# 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

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!

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

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"

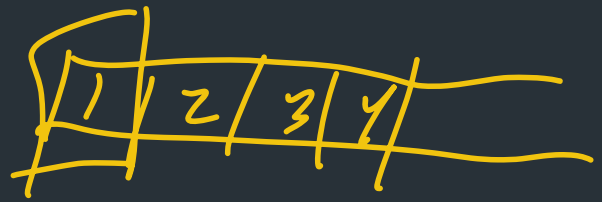
- When segment sent, add segment to queue with some metadata
  => What to store? You decide!
- Start RTO timer ⇒ ONE TIMER PER SOCKET.
- When you get an ACK, reset

# Retransmissions

More info: Lecture 15, RFC6298

Usually, make a "retransmission queue"

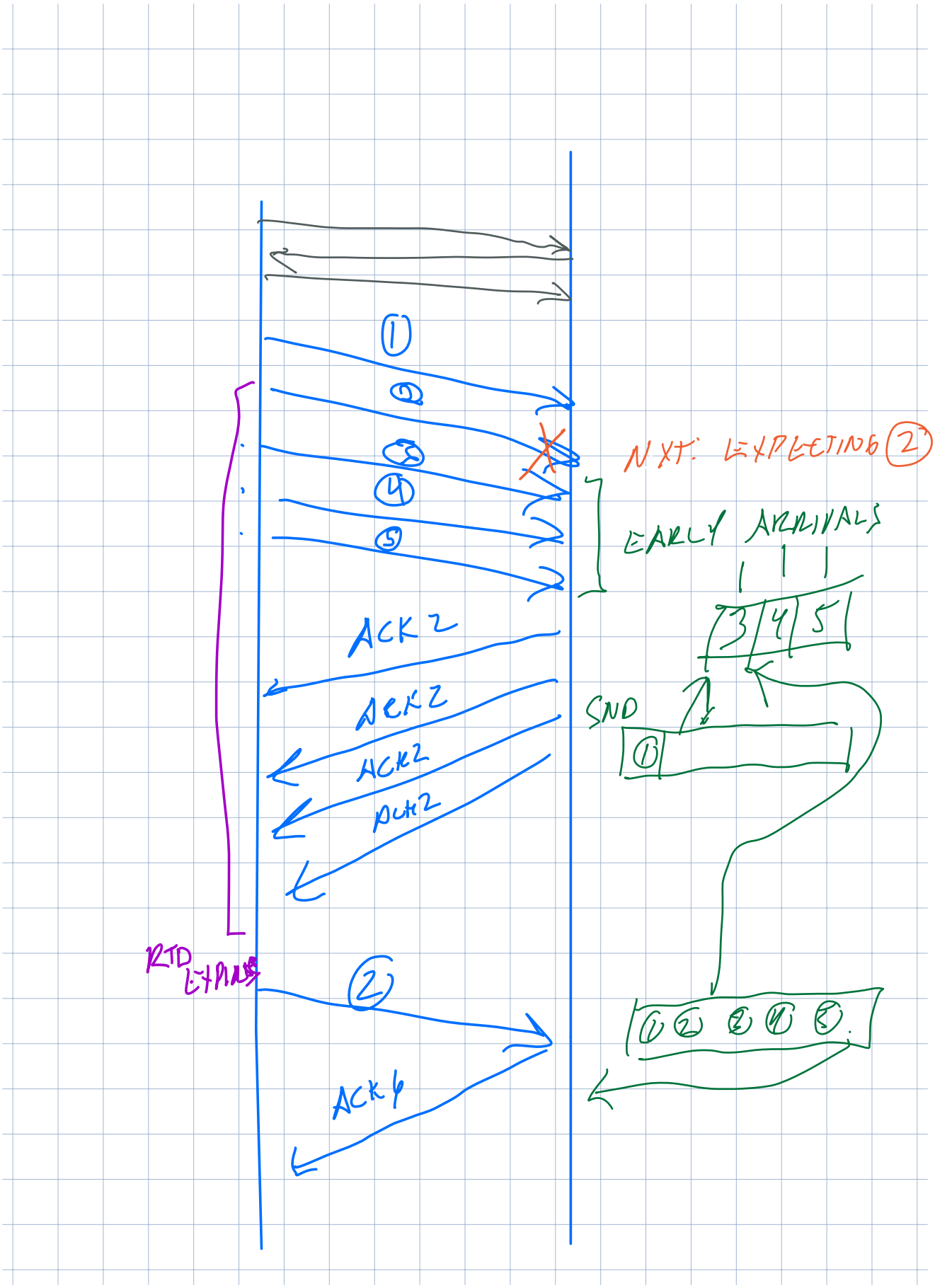
- When segment sent, add segment to queue with some metadata
  => What to store? You decide!
- Start RTO timer, reset on ACK




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.)


1
2
3
4
5
NXT. EXPECTING (2)
EARLY ARRIVALS
3 4 5
ACK 2
ACK 2
ACK 2
ACK 2
SND
1
RTD EXPIRES
2
1 2 3 4 5
ACK 6

# RTO?

More info: Lecture 15, RFC6298

RTO = Retransmission Timeout (RTO)

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

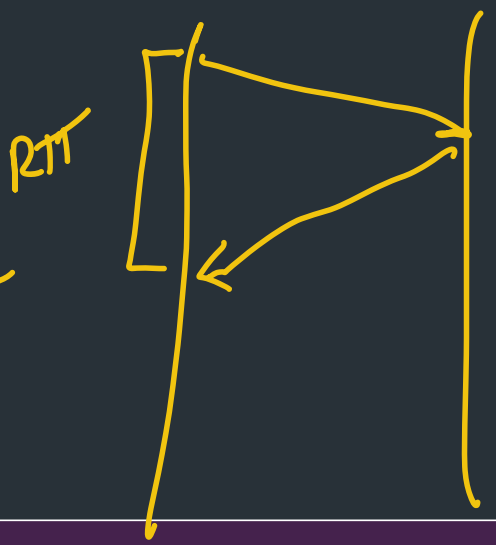
# WHEN TO START/STOP.

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

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

$\alpha$ $\beta$

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"):

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

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

$\alpha$ = "Smoothing factor": .8-.9

$\beta$ = "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
- Retransmissions

=> 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)

