Don't panic: TCP gearup III

Overview

- Final TCP stuff
- Any questions you have

Roadmap

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

Roadmap

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

- Up to 1MB

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!

Demo!

A common thing to notice when you start sf/rf, sometimes you start seeing bugs from IP

=> Run reference with YOUR router, OUR HOST

=> Could help you root out a problem at the interface level

So how do we get there?

Relevant materials

- Lecture 15 (10/24): Sliding window, retransmissions, zero window probing
- Lecture 16 (10/29): connection teardown

- Testing and tools stuff: "Getting started" in TCP docs
  => Can configure reference to drop packets
  => Some more testing notes soon (mostly mirroring what's here)

Retransmissions

More info: Lecture 15, RFC6298

Usually, make a "retransmission queue"

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

  => What to store? You decide!

  ↳ WHEN YOU SENT IT.

Retransmissions

More info: Lecture 15, RFC6298

Usually, make a "retransmission queue"

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

  => What to store? You decide!
- Start RTO timer => ONE TIMER PER SOCKET.
- When you get an ACK, reset

Retransmissions

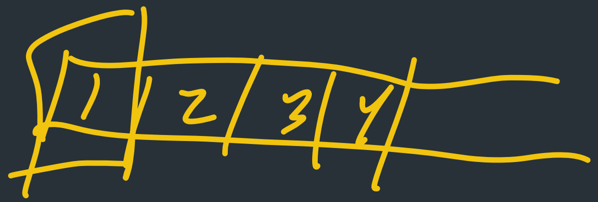
More info: Lecture 15, RFC6298

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 * RTO (up to max)
- If no data after N retransmits => give up, terminate connection

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

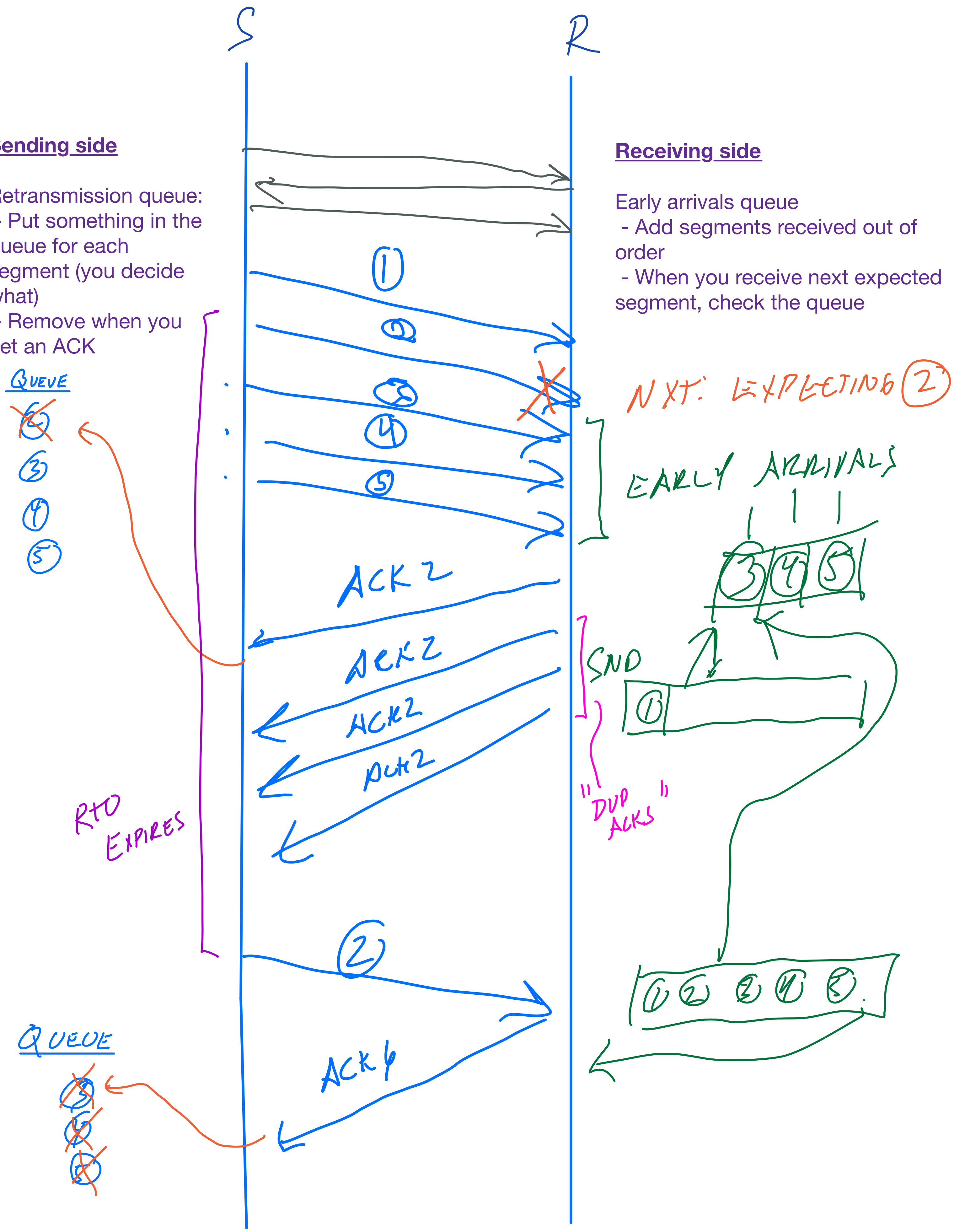
Scratch notes for retransmissions/early arrivals: see recording for a live drawing, lecture 15 for more

Sending side

Retransmission queue:
- Put something in the queue for each segment (you decide what)
- Remove when you get an ACK

Receiving side

Early arrivals queue
- Add segments received out of order
- When you receive next expected segment, check the queue



RTO?

