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

Administrivia

- Sign up for IP grading: this week and next week
- TCP assignment: out now—start early!
  - Gear-up session soon, details forthcoming
  - The next few lectures will help you
  - Schedule Milestone I meeting by Thurs, April 14
- More details soon on what happens after TCP

Topics for today

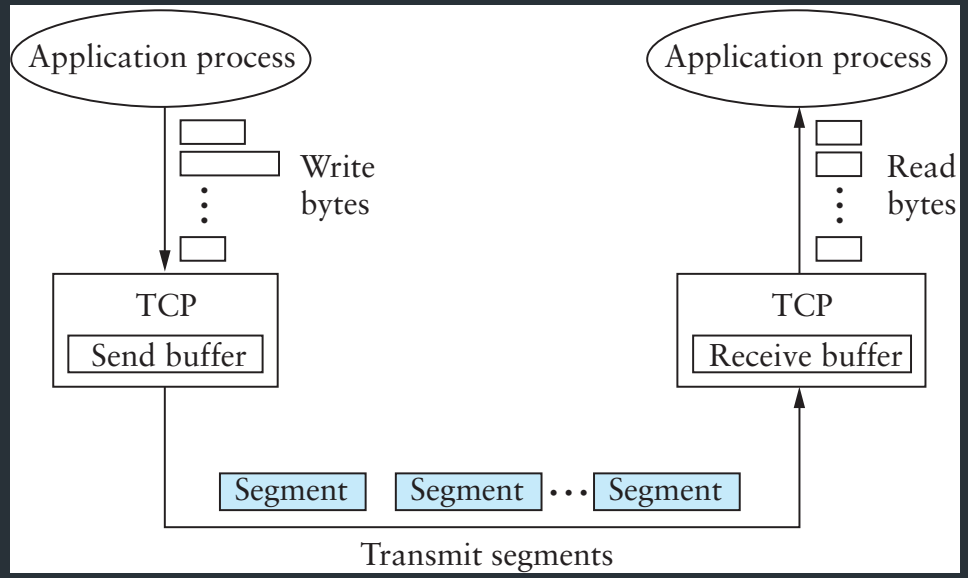
From before break

- TCP: connection setup
- Sockets

New stuff: How to send data

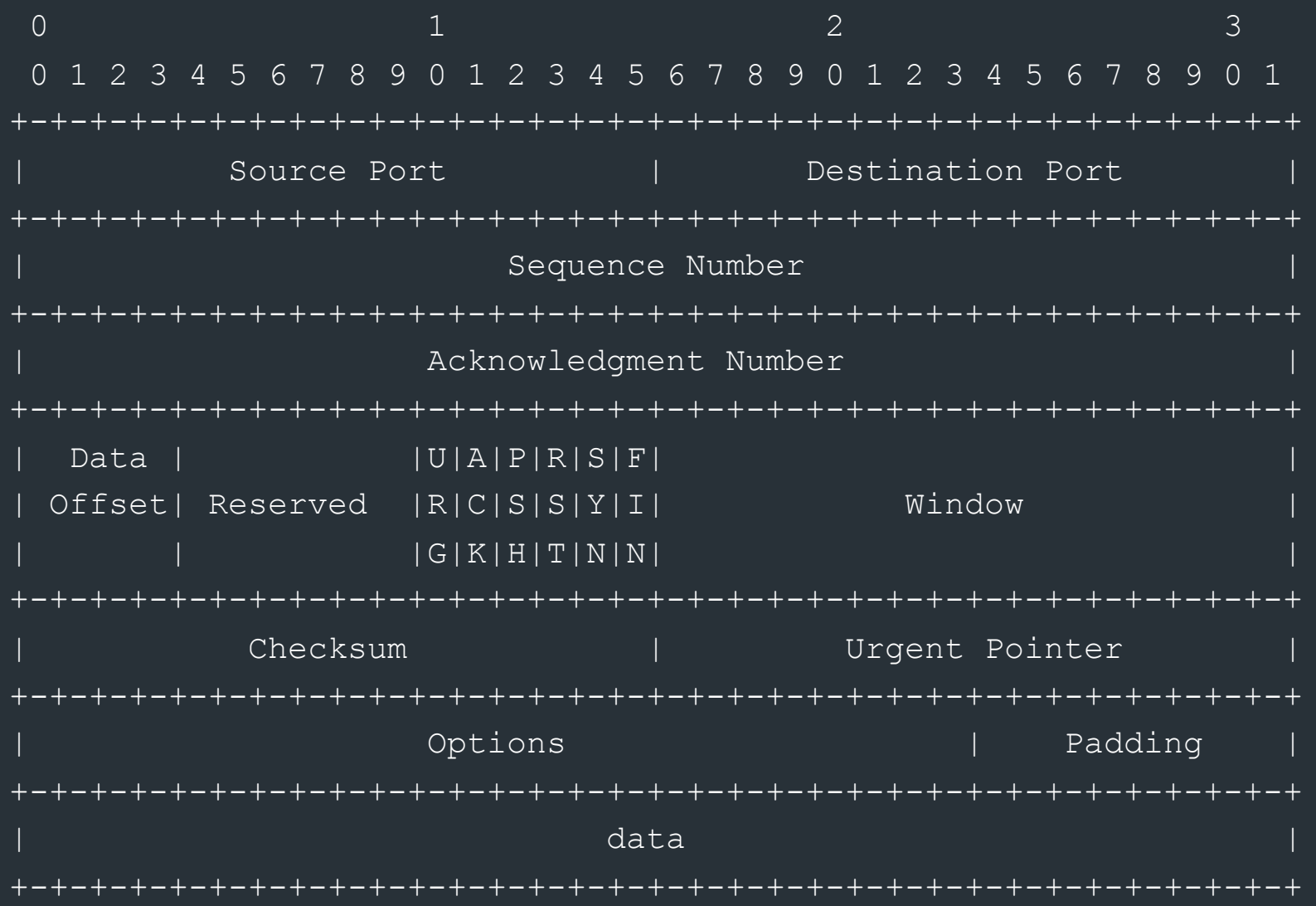
- Flow control: how to send data without overwhelming *receiver*
- Congestion control: how to send data without overwhelming *network*

