ECE 435 – Network Engineering Lecture 14

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

24 February 2025

Announcements

- HW#5 was posted
- Midterm tentatively March 12th
- Will post project info soon

TCP State Machine

- 11 possible states
  - starts in CLOSED
  - LISTEN – waiting for a connection
  - SYN-SENT – started open, waiting for SYN response
  - SYN-RECEIVED – waiting for ACK
  - ESTABLISHED – open, for two-way communication
  - FIN-WAIT-1 – application has said it's finished
  - FIN-WAIT-2 – the other side agreed to release
  - CLOSE-WAIT – waiting for a termination request

  - CLOSING – waiting for an ACK of closing request both sides closed at once
  - LAST-ACK – waiting for ACK from last closing
  - TIME-WAIT – waiting to transition to CLOSED long enough to ensure other side gets last ACK
- There is a large state diagram you can lookup

TCP State Machine – Tools

- Linux has tools that can show you socket states
  - `netstat` was traditional (now obsolete)
  - `ss` (socket status) is current

    has lots of options, things like `ss -a -i` will show more than you ever want to know

```
$ ss -a
Netid State  Recv-Q Send-Q  Local Address:Port      Peer Address:Port
tcp   ESTAB  0      0     192.168.8.146:43294     192.168.8.47:ssh
tcp   LISTEN 0      128     0.0.0.0:ssh             0.0.0.0:*
```

Typical Connection seen by Client

- CLOSED

  user does connect(), SYN sent (step 1 of handshake)
- SYN-SENT

  waits for SYN+ACK, sends ACK (step 3 of handshake)
- ESTABLISHED

  sends/receives packets

  eventually user will close() and send FIN
- FIN-WAIT-1

  FIN sent, waiting for ACK

- FIN-WAIT-2

  one direction closed

  received ACK of FIN, wait for FIN from other side, respond with ACK
- TIME-WAIT

  wait until timeout to ensure all packets done in case ACK got lost
- CLOSED

Typical Connection seen by Server

- CLOSED

  waits for listen()
- LISTEN

  gets SYN, sends SYN+ACK (step 2 of handshake)
- SYN-RECVD

  waits for ACK
- ESTABLISHED

  sends/receives

  FIN comes in from client, sends ACK

- CLOSE-WAIT

  closes itself, sends FIN
- LAST-ACK

  gets ACK
- CLOSED

TCP Reliability – Per Segment

- checksum (algo same as UDP), also drops silently on error
- also covers some fields in IP header to make sure at right place
- TCP checksum is mandatory
- Checksum is fairly weak compared to crc32 in Ethernet
  - Catch the error 99.9984% of time. Is that enough? At gigabit speeds this could be a few packets per second with errors

- See “Stone and Partridge” *When The CRC and TCP Checksum Disagree*

TCP Reliability – Per Flow

- What to do in face of lost packets? Need to notice and retransmit and handle out-of-order
- Sequence number generated for first blob (octet?), 32-bit number in header
- Sender tracks sequence of what has been sent, waiting for ACK
- On getting segment, receiver replies with ACK with number indicating the expected next sequence number, and how much has been received. ”All data preceding X

has been received, next expected sequence number is Y. Send more”

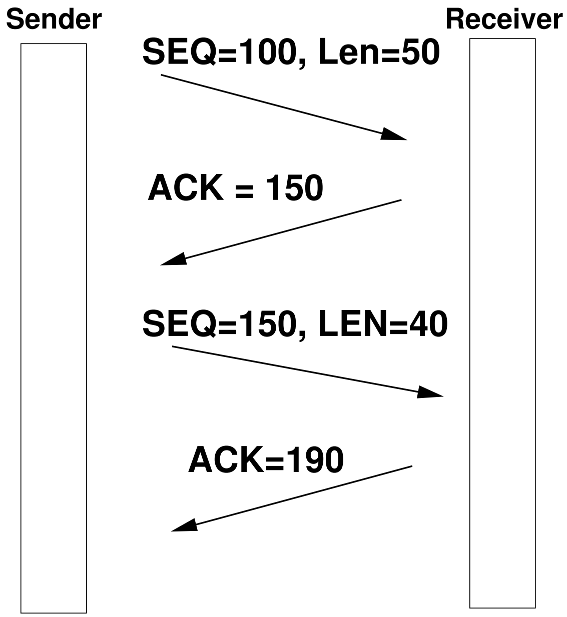
TCP Reliability – Advanced ACK handling

- Cumulative ACK – all previous data previous to the ACK has been received
- Selective ACK (requires options?) – can indicate which missing segments need to be resent

Ways to Notice Transmission Problems

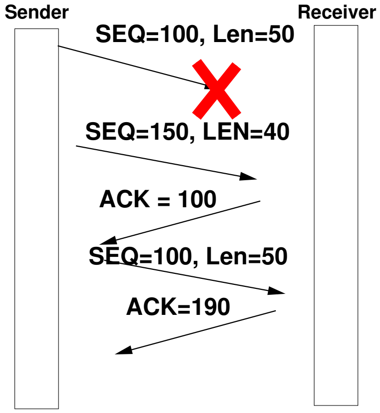
- Checksum
- Acknowledgement
- Time-out

For Comparison: Good Transaction



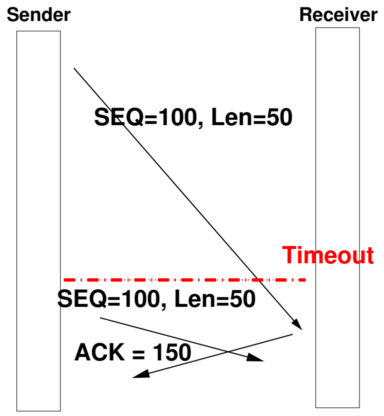
- You don't have to wait for ACK before sending more
- ACKs can be piggybacked on packets going other direction

Error: Corrupted or Lost Packet



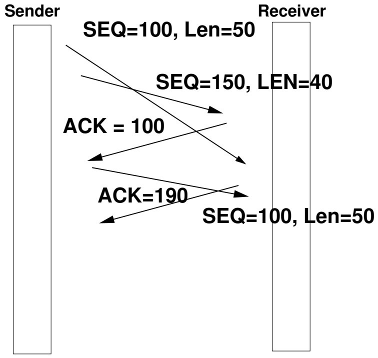
- Packet never made it
- Can't ACK next packet due to missing data, so re-sends previous ACK (ACK=100)
- When re-send? Timer? After 3 duplicate ACKs?

Error: Delay or Duplicate Packet



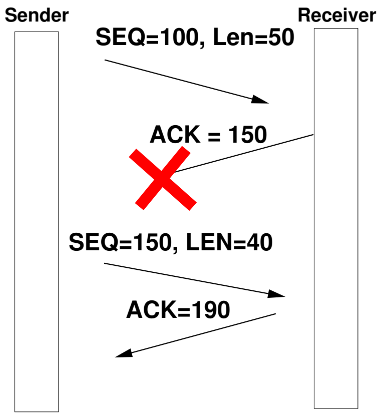
- Duplicate packet (How? On sender timeout happens if ACK not received in reasonable time, so resends)
- Two identical packets arrive at receiver
- TCP discards packets with duplicate SEQ (any security issues with that?)

Error: Out-of-order Packet



- Out-of-order packet
- Do not ACK packet until preceding ones make it
- For performance can queue up out of order ones so they don't have to be resent

Error: Lost ACK



- ACKs cumulative, so if the next packet causes an ACK then it doesn't matter. Otherwise a timeout?

TCP Timers

- Timers can catch when things go missing/go wrong
- What should the timer value be?
  - Too short, send extra packets,
  - Too long and takes long time to notice lost packets.
- On the fly measures round trip time. (RTT) When send segment, start timer, updates. Various algorithms. Often 2 or 4x

TCP Timers

- Connection Timer – after send SYN if no response in time, reset
- Retransmission Timer – retransmit data if no ACK
- Delayed ACK timer – can usually wait for outgoing data and can tag an ACK along for free. If it's been too long and no data is being sent, timer expires and have to send stanadlone ACK
- Persist Timer – solve deadlock where window was 0, so waiting, and missed the update that said window was

open again.
Sends special probe packet. Keep trying every 60s?

- Keepalive Timer – if connection idle for a long time, sends probe to make sure still up
- FIN_WAIT_2 Timer – avoid waiting in this state forever if other side crashes
- TIME_WAIT_TIMER – used in TIME_WAIT to give other side time to finish before CLOSE

Flow Control

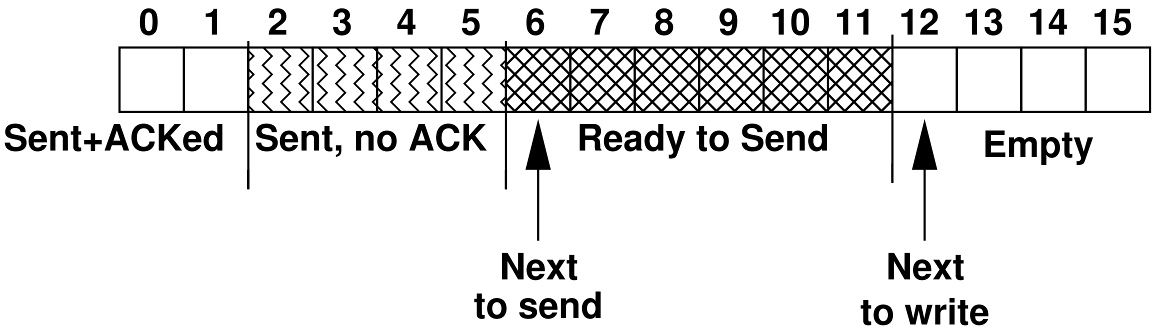
- What happens if a fast computer sending to a slow receiver?
- What if receiver can't keep up?
- Should it just drop packets and request resend when caught up?
- This could potentially waste a lot of sending on the sender's part

How Much Data to Send?

- How much data can be sent before receiving an ACK
- Extreme – just 1 byte. Inefficient (overhead). Also modern systems, a fiber line coast to coast a long time to ACK packet

How does OS/TCP Track Sent Data

- Sliding Window Protocol (Sender Window)
  - Circular buffer holds writes
  - Once data ACKed, can slide (grow/shrink) window
  - Once circular buffer full, write() calls will block



Receiver Window (RWND)

- Receiver “advertises” a window, how much incoming data it can handle
- Example:
  - Receiver has 4k buffer
  - Sender does 2k write (2k/SEQ=0)
  - Receiver sends back ACK=2k, WIN=2048 (can take up to 2k more)
  - Application sends 2k (2k, SEQ=2k)
  - If it is full, receiver might send ACK=4k, WIN=0

- Later once buffer clears up a bit (application reads 2k maybe) sends ACK=4096, WIN=2k
- Sender then sends some more

Receiver Window – Waiting on WIN=0

- When happens when waiting on a WIN=0?
- What if the ACK restarting things gets lost?
- Do you wait forever?
  - ○ Sender can send a "window probe", a 1-byte packet with retransmit window and next byte expected

Window Management / Flow Control

- A simple implementation of TCP might result in a lot of extraneous packets being sent
- Can negatively effect flow control and cause congestion
- Things to note:
  - Senders do not have to transmit incoming data immediately
  - Receivers do not have to ACK immediately
  - Try to avoid 1-byte payloads (which have 40 bytes of overhead if you include TCP and IP headers)

Buffering on Sender Side

- Senders can buffer data
- If know receiver window is 1k, can save up until 1k is ready to send and just send single packet. Can help performance.
- Old Example
  - ○ Typing logged in via telnet (ssh similar, though for encryption reasons you probably wouldn't send just a single byte with ssh)
  - ○ Using editor, press a key. Writes to socket,

immediately sends single-byte packet

- Other end receives it, TCP stack immediately sends ACK with window reduced by 1
- Editor does a read() and gets byte, TCP stack immediately sends updated ACK with window increased by 1
- Editor then actually prints the letter that was typed, which gets sent as another 1-byte packet
- This single key-press results in 4 packets (160x overhead)
- Can we reduce this?

Sender Window Problems

- What if sender only sending 1-byte at time?
- Can do "delayed acknowledgement" where you buffer up to 500ms for additional input. Adds lot of latency.
- Nagel's Algorithm
  - (John Nagel, 1984) RFC896
  - When only sending one byte at a time, send one packet, but buffer the rest until the outstanding data is ACKed
  - Also take into account window size.

- Widely used, can be bad for things like X window forwarding as mouse movements bunched together.
- Interacts poorly with Delayed ACKs
- Often causes despair to people unaware and having to debug latency problems
- TCP_NODELAY option disables.

Receiver Window Problems

- Silly Window Syndrome
- Slow reader on receive side. Application reads one byte out, stack immediately advertises 1-byte window and causes back-and-forth
- Clark's solution
  - If receiving small amounts, close window until buffer half empty and then open again.