Internet Technology

The Transmission Control Protocol (TCP)

Michael Welzl http://www.welzl.at

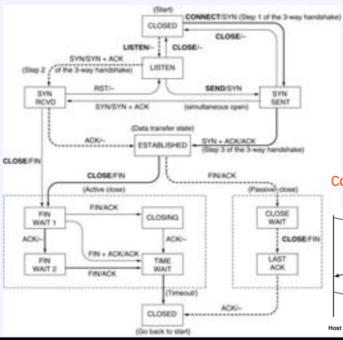
DPS NSG Team http://dps.uibk.ac.at/nsg
Institute of Computer Science
University of Innsbruck, Austria

TCP Header

<table>
<tr><td colspan="8">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="9">Sequence Number</td></tr>
<tr><td colspan="9">Acknowledgement Number</td></tr>
<tr><td>Header Length</td><td>Reserved</td><td>C W R</td><td>E C E</td><td>U R G</td><td>A C K</td><td>P S H</td><td>R S T</td><td>S Y N</td><td>F I N</td><td>Window</td></tr>
<tr><td colspan="8">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="9">Options (if any)</td></tr>
<tr><td colspan="9">Data (if any)</td></tr>
</table>

- 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

TCP Connection Management

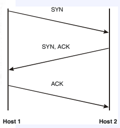


heavy solid line: normal path for a client

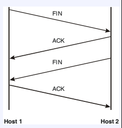
heavy dashed line: normal path for a server

Light lines: unusual events

Connection setup

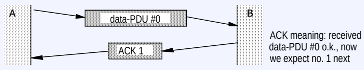


teardown



Error Control: Acknowledgement

ACK ("positive" Acknowledgement)



Purposes:
- sender: throw away copy of SDU held for retransmit,
- time-out cancelled
- msg-number can be re-used

TCP counts bytes, not segments; ACK carries "next expected byte" (#+1)

ACKs are cumulative
- ACK *n* acknowledges all bytes *"last one ACKed"* thru *n-1*

ACKs should be delayed
- TCP ACKs are unreliable: dropping one does not cause much harm
- Enough to send only 1 ACK every 2 segments, or at least 1 ACK every 500 ms (often set to 200 ms)

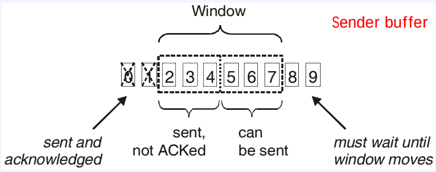
Error Control: Retransmit Timeout (RTO)

- Go-Back-N behavior in response to 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#)
- Original suggestion in RFC 793: Exponentially Weighed Moving Average (EWMA)
  - $SRTT = (1-\alpha)\ SRTT + \alpha\ RTT$
  - $RTO = min(UBOUND, max(LBOUND, \beta * SRTT))$
- Depending on variation, this RTO may be too small or too large; thus, final algorithm includes variation (approximated via mean deviation)
  - $SRTT = (1-\alpha)\ SRTT + \alpha\ RTT$
  - $\delta = (1 - \beta) * \delta + \beta * [SRTT - RTT]$
  - $RTO = SRTT + 4 * \delta$

RTO calculation

- Problem: retransmission ambiguity
  - Segment #1 sent, no ACK received → segment #1 retransmitted
  - Incoming ACK #2: cannot distinguish whether original or retransmitted segment #1 was ACKed
  - Thus, cannot reliably calculate RTO!
- Solution [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: 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; facilitates NAT detection

Window management

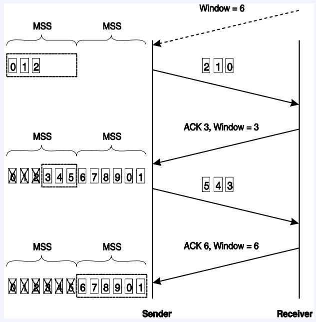


- Receiver “grants“ credit (receiver window, rwnd)
  - sender restricts sent data with window
- Receiver buffer not specified
  - i.e. receiver may buffer reordered segments (i.e. with gaps)

