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

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 (10/24): 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!

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
- When you get an ACK, reset on ACK

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.)

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

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

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"):

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

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

α = "Smoothing factor": .8-.9

β = "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

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

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

How to test?

- Don’t leave sendfile/recv file to the end--try to test as you go
  - You WILL find bugs. Breathe. It’s going to be okay.

- Try to test each part individually, as shown here

How to test?

- Don’t leave sendfile/recv file to the end--try to test as you go
  - You WILL find bugs. Breathe. It’s going to be okay.

- Try to test each part individually, as shown here

Don’t be afraid to write some test code!
 => Eg. To test retransmissions, comment out your ACK processing and see what happens

Testing with packet loss

New REPL command in vrouter reference (out soon):

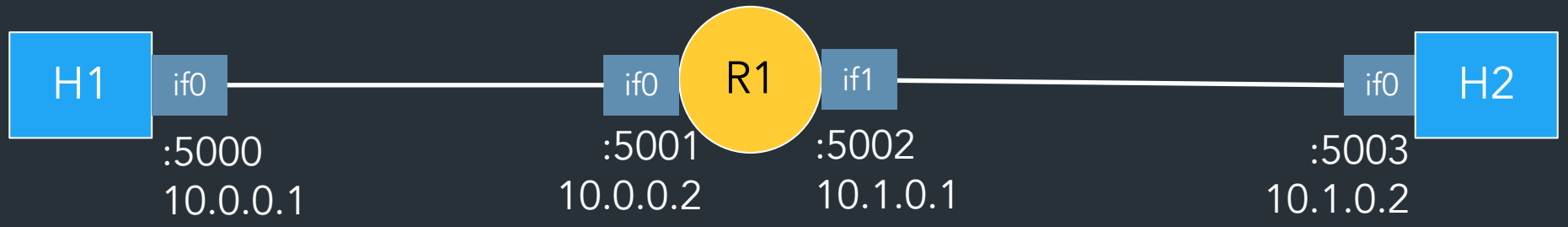
```
> drop 0.01   // Drop 1% of packets
> drop 0.5    // Drop 50% of packets (way too aggressive)



> 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!

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!

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!)

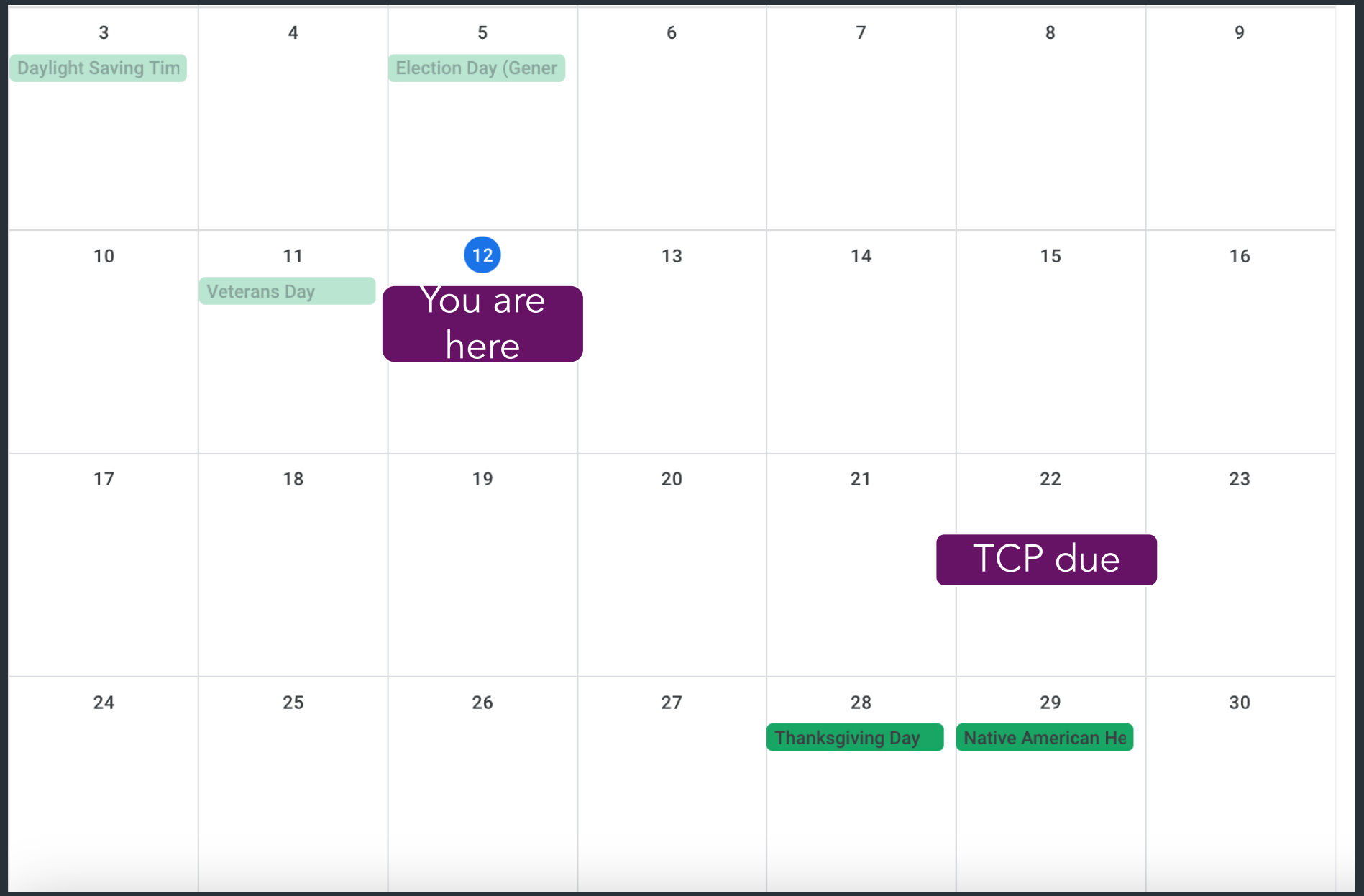
Closing thoughts

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

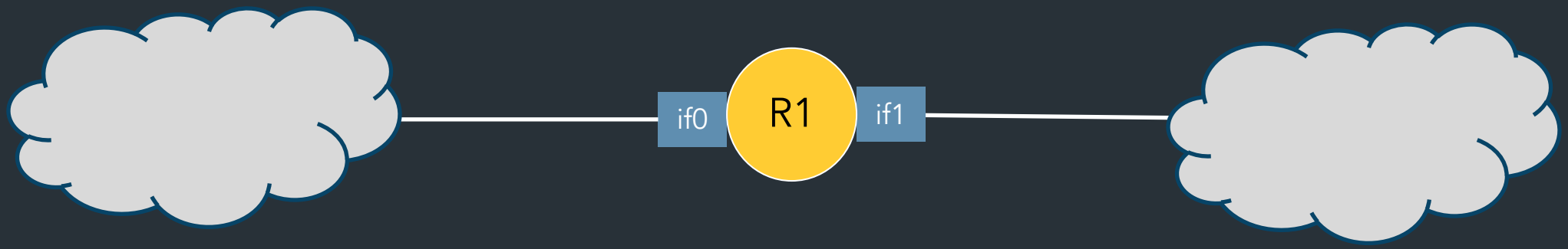
When stuck, take a break and come back to it. It will help.

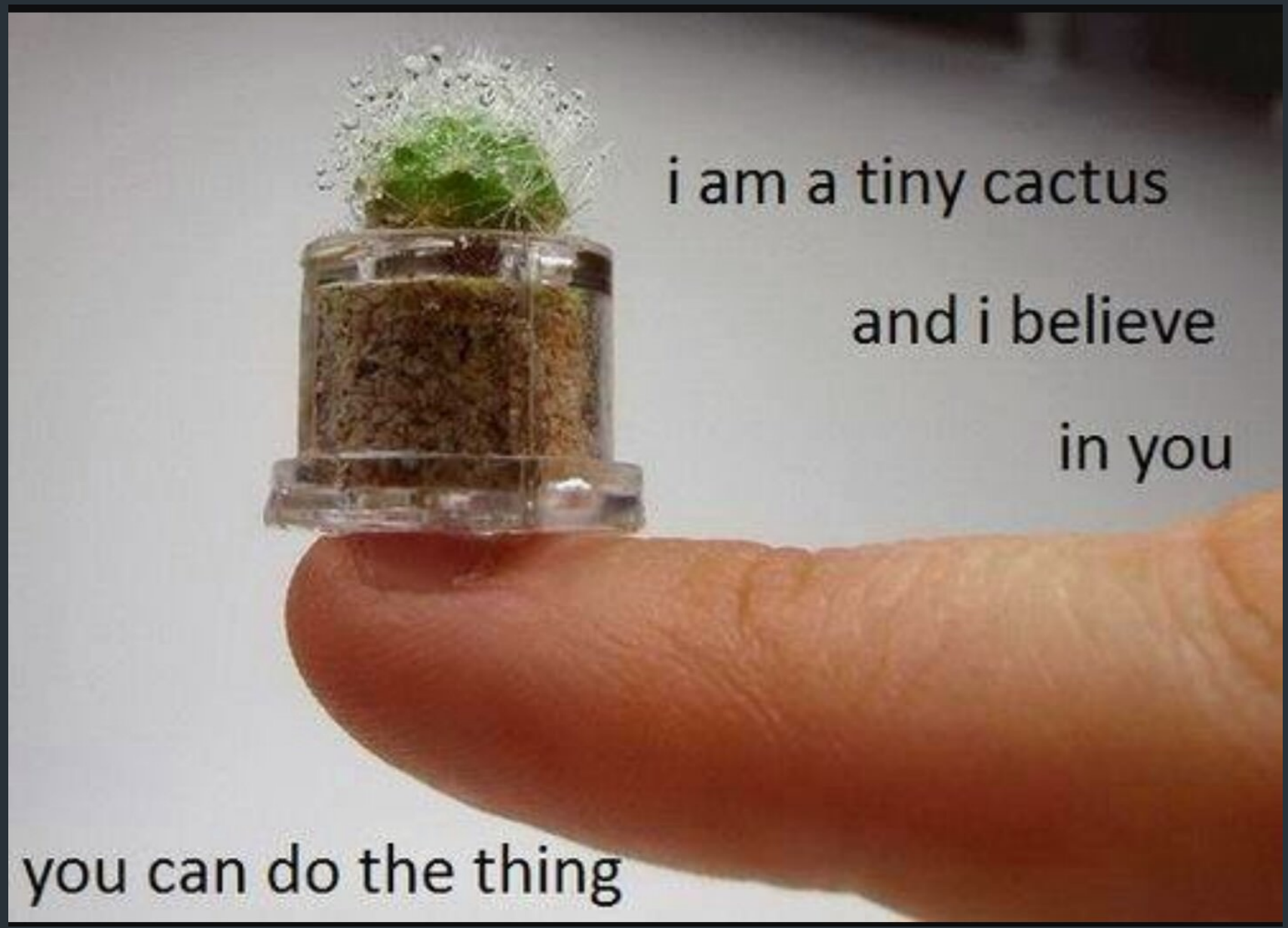
=> Do NOT wait until the last minute.

Don’t panic.


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

One more thing…

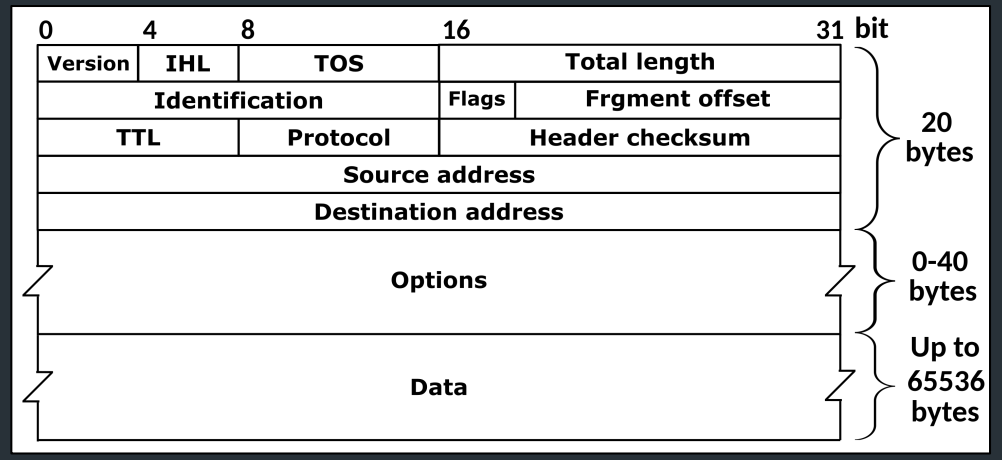



i am a tiny cactus
and i believe
in you
you can do the thing

The TCP checksum

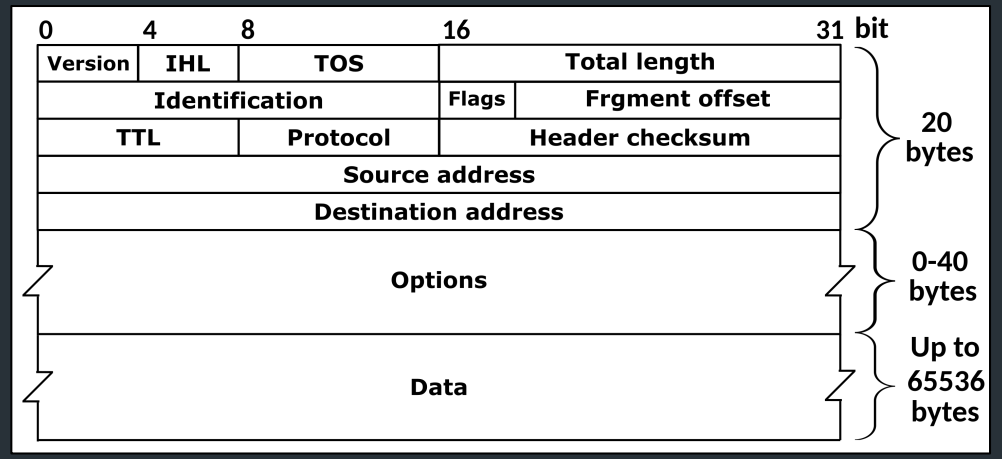
… is pretty weird

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



The TCP checksum

… is pretty weird



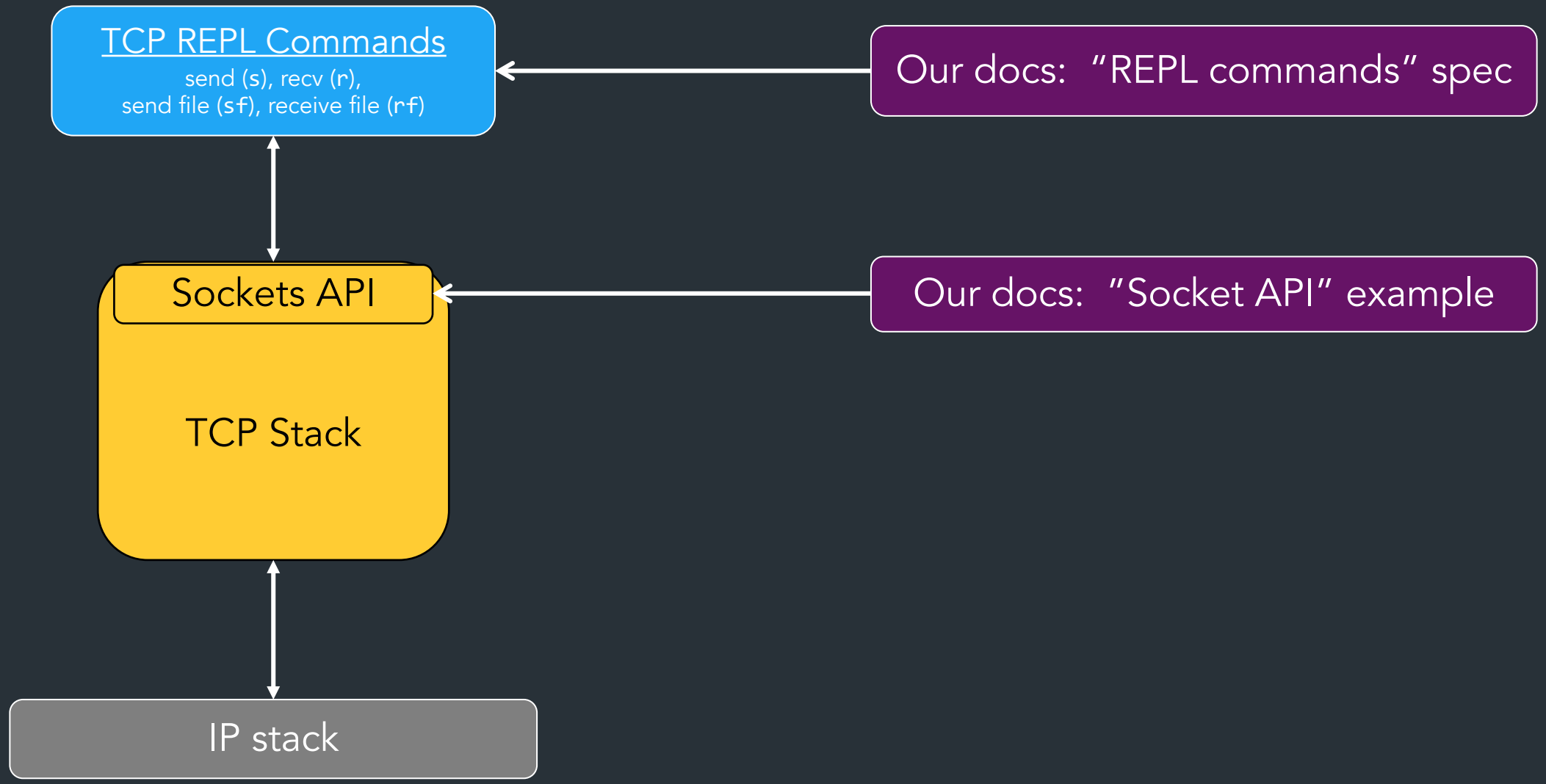
Computing the TCP checksum involves making a "pesudo-header" from TCP header + IP header fields:

TCP pseudo-header for checksum computation (IPv4)

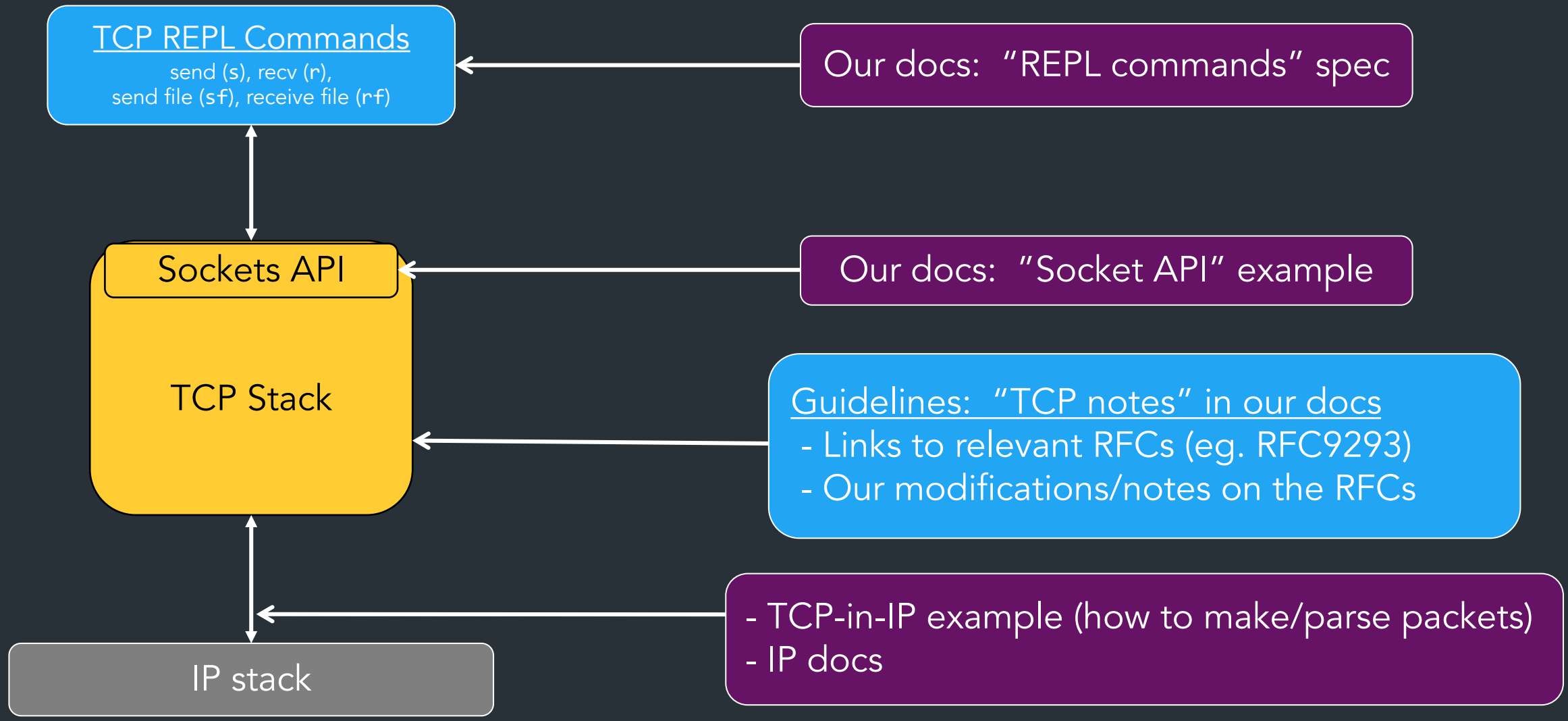
Bit offset	0–3	4–7	8–15	16–31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			
160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

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

Where to get more info



Where to get more info



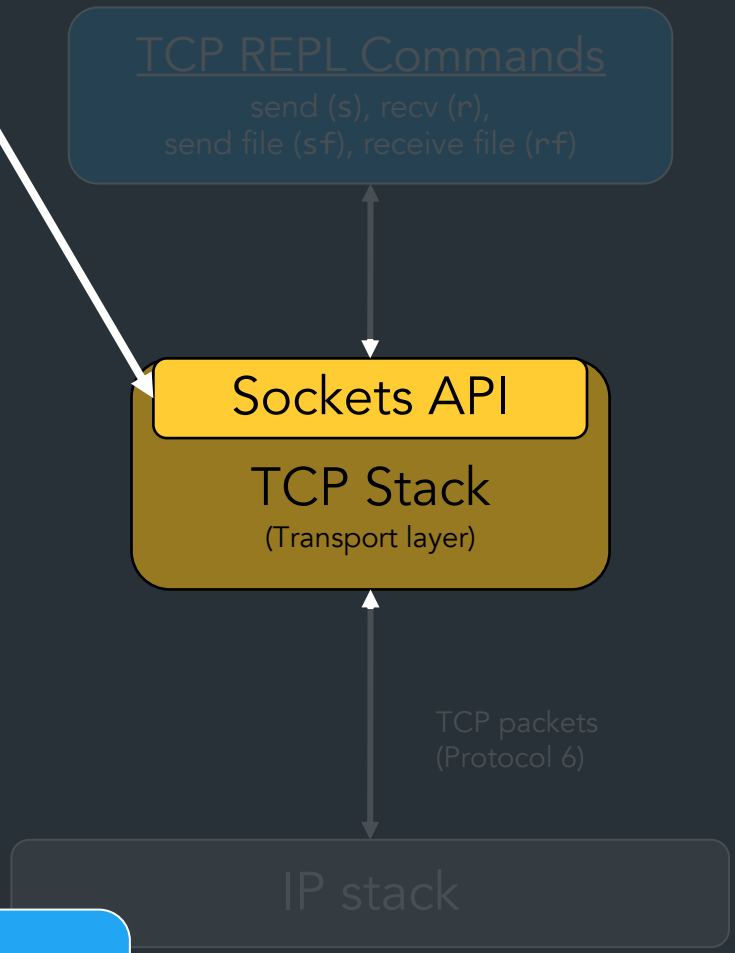
API for sockets: abstraction for creating and using TCP connections

Example: Go's socket API

```
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!)

```
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
```

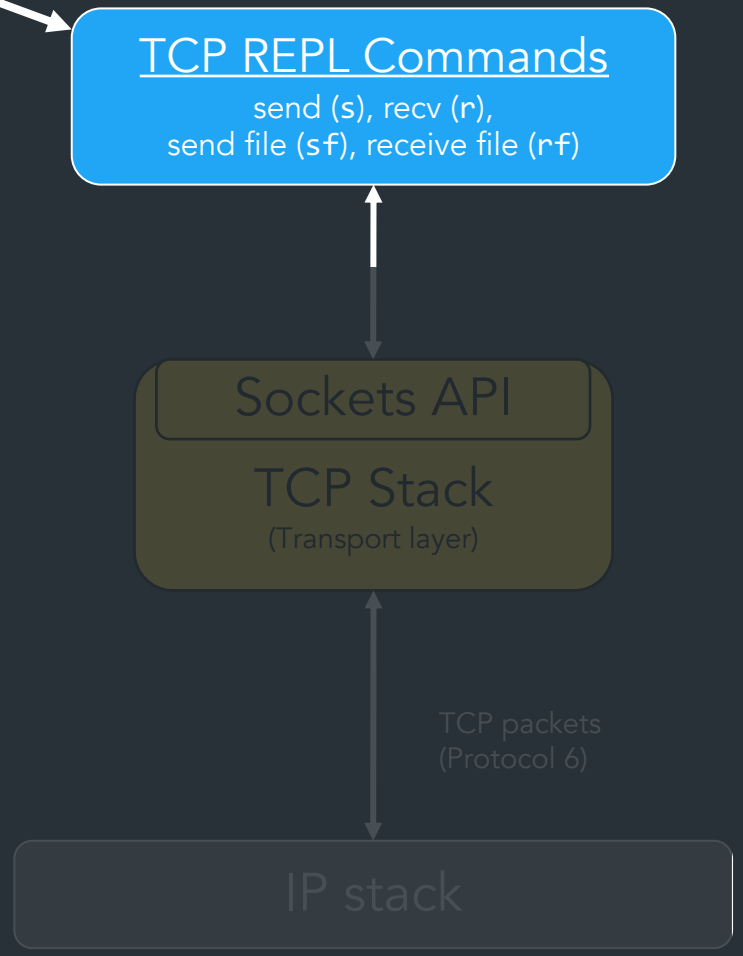
Guidelines: “Socket API” specification in docs

REPL commands: how we'll test your socket API

=> 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



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
    }
}
```

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

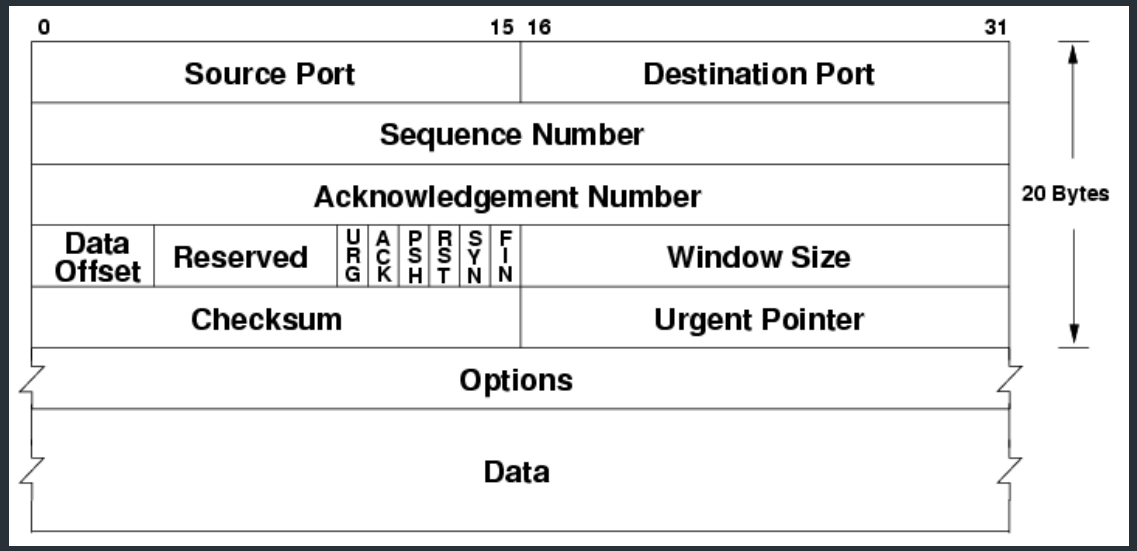
```
int sock_fd = VConnect(addr, port)
. . .
VWrite(sock_fd, some_buffer)
```

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)

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!

Proto	Local (yours)		Remote (theirs)		Socket
	IP	Port	IP	Port	
tcp	1.2.3.4	12345	5.6.7.8	80	(normal struct)
tcp	*	22	*	*	(listen struct)
…	…	…	…	…	…

Key: 5-tuple of (local IP, local port, remote IP, remote port, protocol)

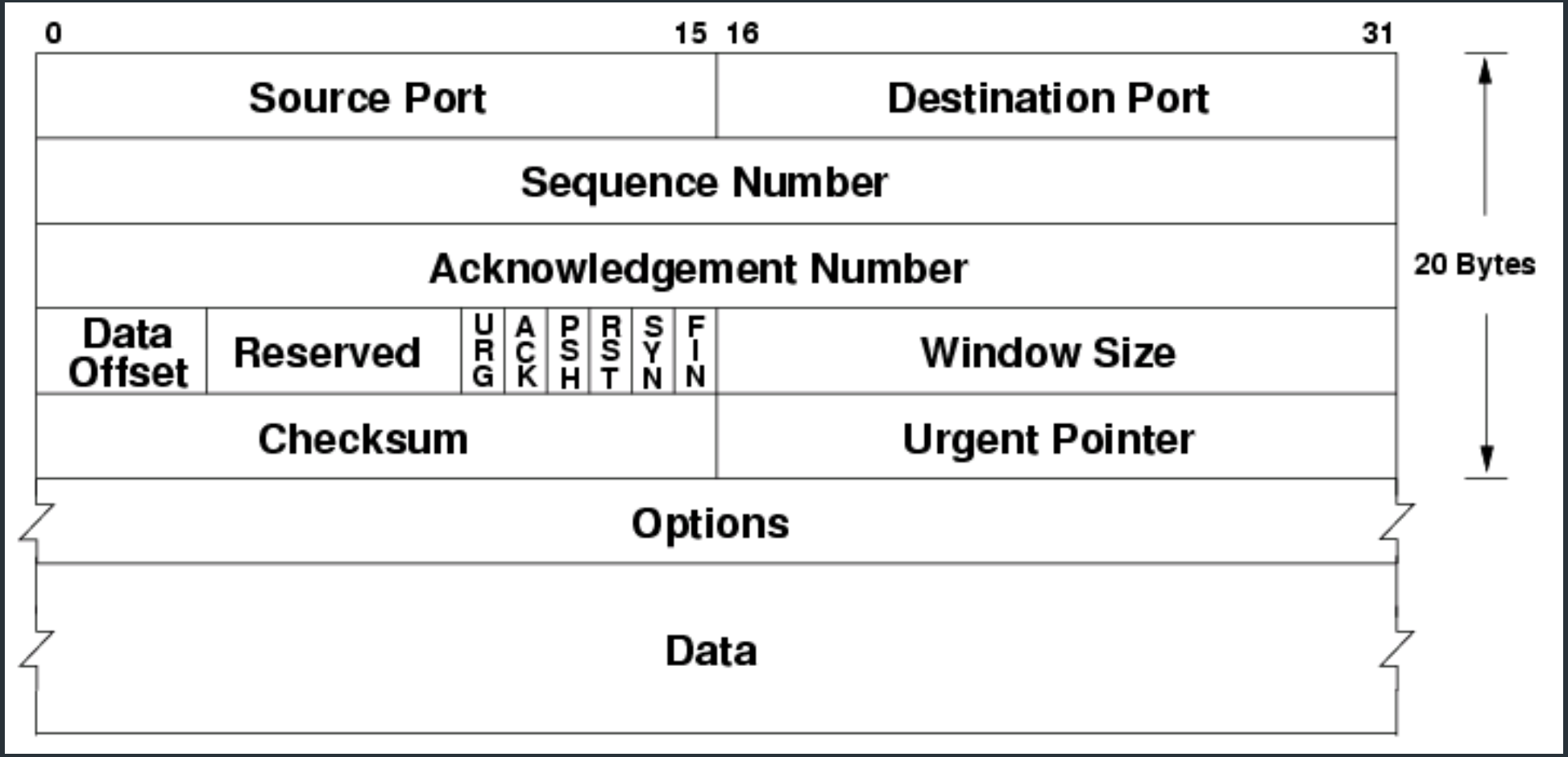
Value: info about a socket
(state, buffers, …)

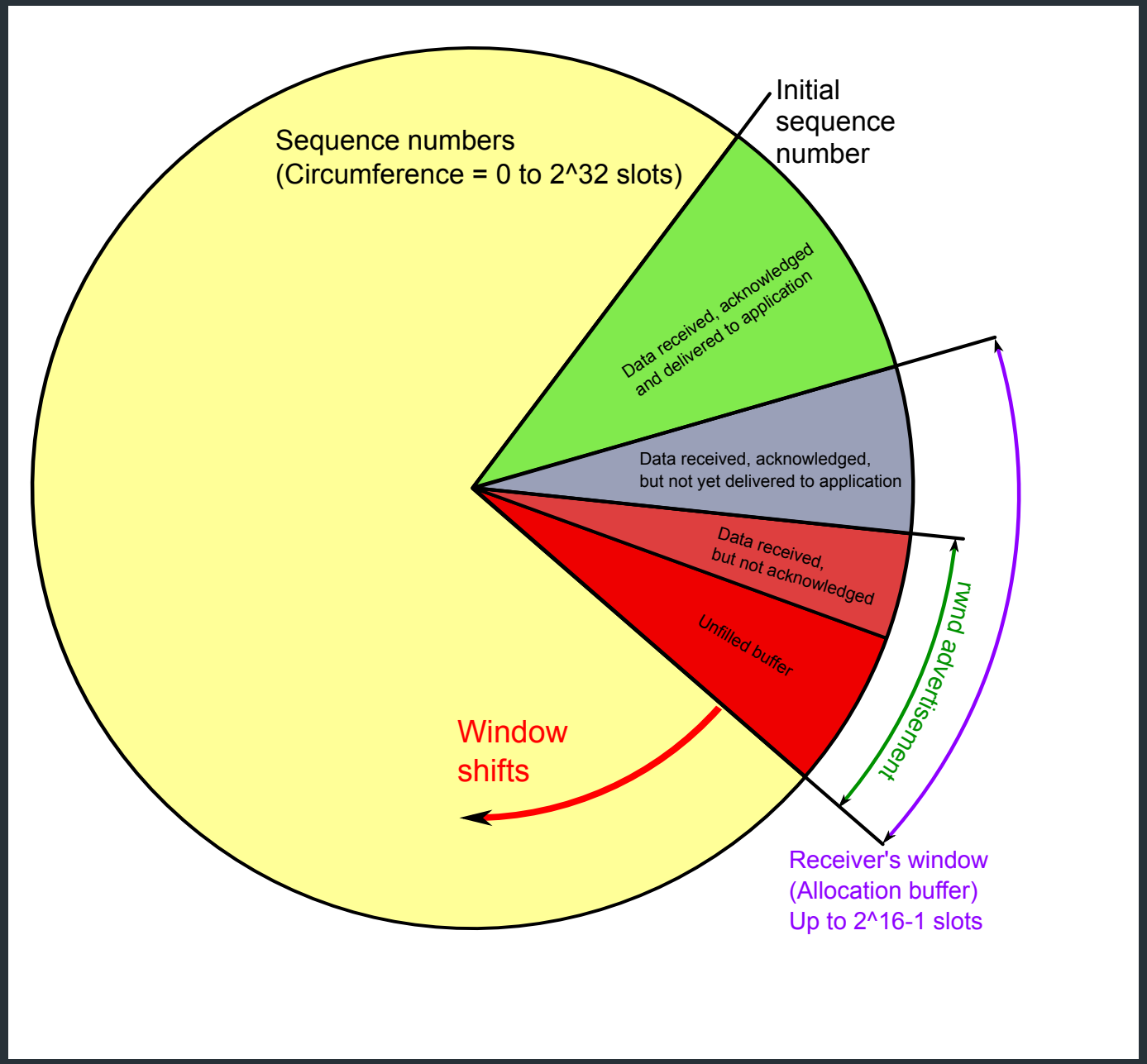
When you receive a TCP packet

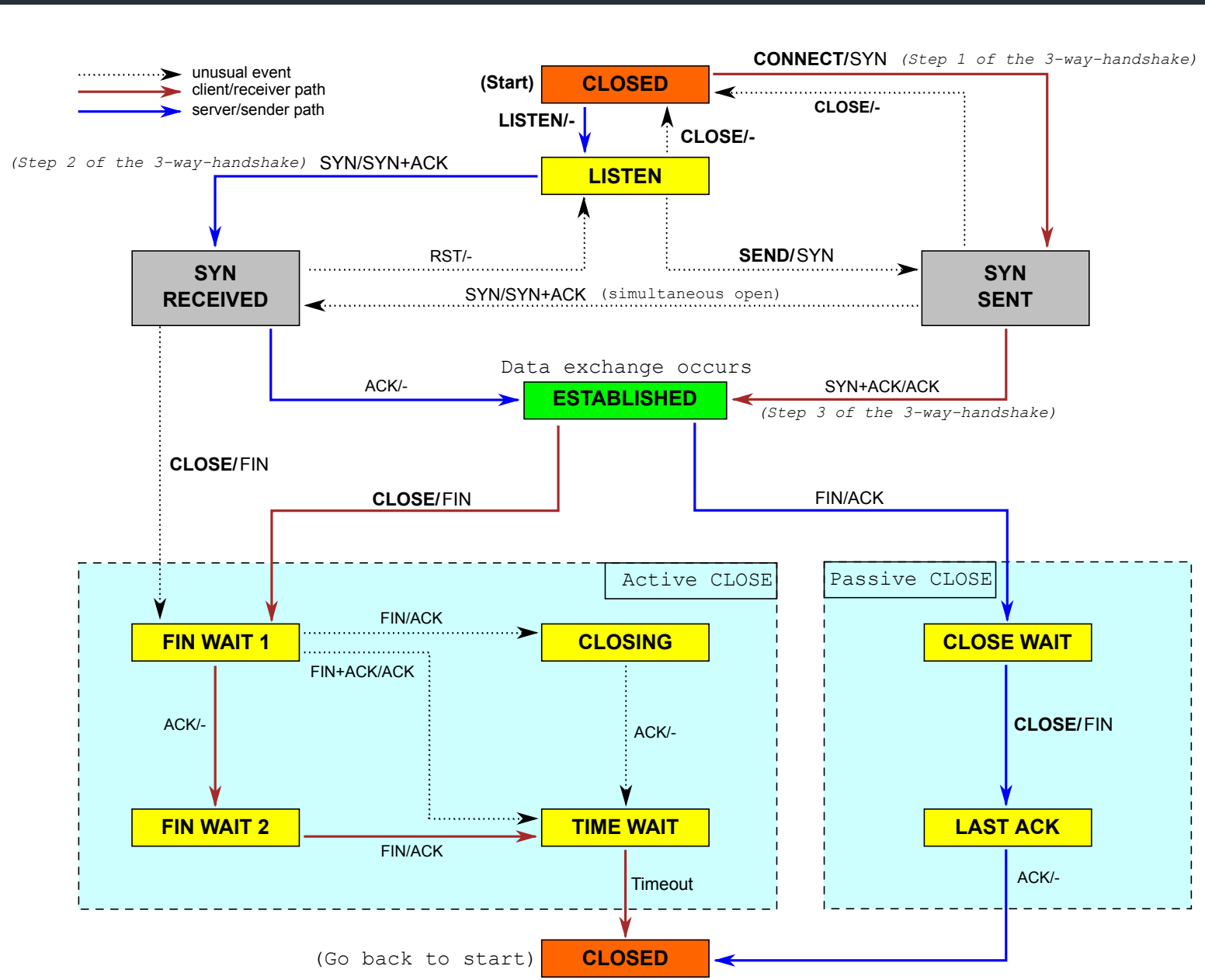
- First, check for a match on the 5-tuple
- Then, check for any open listen sockets

```
> 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
  2   10.1.0.2  9999   10.0.0.1 23234  ESTABLISHED
  3   10.1.0.2  9999   10.0.0.1 55434  ESTABLISHED
```

TCP Header


0
15
16
31
Source Port
Destination Port
Sequence Number
Acknowledgement Number
Data Offset
Reserved
URG
ACK
PSH
RST
SYN
FIN
Window Size
Checksum
Urgent Pointer
20 Bytes
Options
Data


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
FIN+ACK/ACK
CLOSE WAIT
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

Sample Topologies

Some example networks you can test with…

See “sample networks” page for more info, including what kinds of things you can test with each network

```
// Our example API (sending side)
addr, err := netip.ParseAddr("1.2.3.4")
. . .
conn, err := ipstack.VConnect(addr)
. . .

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

```
// Our example API (receiving side)
listenConn, err := ipstack.VListen(9999) // Listen on

. . .


clientConn, err := listenConn.VAccept()
clientConn.VRead(someBuf)
clientConn.Vclose()
```

=> This is not the only way to do the API, more on this later

Guidelines: "Socket API" specification in docs