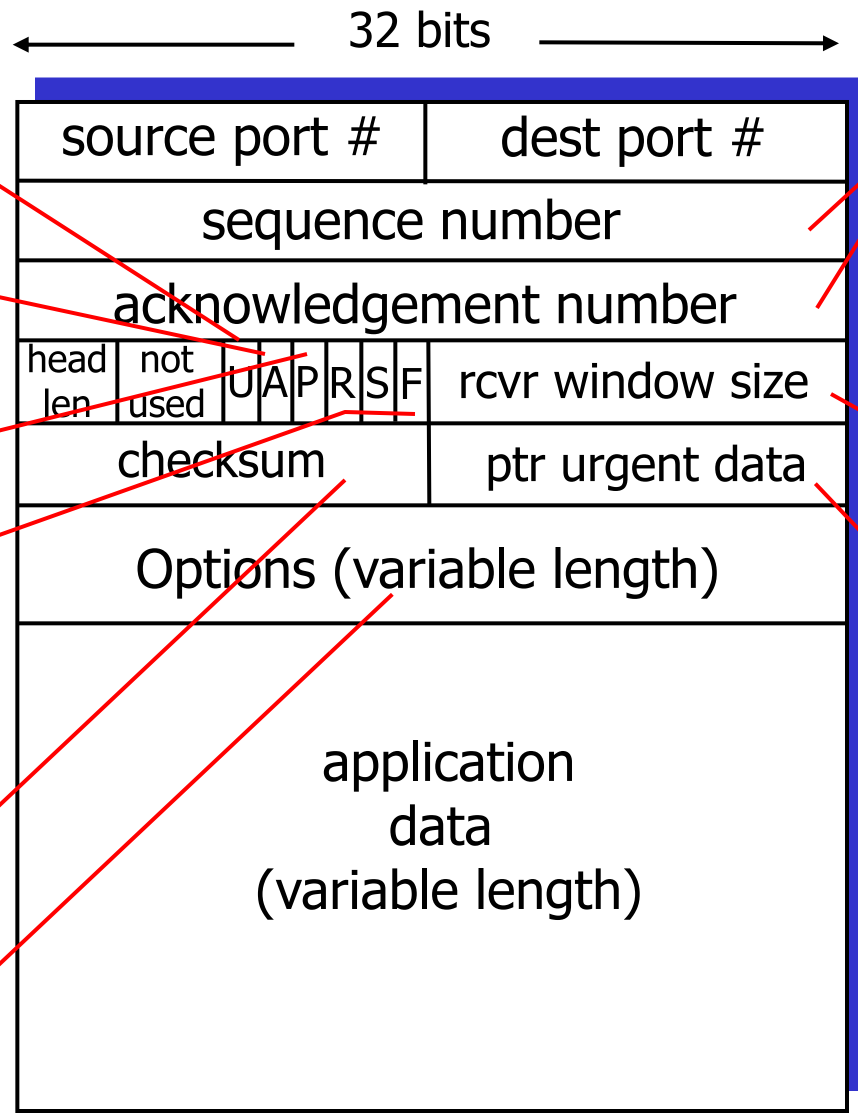
ACK: ACK # Valid
(why?
=do we ACK something!)

PSH: push data
to application now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet
checksum
(as in UDP)

e.g., for partners to
agree on max
segment size (MSS)



counting
by bytes
of data
(bytestream,
not segments!)

"non-stop bytes"
rcvr willing
to accept

Pts to last byte
of urgent data
(not used)

How to send 5 bits
with TCP?
Make it a byte (pad
it) ☺

Question: TCP Packet Length and MSS?

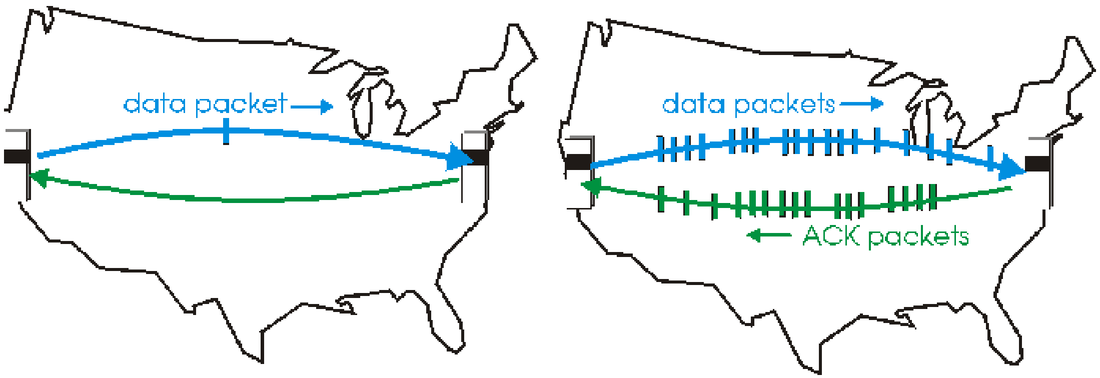
- Unlike UDP, no payload length in TCP header: why?
  - Can compute it from total IP datagram length by subtracting TCP header etc.
- How large is a TCP packet?
  - Unlike UDP, it‘s „managed“ (TCP cuts bytestream into packets)
  - Usually data is split into MSS (max segment size) parts
  - Last packet can be smaller...
  - Sometimes payload even one byte only (e.g., Telnet, or see TCP silly window syndrome later)! Overhead!
- Distribution of packet sizes in the Internet?
  - Many small and many large ones due to ACKs (one-directional connections).
- How does the MSS agreement work?
  - Both parties can suggest an MSS during connection setup
  - Typically: 1024 or 536 bytes if non-local destination (IP packet then 20+20 bytes larger for headers)
- Better large MSS or small MSS?!
  - The larger the better (close to MTU of interface if dest IP address is a local one): less „header overhead“, less „per packet“ store-and-forward overhead...
  - ... but should not be fragmented by lower layers later! (because: different paths of subpackets but entire retransmissions, etc.)

Question: TCP Packet Length and MSS?

- Why fragmentation on layer 3 and layer 2?
  - Historically: not each layer 2 protocol supported own fragmentation: IP need to do it
  - Nowadays, almost always supported, so in IPv6 it's an option
  - Moreover, it always make sense to have path-MTU mechanisms, to avoid further fragmentations along the paths...

TCP Reliability? Simplest Solution?

Stop-and-Wait



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

TCP Reliability? Seq. #'s and ACKs!

Seq. #'s:

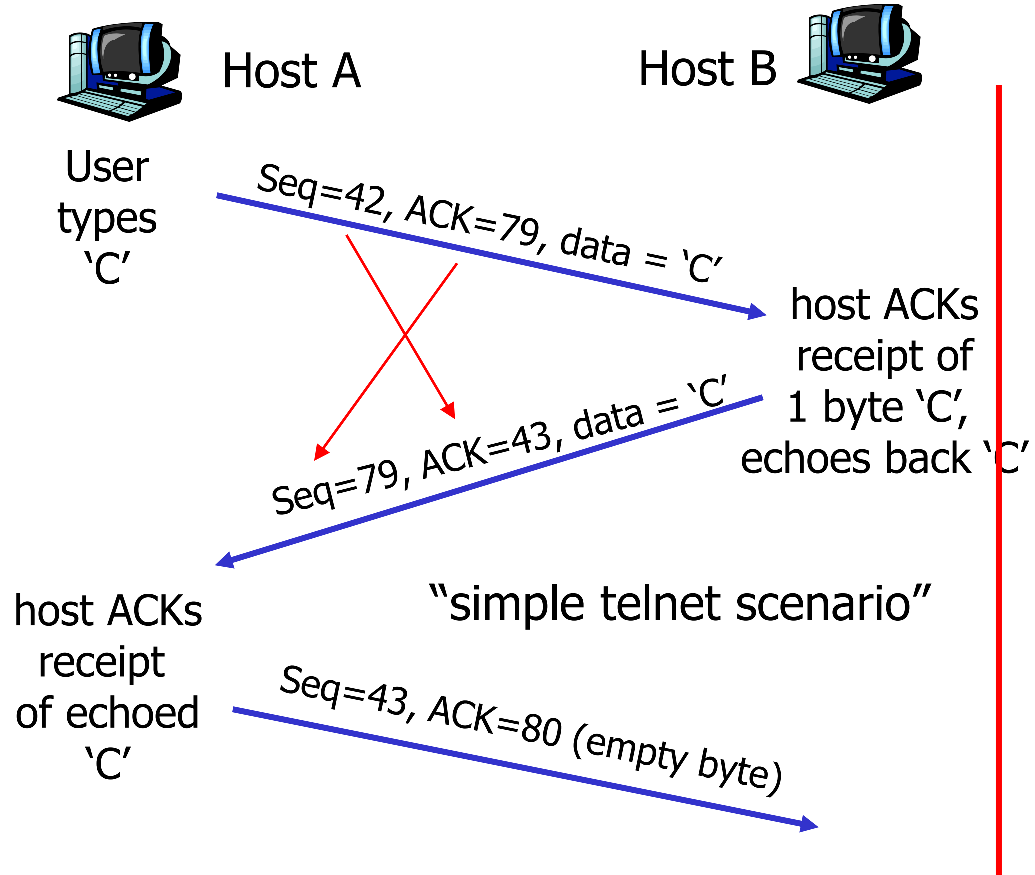
- **Byte stream**
  "Number" of **first byte** in segment's data

ACKs:

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

Q: How does receiver handle out-of-order segments?

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



ACK confirms *all* previous bytes (always send ACK when receiving packet, but maybe with old number).

TCP Reliability? Seq. #'s and ACKs!

Seq. #'s:

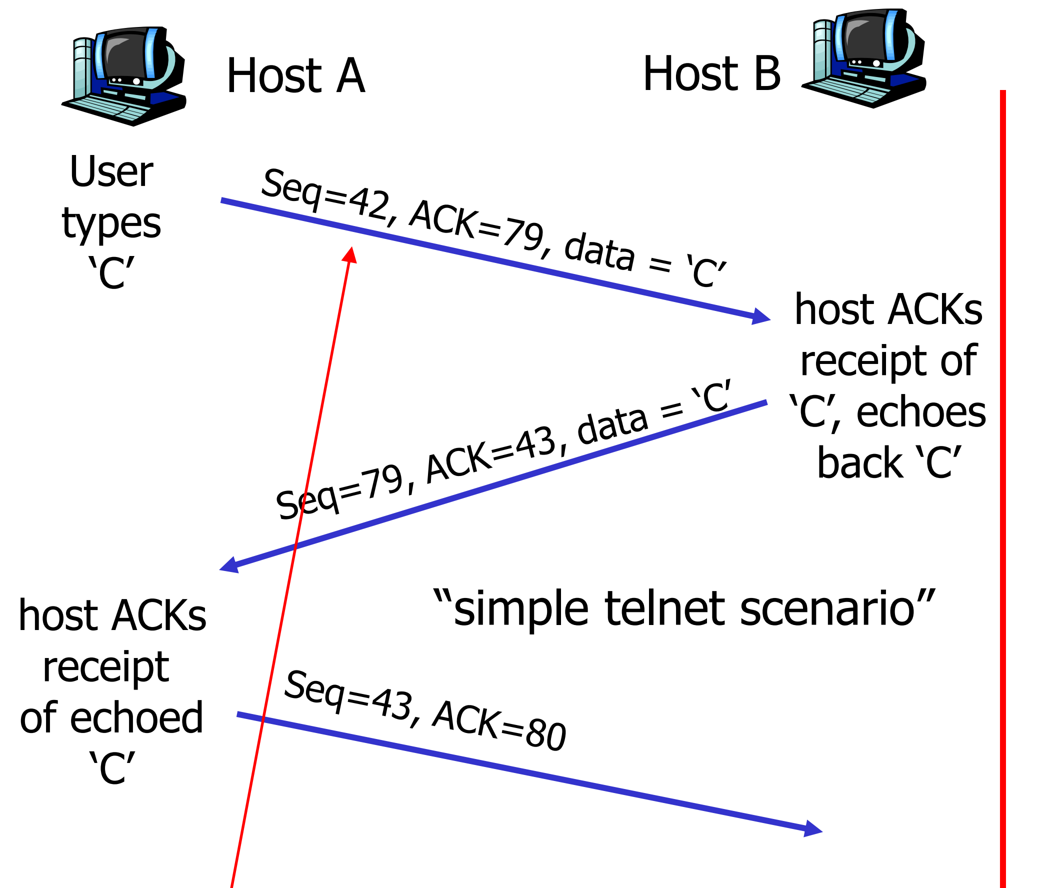
- **Byte stream** "Number" of **first byte** in segment's data

ACKs:

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

Q: How does receiver handle out-of-order segments?

