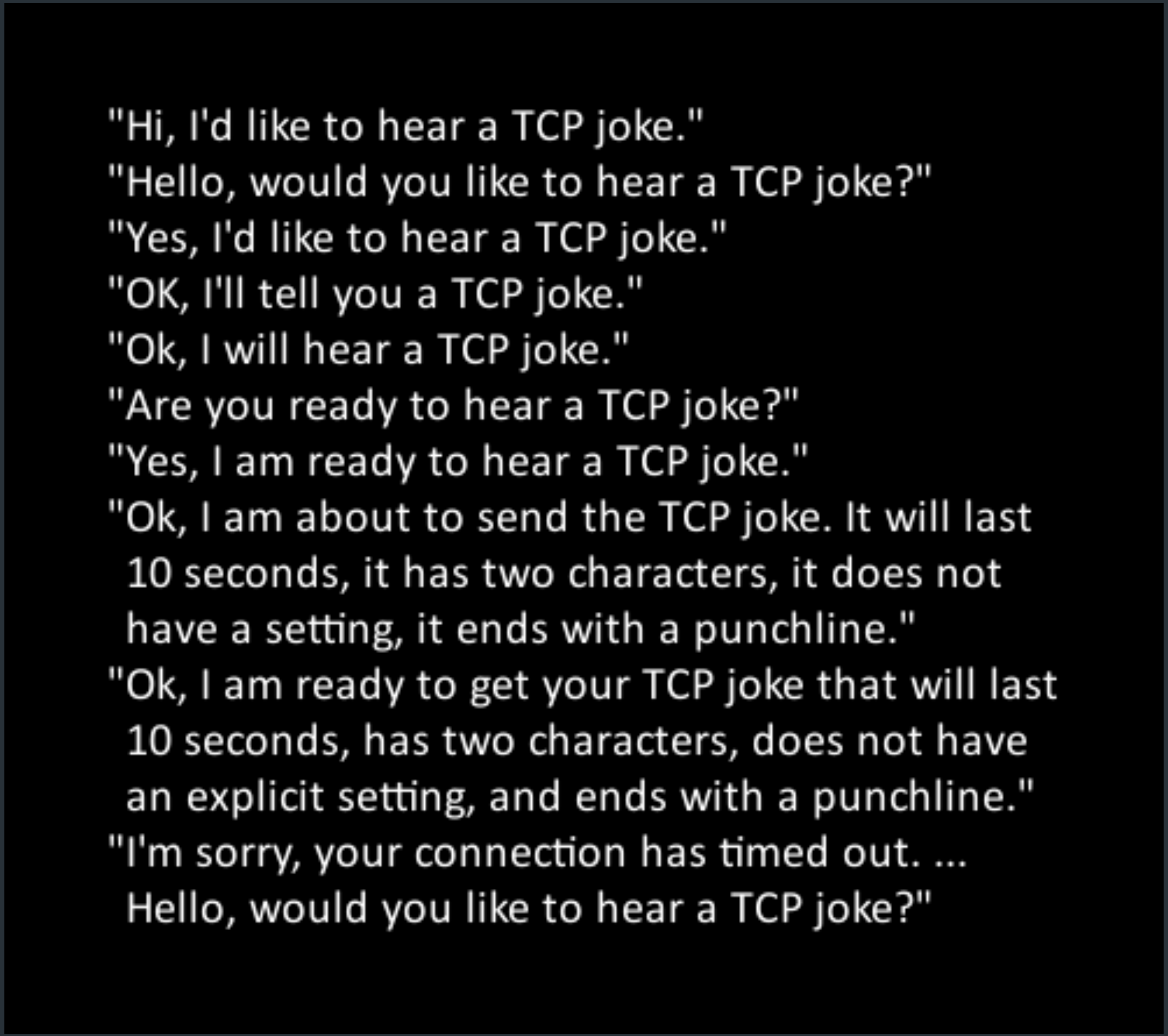
CSCI-1680
Transport Layer II

Data over TCP: Flow Control

Nick DeMarinis

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



"Hi, I'd like to hear a TCP joke."

"Hello, would you like to hear a TCP joke?"

"Yes, I'd like to hear a TCP joke."

"OK, I'll tell you a TCP joke."

"Ok, I will hear a TCP joke."

"Are you ready to hear a TCP joke?"

"Yes, I am ready to hear a TCP joke."

"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."

"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."

"I'm sorry, your connection has timed out. ... Hello, would you like to hear a TCP joke?"

Administrivia

- IP project grading: happening now! Sign up for a meeting if you haven't already
- **TCP assignment: out now—start early!**
  - Gearup I: Thursday 10/26 5-7pm
  - Milestone 1: schedule meeting on/before Thursday, November 2

TCP: The story so far

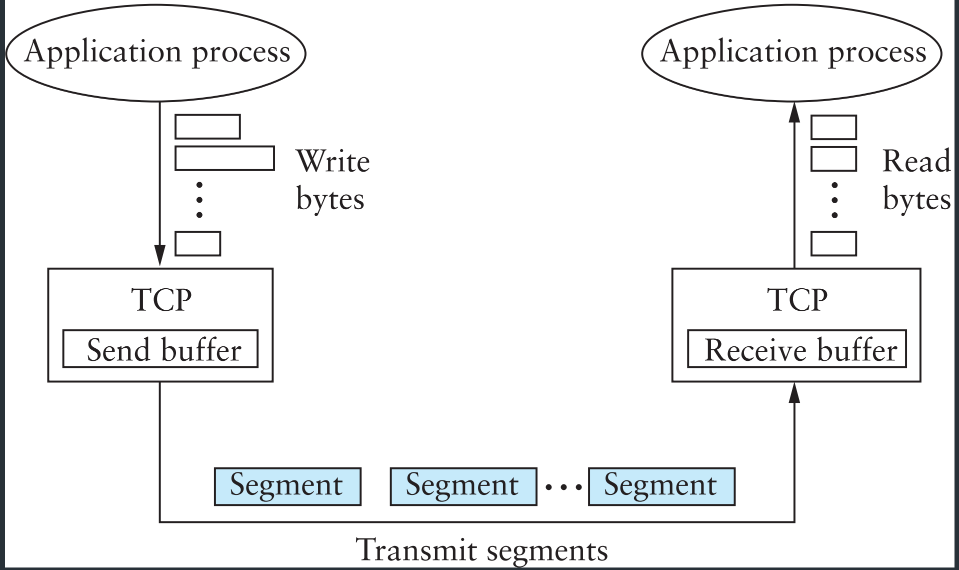
Last lecture

- Sockets
- TCP: connection setup

Today

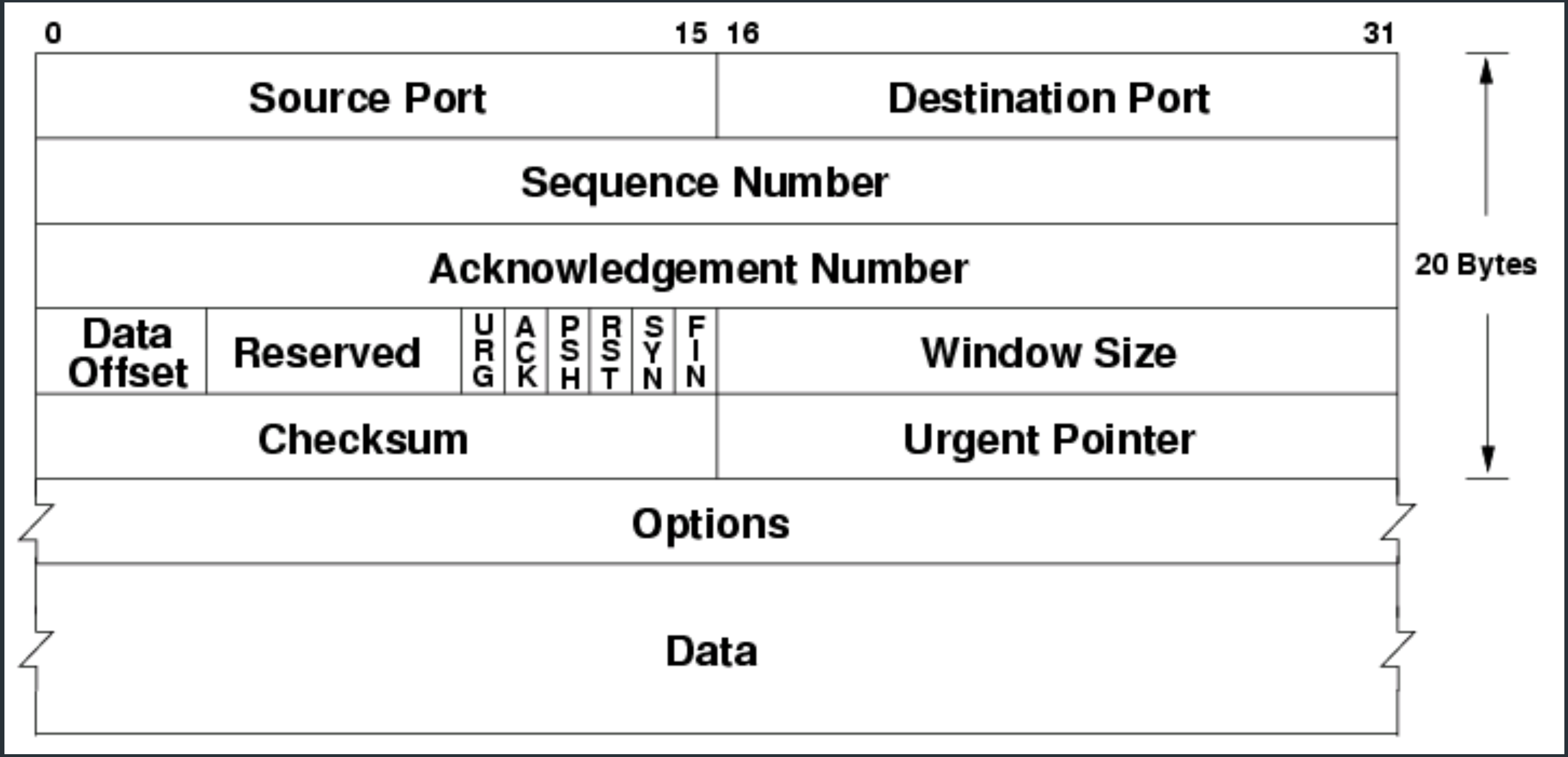
- Basic flow control: How to send data
- Connection teardown

TCP – Transmission Control Protocol



TCP provides a “reliable, connection oriented, full duplex ordered byte stream”

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

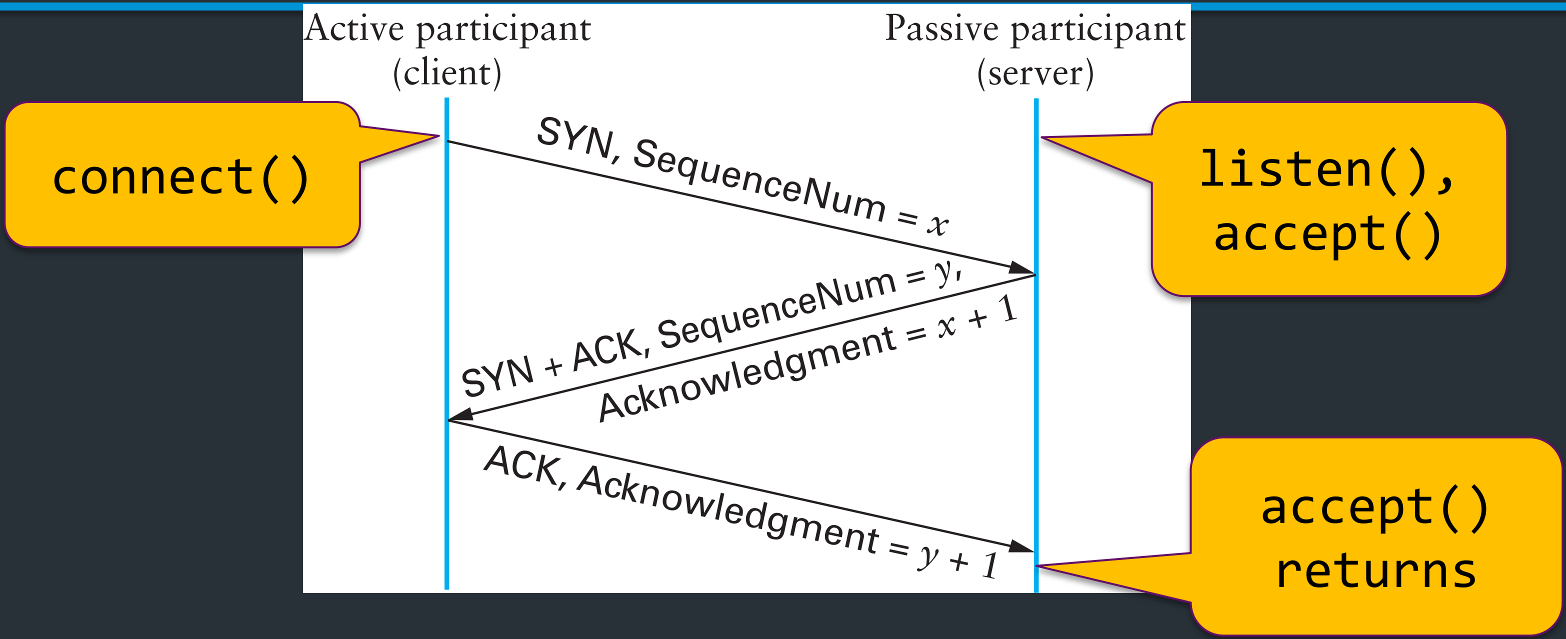
- RST: reset connection (used for errors)
- PSH: push data to the application immediately
- URG: whether there is urgent data

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

Establishing a Connection



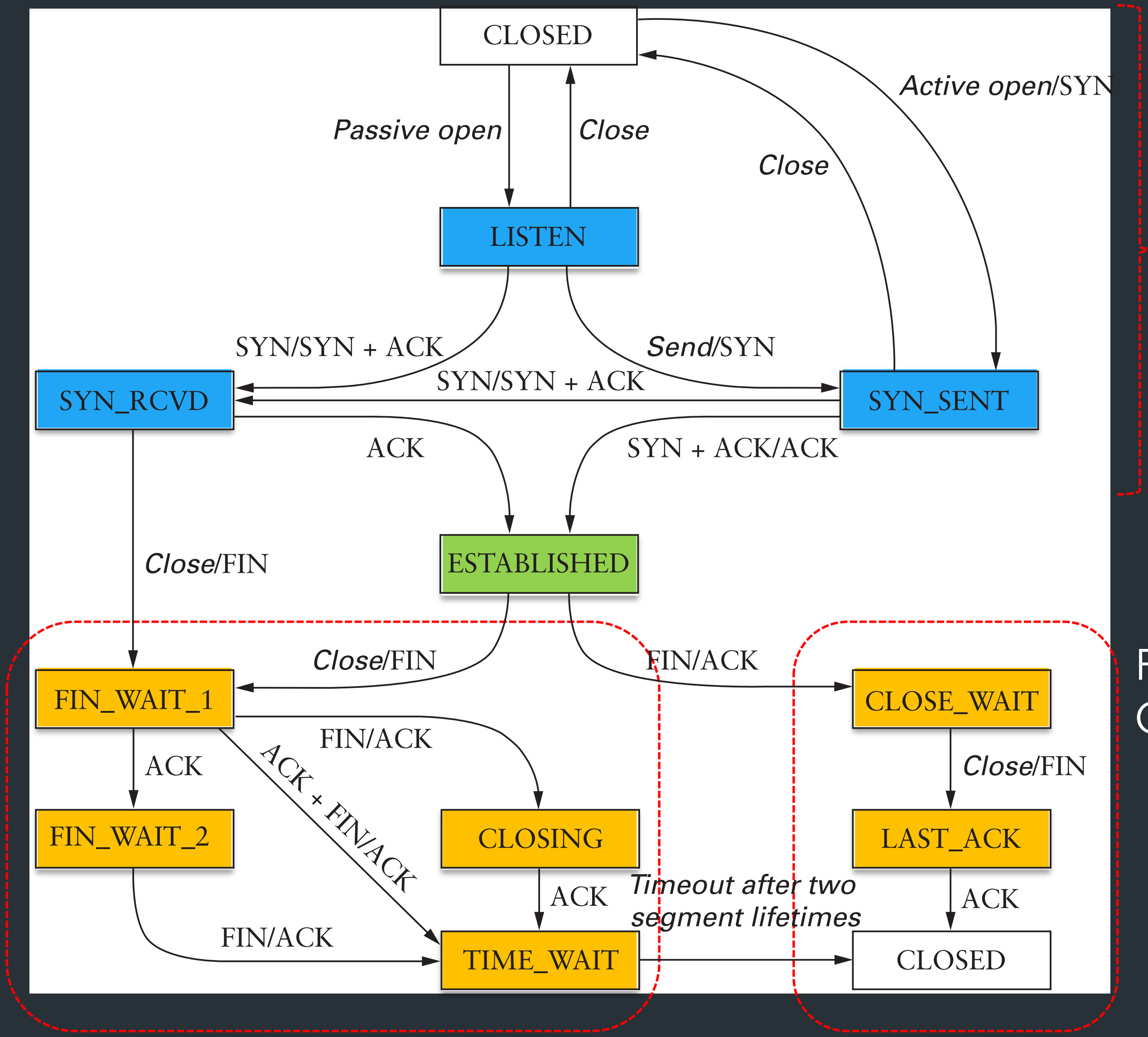
`connect()`

`listen(), accept()`

`accept() returns`

- 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

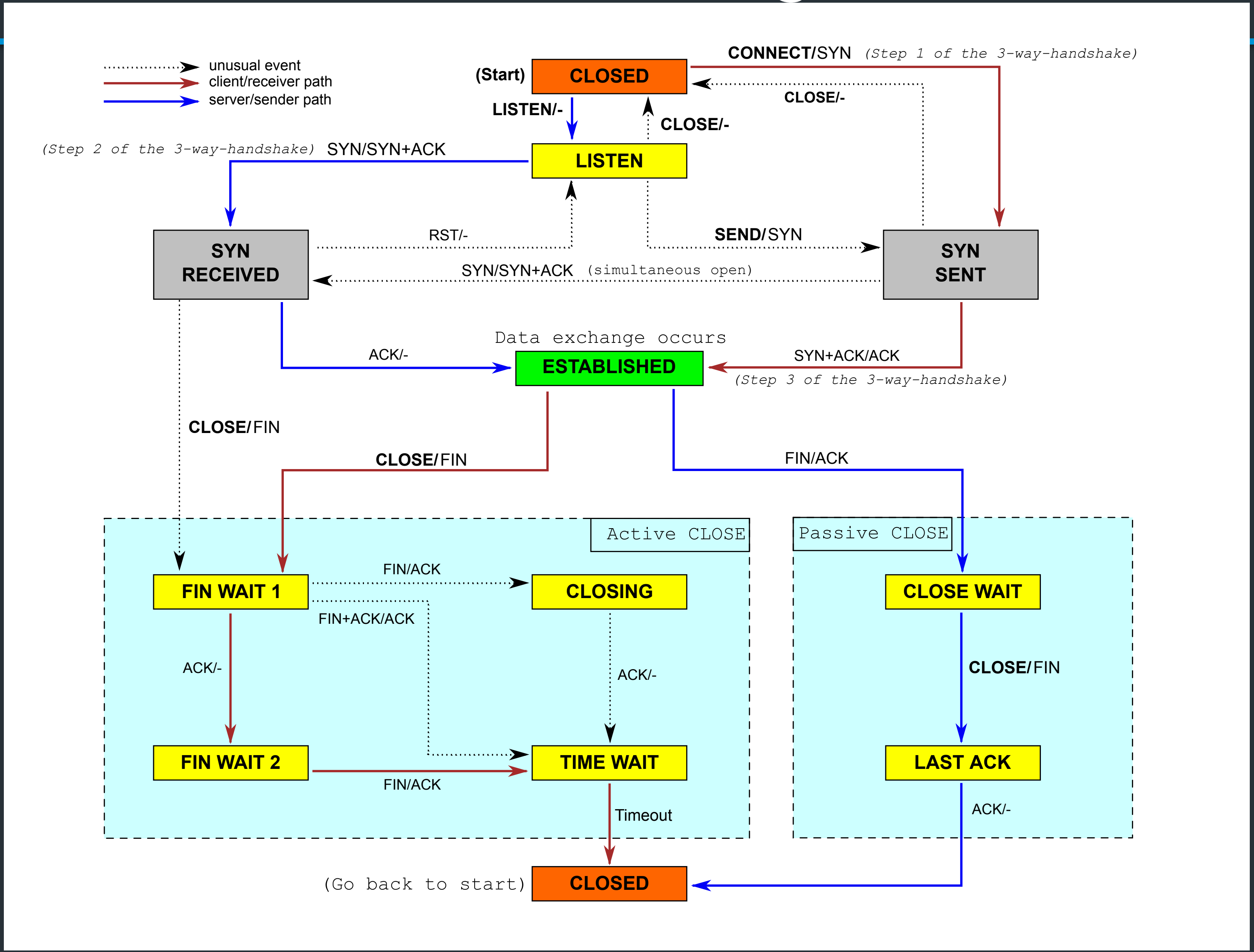


Connection Establishment

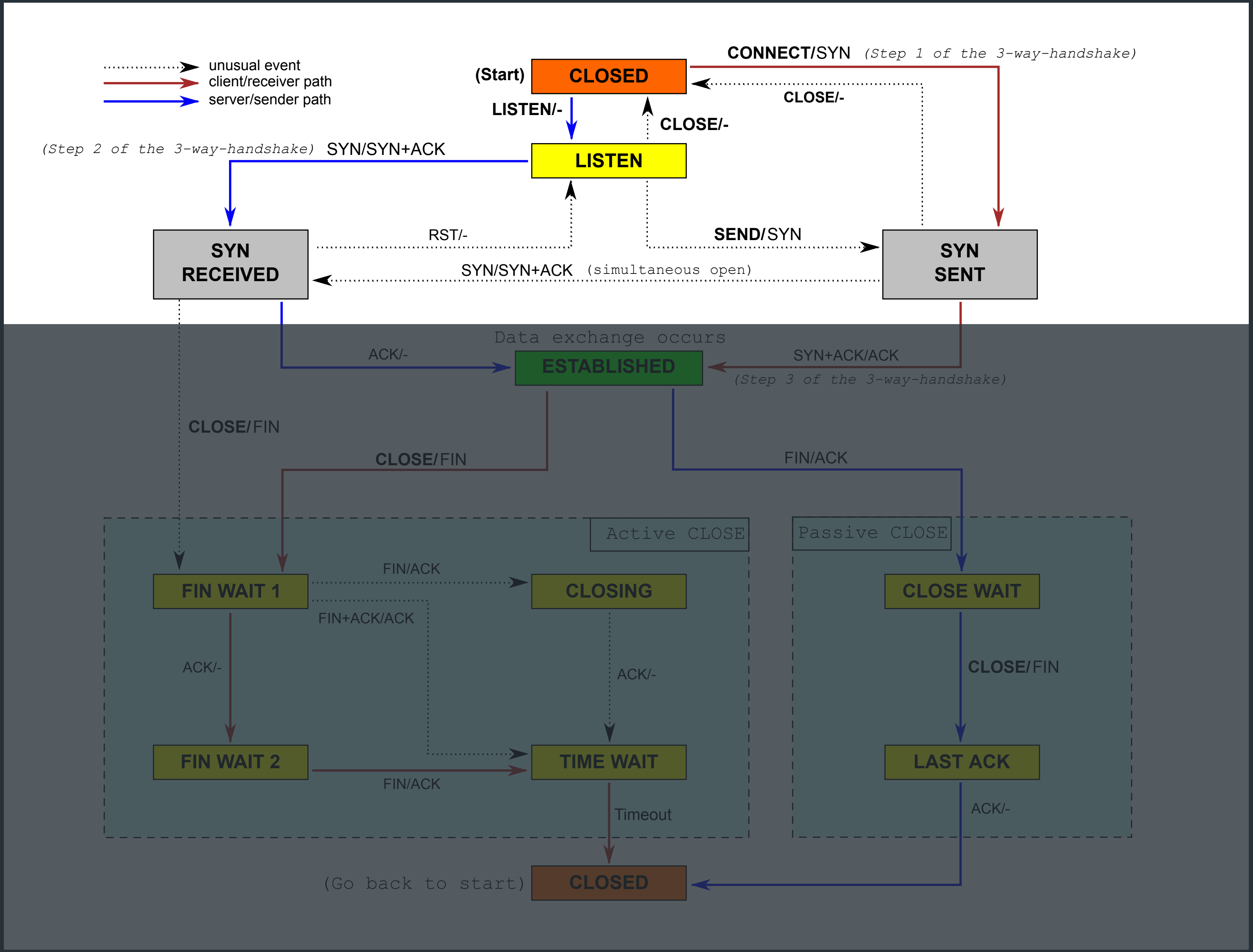
Active close:
Can still receive

Passive close:
Can still send!

TCP State Diagram



We are now here



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?

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)

⇒ When to deallocate?

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, …

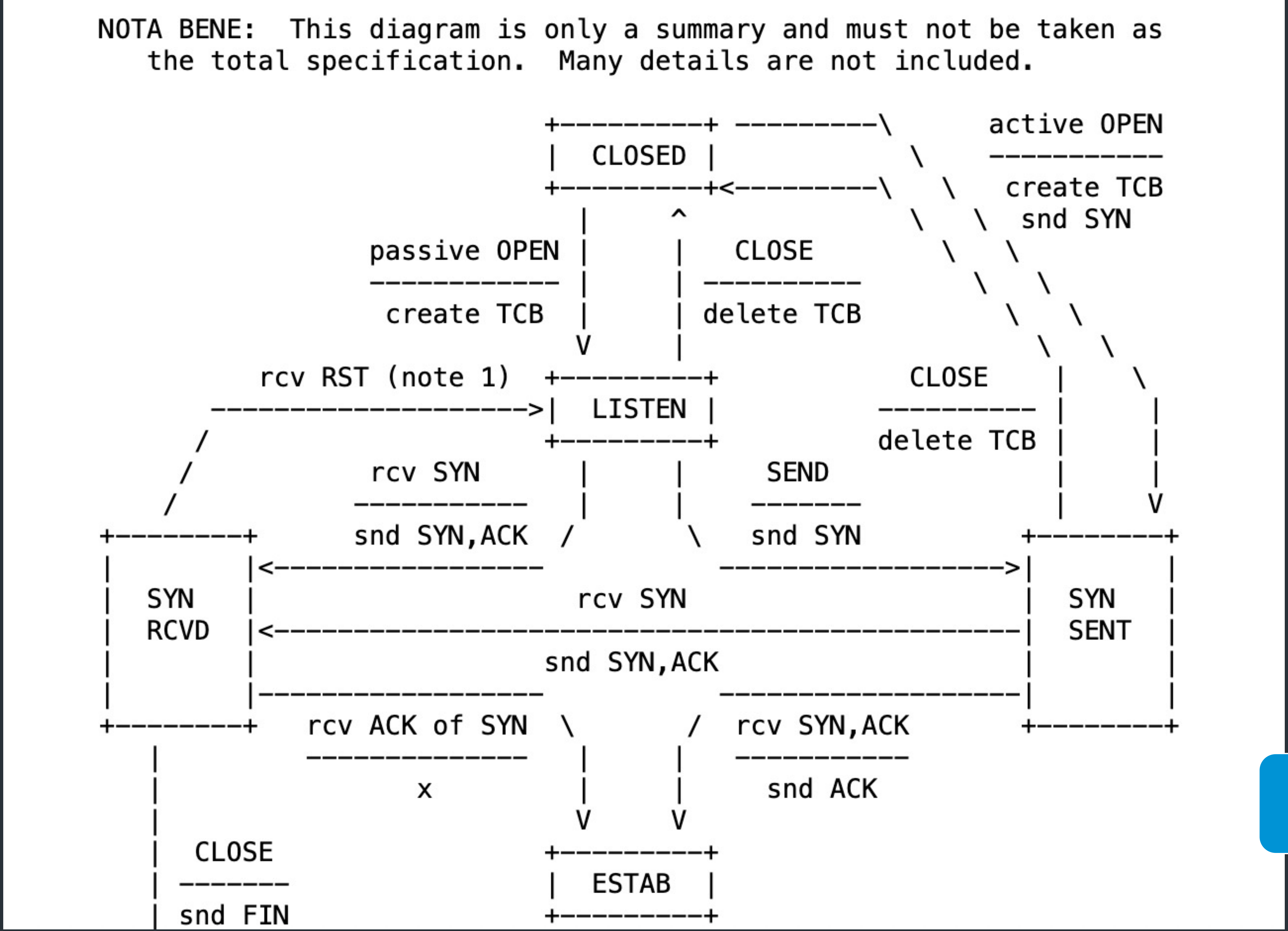
When to allocate?

- Server: listening on a connection*
- Client: Initiating a connection (sending a SYN)
- Server: accepting a new connection (receiving SYN)

When to deallocate?

Only after connection termination is fully completed (CLOSED state)

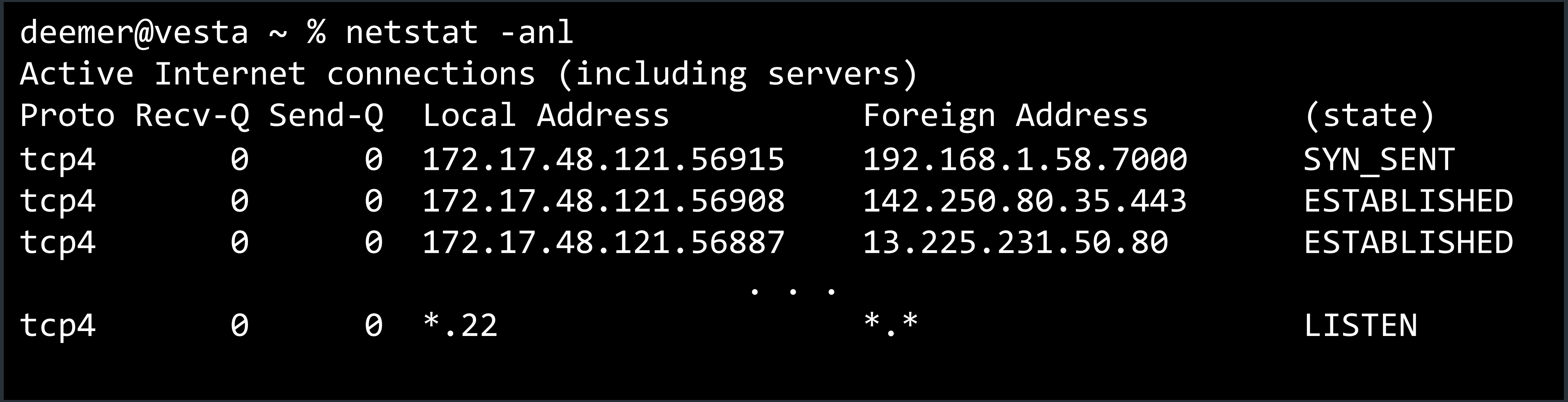
=> If no state, can't meaningfully respond to packet!



```
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

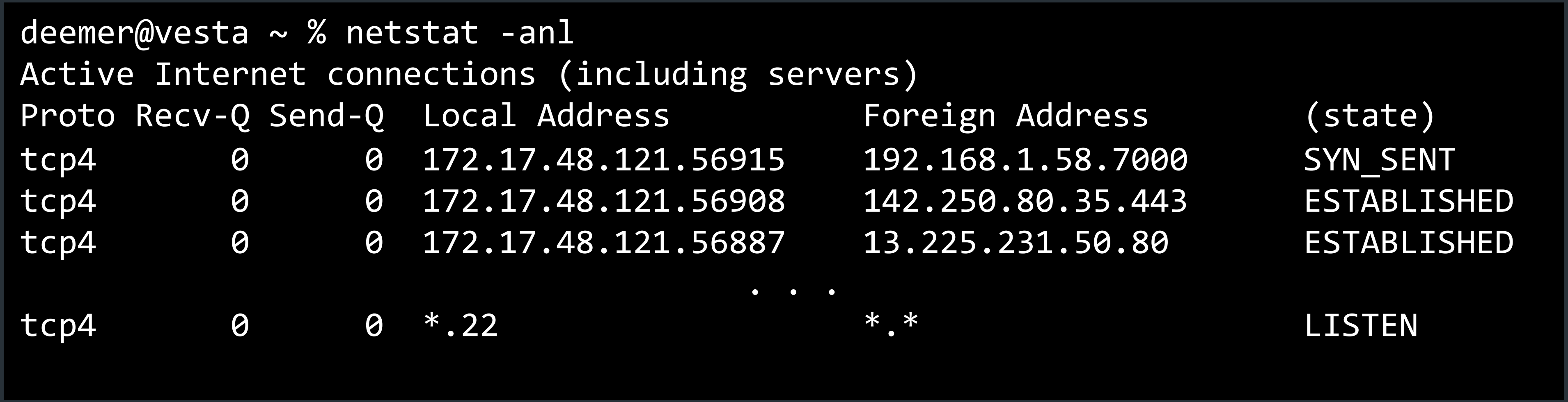
Recall: the socket table



```
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
- Depending on socket type, socket contains TCB

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
- Depending on socket type, socket contains TCB

⇒ For each packet, kernel maps the 5-tuple
(tcp/udp, local IP, local port, remote IP, remote port) => socket

5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn

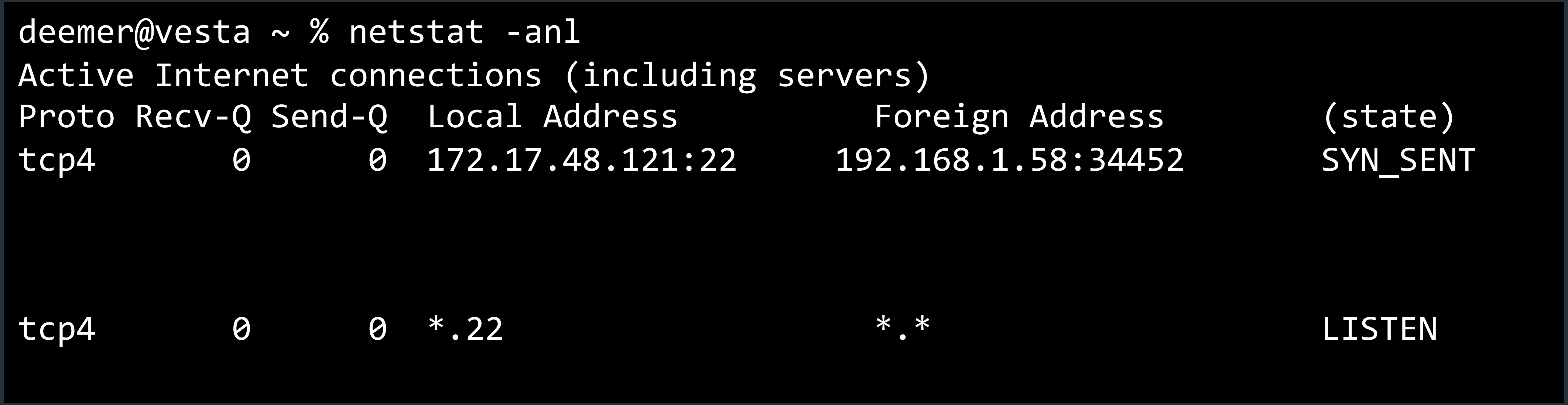
- Kernel maintains socket table: maps (5-tuple) => Socket

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)




tcp4       0      0  *.22                   *.*                    LISTEN
```

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

5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn

- Kernel maintains socket table: maps (5-tuple) => Socket

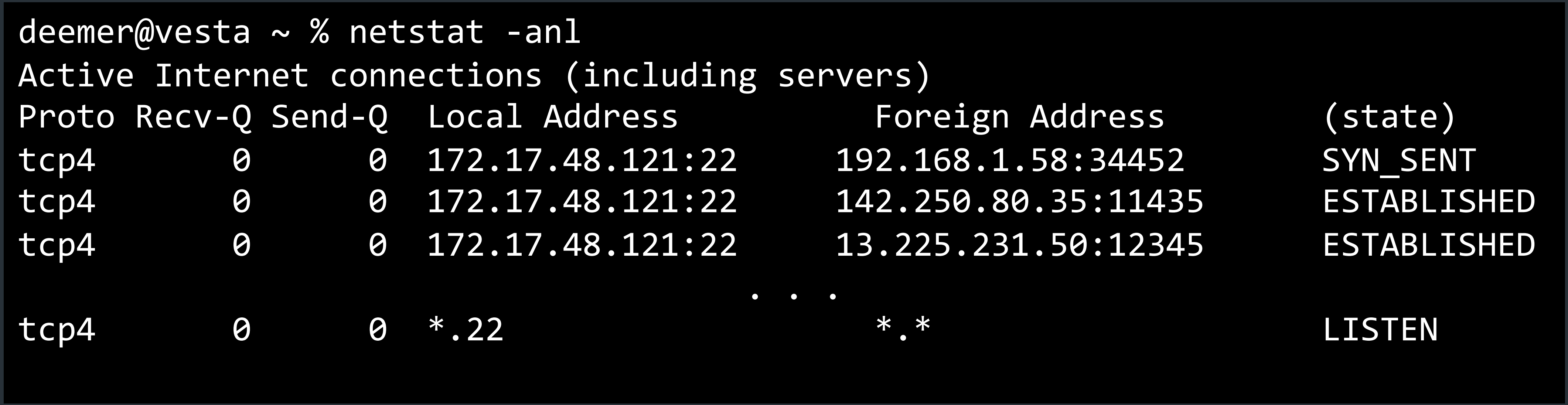


```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.17.48.121:22     192.168.1.58:34452       SYN_SENT



tcp4       0      0  *.22                   *.*                    LISTEN
```

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

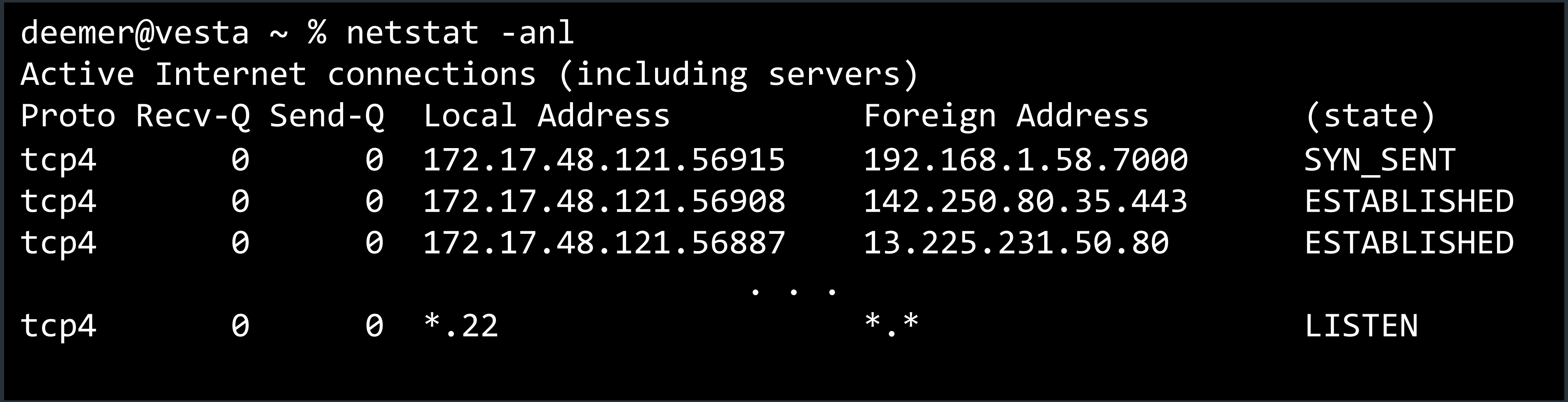
5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn

- Kernel maintains socket table: maps (5-tuple) => Socket



```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  172.17.48.121:22     192.168.1.58:34452       SYN_SENT
tcp4       0      0  172.17.48.121:22     142.250.80.35:11435      ESTABLISHED
tcp4       0      0  172.17.48.121:22     13.225.231.50:12345      ESTABLISHED
                                        . . .
tcp4       0      0  *.22                   *.*                    LISTEN
```

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



```
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

How to pick the initial sequence number?

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

How to pick the initial sequence number?

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

  => Someone might guess the value!

How to pick the initial sequence number?

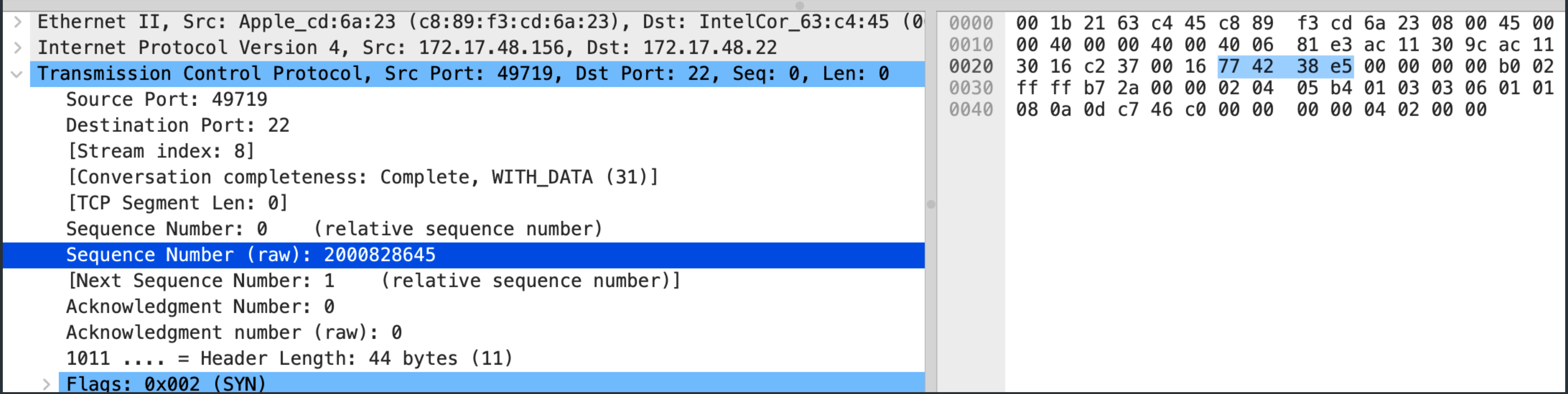
- Protocols based on *relative* seq. numbers based on starting value
- Why not start at 0?

  => Someone might guess the value!

=> 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



```
> Ethernet II, Src: Apple_cd:6a:23 (c8:89:f3:cd:6a:23), Dst: IntelCor_63:c4:45 (0
> Internet Protocol Version 4, Src: 172.17.48.156, Dst: 172.17.48.22
v Transmission Control Protocol, Src Port: 49719, Dst Port: 22, Seq: 0, Len: 0
    Source Port: 49719
    Destination Port: 22
    [Stream index: 8]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 2000828645
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1011 .... = Header Length: 44 bytes (11)
  > Flags: 0x002 (SYN)
```

```
0000  00 1b 21 63 c4 45 c8 89  f3 cd 6a 23 08 00 45 00
0010  00 40 00 00 40 00 40 06  81 e3 ac 11 30 9c ac 11
0020  30 16 c2 37 00 16 77 42  38 e5 00 00 00 00 b0 02
0030  ff ff b7 2a 00 00 02 04  05 b4 01 03 03 06 01 01
0040  08 0a 0d c7 46 c0 00 00  00 00 04 02 00 00
```

Observation: new connections use memory!

Observation: new connections use memory!

What happens if you send a someone lots of SYN packets?

Observation: new connections use memory!

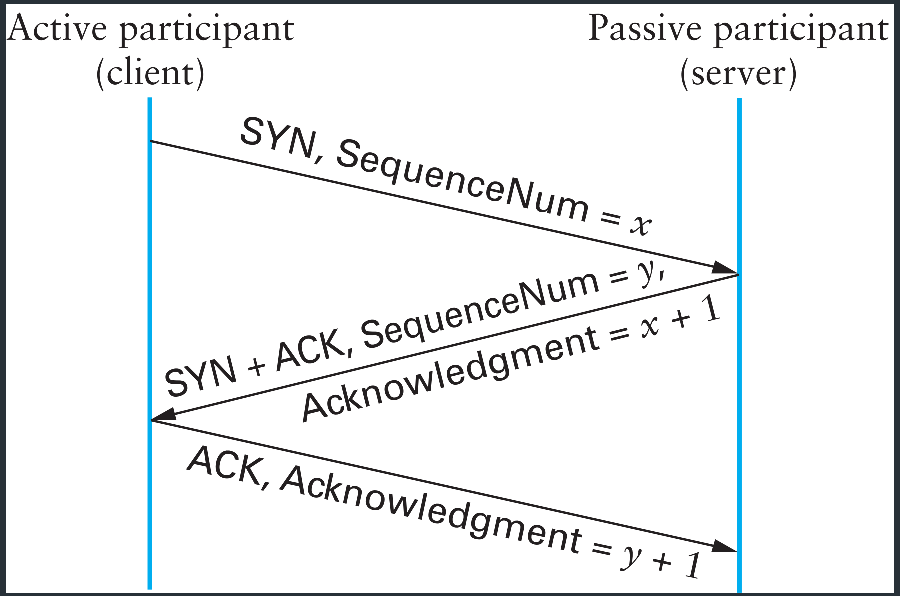
What happens if you send a someone lots of SYN packets?

SYN flood => type of Denial of Service (DOS) attack

SYN flood => type of Denial of Service (DOS) attack
=> Especially bad when attack traffic comes from multiple sources (more on this later)

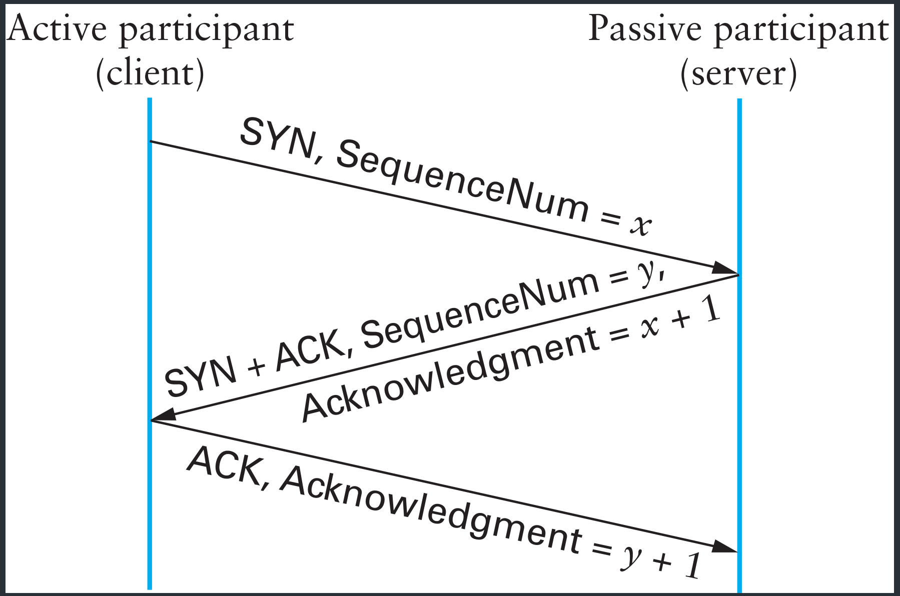
A hacky solution: SYN cookies

- 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?



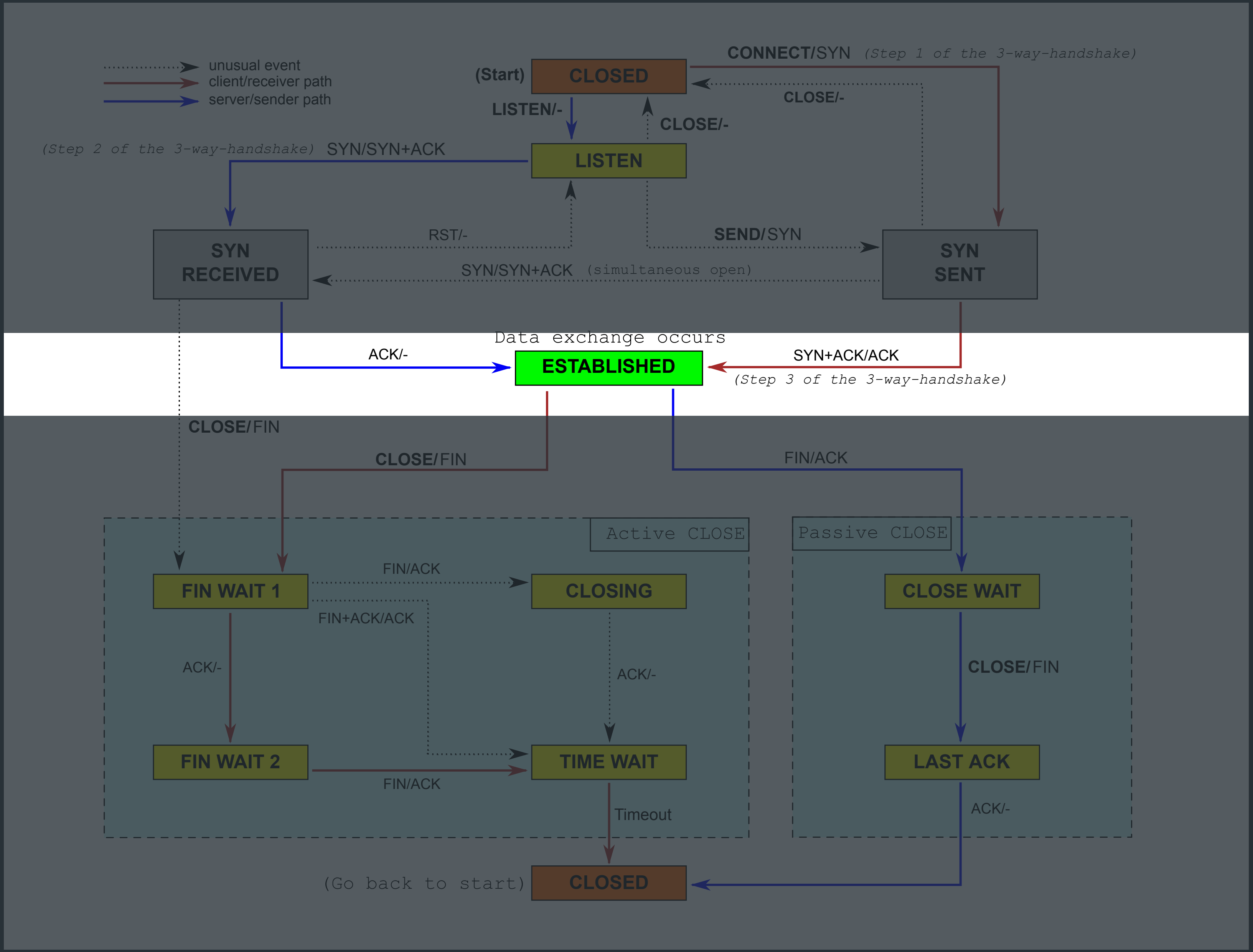
A hacky solution: SYN cookies

- 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)



- *Nowadays: filtering in kernel (or in network) on number of new connections per time (esp. on servers)*
  = > More on this later!

Sending data



Sending data

Flow control: don’t send more data than the receiver can handle

- TCP stack divides data into packets called *segments*

Questions

- When to send data?
- How much data to send?
  - – Data is sent in MSS-sized segments
    - MSS = Maximum Segment Size (TCP packet that can fit in an IP packet)
    - Chosen to avoid fragmentation

Sending data: the basic idea

- Start: app calls Send(), loads send buffer

Sending data: the basic idea

- Start: app calls Send(), loads send buffer
- TCP stack divides data into packets called segments

Sending data: the basic idea

- Start: app calls Send(), loads send buffer
- TCP stack divides data into packets called segments

Key challenges

- When to send data?
- How much data to send?

Sending data: the basic idea

- Start: app calls Send(), loads send buffer
- TCP stack divides data into packets called segments

Key challenges

- When to send data?
- How much data to send?

⇒ Flow control (now): don’t send more data than the receiver can handle

⇒ *Congestion control (much later) don’t send more data than the network can handle*

Terminology: MSS

MSS: Maximum segment size

- Largest segment a TCP can send
- Can be configurable

- Nowadays: sender and receiver negotiate using TCP options (out of scope for this class)

=> For project: just a fixed value

Simplest TCP sender: stop and wait

Simplest method: Stop and Wait

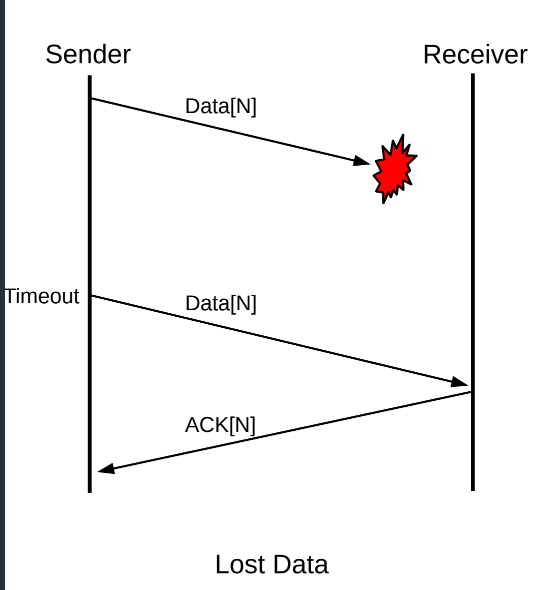
Consider sending one packet at a time

- S: Send packet, wait
- R: Receive packet, send ACK
- S: Receive ACK, send next packet

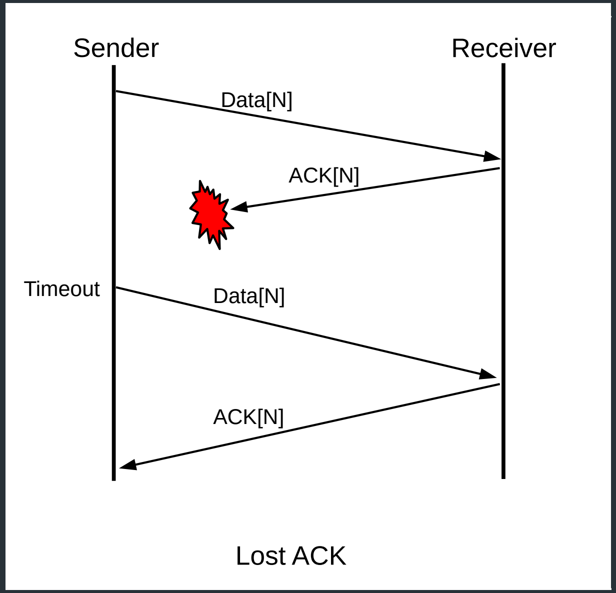
  OR

  No ACK within some time (RTO), timeout and retransmit

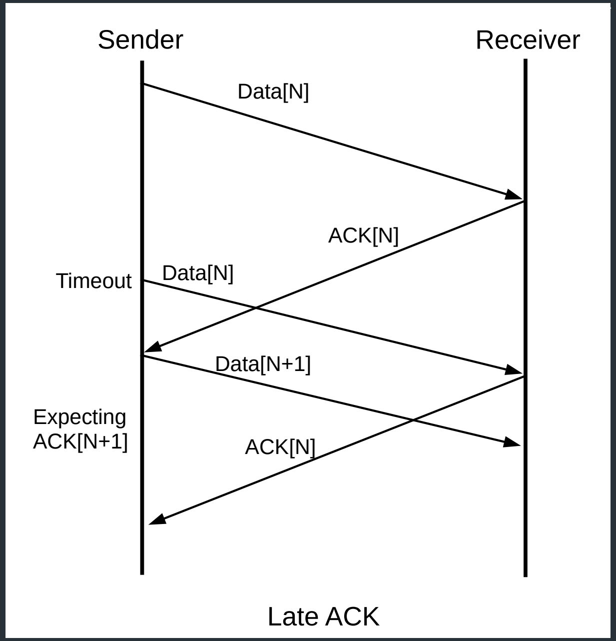
What can go wrong?



Lost Data

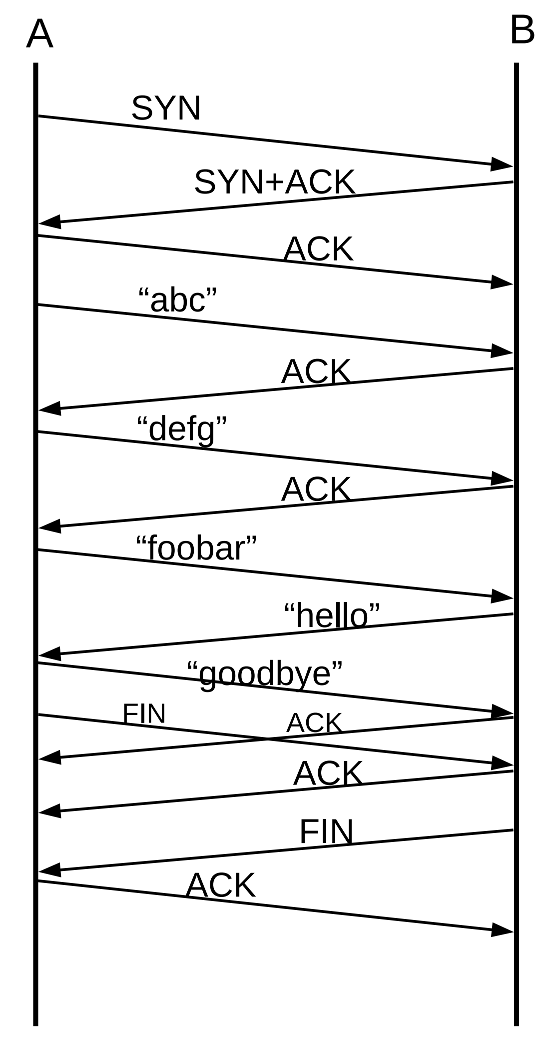


Lost ACK



Late ACK

Sequence number example



	A sends	B sends
1	SYN, **seq=0**	
2		SYN+ACK, seq=0, **ack=1** (expecting)
3	ACK, **seq=1**, ack=1 (ACK of SYN)	
4	“abc”, **seq=1**, ack=1	
5		ACK, seq=1, **ack=4**
6	“defg”, **seq=4**, ack=1	
7		seq=1, **ack=8**
8	“foobar”, **seq=8**, ack=1	
9		seq=1, **ack=14**, “hello”
10	**seq=14**, ack=6, “goodbye”	
11,12	**seq=21**, ack=6, FIN	seq=6, **ack=21** ;; ACK of “goodbye”, crossing packets
13		seq=6, **ack=22** ;; ACK of FIN
14		seq=6, **ack=22**, FIN
15	**seq=22**, ack=7 ;; ACK of FIN	

Problems?

Can we do better?

Better Flow Control: Sliding window

- Part of TCP specification (even before 1988)
- Send multiple packets at once, based on a *window*
- Receiver uses window header field to tell sender how much space it has

TCP and buffering

Recall: TCP stack responsibilities

- Sender: breaking application data into segments
- Receiver: receiving segments, reassembling them in order

- Need to buffer data

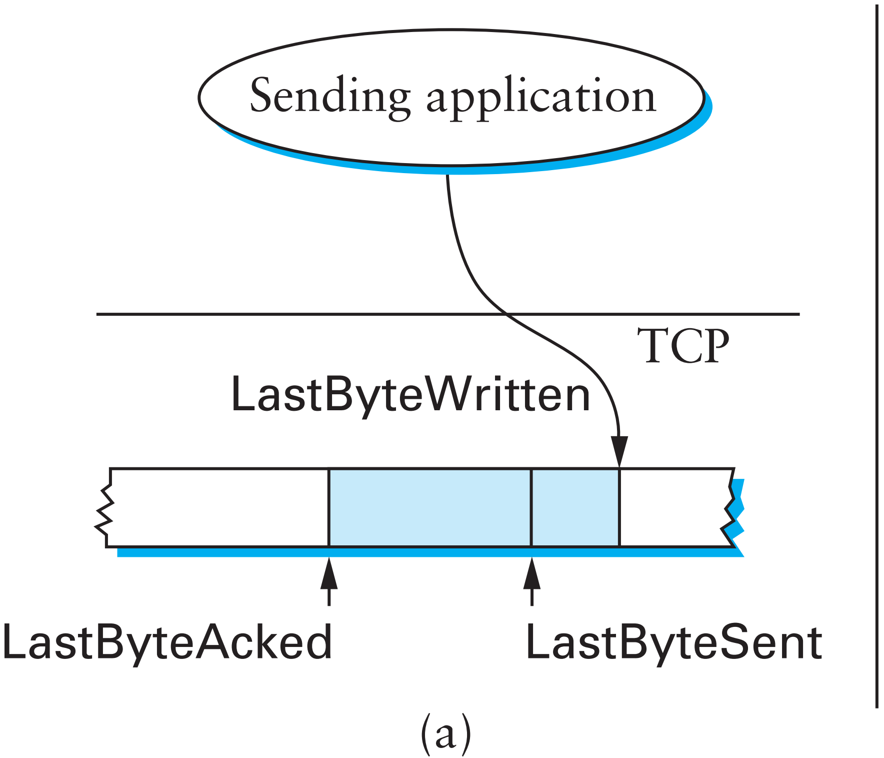
Sliding window: in abstract terms

- Window of size w
- Can send at most w packets before waiting for an ACK
- Goal
  - Network “pipe” always filled with data
  - ACKs come back at rate data is delivered => “self-clocking”

Sender example

Receiver example

Flow Control: Sender



Invariants

- LastByteSent – LastByteAcked <= AdvertisedWindow
- EffectiveWindow = AdvertisedWindow – (BytesInFlight)
- LastByteWritten – LastByteAcked <= MaxSendBuffer

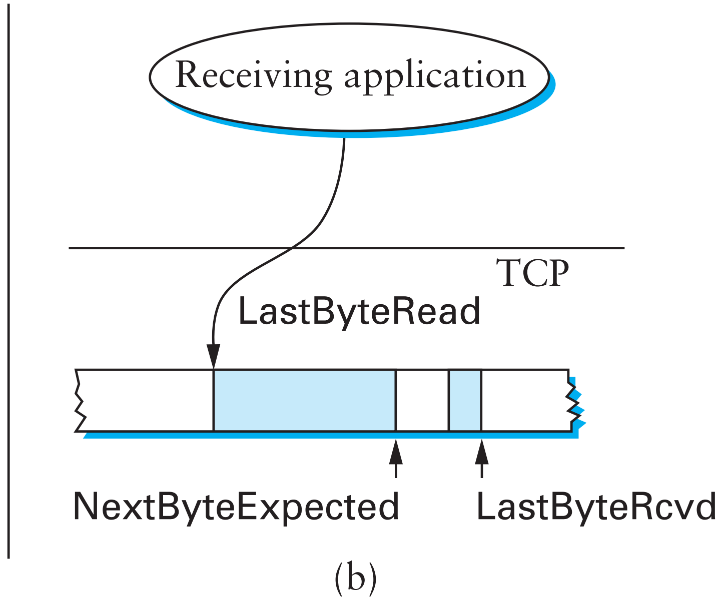
Useful Sliding Window Terminology:
RFC 9293, Sec 3.3.1

Flow control: receiver

Useful Sliding Window Terminology:
RFC 9293, Sec 3.3.1

- Can accept data if space in window
- Available window =
  BufferSize– ((NextByteExpected-1) - LastByteRead

- On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    - Deliver to application (Update LastByteReceived)
    - If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    - Queue as early arrival

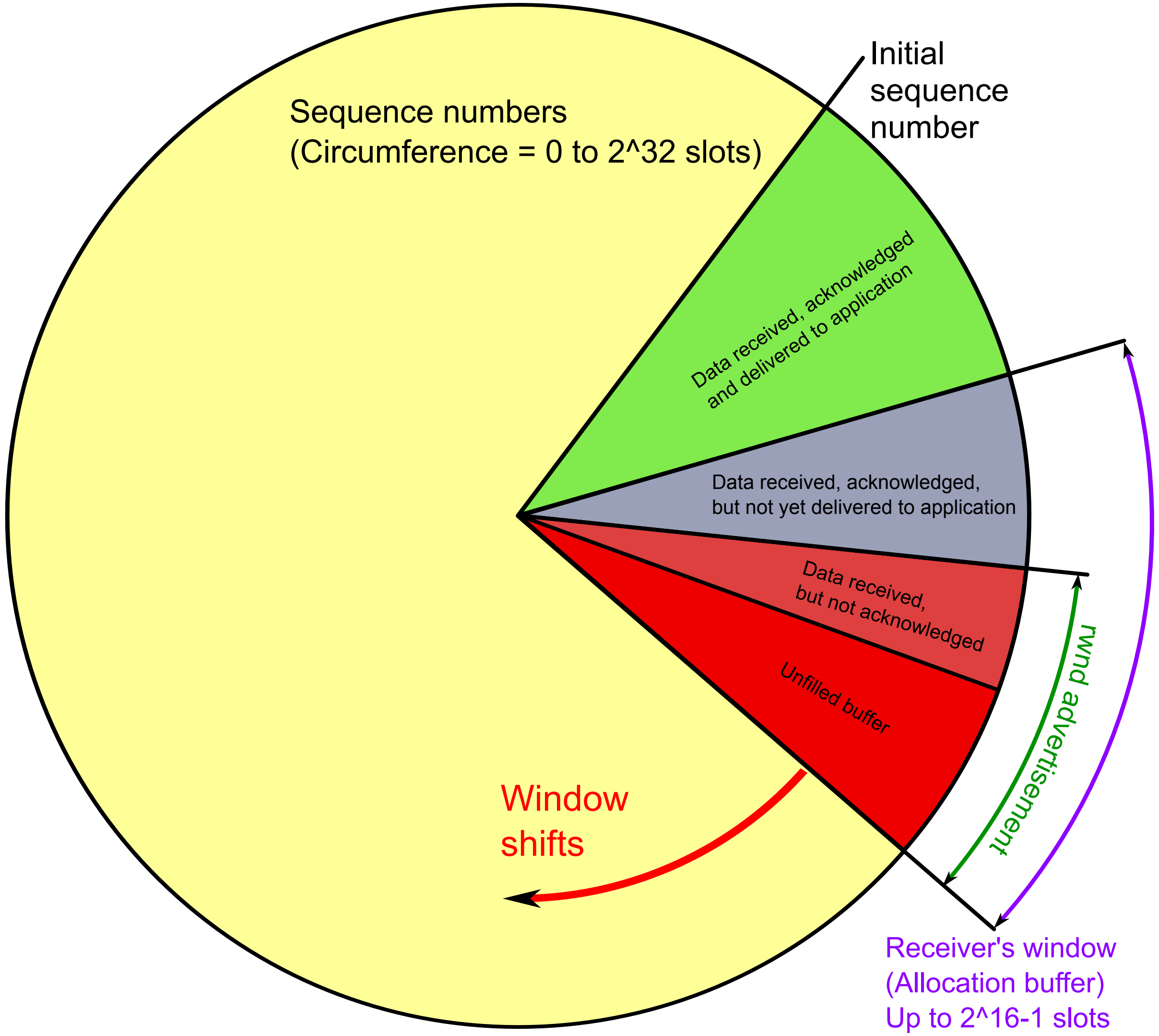


- Send ACK for highest contiguous byte received, available window

Flow Control

- Advertised window can fall to 0
  - How?
  - Sender eventually stops sending, blocks application

- Resolution: zero window probing: sender sends 1-byte segments until window comes back > 0



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

Some Visualizations

- Normal conditions: https://www.youtube.com/watch?v=zY3Sxvj8kZA

- With packet loss: https://www.youtube.com/watch?v=lk27yiITOvU

How do ACKs work?

- ACK contains *next expected sequence number*
- If one segment is missed but new ones received, send duplicate ACK
- Retransmit when:
  - Receive timeout (RTO) expires
  - Possibly other conditions, for certain TCP variants (eg. 3 dup ACKs)

- How to set RTO?

When to time out?

RFC793, Sec 3.7

Should expect an ACK within one Round Trip Time (RTT)

- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version ("smoothed RTT"):

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

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

α = "Smoothing factor": .8-.9

$\boldsymbol{\beta}$ = "Delay variance factor": 1.3—2.0

This is only the beginning…

- Problem 1: what if segment is a retransmission?
  - Solution: don’t update RTT if segment was retransmitted

- Problem 2: RTT can have high variance
  - Initial implementation doesn’t account for this
  - Congestion control: modeling network load