

UiO : Faculty of Mathematics and Natural Sciences
University of Oslo

Department of Informatics
Networks and Distributed Systems (ND) group

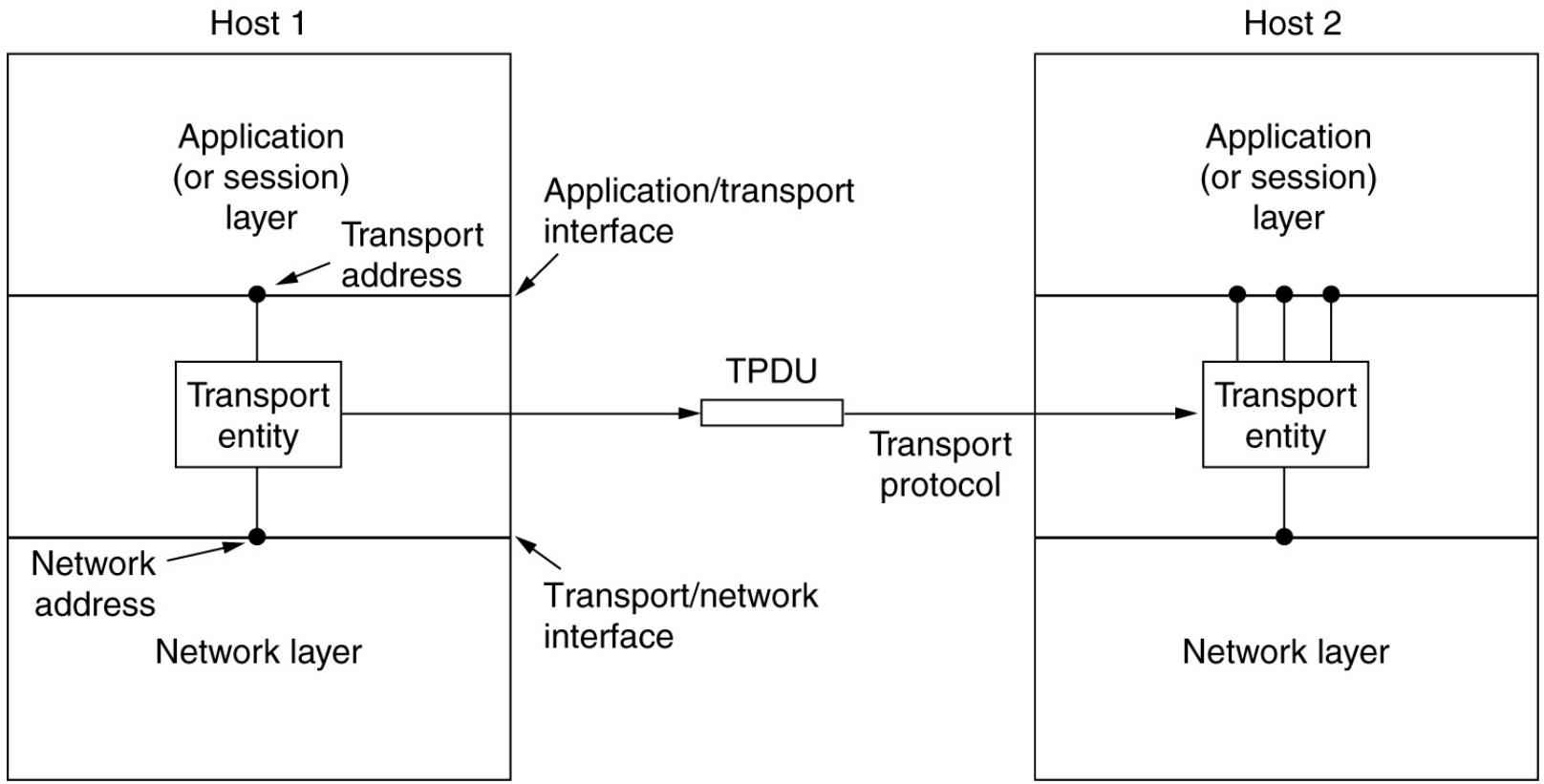
IN3230 / IN4230

The Internet transport layer

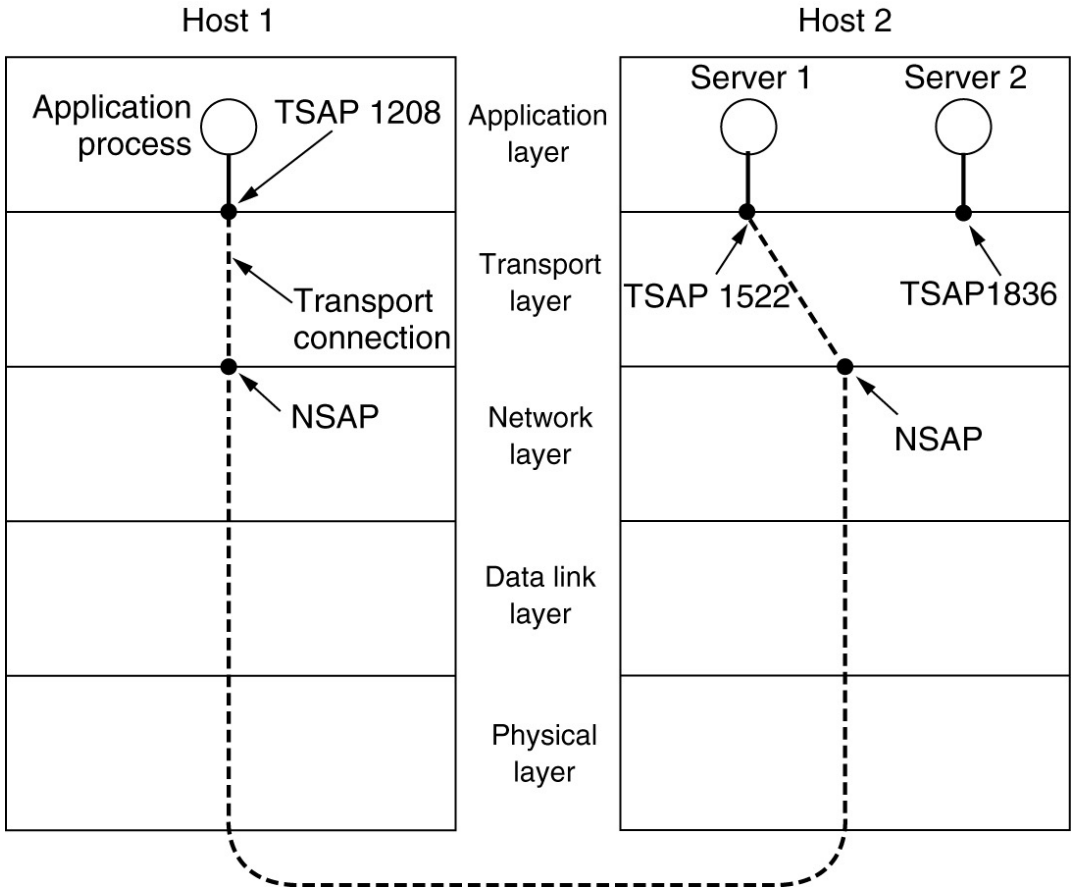


Michael Welzl

Where we are in the stack...

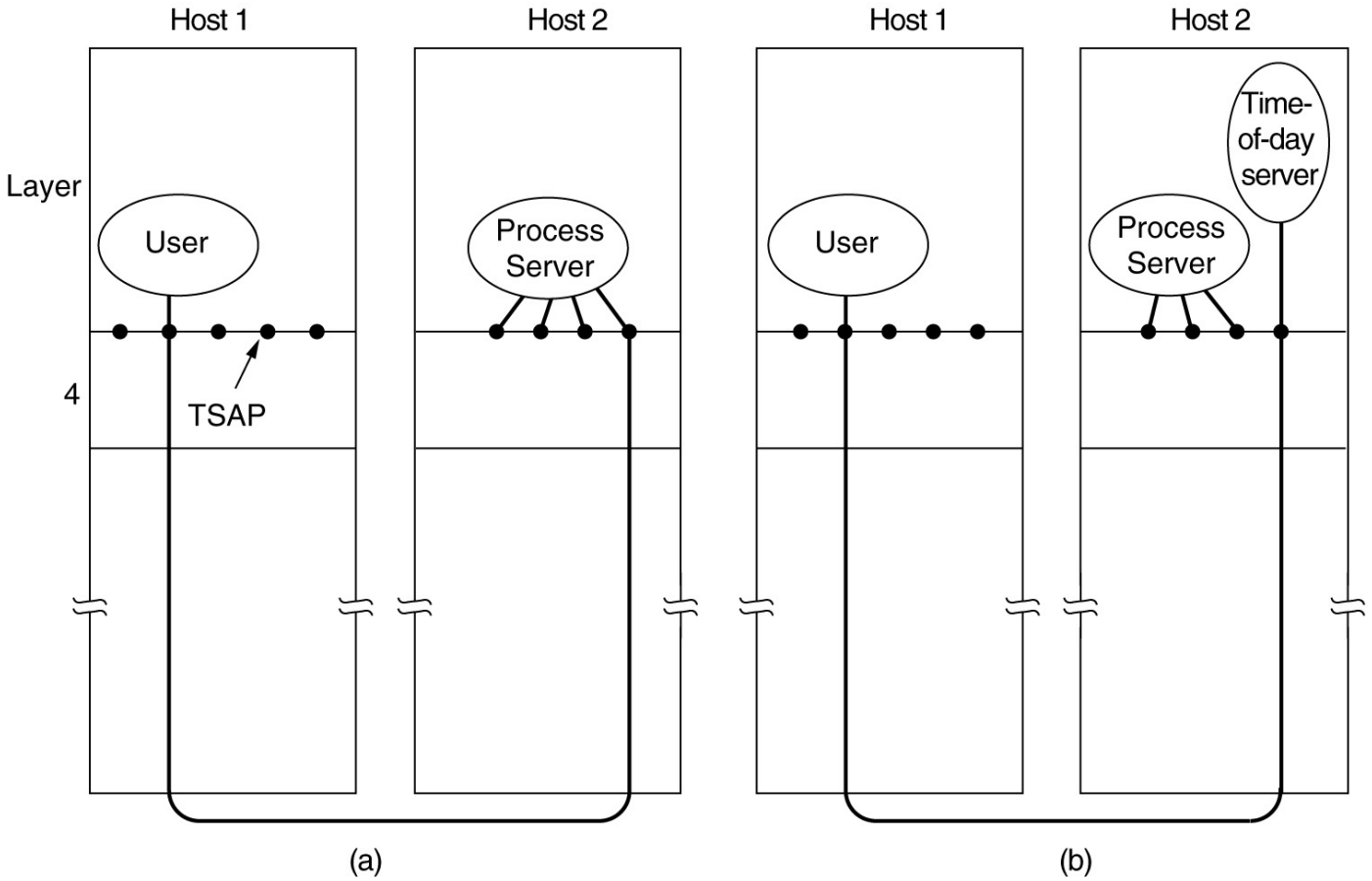


Addressing [Tanenbaum book]



TSAPs, NSAPs and transport connections

Conn. establishment [Tanenbaum book]



How a user process in host 1 could establish a connection with a time-of-day server in host 2

The Internet transport layer

- Services are (mostly) defined by two protocols
  - UDP (connectionless): sends a "datagram"
  - TCP (connection oriented): transfers a reliable bytestream (of "segments")

- Addressing: port numbers
  - Choosing a service during connection establishment: well-known ports

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Berkeley sockets:
TCP service primitives

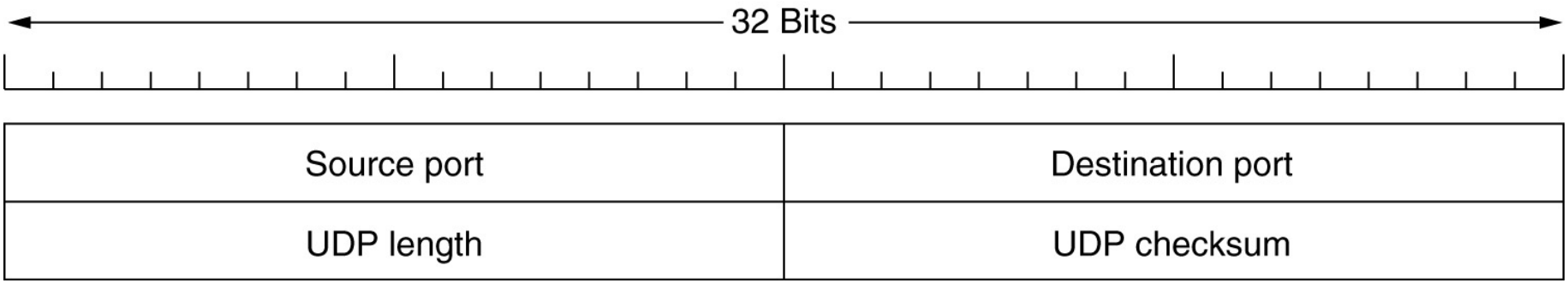
Evolution of the Internet's transport layer

- Only TCP and UDP... so long that it has become almost impossible to use something else. "**Ossification**"
  - E.g., in-network devices make X/IP fail unless X != {TCP, UDP}
- Try-a-different-protocol-else-fall-back hard to implement; thus, protocols now often developed in user space, over UDP, per app.
  - Wheel re-invention: e.g. multi-streaming is in SCTP, Adobe's RTMFP, QUIC
  - Other important QUIC features: *everything* is encrypted; faster handshake
  - Think "TCP++" for these protocols. Good to understand TCP first!

IFI research!

- IETF Transport Services (TAPS):
  new API that lets applications choose service instead of protocol
  - Flexible protocol choice, try-else-fall-back complexity: all below the API
  - Apple's Network.Framework (since beta for iOS 12 and macOS Mojave, presented at WWDC 2018) conforms to TAPS; under discussion for C++ library

UDP



- UDP = IP + 2 features:
  - Ports: identify communicating instances with similar IP address (transport layer)
  - Checksum: Adler-32 covering the whole packet
    - checksum field = 0: no checksum at all

- Usage of UDP: unreliable data transmission (DNS, SNMP, real-time streams, ..)

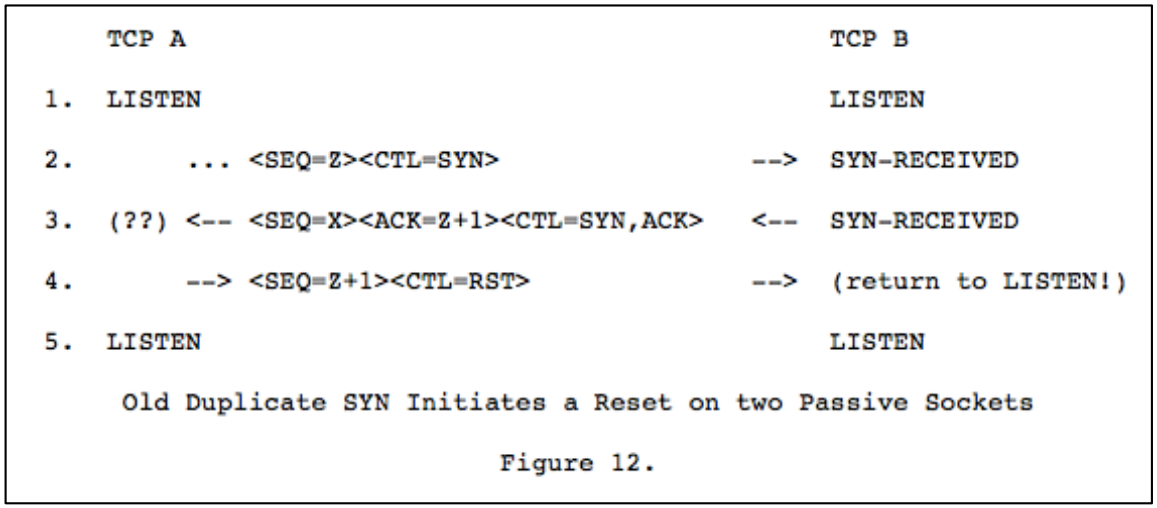
TCP

How it <u>really</u> is today.

