CSCI-1680
Transport Layer I

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti

Administrivia

- IP: due tonight!
  - Look for email today/tomorrow about grading meetings
    + feedback survey

"Between the time you've handed in and the demo meeting, you can continue to making small changes and bug fixes and push them to your git repo"

  - *OK: Fixing bugs, code cleanup, README*
  - *Not OK: Implementing RIP, adding new features*

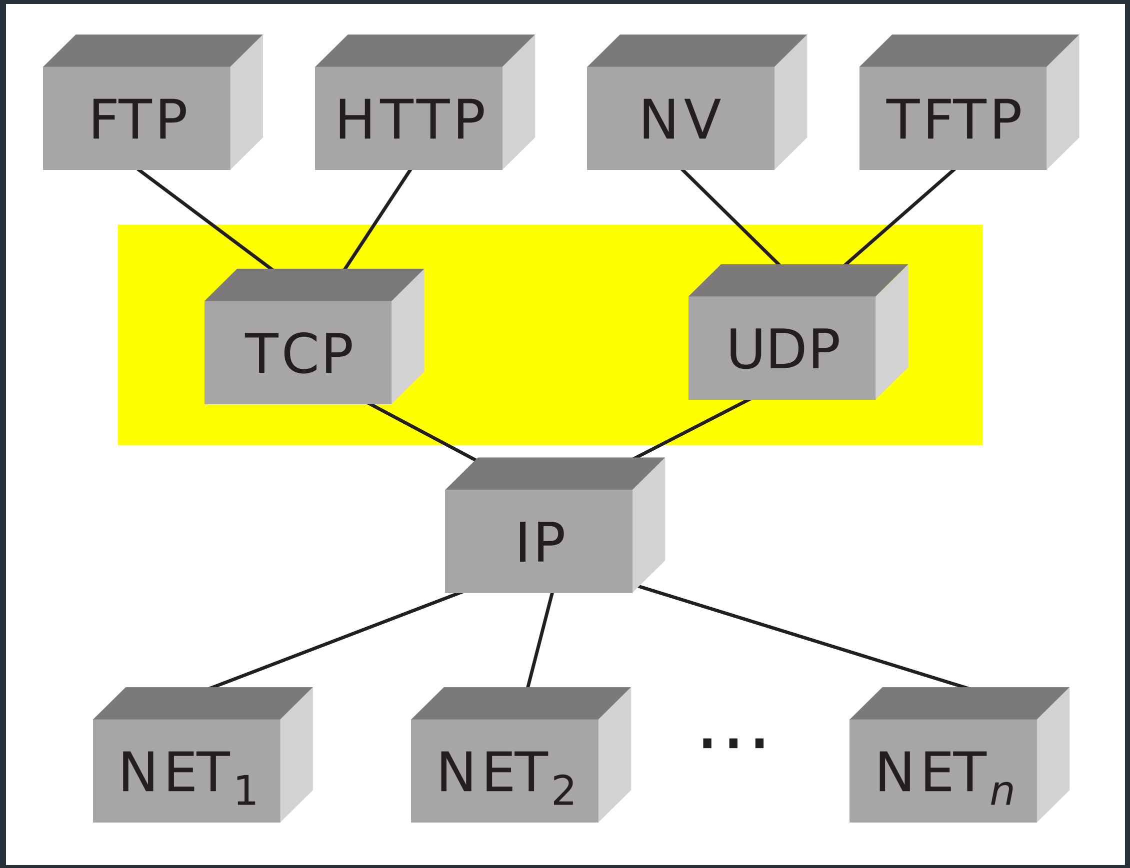
Administrivia

- HW2 is out (finally!): Due Monday, Oct 30
- HW3 will be super short: out Oct 31, due Nov 7

- TCP: Should be out tomorrow
  - Gearup on Monday, Oct 23 6-8pm in CIT316

Today

- Light overview of the transport layer and TCP
  - Why we need TCP
  - What components are involved
  - What you will do in the project