More info: Lecture 15, RFC6298

WHEN TO START/STOP.

RTO = Retransmission Timeout (RTO)

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

When you get an ACK, update RTO

RTT = ONE MEASUREMENT

SRTT = SMOOTHED RTT

=> WEIGHTED AVG

Example upper/lower bounds

RTOmin ~= 100ms

RTOmax ~= 5sec

RTO?

More info: Lecture 15, RFC6298

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

$$RTT \leftarrow \alpha (RTT_{NEW}) + (1-\alpha) SRTT$$

α β

Example upper/lower bounds

RTOmin ~= 100ms

RTOmax ~= 5sec

Computing RTO

Strategy: *measure* expected RTT based on ACKs received

Use exponentially weighted moving average (EWMA)

- RFC793 version ("smoothed RTT"):

$$SRTT = (\alpha * SRTT_{Last}) + (1 - \alpha)* RTT_{Measured}$$

$$RTO = max(RTO_{Min}, min(\boldsymbol{\beta} * SRTT, RTO_{Max}))$$

α = "Smoothing factor": .8-.9

$\boldsymbol{\beta}$ = "Delay variance factor": 1.3—2.0

RTO_{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

=> Channels, condition variables, etc. are your friends

Out of order segments

More info: Lecture 15

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

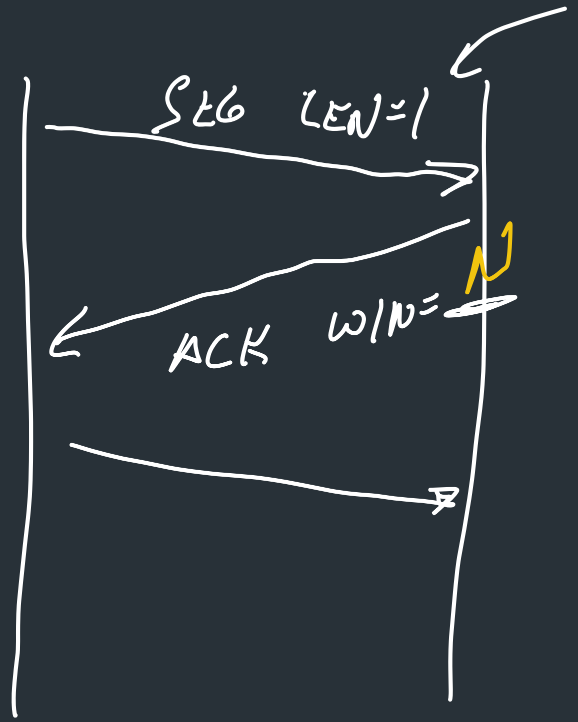
Zero window probing (ZWP)

More info: Lecture 15

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

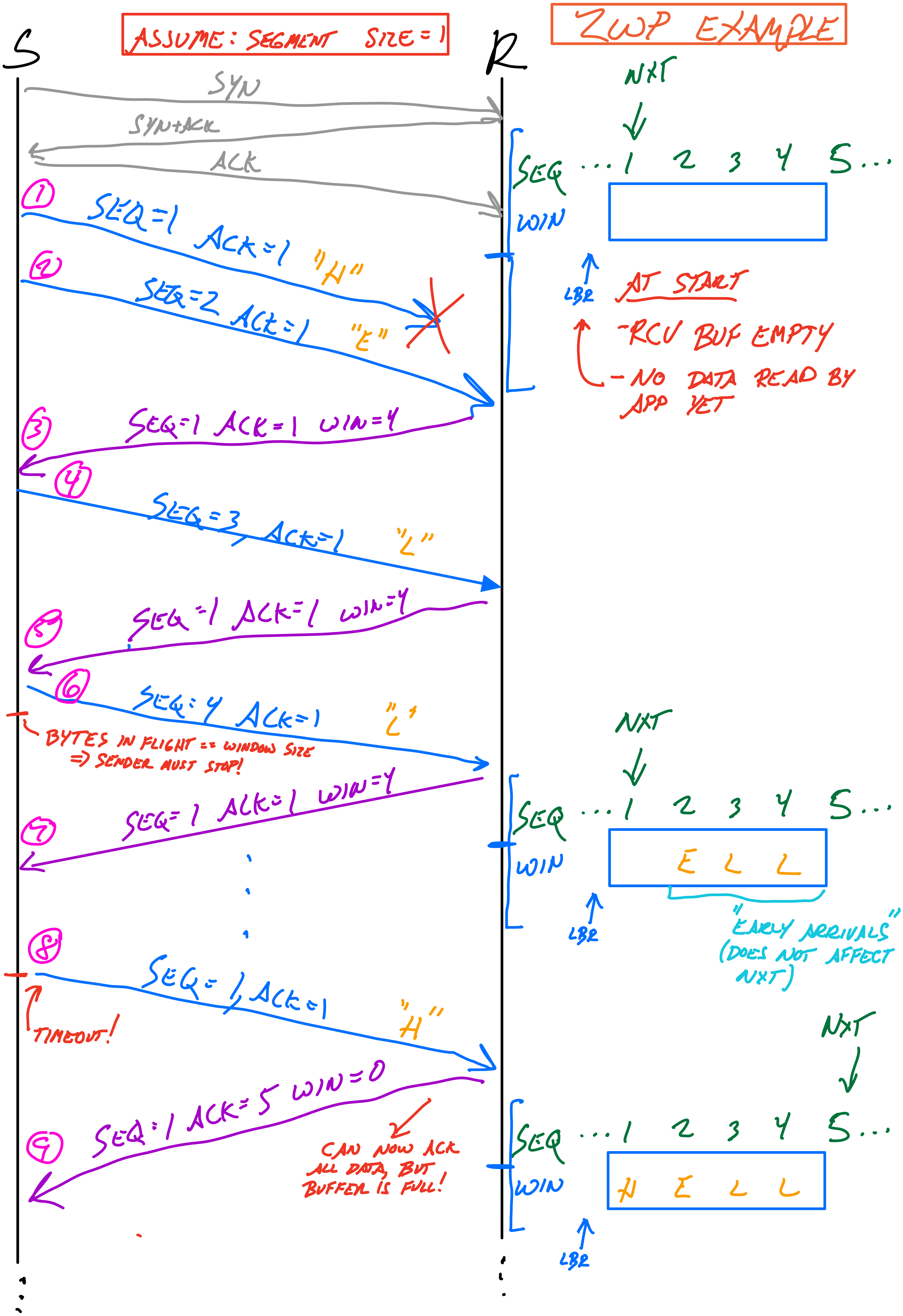
- Stop sending segments
- At a periodic intervals, send 1 byte segments until receiver sends back window > 0 bytes