Skipping the very basic things that you should already know from before.

The beginning (1970s, up to 1981)

- RFC 793, 1981, Jon Postel
  - Front page says: "DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION" and "prepared for Defense Advanced Research Projects Agency"
  - Goal was obviously to make it super reliable:
    85 pages, covering
    many corner cases



	TCP A		TCP B
1.	LISTEN		LISTEN
2.	... <SEQ=Z><CTL=SYN>	-->	SYN-RECEIVED
3.	(??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>	<--	SYN-RECEIVED
4.	--> <SEQ=Z+1><CTL=RST>	-->	(return to LISTEN!)
5.	LISTEN		LISTEN

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 12.

- Robustness is never a bad thing, so complexity has remained
  - Some things slightly obscure: e.g., half-closed connections: after saying "FIN", host can still receive, with no time limit, until other host says "FIN".

Some old things are best forgotten

- Push bit (PSH)
- Urgent pointer (URG)

- Generally, maybe don't read RFC 793…
  - **RFC 9293**, published in 2022:
    "This document obsoletes RFC 793, as well as 879, 2873, 6093, 6429, 6528, and 6691 that updated parts of RFC 793. It updates RFC 1122 (..) RFC 5961 (..)"

- Also consider: TCP spec roadmap (RFC 7414)
  - And implementations diverge...

Later 80‘s, and 90‘s

- FreeBSD was "reference" implementation
  - Matched Stevens book [K.Fall, R. Stevens: TCP/IP Illustrated] patches over time were if-clauses in the code; later revamped
  - Originally, much code written by people who also wrote the RFCs (Van Jacobson, Sally Floyd, Mark Allman, Matt Mathis, ..)
  - Many of them also wrote ns-2 simulator code
  - These people guided the design in the IETF
- Van Jacobson "saved the Internet" after congestion collapse, with code + SIGCOMM 1988 paper about it: "Congestion Avoidance and Control"
- Linux implementation also common and well known in IETF, but completely different code (segment-, not byte-based)
  - Focus of Google (more later)

Resulting IETF view

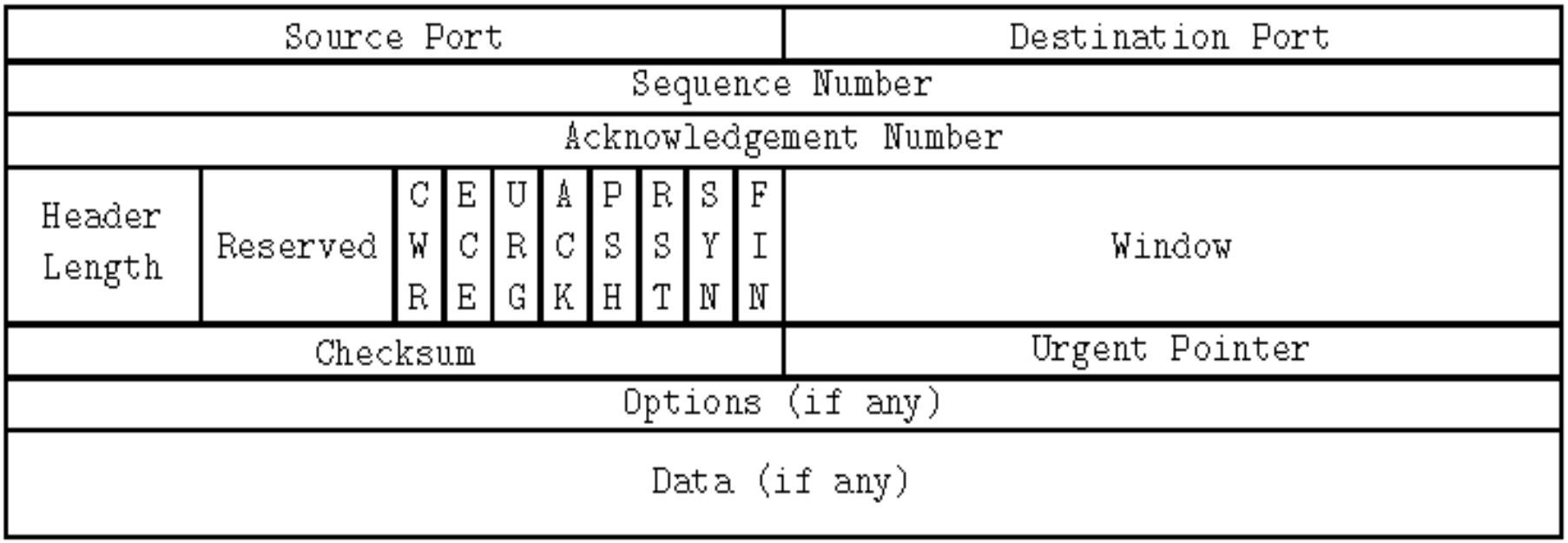
- TCP has been working for a long time, so let's be careful
  ➔ making it more robust is okay.
  - Note: WG is called "TCP Maintenance and Minor Extensions (TCPM)"
  - ...and let's be careful about congestion control in particular.

- Our "early heroes" did a great job, so never break their rules
  ➔ important congestion control principles:
  1. ACK clocking ("conservation of packets" principle)
  2. Timeout means that the network is empty

Keep them in mind

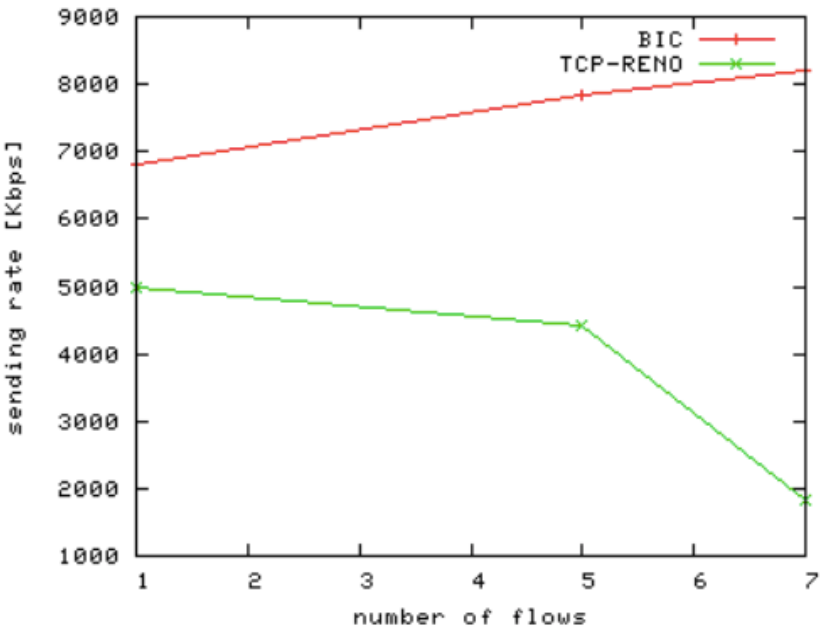
- But, note: the IETF also tries to stay meaningful
  - Voluntary, we have no "Internet police"
  - We should be happy that companies such as Google (in case of TCP) keep coming to the IETF to tell us what they do

TCP Header

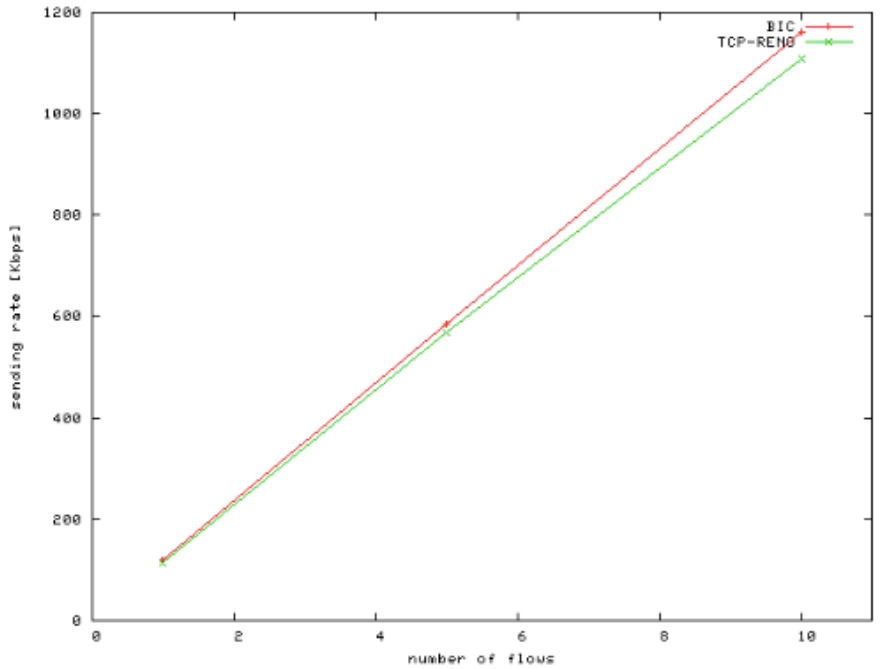


- Flags indicate connection setup/teardown, ACK, ..
- If no data: packet is just an ACK
- Window = advertised window from receiver (flow control)
  - Field size limits sending rate in today's high speed environments; solution: Window Scaling Option – both sides agree to left-shift the window value by N bit

The importance of Window Scaling



Local testbed



PlanetLab
(Austria => Brazil)

- 2007 measurements with Linux
  - TCP BIC vs. TCP-Reno competition
- OSes gradually increase factor
  - SIGCOMM 2017 QUIC paper: 4.6% rwnd-limited TCP connections

TCP error control: Acknowledgements

- ACK ("positive" Acknowledgement) serves multiple purposes:
  - sender: throw away copy of data held for retransmit
  - time-out cancelled
  - msg-number can be re-used
- TCP ACKs are cumulative
  - ACK *n* acknowledges everything up to *n-1*
- ACKs should be delayed *(except when sending duplicates – why? later!)*
  - TCP ACKs are unreliable: dropping one does not cause much harm
  - Enough with 1 ACK every 2 segments, or at least 1 every 500 ms (often: 200 ms)
- TCP counts bytes; ACK carries "next expected byte" (#+1)
  - Sender sends them as "segments", ideally of size SMSS
  - Nagle algorithm delays sending to collect bytes & avoid sending tiny segments (can be disabled)

Following slides: segment numbers for simplicity (imagine 1-byte segments)

Error control: Timeout

Remember: that's the first congestion signal. Back to square 1 (SS from cwnd=1).

- Go-Back-N behavior in response to timeout

- Retransmit Timeout (RTO) timer value difficult to determine:
  - too long => bad in case of msg-loss; too short => risk of false alarms
  - General consensus: too short is worse than too long; use conservative estimate

- Calculation: measure RTT (Seg# ... ACK#) , then:
  original suggestion in RFC 793: Exponentially Weighed Moving Average (EWMA)
  - SRTT = (1-α) SRTT + α RTT
  - RTO = min(UBOUND, max(LBOUND, β * SRTT))

- Depending on variation, result may be too small or too large; thus, final algorithm includes variation (approximated via mean deviation)
  - SRTT = (1-α) SRTT + α RTT
  - δ = (1 - β) * δ + β * [SRTT - RTT]
  - RTO = SRTT + 4 * δ

That's not how Linux does it

RTO calculation

- Problem: retransmission ambiguity
  - Segment #1 sent, no ACK received → segment #1 retransmitted
  - Incoming ACK #2: cannot tell whether original or retransmitted segment #1 was ACKed
  - Thus, cannot reliably calculate RTO!

- Solution 1 [Karn/Partridge]: ignore RTT values from retransmits
  - Problem: RTT calculation especially important when loss occurs; sampling theorem suggests that RTT samples should be taken more often

- Solution 2: Timestamps option
  - Sender writes current time into packet header (option)
  - Receiver reflects value
  - At sender, when ACK arrives, RTT = (current time) - (value carried in option)
  - Problems: additional header space; historical: facilitates NAT detection

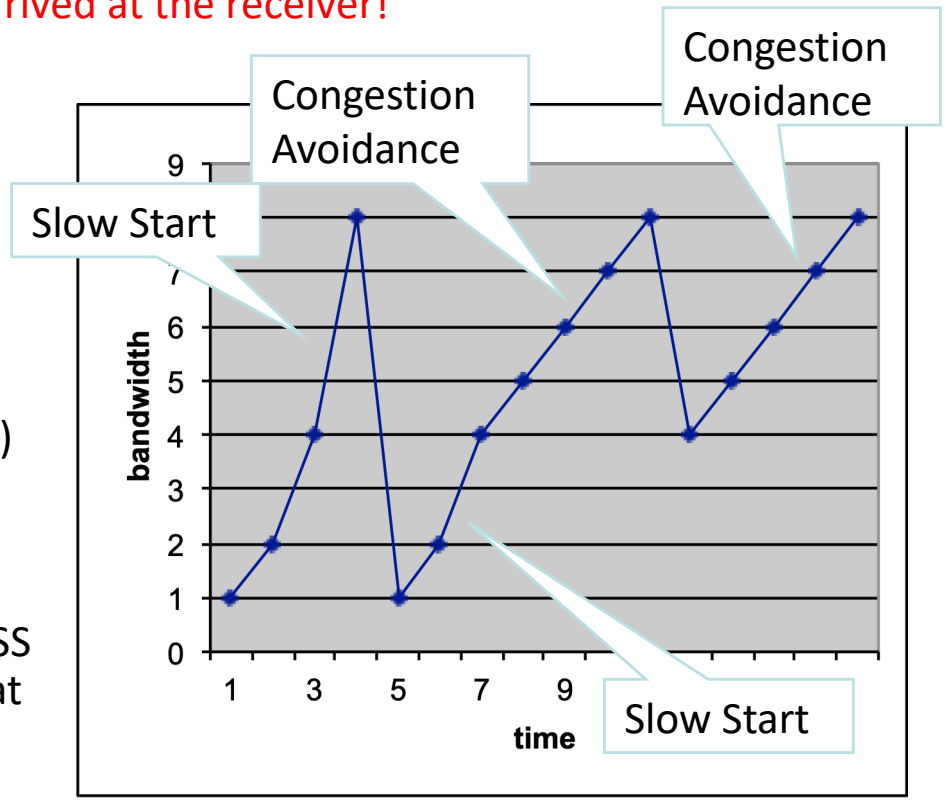
- Note: because of how RTO is calculated, not much gain from sampling more than once per RTT

Fast Retransmit / Fast Recovery (FR/FR) (Reno)

Reasoning: slow start = restart; assume that network is empty

But even similar incoming ACKs indicate that packets arrived at the receiver!

Thus, slow start reaction = too conservative.



1. Upon reception of third duplicate ACK (DupACK):
   ssthresh = FlightSize/2

2. Retransmit the lost segment (fast retransmit);
   cwnd = ssthresh + 3*SMSS
   ("inflates" cwnd by the number of segments (three) that have left the network and which the receiver has buffered)

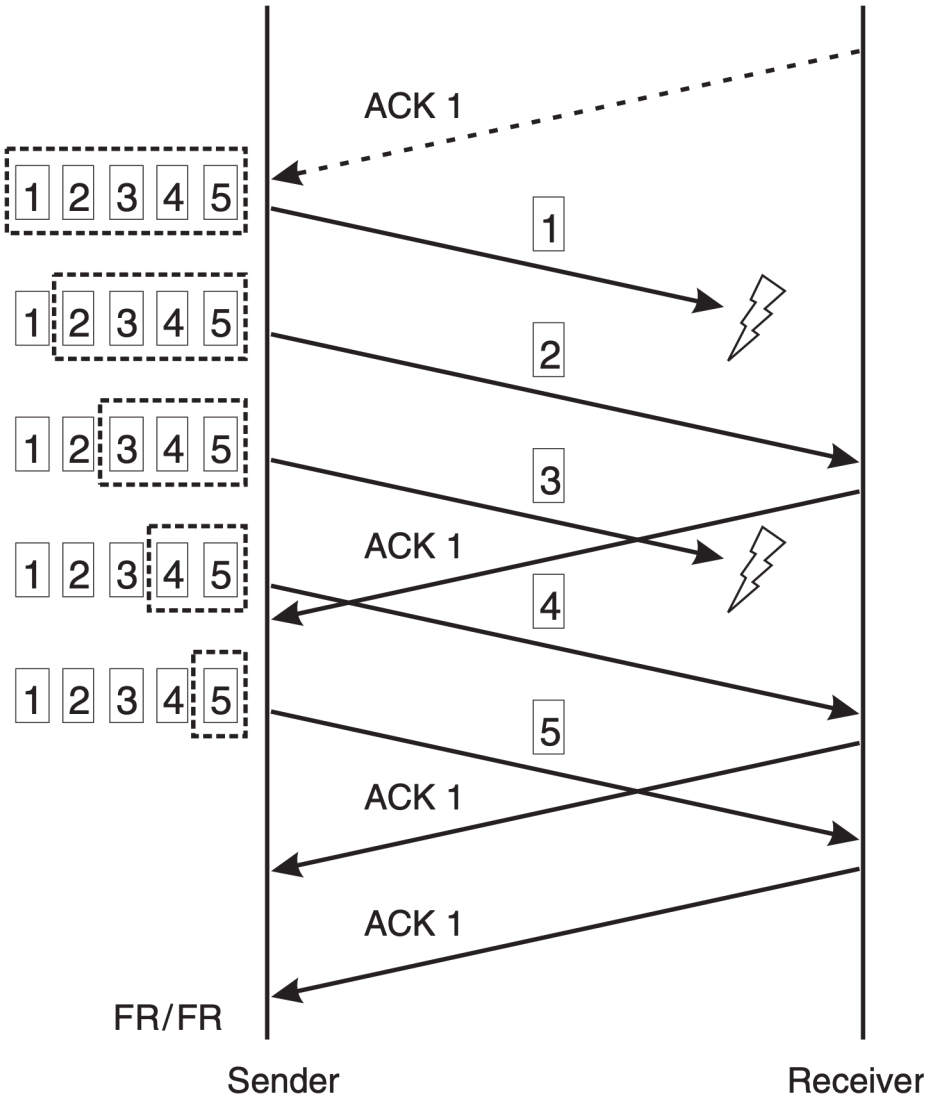
3. For each additional DupACK received: cwnd += SMSS
   (inflates cwnd to reflect the additional segment that has left the network)

4. Transmit a segment, if allowed by the new value of cwnd and rwnd

5. Upon reception of ACK that acknowledges new data (“full ACK“):
   "deflate" window: cwnd = ssthresh (the value set in step 1)

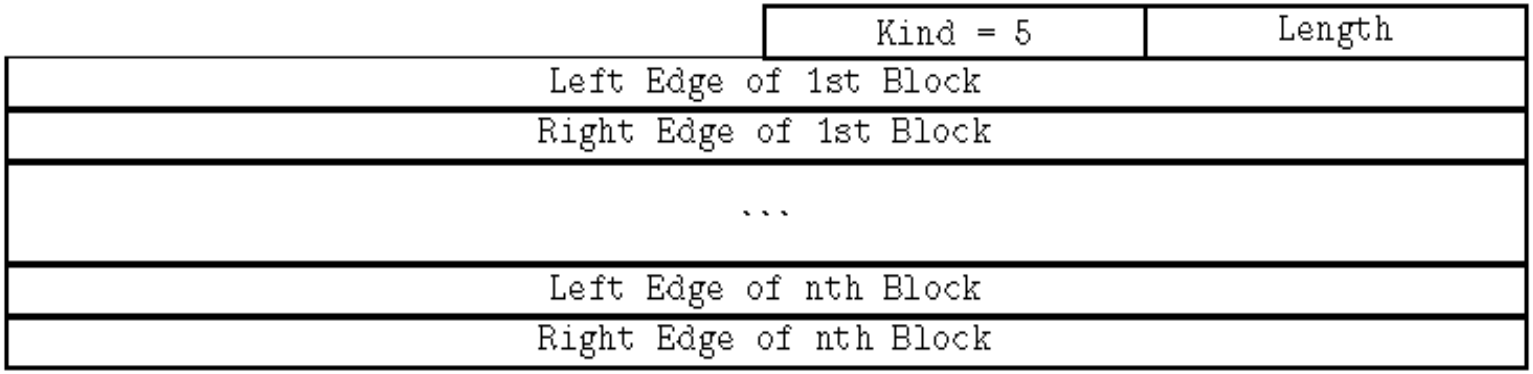
Remember: goal is to quickly + correctly reach ssthresh

Multiple dropped segments



- Sender cannot detect loss of multiple segments from a single window
  - Insufficient information in DupACKs
- NewReno:
  - stay in FR/FR when partial ACK arrives after DupACKs
  - retransmit single segment
  - only full ACK ends process
- Important to obtain enough ACKs to avoid timeout
  - Limited transmit: also send new segments for first two DupACKs
  - Early retransmit: resend old data if there's not enough to send

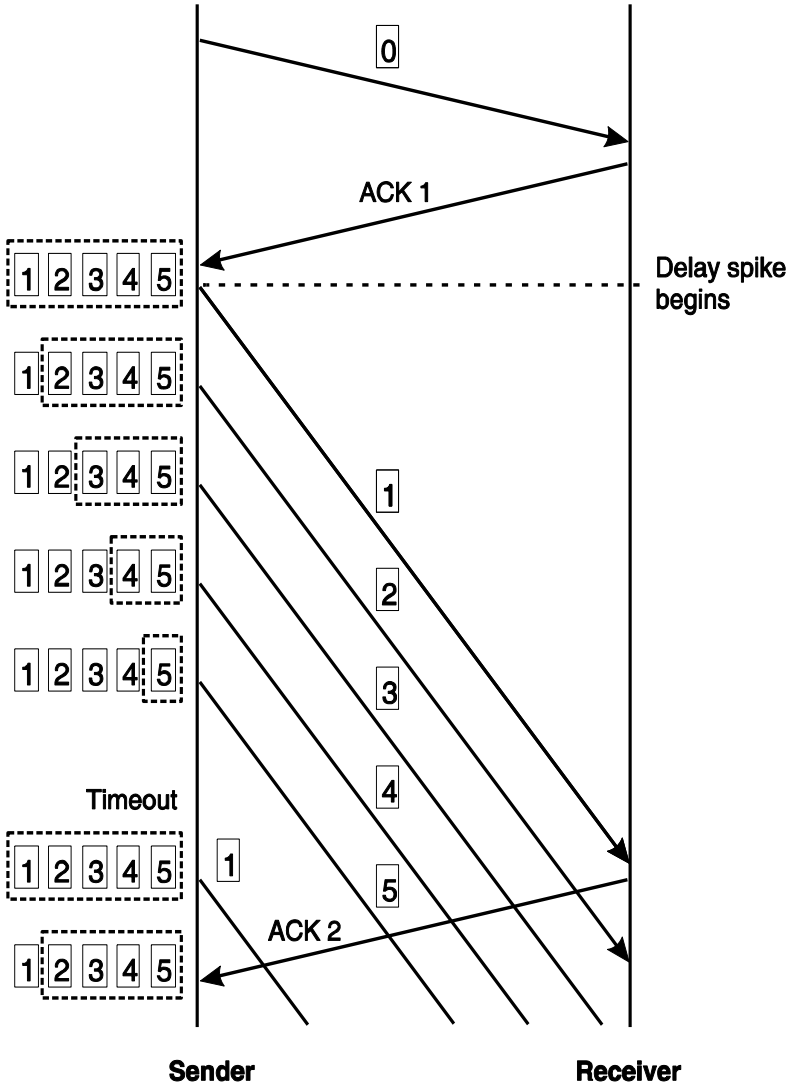
Selective ACKnowledgements (SACK)