- A: TCP spec doesn't say, – up to implementer (e.g., throw away or buffer until gap filled)



How is first seqno chosen?„Randomly"!
Why? To avoid confusion with older connections (if packets still on fly)!

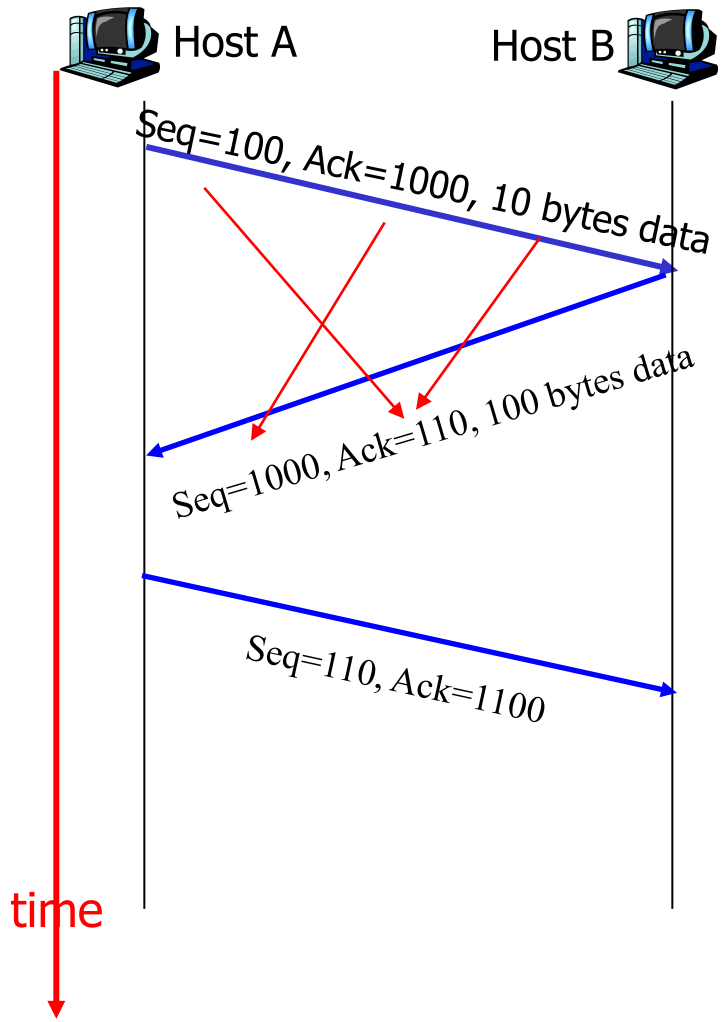
Example with Larger Packets

Seq. #'s:

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

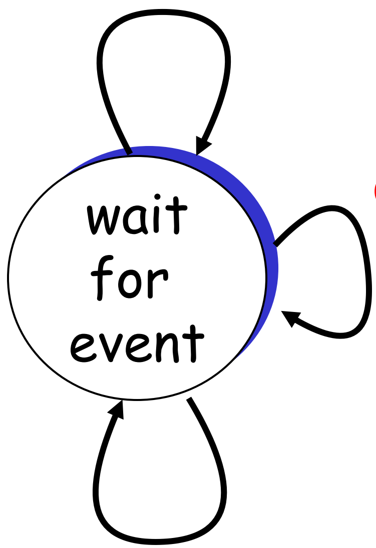
ACKs:

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



TCP: Reliable Data Transfer by Simple State Machine (Sender)

event: **data received** from application above
create, send segment



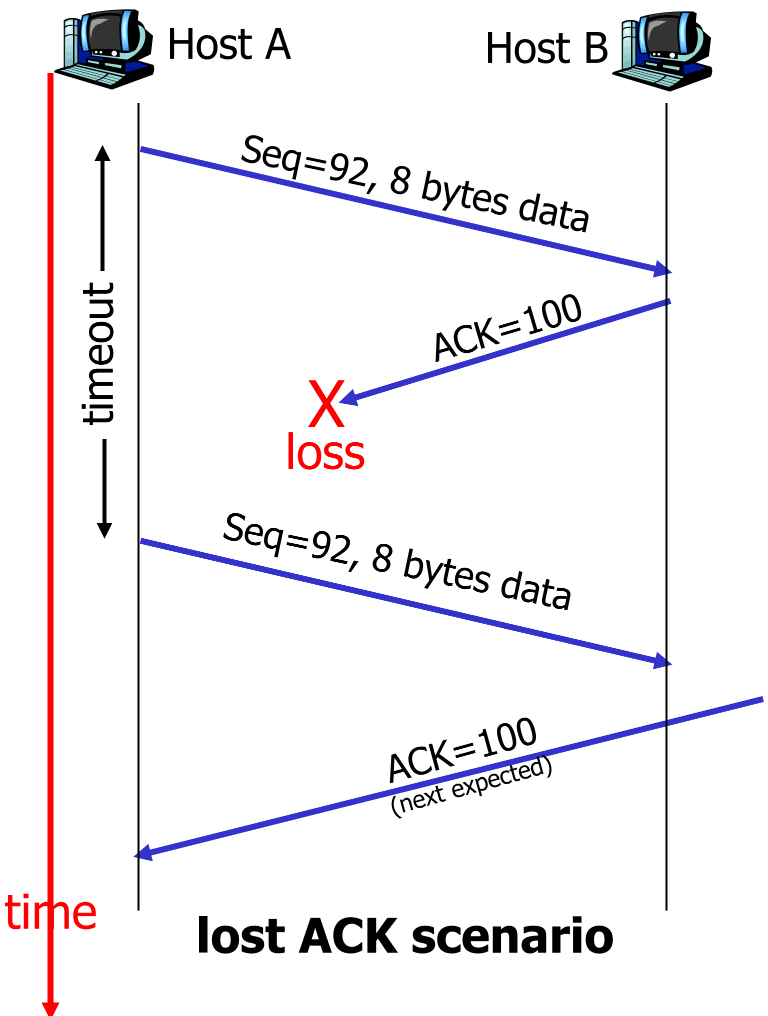
event: timer **timeout** for segment with seq # y
retransmit segment

event: **ACK received**, with ACK # y
ACK processing

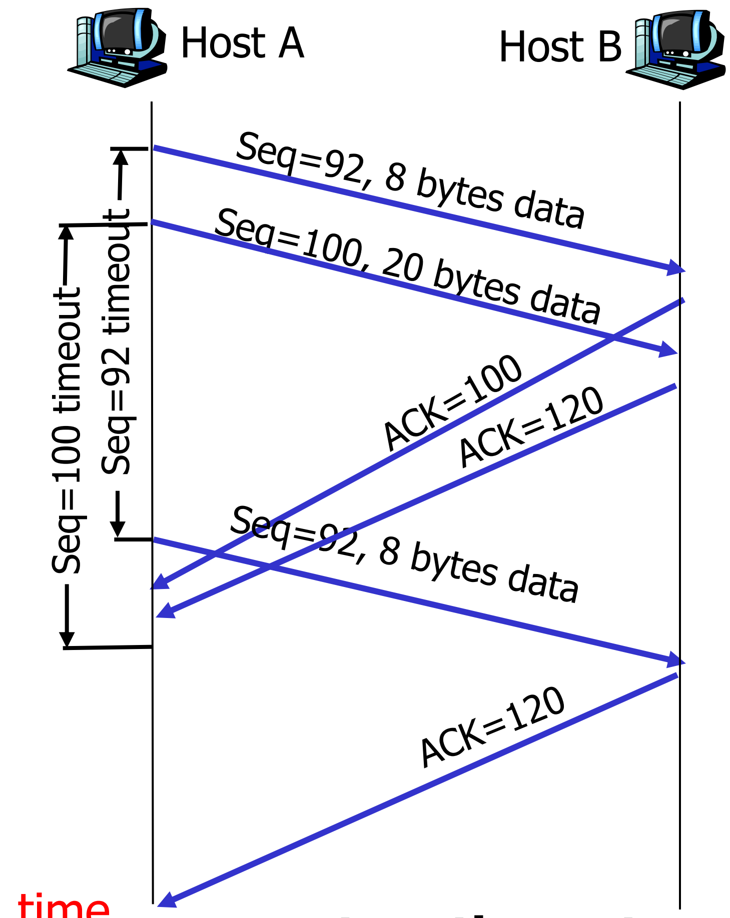
- Simplified sender assumption
  - One way data transfer
  - No flow, congestion control
- Packet loss detection?
  - Retransmission timeout
  - Fast retransmit (why?)
    - Three duplicate ACKs (no congestion as data still gets through!)

- Retransmission mechanism (at timeout or dup ACK)
  - ARQ (automated repeat request, e.g., at timeout): Go-Back-N (allow N unACKed packets, then send all again starting from first unACKed packet / loss => simple receiver with buffer size 1), selected retransmissions (receiver continues accepting and ACKing packets after a loss, but ACK's the last before gap: sender will send unACKed and then continue where stuck before!)

TCP: Retransmission Scenarios



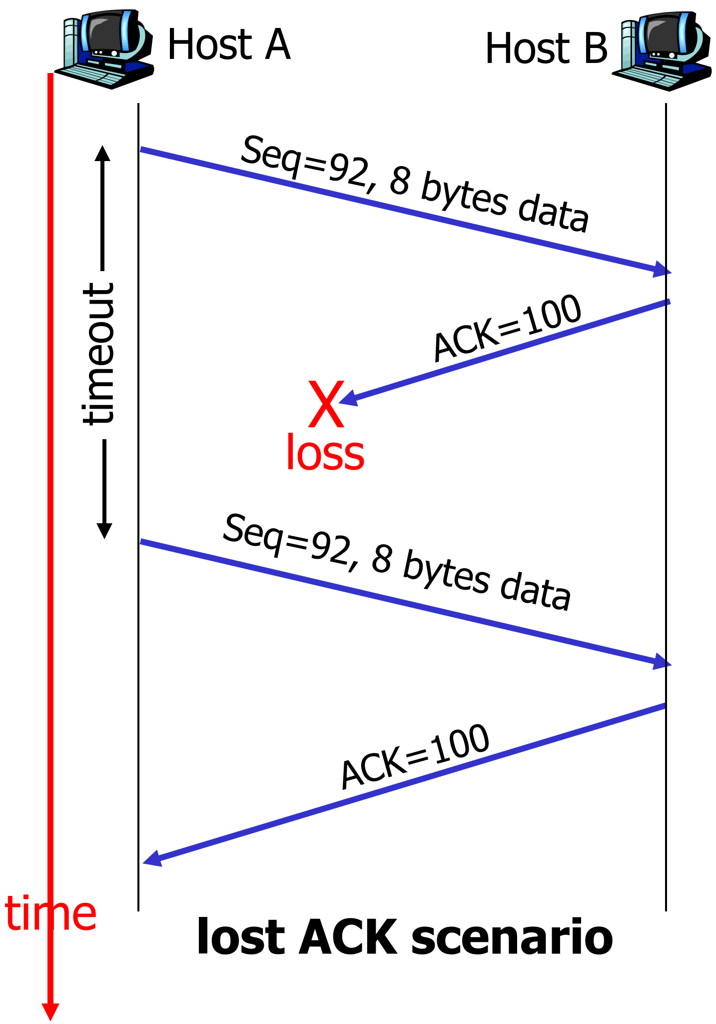
lost ACK scenario



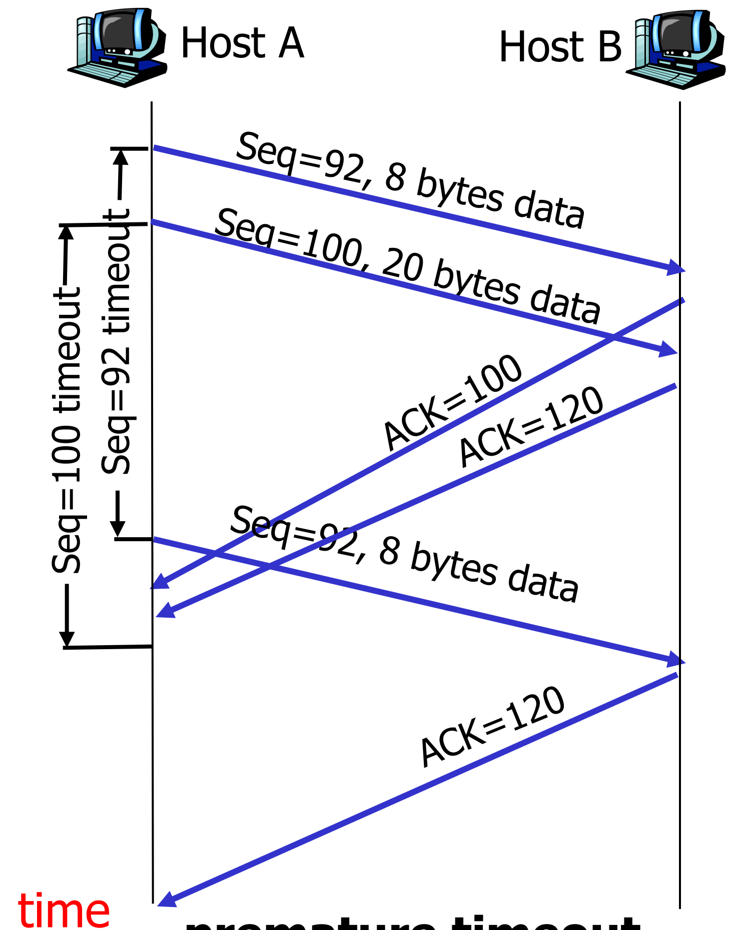
premature timeout,
cumulative ACKs

Are there many TCP losses?!
Yes, TCP always entails losses (see later)! Try yourself!