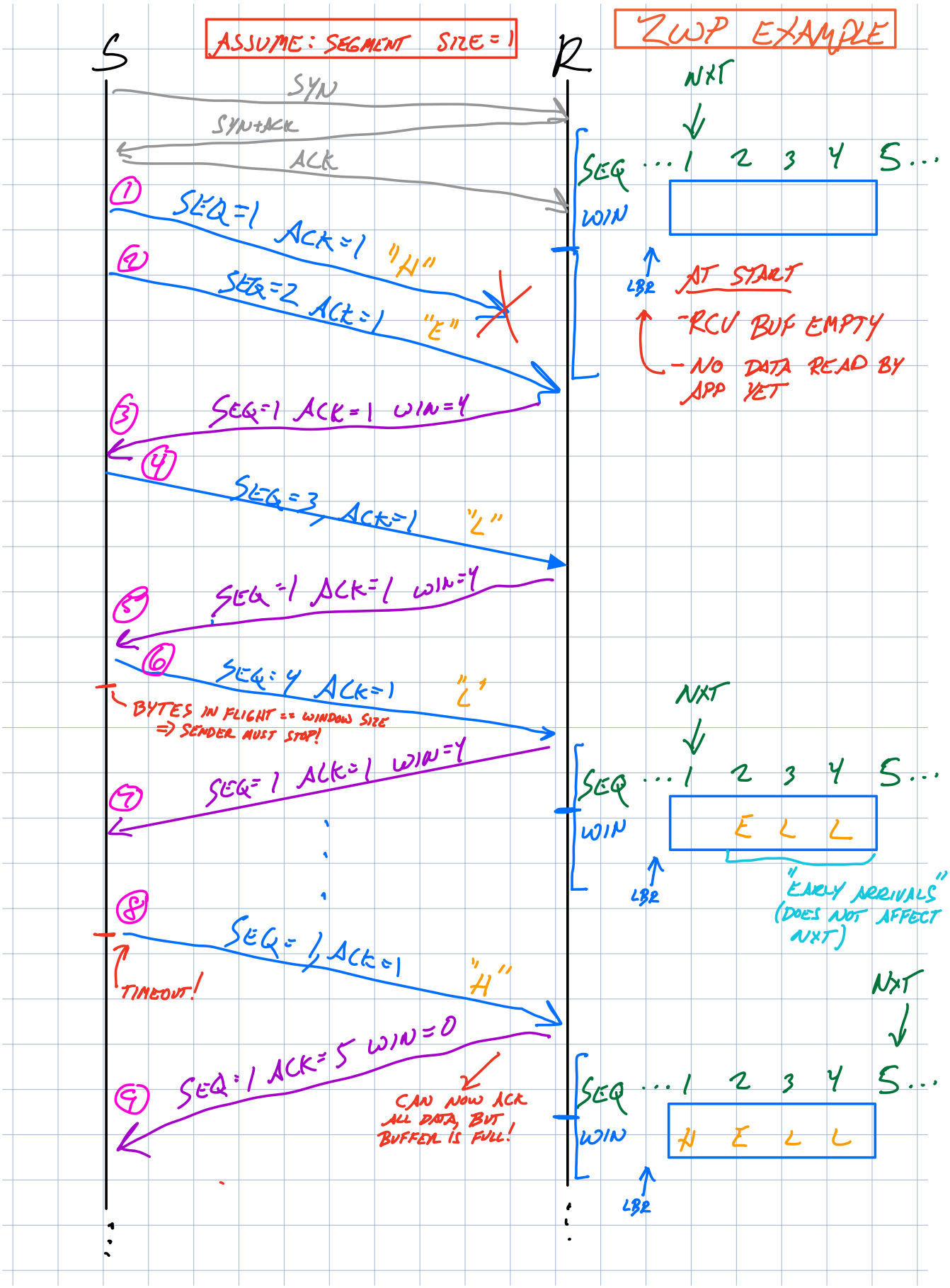
More info: Lecture 15

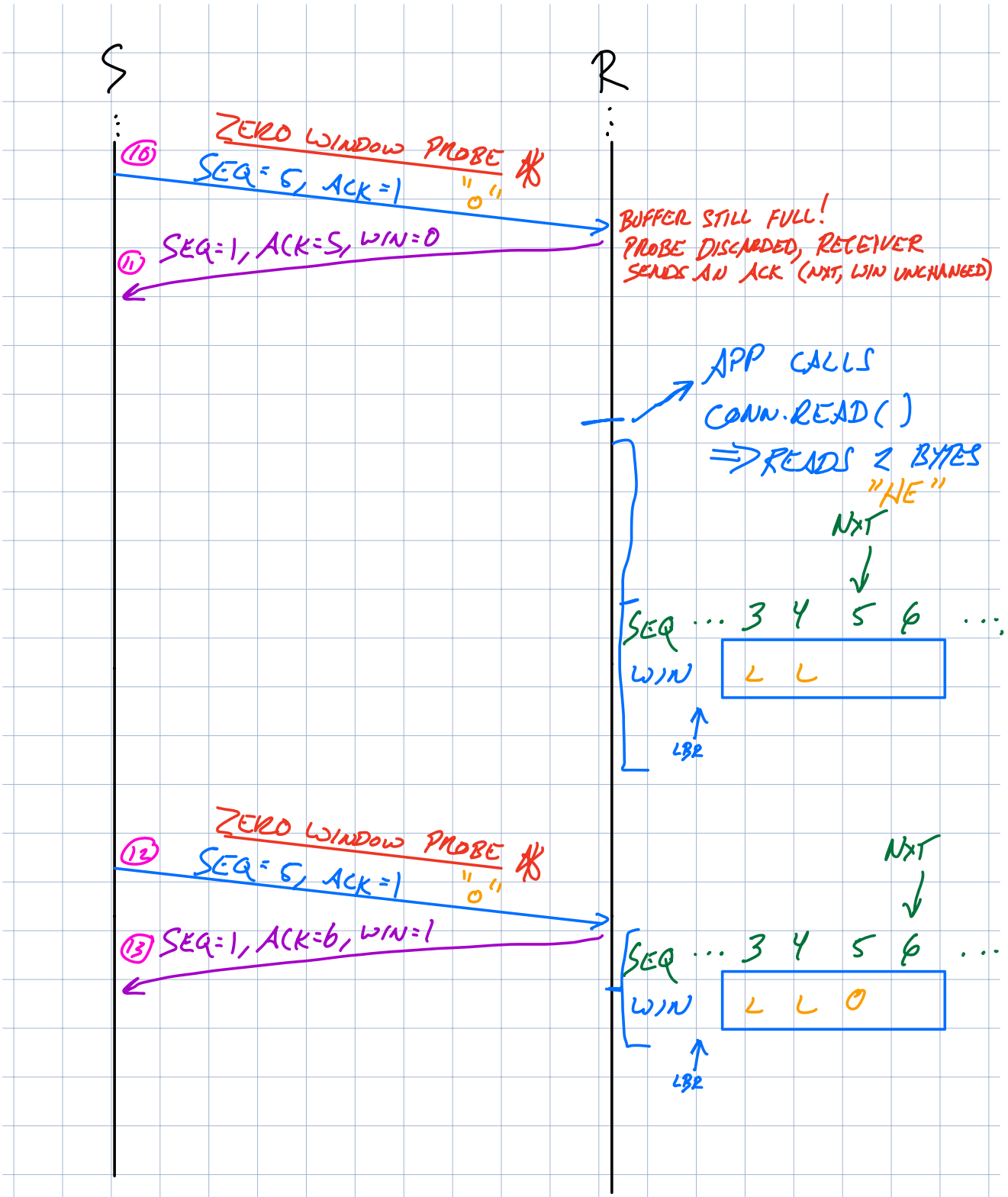
When receiver's window is full, sender enters zero window probing mode

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

1 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.


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

# 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):

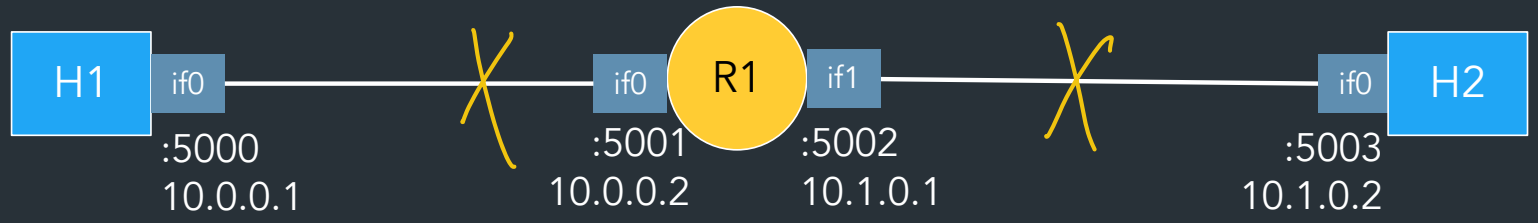
```
> drop 0.01   // Drop 1% of packets
> drop 0.5    // Drop 50% of packets (way too aggressive)



> 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

# 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!)

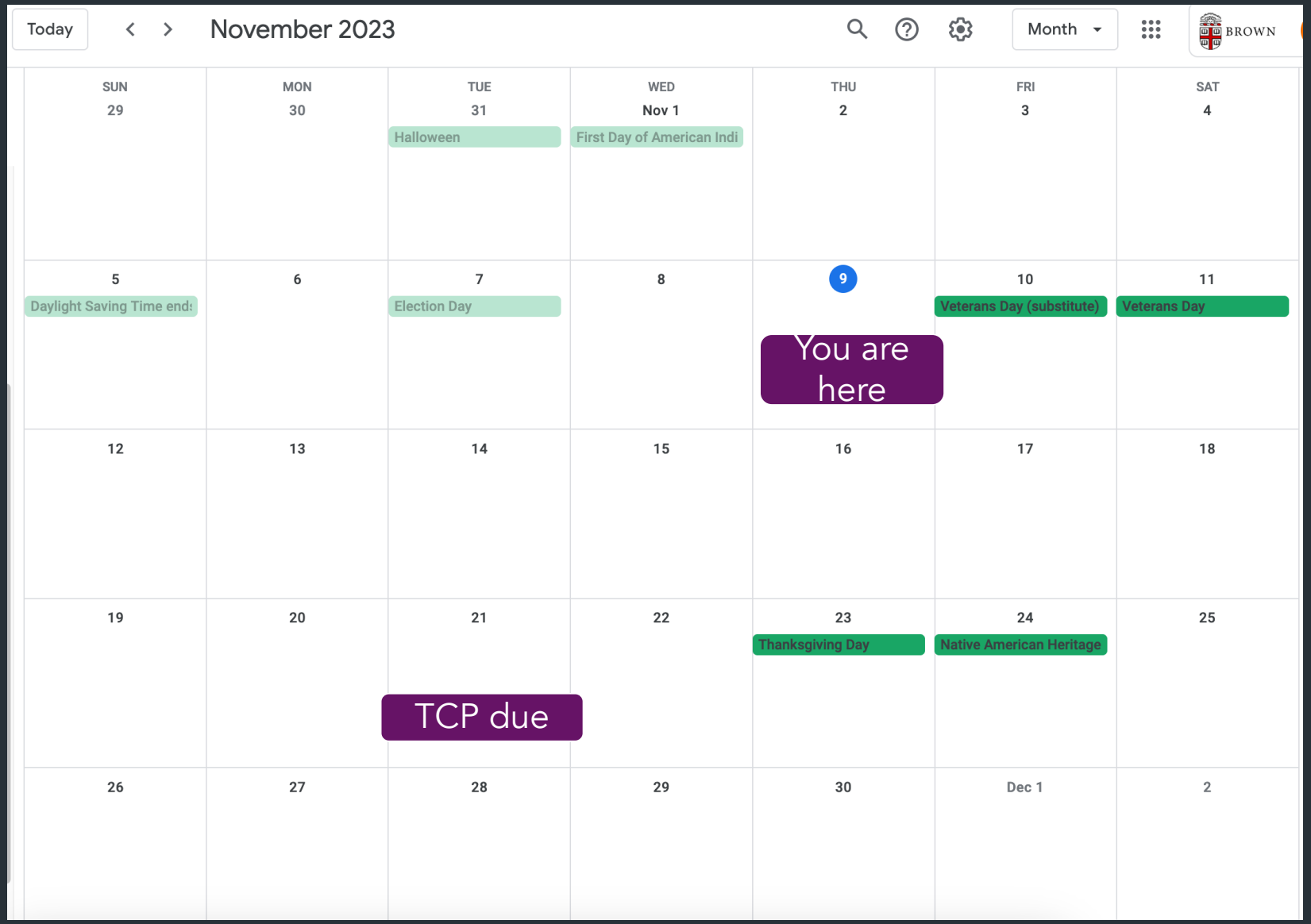
# Closing thoughts

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

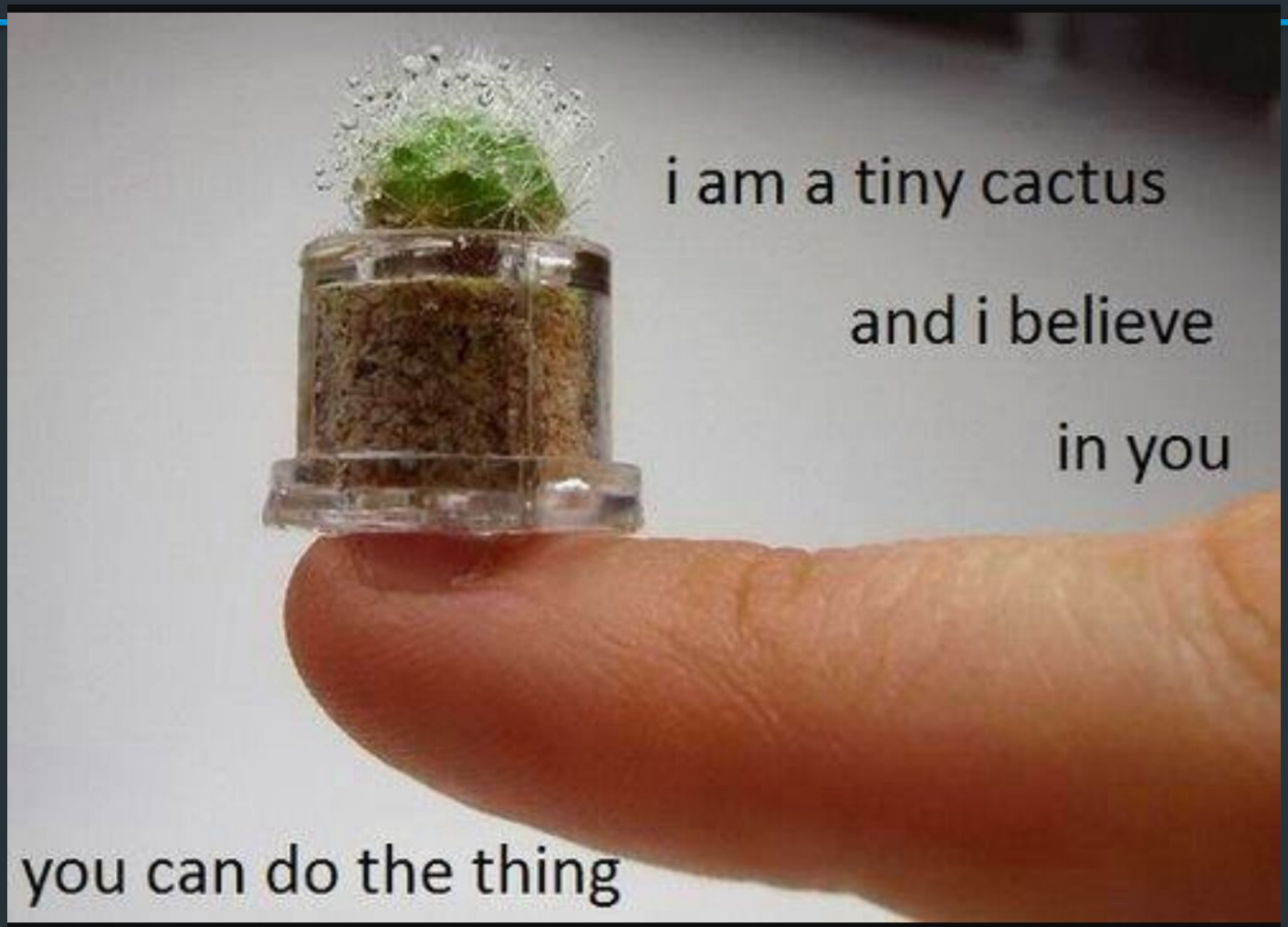
When stuck, take a break and come back to it.  It will help.

=> Do NOT wait until the last minute.

Don’t panic.

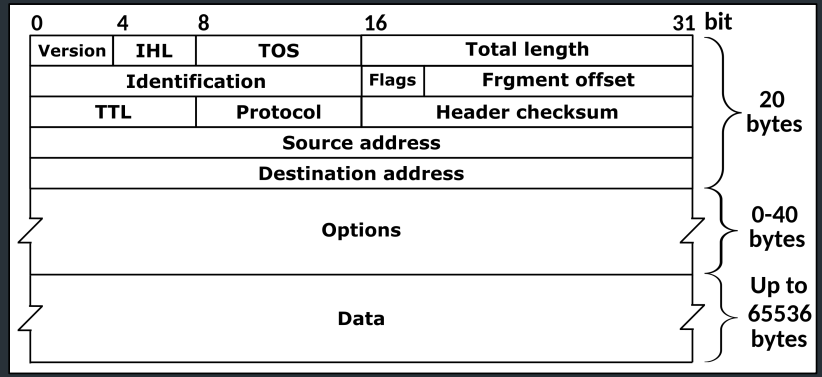

Today
November 2023
Month
BROWN
SUN
MON
TUE
WED
THU
FRI
SAT
29
30
31
Halloween
Nov 1
First Day of American Indi
2
3
4
5
Daylight Saving Time ends
6
7
Election Day
8
9
You are here
10
Veterans Day (substitute)
11
Veterans Day
12
13
14
15
16
17
18
19
20
21
TCP due
22
23
Thanksgiving Day
24
Native American Heritage
25
26
27
28
29
30
Dec 1
2

# Breathe



# The TCP checksum

… is pretty weird



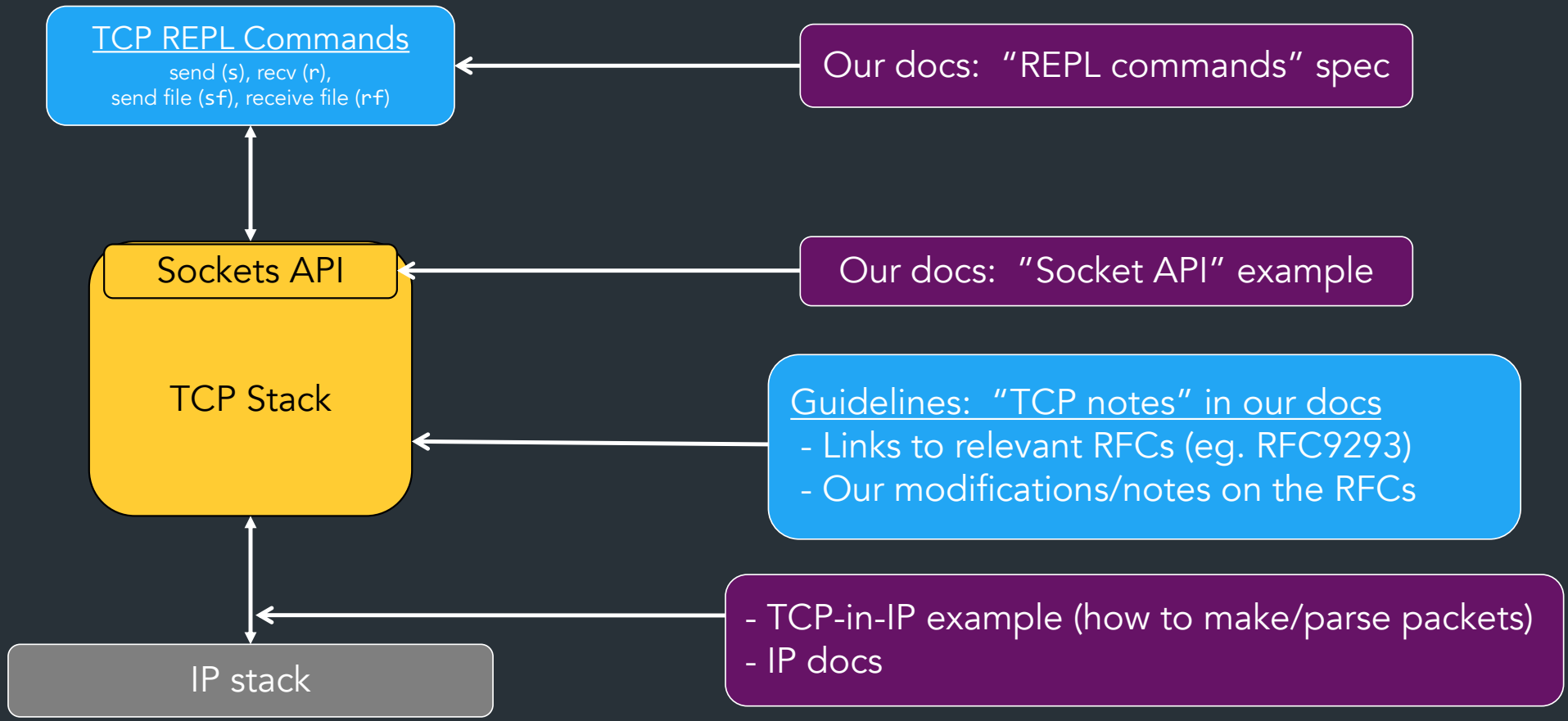