- Example on NewReno slide: send ACK 1, SACK 3, SACK 5 in response to segment #4
- Better sender reaction possible
  - (New)Reno can only retransmit 1 segment / RTT, SACK can retransmit more
  - Particularly advantageous when window is large (long fat pipes)
- Extension: DSACK informs the sender of duplicate arrivals
- Reaction to SACK open to implementer, but must follow general CC rules
  - Originally: IETF-recommended "conservative" algorithm (RFC 6675)
  - Nowadays, most common: **Recent Acknowledgment (RACK)**; details later...

Spurious timeouts

- Possible occurrence in e.g. wireless scenarios (handover): sudden delay spike
- Can lead to timeout
  → slow start
  - But: underlying assumption: “pipe empty“ is wrong! (“spurious timeout“)
  - Old incoming ACK after timeout should be used to undo the error
- Several methods proposed
  Examples:
  - Eifel Algorithm: use timestamps option to check: timestamp in ACK < time of timeout?
  - DSACK: duplicate arrived
  - F-RTO: after RTO, send one retransmit, then, if ACK advances the window, send new data; if new data gets ACKed, timeout was spurious



Appropriate Byte Counting

- Increasing in Congestion Avoidance mode: common implementation (e.g. Jan'05 FreeBSD code): cwnd += SMSS*SMSS/cwnd for every ACK (same as cwnd += 1/cwnd if we count segments)
  - Problem: e.g. cwnd = 2: 2 + 1/2 + 1/ (2+1/2)) = 2+0.5+0.4 = 2.9
    thus, cannot send a new packet after 1 RTT
  - Worse with delayed ACKs (cwnd = 2.5)
  - Even worse with ACKs for less than 1 segment (consider 1000 1-byte ACKs)
    → too aggressive!

- Solution: Appropriate Byte Counting (ABC)
  - Maintain bytes_acked variable; send segment when threshold exceeded
  - Works in Congestion Avoidance; but what about Slow Start?
    - Here, ABC + delayed ACKs means that the rate increases in 2*SMSS steps
    - If a series of ACKs are dropped, this could be a significant burst ("micro-burstiness"); thus, limit of 2*SMSS per ACK recommended

Proportional Rate Reduction (PRR)

- Generalization (any back-off factor) of Linux' Rate-Halving
  - Rate-halving avoids burst + pause behavior of FACK or RFC 6675 "conservative loss recovery" algorithm; "paces" segments
  - Implements, for FR, common logic: Slow Start when cwnd<ssthresh

  Example from RFC6937 (X = lost, N = new, R = retransmit):

RFC 6675

ack#	X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cwnd:		20	20	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
pipe:		19	19	18	18	17	16	15	14	13	12	11	10	10	10	10	10	10	10	10
sent:		N	N	R							N	N	N	N	N	N	N	N		

Rate-Halving (Linux)

ack#	X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cwnd:		20	20	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11
pipe:		19	19	18	18	17	17	16	16	15	15	14	14	13	13	12	12	11	11	10
sent:		N	N	R	N		N		N		N		N		N		N		N	

Next: measures to help short flows

- Short flows are often interactive; latency matters
  - Large bulk data transfer not usually latency-critical

- Every packet matters:
  drop => retransmit => user-perceived latency
  - Good to send much, fast (speed up slow start)
  - Shaving off round-trips: when all the data can be sent in e.g. 1 RTT, handshake latency = ½ of the time
  - Tail loss: FR can't work when no more data to send, hence no more ACK arrives

- Note: short flows ≈ application-limited flows ("thin streams") (also: rwnd-limited flows!)

Short flows are common

IFI research

- 6 minute Internet traffic traces from 12:59, 17. 1. 2019
  - Center for Applied Internet Data Analysis (CAIDA): “equinix-nyc” high-speed network monitor node, located at the Equinix datacenter in New York and connected to an OC192 backbone link (9953 Mbps) of a Tier-1 ISP between New York, NY,and Sao Paulo, Brazil.
  - Consider connections that terminate within this time interval

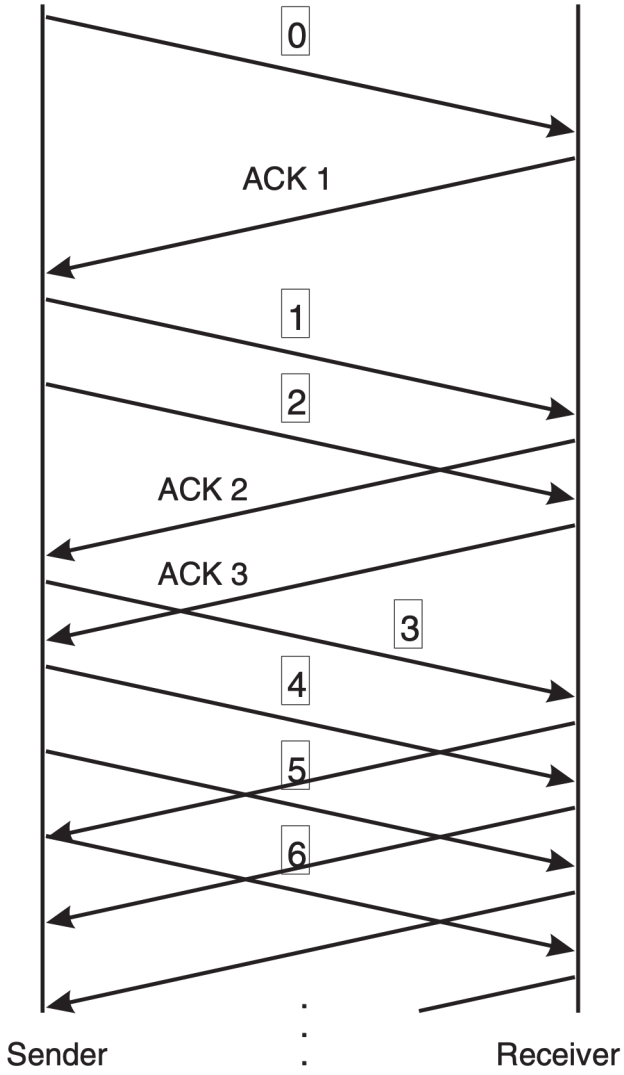
- Statistics
  - Server-client / client-server: 75% of connections shorter than 10 / 5 Kbit
  - Total: 88042 connections, ~ 48.5 Gbyte
  - Client to server: 3952 (4.49%) of connections had retransmissions
  - Server to client: 2523 (2.86%) of connections had retransmissions
  - Total: 6475 (7.35%) of connections had retransmissions

- No retransmission ➔ no loss ➔ no capacity probing

Increasing the Initial Window (IW)

- Slow start: 3 RTTs for 3 packets = inefficient for very short transfers
  - Example: HTTP Requests

- Thus, initial window since ~2002:
  IW = min(4*MSS, max(2*MSS, 4380 byte))
  (typically 4)
- Since ~2013:
  IW = min (10*MSS, max (2*MSS, 14600))
  (typically 10)
  - Adopted in Linux as default since kernel 2.6.39 (May 2011)
  - Note: cwnd after timeout ("Loss Window" (LW)) still 1



TCP Fast Open (TFO)

Similar functionality exists in QUIC !