TCP: Retransmission Scenarios



lost ACK scenario



premature timeout, cumulative ACKs

Question: Can sender distinguish whether data or ACK got lost?
No...

TCP (cumulative) ACK Generation [RFC 1122, RFC 2581]

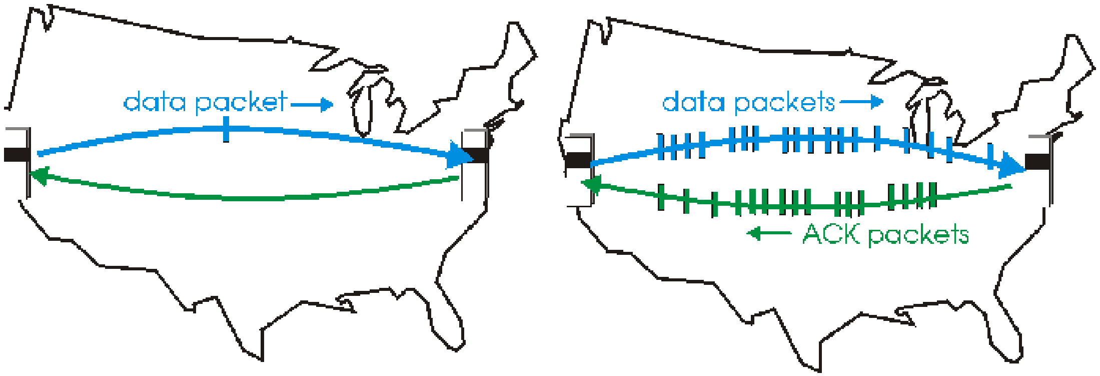
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	**Delayed ACK**. Wait up to 500ms for next segment. If no next segment, send ACK. Why? Reduces ACK traffic (cumulative…)
Arrival of in-order segment with expected seq #. 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: no congestion?)
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Some further thoughts on ACKs...?

- Alternative protocols?
- Cumulative ACKs vs Selective Repeat?
  - Selective better when large windows and large RTT x bandwidth product (=> many packets „on the fly", repeat all packets in big pipeline)...
  - ... but then receiver has to ACK packets individually, sender and receiver no longer synchronous, more complex receiver, more sequence numbers needed, etc.
- What about explicit NAKs („please repeat number 5"), etc.?
- See [Kurose]

Pipelining (many packets "on the fly")

- Why needed?
- Stop-and-Wait vs Pipelining: throughput depends on latency!



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

- Question: How many sequence numbers are needed for stop-and-wait protocol?
- 1 Bit enough! (retransmit or new...)

TCP Retransmission Timeout

- Recall: Timeout as method to detect loss!

- But: what is a good timeout value? If receiver far away: should be larger...
- ... and should depend on connection state, be robust to fluctuations!

- TCP uses one timer for one pkt only
  - i.e., not one for each non-ACKed packet (think of it as timer for oldest non-ACKed packet, in reality more complicated)

TCP Retransmission Timeout

- Retransmission Timeout (RTO) calculated dynamically
  - Why dynamic?
  - Network is dynamic! Route changes, high load, etc. => timeout should reflect that packet was really lost (independent of route)!
  - Based on Round Trip Time estimation (RTT) (why not oneway?)
  - Wait at least one RTT before retransmitting
  - Importance of accurate RTT estimators?
    - Low RTO → unneeded retransmissions
    - High RTO → poor throughput
  - RTT estimator must adapt to change in RTT
    - But not too fast, or too slow!
  - Spurious timeouts (e.g., wrong RTO expiry due to aggressive timer update in case of dynamic network changes due to handover/mobility)
    - "Conservation of packets" principle violated – TCP in inefficient slow start mode with small windows but more than a window worth of packets in flight!
    - E.g., Eifel detection algorithm to circumvent inefficiencies

Retransmission Timeout Estimator

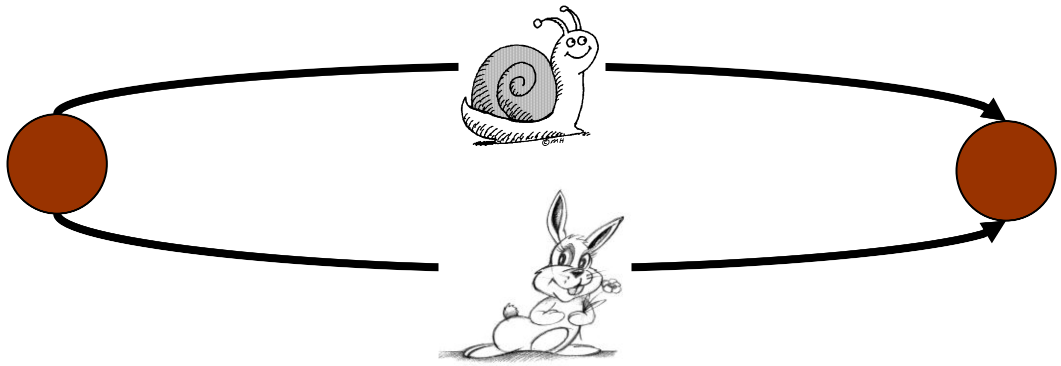
- Round trip times exponentially averaged (adapt but not too fast):
  - New RTT = α (old RTT) + (1 - α) (new sample)
  - α = 0.875 for most TCP's
- Retransmit timer set to β RTT, where β = 2
  - Every time timer expires, RTO exponentially backed-off
- Key observation: At high loads round trip variance is high
- Solution (currently in use): account for variance!
  - Base RTO on RTT and standard deviation of RTT:
    RTT + 4 * rttvar
  - New rttvar = α (old rttvar) + (1-α) * dev
    - dev = linear deviation over sample (also referred to as mean deviation)
    - inappropriately named – actually smoothed linear deviation
  - RTO is discretized into ticks of 500ms (RTO >= 2 ticks)
    - Initially: 3 sec (actively reload in browser can be faster than wait for timer timeout...)
    - High because of OS interrupts (also inaccurate)...

$$\mathrm{MD} \equiv \frac{1}{N}\sum_{i=1}^{N}|x_i - \bar{x}|,$$

Question: Why measure RTT instead of simple delay from sender to receiver?

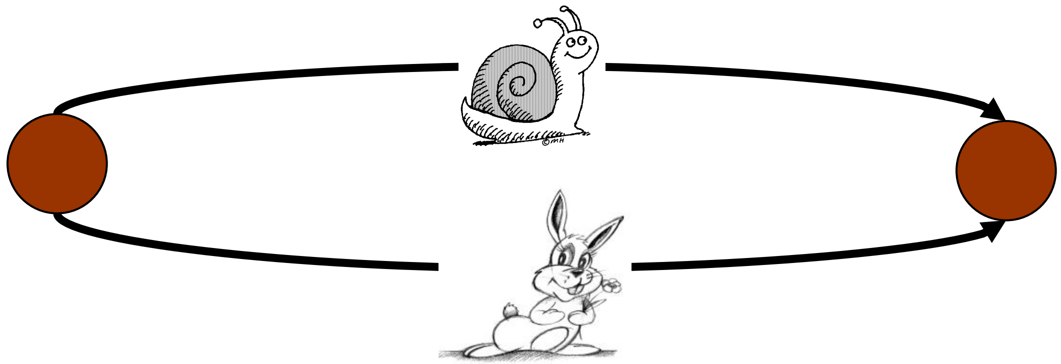
Can be measured locally (without clock synchronization)...

Example



- What happens to TCP throughput?
- High variance (some packets fast some slow) => high RTO (late retransmissions when needed)
- Many packets out of order, so many (unnecessary?) retransmissions (duplicate ACKs?)
- Throughput in the order of slow link only...?
- Try it out! ns2, tcpdump, ...

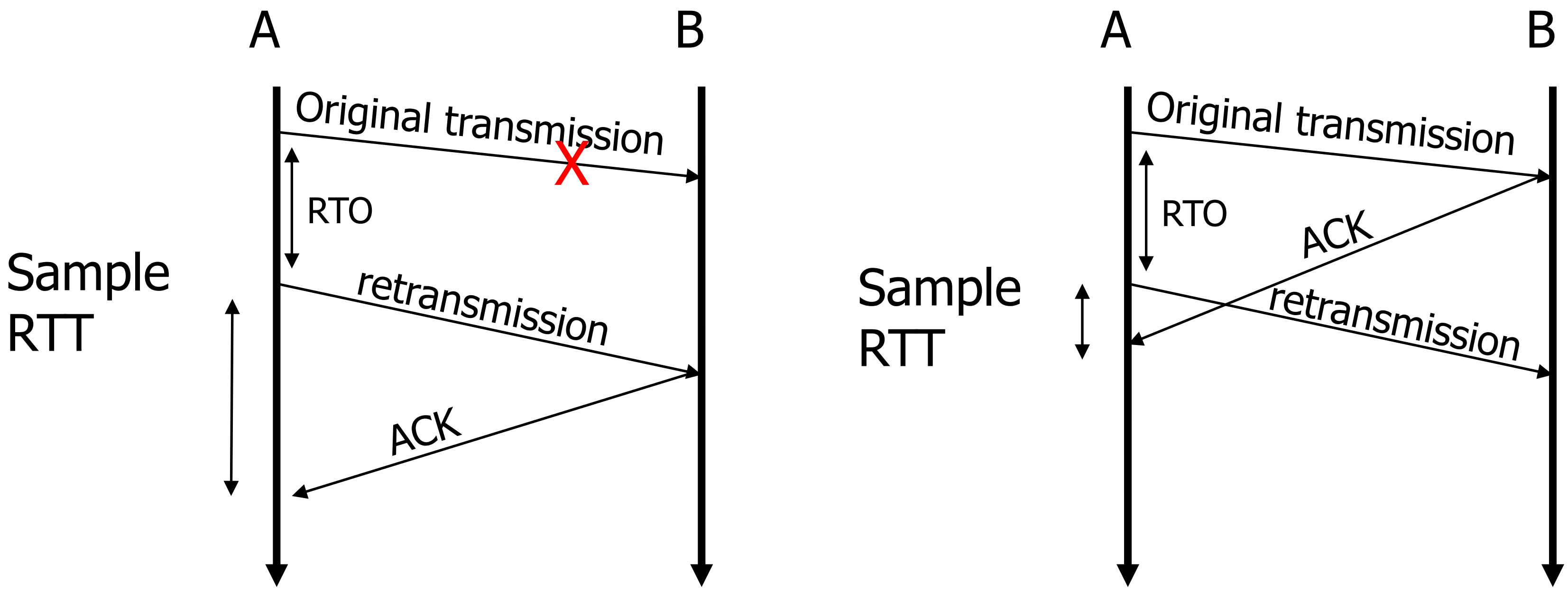
Q&A



- How likely is it that packets take different paths?
- Unlikely, only over larger time frames...
- ... and if, than most likely inside ISP only (for load balancing) (late retransmissions when needed)
- How likely is it that to-path different from backward-path?
- Very likely! E.g., hot potato routing, see later!

Retransmission Ambiguity

- How to sample RTT? Under retransmissions??

