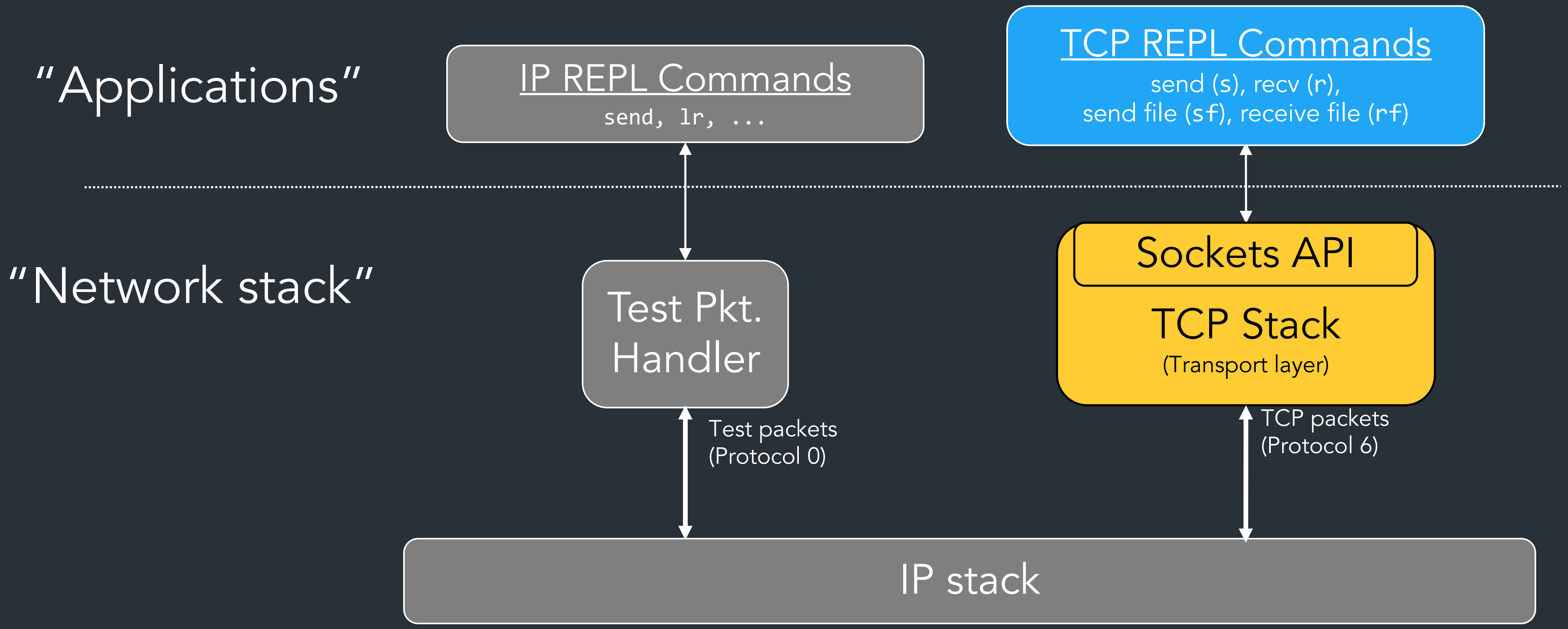
TCP Gearup I

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

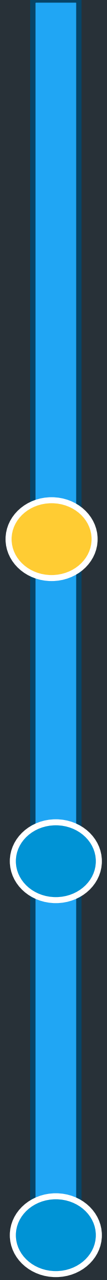
(Transport layer)

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

Key resources

- Lecture 14: Send/recv basics
- Lecture 15: 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" and "Testing with Wireshark" in the docs

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

- Should 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
- You MAY use a library, but you should decide if this is what you want

For detailed info
=> RFC9293 Sec 3.3: what all the variables mean
⇒ Lecture 15: 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

Performance requirement: send/recv process **`MUST`** be event driven

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

Implementing Vread/VWrite

Performance requirement: send/recv process `MUST` be event driven

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

Where does this apply?

- REPL: s, r, sf, rf
- VRead/VWrite
- Deciding when to send, or check for new data
- Retransmissions

Implementing VRead/VWrite

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

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

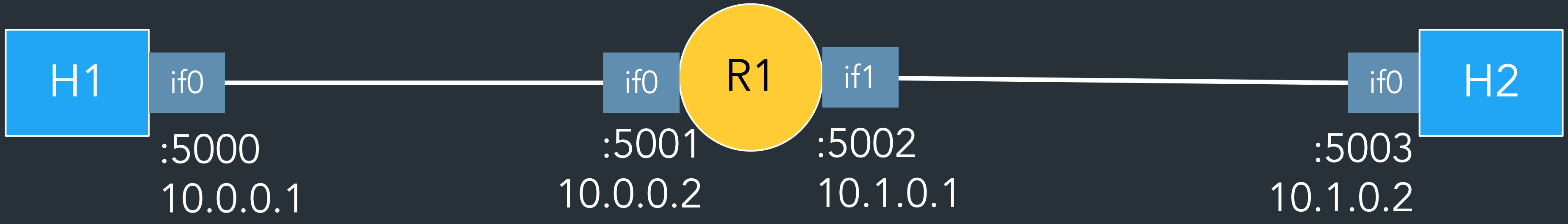
Where does this apply?

- REPL: s, r, sf, rf
- VRead/VWrite
- Deciding when to send, or check for new data
- Retransmissions

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

Channels?

How to test TCP



Useful wireshark mechanics

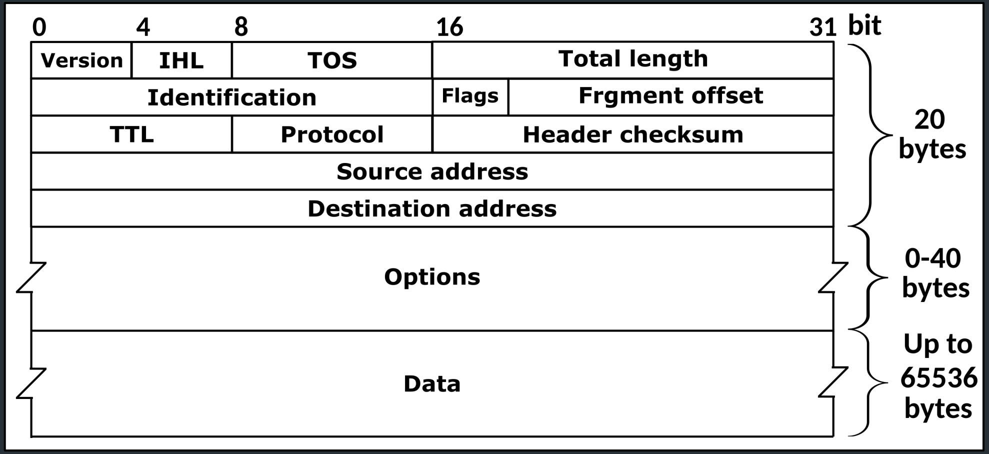
- SEQ/ACK analysis
- Follow TCP stream
- Validating the checksum

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

The TCP checksum

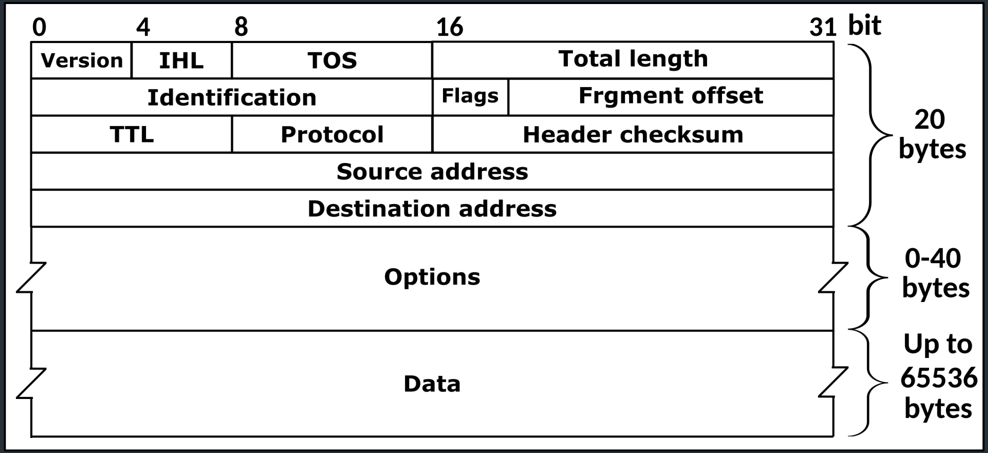
… is pretty weird

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



The TCP checksum

… is pretty weird



0 4 8 16 31 bit

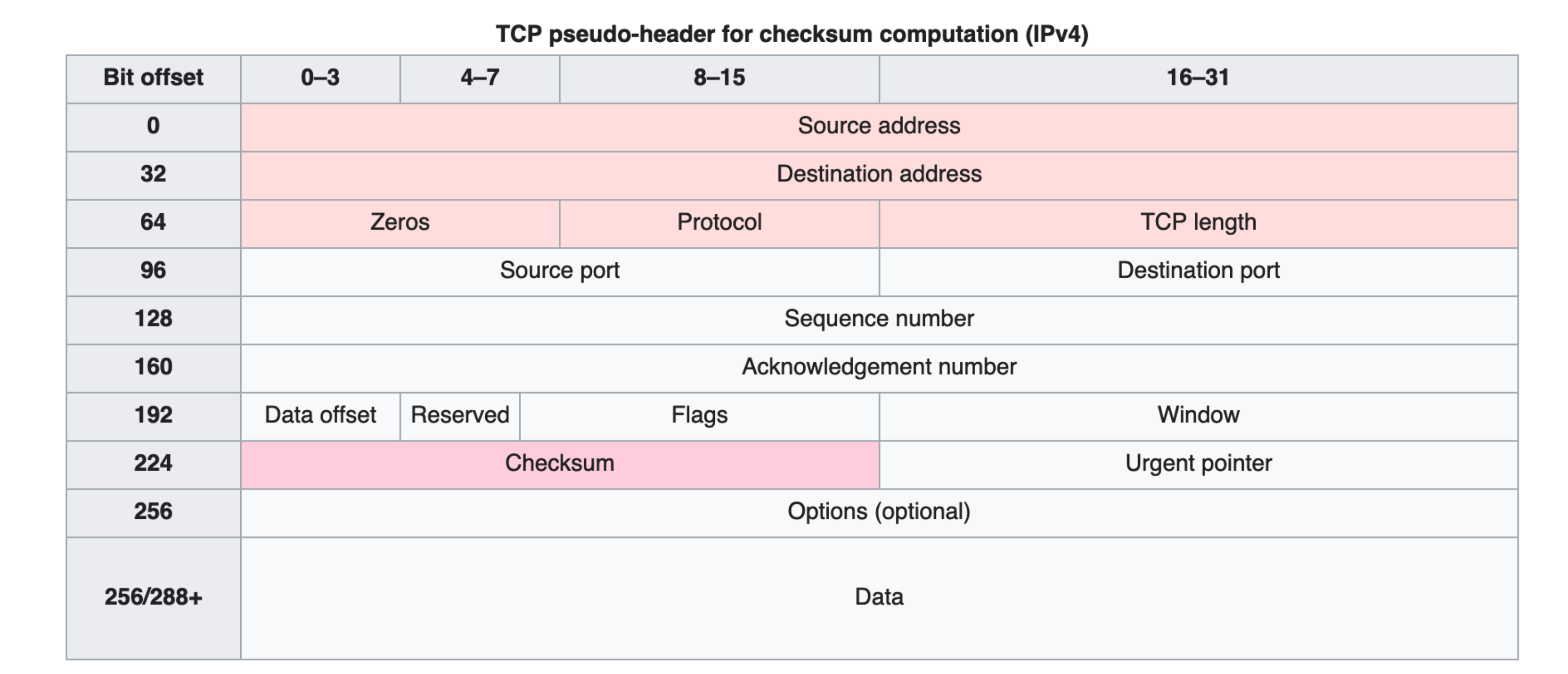
Version	IHL	TOS	Total length
Identification		Flags	Frgment offset
TTL		Protocol	Header checksum
Source address			
Destination address			
Options			
Data			

20 bytes

0-40 bytes

Up to 65536 bytes

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!

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!)

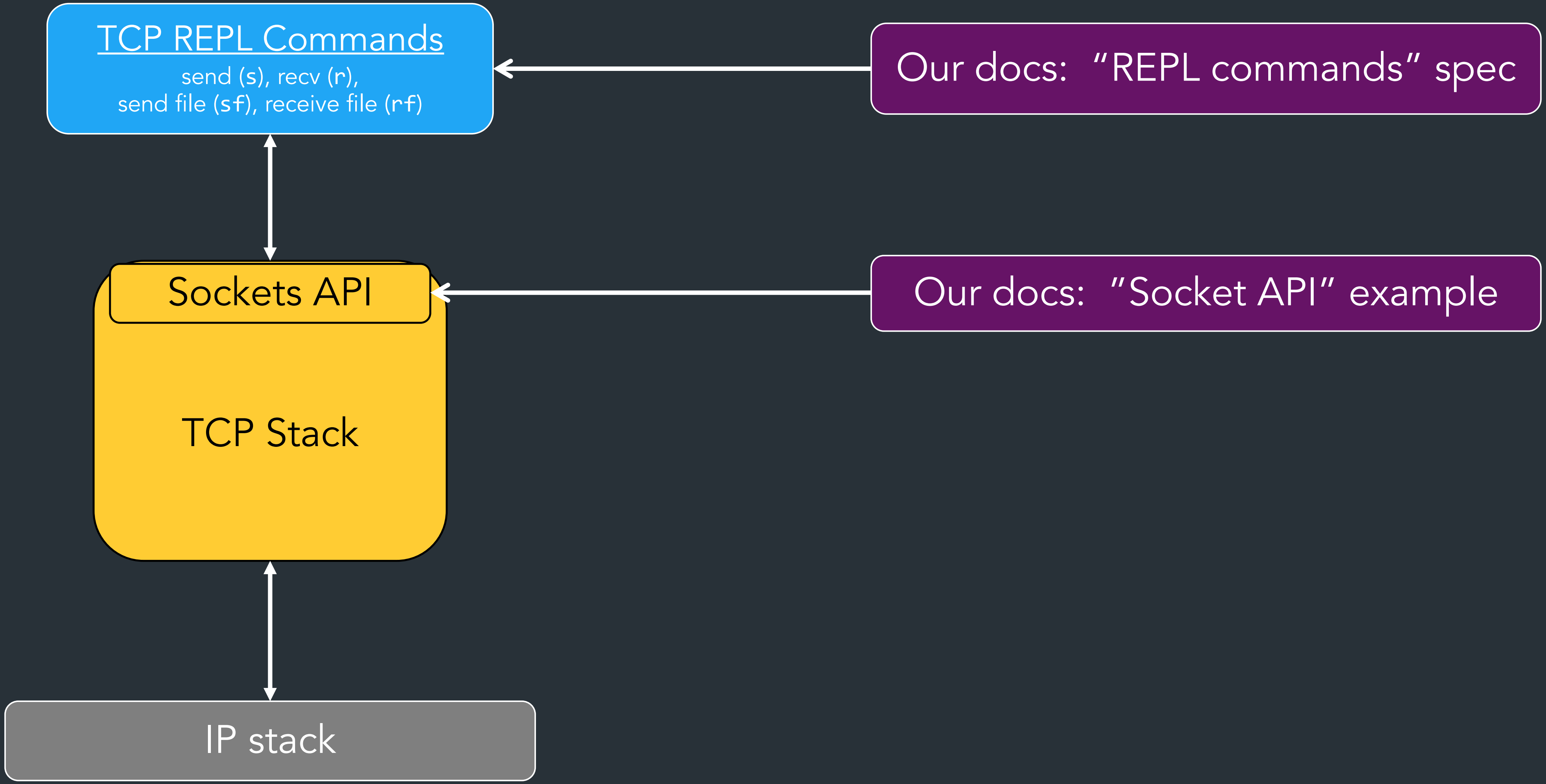
Custom vnet_run configurations

Roadmap

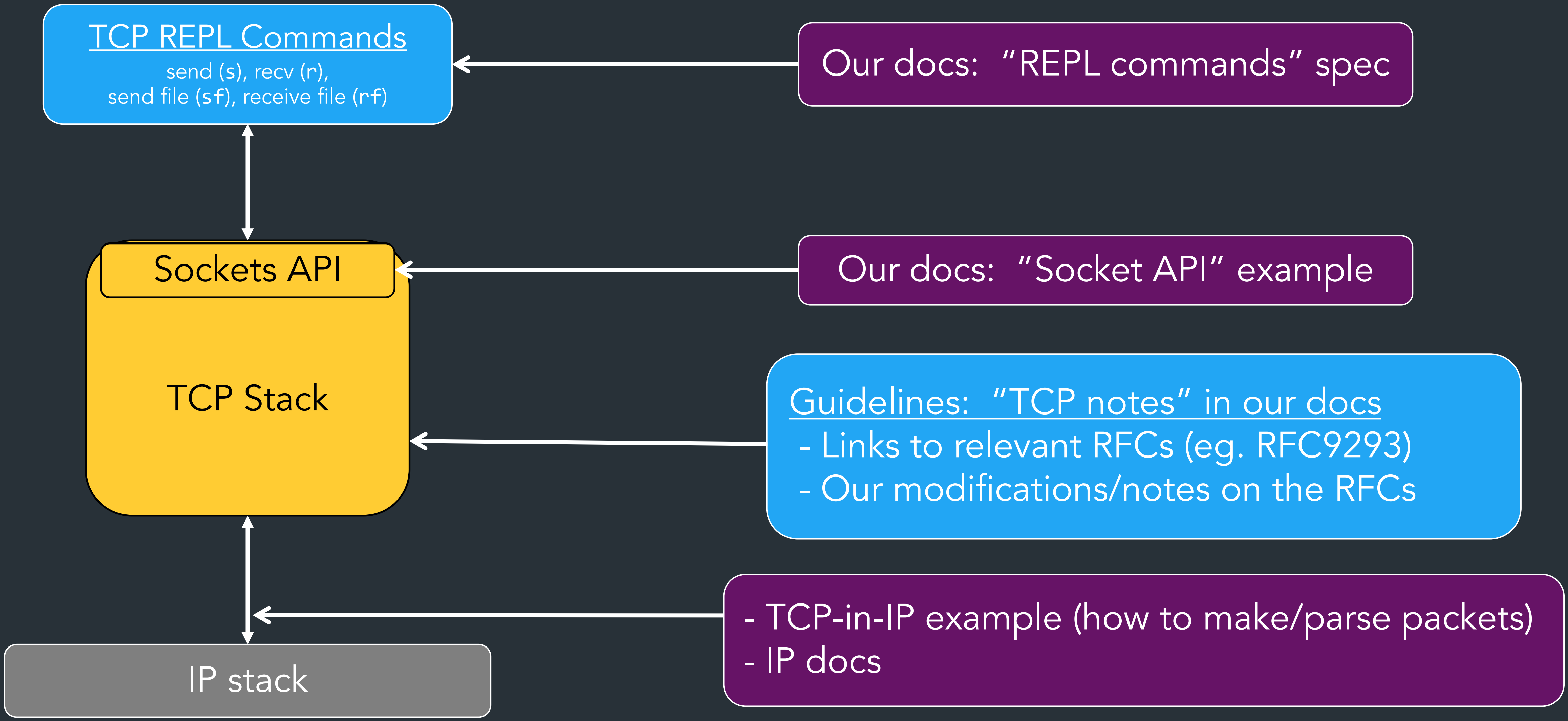
Final deadline

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

Where to get more info


TCP REPL Commands
send (s), recv (r),
send file (sf), receive file (rf)
Our docs: "REPL commands" spec
Sockets API
Our docs: "Socket API" example
TCP Stack
IP stack

Where to get more info



TCP REPL Commands

send (s), recv (r),
send file (sf), receive file (rf)

Sockets API

TCP Stack

IP stack

Our docs: "REPL commands" spec

Our docs: "Socket API" example

Guidelines: "TCP notes" in our docs

- Links to relevant RFCs (eg. RFC9293)
- Our modifications/notes on the RFCs

- TCP-in-IP example (how to make/parse packets)
- IP docs

API for sockets: abstraction for creating and using TCP connections

Example: Go’s socket API

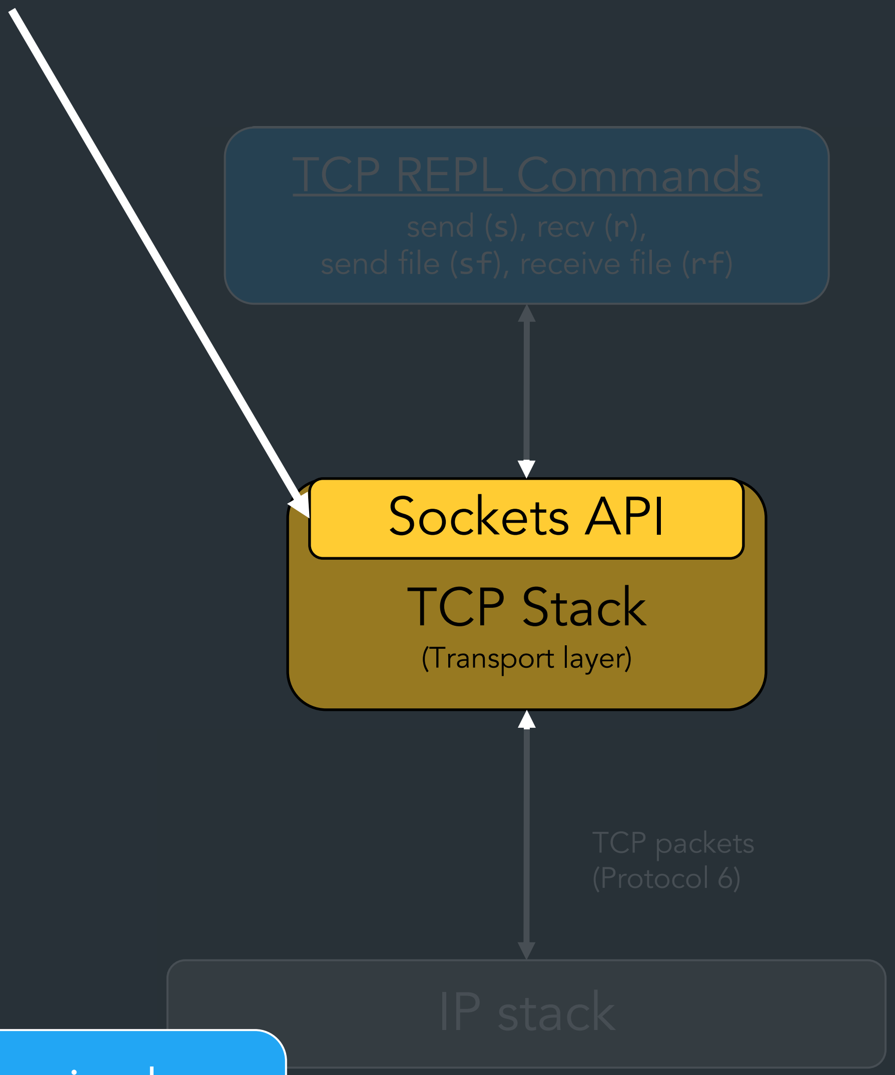
```
conn, err :=  net.Dial(“tcp”, “10.0.0.1:80”)
. . .

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

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

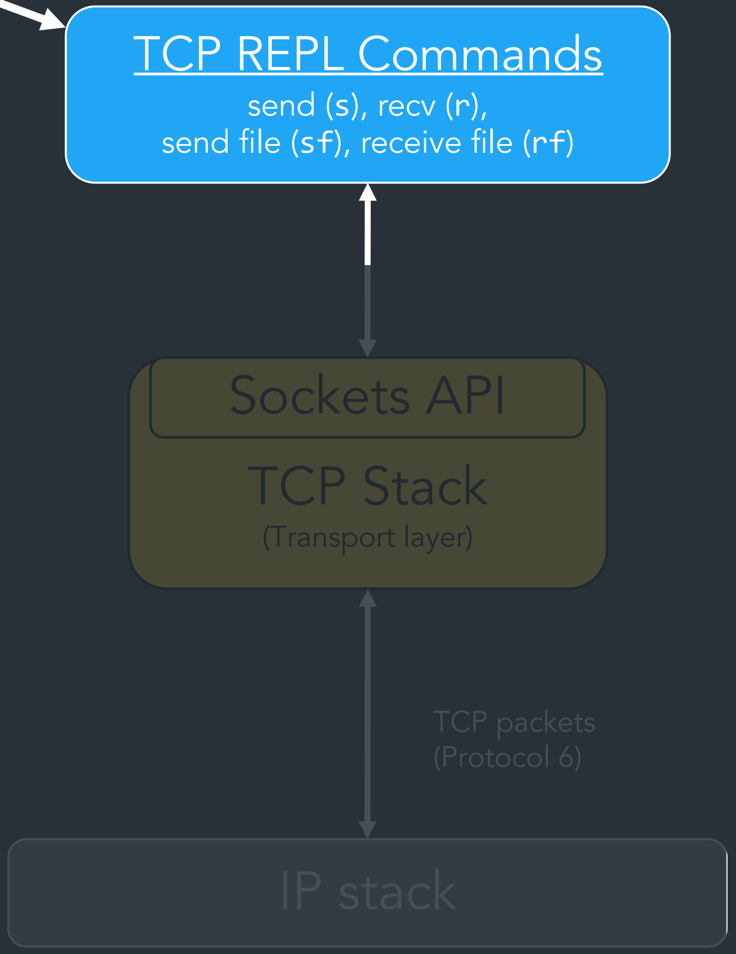
```
// Basic stuff (test your API)
a Listen on a port; accept new connections
c Connect to a TCP socket
ls List sockets

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

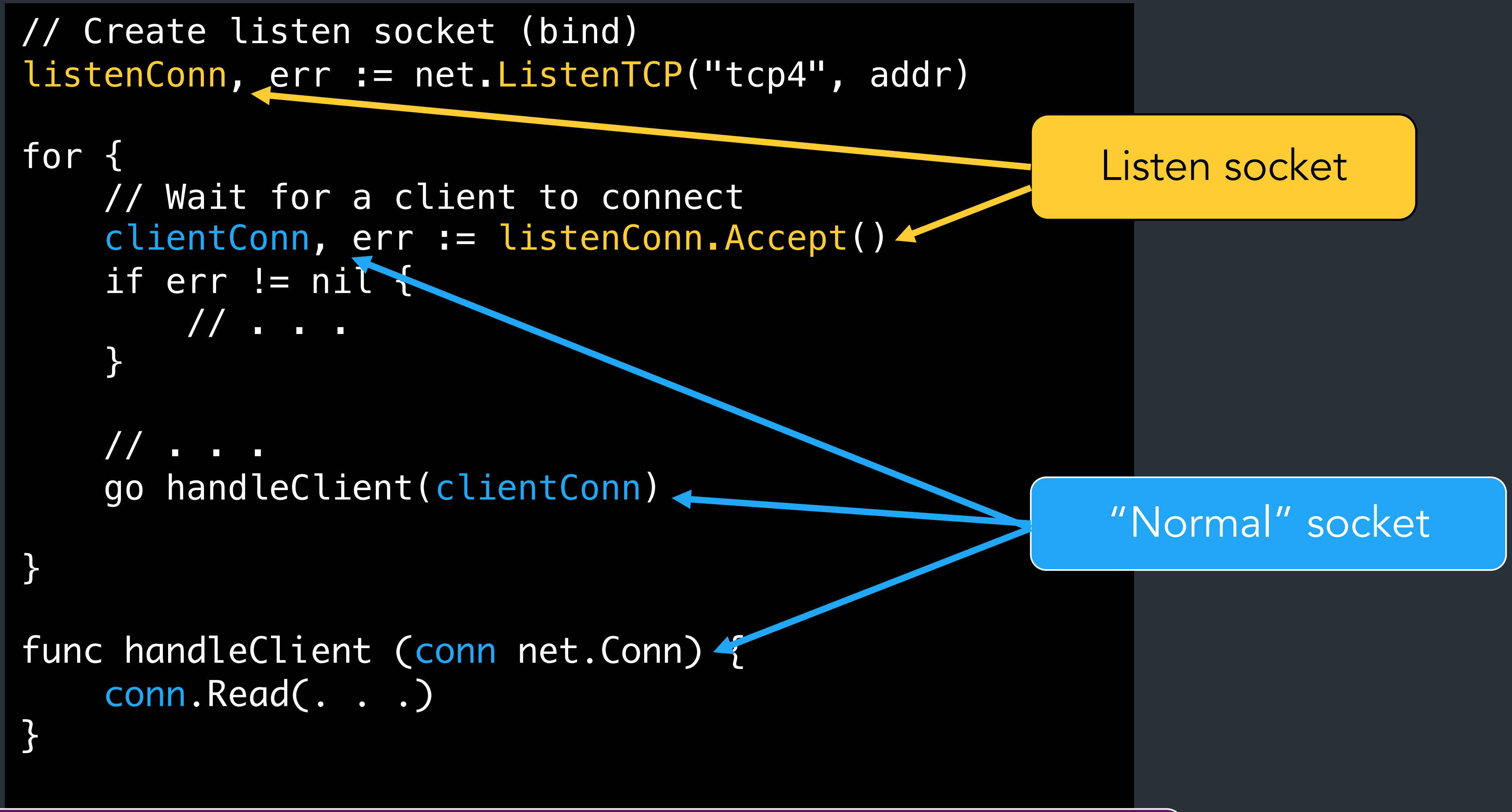
- “Passive OPEN” in RFC
- Returns new listen socket

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

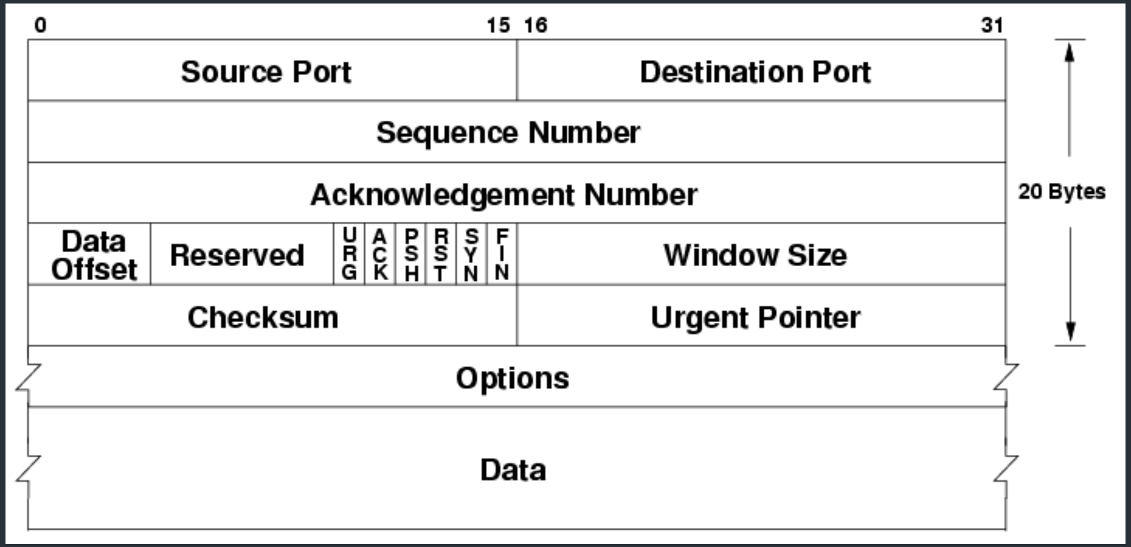
```
int sock_fd = VConnect(addr, port)
. . .
VWrite(sock_fd, some_buffer)
```

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!

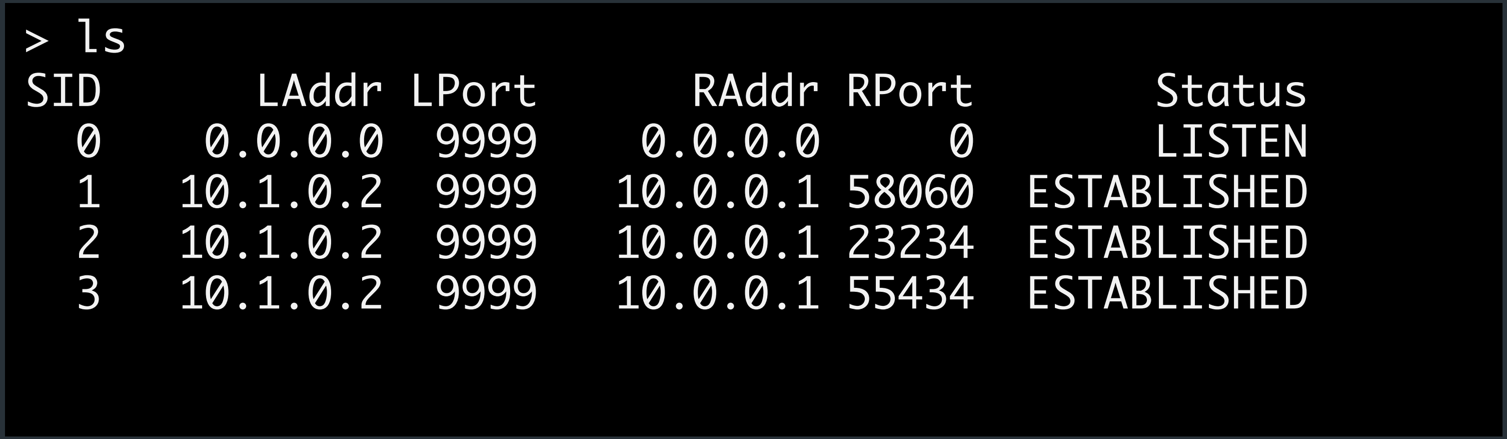
Proto	Local (yours)		Remote (theirs)		Socket
	IP	Port	IP	Port	
tcp	1.2.3.4	12345	5.6.7.8	80	(normal struct)
tcp	*	22	*	*	(listen struct)
…	…	…	…	…	…

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

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

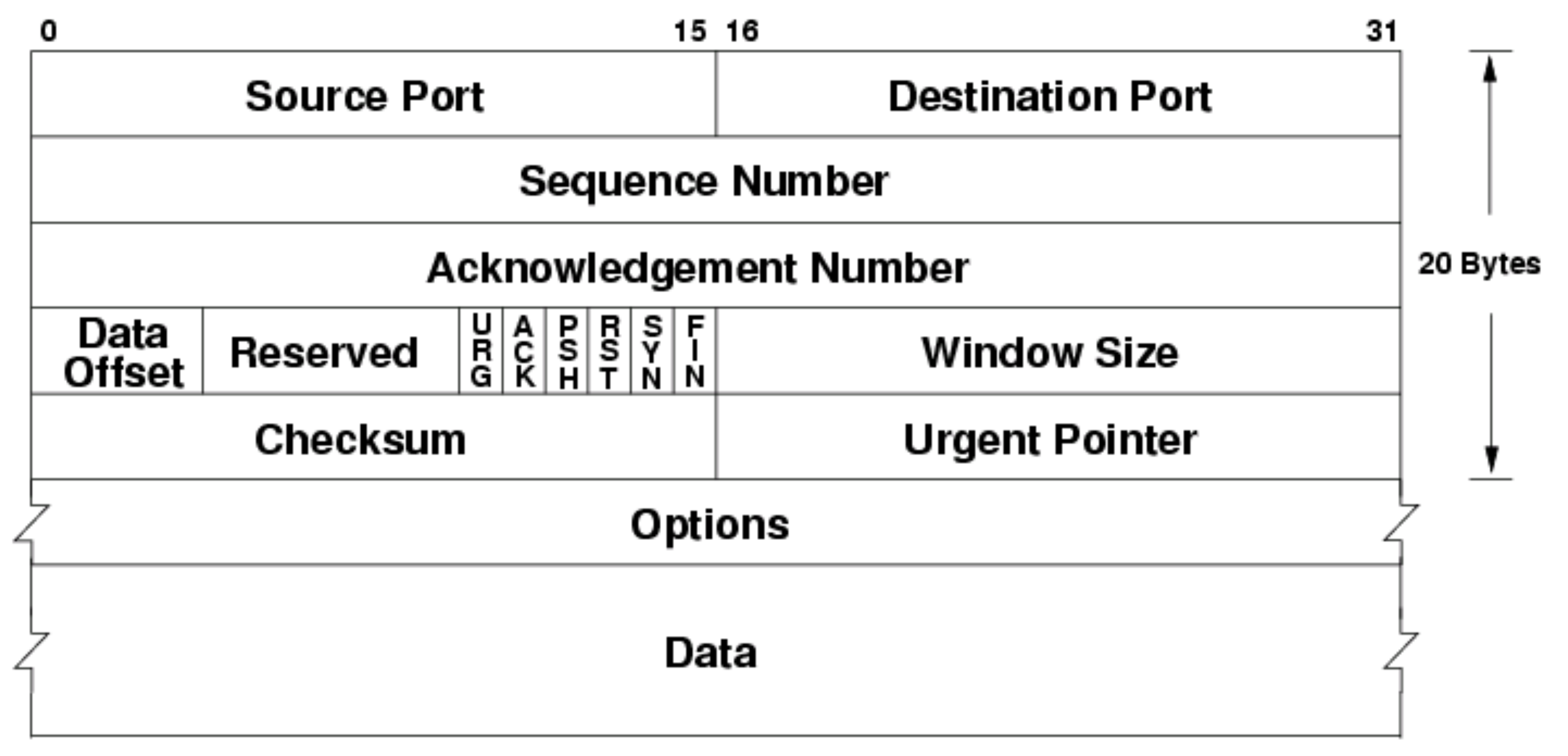
When you receive a TCP packet

- 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

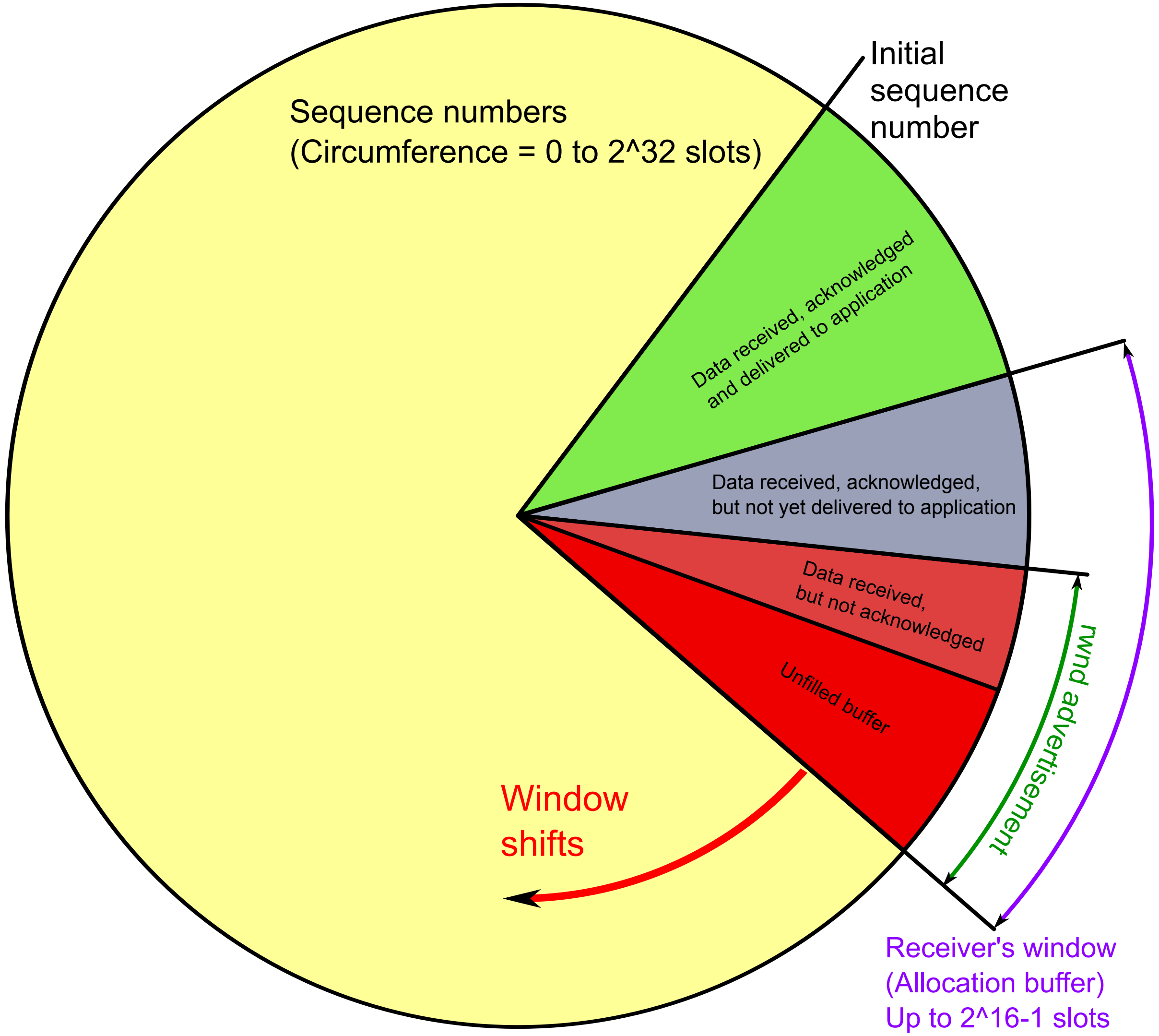
Checksum

Urgent Pointer

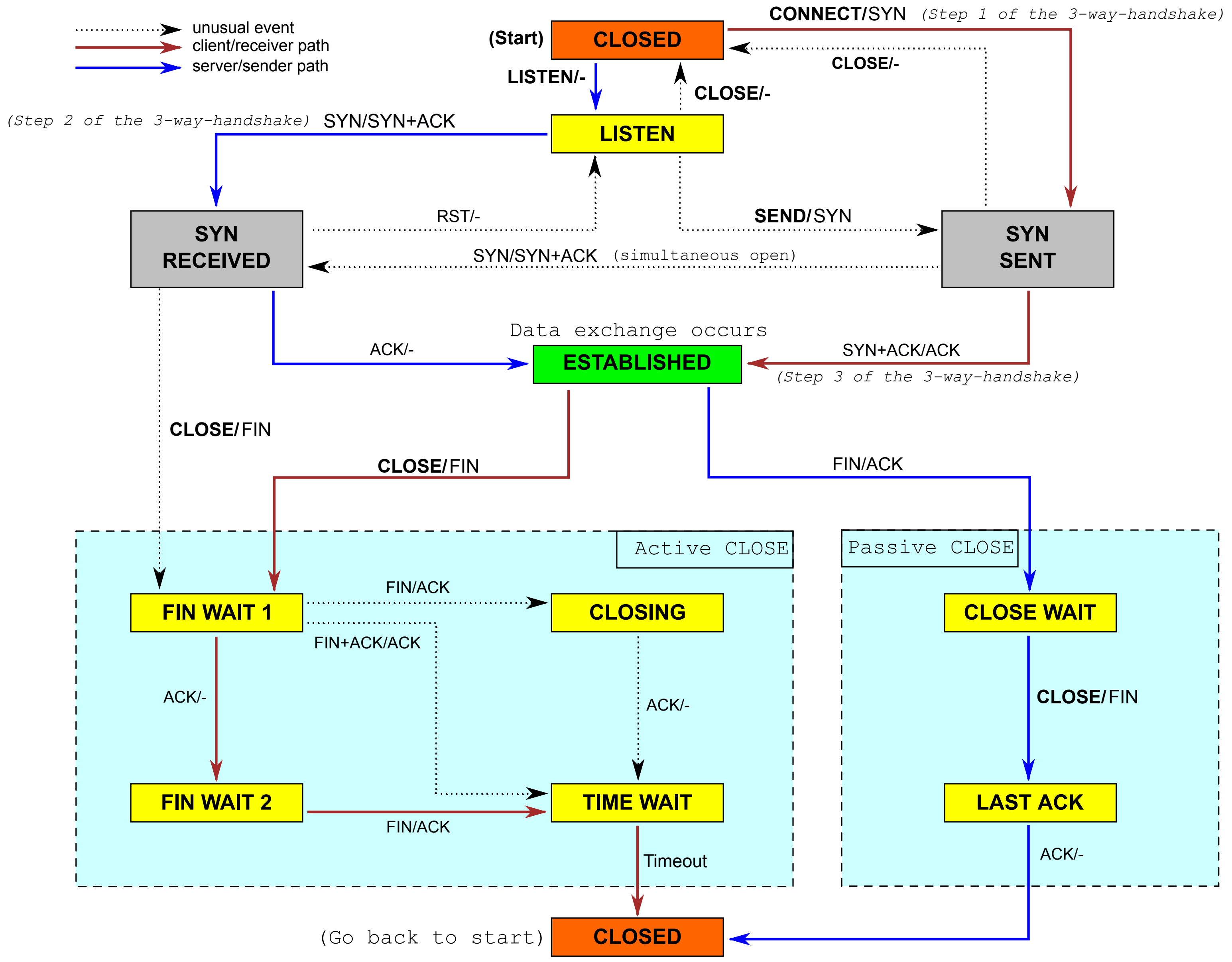
20 Bytes

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