Even More TCP, part 2

TCP/IP class

Jim Binkley

outline

- tcp network trace
- timers
- persistance
- keepalive
- tcp options
- tcp current/future research

TCP network trace, 1 of 3

shark> **rsh fish ls .profile**
#etherfind -v -between fishbait shark
1. TCP from shark.1023 to fish.shell seq 45b2c600,
syn, window 4096, <mss 1024>
2. TCP from fish.shell to shark.1023 seq 5e22a200,
ack 45b2c601, syn window 4096, <mss 1024>
3. TCP from shark.1023 to fish.shell seq 45b2c601,
ack 5e22a201, window 4096
4.TCP from shark.1023 to fish.shell seq 45b2c601,
ack 5322a201, window 4096, 5 bytes data

find the 3-way handshake. What did the sequence numbers do?

TCP network trace, 2 of 3

5. TCP from fish.shell to shark.1023 seq 5e22a201,
   ack 45b2c606, window 4096
6. TCP from shark.1023 to fish.shell seq 45b2c606,
   ack 5e22a201, window 4096, 20 bytes data (!)
7. TCP from fish.shell to shark.1023 seq 5e22a201,
   ack 45b2c61a, window 4096, 1 byte data
8. TCP from shark.1023 to fish.shell seq 45b2c61a,
   ack 5322a202, window 4096
9. TCP from fish.shell to shark.1023 seq 5e22a202,
   ack 45b2c61a, window 4096, 9 bytes data

TCP trace, FINs? (finally)

10. TCP from fish.shell to shark.1023 seq 5E22A20B, ack 45B2C61A, FIN, window 4096,
11. TCP from shark.1023 to fish.shell seq 45B2C61A, ack 5E22A20C, window 4087,
12. TCP from fish.shell to shark 1023 seq 5E22A20B, ack 45B2C61A, FIN, window 4096,
13. TCP from shark.1023 to fish.shell seq 45B2C61A, ack 5E22A20C, window 4096,
14. TCP from shark.1023 to fish.shell seq 45B2C61A, ack 5E22A20C, FIN, window 4096,
15. TCP from fish.shell to shark.1023 seq 5E22A20C, ack 45B2C61B, window 4096

TCP Timers

- 1. ack timer - receive ACK within a certain time, else resend (send-side)
- 2. delayed ack timer - try and wait a bit after getting data before sending ack
  (recv-side)
- 3. persistance timer - if window is closed, send 1 byte of data
- 4. keepalive timer - make sure conn. is up
- 5. time wait timer, 1 minute or so, make sure lack ACK and all sequence numbers associated with connection are gone from Internet

silly-window syndrome

- occuring if due to timers, less data than window size is exchanged
- reduces protocol to simple positive ACK with retransmission
- e.g., if recv advertises 1k windows, then later would advertise 4k window, sender might only send 1k to fill window,
- recv must wait

SWS/persistance timer

- recv must not advertise small segments
- sender may use persistance timer to wait for larger window
- persistance timer backoff similar to or same as exponential backoff
- persistance never stops though...

keepalives

- by default (some?) TCP connections can STOP exchanging data and remain up
- this is different from persistance (we don’t have any data to send at the moment)
- routers may reboot or routes could even change physically
- end systems don’t know and don’t care

keepalives

- TCP has “feature” called keepalive timer
- intent is for server that has resources used so that it can “hang up” and reclaim resources if client is not using them actively
- if you use telnet, power-off host, you may leave a “half-open” telnetd on server
- **should this be done by app or by tcp?**

keepalives

- mechanism - one side sends “keepalive” probe every 2 hours, plus 10 * 75 second more probes (2 hours, 12 minutes)
- probe may have seq number set to current - 1 with no valid data (we just want an ACK)
- if no response (nobody home), connection is closed, or RST if rebooted
- time is set in operating system and applies to all possible applications - may not be tunable

keepalive - mobility

- how would a keepalive be good/bad for mobile hosts that might be disconnected for long periods of time?
- some apps like ftp servers have their own app timers built in - no command in N seconds, connection is closed
- good for servers that need to conserve memory
- bad for mobile apps that want to remain connected

options

- header can have options
- options include: NOOP, END, MSS, Window scale factor, Timestamp, etc.
- TLV, 1 byte for tag, 1 byte for length
- MSS or max segment size is negotiated usually via SYN exchange, 2 bytes max
- option can arrive with any segment, should be ignored if not understood

new wrinkles in TCP?!

- path MTU discovery
  - TCP starts with min(local MTU, remote MSS)
  - can’t exceed MSS of other end (default 536)
  - send packets with DF bit sent
  - if we get ICMP error, decrease and try again
  - MSS should be MTU of outgoing i/f
- not so new, hopefully all TCPs use this
  - note: **you cannot block ICMP don’t fragment errors at border router/firewalls**

window-scale option

- goal: make window bigger
- sent in SYN, both sides must exchange else scaling factor remains 0, allowing interoperation with older TCPs
- value is 0, 14 indicates left-shift of real window size contained in 32 bit counter
- S = 0, 65535 * 2 ** 0 = 65535 (no change)
- S = 1, 65535 * 2**1 == 131070
- S = 2, 65535 * 2 ** 2 == 262140, up to S = 14

timestamp option

- value placed in each packet sent
- reflected by recv in ACK returned
- allows TCP to get better granularity of timing since in real life TCP acks once per window
- no clock sync needed between hosts, key idea: time is self-referential
- allows for better RTT calculation

PAWS option - protection against wrapped sequence numbers

- with high speed connections (and even if we use timestamp scaling) the sequence number used by TCP may wrap before packets can leave the network
- gigabit network could wrap in 34 seconds
- timestamp is really only a monotonically increasing value
- can be used to extend sequence # to 64 bits and thus avoid this problem, (ts, sequence) pair

T/TCP transactional TCP

- transaction is defined as follows:
  - avoid connection overhead, if possible we want RPC, send one request, get back ACK or value
  - latency should be reduced to RTT plus service turnaround time
  - server should detect dups and not replay transaction

Transactional TCP

- rfc 1379
- basic idea
  - avoid 3-way handshake
  - shorten TIME_WAIT state
- alternative is Cheritons VMTP - Virtual Message Transaction Protocol, RFC 1045, 1988, can do multicast

re transactions: TCP versus UDP versus ?

- neither TCP nor UDP ideal
- if we use UDP, depending on goal, might add retransmission, sequencing, slow start, checksums, adaptive retransmission ??
- TCP has connection overhead
- idea is to make TCP support transactions with a few mods
- con: still won’t support broadcast/multicast

how might TCP handle mobility?

- problem may exist if mobile devices lose lots of packets at link layer?
- how do we differentiate slow-start and simply being disconnected?

TCP and you know what (WWW)

- does it make sense for HTTP to use TCP?
  - contrarian POV: TCP built for lots of i/o in i/o sream, needs time with slow-start to optimize itself
- HTTP 1.1 with “persistant connection” should help here