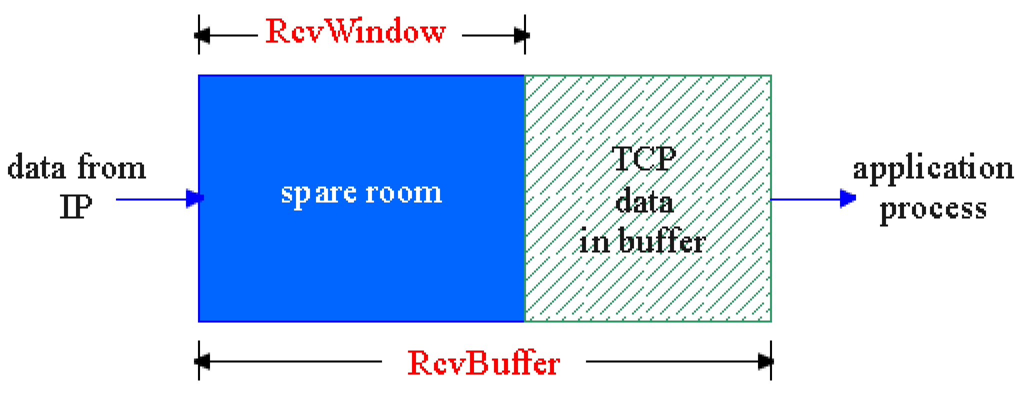


- Karn's RTT Estimator
  - If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

TCP Flow Control: Why and how?

Principle: sliding windows!

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`**

Avoids problems if fast computer sends data on, e.g., a mobile phone...!

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 then 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
  - If sliding window < reasonable segment size: too many small packets in flight (bigger header than contents, etc.)
  - Limit the # of smaller pkts than MSS (max segment size) to one per RTT

What if receiver window is size zero and receiver has nothing to send?

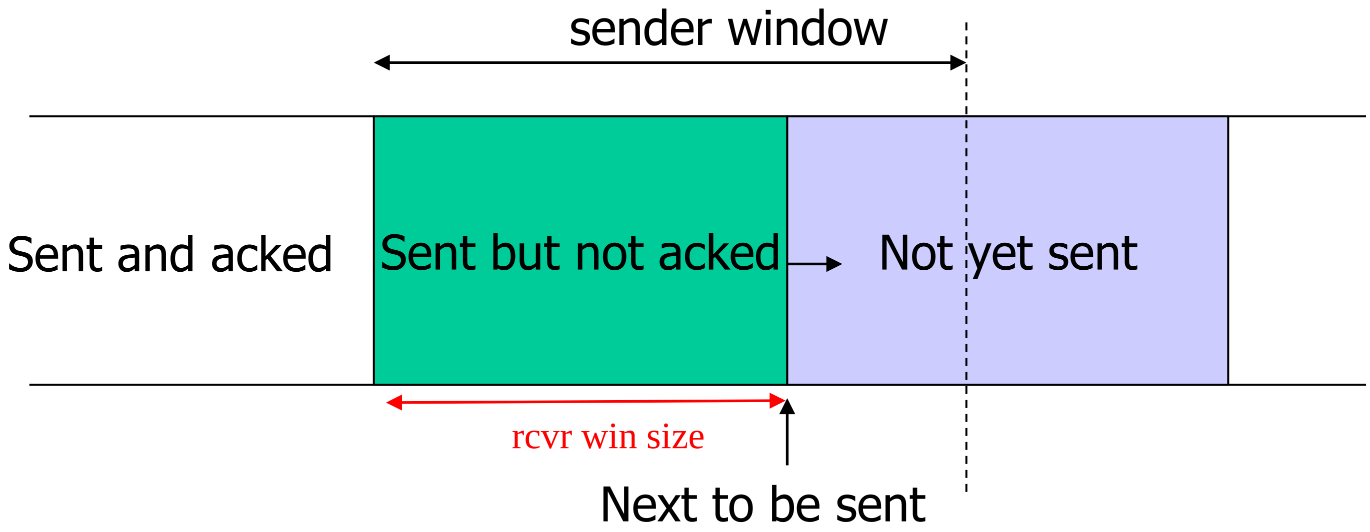
Sender will never learn when receiver has free capacity again! Sender probes with exponential backoff!

Question: When does TCP send a packet?

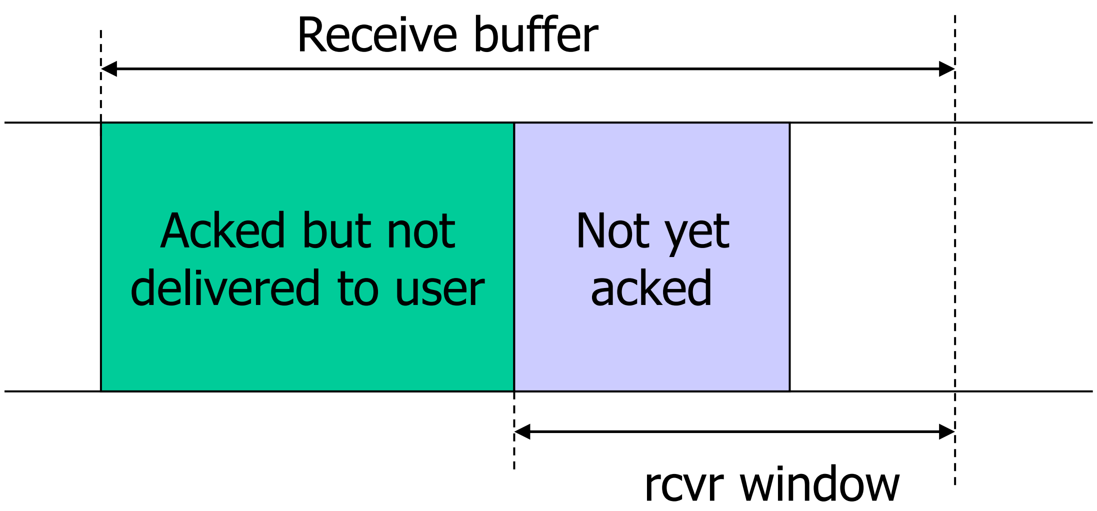
- Immediately when data is sent to TCP
- ... but need to flush explicitly for small amount of data!
- But in order to avoid too small windows: not next time (Nagle algorithm: keep # small packets per RTT small)
- ... wait until receiver has MSS available!
- If window = 0, exponential probing...

Window Flow Control:

Sender Side



Receiver Side

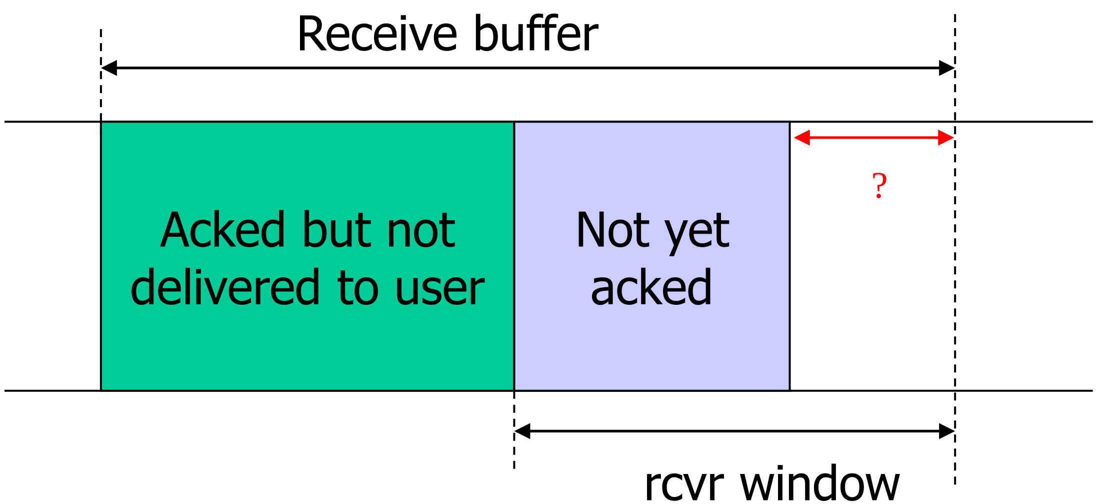


Window Flow Control:

Why not here? Non-ACKED not known? Small anyway?

May not be small! If out-of-order packets, there could be many! (Plus at most one delayed packet.) But what flow control is about is *delay to application*! This matters here. (Receiver window should be of size 2*RTT*bw to allow for retransmit.)

Receiver Side



Ideal Window Size?

- Need to store as many packets as are unACKed in flight... So?
- Ideal size = delay * bandwidth
  - Delay-bandwidth product (RTT * bottleneck bitrate)
- Window size < delay*bw ⇨ wasted bandwidth
- Window size > delay*bw ⇨
  - Queuing at intermediate routers (more than bottleneck rate arrives) ⇨ increased RTT
  - Eventually packet loss

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments (i.e., they set up state!)

- 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 (less than closing?)
- TCP Half-Close (four-way handshake)
- Connection aborts via RSTs (resets)

Example? No such TCP service running on this machine.
(Like corresponding ICMP message in UDP.)
But RST indicates absence of firewall?

TCP Connection Management (2)

Three way handshake:

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

- Specifies initial seq # (why?) Random for robustness!
- Specifies initial window #
- No application data

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

- ACKs received SYN (= 1 Byte)
- Allocates buffers
- Specifies server → receiver initial seq. #
- Specifies initial window #

Typically done after step 3 only: why?
SYN-flood attacks (see also SYN Cookies)

Step 3: Client system receives SYNACK

Data here?
Theoretically yes, but it's a system call...

Try with tcpdump or wireshark (e.g., telnet to **bsdi**):

SYN (ISN seq: 1617152000, with 0 bytes)
(also includes: window size, MSS, ...)

TCP ports
(e.g., discard service)

SYNACK
(SYN requires one byte)

```
1   0.0                  sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
2   0.002310 ( 0.0023)   bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                               ack 1617152001
3   0.003098 ( 0.0008)   sun.1090 > bsdi.7777: . ack 1
```

time

time delta

ACK

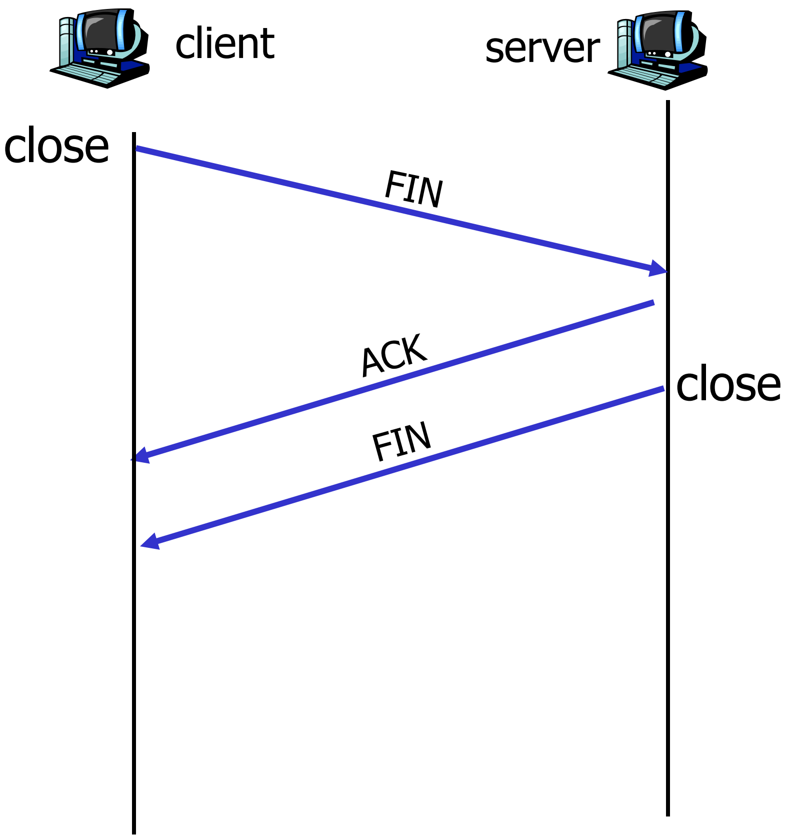
TCP Connection Management (3)

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.