- Builds on T/TCP idea: allow HTTP GET on SYN, respond with data + SYN/ACK
  - There, the problem was: DoS attack surface
- Solution:
  - First handshake like normal, server gives client cookie and remembers it (locally configurable time)
  - Later handshakes: SYN + data + cookie
- Remaining problem: server cannot tell original from retransmitted SYN => application must be able to accept duplicate data (changes semantics, also API)
  - Not a big problem for a web server



Tail loss

- Consider the "tail" of a transmission
  - e.g., segments 8, 9, 10 of a total 10-segment transfer

- Segment 8 lost: we get 2 DupACKs
  - If we have new data to send, Limited Transmit allows us to do that (which will give us another DupACK and we can enter FR, where we can retransmit)
  - Else, Early Retransmit allows us to resend segment 8

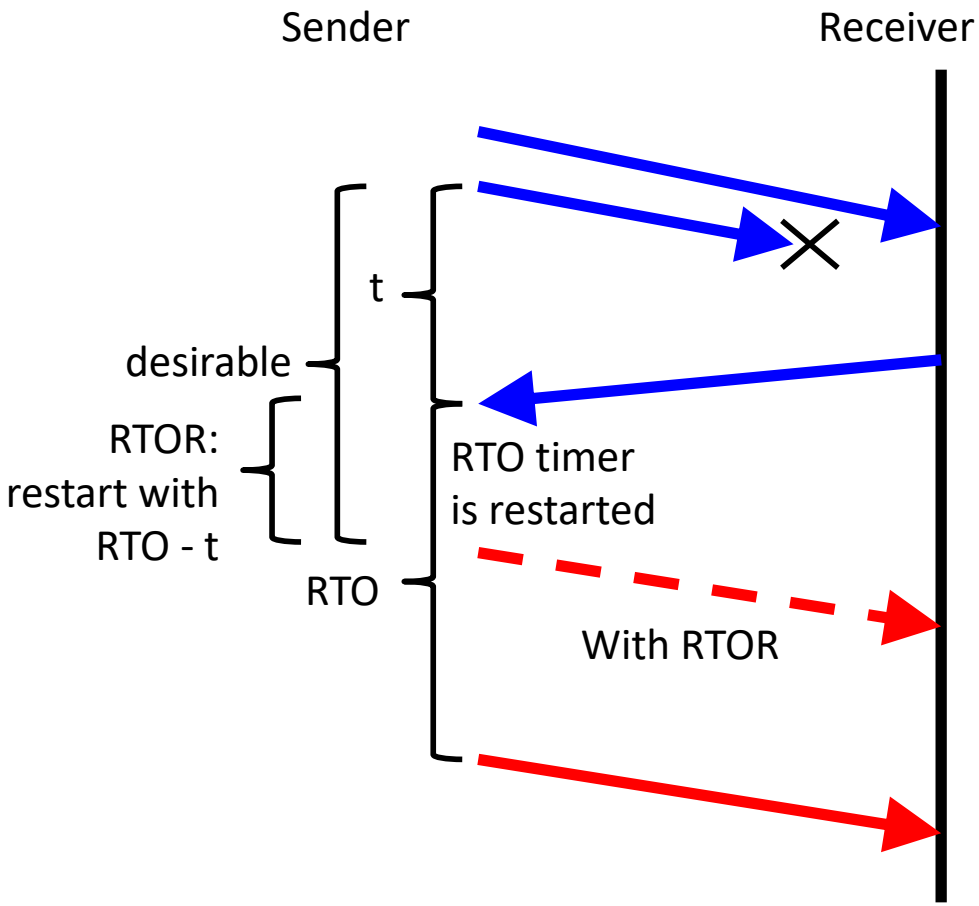
- Segment 10 lost: we get no more ACKs, only the RTO can help us...

RTO Restart (RTOR)

IFI research

RFC 7765

- In some cases TCP/SCTP must use RTO for loss recovery
  - e.g., if a connection has 2 outstanding packets and 1 is lost
- However, the effective RTO often becomes RTO = RTO + t
  - Where t ≈ RTT [+delACK]
- The reason is that the timer is restarted on each incoming ACK (RFC 6298, RFC 4960)
- RTOR rearms timer as: RTO = RTO - t



Tail Loss Probe (TLP)

- From draft-dukkipati-tcpm-tcp-loss-probe-01:

  "Measurements on Google Web servers show that approximately 70% of retransmissions for Web transfers are sent after the RTO timer expires, while only 30% are handled by fast recovery."

  "...distribution of RTO/RTT values on Google Web servers.
  [percentile, RTO/RTT]: [50th percentile, 4.3]; [75th percentile, 11.3];
  [90th percentile, 28.9]; [95th percentile, 53.9]; [99th percentile, 214]."
  "... typically caused by variance in measured RTTs..."

- Idea: more aggressive timer allows to send one single packet ("probe") before RTO fires
  - timer: 2*SRTT
    (+extra time for DelACK if FlightSize==1)
  - new, if data available, else resend

Recent ACKnowledgment (RACK)

IFI research: minor contribution

- Main idea: use time instead of sequence numbers (avoid basing logic on DupThresh)
  - Multiple benefits: eliminates need for much loss recovery logic (drastic simplification!), works with every packet (also retransmits), ..
- Packet A is lost if some packet B sent sufficiently later is (s)acked
  - "Sufficiently later": later by at least a "reordering window" (RACK.reo_wnd, default min_rtt / 4)
  - min_rtt calculated from RTTs per ACK; tried seeding with SRTT or most recent RTT, no major difference
- Also: arm a timer to detect loss in case no ACK arrives
  - TLP is a special case; merged with RACK [RFC 8985]
  - Conceptually, RACK arms a (virtual) timer on every packet sent, times updated with new RTT samples

RACK+TLP: now used almost(?) everywhere!

RACK examples: sender sends P1, P2, P3

(more than RACK.reo_wnd time in between them)

- Example 1: P1 and P3 lost
  - P2 SACK arrives => P1 lost, retransmit (R1)
  - R1 is cumulatively ACKed => P3 lost, retransmit (R3)
  - No timer needed
- Example 2: P1 and P2 lost
  - P3 SACK arrives => P1, P2 lost, retransmit (R1, R2)
  - R1 lost again but R2 SACKed => R1 lost, retransmit
  - Common with rate limiting from token bucket policers with large bucket depth and low rate limit
  - Retransmissions often lost repeatedly because CC. requires multiple RTTs to reduce the rate below the policed rate

- No DupACK based solution can detect such losses!

Conclusion on TCP

- We focused on what is commonly implemented
  - We also skipped some things: header compression, authentication, sequence number attacks, implementation specifics (e.g. TCP_NOTSENT_LOWAT)...
- RFCs contain many more (partially implemented) TCP mechanisms
  - Appropriate behavior when the sender doesn't fully utilize cwnd
    - Reducing cwnd when the sender doesn't have data to send (CWV)
    - Correctly increasing cwnd when the sender is rate-limited
  - Being more robust to reordering (TCP-NCR)
  - Doing congestion control for ACKs (ACK-CC)
  - Adjusting user timeout (when TCP says "it's over") at both ends (UTO)
  - Constraining Slow Start overshoot (HyStart++, Limited Slow-Start)
- ...and some old ideas "took off" later (e.g. T/TCP, ECN)
  - Or not? Time will tell

Extra slides
(not pensum)

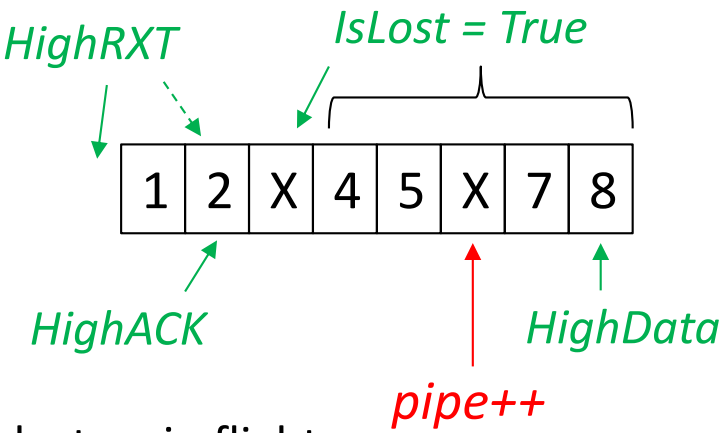
RFC 6675 SACK loss recovery

SACK loss recovery: definitions

- **HighData**: highest seqno transmitted
- **HighACK**: seqno of highest cumulatively ACKed byte
- **HighRxt**: highest seqno retransmitted in this loss recovery phase
- **Pipe**: sender's estimate of the number of bytes outstanding in the network. Key variable for cc. because now this number is explicit.
- **DupAcks**: # DupACKs received since last cumulative ACK
  (DupACK = segment containing SACK block that identifies previously unacked and un-SACKed bytes between HighACK and HighData)
- **DupThresh**: # DupACKs needed to trigger retransmission (normally 3)
- **Scoreboard**: data structure to keep track of sequence number ranges
- **RescueRxt**: highest seqno which has been optimistically retransmitted to prevent stalling of the ACK clock when there is loss at the end of the window and no new data is available for transmission.

SACK loss recovery: scoreboard functions

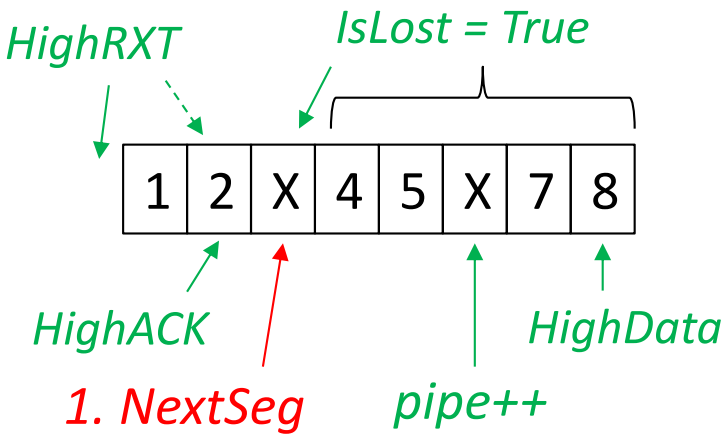
- **Update()**
  - Mark all cumulatively ACKed or SACKed bytes, record total # SACKed bytes
- **IsLost(SeqNo)**
  - True when either DupThresh discontiguous SACKed sequences have arrived above SeqNo, or more than (DupThresh - 1) * SMSS bytes with sequence No's > SeqNo have been SACKed
- **SetPipe()**
  - pipe = 0
  - for(S1=HighACK; S1<=HighData; S1++)
    - if(scoreboard[S1] unsacked)
      - if !IsLost(S1):
        - » pipe++ // not SACKed, not lost => in flight
      - if S1 <= HighRxt:
        - » pipe++ // retransmitted, not lost => 2* in flight



SACK loss recovery: scoreboard functions /2

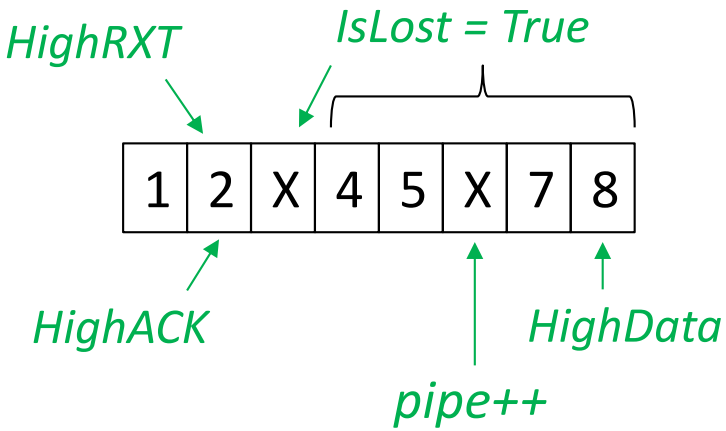
- **NextSeg()** (what to transmit next)
  1. if there exists S2 such that:
     a) S2 > HighRxt
     b) S2 < highest byte covered by any received SACK
     c) IsLost(S2) == true

     ... return max. SMSS sequence range starting with S2
  2. else, if there is unsent data and rwnd allows, return max. SMSS sequence range starting with HighData+1
  3. else, if there exists S3 for which 1a) and 1b) are true, return one segment of max. SMSS bytes starting with S3
  4. else, if HighACK > RescueRxt (or RescueRxt undefined), return one segment of max. SMSS bytes that includes the highest outstanding unSACKed seq.no, and set RescueRxt to RecoveryPoint *(HighData)*. Do not update HighRxt.
  5. else fail (nothing returned)
- Rules 3 and 4 are a retransmission "last resort"



When an ACK with SACK info arrives...

- Run Update()
- Cumulative ACK? If so, DupAcks = 0
- DupACK? If so, and not in FR yet:
  - DupAcks++
  1. If DupAcks >= DupThresh, goto (4)
  2. If DupAcks < DupThresh but IsLost(HighACK + 1), goto (4)
  3. Send new segments (Limited Transmit):
     1. Set HighRxt to HighACK
     2. Run SetPipe ()
     3. If (cwnd - pipe) >= 1 SMSS, there exists previously unsent data, and rwnd allows, transmit up to 1 SMSS of data starting with the byte HighData+1 and update HighData to reflect this transmission, then return to (3.2)
     4. Terminate processing of this ACK



...cont'd: entering FR/FR (step 4)

1. RecoveryPoint = HighData
2. ssthresh = cwnd = (FlightSize / 2)
   (Segments sent as part of Limited Transmit not counted in FlightSize)
3. Retransmit the first data segment presumed dropped:
   the segment starting with sequence number HighACK + 1.
   To prevent repeated retransmission of the same data or a premature rescue retransmission, set both HighRxt and RescueRxt to the highest sequence number in the retransmitted segment.
4. Run SetPipe()
5. In order to take advantage of potential additional available cwnd, proceed to transmission step of FR algorithm (next)

Also upon timeout!

FR algorithm

- ACK arrives: Cumulative ACK for seqno > RecoveryPoint?
  - yes: end FR; keep scoreboard info above HighACK
  - no: run Update() and SetPipe()

- cwnd-pipe >= 1 SMSS? transmit segments!
  1. Send based on NextSeg() (stop this if NextSeg() fails)
  2. If any of the bytes sent in 1) are below HighData, set HighRxt to the highest sequence number of the retransmitted segment unless NextSeg() rule (4) was invoked for this retransmission. (rescueRxt)
  3. If any of the bytes sent in 1) are above HighData, update HighData
  4. pipe += bytes transmitted in 1)
  5. If cwnd - pipe >= 1 SMSS, return to 1)