Send 1 byte of real data (whatever is next in send buffer)

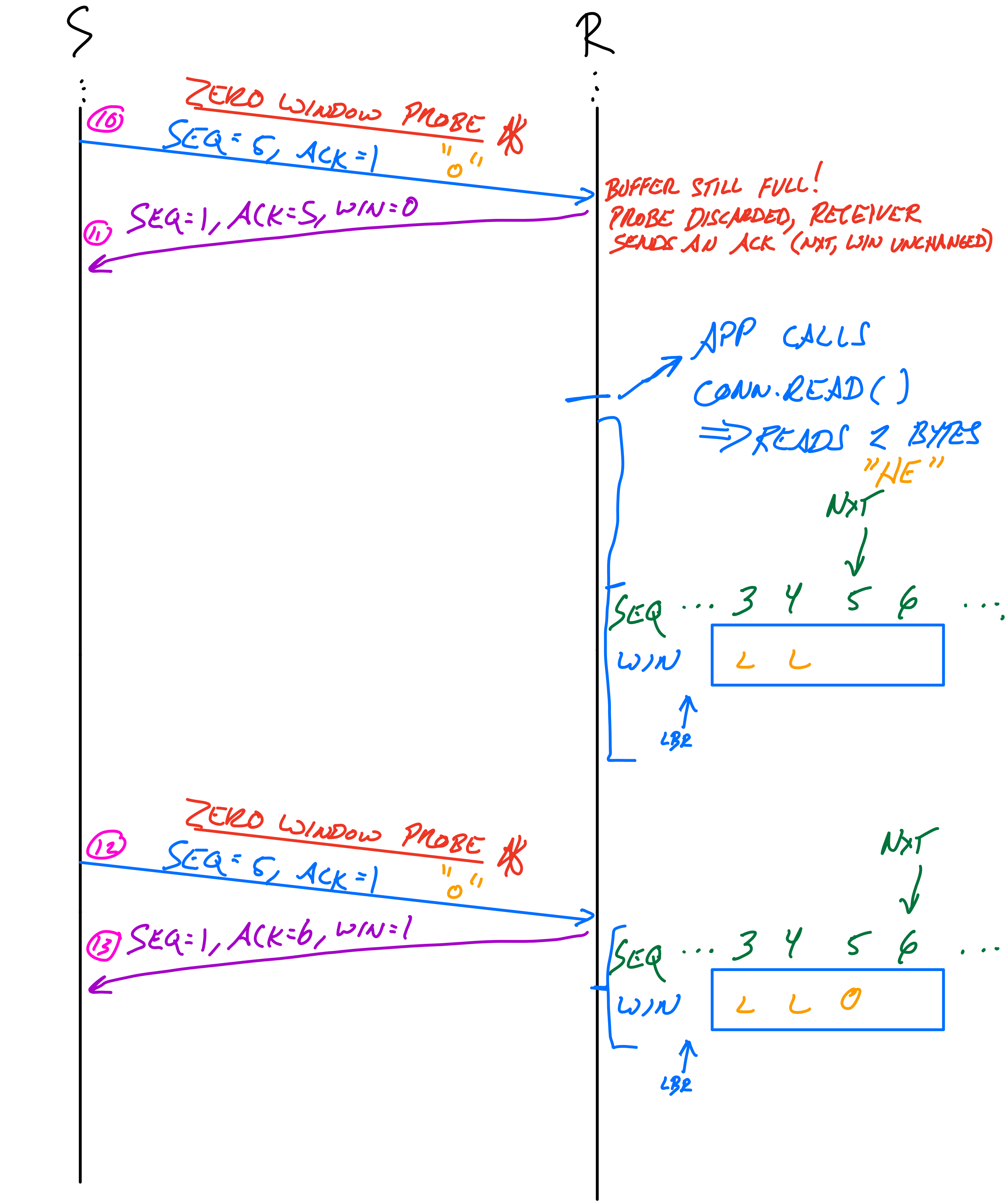


The next page has an example for zero window probing and retransmissions—it's a bit more involved than we discussed in the gearup but should be useful for seeing how it works and interacts with your buffers.