Computing the TCP checksum involves making a "pesudo-header" from TCP header + IP header fields:

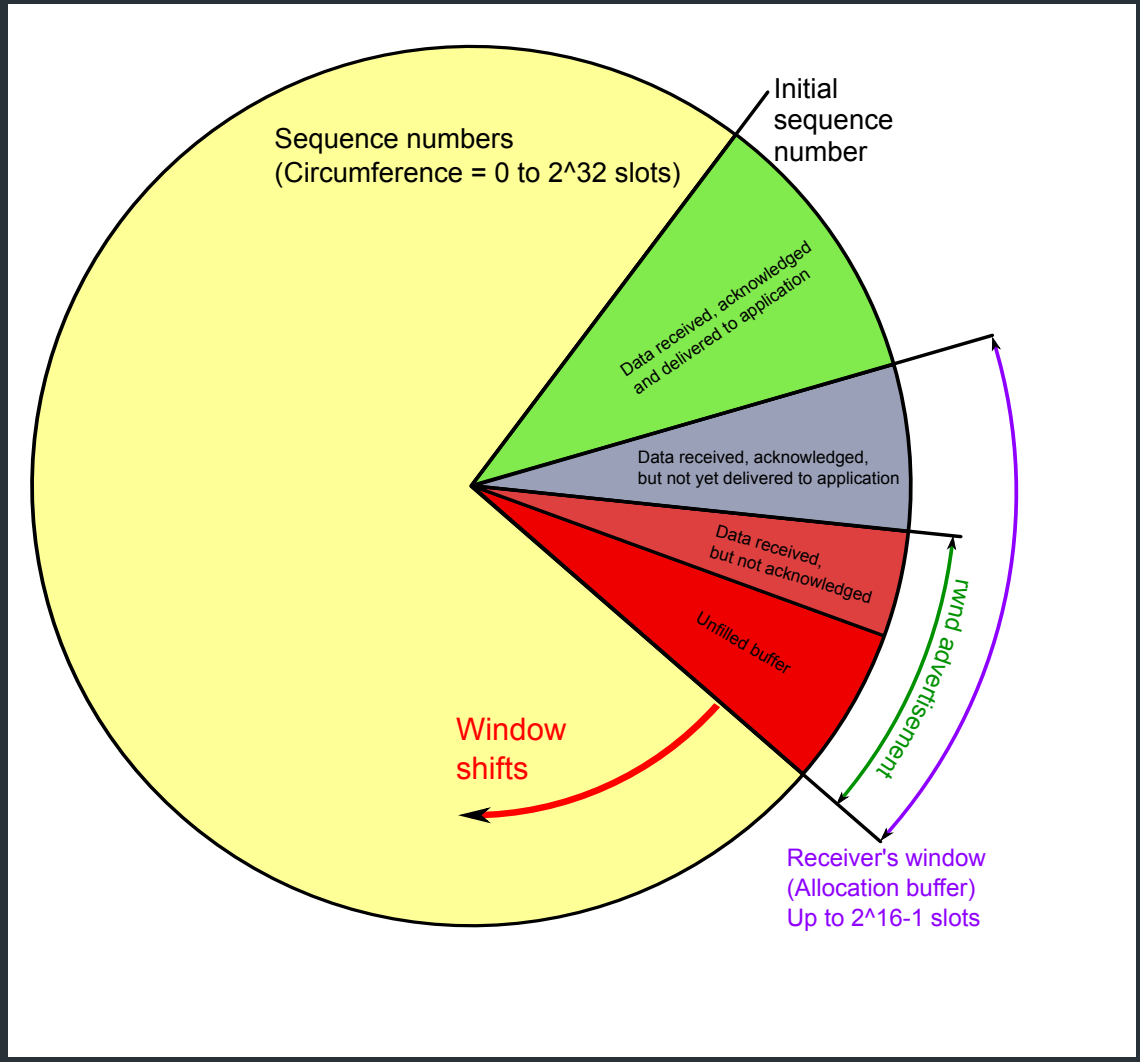
**TCP pseudo-header for checksum computation (IPv4)**

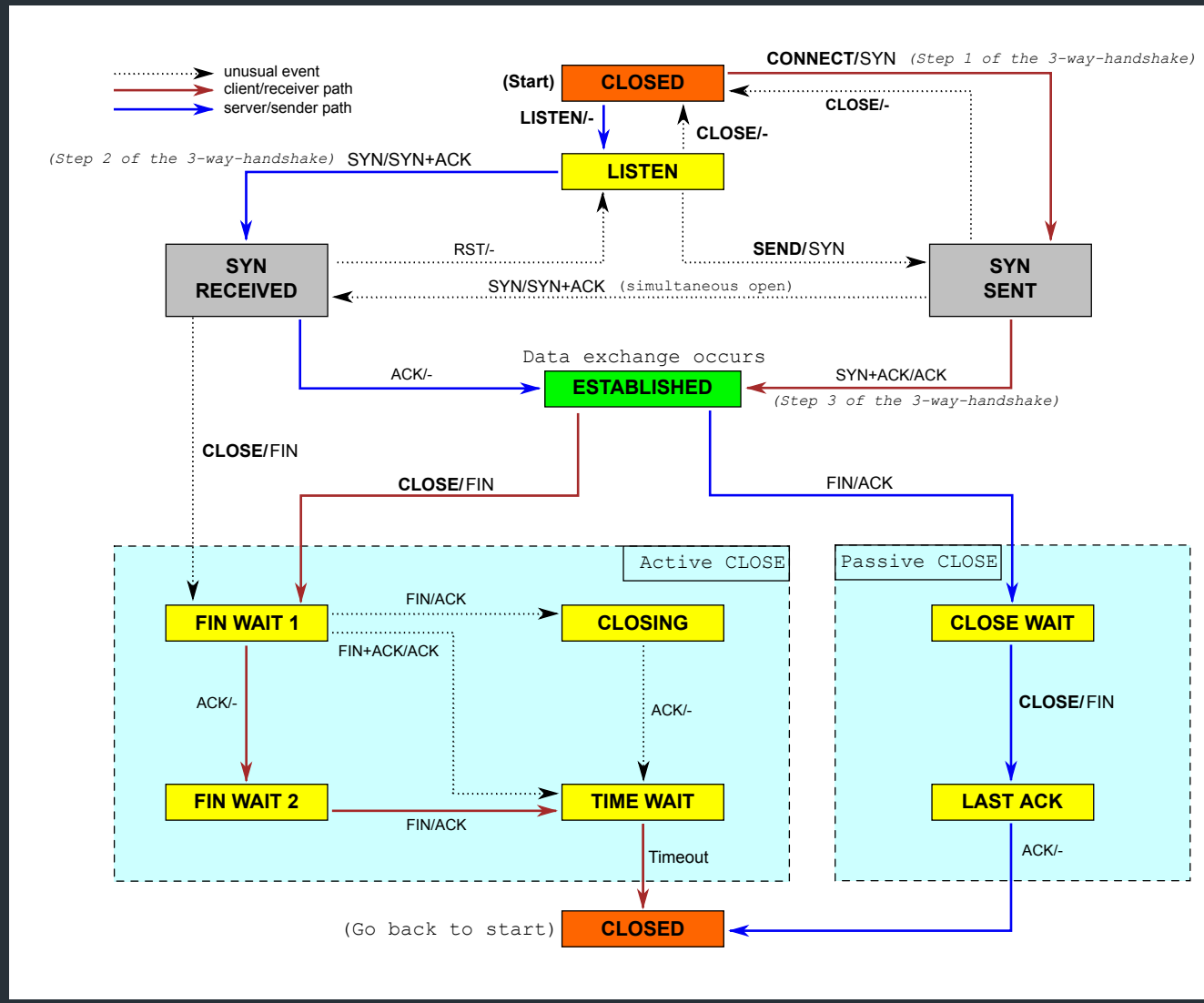
| Bit offset | 0–3 | 4–7 | 8–15 | 16–31 |
|---|---|---|---|---|
| 0 | Source address | | | |
| 32 | Destination address | | | |
| 64 | Zeros | | Protocol | TCP length |
| 96 | Source port | | | Destination port |
| 128 | Sequence number | | | |
| 160 | Acknowledgement number | | | |
| 192 | Data offset | 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
Window
shifts
rwnd advertisement
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
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE WAIT
CLOSE/FIN
FIN WAIT 2
FIN/ACK
TIME WAIT
LAST ACK
Timeout
ACK/-
(Go back to start)
CLOSED