Don’t panic: TCP gearup III
Overview

- Final TCP stuff
- Any questions you have
Milestone I

- Start of your API and TCP stack
- Listen and establish connections => create sockets/TCB
- TCP handshake
- accept, connect, and start of ls REPL commands
Milestone II

- Basic sending and receiving using your sliding window/send receive buffers
- Plan for the remaining features
Roadmap

Final deadline

• Retransmissions (+ computing RTO from RTT)
• Out-of-order packets
• Sending and receiving files (sf, rf)
• Zero-window probing
• Connection teardown (CL)
Sendfile/Recvfile

Using your socket API, send/recv a file

Sendfile
• Open a file, VConnect, call VWrite in a loop

Recvfile
• Listen on a port, Open a file, call VRead in a loop

=> This is the ultimate test: your implementation should be similar to how you’d use a real socket API!
So how do we get there?
Relevant materials

- Lecture 15 (10/26): Sliding window, retransmissions, zero window probing
- Lecture 16 (10/31): connection teardown

- Testing and tools stuff: “TCP getting started” in docs
  => New IP reference for testing with packet loss => announcement soon
Retransmissions

Usually, make a “retransmission queue”

• When segment sent, add segment to queue with some metadata

  => What to store? You decide!

\[\text{ld when you sent it.}\]
Retransmissions

Usually, make a “retransmission queue”

- When segment sent, add segment to queue with some metadata
  
  => What to store? You decide!

- Start RTO timer

- When you get an ACK, reset

More info: Lecture 15, RFC6298
Retransmissions

Usually, make a “retransmission queue”
• When segment sent, add segment to queue with some metadata
  => What to store? You decide!
• Start RTO timer, reset on ACK

When RTO timer expires
• Retransmit earliest unACK’d segment
• \( RTO = 2 \times RTO \) (up to max)
• If no data after \( N \) retransmits => give up, terminate connection

⇒ RFC6298 is your friend! Use it! (edge cases, etc.)
RTO?

RTO = Retransmission Timeout (RTO)

=> Based on expected RTT: “how long until you SHOULD get an ACK?”

When you get an ACK, update RTO

\[ \text{RTT} = \text{ONE MEASUREMENT} \]

\[ \text{SRTT} = \text{SMOOTHED RTT} \]

=> WEIGHTED AVG

Example upper/lower bounds
RTOmin \( \sim \) 100ms
RTOmax \( \sim \) 5sec

More info: Lecture 15, RFC6298
RTO?

RTO = Retransmission Timeout (RTO)

=> Based on expected RTT: “how long until you SHOULD get an ACK?”

When you get an ACK, update RTO

=> Smoothed weighted moving average of recent RTTs

\[ \text{RTT} = \alpha (\text{RTT}_{\text{new}}) + (1-\alpha) \text{SRTT} \]

Example upper/lower bounds
RTOmin ~= 100ms
RTOmax ~= 5sec

More info: Lecture 15, RFC6298
Computing RTO

Strategy: measure expected RTT based on ACKs received

Use exponentially weighted moving average (EWMA)
- RFC793 version (“smoothed RTT”):
  \[
  \text{SRTT} = (\alpha \times \text{SRTT}_{\text{last}}) + (1 - \alpha) \times \text{RTT}_{\text{measured}} \\
  \text{RTO} = \max(\text{RTO}_{\text{min}}, \min(\beta \times \text{SRTT}, \text{RTO}_{\text{max}}))
  \]

\(\alpha\) = “Smoothing factor”: .8-.9
\(\beta\) = “Delay variance factor”: 1.3—2.0
\(\text{RTO}_{\text{min}}\) = 1 second

RFC793, Sec 3.7
RFC6298 (slightly more complicated, also measures variance)
UPDATE on perf requirement

**Performance requirement:** send/recv process MUST be event driven
- No busy-waiting
- `time.Sleep` MUST NOT BLOCK SEND/RECV process

*Okay to use sleep, `time.Ticker` to have separate thread trigger an event, like retransmissions*

Where does this apply?
- REPL: s, r, sf, rf
- VRead/VWrite
- Deciding when to send, or check for new data
- Retransmissions

=> Channels, condition variables, etc. are your friends
Out of order segments

Usually, make a “early arrival queue”
• When segment arrives, add to queue if it’s not the next segment
  => What to store? You decide!
• As more segments arrive, check the top of the queue to see if it fills in any gaps
Zero window probing (ZWP)

When receiver’s window is full, sender enters zero window probing mode:

• Stop sending segments
• At a periodic intervals, send 1 byte segments until receiver sends back window > 0 bytes

/ BYTE OF REAL DATA, WHATEVER IS NEXT IN SEND BUF. /
The next page has an example for zero window probing and retransmissions—it’s a bit more involved than we discussed in the gearup but should be useful for seeing how it works and interacts with your buffers.
ASSUME: SEGMENT SIZE = 1

SYN
SYN+ACK
ACK

① SEQ = 1 ACK = 1 "H"

② SEQ = 2 ACK = 1 "E"

③ Seq = 1 ACK = 1 WIN = Y

④ Seq = 3 ACK = 1 "L"

\[\text{BYTES IN FLIGHT = WINDOW SIZE} \Rightarrow \text{SENDING MUST STOP!}\]

⑤ Seq = 1 ACK = 1 WIN = Y

⑥ Seq = 4 ACK = 1 "L"

⑦ Seq = 1 ACK = 1 WIN = Y

⑧ Seq = 1 ACK = 1 "H"

\[\text{TIMEOUT!} \Rightarrow \text{CAN NOW ACK ALL DATA, BUT BUFFER IS FULL!}\]

⑨ Seq = 1 ACK = 5 WIN = 0

\[\text{ZWP EXAMPLE}\]

\[\text{RCU BUF EMPTY} \Rightarrow \text{NO DATA READ BY APP YET}\]

\[\text{AT START}\]

\[\text{NXT} \Rightarrow \text{SUCCESSFUL TRANSFER}\]

\[\text{NXT} \Rightarrow \text{SUCCESSFUL TRANSFER}\]

\[\text{WIN}\]

\[\text{WIN}\]

\[\text{WIN}\]

\[\text{WIN}\]
*NOTE: ZERO WINDOW PROBES ARE ALWAYS ONE BYTE REGARDLESS OF THE SEGMENT SIZE. IN THIS EXAMPLE, WE HAVE BEEN USING 1-BYTE SEGMENTS THROUGHOUT—THIS IS A COINCIDENCE!
Zero window probing

When receiver’s window is full, sender enters zero window probing mode
• Stop sending segments
• At a periodic intervals, send 1 byte segments until receiver sends back window > 0 bytes

How to test?
• On one side, listen on a port: a 9999
• On other side, send a file
Connection teardown

4-way connection close process => see the lecture for details

• VClose just starts the connection close process
  => TCB not deleted until connection goes to CLOSED state
Testing with packet loss

New REPL command in vrouter reference (out soon):

```plaintext
> drop 0.01    // Drop 1% of packets
> drop 0.5     // Drop 50% of packets (way too aggressive)
> drop 1       // Drop ALL packets (equivalent to “down”)
> drop 0       // Drop no packets
```

Also: can set by running vrouter with `--drop`
Custom vnet_run configurations
How to test TCP

Useful wireshark mechanics
- SEQ/ACK analysis
- Follow TCP stream
- Validating the checksum

Note: watching traffic in wireshark works differently in this project!
=> See Gearup II, “TCP getting started” guide for details
Reference implementation

• Our implementation of TCP
• Try it and compare with your version!

Note: we’re using a new reference this year (after 8+ years!)
• We’ve tested as best we can, but there may be bugs
• See Ed FAQ, docs FAQ for list of known bugs
• Let us know if you have issues!

⇒ If the spec disagrees with the reference implementation, the spec wins—don’t propagate buggy behavior (please help us find any discrepancies!)
Closing thoughts

Do not underestimate these last parts--it will take time to debug and test them.

When stuck, take a break and come back to it. It will help.

=> Do NOT wait until the last minute.

Don’t panic.
Breathe

i am a tiny cactus
and i believe
in you

you can do the thing
The TCP checksum

… is pretty weird

Computing the TCP checksum involves making a "pesudo-header" from TCP header + IP header fields:

⇒ See the TCP-in-IP example for a demo of how to compute/verify it
Where to get more info

TCP REPL Commands
- send (s), recv (r), send file (sf), receive file (rf)

Our docs: "REPL commands" spec

Our docs: "Socket API" example

Guidelines: "TCP notes" in our docs
- Links to relevant RFCs (e.g., RFC9293)
- Our modifications/notes on the RFCs

- TCP-in-IP example (how to make/parse packets)
- IP docs

Sockets API

TCP Stack

IP stack
Unfilled buffer

Data received, but not acknowledged

Data received, acknowledged, but not yet delivered to application

Data received, acknowledged and delivered to application

Sequence numbers (Circumference = 0 to $2^{32}$ slots)

Initial sequence number

Receiver's window (Allocation buffer) Up to $2^{16} - 1$ slots

Window shifts

Receiver's window (Allocation buffer)

Initial sequence number

Receiver's window (Allocation buffer)
CLOSED

LISTEN

SYN RECEIVED

SYN/SYN+ACK

SYN SENT

RST/
SYN/SYN+ACK
(simultaneous open)

Data exchange occurs

CLOSE/FIN

FIN/Ack

FIN WAIT 1

FIN/Ack

CLOSING

FIN+ACK/ACK

FIN WAIT 2

FIN/Ack

TIME WAIT

FIN/Ack

CLOSE WAIT

LAST ACK

ACTIVE CLOSE

TIMEOUT

PASSIVE CLOSE

FIN/ACK

ACTIVE CLOSE

TIMEOUT

PASSIVE CLOSE

FIN/ACK

CLOSE/FIN

TIME OUT

CLOSED