After that is an annotated example of how zero-window probing should look in wireshark



ZWP EXAMPLE
ASSUME: SEGMENT SIZE = 1
S
R
SYN
SYN+ACK
ACK
1
SEQ=1 ACK=1 "H"
2
SEQ=2 ACK=1 "E"
SEQ
WIN
NXT
... 1 2 3 4 5 ...
LBR
AT START
-RCV BUF EMPTY
-NO DATA READ BY APP YET
3
SEQ=1 ACK=1 WIN=4
4
SEQ=3 ACK=1 "L"
5
SEQ=1 ACK=1 WIN=4
6
SEQ=4 ACK=1 "L"
BYTES IN FLIGHT == WINDOW SIZE
=> SENDER MUST STOP!
7
SEQ=1 ACK=1 WIN=4
E L L
"EARLY ARRIVALS"
(DOES NOT AFFECT NXT)
8
SEQ=1 ACK=1 "H"
TIMEOUT!
9
SEQ=1 ACK=5 WIN=0
CAN NOW ACK ALL DATA, BUT BUFFER IS FULL!
H E L L



* 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 JUST A COINCIDENCE!

Zero window probing

More info: Lecture 15

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

Custom vnet_run configurations

Zero-window probing: in wireshark

(Try it with the reference!)

Part 1: Window fills up, start sending probes

Pkts 127-130: Sending segments as normal
=> eventually fills up window

Pkt 131: Receiver ACKs with WIN=0
=> sender still has data to send, so it must start probing

127	0.002…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=61777 Win=3759 Len=0
128	0.002…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=62465 Win=3071 Len=0
129	0.002…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=63825 Win=1711 Len=0
130	0.002…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=64513 Win=1023 Len=0
131	0.002…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=65536 Win=0 Len=0
132	1.003…	10.0.0.1	10.1.0.2	TCP	83	[TCP ZeroWindowProbe] 63582 → 9999 [ACK] Seq=65536 Ack=1 Win=65535 Len=1 [TCP segment of
133	1.003…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindowProbeAck] [TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=65536 Win=0 Len=0
134	2.003…	10.0.0.1	10.1.0.2	TCP	83	[TCP ZeroWindowProbe] 63582 → 9999 [ACK] Seq=65536 Ack=1 Win=65535 Len=1 [TCP segment of
135	2.004…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindowProbeAck] [TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=65536 Win=0 Len=0
136	3.004…	10.0.0.1	10.1.0.2	TCP	83	[TCP ZeroWindowProbe] 63582 → 9999 [ACK] Seq=65536 Ack=1 Win=65535 Len=1 [TCP segment of
137	3.004…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindowProbeAck] [TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=65536 Win=0 Len=0
138	4.004…	10.0.0.1	10.1.0.2	TCP	83	[TCP ZeroWindowProbe] 63582 → 9999 [ACK] Seq=65536 Ack=1 Win=65535 Len=1 [TCP segment of

Pkt 132-138: Sender periodically sends 1-byte probes => receiver ACKs with updated window
Things to note:
- Probe is contains the NEXT byte in the data stream (here, seq 65536). This is purposely outside the receiver's window!
- Probe has length 1
- Receiver ACK sends ACK, but can't accept the segment (ACK number doesn't change from 131, when window was full)

Part 2: Recovery: Eventually, receiver reads some data, freeing up window space

(in this example, h2 reads 4096 bytes)

Pkt 139 (ACK for probe packet 138): space is available, so ACK now has updated window size. Sender can resume sending now!

(Okay for wireshark to flag this as "ACKed unseen segment")

139	4.005…	10.1.0.2	10.0.0.1	TCP	82	[TCP ACKed unseen segment] 9999 → 63582 [ACK] Seq=1 Ack=65537 Win=4095 Len=0
140	4.005…	10.0.0.1	10.1.0.2	TCP	14…	63582 → 9999 [ACK] Seq=65536 Ack=1 Win=65535 Len=1360 [TCP segment of a reassembled F
141	4.005…	10.0.0.1	10.1.0.2	TCP	14…	63582 → 9999 [ACK] Seq=66896 Ack=1 Win=65535 Len=1360 [TCP segment of a reassembled F
142	4.005…	10.0.0.1	10.1.0.2	TCP	14…	63582 → 9999 [ACK] Seq=68256 Ack=1 Win=65535 Len=1360 [TCP segment of a reassembled F
143	4.005…	10.0.0.1	10.1.0.2	TCP	98	[TCP Window Full] 63582 → 9999 [ACK] Seq=69616 Ack=1 Win=65535 Len=16 [TCP segment of
144	4.005…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=66896 Win=2736 Len=0
145	4.005…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=68256 Win=1376 Len=0
146	4.005…	10.1.0.2	10.0.0.1	TCP	82	9999 → 63582 [ACK] Seq=1 Ack=69616 Win=16 Len=0
147	4.005…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=69632 Win=0 Len=0
148	5.006…	10.0.0.1	10.1.0.2	TCP	83	[TCP ZeroWindowProbe] 63582 → 9999 [ACK] Seq=69632 Ack=1 Win=65535 Len=1 [TCP segment
149	5.006…	10.1.0.2	10.0.0.1	TCP	82	[TCP ZeroWindowProbeAck] [TCP ZeroWindow] 9999 → 63582 [ACK] Seq=1 Ack=69632 Win=0 Le

Pkts 140-143: Sender resumes sending, eventually fills up window again

Note: In this version, sender resends the probe byte (seq 65536) as part of first segment. You're not required to emulate this--it's okay (and technically more efficient) to resume from seq 65537 instead. Nick will update the reference to fix this next year :)