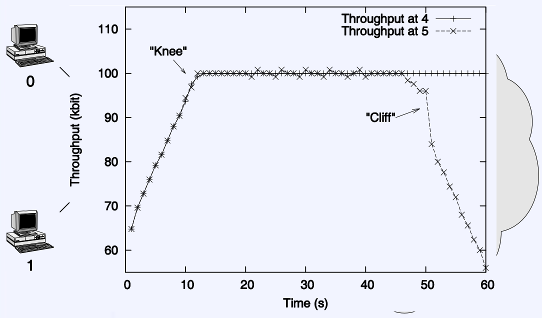
Silly Window Syndrome (SWS)

Called „congestion collapse“ by John Nagle in RFC 896

- Consider telnet: slow typing = large header overhead
  - Solution: wait until segment is filled at the sender (exception: PUSH bit)
  - But what about Is <return>?
- Nagle algorithm: sender waits until SMSS bytes can be sent
  - but 1 small segment /RTT allowed
  - A TCP implementation must support disabling Nagle
- Also, receiver mechanism: slowly reduce rwnd when less than a segment of incoming data until window boundary reached
  - Note that delayed ACKs also help: ACK 3 would not have happened



Congestion collapse



Global congestion collapse in the Internet

Craig Partridge, Research Director for the Internet Research Department at BBN Technologies:

```
Bits of the network would fade in and out, but usually only for TCP. You
could ping. You could get a UDP packet through. Telnet and FTP would fail
after a while. And it depended on where you were going (some hosts were
just fine, others flaky) and time of day (I did a lot of work on weekends
in the late 1980s and the network was wonderfully free then).

Around 1pm was bad (I was on the East Coast of the US and you could tell
when those pesky folks on the West Coast decided to start work...).

Another experience was that things broke in unexpected ways - we spent a
lot of time making sure applications were bullet-proof against failures.
(..)

Finally, I remember being startled when Van Jacobson first described how
truly awful network performance was in parts of the Berkeley campus. It
was far worse than I was generally seeing. In some sense, I felt we were
lucky that the really bad stuff hit just where Van was there to see it.
```

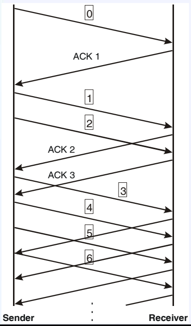

Internet congestion control: History

- **1968/69:** dawn of the Internet
- **1986:** first congestion collapse
- **1988:** "Congestion Avoidance and Control" (Jacobson)
  Combined congestion/flow control for TCP
  (also: variation change to RTO calculation algorithm)
- Goal: stability - in equilibrum, no packet is sent into the network until an old packet leaves
  - ack clocking, “conservation of packets“ principle
  - made possible through window based stop+go - behaviour
- Superposition of stable systems = stable → network based on TCP with congestion control = stable

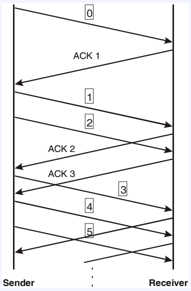
TCP Congestion Control: Tahoe

- Distinguish:
  - flow control: protect receiver against overload
    (receiver "grants" a certain amount of data ("receiver window" (rwnd)))
  - congestion control: protect network against overload
    ("congestion window" (cwnd) limits the rate: min(cwnd,rwnd) used!)
- Flow/Congestion Control combined in TCP. Two basic algorithms:
  (window unit: SMSS = Sender Maximum Segment Size, usually adjusted to Path MTU; init cwnd<=2 (*SMSS), ssthresh = usually 64k)
- Slow Start: for each ack received, increase cwnd by 1
  (exponential growth) until cwnd >= ssthresh
- Congestion Avoidance: each RTT, increase cwnd by at most one segment
  (linear growth - "additive increase")
- Timeout: ssthresh = FlightSize/2 (exponential backoff - "multiplicative decrease"), cwnd = 1; FlightSize = bytes in flight (may be less than cwnd)

Slow start and Congestion Avoidance



- Slow start: 3 RTTs for 3 packets = inefficient for very short transfers
- Example: HTTP Requests
- Thus, initial window IW = min(4*MSS, max(2*MSS, 4380 byte))

