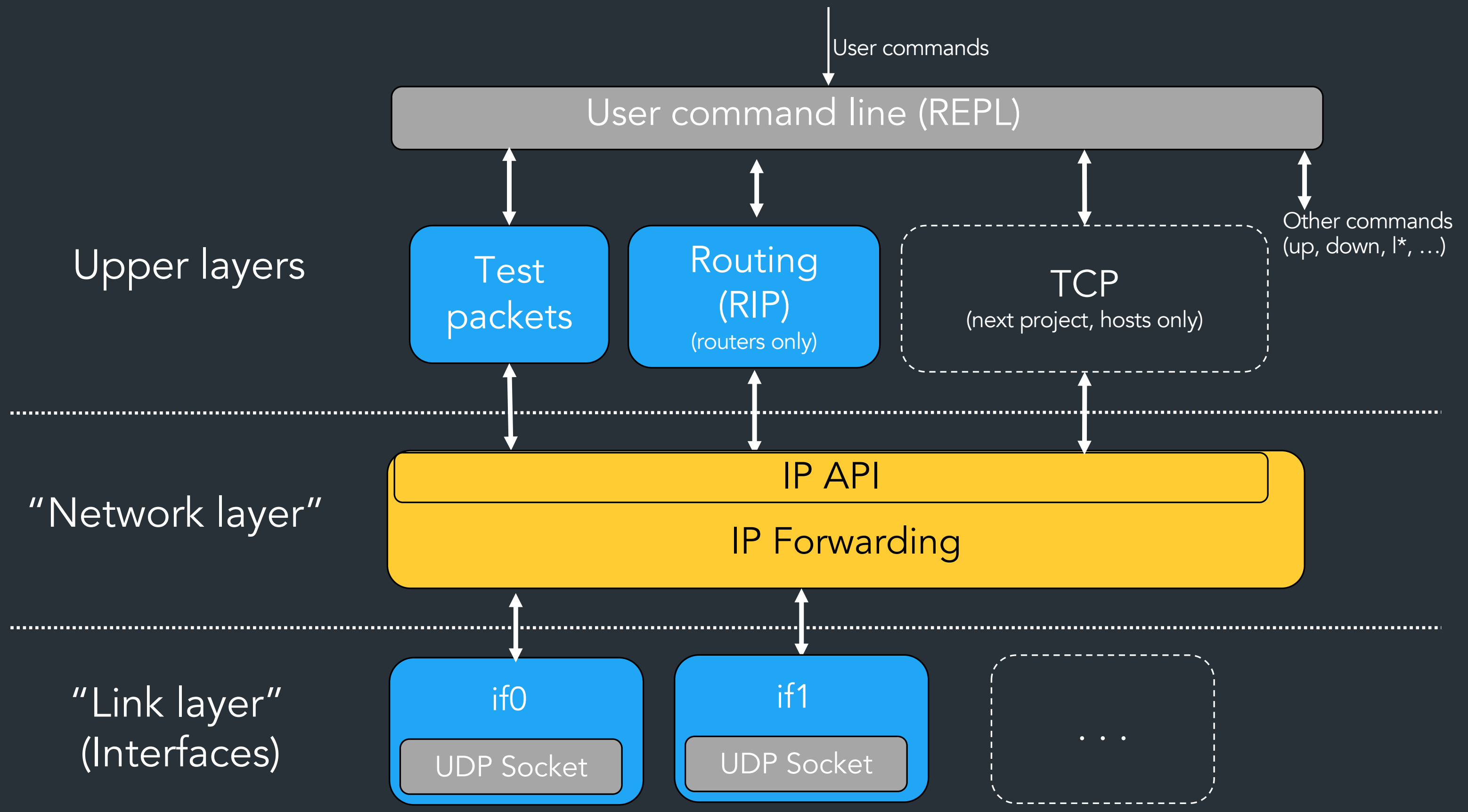
TCP Gearup I

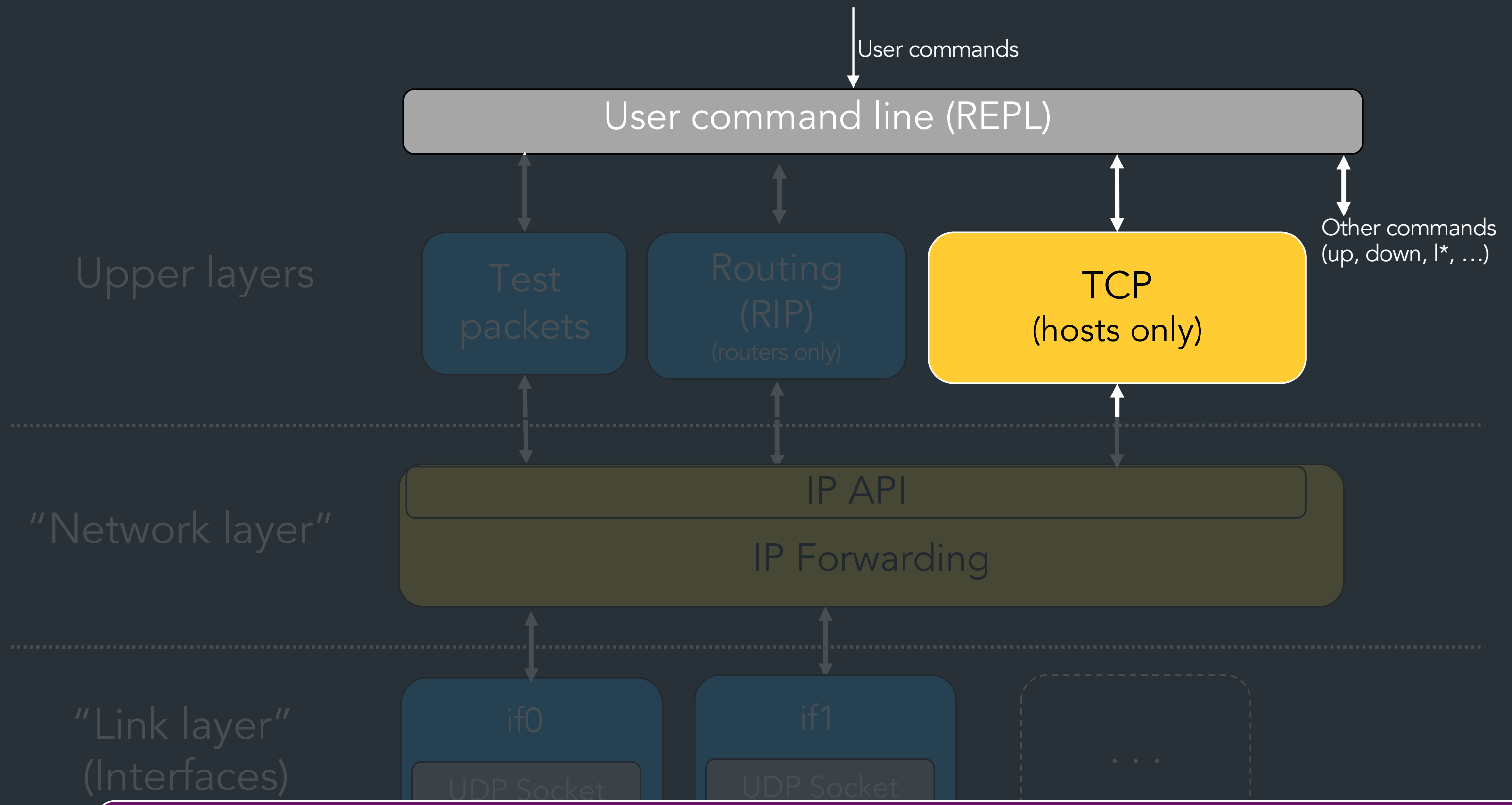
Overview

- How this project fits into IP
- What you will build
- How to debug/test in wireshark
- Implementation notes
- Any questions you have

The Big Picture: Last time

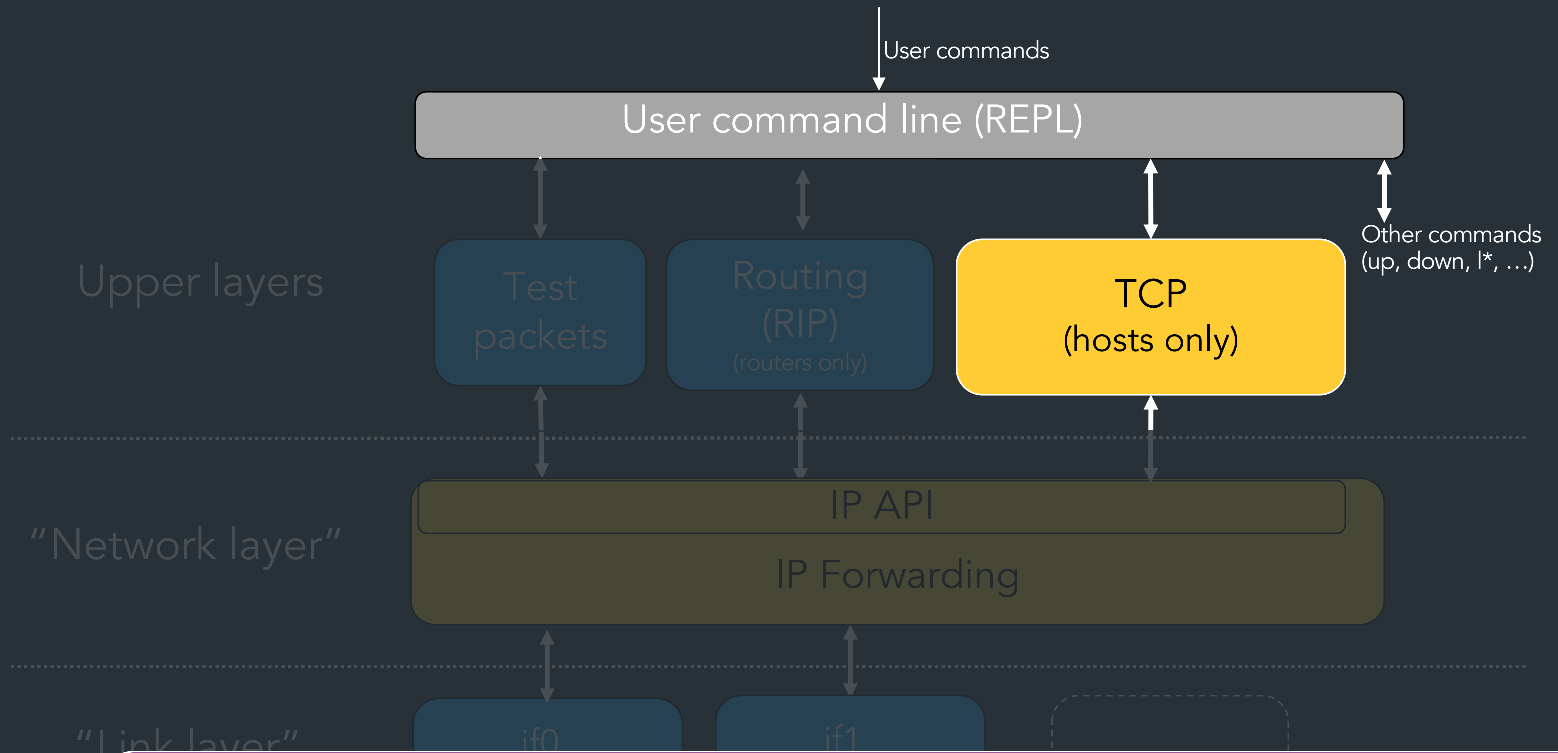


Where we are now



⇒ A new “higher layer” in your IP stack (on the same level as test packets)

Where we are now

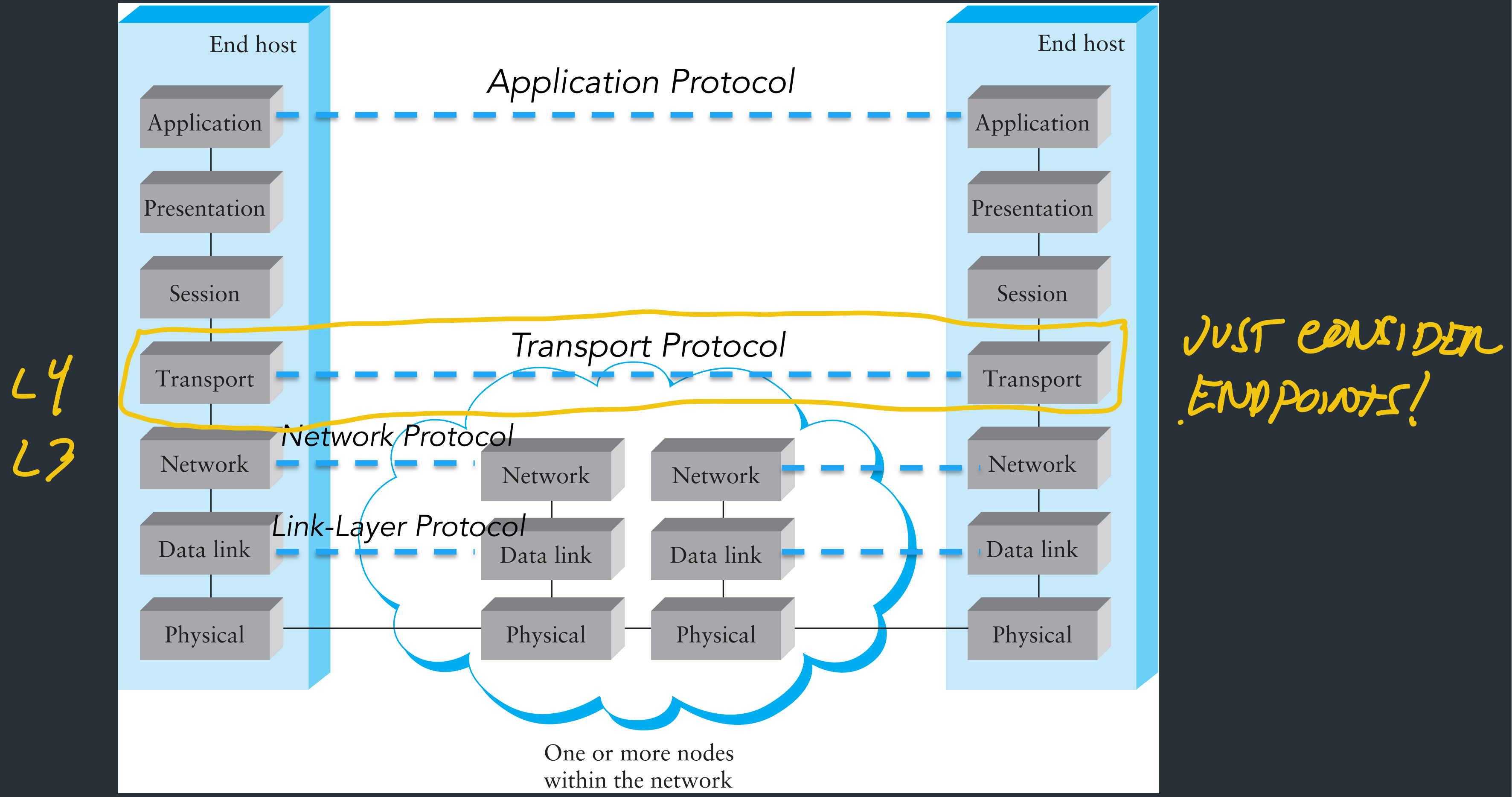


⇒ A new "higher layer" in your IP stack (on the same level as test packets)

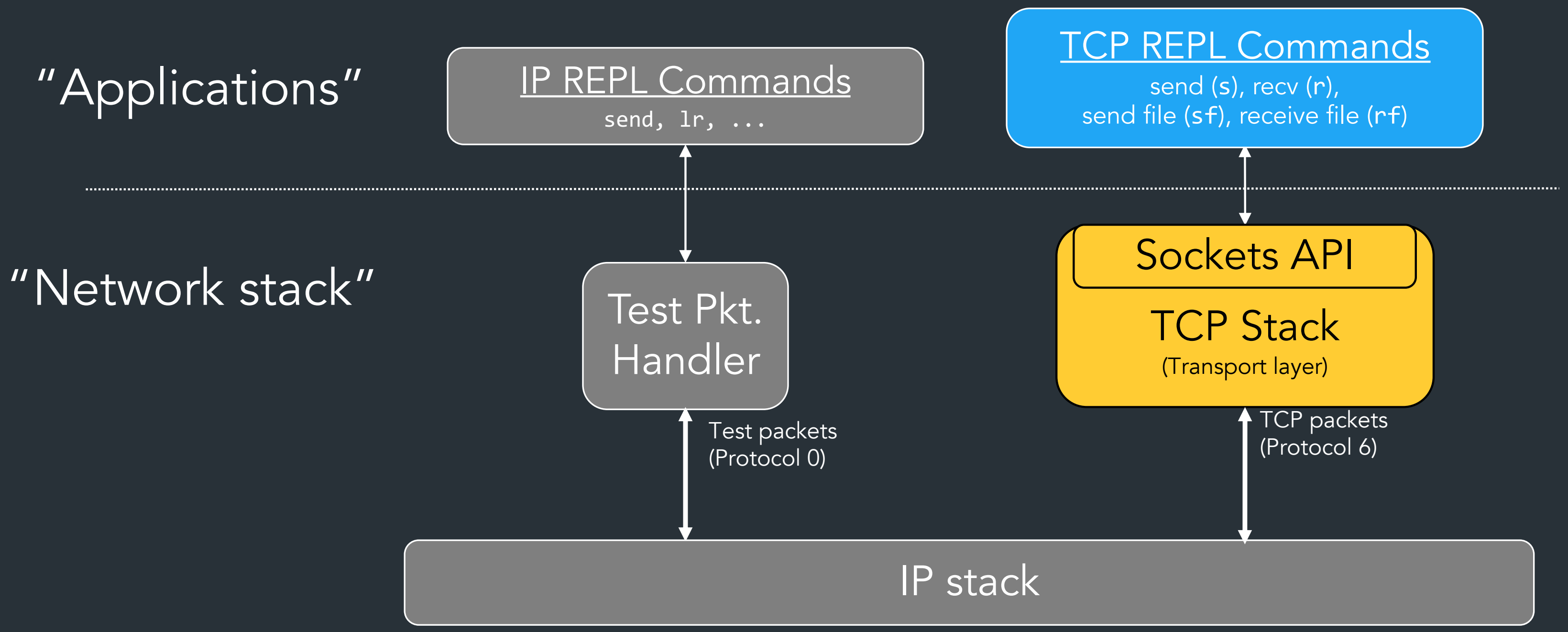
⇒ For hosts ONLY

⇒ You are done modifying your router at this point

Remember this picture?

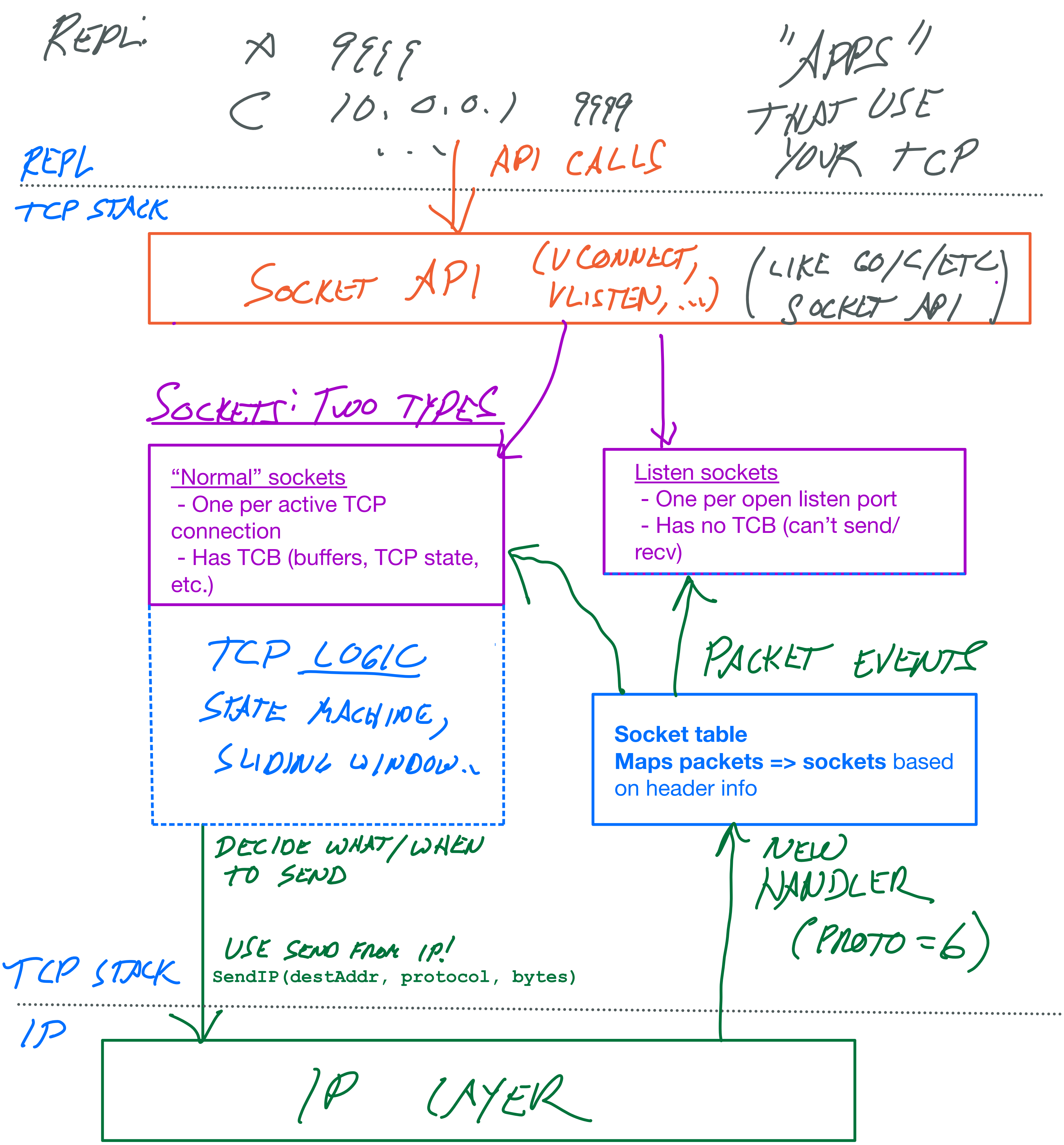


Let’s break it down



What goes in your TCP stack?

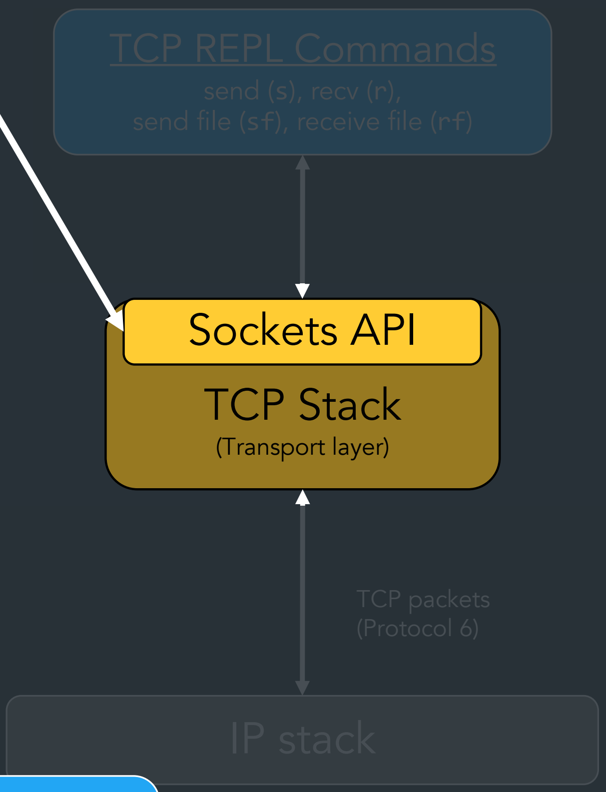
TCP STACK: THE COMPONENTS



API for sockets: abstraction for creating and using TCP connections

Model: Go's socket API (you'll make your own!)

```
conn, err :=  net.Dial("tcp", "10.0.0.1:80")
. . .

someBuf := make([]byte, . . .)
conn.Write(someBuf)
```



Example: our socket API (yours can look different)

```
conn, err := tcpstack.VConnect(addr, port)
. . .

someBuf := make([]byte, . . .)
conn.VWrite(someBuf)
```

Guidelines: "Socket API" specification in docs
(You get to design your own API!)

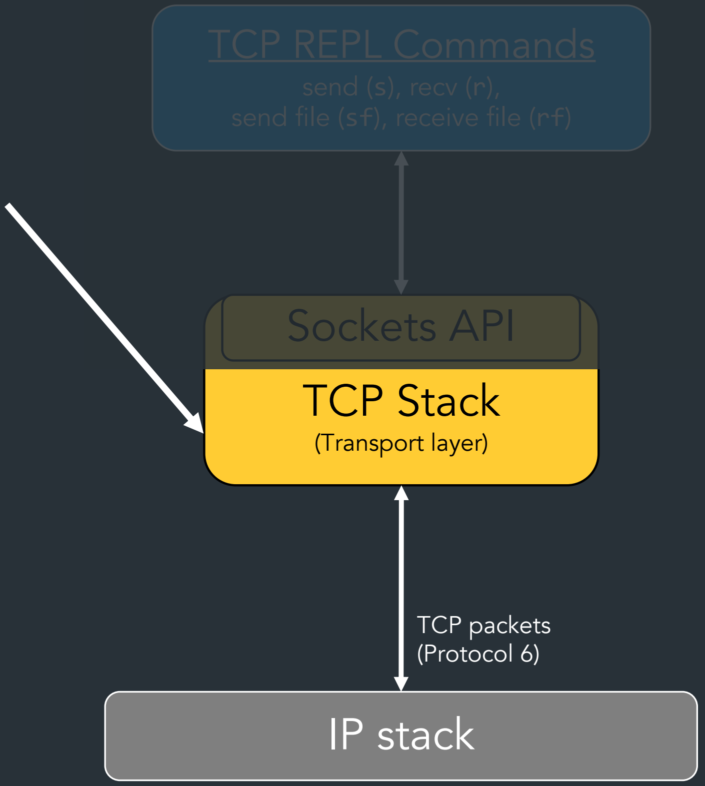
Focus for Milestone 1

```
VListen(port)           // Listen on a port
VConnect(addr, port)    // Connect to a socket
VAccept(. . .)          // Accept new connections (more on this later)



VWrite(. . .)            // Send on a socket
VRead(. . .).            // Recv on a socket



VClose(. . .)            // Close a socket
```

Guidelines: “Socket API” specification in docs

<u>TCP stack</u>: logic that happens "under the hood" to make sockets work (ie, the TCP protocol)

- Should be a separate library you initialize at host startup (like your IP stack)
- Uses your IP stack to send/recv packets
  - IPSend(destIP, protocol, bytes)
  - New handler for TCP (protocol #6)



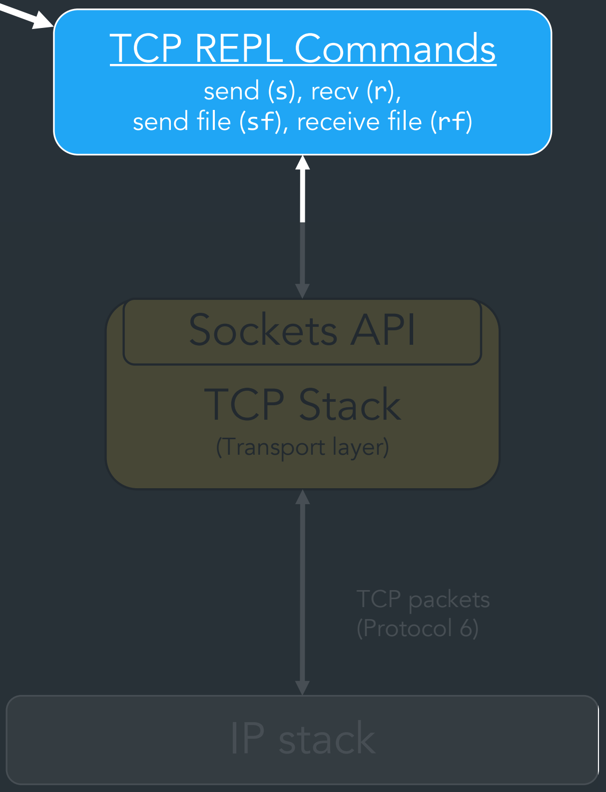
Guidelines: "TCP notes" in docs

REPL commands: how we'll test your

=> Think of these like "applications" that use your socket API

```
// Basic stuff (test your API)
a Listen on a port; accept new connections
c Connect to a TCP socket
ls List sockets

s Send on a socket
r Receive on a socket


cl Close socket

// Ultimate goal
sf Send a file
rf Receive a file
```

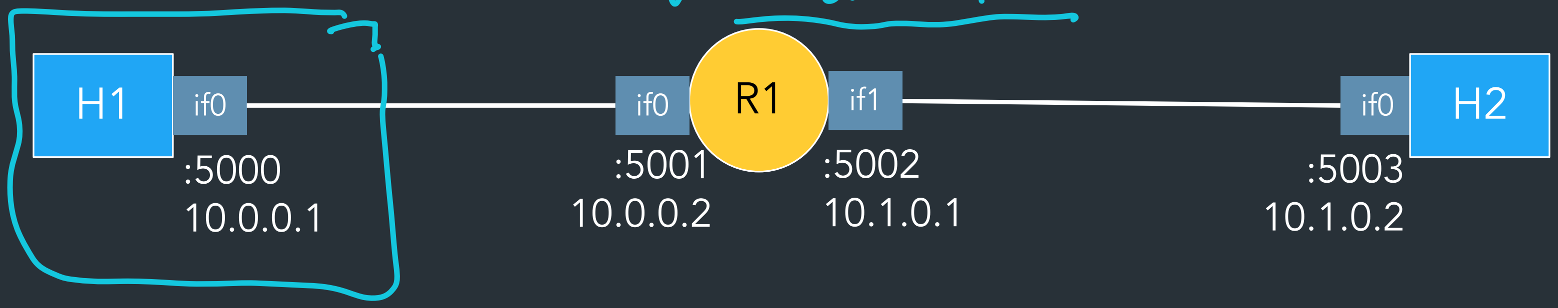
Focus for Milestone 1



Demo!

How to test TCP

IN WIRESHARK, MUST CAPTURE ON JUST ONE HOST!



Most of the time, use linear-r1h2 network

- Only one router, no need for RIP
- Can mainly use reference router
  - Will release an updated reference router next week (has extra features for later in project)

=> **Make sure your IP forwarding works with the reference router!! (Test with your host, our router)**

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

Roadmap

Milestone I

- Initial design for API and TCP stack
- Listen and establish connections => create sockets/TCB
- TCP handshake
- accept, connect, and start of ls REPL commands

How to think about connections

aka. Most important thing for Milestone 1

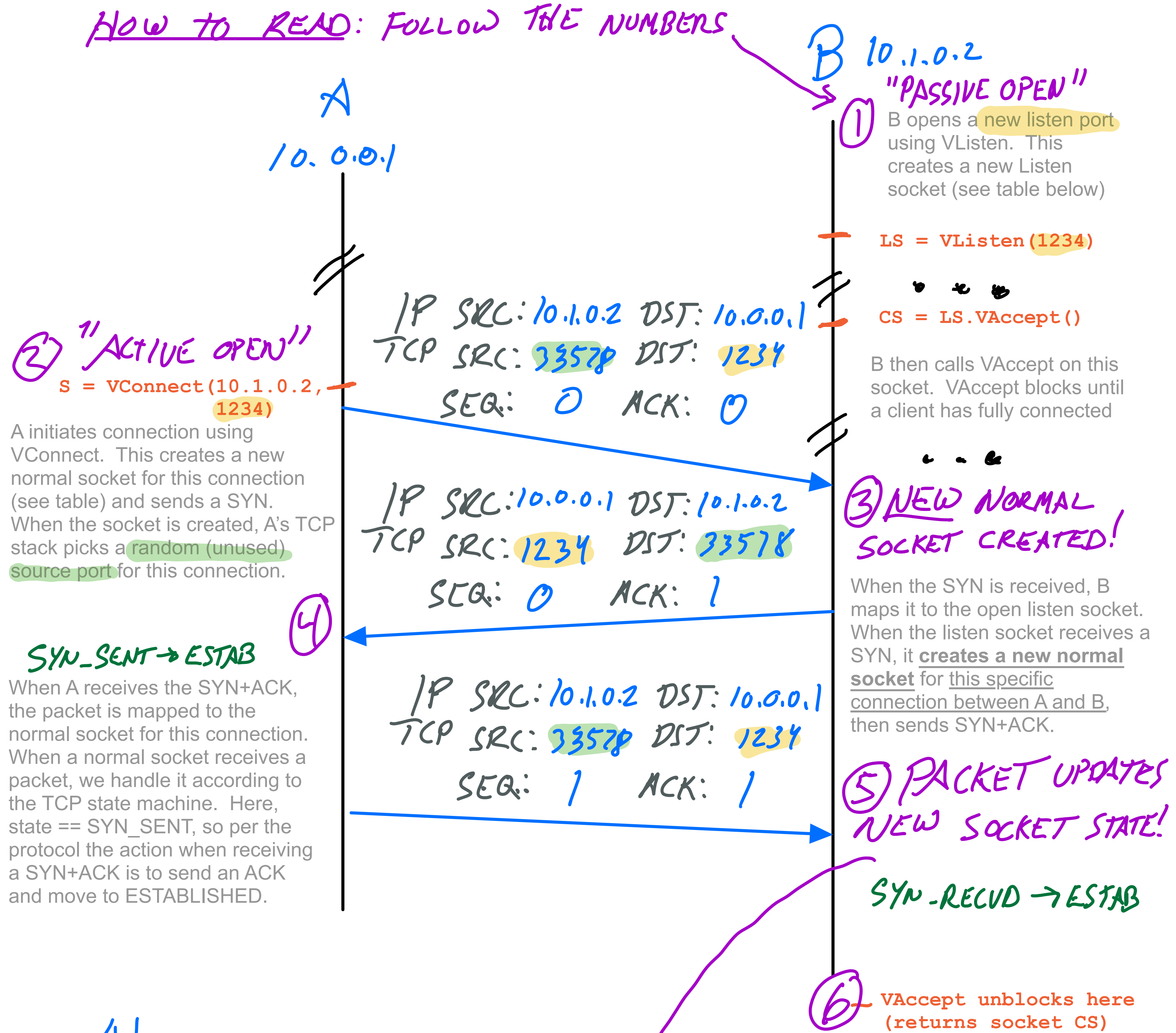
```
> ls
SID        LAddr  LPort       RAddr  RPort       Status
  0      0.0.0.0   9999     0.0.0.0      0       LISTEN
  1     10.1.0.2   9999    10.0.0.1  58060  ESTABLISHED
```

Relevant concept material

- Lec 12 (ports), Lec 13 (TCP handshake)
- HW2 problem 3

How to think about connection setup

Scenario: - B listens on port 1234 (ie, "a 1234")
- A connects to B's port (ie, "c 10.1.0.2 1234")



A's TABLE

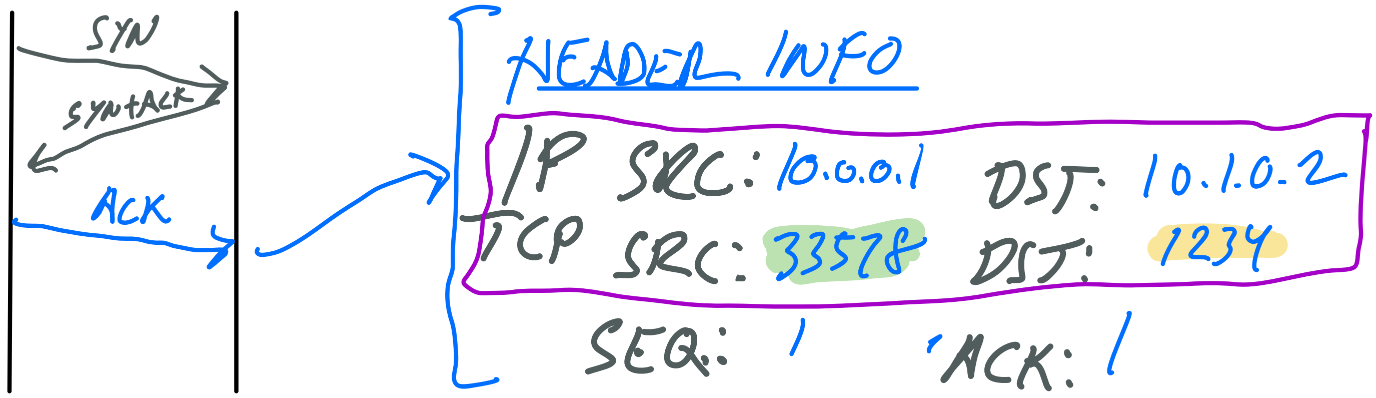
	LOCAL		REMOTE		STATE
	IP	PORT	IP	PORT	
2	10.0.0.1	33578	10.1.0.2	1234	~~SYN_SENT~~ ESTAB (4)

B's TABLE

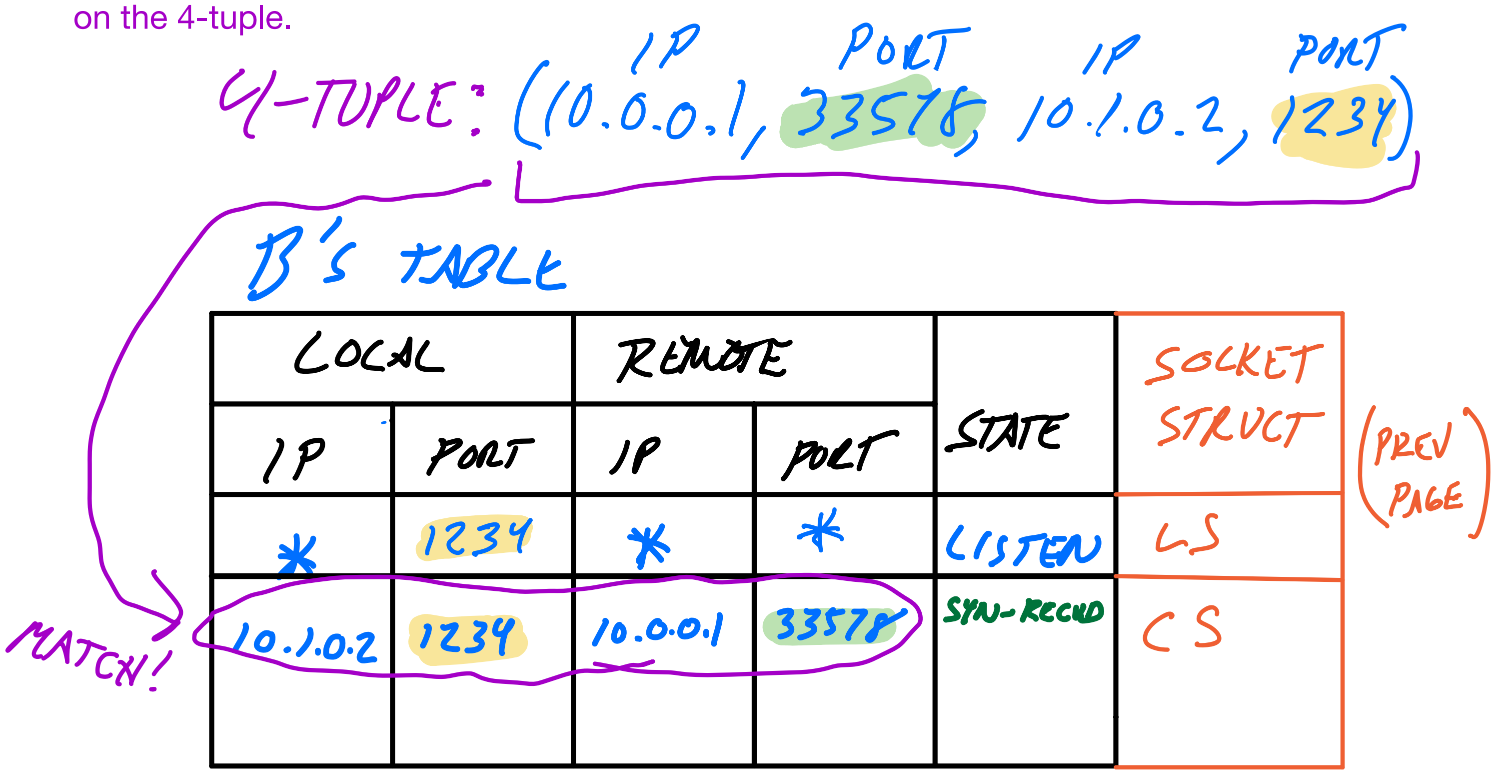
	LOCAL		REMOTE		STATE
	IP	PORT	IP	PORT	
1	*	1234	*	*	LISTEN
3	10.1.0.2	1234	10.0.0.1	33578	~~SYN_RECVD~~ ESTAB (6)

How to know it goes to this specific socket, and not the listen socket? See next page.

How do we map an incoming packet to a socket? To take a look at this, let's examine what happens to the last packet in the handshake when it's received by B (step 5 above):



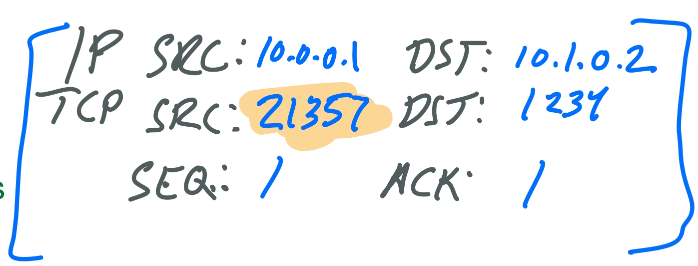
The packet's source/dest IP and port numbers act like a unique identifier that identifies this connection => **this is called the 4-tuple** We map packets to normal sockets based on the 4-tuple.



To summarize, here's how the matching process works.
When receiving packet P, check the socket table for a matching socket:

1. Check for a normal socket with a matching 4-tuple, e.g., (dstIP, dstPort, srcIP, srcPort)
2. If there is no matching normal socket, check for a listen socket where localPort == P.dstPort
3. If no match, this packet isn't for any known socket, so drop the packet.

Another example: What if we received a different packet that looked like this?



This packet has a different source port, so it has a different 4-tuple! Therefore, it must be for another connection (or it's an attempt to start a new one.

=> Thus, this packet should map to the **listen socket**

Most important socket API calls for setting up connections

(I.E. MILESTONE I) :)

VListen
- "Passive OPEN" in RFC
- "I want new hosts to connect to me on this port"

=> Returns a listen socket

VAccept
- Input: a listen socket
- Block until a client has connected and this new client connection is in the ESTABLISHED state
  - => SO you can't send/recv until you're in ESTABLISHED

=> Returns a normal socket

VConnect
- Initiate connection to given IP and port
- "Active OPEN" in RFC
- Block until connection established, or until abort
- Returns a normal socket we can use

Connection setup API: recap

VConnect

- "Active OPEN" in RFC
- Initiates new connection, returns normal socket
- Blocks until connection is established, or times out

VListen

- "Passive OPEN" in RFC
- Returns new listen socket

VAccept

- Input: a listen socket
- Blocks until a client connection is established
- Returns new normal socket

How exactly you implement this is up to you, but your API should have calls like this
(This isn't arbitrary—it matches what the kernel API looks like)

Think back to your Snowcast server…

```
// Create listen socket (bind)
listenConn, err := net.ListenTCP("tcp4", addr)

for {
    // Wait for a client to connect
    clientConn, err := listenConn.Accept()
    if err != nil {
        // . . .
    }

    // . . .
    go handleClient(clientConn)

}

func handleClient (conn net.Conn) {
    conn.Read(. . .)
}
```

Listen socket

"Normal" socket

Why separate listen and accept?
=> Need to be able to handle multiple client connections!

Your "a" command will look similar…

```
func ACommandREPL() { // Runs as separate thread/goroutine

    // Create listen socket (bind)
    listenConn, err := tcpstack.VListen(port)

    for {
        // Wait for a client to connect
        clientConn, err := listenConn.VAccept()
        if err != nil {
            // . . .
        }

        // Store clientConn to use by other REPL commands
    }
}
```

Summary: two types of sockets

Type	When created	What it does	What's in it?*
Listen sockets	"a" command (VListen) => VTCPListener in API example	• "I want to receive new connections on this port" • Always in state LISTEN • Not connected to another endpoint! (can't send/recv on it, has no TCB	• List of sockets for new/pending connections
"Normal" sockets	"c" command (VConnect) "a" command (VAccept) => VCTPConn in API example	• Used for "normal" TCP connections between endpoints	• TCB (send/recv buffers, all other TCP protocol state)

*: At minimum, for now

Implementation stuff

Ways to build the API

More info: "Socket API example" in docs

```
conn, err := tcpstack.VConnect(addr, port)
. . .
conn.VWrite(someBuf)
```

Go-style

- VConnect/VCccept/VListen return structs for normal/listen sockets
- Other functions (VAccept, VWrite, …) are methods on these structs

```
int sock_fd = VConnect(addr, port)
. . .
VWrite(sock_fd, some_buffer)
```

C-style

- VConnect/VCccept/VListen return numbers (like file descriptors)
- Other functions (VAccept, VRead, …) take socket number as argument

Ways to build the API

More info: "Socket API example" in docs

```
conn, err := tcpstack.VConnect(addr, port)
. . .
conn.VWrite(someBuf)
```

Go-style

- VConnect/VCccept/VListen return structs for normal/listen sockets
- Other functions (VAccept, VWrite, …) are methods on these structs
- In REPL: map socket ID => struct

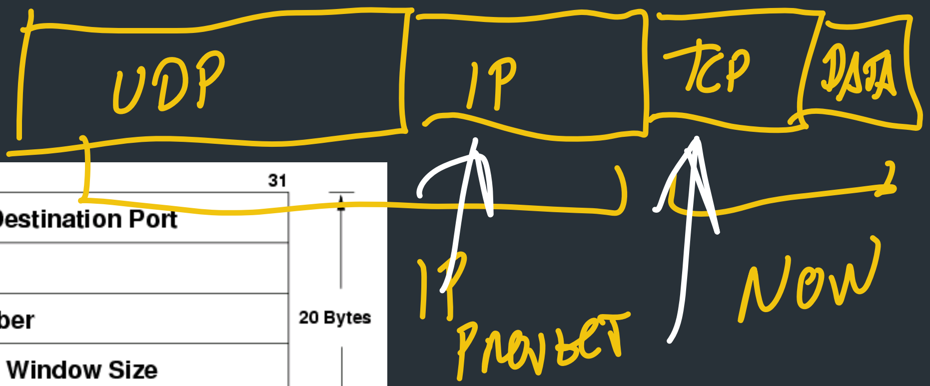
```
int sock_fd = VConnect(addr, port)
. . .
VWrite(sock_fd, some_buffer)
```

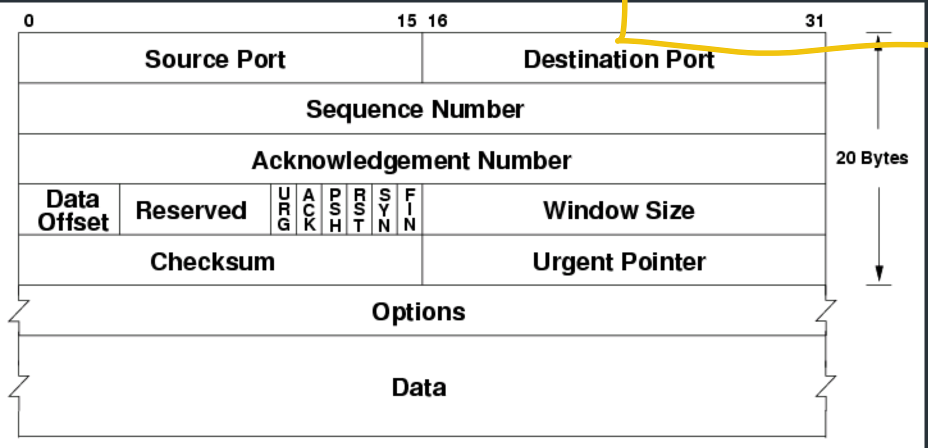
C-style

- VConnect/VCccept/VListen return numbers (like file descriptors)
- Other functions (VAccept, VRead, …) take socket number as argument
- In TCP stack: map socket ID => struct

=> How you implement this is up to you (don't even need to pick one of these)!

Building TCP packets



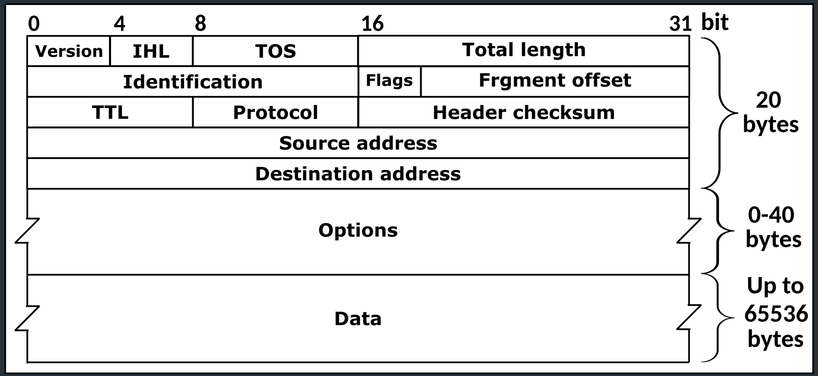


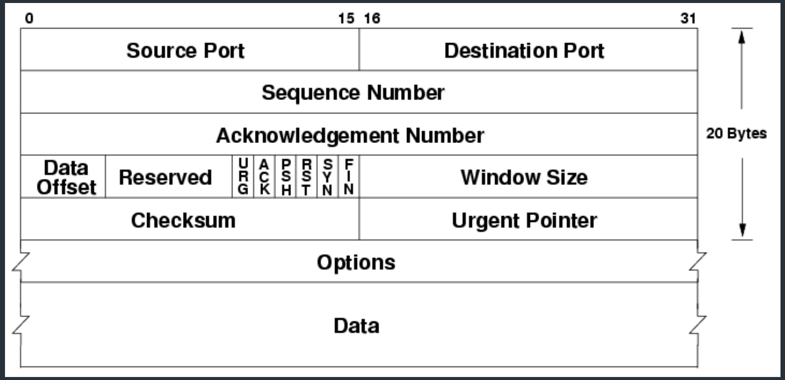
- MUST use standard TCP header
- Encapsulation: TCP packet => payload of virtual IP packet
- Once again, you don't need to build/parse this yourself

⇒ See the TCP-in-IP example for a demo on how to build/parse a TCP header (mostly uses same libraries as before)

The TCP checksum

… is pretty weird





Computing the TCP checksum involves making a "pesudo-header" out of some IP and TCP 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

⇒ You don't need this working for milestone 1

⇒ 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 switched to a new reference last 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

<u>Milestone I</u>

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

Be prepared to talk about what goes in your data structures, design plan, etc, similar to your IP milestone

Roadmap

Milestone II

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

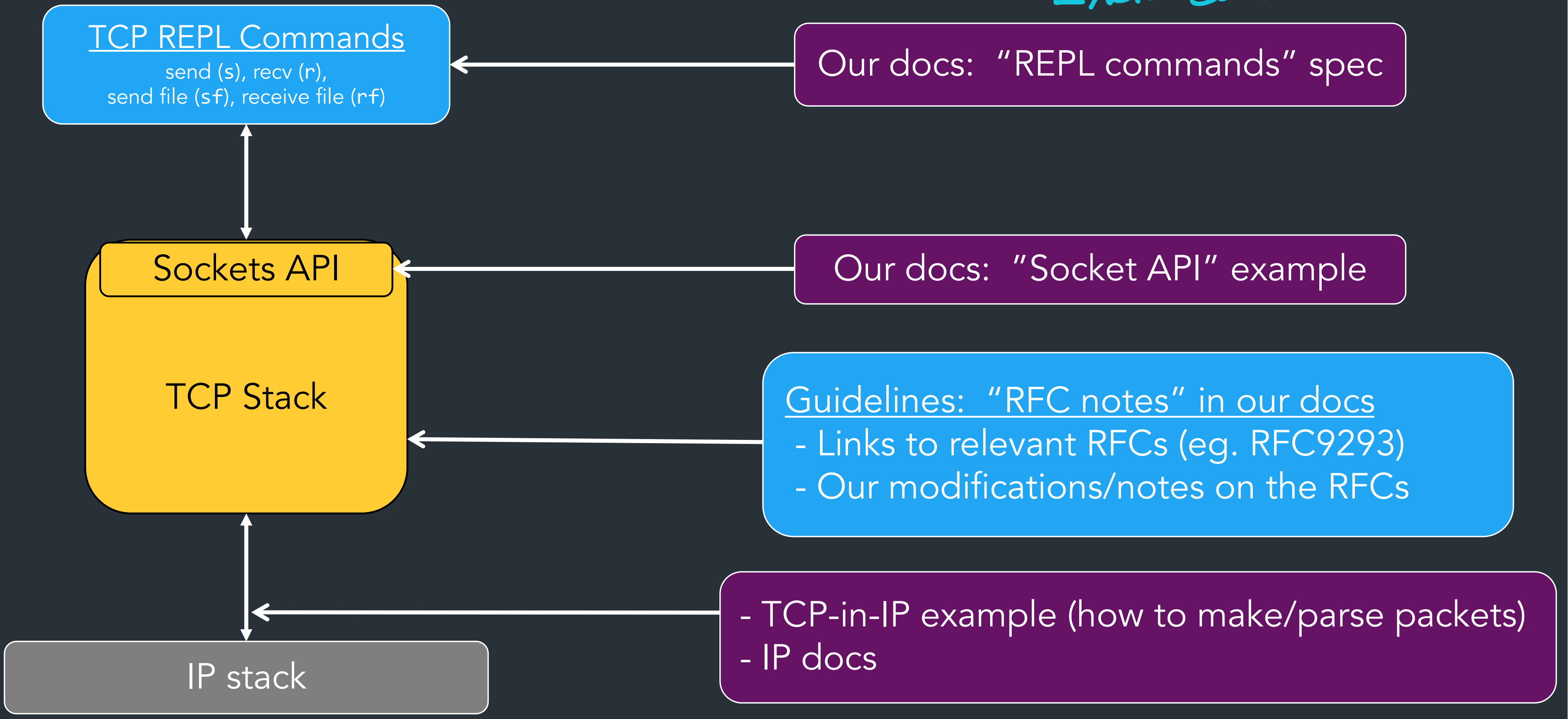
Roadmap

Final deadline

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

Where to get more info

WE HAVE DOCS FOR EACH COMPONENT!



Closing thoughts

- Use your milestone time wisely!
- Wireshark is the best way to test—use it!
- As you work with your IP code, consider refactoring!
  - You're going to be working with this code for >= 3 weeks
- Stuck? Don't know what's required? Just ask!
  (And see Ed FAQ)

We are here to help!