Connection teardown

More info: Lecture 16

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):

```
> 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`

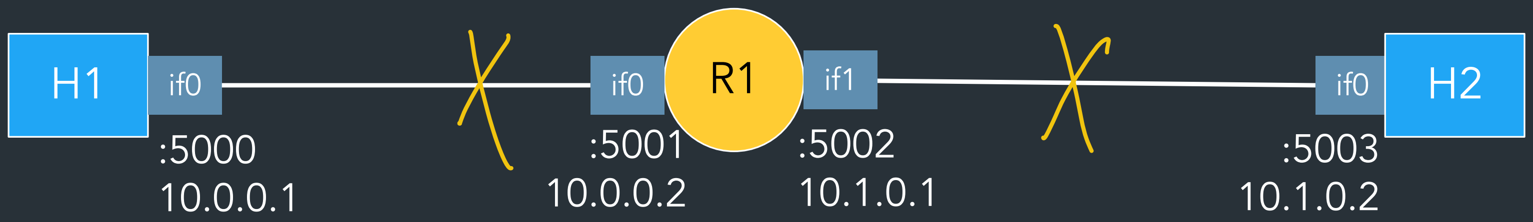
Custom vnet_run configurations

With ~30s of work, you can set up a config file for vnet_run to easily let you...
- Run custom configurations of vhost/vrouter (your h1, reference h2, etc.)
- Automatically configure drop rate at startup (save on typing!)
- Turn on logging

=> See recording for a demo (also "Custom vnet_run configurations" in Docs > "Tools and resources)

How to test TCP

More docs coming soon!



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!)

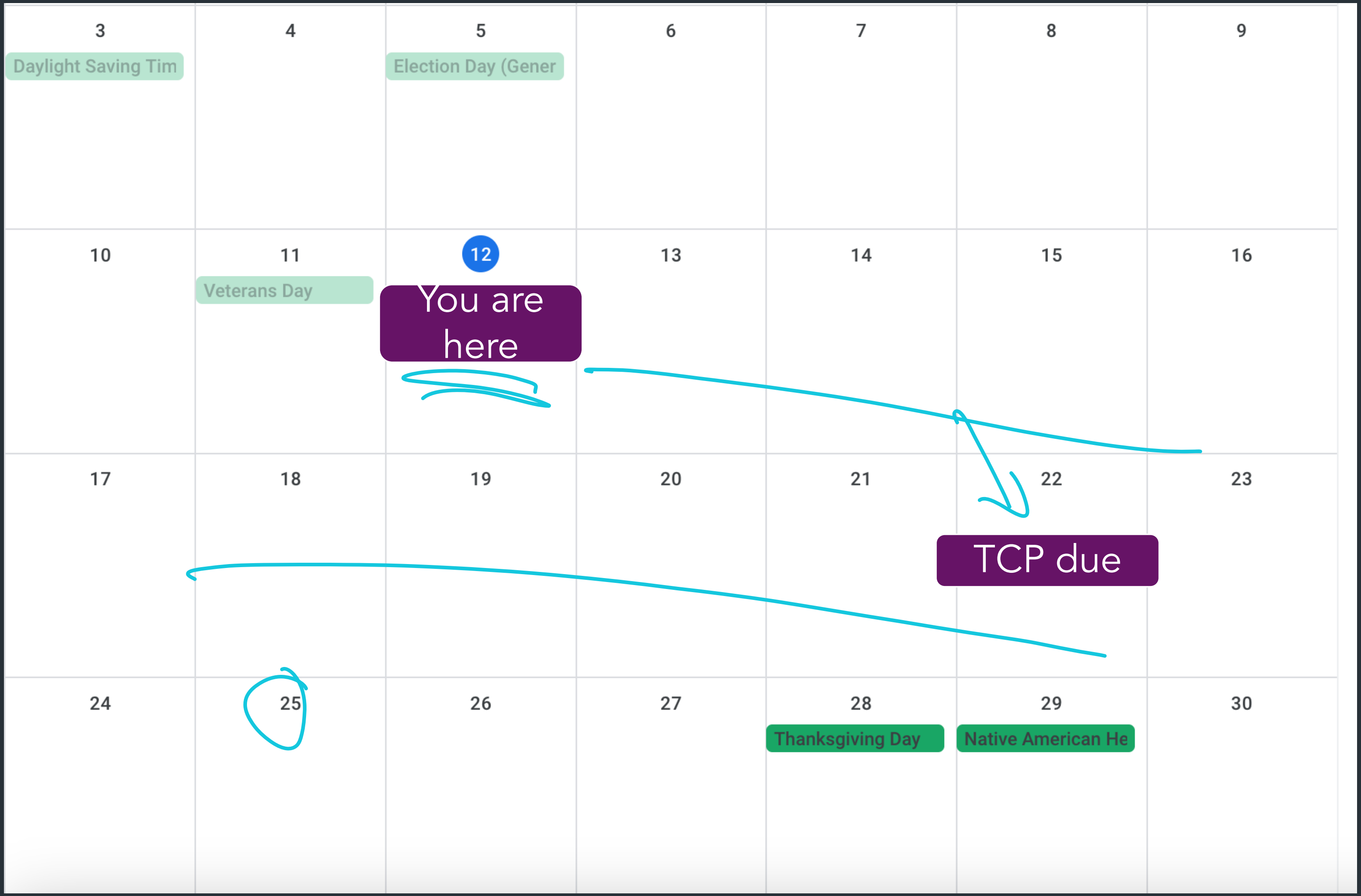
Closing thoughts

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

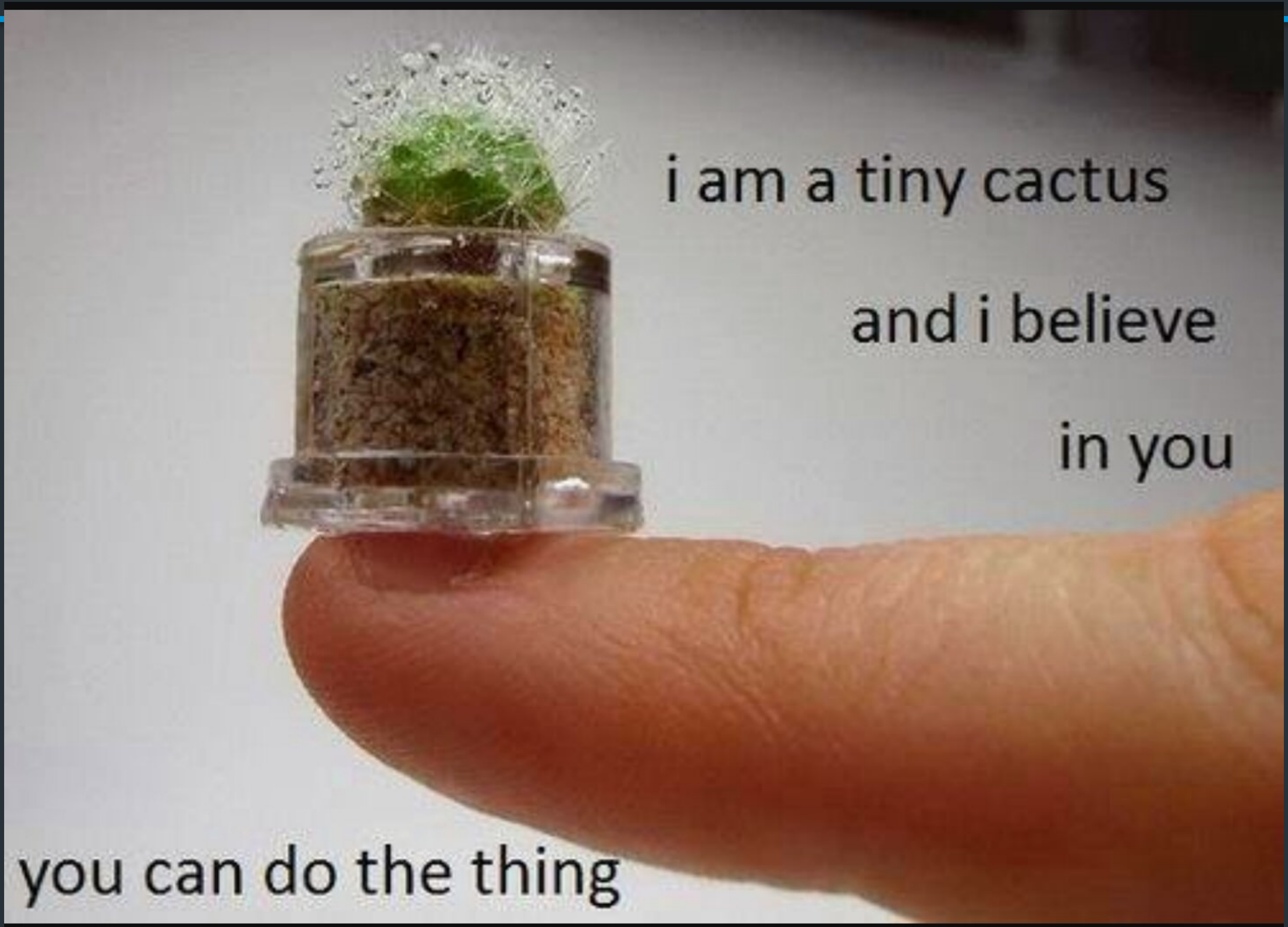
When stuck, take a break and come back to it. It will help.

=> Do NOT wait until the last minute.

Don't panic.

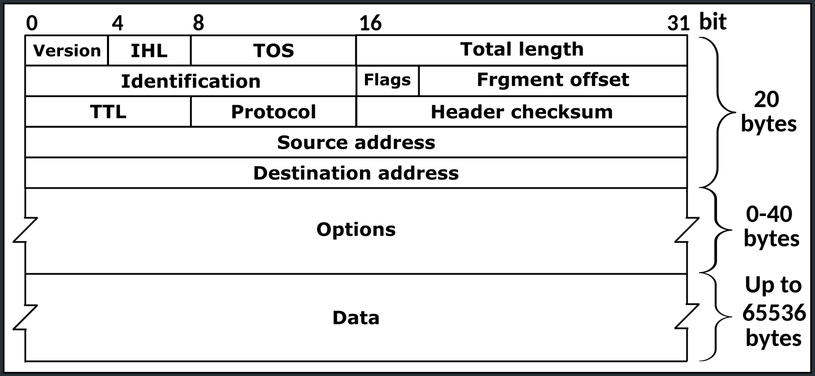

3
4
5
6
7
8
9
Daylight Saving Tim
Election Day (Gener
10
11
12
13
14
15
16
Veterans Day
You are here
17
18
19
20
21
22
23
TCP due
24
25
26
27
28
29
