ECE 435 – Network Engineering
Lecture 9

Vince Weaver

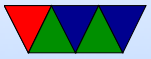
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

26 September 2017

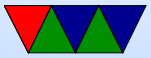
Announcements

- HW#4 was posted, due Thursday
- Hexdump format
- Midterm next Tuesday details next class



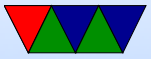
HW#3 Review

- md5sum/encryption, seems to have gone well
- e-mail
  - First warning sign – says its from a bank, but the return address is from a Florida dental school
    Also not a bank of mine
  - pop from deater.net via fetchmail
  - LMTP – local mail transport. LHLO. No mail queue, says right away whether deliver mail is possible.
  - encrypted and verified from UFL, but sent from



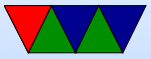
videotron.ca cablemodem

- Virus scanned and SPAM scanned, just sort of barely passed
- pdf attached probably had some sort of exploit or phishing document. Didn't open.

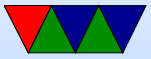


TCP

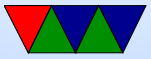
- Transmission Control Protocol
- RFC 793 / 1122 / 1323
- Reliable, in-order delivery.
- Adapts to network congestion
- Takes data stream, breaks into pieces smaller than 64k (usually 1460 to fit in Ethernet) and sends as IP



- No guarantees all packets will get there, so need to retransmit if needed.
- Multiple connections can share same port (i.e. webserver on port 80 can handle multiple simultaneous requests)
- Point-to-point (can't multicast)
- Full duplex
- Byte stream, if program does 4 1024byte writes there's no guarantee the other end sees 4 chunks of 1024, only 4k stream of bytes is guaranteed.



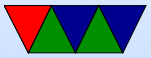
- PUSH flag can be sent that says not to buffer (For example, if interactive command line)
- URGENT flag can be sent that says to transmit everything and send a signal on the other side that things are urgent.



TCP Header

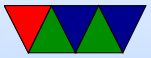
Fixed 20-byte header. Up to 64k-20 in size. Data can be empty.

16-bits	16-bits
Source Port	Destination Port
Sequence Number	
Acknowledgement Number	
Length(4) URG/ACK/PSH/RST/SYN/FIN	Window Size
Checksum	Urgent Pointer
Options (0-32)	
Data (optional)	

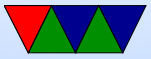


TCP Header Format

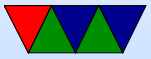
- 16-bit source port
- 16-bit dest port
- 32-bit sequence number
- 32-bit ack number next byte expected, not last one received
- 4-bit header length number of 32-bit chunks (includes header)
- 6-bit reserved (not used) ECN bits
- 6 bits of flags



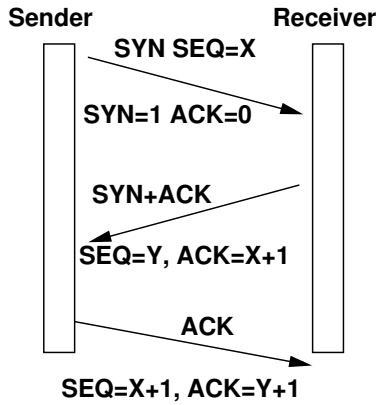
- U (URGent) – also the urgent pointer puts to urgent byte
- ACK (acknowledge) – 1 if ack field valid, otherwise ack field ignored
- PSH – receiver should process the data immediately and not buffer it waiting for more to come in
- RST (reset) – reset a connection because something has gone wrong
- SYN (synchronize) – used to establish connection CONNECTION REQUEST (SYN=1,ACK=0) and CONNECTION ACCEPTED (SYN=1,ACK=1)



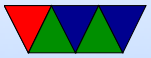
  - FIN – used to release a connection
- 16-bit window size – Only in ACK, says how many bytes to send back. This can be 0, which means I received everything but I am busy and can't take any more right now (can send another ACK with same number and nonzero window to restart)
- 16-bit checksum – similar to UDP also with pseudo header
- 16-bit urgent pointer
- options (32-bit words) – we'll discuss these later
- data



TCP Opening Connection



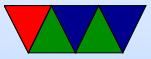
- Three-way handshake (Tomlinson 1975)
  - Server does LISTEN/ACCEPT to wait for connection.
  - Client issues CONNECT: destination/port/size, etc.
  - CONNECT chooses random initial sequence number (ISN) X



Sends SYN(SEQ=X) (SYN=1 ACK=0) with port and sequence number

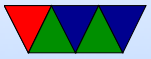
- Server receives packet. Checks if listening on that port; if not send back a packet with RST to reject.
- Otherwise it can accept
  sends back ACK(X+1) plus SYN(SEQ=Y) with sequence of own
- Client then responds with the server SYN ACK(Y+1) SEQ=x+1
- Connection is established

- SYN number picked, not to be 0. Originally clock based



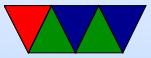
(random these days?). If machine reboots should wait for maximum lifetime to make sure all close

- Why do this? What happens with simultaneous connection?



TCP Closing Connection

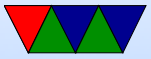
- Closing connection
- Although full duplex, almost like two independent one-way connections, released independently
  - one side sends packet with FIN
  - other side sends ACK of FIN, that direction is shut down
  - other direction can keep sending data though
  - at some point other side sends FIN
  - this is ACKed



– Two army problem?

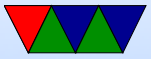
  Two generals on opposite side trying to co-ordinate attack. Any message can be intercepted by enemy. So say "attack at 9pm" but that could be lost. Could require other side to send reply, but that could be lost. You need infinite messages to guarantee it got through.

  If FIN not ACKed within two packet lifetimes, will close anyway. The other side eventually notices and closes too.



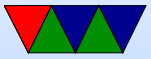
TCP State Machine

- 11 possible states
  - starts in CLOSED
  - LISTEN – waiting for a connection
  - SYN-SENT – started open, waiting for a returning SYN
  - SYN-RECEIVED – waiting for ACK
  - ESTABLISHED – open, two-way communication can happen
  - FIN-WAIT-1 – application has said it's finished



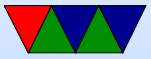
- FIN-WAIT-2 – the other side agreed to release
- CLOSE-WAIT – waiting for a termination request
- CLOSING – waiting for an ACK of closing request both sides closed at once
- LAST-ACK – waiting for ACK from last closing
- TIME-WAIT – waiting to transition to CLOSED long enough to ensure other side gets last ACK

- large state diagram

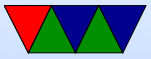


Typical Connection seen from Client

- 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

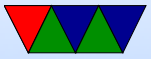


- FIN-WAIT-2
  one direction closed
  received ACK of FIN, wait for FIN from other side, respond with ACK
- TIME-WAIT
  wait until timeout to ensure all packets done in case ACK got lost
- CLOSED

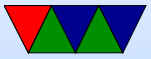


Typical Connection seen from Server

- 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

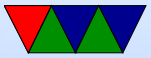


- CLOSE-WAIT
  , closes itself, sends FIN
- LAST-ACK
  gets ACK
- CLOSED



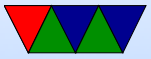
TCP Reliability

- Per-segment error control
  - checksum, Same as UDP.
  - 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
- Per-flow reliability
  - What to do in face of lost packets? Need to notice



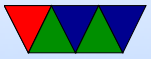
and retransmit and handle out-of-order

- 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"
- Selective ACK – has received segment indicated by



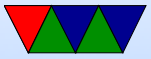
ACK

– Cumulative ACK – all previous data previous to the ACK has been received



Receiver Window

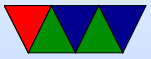
- Receiver Window (RWND)
- example
  - Receiver has 4k buffer
  - Sender does 2k write (2k/SEQ=0)
  - Receiver sends back ACK=2k, WIN=2048 (can take up to 2k)
  - Application sends 2k (2k, SEQ=2k)
  - If it is full, receiver might send ACK=4k, WIN=0
  - Later once buffer clears up a bit (application reads 2k



maybe) sends ACK=4096, WIN=2k

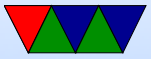
- Sender then sends some more

- When waiting on a WIN=0 can send two things, URG to kill the connection, or a 1-byte packet to have retransmit window and next byte expected (in case the ack restarting was lost, otherwise deadlock)



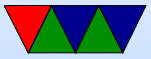
Window Management / Flow Control

- Senders do not have to transmit incoming data immediately
- Receivers do not have to ACK immediately
- Can have a lot of overhead to send 40byte packet for one byte payload
- Senders can buffer data, for example if know window is 4k can wait until they have 4k. Can help performance.
- For example, typing at keyboard on telnet/ssh might send when an editor. Press key, send a packet. Get



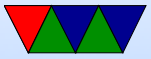
ACK. Then when read, another ACK updating window size. Then finally draw char on screen, send packet with that. 4 packets for one keypress

- One way to help is avoid window updates for up to 500ms in hope they can tag along with a real outgoing packet



Silly Window Syndrome

- Worst case, read one byte at a time, and then huge packet overhead for just one byte. Can be sender or receiver.
- Solution on sender end Nagel's algorithm – when data coming in one byte at time, send first then buffer rest until the first byte acknowledged. Also take into account window size. Widely used, can be bad for things like X window forwarding as mouse movements bunched together. TCP_NODELAY option disables.



- Solution on receiver end Clark's solution – application reading out the bytes one at a time. Send window updates each time, other side resend one byte, send message window full, etc. Solution (Clark) to wait until buffer is the original max segment size, or half empty
- Another – Delayed ACK
  delay the ACK so the other side has time to queue up data
  Can't delay by more than 500ms though