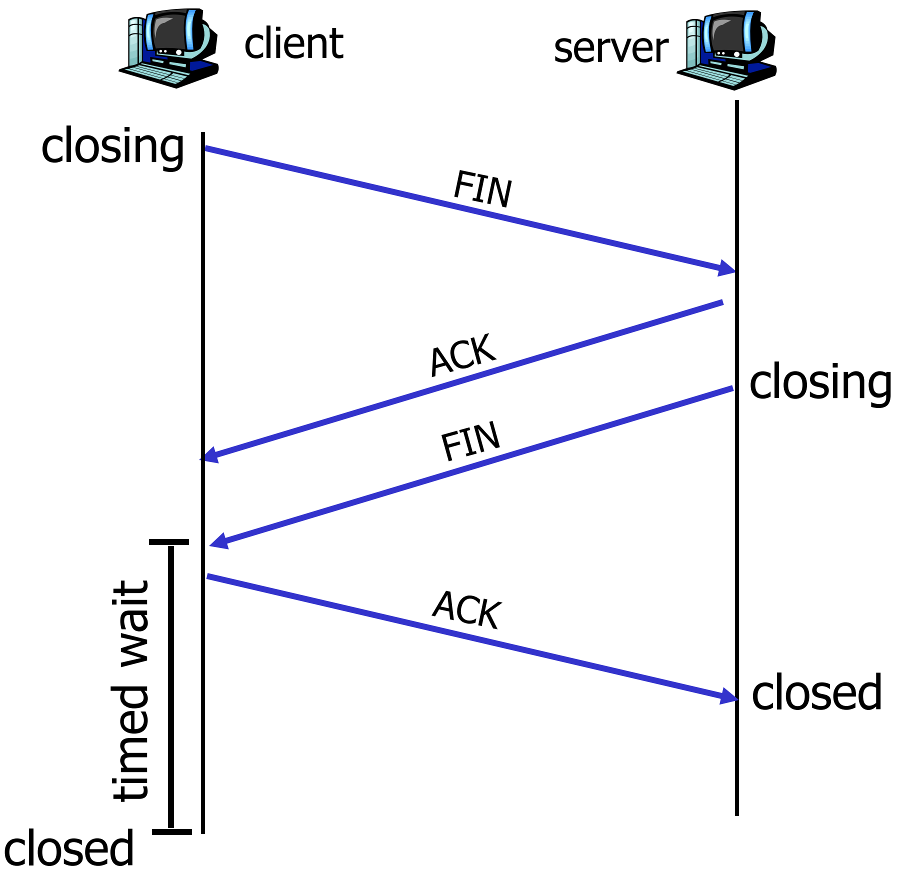
TCP Connection Management (4)

Step 3: Client receives FIN, replies with ACK.

- Enters "timed wait" (why? Byzantine generals?) – will respond with ACK to received FINs (can it be closed on full agreement in lossy environment?)

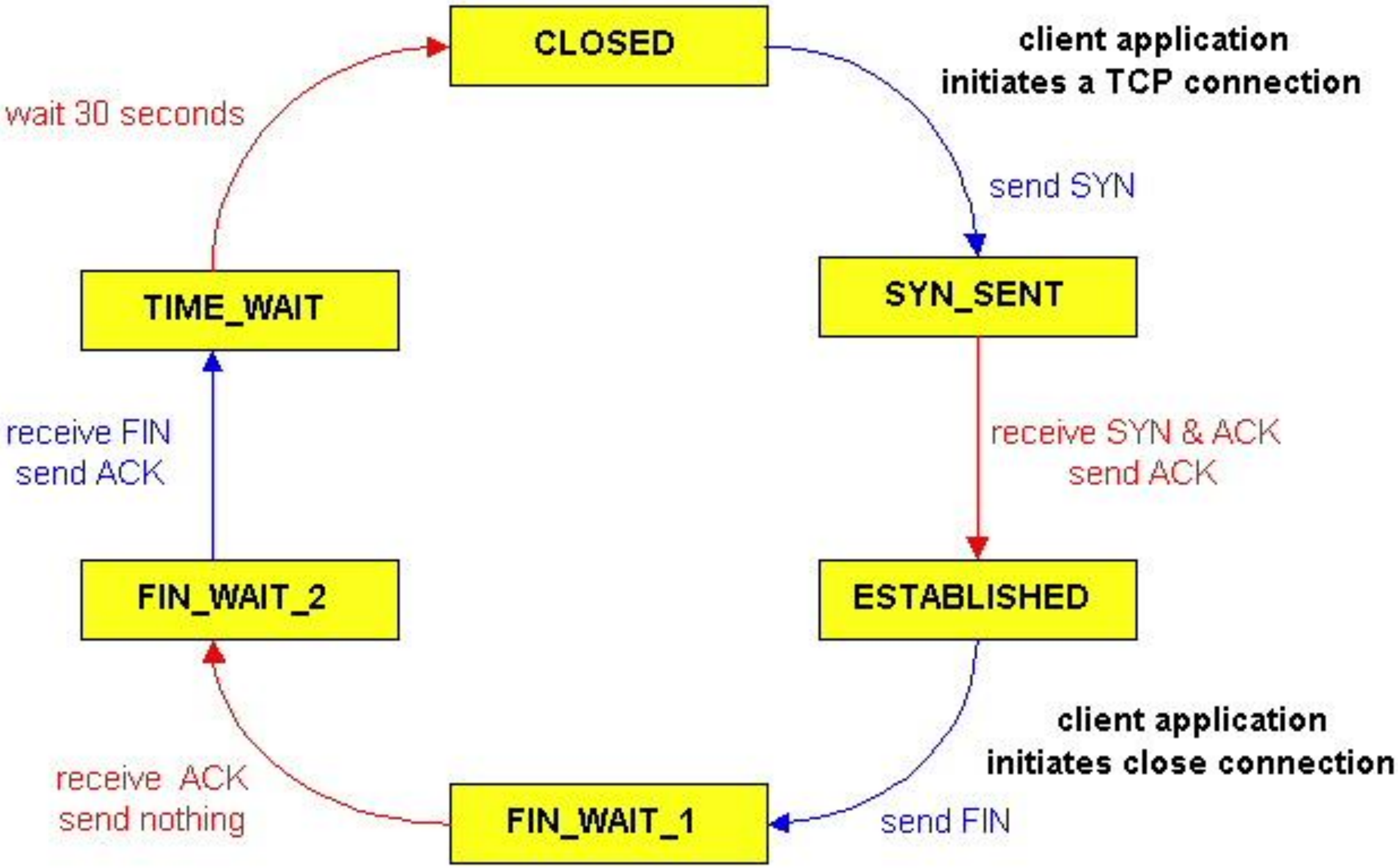
Step 4: Server, receives ACK. Connection closed.

Note: With small modification, can handle simultaneous FINs.



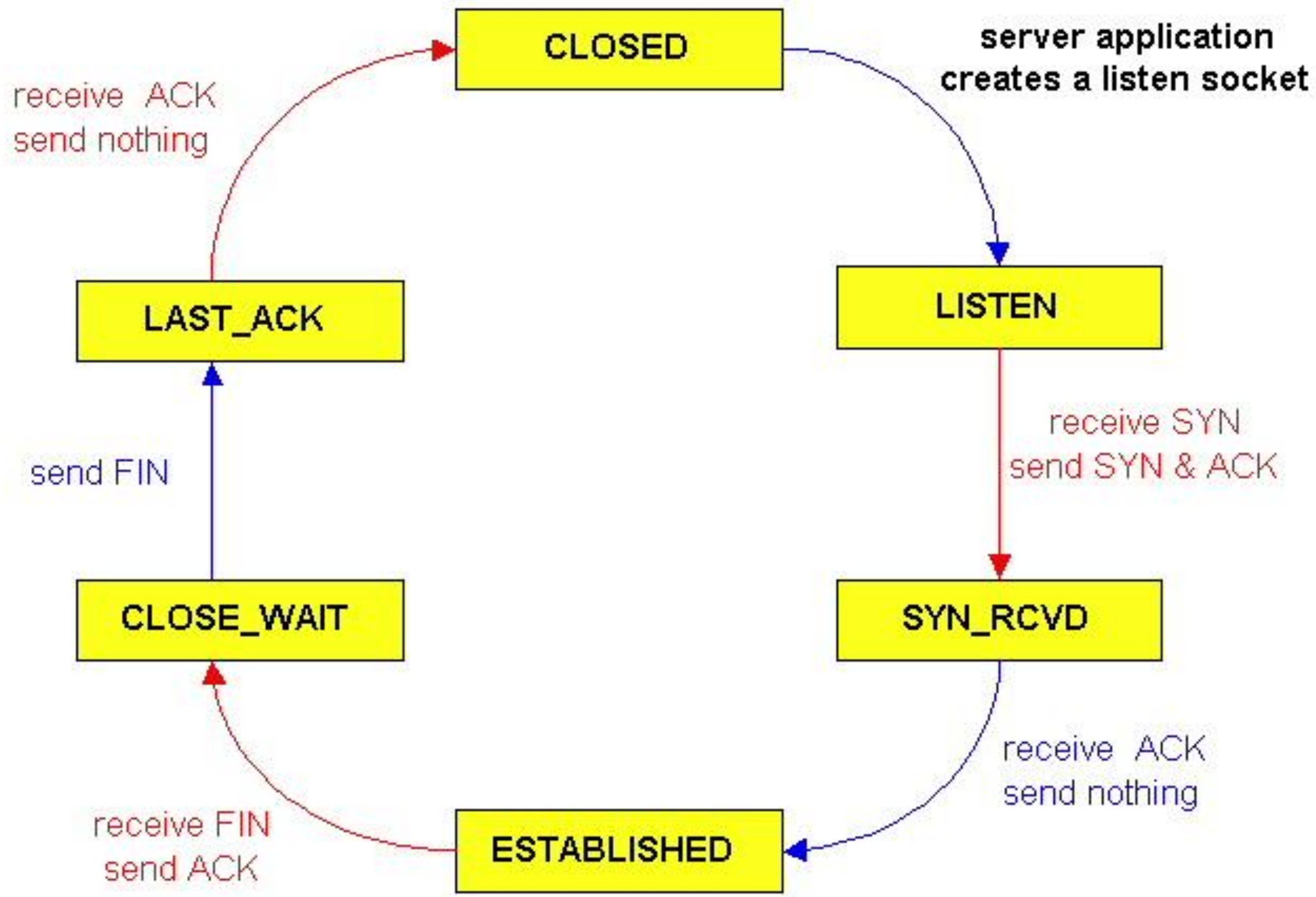
TCP Connection Management (5)

TCP **client** lifecycle

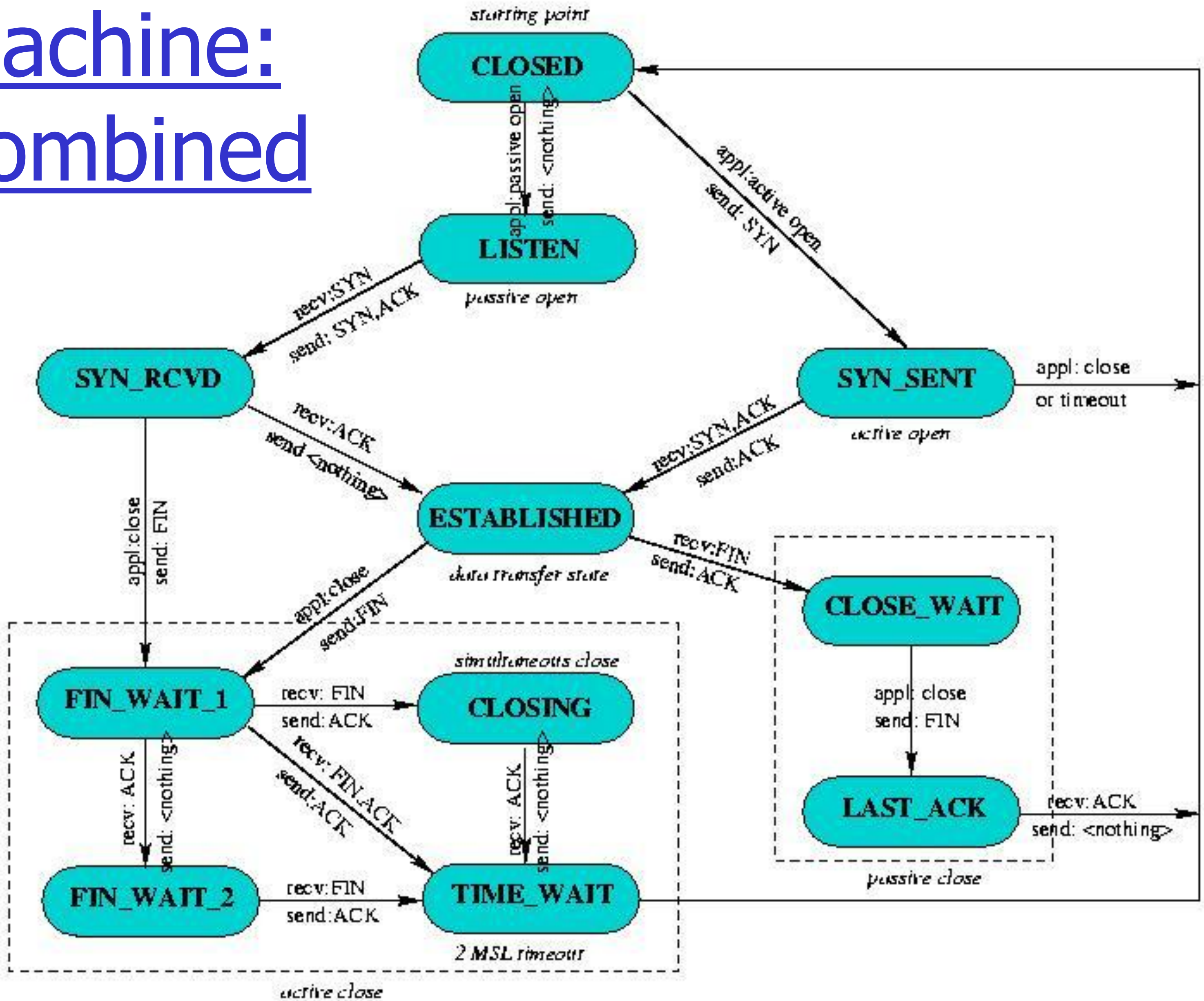


TCP Connection Management (6)

TCP **server** lifecycle



TCP state machine: Combined



Excursion: Congestion Control Principles

Why congestion control?

TCP Acknowledgement Clocking

- Already seen: TCP is "self-clocking"/"ACK-clocking" (ACKs define pace): data only ACKed when received, and new data only sent when ACKed, …
- Ensures an "equilibrium" (rate of ACK = rate of data)
- But how to get started and control congestion?
  - Slow Start
  - Congestion Avoidance
- Other TCP features
  - Fast Retransmission
  - Fast Recovery
- How to achieve?
  - Again: sliding window principle!
  - Congestion window (cnwd) similar to flow control window (rcvr win), limits amount of unACKed packets! (If cnwd full of unACKed: wait/backoff!)

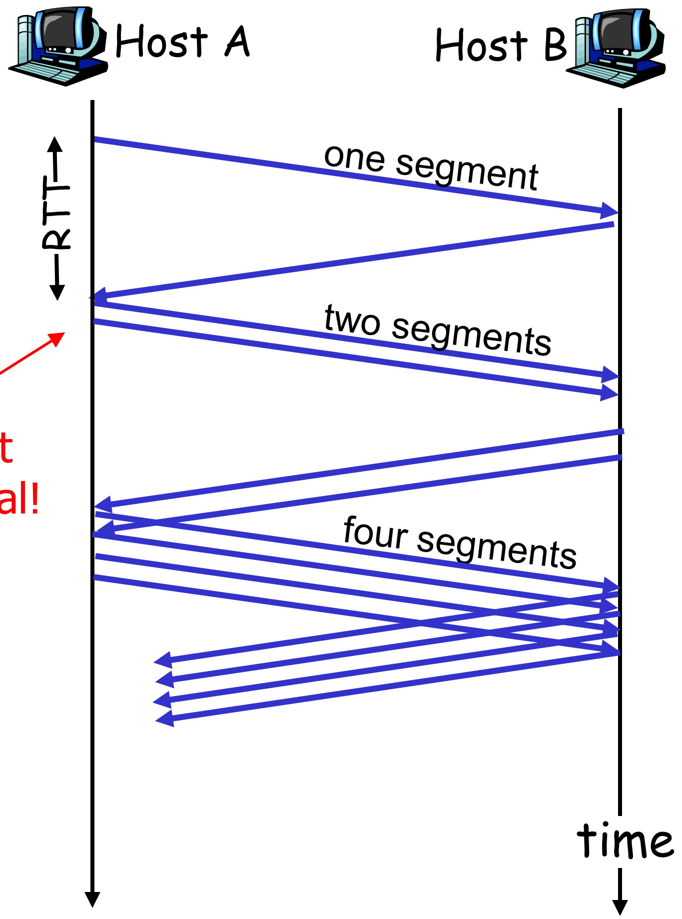
TCP Congestion Control: cnwd

- Principle: "Probing" for usable bandwidth?
  - Ideally: Transmit as fast as possible (**`cnwd`** as large as possible) without loss
  - *Increase* **`cnwd`** until loss (congestion)
  - Loss (timeout, dup ACK): *Decrease* **`cnwd`**, then begin probing (increasing) again
- Goals?
  - Use resources efficiently
  - Do not overload
  - Be „collaborative"
  - ...?
- Two "phases"
  - Slow start
  - Congestion avoidance
- Important variables:
  - **`cnwd`**
  - **`threshold (ssthresh)`**: Defines threshold between slow start phase and congestion avoidance phase

TCP Slowstart

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event?
  - Timeout or or three duplicate ACKs
  - No NAKs...

Note: each segment ACKed = exponential!



Slowstart algorithm

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

Recall: parallel to this sender we are also bounded by receiver window size!

Congestion Avoidance

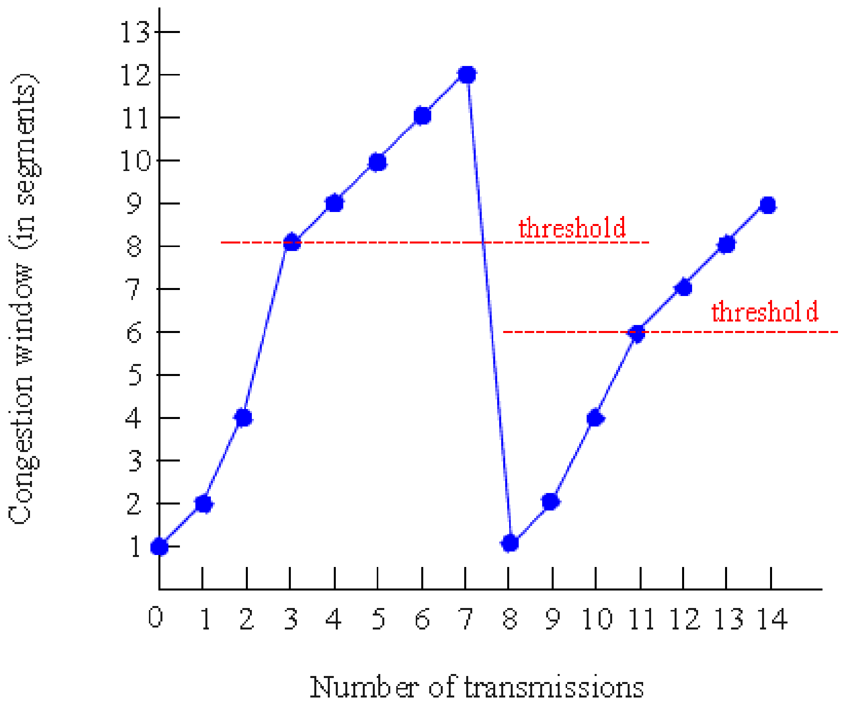
- Assumption: loss implies congestion – why? good?
  - Unfortunately, no explicit infos from routers normally... (sometimes in LAN possible)
  - Not necessarily true on all link types (e.g.?)
  - E.g., not true for wireless networks!
- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set threshold to 0.5W (multiplicative decrease)
- Upon receiving new ACK
  - Increase cwnd by 1/cwnd MSS (not "+1 MSS": cong. avoidance)
  - Results in additive increase! (why? one more for full window only!)

Recall: window size should not go below 1 MSS...
What is worse: a timeout or a duplicate ACK?
Timeout! Duplicate ACK: network still okay?

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]



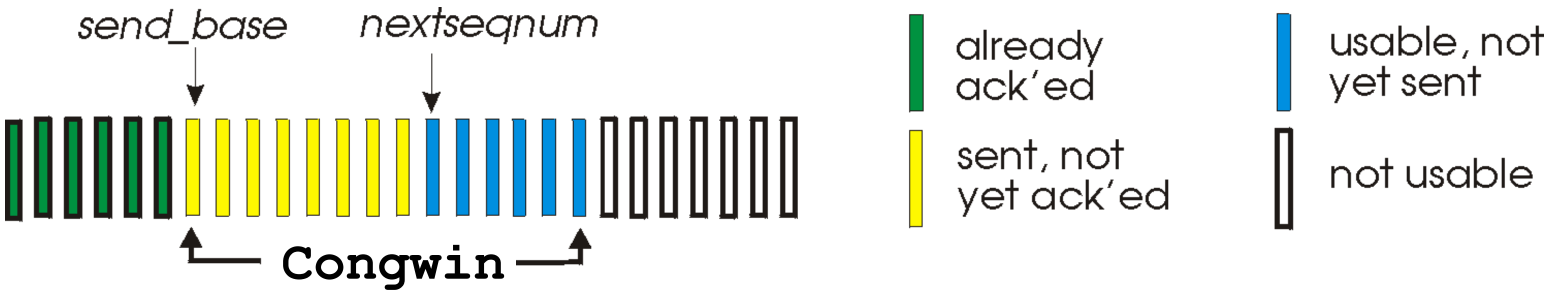
1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs [today most popular TCP]

Return to Slow Start

- If packet is lost we lose self clocking
  - Lost packet/ACK => cannot clock TCP window precisely (ACK rate does not equal data rate)
  - Need to implement slow-start and congestion avoidance together
- When **timeout** occurs
  - Set threshold to 0.5 W (current window size)
  - Set cwnd to one segment
- When **three duplicate acks** occur:
  - Set threshold to 0.5 W
  - Retransmit missing segment == Fast Retransmit (= retransmit before timer expires for it!)
  - cwnd = threshold + number of **dupacks** (not to one!)
  - Upon receiving **new acks** cwnd = threshold (cut in half, necessary so cwnd = W again: after many dupacks, an ACK frees up much space in the cwnd, the corresponding sending burst should be avoided!)
  - Use congestion avoidance == Fast Recovery (= no slow start, = TCP Reno from exercises but not TCP Tahoe)

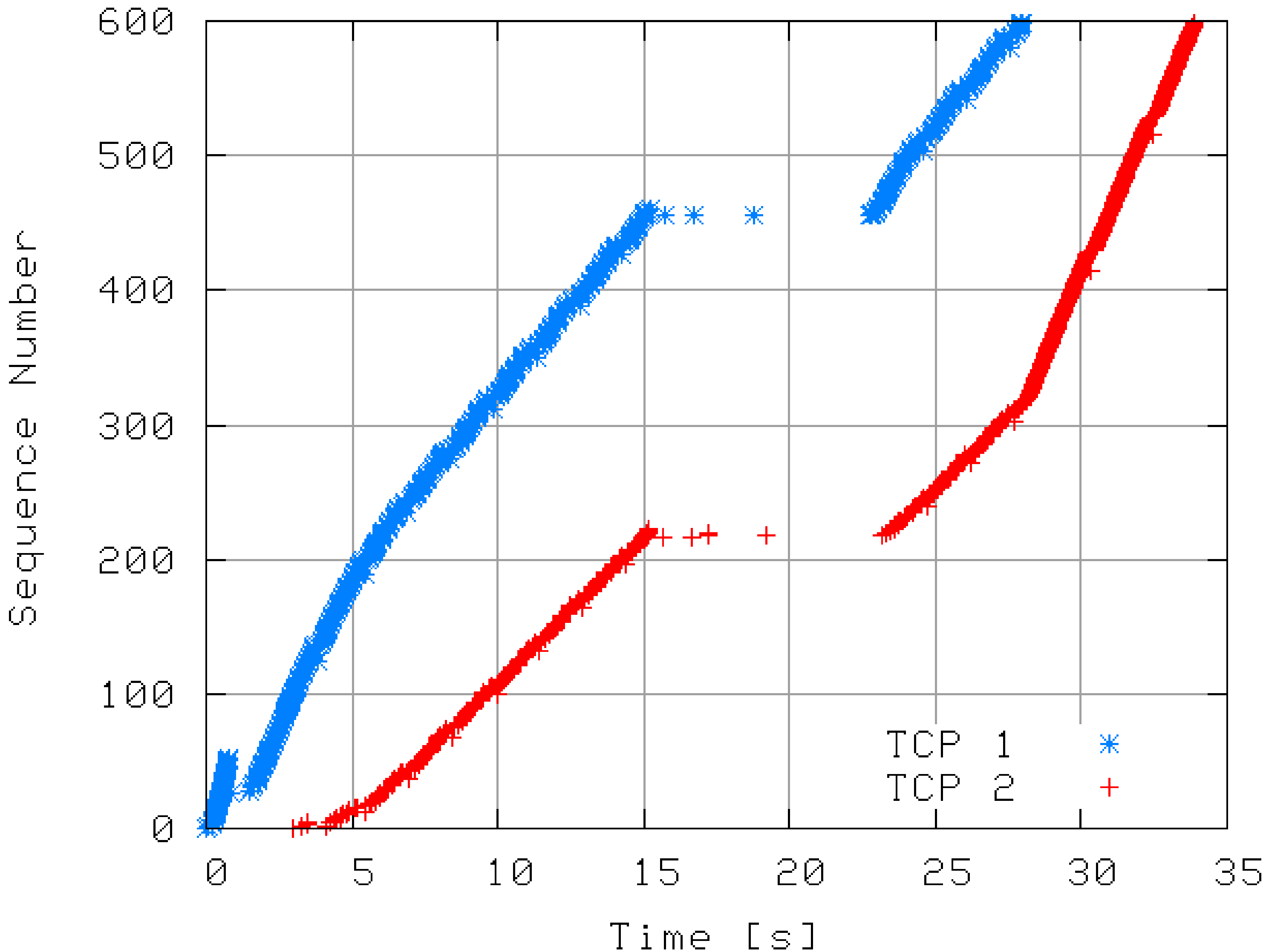
TCP Congestion Control: Summary

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

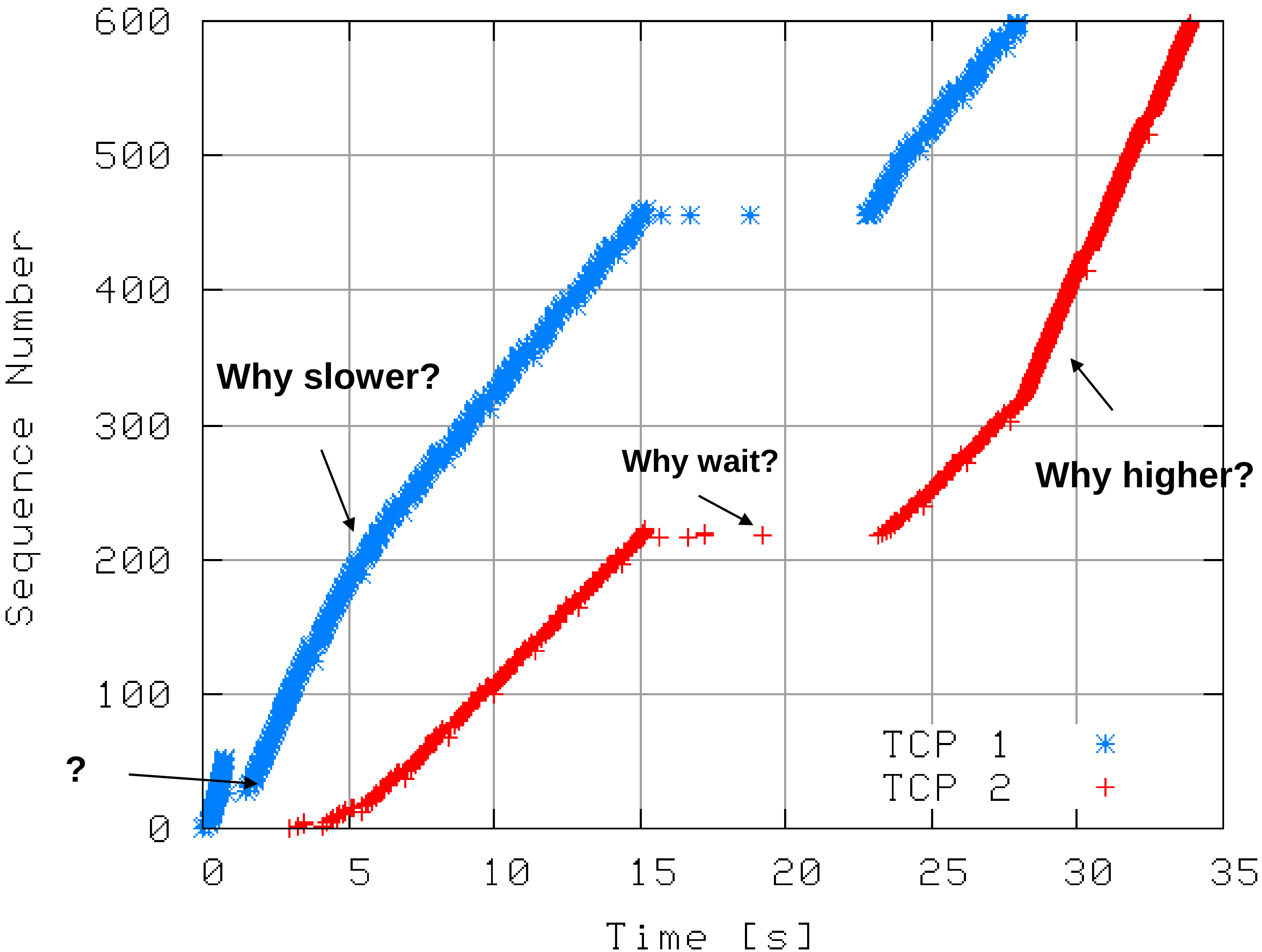


- W segments, each with MSS bytes sent in one RTT

Example 1: What is going on?

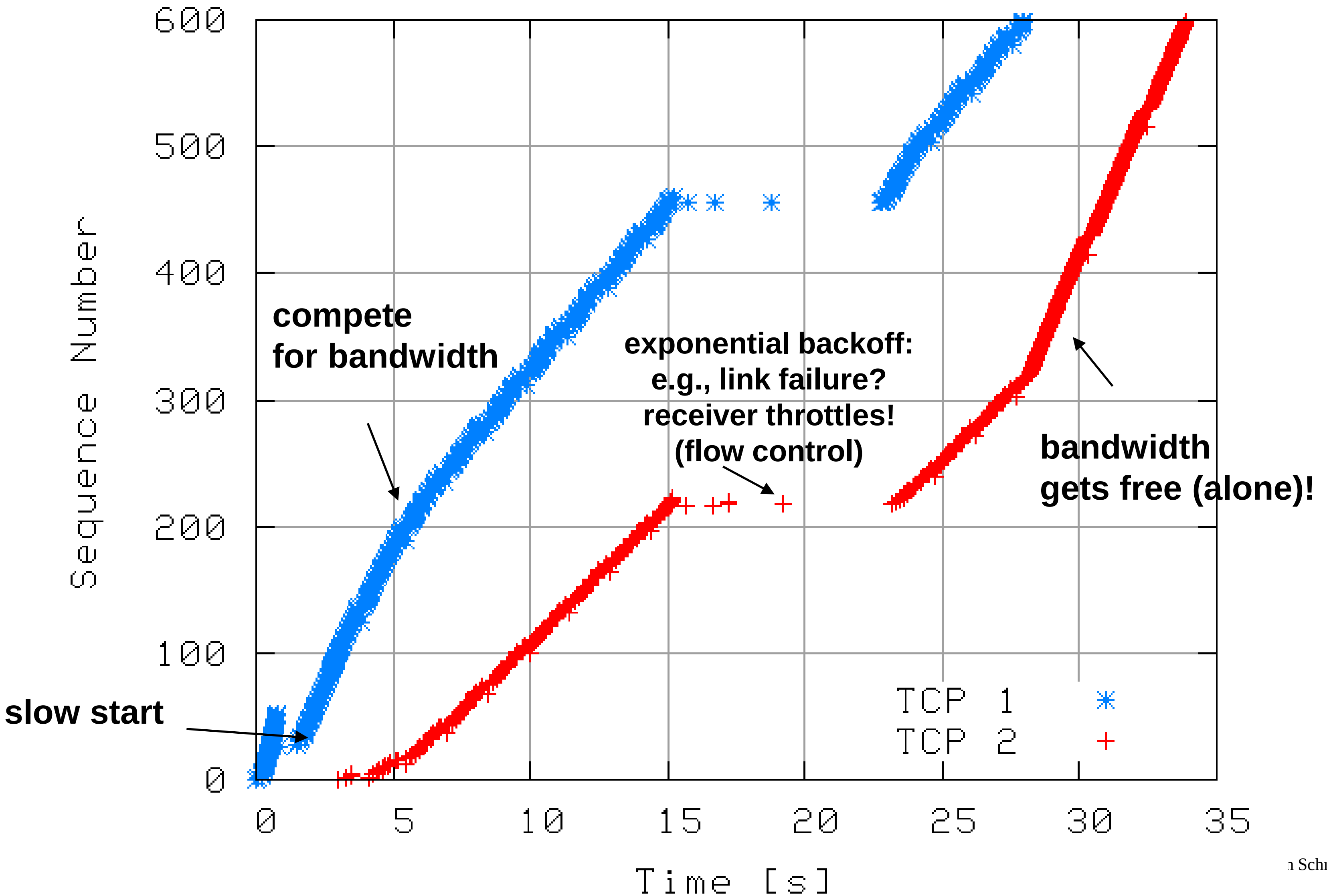


Example 1: What is going on?

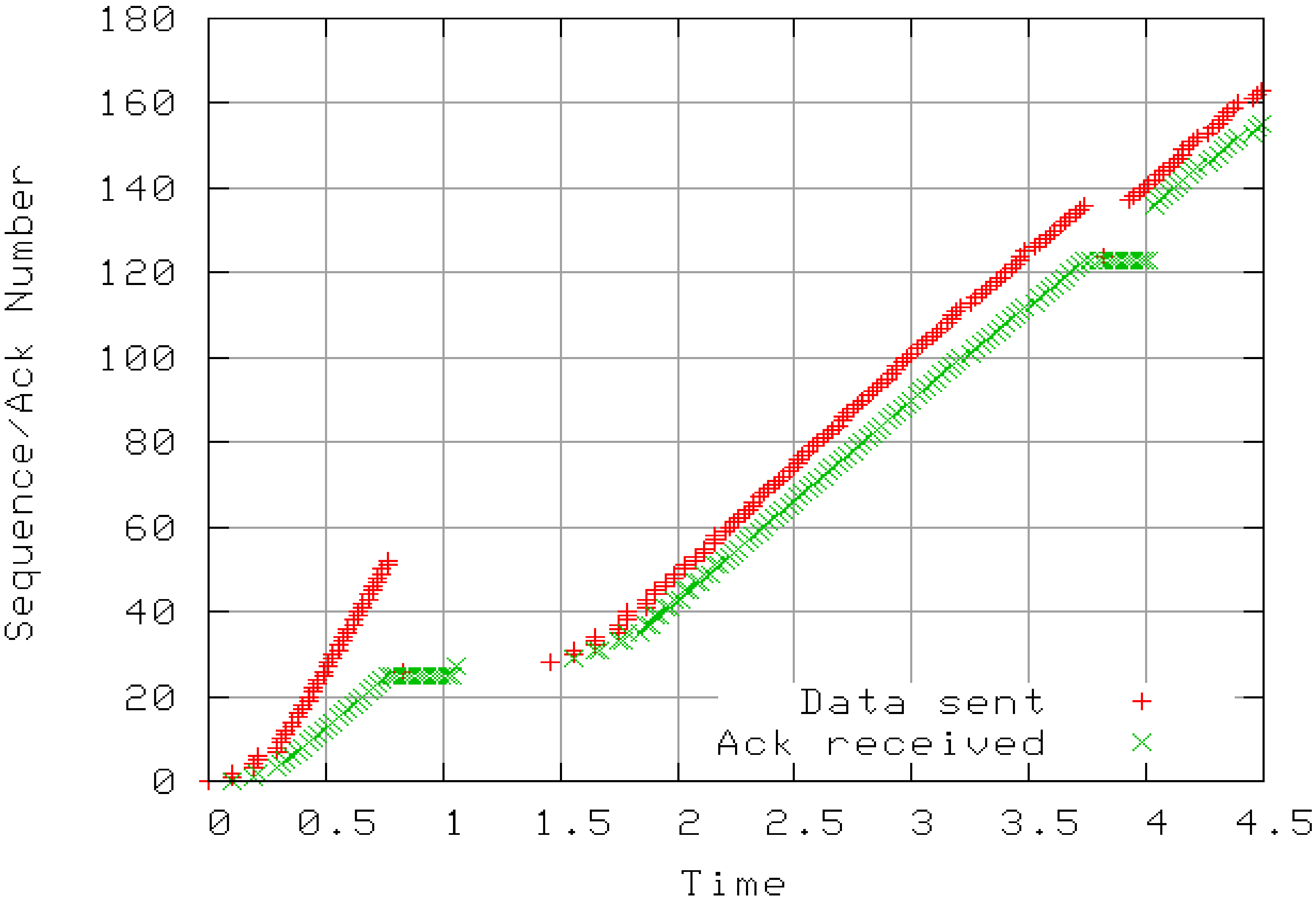


Example 1:

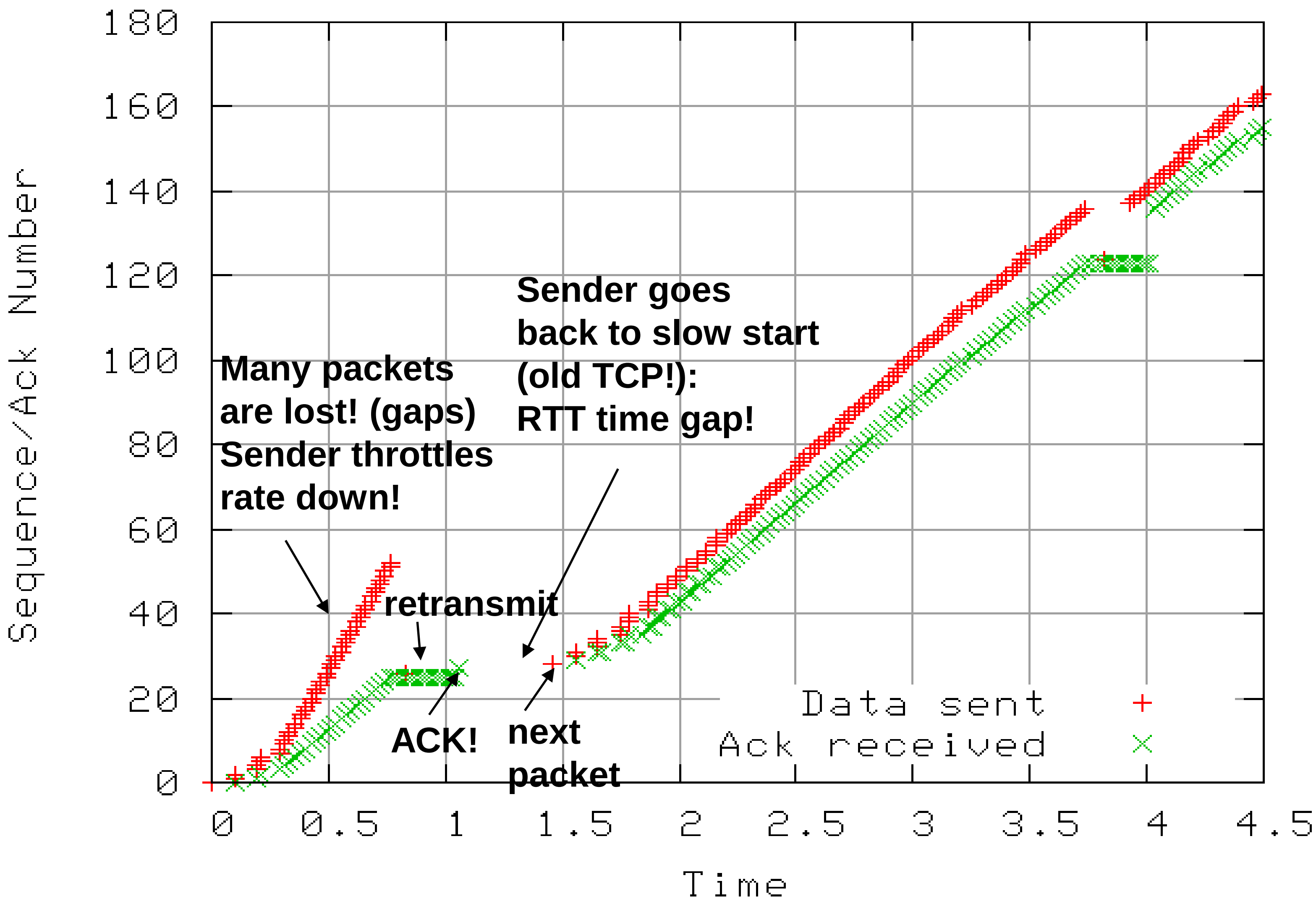
Recall: when receiver window down to zero, to avoid deadlock (no new data, no opportunity to reply for receiver...), sender probes!



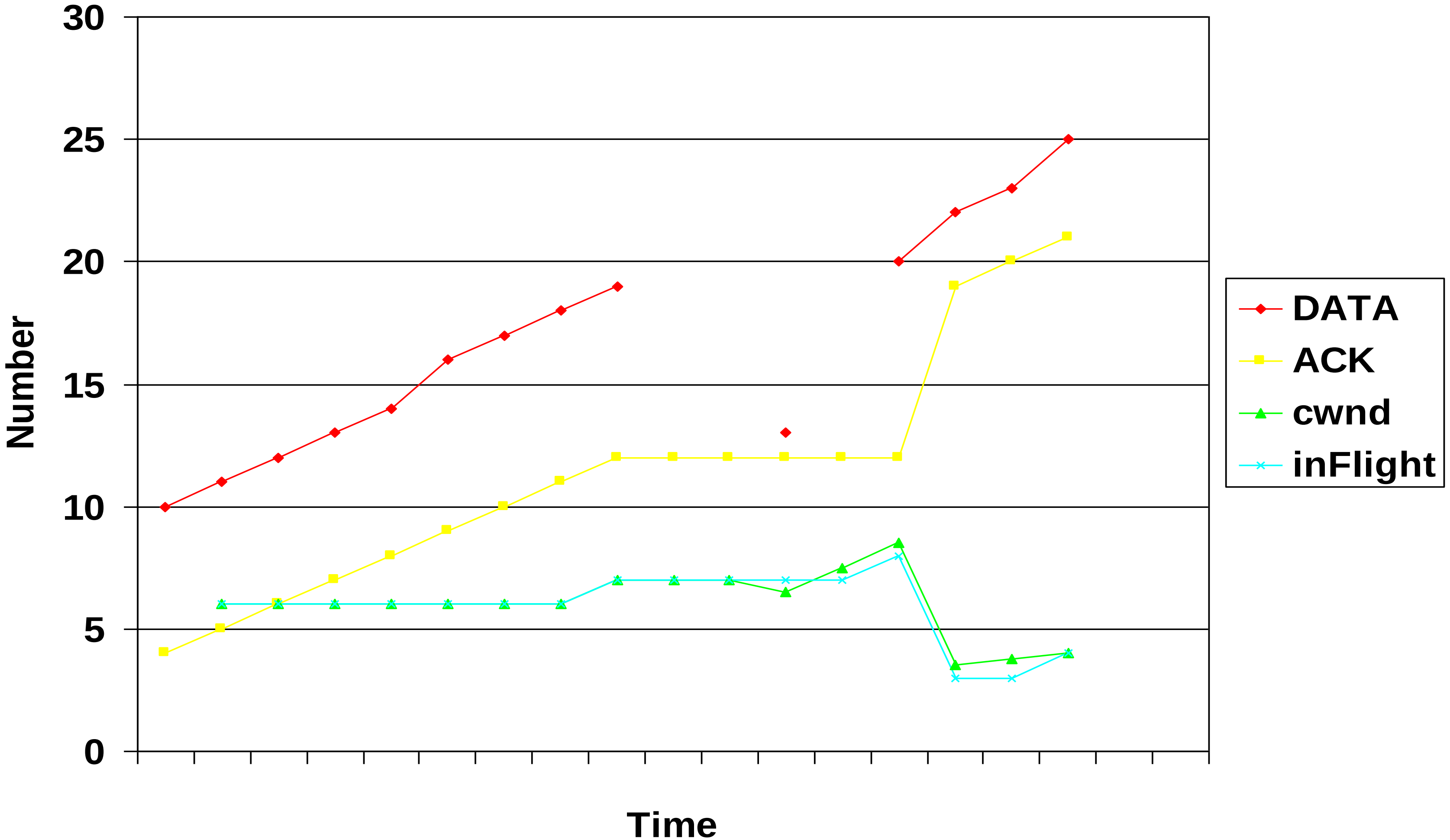
Example 2: What's going on?



Example 2: What's going on?

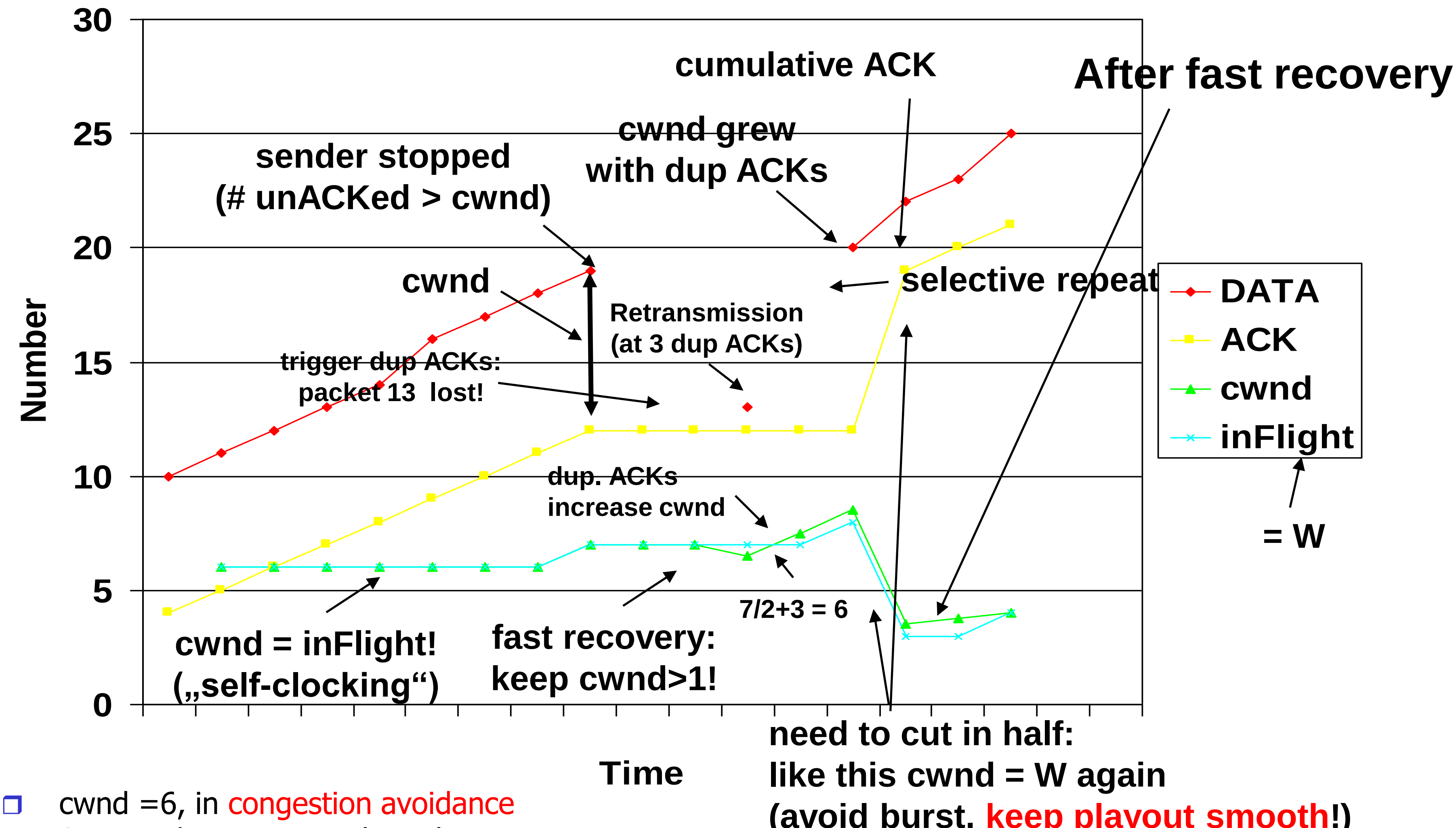


Fast Recovery Example: What happens?



- How many packets got lost? Which ones?
- Fast recovery or not? Selective repeat or repeat all? ...

Fast Recovery Example: What happens?



- cwnd =6, in congestion avoidance
- Seq numbers increase linearly over time
- Triple ACK: packets still received but one missing / out of order => fast recovery

TCP Flavors / Variants

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

Extensions

- Avoiding timeouts and unnecessarily retransmissions?
- 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

- Wireless TCP, TCP for datacenters, ...

- 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