30
Thanksgiving Day
Native American He

Breathe



The TCP checksum

… is pretty weird



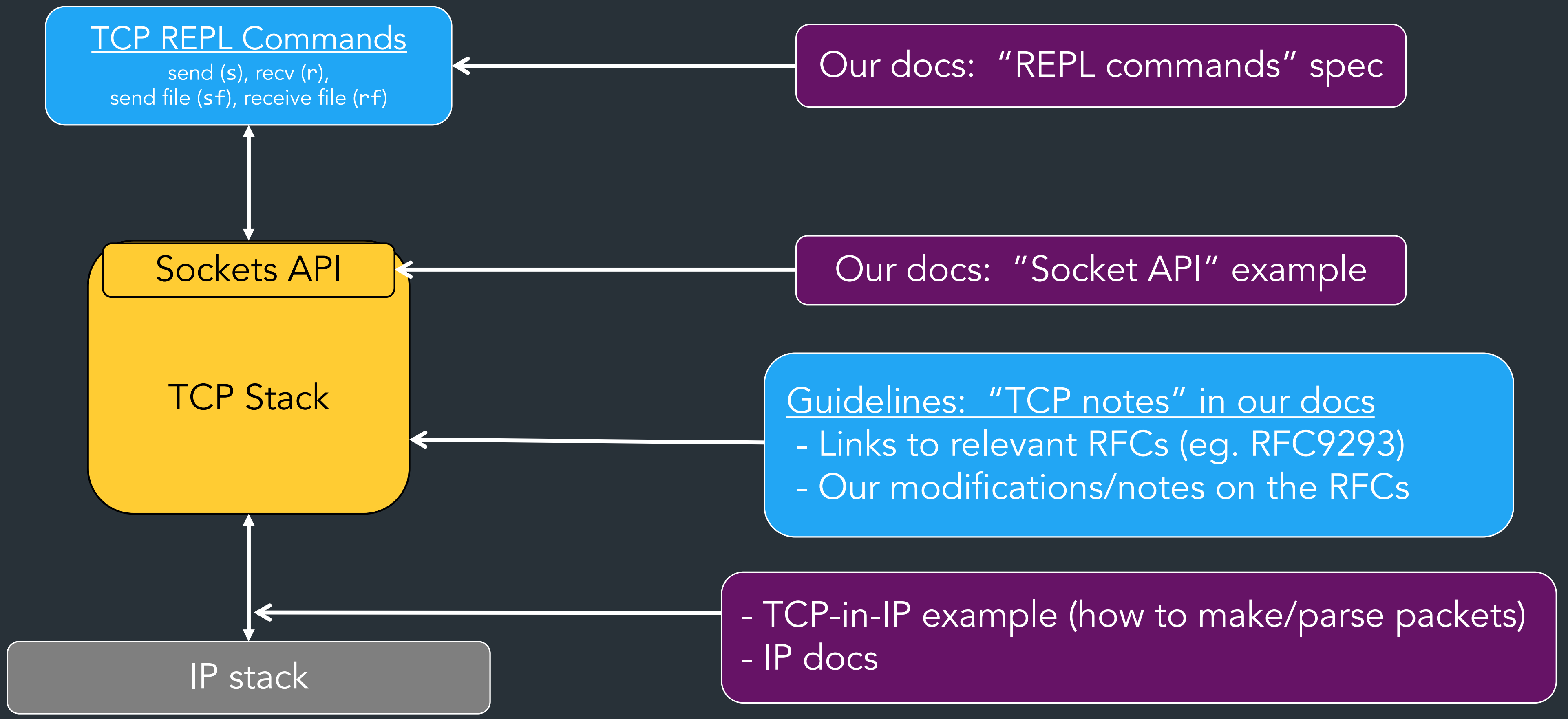
Computing the TCP checksum involves making a “pesudo-header” from TCP header + IP header fields:

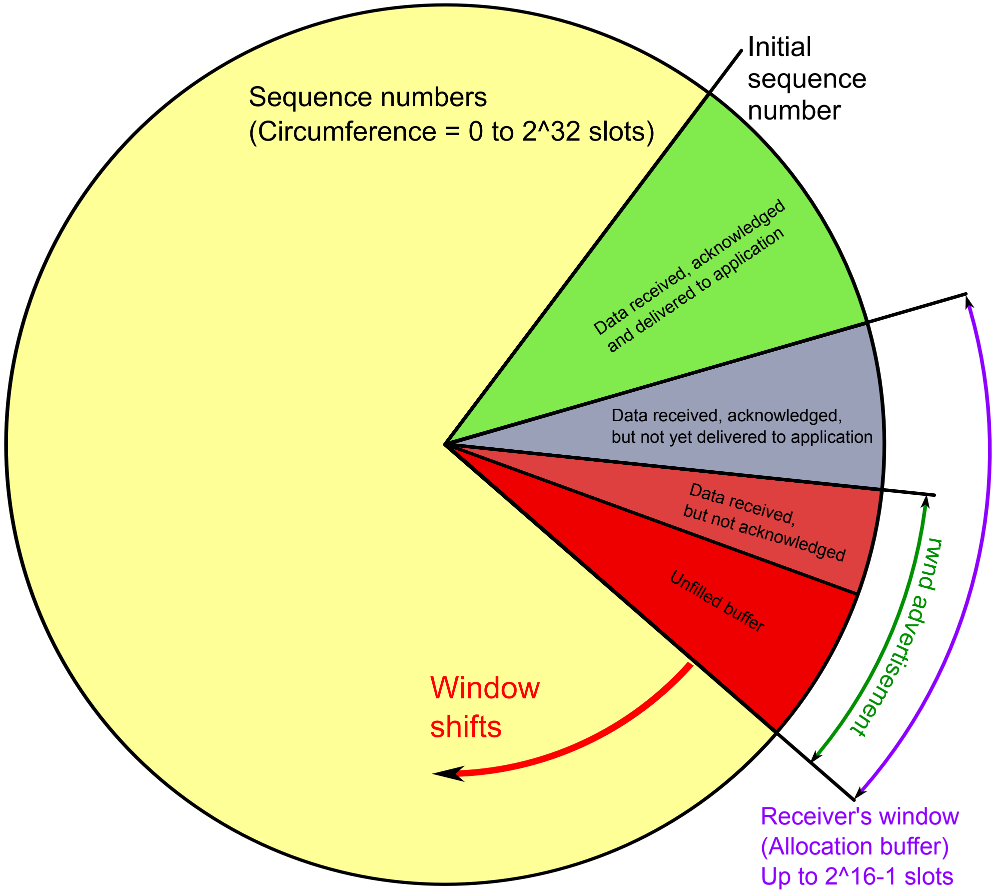
TCP pseudo-header for checksum computation (IPv4)

Bit offset	0–3	4–7	8–15	16–31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			
160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

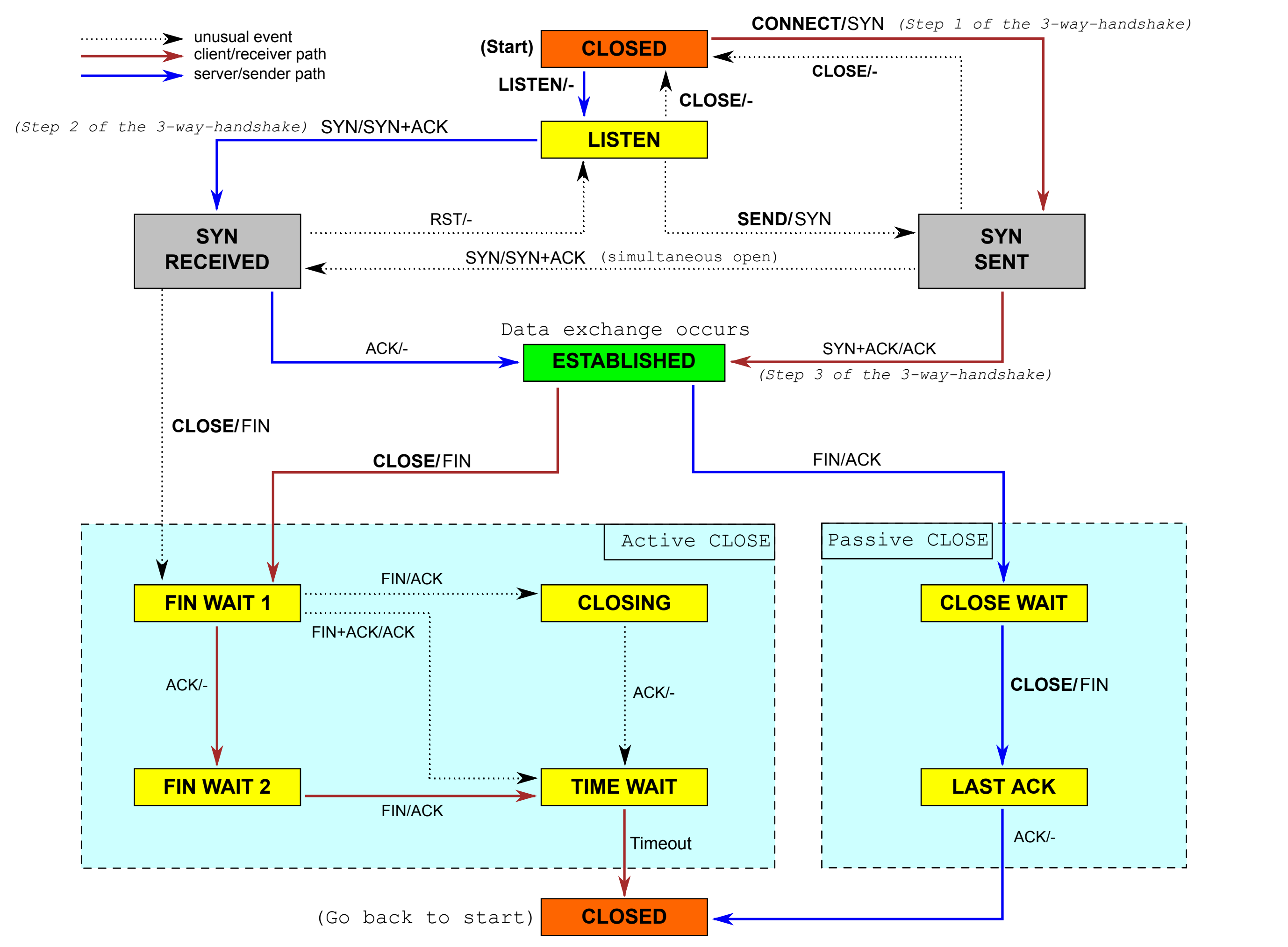
⇒ See the TCP-in-IP example for a demo of how to compute/verify it

Where to get more info





Sequence numbers
(Circumference = 0 to 2^32 slots)
Initial
sequence
number
Data received, acknowledged
and delivered to application
Data received, acknowledged,
but not yet delivered to application
Data received,
but not acknowledged
Unfilled buffer
rwnd advertisement
Window
shifts
Receiver's window
(Allocation buffer)
Up to 2^16-1 slots


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
LISTEN
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
SYN RECEIVED
RST/-
SEND/SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED