Today
- Light intro to TCP
- TCP handshake

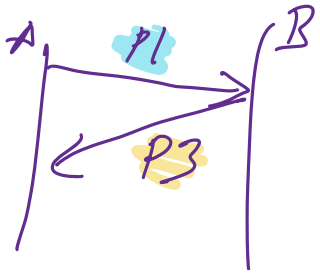
Administrivia
- IP grading: look for announcement later today
    - If you fix bugs while working on TCP => push and resubmit
- TCP out today
    - Look for announcement with handout
    - Look for email with a new stencil link (even if keeping same team)
    - Gearup: Thursday 3/12 5pm in CIT 165
        - Intro, how to think about Milestone I (due before spring break)

Lecture 13: TCP Intro

Warmup: You capture the following packets on your router. How many unique connections are present?

Pkt#	Proto	Source		Destination		CONN#
		IP	Port	IP	Port	
P1	TCP	10.0.0.1	12345	1.2.3.4	80	1
P2	TCP	10.0.0.2	55444	5.6.7.8	443	2 DIFF IP, PORTS
P3	TCP	1.2.3.4	80	10.0.0.1	12345	1 RESPONSE TO P1
P4	TCP	10.0.0.1	43233	1.2.3.4	80	3

SAME HOSTS, BUT DIFFERENT SRC PORT => DIFFERENT CONN TO SAME DESTINATION (EG. NEW TAB)



5-tuple of (local IP, local port, remote IP, remote port, protocol number) uniquely identifies one connection

=> called a "flow"

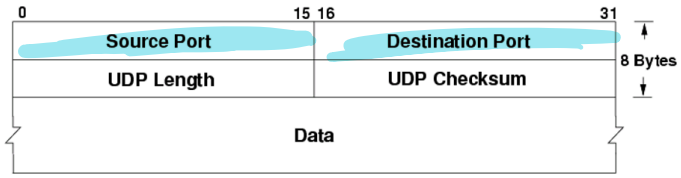
Transport layer: the story so far

- Provides support for different applications via **ports**
- OS provides interface to applications via **sockets**

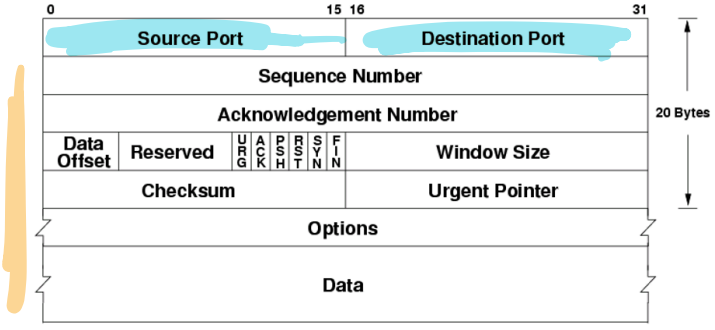
Transport layer is (usually) part of the OS => service provided to apps

The headers

UDP header



TCP header



Motivation: sending a big file

```
cp ~/dir/all-my-files.zip ~/some-other-dir
```

vs.

```
scp ~/dir/all-my-files.zip  1.2.3.4:/some/other/dir
```

Brainstorm: What are some of the challenges involved in implementing the network part? What features do we need?

Starting point: just our SendIP function:

```
SendIP(destIP, protocol, []bytes)
```

Challenges / problems to solve?

- Need to know where data ends
- Packets may be dropped
- Packets may get corrupted (crumpled up)
- Packets may get reordered

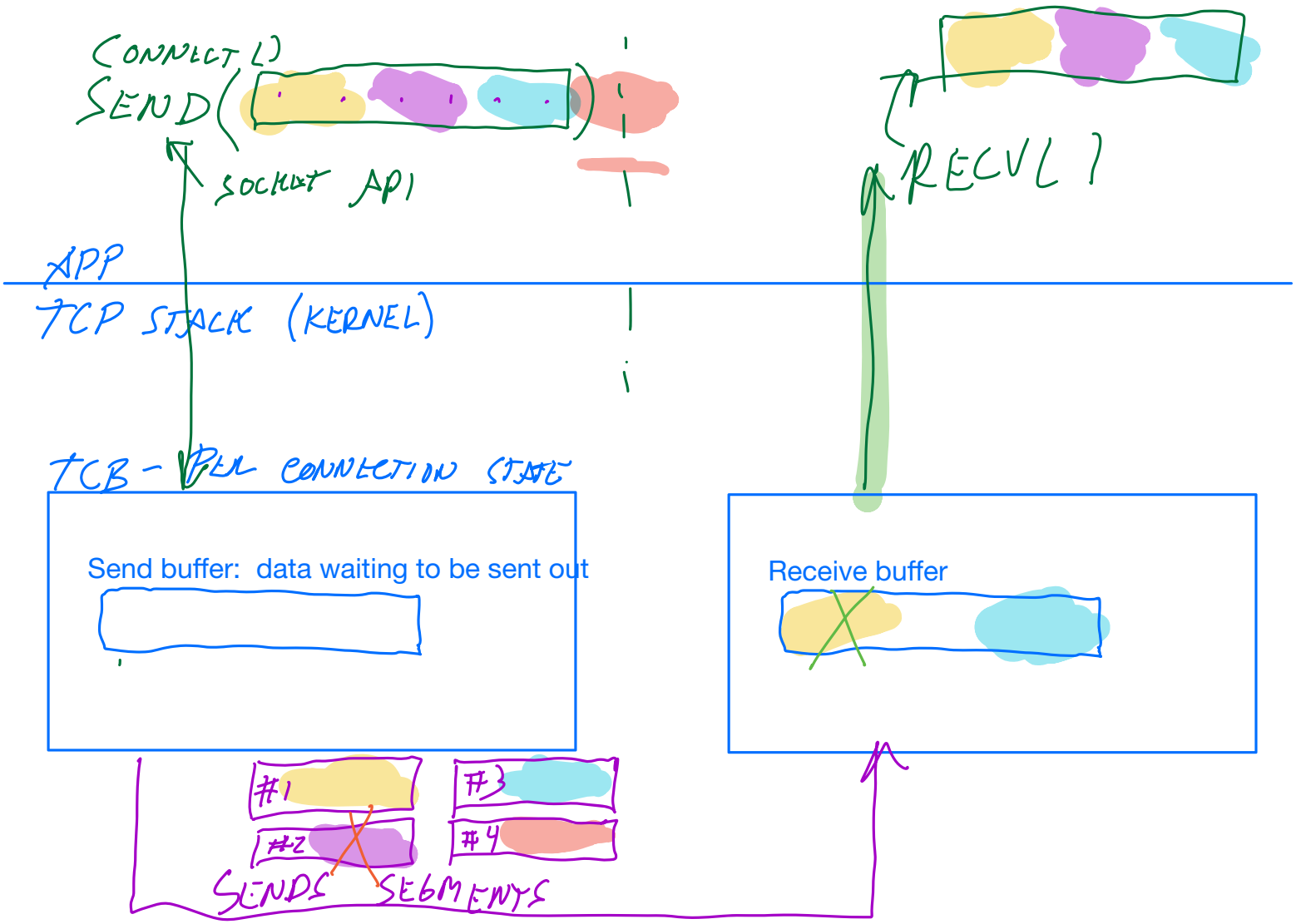
- Sender needs to know the data arrived
- Need to make sure sender doesn't overwhelm receiver
  => "flow control"
- Need to make sure we don't overwhelm networks
  => "congestion control"

Other great challenges from our discussion in class, but not handled by TCP:

- Packets might be intercepted (i.e., eavesdropping)
  => TCP doesn't solve this (instead, we deal with it at other layers--more info later!)

- Packets may take different paths => this part is handled by network layer, but may lead to reordering/dropped packets!

TCP: The Big Picture

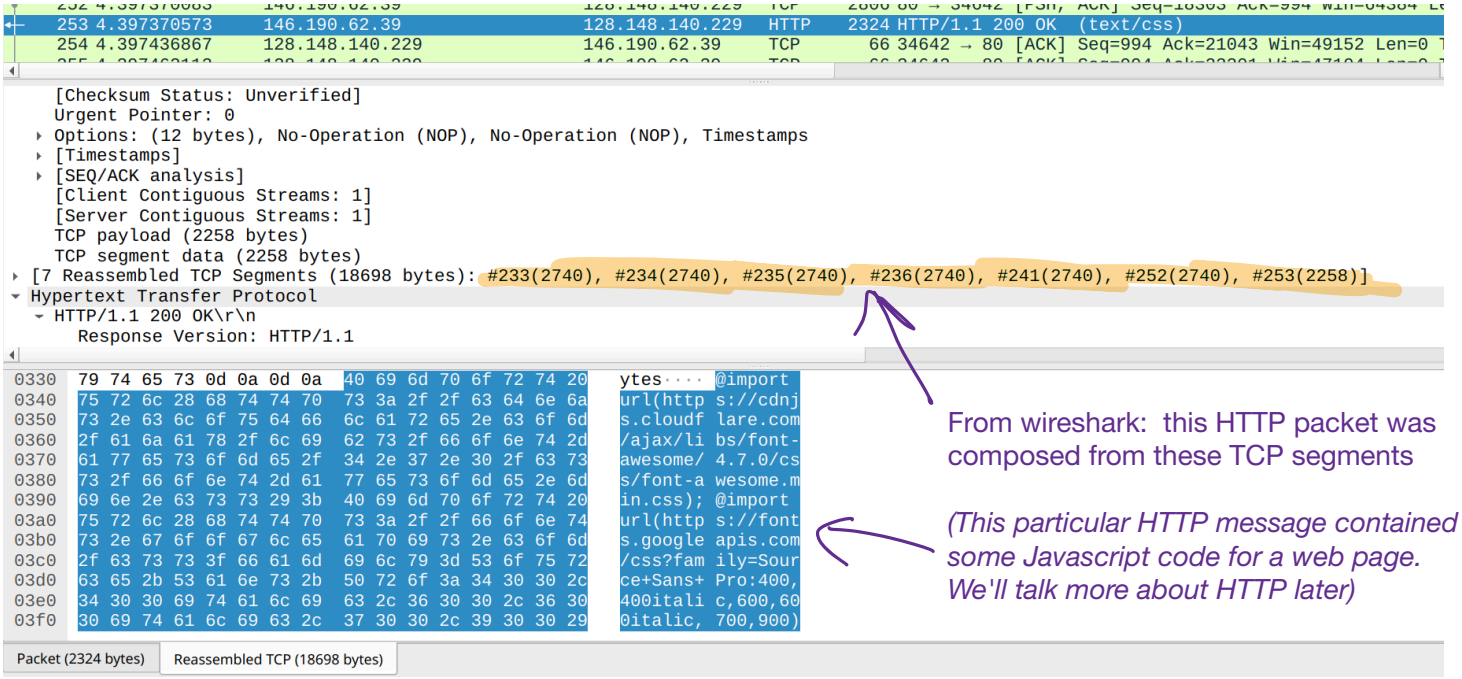


Sending side
- Buffers data from app to be sent
- Divides data into segments
- Track which segments have been received, which have been dropped (retransmit on timeout)
- Flow control: if receiver has no more buffer space, stop sending

Receiving side
- Arrange segments in order in recv buffer
- Sends back "acknowledgements" + other info
- App "pulls" data from the receive buffer, frees up more space for data to arrive from network

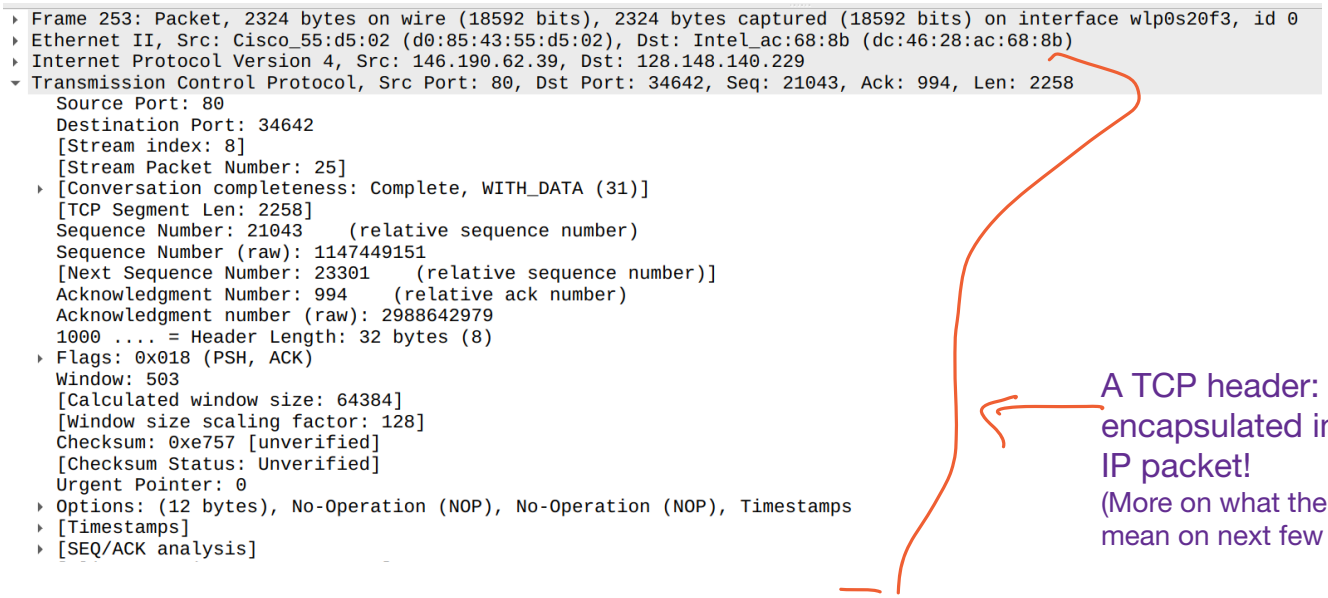
In practice, both sides of the connection can send and receive (full-duplex)
=> Both sides have send and receive buffers
=> (Can use the same socket to send and receive)

Examples: in wireshark



From wireshark: this HTTP packet was composed from these TCP segments

(This particular HTTP message contained some Javascript code for a web page. We'll talk more about HTTP later)



A TCP header: encapsulated inside IP packet!

(More on what the fields mean on next few pages)

Key features

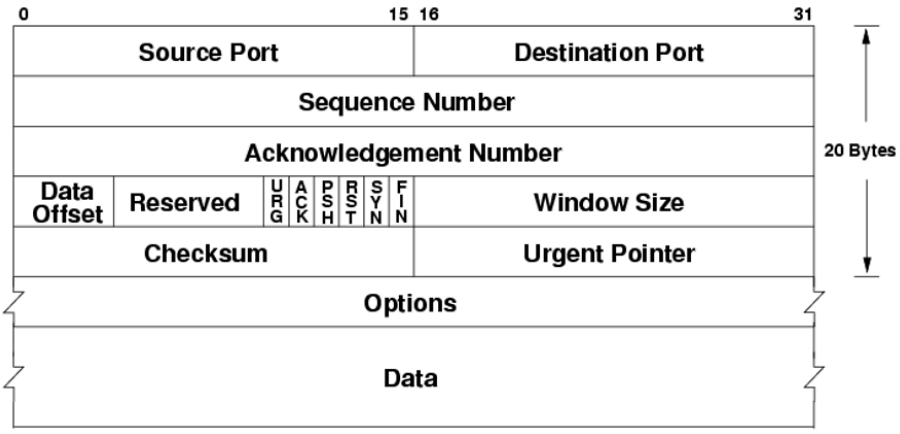
Special note: for project, you should read the (newer) RFC9293 instead. We'll talk about why next class.

- Initially defined by: RFC 793 (1981) (+ many others now)
- Creates concepts of connections between two endpoints
    - Each connection has its own state

- "Reliable, connection-oriented, byte stream" (we will unpack what this means)

ORDERED SEQUENCE OF BYTES

The header



Most important TCP fields:
- Port numbers: handle multiplexing (what connection this is)

- Sequence number (SEQ): where segment is in the data stream (in bytes)

- Acknowledgement number (ACK): the next sequence number expected (also in bytes)
Example: "I have up to byte 1024, give me byte 1025 next"

- Window: how much data you are willing to receive (how much space is in the receive buffer)

- Flags: status bits that indicate where we are in the connection
  - SYN, ACK, FIN, RST, ...

[redacted]

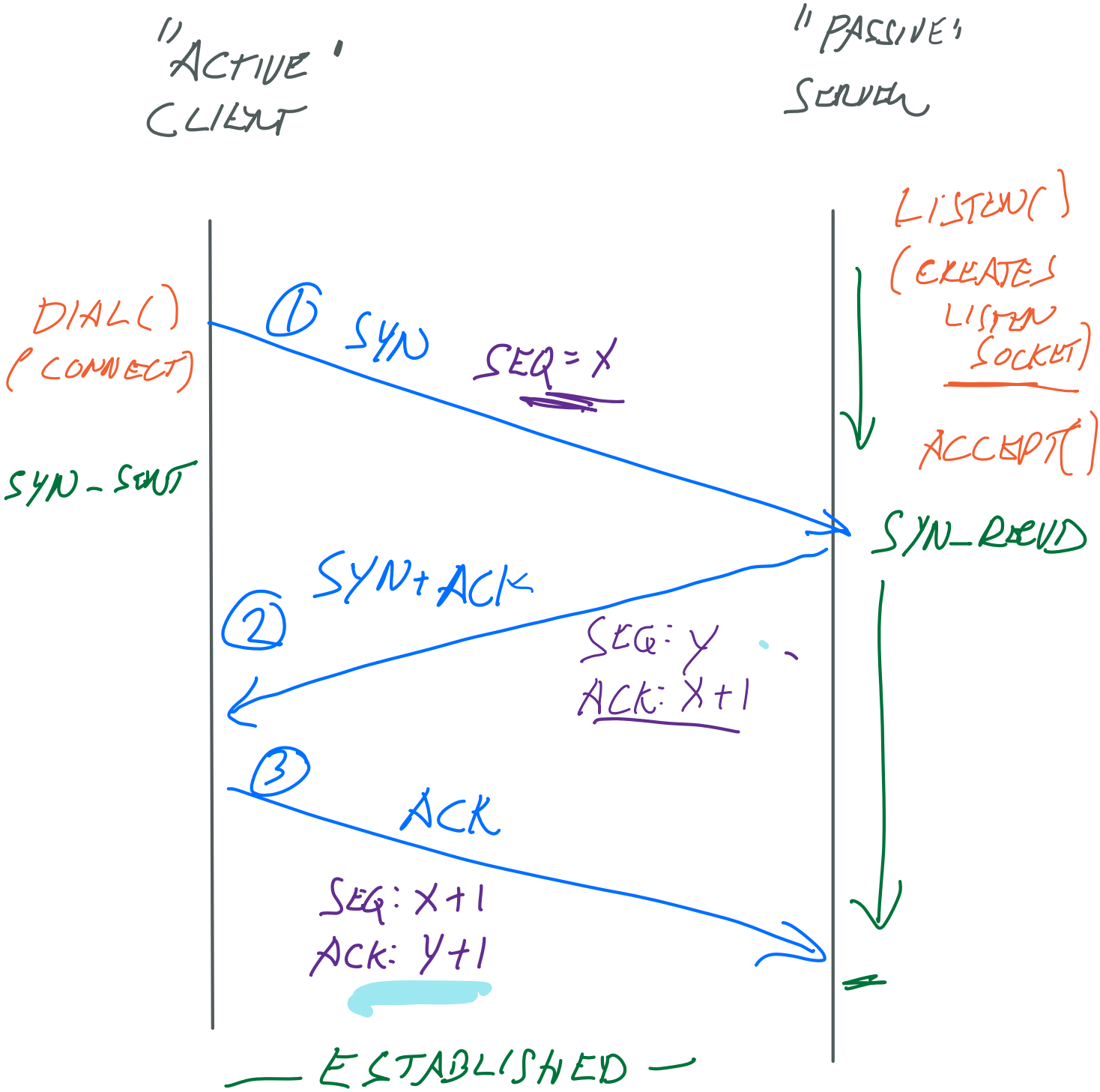
At the socket level: sending a big file

```
func sender() {
  fd, _ := os.Open("all-my-files.zip")

  conn, _ := net.Dial("1.2.3.4:80")

  buf := ReadTheWholeFile(fd)
  conn.Write(buf)
}
```

CONNECT()

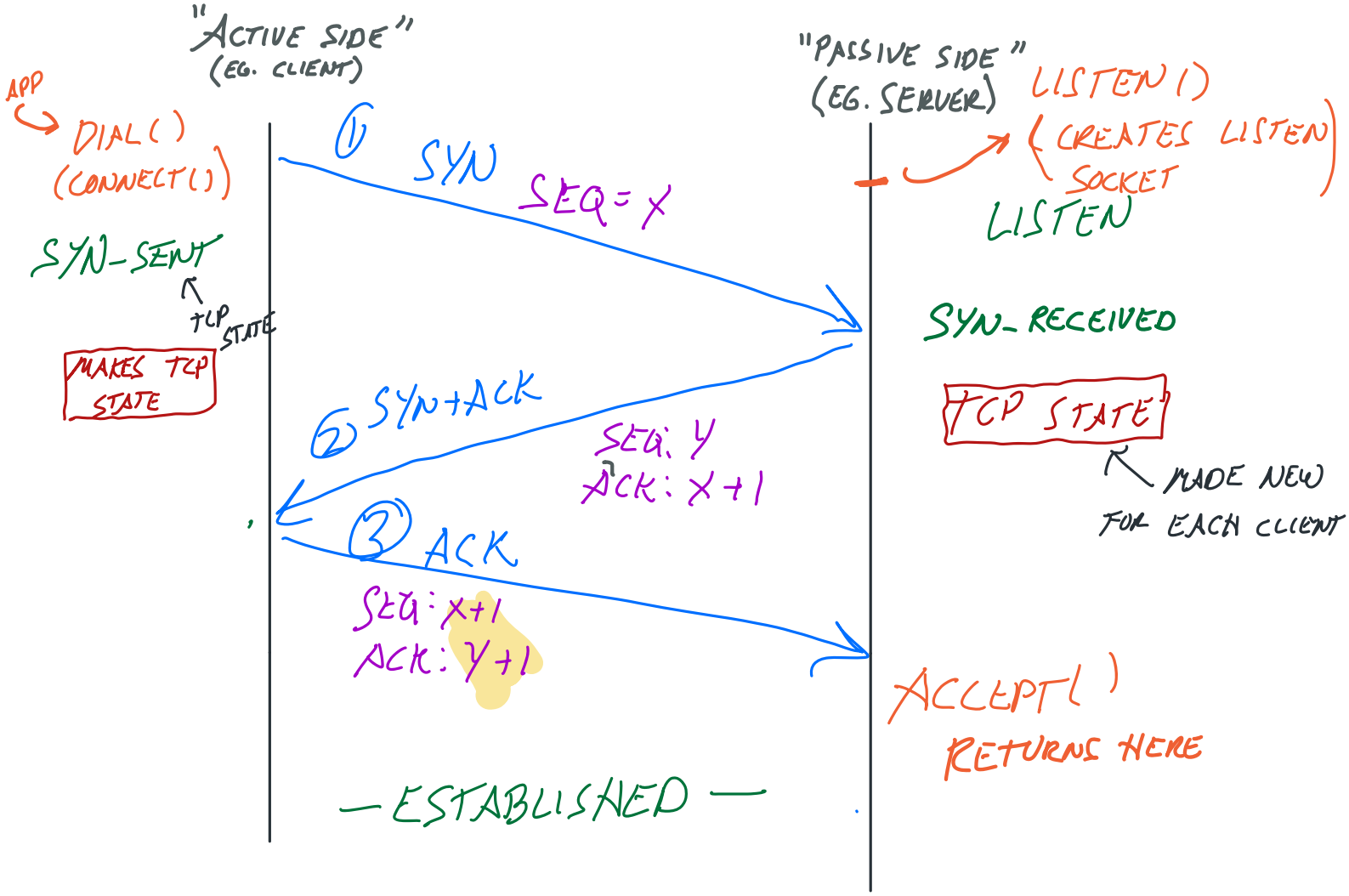
```
func receiver() {
  listenConn, err := net.Listen(“:80”)
  clientConn, err := listenConn.Accept()

  buf := make([]byte, . . .)
  conn.Read(buf)

  fd = os.Open(”copy-of-files.zip”)
  fd.Write(buf)
}
```

Establishing a connection: The TCP Handshake



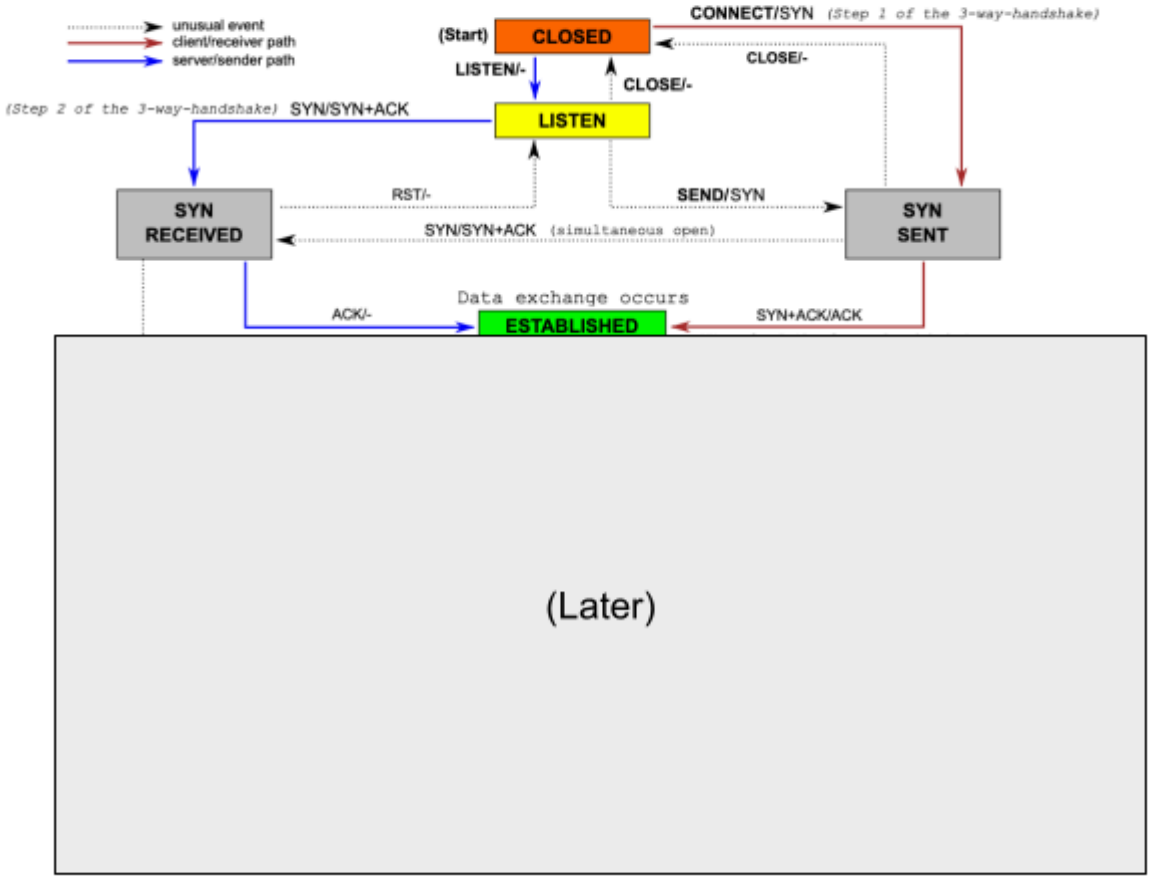
What has happened
- Sender sends something to receiver
- Both sides agree on a starting sequence number
- Both sides in ESTABLISHED state
- Both sides have a socket representing this connection

Establishing a connection: The TCP Handshake



1. Sender sends SYN with random sequence number X
2. Receiver sends SYN+ACK with its own random sequence number Y, acknowledges sender's sequence number with ACK=X+1
3. Sender acknowledges receiver's sequence num with ACK for Y+1 (packet also has SEQ=X+1, since it comes after packet (1))

⇒ 3-WAY HANDSHAKE



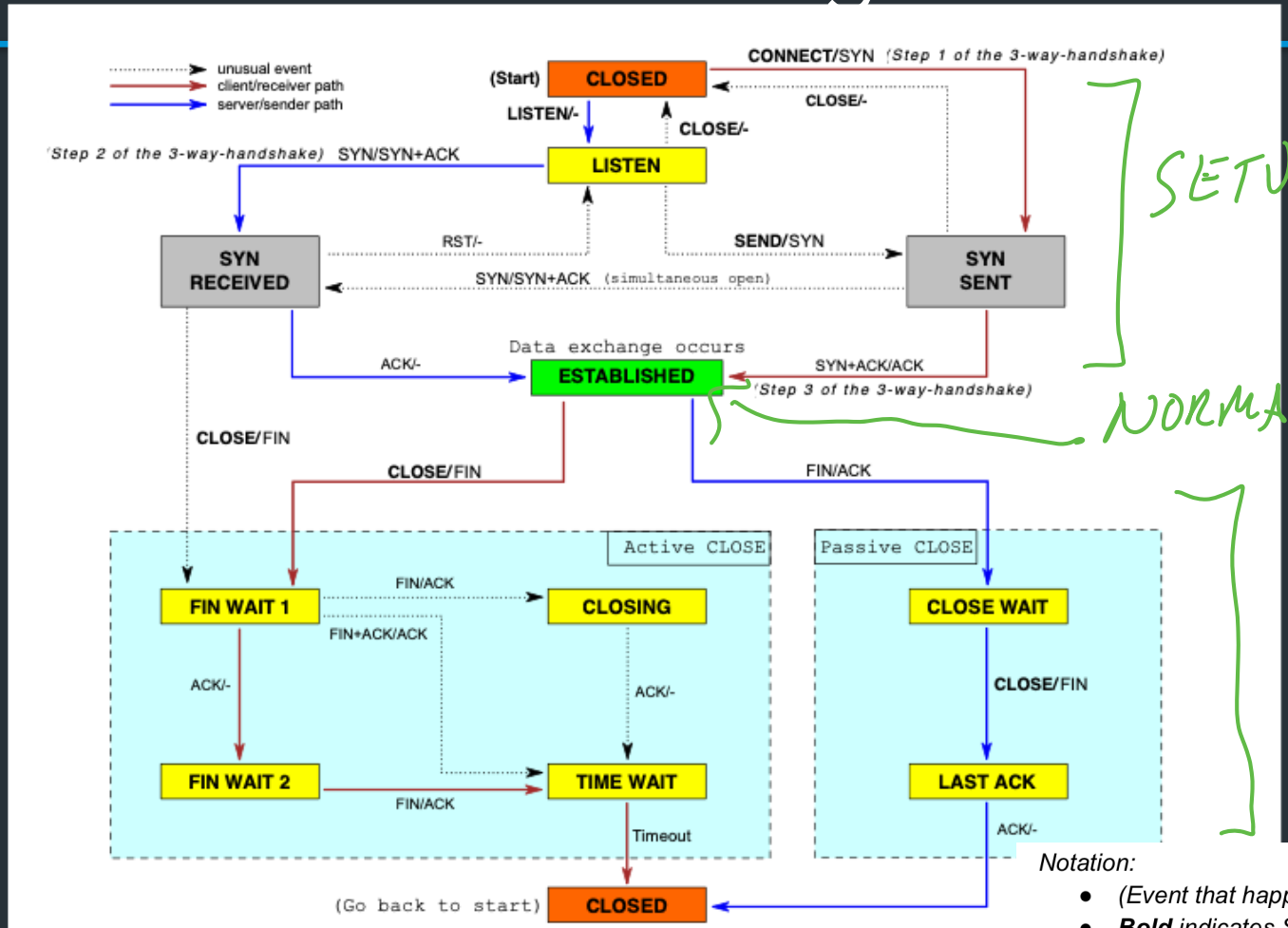
(from wikipedia)

Notation:

- *(Event that happens)/(packet sent in response)*
- ***Bold** indicates Socket API calls (e.g., syscalls)*

TCP State Diagram

(From wikipedia)



Notation:

- *(Event that happens)/(packet sent in response)*
- ***Bold** indicates Socket API calls (e.g., syscalls)*

Sequence numbers: why is the initial sequence number random?

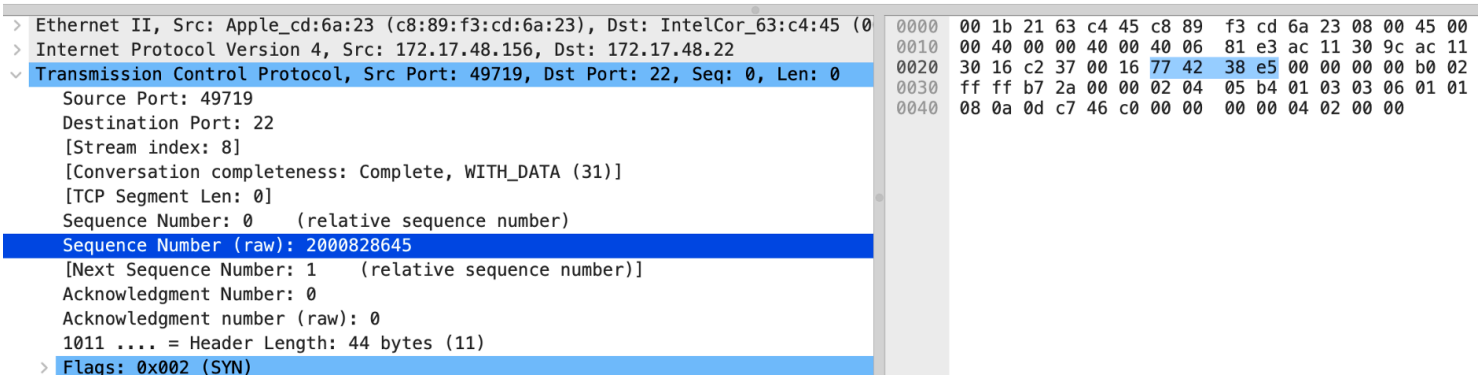
Initial Sequence Number (ISN) => each side picks their own

- Need to make sure sequence numbers aren't reused if host re-uses source port number
- Security reasons: need to make sure adversary can't guess start of connection

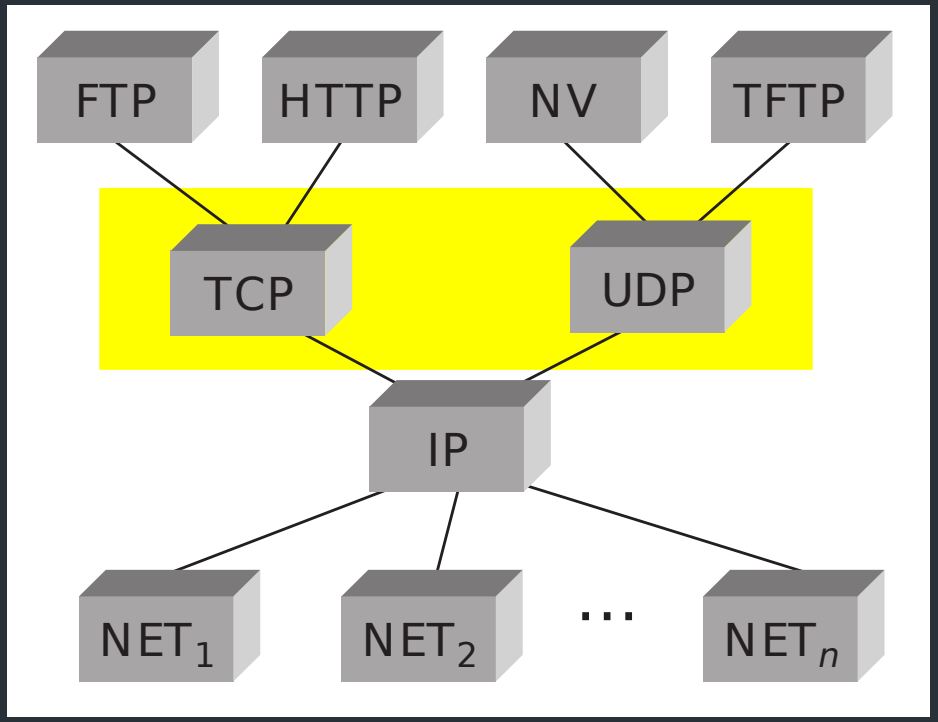
Official way: RFC9293 Sec 3.4.1 describes a procedure based on a timer and a cryptographic hash

=> for our project, just pick a random integer :)

Relative sequence numbering in Wireshark:



Next few pages contain some older notes -- these cover the same content, but have some more pictures. Read on if you're interested / want more info!



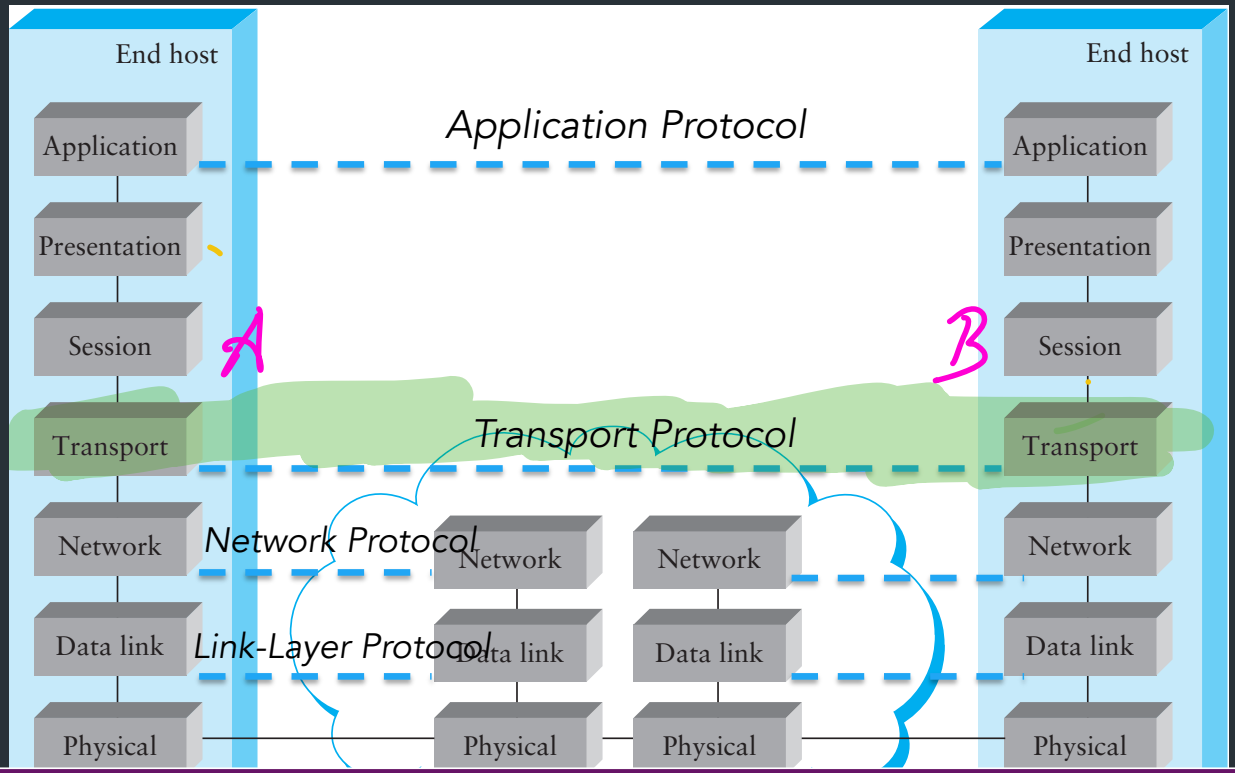
IDEA OF CONNECTIONS FOR APPS

Transport layer: the story so far

- Provides support for different applications via ports
- OS provides interface to applications via sockets

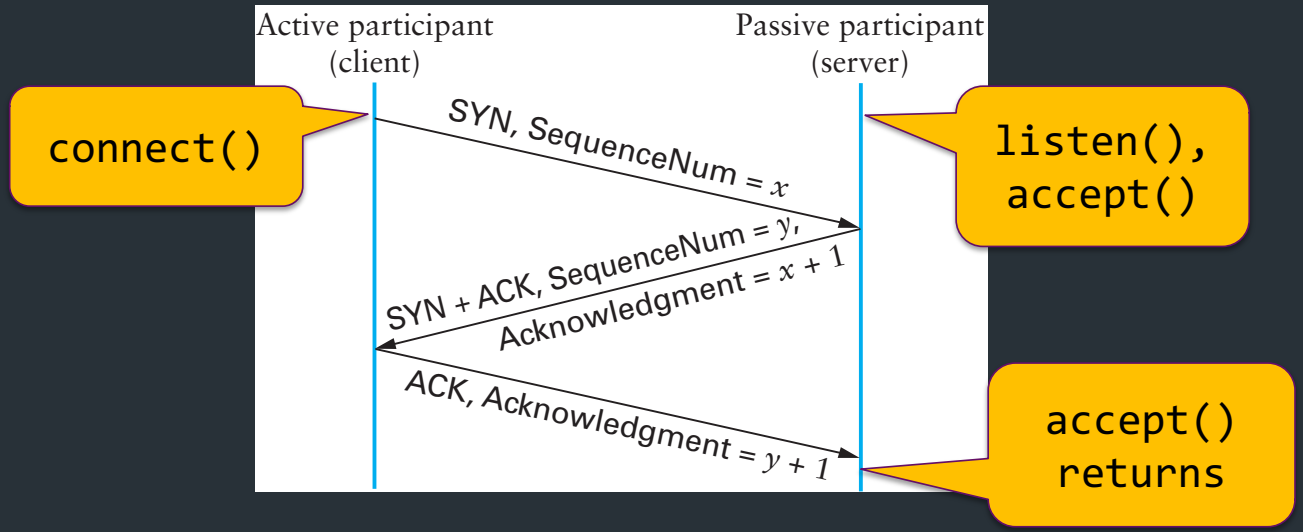
⇒ For now: transport layer is part of OS, service provided to apps

From Lec 2: OSI Model



New abstraction: builds on network layer
=> Just care about endpoint, not individual routers on path
=> All mechanisms run on endpoints (not routers) => Why?

Establishing a Connection



- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: OS may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout

How do we tell two connections apart?

=> Port numbers

- 5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Connection
- Kernel maintains socket table: maps (5-tuple) => Socket

- If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection