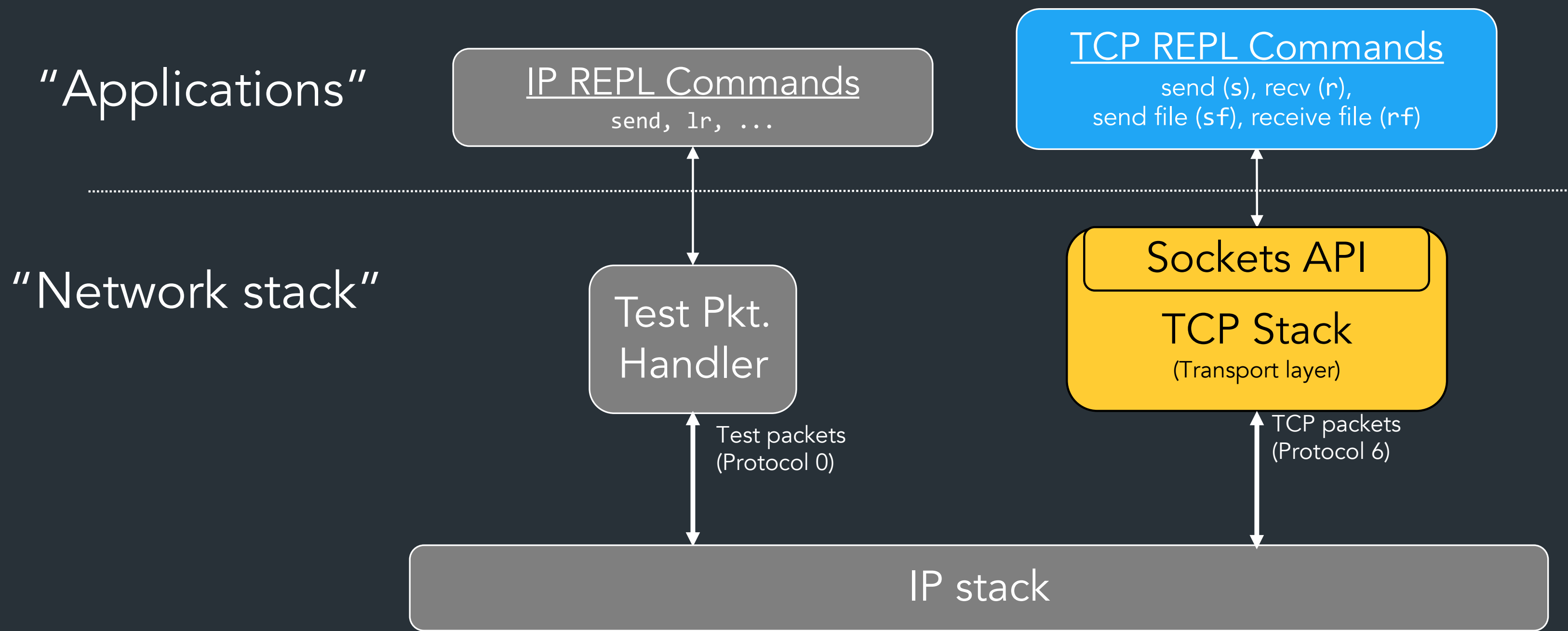
TCP Gearup II

Overview

- How to think about send/recv
- About buffers
- How to debug/test in wireshark
- Implementation notes
- Any questions you have


"Applications"
IP REPL Commands
send, lr, ...
TCP REPL Commands
send (s), recv (r),
send file (sf), receive file (rf)
"Network stack"
Test Pkt. Handler
Sockets API
TCP Stack
(Transport layer)
Test packets
(Protocol 0)
TCP packets
(Protocol 6)
IP stack

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

NO RETRANSMISSIONS

Relevant materials

- Lecture 14 (10/24): Send/recv basics
- Lecture 15 (10/26): How sliding window works, retransmissions, zero-window probing

- HW3: Do it sooner rather than later—it will help!

- Testing and tools stuff: "TCP getting started" in docs
  => More coming soon!

VWrite ("s" command in REPL)
- Input: some normal socket, data you want to send
    => You need to define your send/recv buffer, what variables/state etc you need to represent them
- Load data into your send buffer
- Block if send buffer is full, otherwise return number of bytes send

VRead ("r" command)
- Input: normal socket, buffer for recevied data
- Read from recv buffer, write that data to whatever buffer was passed in
- If recv buffer is empty, block
- Return: number of bytes read***

Your goals:
- Definingi data structures (buffers, etc), variables for how you keep track of things in the buffer
- Receive packets, load them into recv buffer
- SEnd packets from send buffer

Sending and receiving: API

More info: "Socket API example" in docs

VWrite ("s" command)

- Input: normal socket, data to send
- Loads data into send buffer
- Block if send buffer is full

VWrite ("r" command)

- Input: normal socket, buffer for received data
- Read from recv buffer, write to app buffer
- Block if recv buffer empty
- Return: number of bytes read

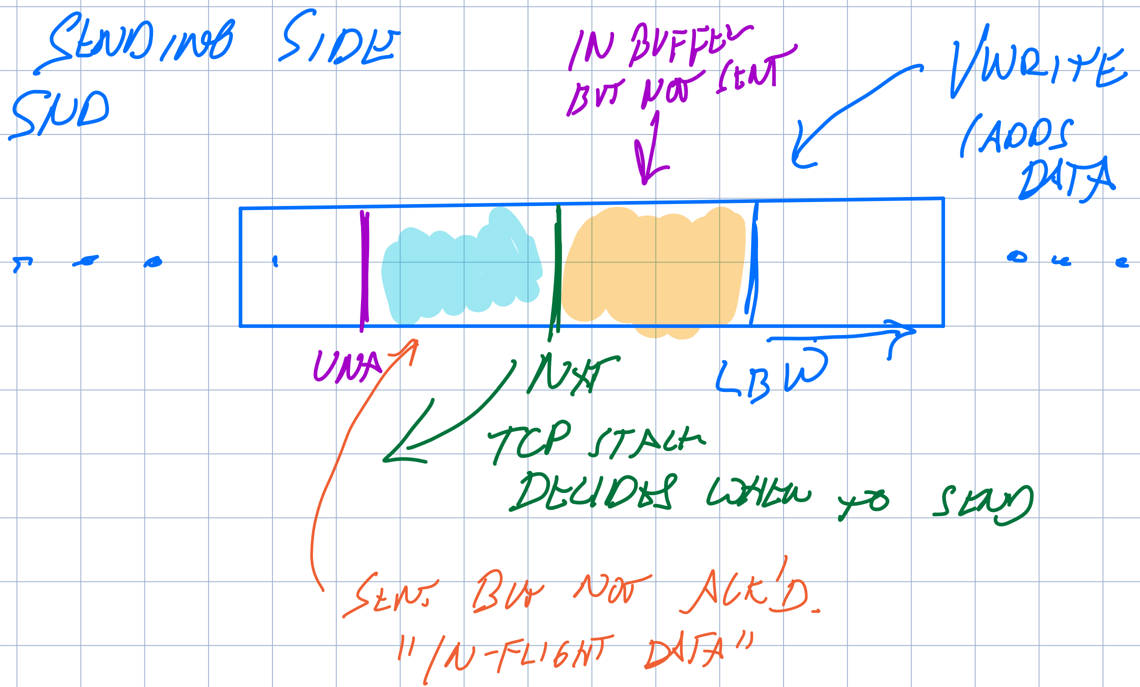
Demo!

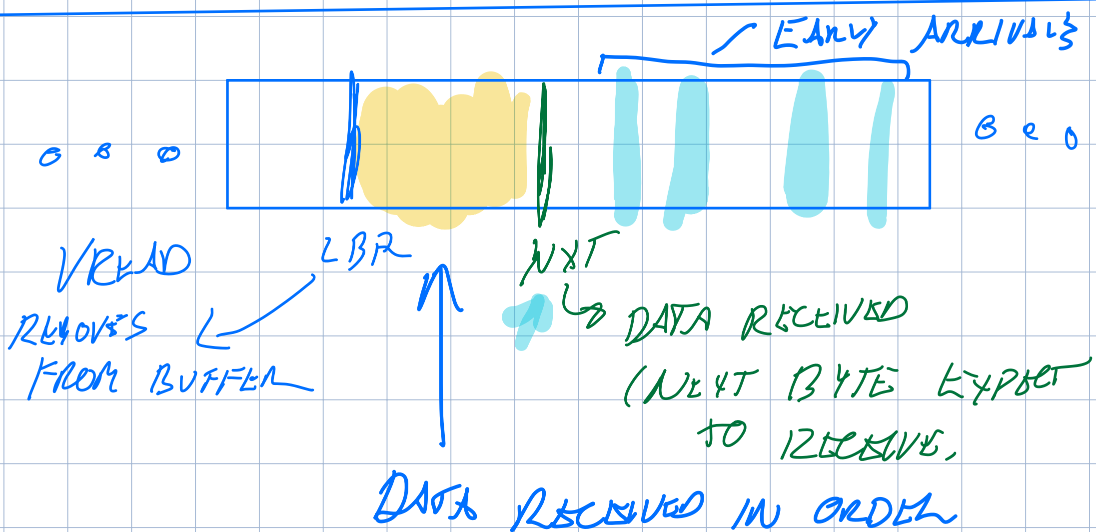
Your buffers

SND BUF
RCV BUF

- Must use a circular buffer
- You get to decide on mechanics
  - How to keep track of read/write pointers
  - How to translate between sequence numbers => buffer indices

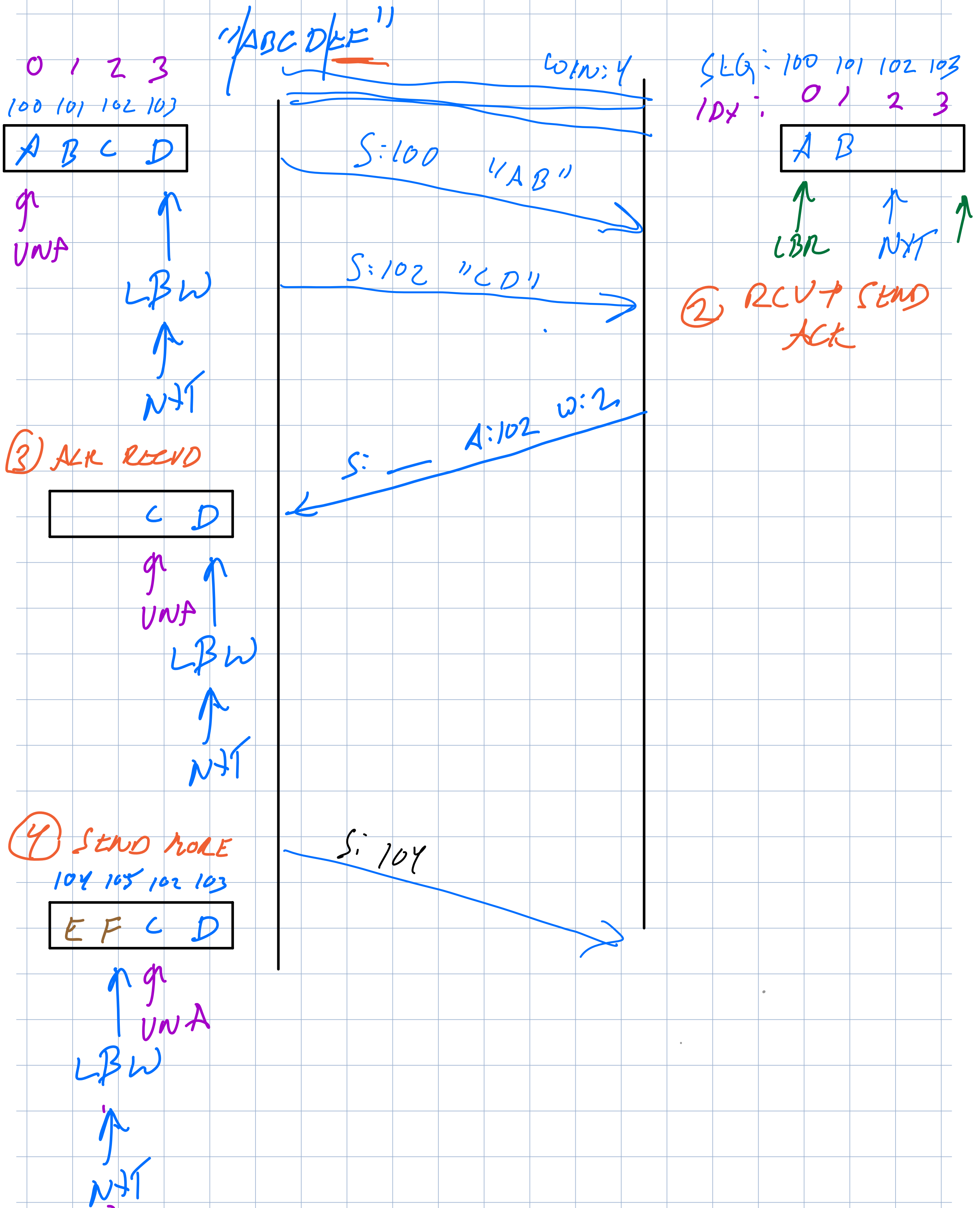
For detailed info
=> RFC9293 Sec 3.3: what all the variables mean
⇒ Lecture 15 (Oct 26): detailed breakdown of how to use buffers





[Want to see a better version of this? See the notes from lecture 15.]

For more info on this part, we recommend doing HW3–it is designed to help here!



Your buffers

- Must use a circular buffer
- You get to decide on mechanics
  - How to keep track of read/write pointers
  - How to translate between sequence numbers => buffer indices
- Later: okay to store some data outside send/recv buffer
  - Out-of-order segments, unACK'd segments, …

For detailed info
=> RFC9293 Sec 3.3: what all the variables mean
=> Lecture 15 (Oct 26): detailed breakdown of how to use buffers

What happens in the TCP stack?

Your TCP stack will have some threads—you decide what they do

When you get a new packet…

=> Look up 4-tuple in socket table => find socket struct

=> Socket struct => all your per-connection TCP state (buffers, sequence numbers, etc.…)

What to do with each segment? RFC9293 Sec 3.7.10 is your friend
=> + our modifications in "TCP notes" docs

Implementing VRead/VWrite

<u>Performance requirement:</u> send/recv process **MUST** be event driven

- No `time.Sleep`
- No busy-waiting

<u>Where does this apply?</u>

- REPL: s, r, sf, rf
- VRead/VWrite
- Deciding when to send, or check for new data
- ~~Retransmissions~~ (TICKERS + STUFF FINE FOR RETRANSMISSIONS

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

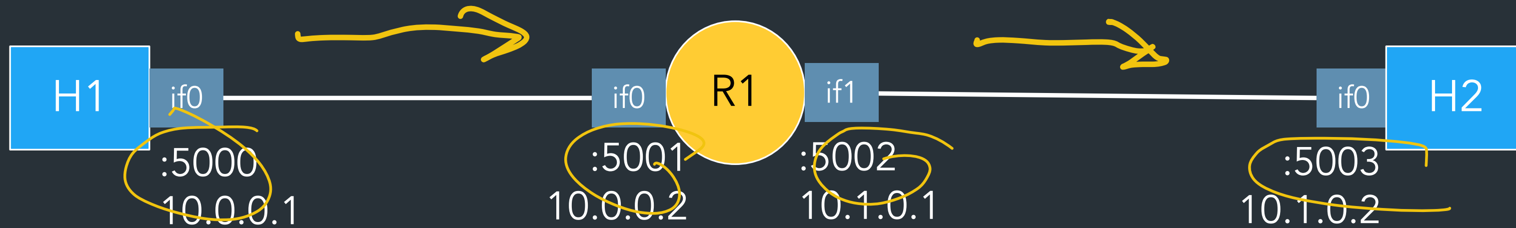
Channels?

SEE CODE DISCUSSION IN VIDEO (REALLY)

⇒ "CHANNELS DEMO" IN DOCS + RESOURCES

How to test TCP

More docs coming soon!



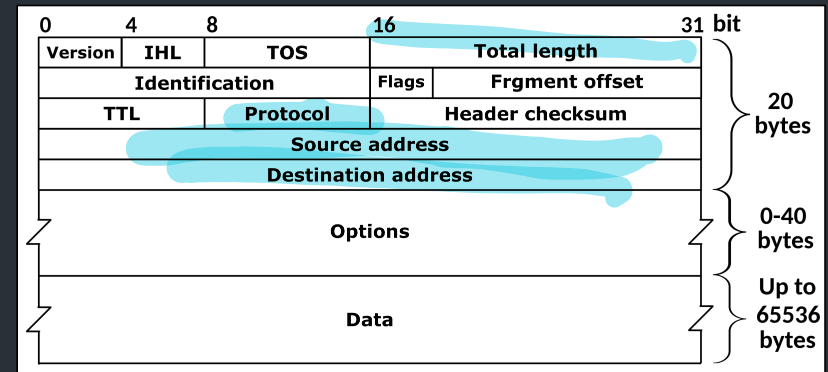
Useful wireshark mechanics

- SEQ/ACK analysis
- Follow TCP stream
- Validating the checksum

Note: watching traffic in wireshark works differently in this project!
=> See "TCP getting started" guide for details

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

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!)

Roadmap

Milestone II

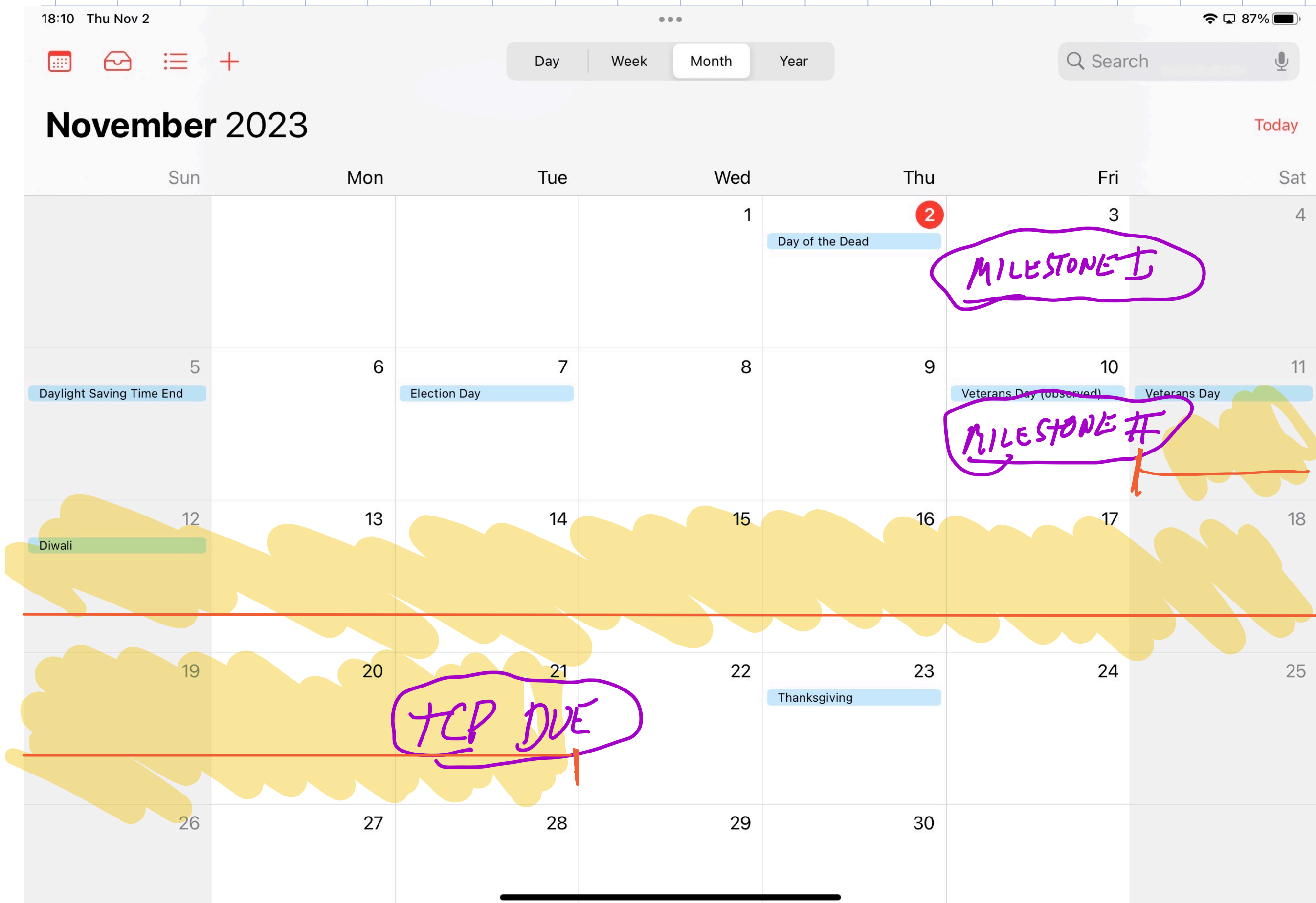
- Basic sending and receiving using your sliding window/send receive buffers
- Plan for the remaining features

NO RETRANSMISSIONS

Roadmap

Final deadline

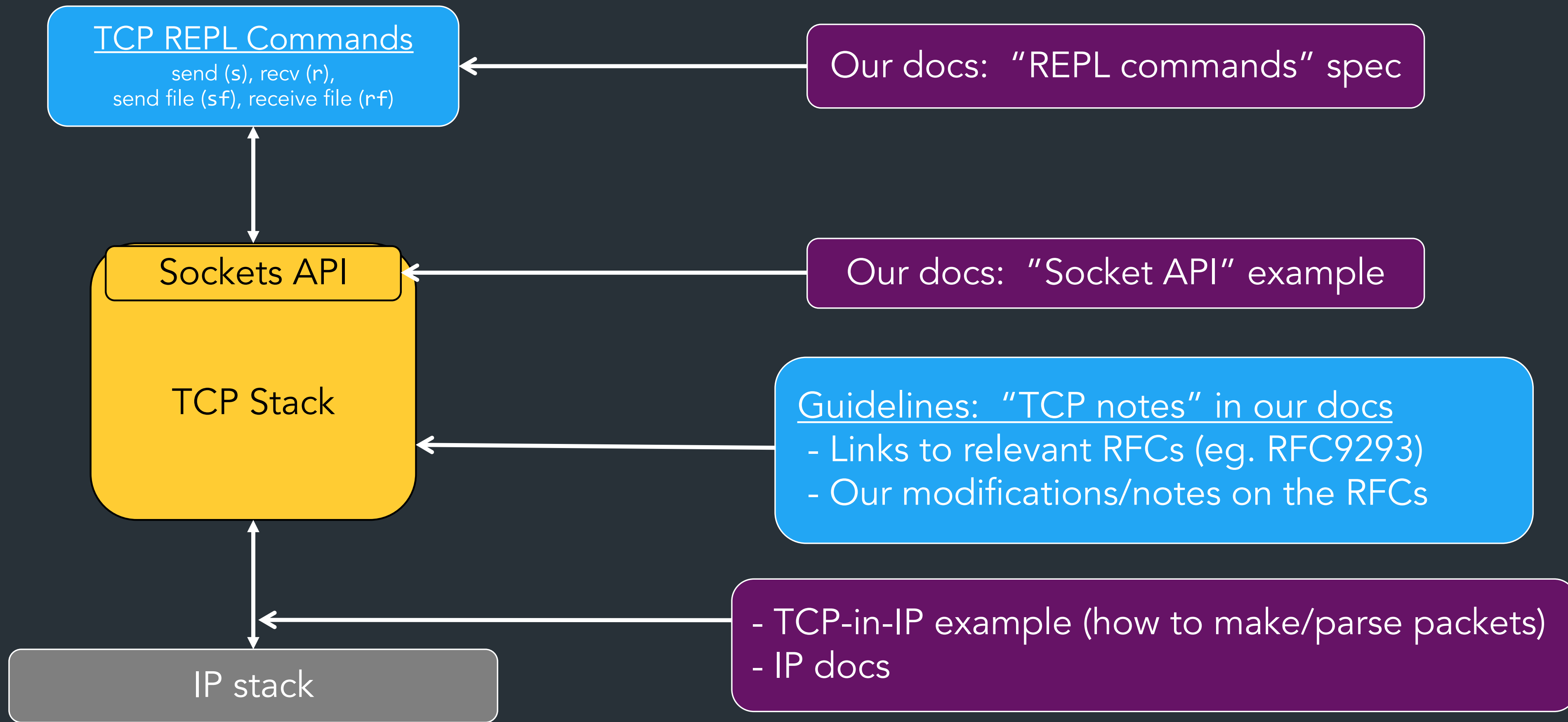
- Retransmissions (+ computing RTO from RTT)
- Zero-window probing
- Connection teardown
- Sending and receiving files (sf, rf)



The features you need after Milestone II are not trivial—there is a lot of testing and debugging involved, so <u>do not underestimate this part</u>. All of your other course deadlines are set in order to ensure you have enough time between Milestone II and the TCP deadline.

<u>What this means now:</u> make sure you use your Milestone II time wisely so that you can spend the time afterward to focus on the other features!

Where to get more info

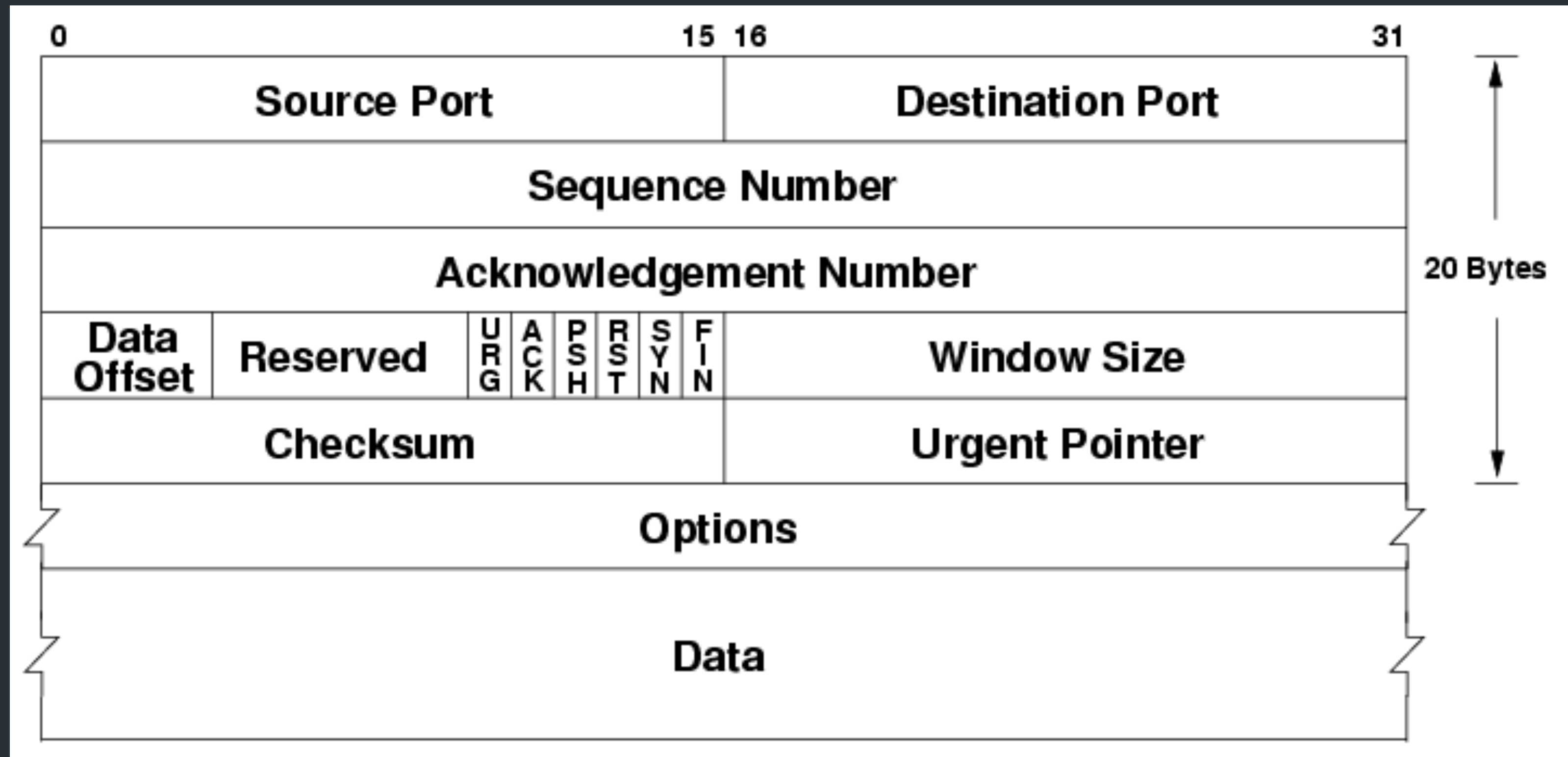


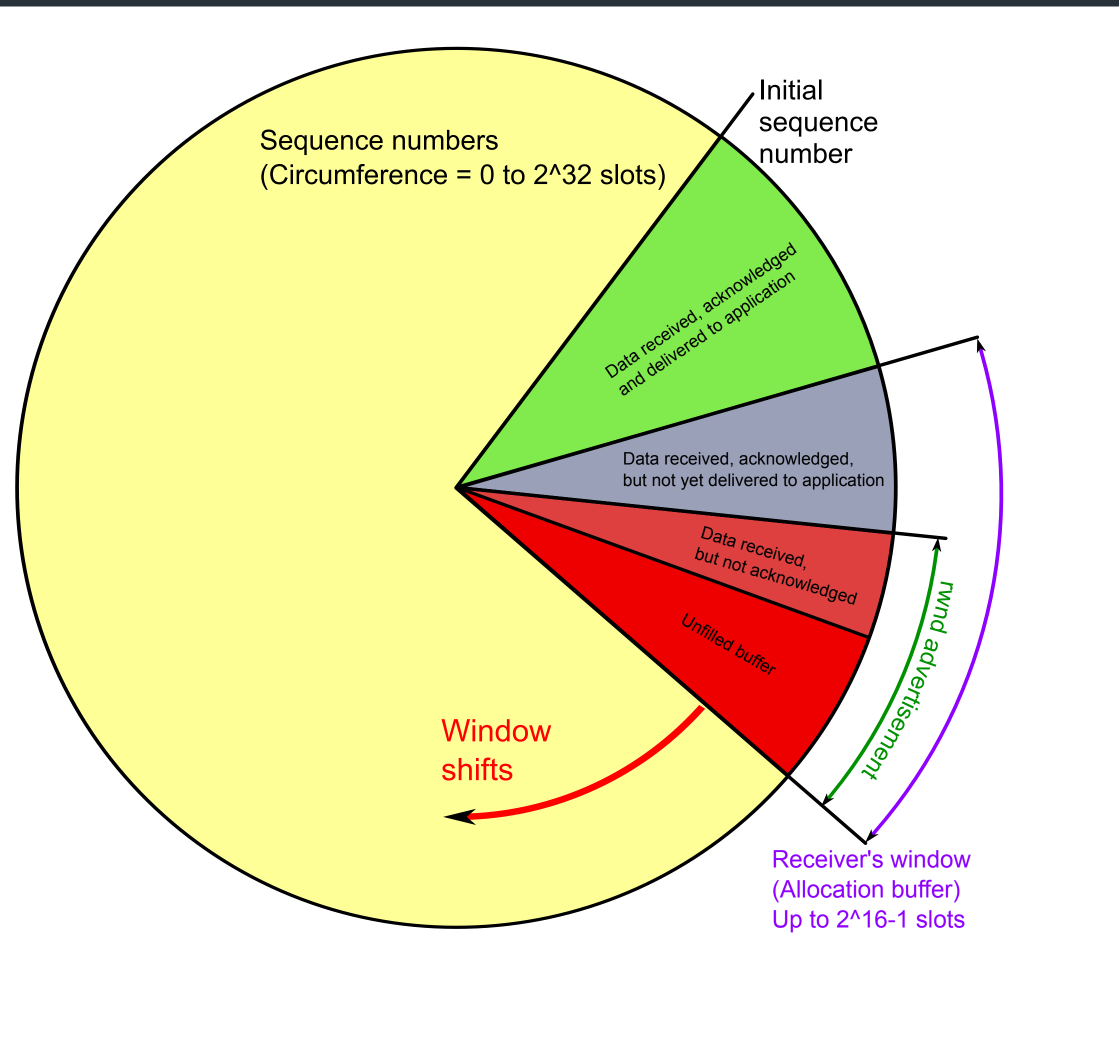
Closing thoughts

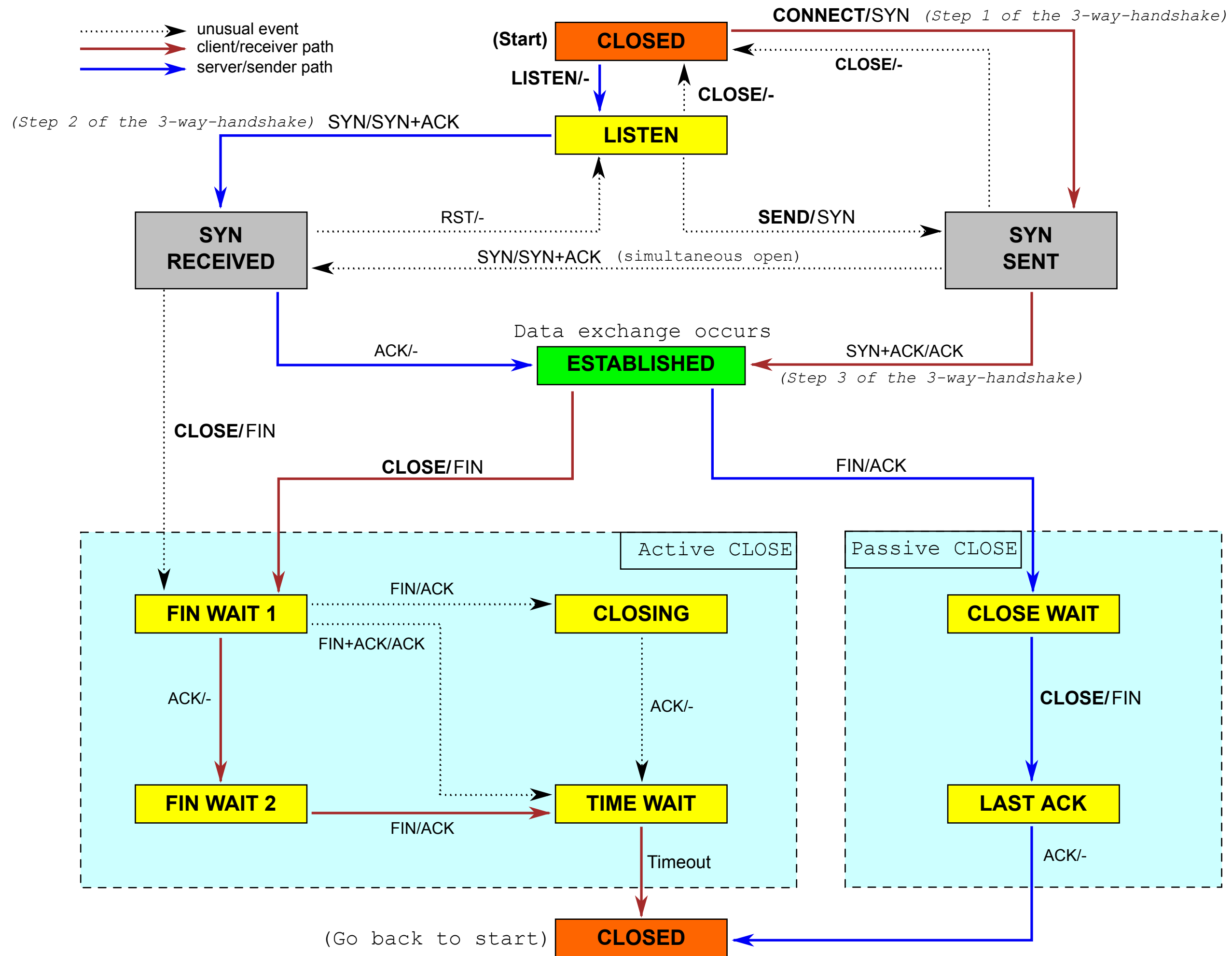
- Use your milestone time wisely!
- Wireshark is the best way to test—use it!
- Stuck? Don't know what's required? Just ask!
  (And see Ed FAQ)

We are here to help!

TCP Header




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