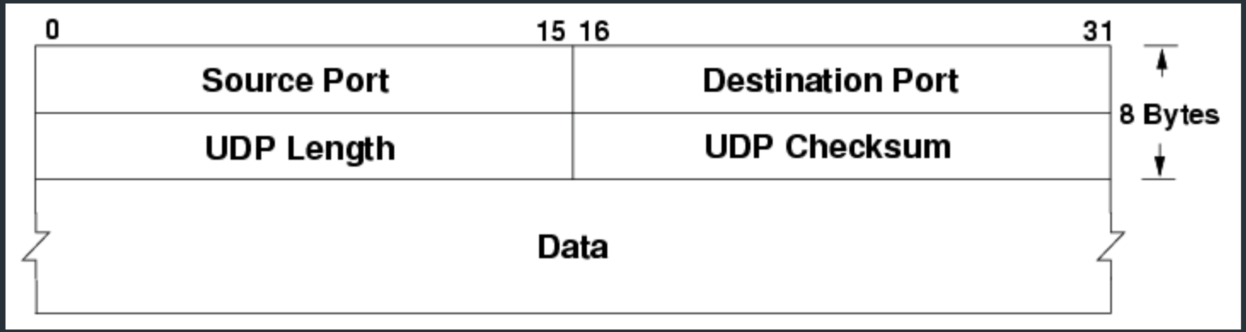
Transport layer: the story so far

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

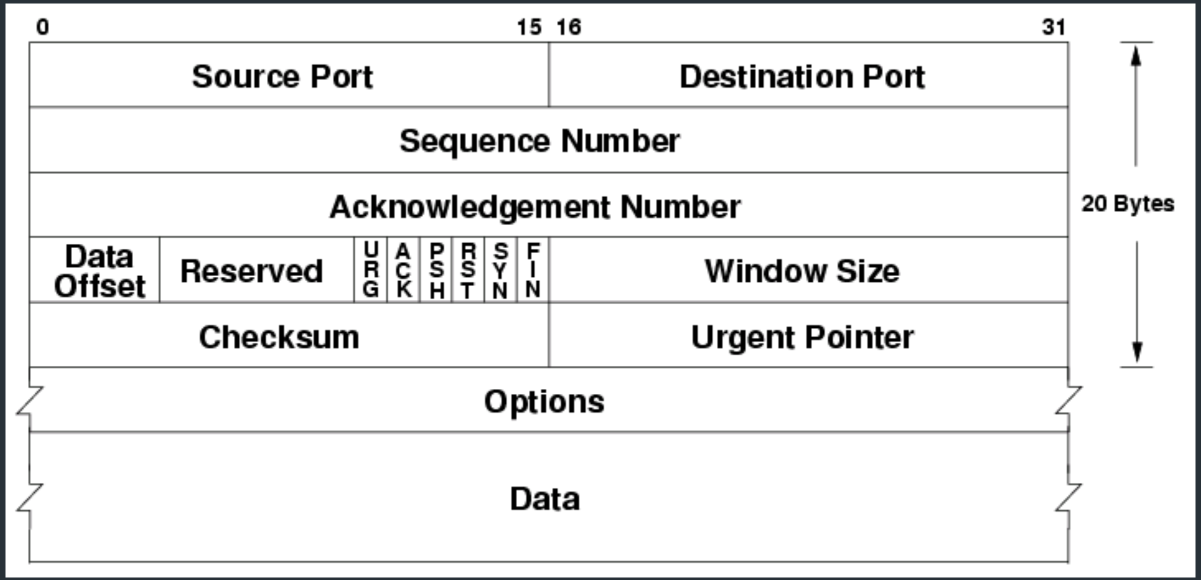
⇒ For now: transport layer is part of OS, service provided to apps

The headers

UDP



TCP



Port numbers are part of these headers
=> OS uses these to map to sockets

Motivation: sending a big file

A problem, in pseudocode:

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

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

What are some challenges with implementing the network part?

Motivation: sending a big file

A problem, in pseudocode:

```
func sender() {
  fd, _ := os.Open("all-my-files.zip")
  conn, _ := net.Dial("1.2.3.4:80")
  buf := ReadTheWholeFile(fd)
  conn.Write(buf)

}
```

```
func receiver() {
  conn, err := net.Listen(“:80”)
  buf := make([]byte, . . .)
  conn.Read(buf)

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

What are some challenges with implementing the network part?

Motivation: sending a big file

A problem, in pseudocode:

```
func sender() {
  fd, _ := os.Open("all-my-files.zip")
  conn, _ := net.Dial("1.2.3.4:80")
  buf := ReadTheWholeFile(fd)
  conn.Write(buf)

}
```

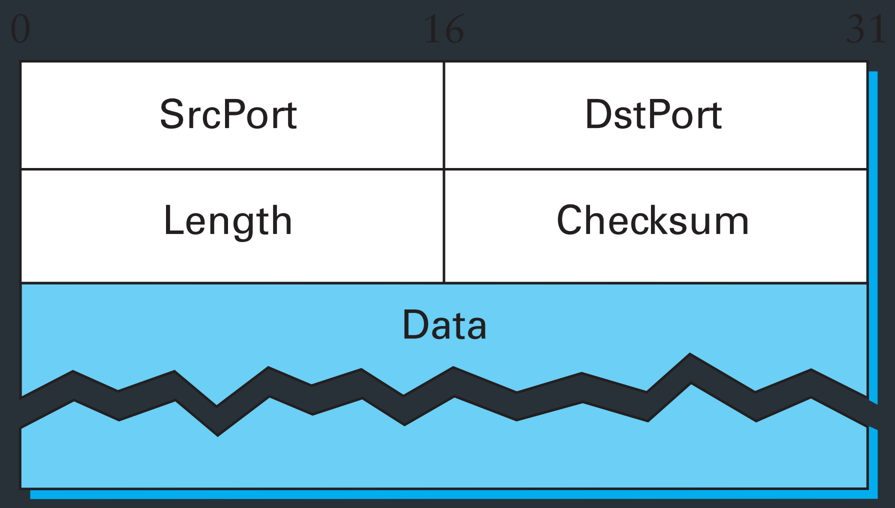
```
func receiver() {
  conn, err := net.Listen(“:80”)
  buf := make([]byte, . . .)
  conn.Read(buf)

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

⇒ How do we get data from A->B, reliably?

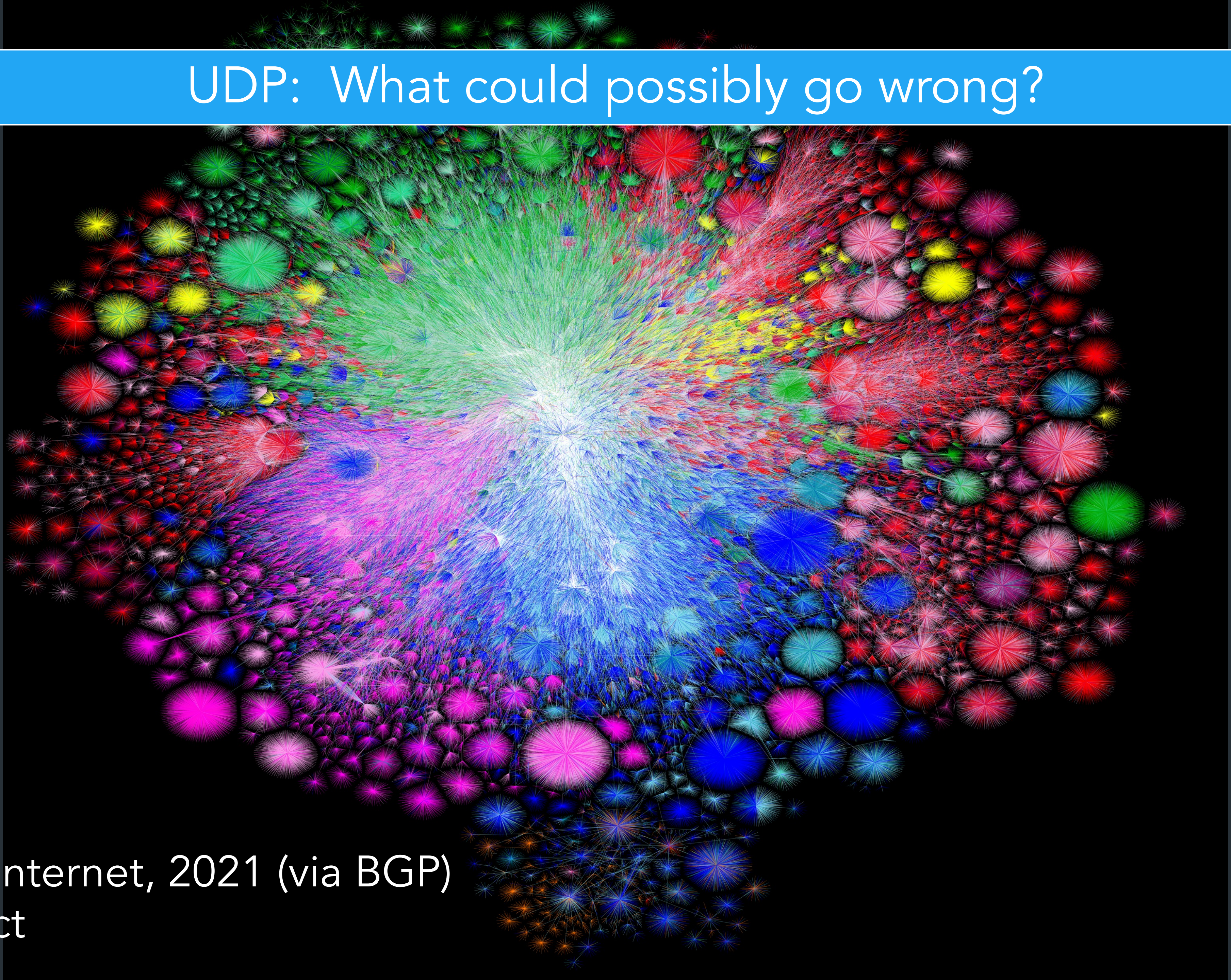
How does the transport layer help us do this?

UDP: User Datagram Protocol

Send a message between ports… and nothing else



UDP: What could possibly go wrong?



Map of the Internet, 2021 (via BGP)
OPTE project

Problem: Reliability

Packets could…

- Dropped packets
- Duplicate packets
- Packets arrive out of order

Multiple hops and paths => Lots of opportunities for failure!
=> TCP has mechanisms to deal with this

Also: performance challenges

- *Hosts* have different (and unknown!) resources

- *Network* has unknown resources

  => Varying RTT, link bandwidth

Also: performance challenges

- *Hosts* have different (and unknown!) resources

  => *Flow control: how much data can we send to receiver?*

- *Network* has unknown resources

  => Varying RTT, link bandwidth

  => Congestion control: must not overload network

Also: performance challenges

- *Hosts* have different (and unknown!) resources

  => *Flow control: how much data can we send to receiver?*

- *Network* has unknown resources

  => Varying RTT, link bandwidth

  => Congestion control: must not overload network

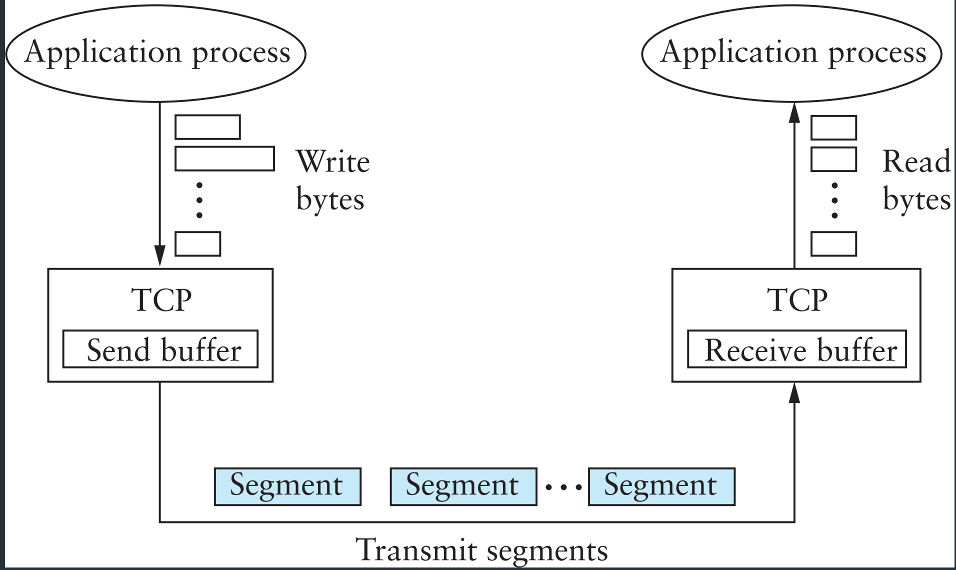
Two performance goals:

1. Must not overwhelm receiver, or network (critical!!)
2. Maximize throughput => best performance

So how does it work?

TCP: the big picture

TCP – Transmission Control Protocol



- Service model: "reliable, connection oriented, full duplex ordered byte stream"
- Flow control: If one end stops reading, writes at other eventually stop/fail
- Congestion control: Keeps sender from overloading the network

TCP: Key features

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

TCP: Key features

- Initially: RFC 793 (1981) (+ many others now)
- Creates concept of connections between two endpoints

  => Each connection has its own state
- End-to-end protocol
  - Minimal assumptions on the network
  - All mechanisms run on the end points (ie, not routers)

TCP: Key features

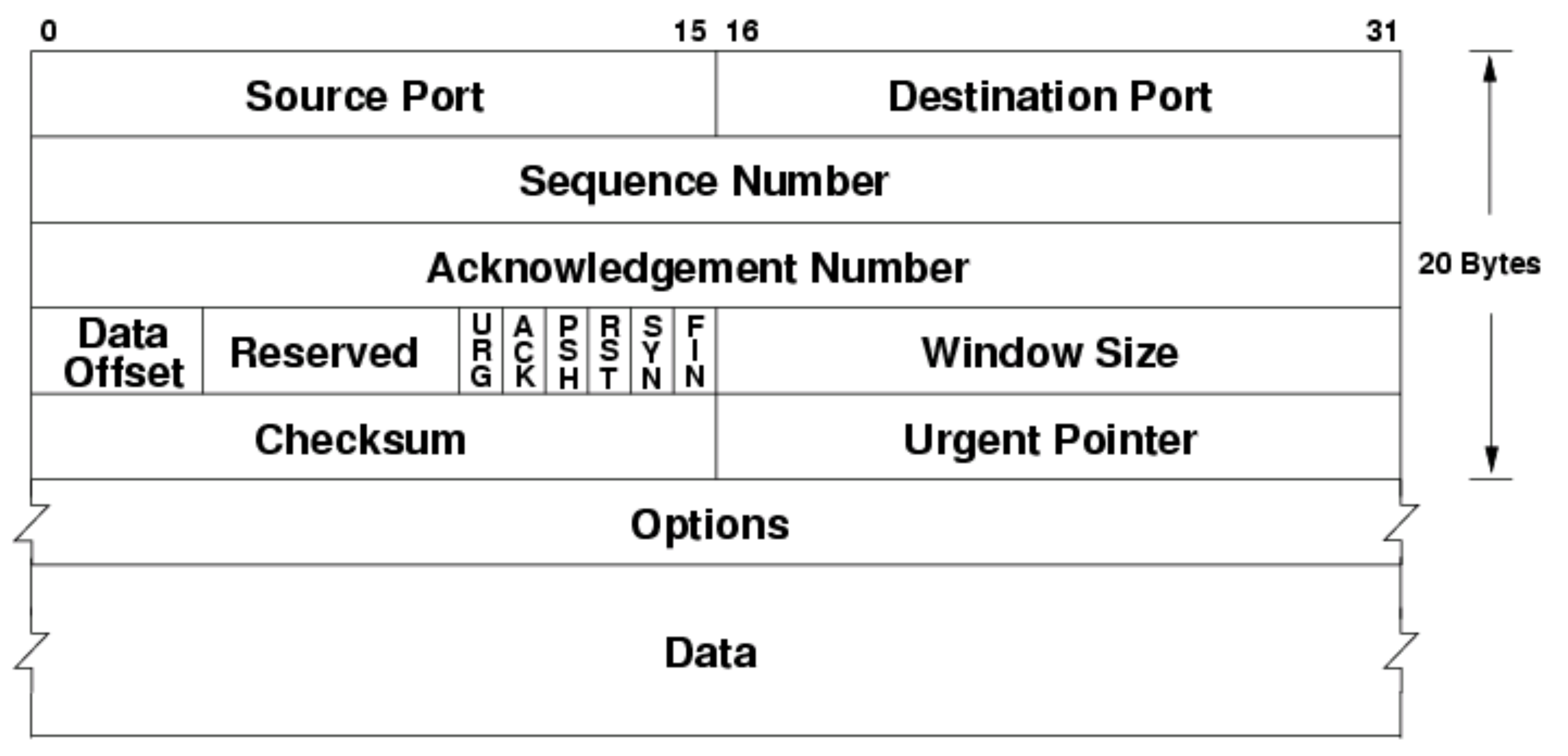
- Initially: RFC 793 (1981) (+ many others now)

- Creates concept of connections between two endpoints

  => Each connection has its own state
- End-to-end protocol
  - Minimal assumptions on the network
  - All mechanisms run on the end points (ie, not routers)

Why is this important?

TCP Header


0
15
16
31
Source Port
Destination Port
Sequence Number
Acknowledgement Number
Data Offset
Reserved
U R G
A C K
P S H
R S T
S Y N
F I N
Window Size
Checksum
Urgent Pointer
Options
Data
20 Bytes

Important Header Fields

- Ports: multiplexing
- Sequence number
  - Where segment is in the stream (in bytes)
- Acknowledgment Number
  - Next expected sequence number
- Window
  - How much data you're willing to receive
- Flags…

Important Header Fields: Flags

- SYN:
- ACK:
- FIN:

- RST: reset connection (used for errors)
- PSH: push data to the application immediately
- URG: whether there is urgent data

Important Header Fields: Flags

- SYN: establishes connection (“synchronize”)
- ACK: this segment ACKs some data (all packets except first)
- FIN: close connection (gracefully)

- RST: reset connection (used for errors)
- PSH: push data to the application immediately
- URG: whether there is urgent data

Less important header fields

- Checksum: Very weak, like IP
  - Has weird semantics ("pseudo header"), more on this later…
- Data Offset: used to indicate TCP options (mostly unused)
- Urgent Pointer

TCP Standards: The Many RFCs

TCP Standards: The Many RFCs

RFC documents [edit]

- RFC 675 – Specification of Internet Transmission Control Program, December 1974 Version
- RFC 793 – TCP v4
- RFC 1122 – includes some error corrections for TCP
- RFC 1323 – TCP Extensions for High Performance [Obsoleted by RFC 7323]
- RFC 1379 – Extending TCP for Transactions—Concepts [Obsoleted by RFC 6247]
- RFC 1948 – Defending Against Sequence Number Attacks
- RFC 2018 – TCP Selective Acknowledgment Options
- RFC 5681 – TCP Congestion Control
- RFC 6247 – Moving the Undeployed TCP Extensions RFC 1072, 1106, 1110, 1145, 1146, 137
- RFC 6298 – Computing TCP's Retransmission Timer
- RFC 6824 – TCP Extensions for Multipath Operation with Multiple Addresses
- RFC 7323 – TCP Extensions for High Performance
- RFC 7414 – A Roadmap for TCP Specification Documents
- RFC 9293 – Transmission Control Protocol (TCP)

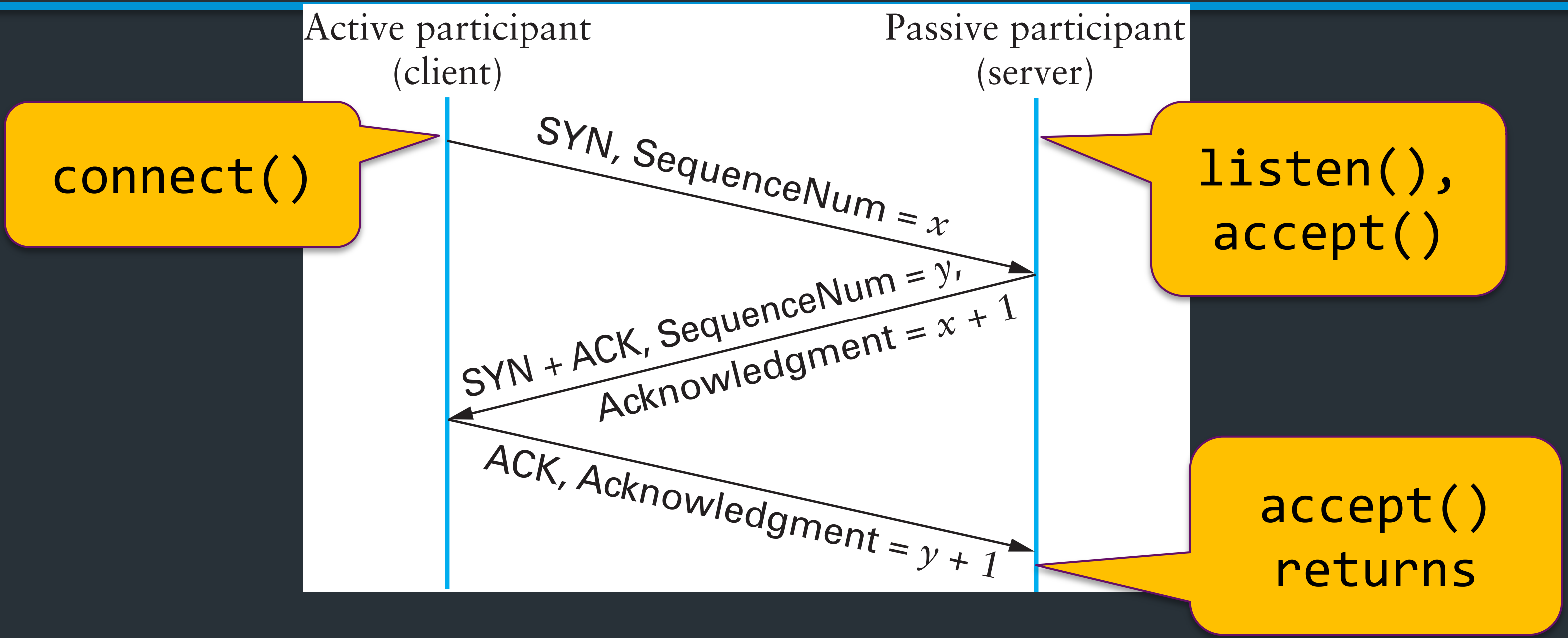
The One RFC

28 Internet Standard
(IETF) W. Eddy, Ed.
MTI Systems
STD: 7 August 2022
Request for Comments: 9293
Obsoletes: 793, 879, 2873, 6093, 6429, 6528,
6691
Updates: 1011, 1122, 5961
Category: Standards Track
ISSN: 2070-1721

Establishing a Connection

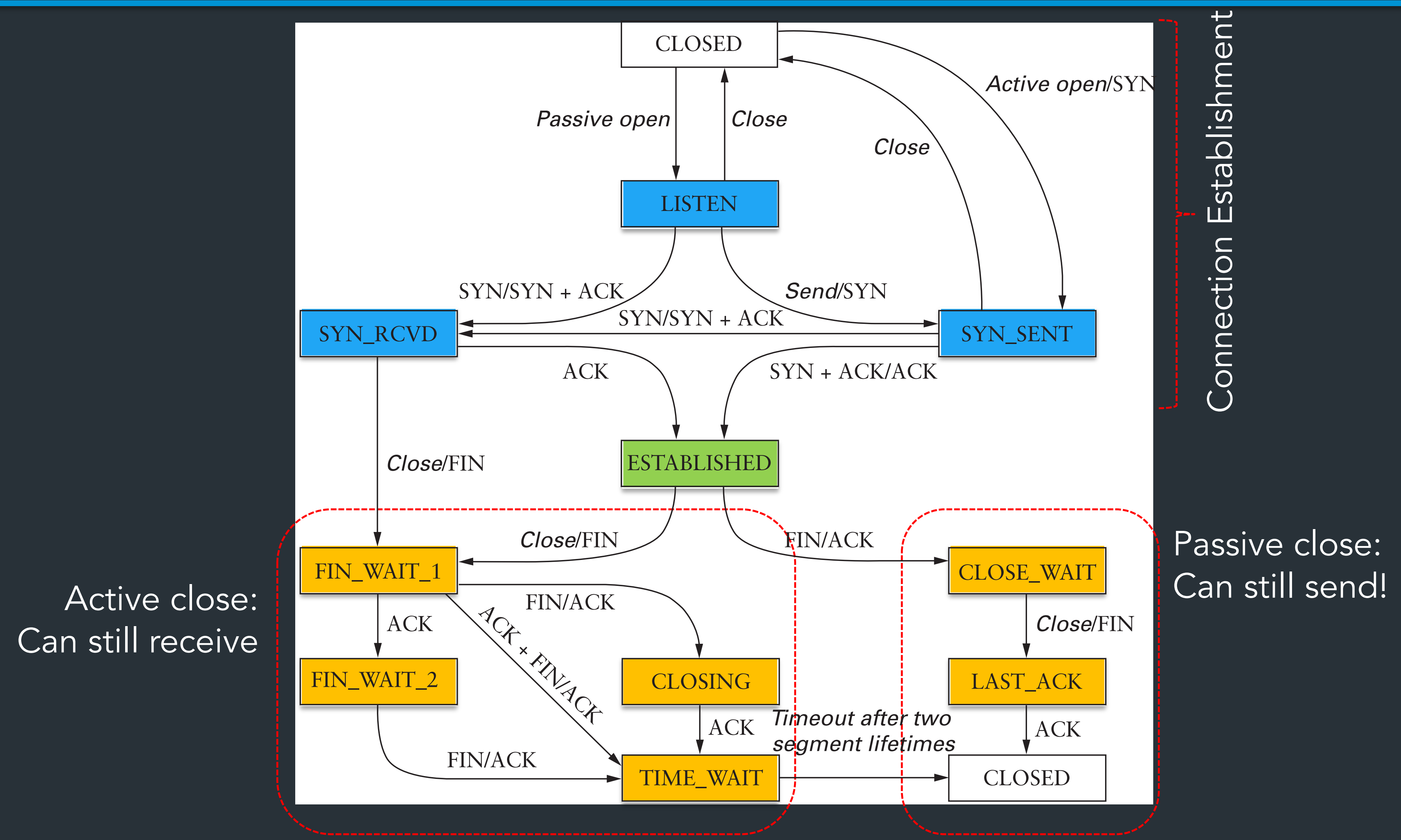
Goals
- Contact the other side (or error)
- Both sides agree on initial sequence numbers

Establishing a Connection

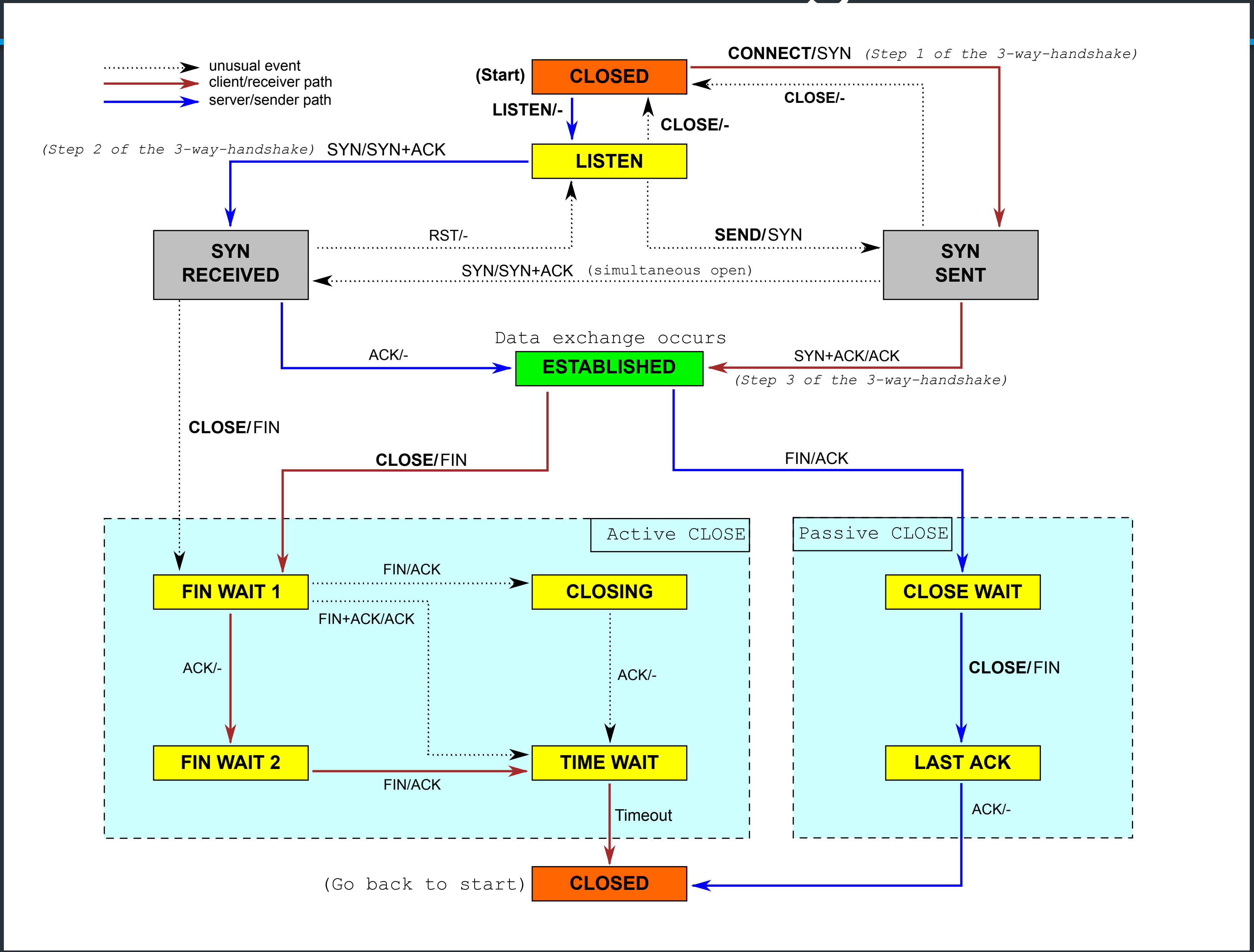


- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: OS may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout

Summary of TCP States



TCP State Diagram



Sequence numbers

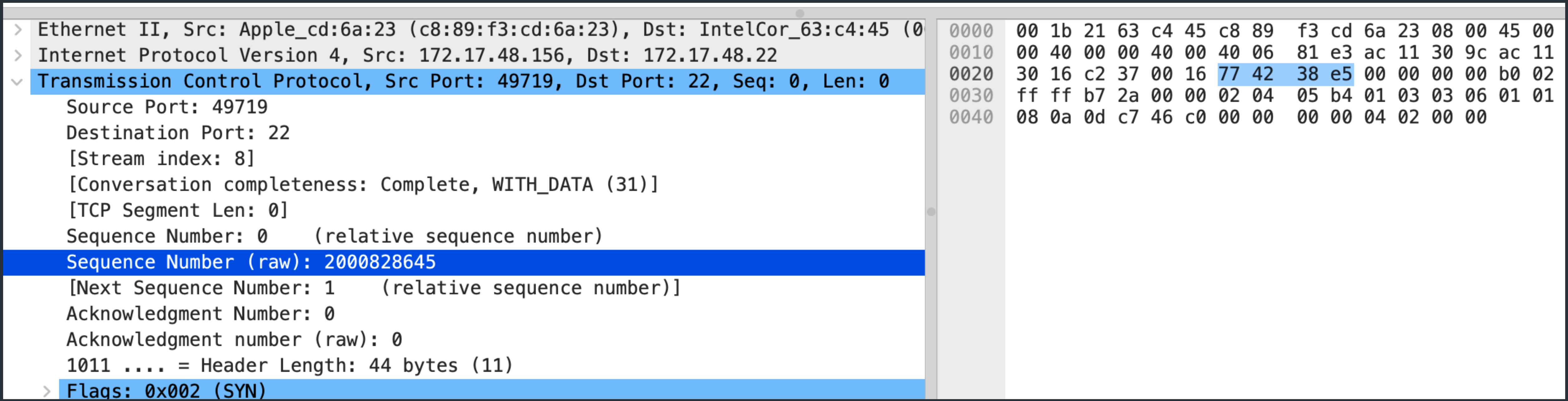
How to pick the initial sequence number?

- Protocols based on *relative* sequence numbers based on starting value
- Why not start at 0?

- RFC9293, Sec 3.4.1: Procedure for picking ISN, based on timer and cryptographic hash

  => For project, just pick a random integer :)

Relative Sequence Numbering



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

Netstat

```
deemer@vesta ~/Development % netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  10.3.146.161.51094     104.16.248.249.443     ESTABLISHED
tcp4       0      0  10.3.146.161.51076     172.66.43.67.443       ESTABLISHED
tcp6       0      0  2620:6e:6000:900.51074 2606:4700:3108::.443   ESTABLISHED
tcp4       0      0  10.3.146.161.51065     35.82.230.35.443       ESTABLISHED
tcp4       0      0  10.3.146.161.51055     162.159.136.234.443    ESTABLISHED
tcp4       0      0  10.3.146.161.51038     17.57.147.5.5223       ESTABLISHED
tcp6       0      0  *.51036                *.*                    LISTEN
tcp4       0      0  *.51036                *.*                    LISTEN
tcp4       0      0  127.0.0.1.14500        *.*                    LISTEN
```

Keeping state: the TCB

State for a TCP connection kept in Transmission Control Buffer (TCB)

- Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK'd segments, …
- When to allocate?
  - Server: listening on a connection*
  - Client: Initiating a connection (sending a SYN)
  - Server: accepting a new connection (receiving SYN)

Recall: the socket table

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.17.48.121.56915    192.168.1.58.7000      SYN_SENT
tcp4       0      0  172.17.48.121.56908    142.250.80.35.443      ESTABLISHED
tcp4       0      0  172.17.48.121.56887    13.225.231.50.80       ESTABLISHED
                                         . . .
tcp4       0      0  *.22                   *.*                    LISTEN
```

- Each connection has an associated TCB in the kernel
- For each packet, kernel maps the 5-tuple
  (tcp/udp, local IP, local port, remote IP, remote port) => socket
- Depending on socket type, socket contains TCB

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.17.48.121.56915    192.168.1.58.7000      SYN_SENT
tcp4       0      0  172.17.48.121.56908    142.250.80.35.443      ESTABLISHED
tcp4       0      0  172.17.48.121.56887    13.225.231.50.80       ESTABLISHED
                                        . . .
tcp4       0      0  *.22                   *.*                    LISTEN
```

Two "types" of sockets:

- "Normal" sockets
- Listen sockets

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.17.48.121.56887    13.225.231.50.80       ESTABLISHED
                                        . . .
tcp4       0      0  *.22                   *.*                    LISTEN
```

"Normal" sockets

- Connection between two specific endpoints
- Can send/recv data

Listen sockets

- Created by receiver to accept new connections
- When a client connects, client info gets queued by kernel
- When server process calls accept(), a new ("normal") socket is created between the server and that client

```
NOTA BENE:  This diagram is only a summary and must not be taken as
   the total specification.  Many details are not included.

                            +---------+ ---------\      active OPEN
                            |  CLOSED |            \    -----------
                            +---------+<---------\   \   create TCB
                              |     ^              \   \  snd SYN
                 passive OPEN |     |   CLOSE        \   \
                 ------------ |     | ----------       \   \
                  create TCB  |     | delete TCB         \   \
                              V     |                      \   \
          rcv RST (note 1)  +---------+            CLOSE    |    \
       -------------------->|  LISTEN |          ---------- |     |
      /                     +---------+          delete TCB |     |
     /           rcv SYN      |     |     SEND              |     |
    /           -----------   |     |    -------            |     V
+--------+      snd SYN,ACK  /       \   snd SYN          +--------+
|        |<-----------------           ------------------>|        |
|  SYN   |                    rcv SYN                     |  SYN   |
|  RCVD  |<-----------------------------------------------|  SENT  |
|        |                  snd SYN,ACK                   |        |
|        |------------------           -------------------|        |
+--------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +--------+
   |         --------------   |     |   -----------
   |                x         |     |     snd ACK
   |                          V     V
   |  CLOSE                 +---------+
   | -------                |  ESTAB  |
   | snd FIN                +---------+
```

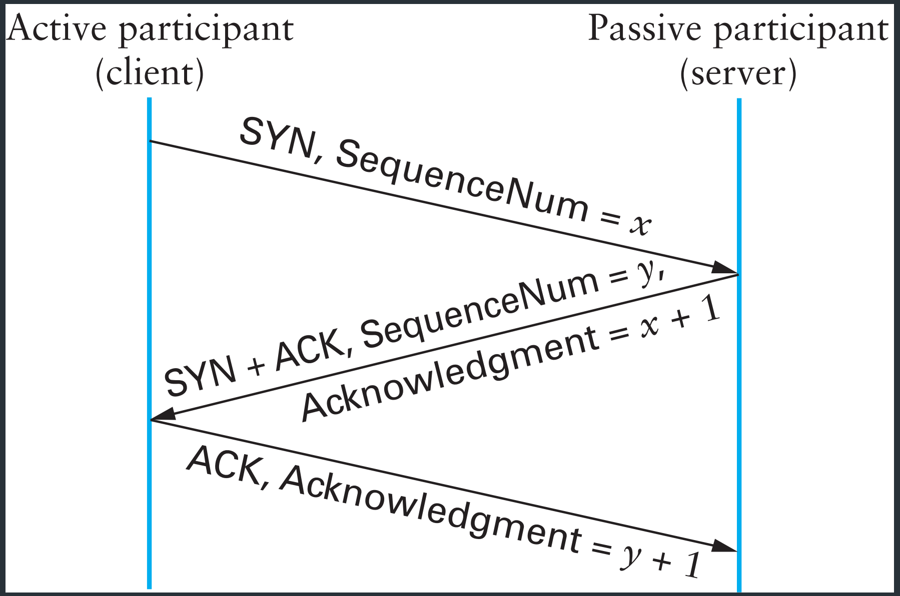
RFC 9293, Sec 3.3.2

SYN flooding

What happens if you send a someone huge number of SYN packets?

A hacky solution: SYN cookies

- Don't allocate TCB on first SYN
- Encode some state inside the initial sequence number that goes back to the client (in the SYN+ACK)
- What gets encoded?
  - Coarse timestamp
  - Hash of connection IP/port
  - Other stuff (implementation dependent)
- Better ideas?

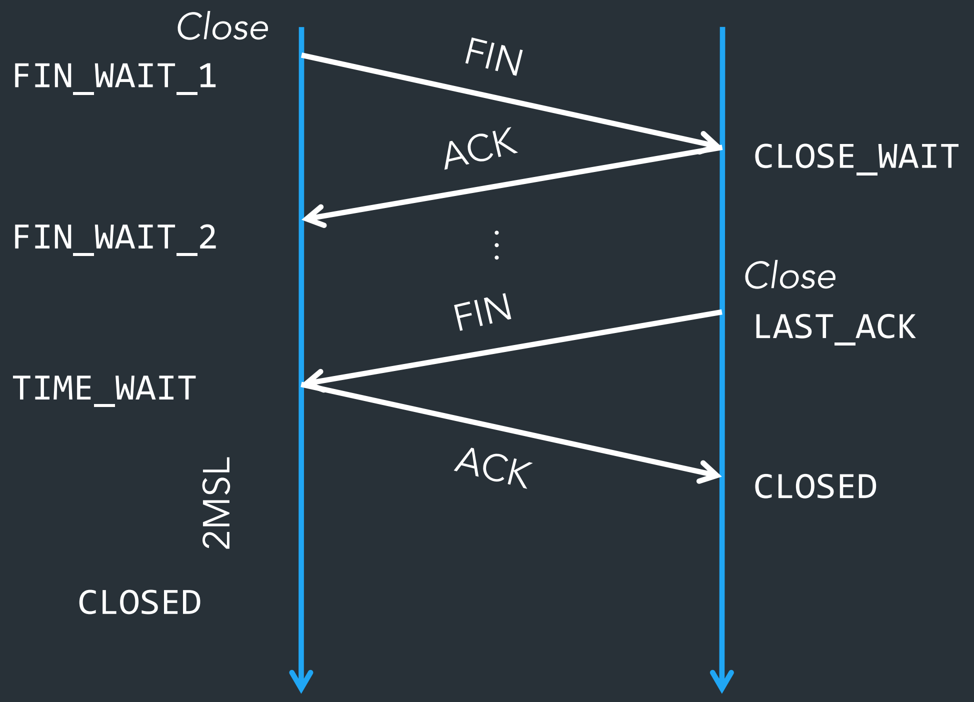


Next class

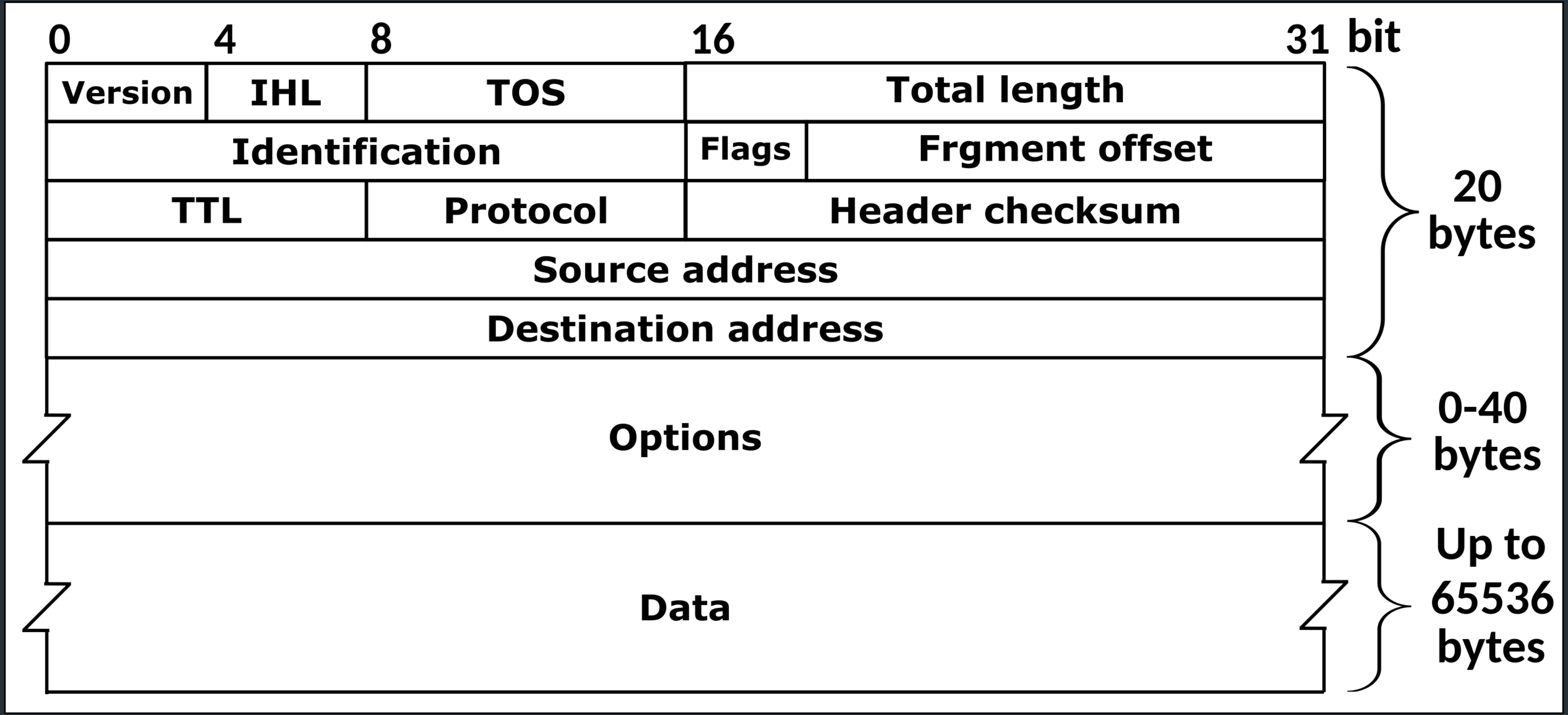
- Sending data over TCP

Connection Termination

- FIN bit says no more data to send
  - Caused by close or shutdown
  - Both sides must send FIN to close a connection
- Typical close



The IPv4 Header



Defined by RFC 791
RFC (Request for Comment): defines network standard