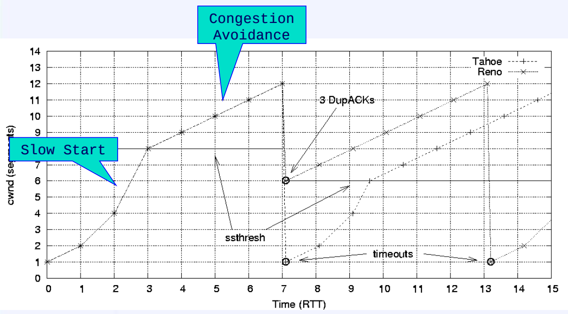


Fast Retransmit / Fast Recovery (Reno)

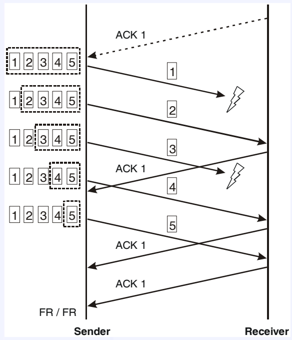
Reasoning: slow start = restart; assume that network is empty
But even similar incoming ACKs indicate that packets arrive at the receiver!
Thus, slow start reaction = too conservative.

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

Tahoe vs. Reno



One window, 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 segment for first two DupACKs

Example: ACK 3

Example: ACK 6

Non-Congestion Robustness (NCR)

- Assumption: 3 DupACKs clearly indicate loss
  - Can be incorrect when packets are reordered
- Reordering is not rare
  - And new mechanisms in the network could be devised if TCP was robust against reordering (e.g. consider splitting a flow on multiple paths)
- Approach: Increase the number of DupACKs N to approx. 1 cwnd
- Extended Limited Transmit; 2 variants
  - Careful Limited Transmit: send 1 new packet for every other DupACK until N is reached (halve sending rate, but send new data for a while)
  - Aggressive Limited Transmit: send 1 new packet for every DupACK until N is reached (delay halving sending rate)
  - Full ACK ends process

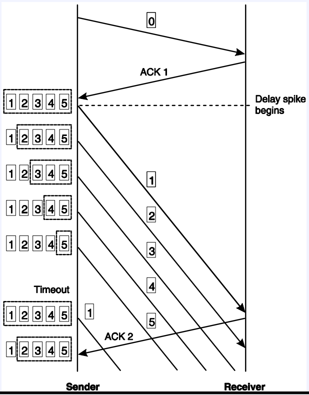
Selective ACKnowledgements (SACK)

Kind = 5	Length
Left Edge of 1st Block	
Right Edge of 1st Block	
...	
Left Edge of nth Block	
Right Edge of nth Block	

- Example on previous slide: send ACK 1, SACK 3, SACK 5 in response to segment #4
- Better sender reaction possible
  - Reno and NewReno can only retransmit a single segment per window
  - SACK can retransmit more (RFC 3517 – maintain scoreboard, pipe variable)
  - Particularly advantageous when window is large (long fat pipes)
- but: requires receiver code change
- Extension: DSACK informs the sender of duplicate arrivals

Spurious timeouts

- Common occurrence in 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: check for ACKs that shouldn't arrive after Slow Start



Appropriate Byte Counting

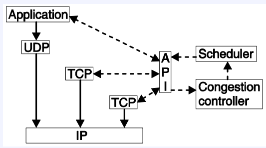
- Increasing in Congestion Avoidance mode: common implementation (e.g. Jan'05 FreeBSD code): cwnd += SMSS*SMSS/cwnd for every ACK (same as cwnd += 1/cwnd if we count segments)
  - Problem: e.g. cwnd = 2: 2 + 1/2 + 1/ (2+1/2)) = 2+0.5+0.4 = 2.9
    thus, cannot send a new packet after 1 RTT
  - Worse 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

Limited Slow Start and cwnd Validation

- Slow start problems:
  - initial ssthresh = constant, not related to real network
    this is especially severe when cwnd and ssthresh are very large
    - Proposals to initially adjust ssthresh failed: must be quick and precise
  - Assume: cwnd and ssthresh are large, and avail.bw. = current window + 1 SMSS/RTT ?
    - Next updates (cwnd++ for every ACK) will cause many packet drops
- Solution: Limited Slow Start
  - cwnd <= max_ssthresh: normal operation; recommend. max_ssthresh=100 SMSS
  - else K = int(cwnd/(0.5*max_ssthresh), cwnd += int(MSS/K)
  - More conservative than Slow Start:
    for a while cwnd+=MSS/2, then cwnd+=MSS/3, etc.
- Cwnd validation
  - What if sender stops, or does not send as much as it could?
    - maintain cwnd = wrong if break is long (not related to real network anymore)
    - reset = too conservative if break is short
    - Solution: slowly decay TCP parameters - cwnd /= 2 every RTT,
      ssthresh = between previous and new cwnd

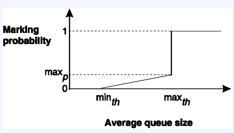
Maintaining congestion state

- TCP Control Block (TCB): information such as RTO, scoreboard, cwnd, ..
- Related to network path, yet separately stored per TCP connection
  - Compare: layering problem of PMTU storage
- TCB interdependence: affects initialization phase
  - Temporal sharing: learn from previous connection (e.g. for consecutive HTTP requests)
  - Ensemble sharing: learn from existing connections here, some information should change - e.g. cwnd should be cwnd/n, n = number of connections; but less aggressive than "old" implementation
- Congestion Manager
  - One entity in the OS maintains all the
  - congestion control related state
  - Used by TCP's and UDP based applications
  - Hard to implement, not really used

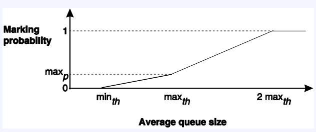


Active Queue Management

- Monitor queue, do not only drop upon overflow ⇒ more intelligent decisions
- Goals: eliminate phase effects, manage fairness ("punish" flows that are too aggressive)
  - Aggressive flows have more packets in the queue; thus, dropping a random one is more likely to affect such flows
  - Also possible to differentiate traffic via drop function(s)



RED

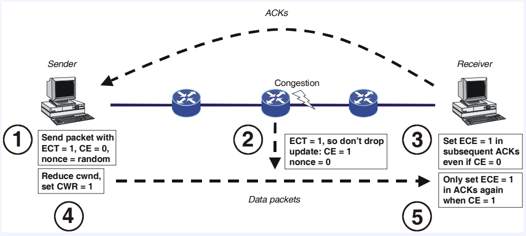


RED in "gentle" mode

Explicit Congestion Notification (ECN)

- Explicit Congestion Notification (ECN)
  - Instead of dropping, set a bit
- Receiver informs sender about bit; sender behaves as if a packet was dropped
  ⇒ actual communication between end nodes and the network
- Note: ECN = true congestion signal (i.e. clearly not corruption)
- Typical incentives:
  - sender = server; efficiently use connection, fairly distribute bandwidth
    - use ECN as it was designed
  - receiver = client; goal = high throughput, does not care about others
    - ignore ECN flag, do not inform sender about it
- Need to make it impossible for receiver to lie about ECN flag when it was set!
  - Solution: nonce = random number from sender, deleted by router when setting ECN
  - Sender believes „no congestion" iff correct nonce is sent back

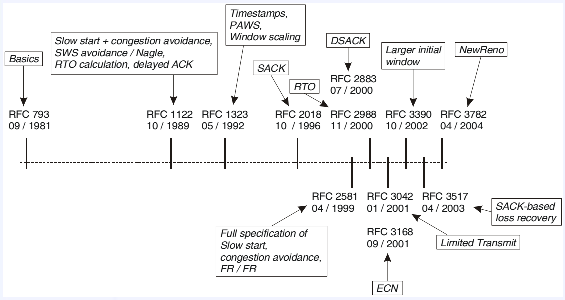
ECN in action



- Nonce provided by bit combination:
  - ECT(0): ECT=1, CE=0
  - ECT(1): ECT=0, CE=1
- Nonce usage specification still experimental

TCP History

Standards track TCP RFCs which influence when a packet is sent (status: October 2007)



References

- Michael Welzl, "Network Congestion Control: Managing Internet Traffic", John Wiley & Sons, Ltd., August 2005, ISBN: 047002528X
- M. Hassan and R. Jain, "High Performance TCP/IP Networking: Concepts, Issues, and Solutions", Prentice-Hall, 2003, ISBN:0130646342
- M. Duke, R. Braden, W. Eddy, E. Blanton: "A Roadmap for TCP Specification Documents", RFC 4614, September 2006
- NCR (Extended Limited Transmit): RFC 4653
- http://www.ietf.org/html.charters/tcpm-charter.html
- Which TCP features are used in Windows Vista, and why? See: http://www3.ietf.org/proceedings/07mar/slides/tsvarea-3/sld1.htm