TCP – Transmission Control Protocol



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

TCP Header


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data Offset
Reserved
U R G
A C K
P S H
R S T
S Y N
F I N
Window
Checksum
Urgent Pointer
Options
Padding
data

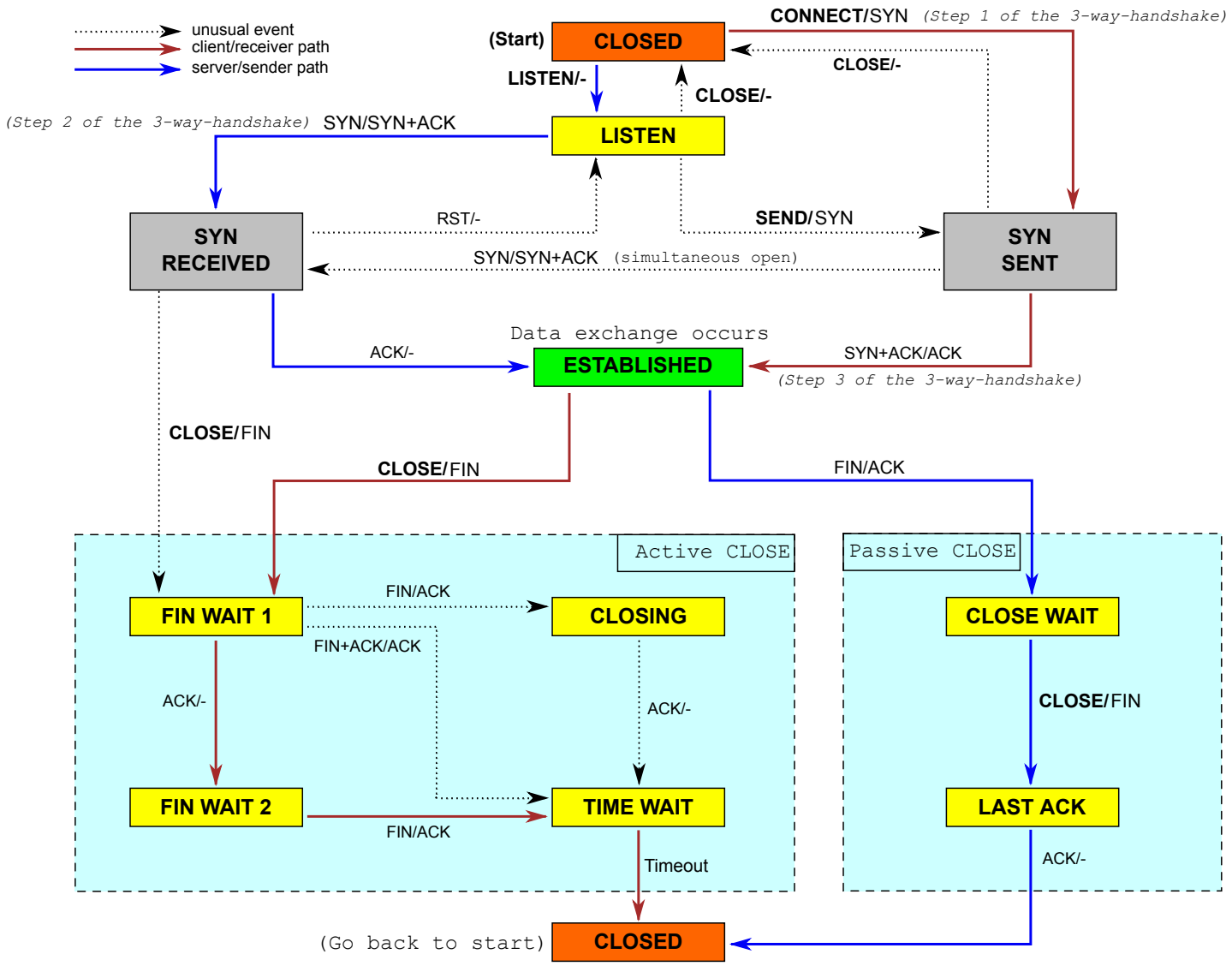
Most important header fields

- Ports: multiplexing
- Sequence number
  - Correspond to *bytes*, not packets!
- Acknowledgment Number
  - Next expected sequence number
- Window: willing to receive
  - Lets receiver limit SWS (even to 0) for flow control
- Checksum: a (really weak) checksum, see RFC

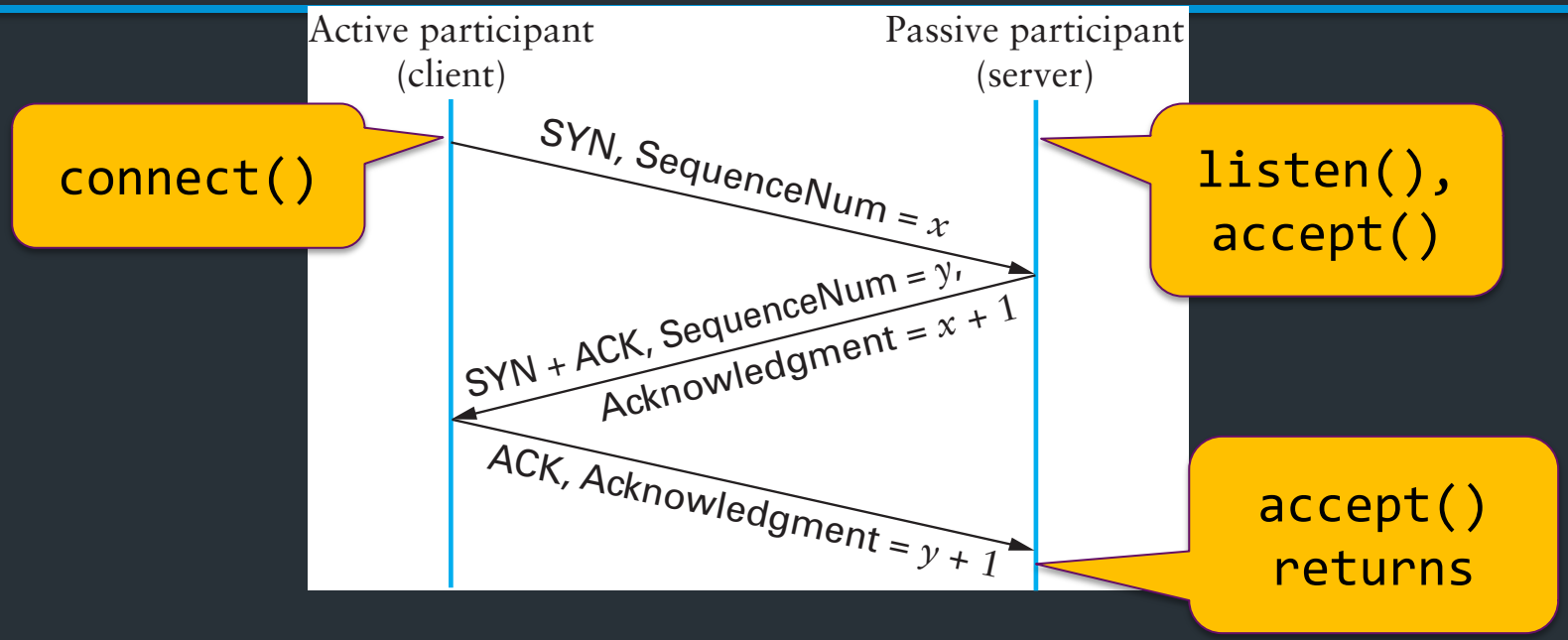
Header Flags

- URG: whether there is urgent data
- ACK: ack no. valid (all but first segment)
- PSH: push data to the application immediately
- RST: reset connection
- SYN: synchronize, establishes connection
- FIN: close connection

TCP State Diagram



Establishing a Connection



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

Sequence numbers

How to pick the initial sequence number?

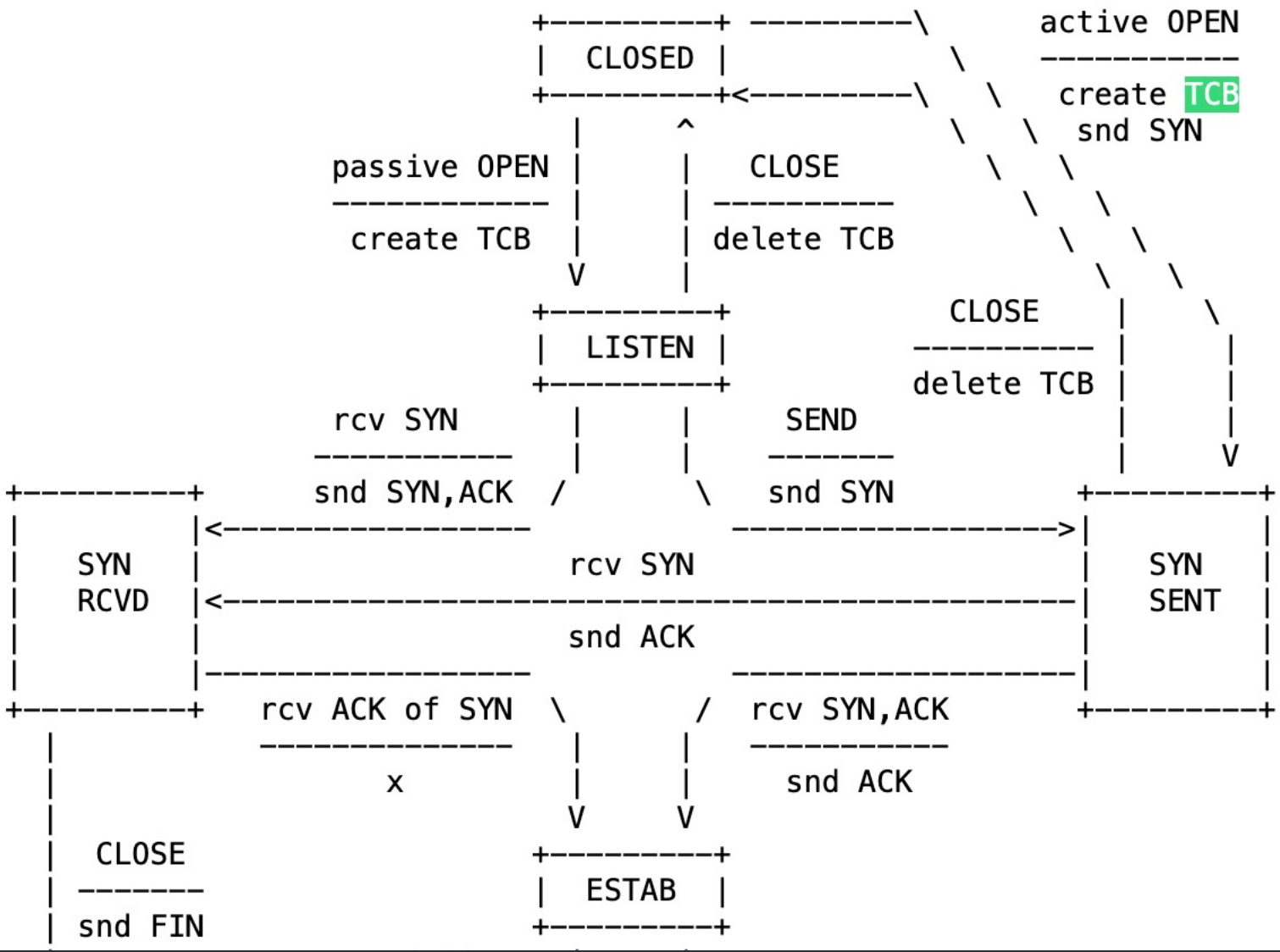
- Protocols based on *relative* sequence numbers based on starting value
- But why not start at 0?

- Instead, pick an arbitrary number

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



RFC 793, Sec 2.4

```
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
```

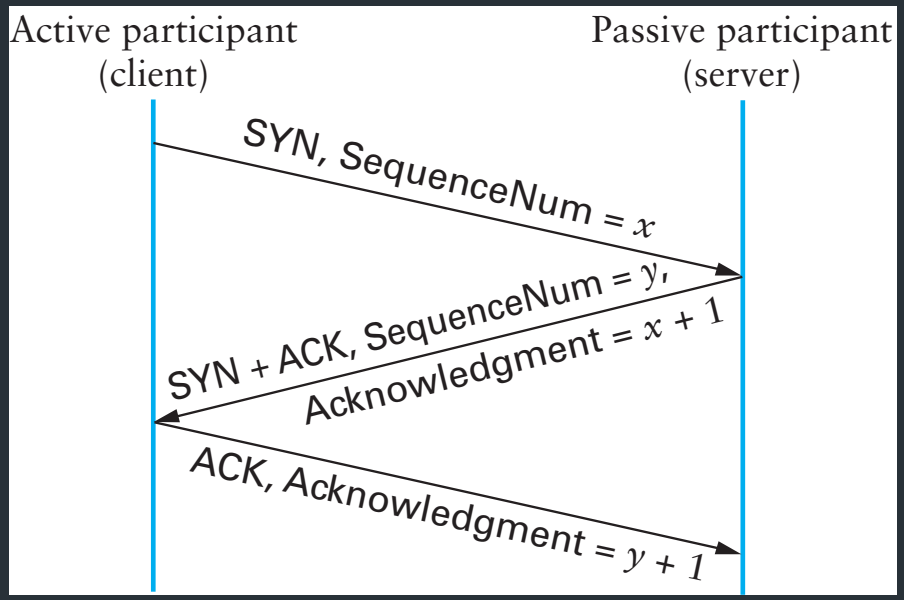
- Each connection has an associated TCB in the kernel
- For each packet kernel maps the 5-tuple (tcp/udp, src IP, src port, dst IP, dst port) to a socket

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?



Sending data

We should not send more data than the receiver can take: *flow control*

- When to send data?
  - Sender can delay sends to get larger segments
- 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

Simplest method: Stop and Wait

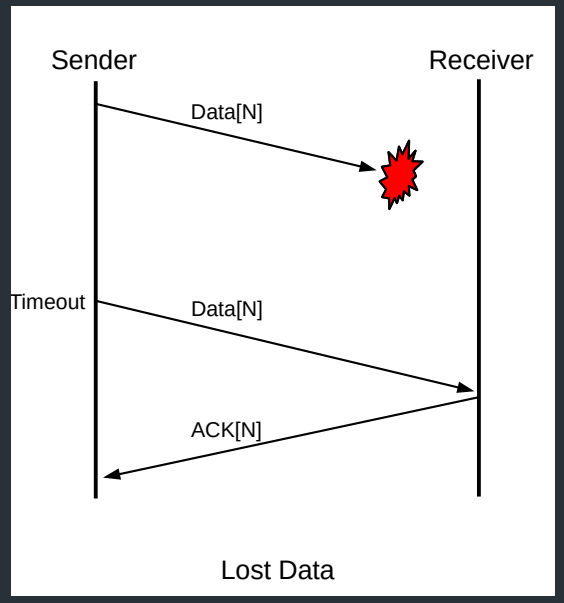
Consider sending one packet at a time

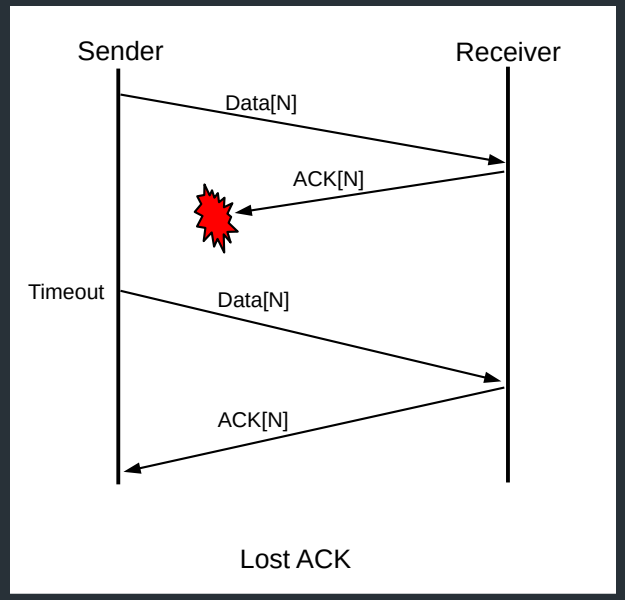
- S: Send packet, wait
- R: Receive packet, send ACK
- S: Receive ACK, send next packet

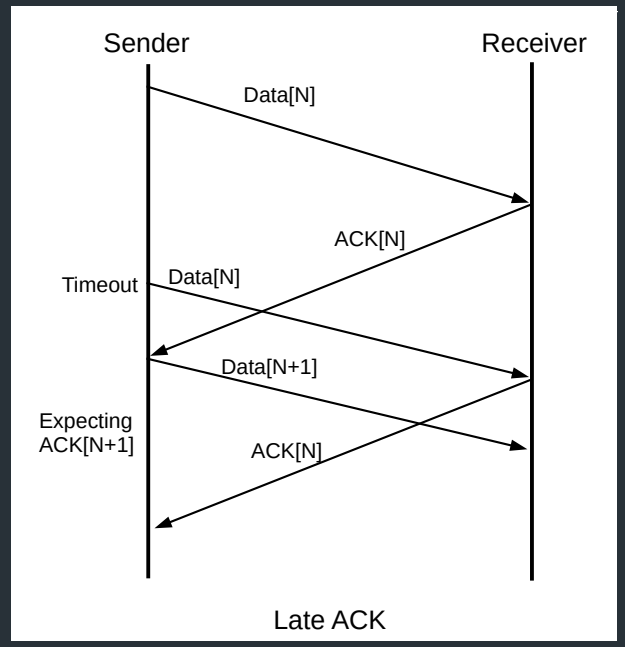
OR

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

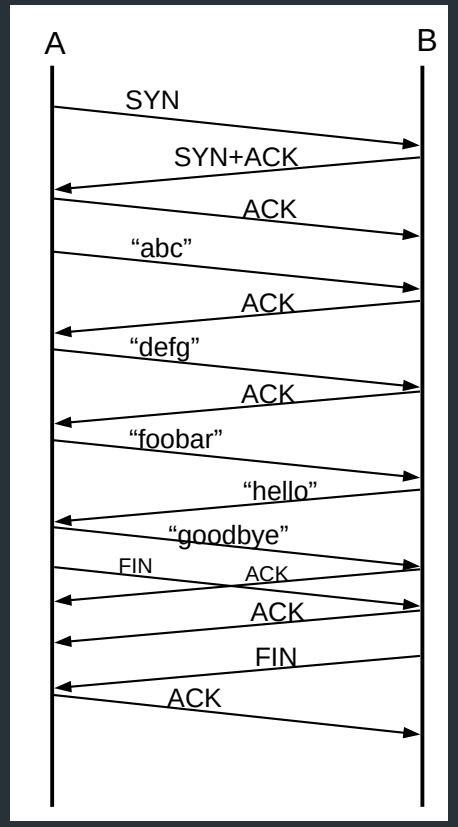
What can go wrong?







Sequence number example

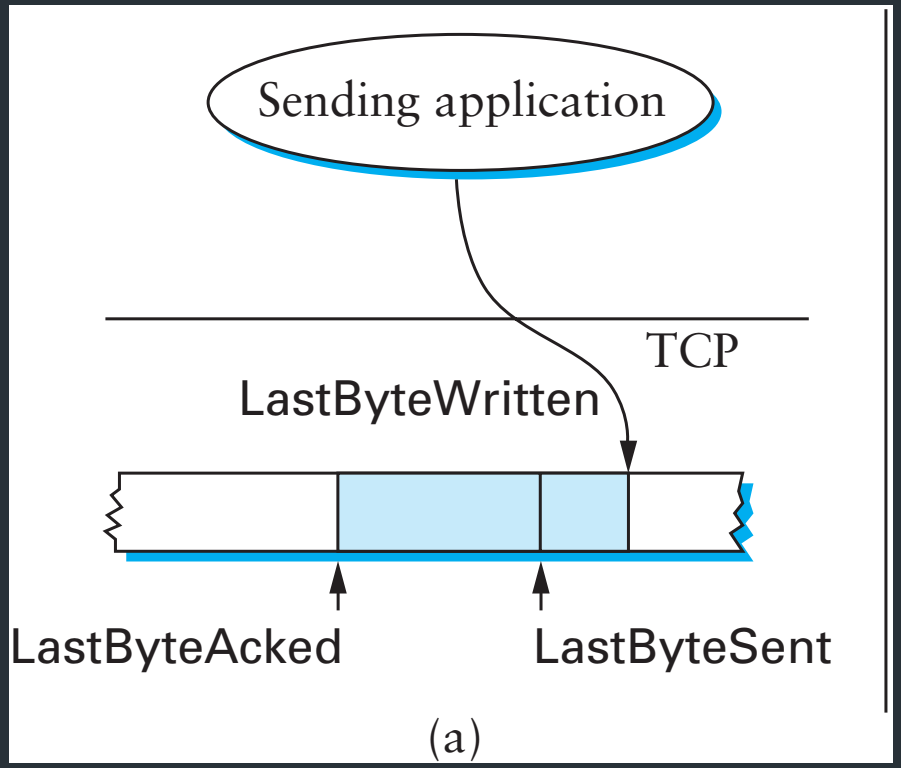


	A sends	B sends
1	SYN, **seq=0**	
2		SYN+ACK, seq=0, **ack=1** (expecting)
3	ACK, **seq=1**, ack=1 (ACK of SYN)	
4	“abc”, **seq=1**, ack=1	
5		ACK, seq=1, **ack=4**
6	“defg”, **seq=4**, ack=1	
7		seq=1, **ack=8**
8	“foobar”, **seq=8**, ack=1	
9		seq=1, **ack=14**, “hello”
10	**seq=14**, ack=6, “goodbye”	
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	

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

Flow Control: Sender

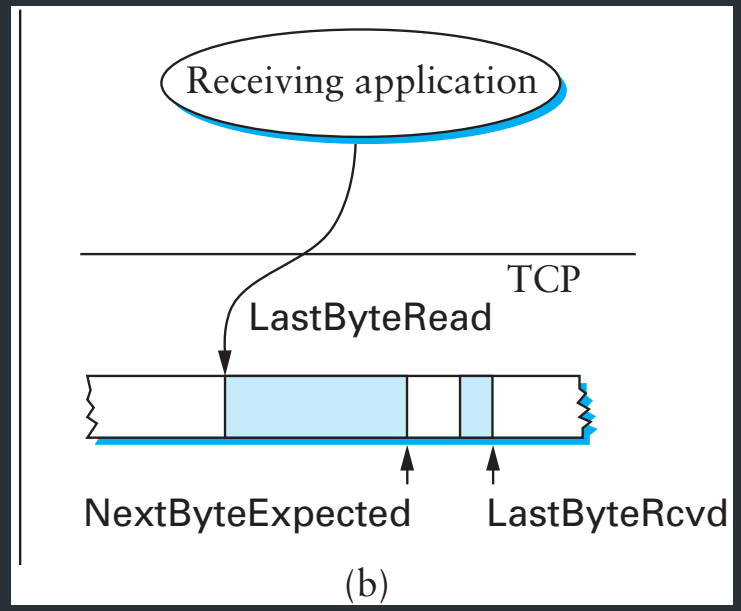


Invariants

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

Useful Sliding Window Terminology:
RFC 793, Sec 3.3

Flow control: receiver



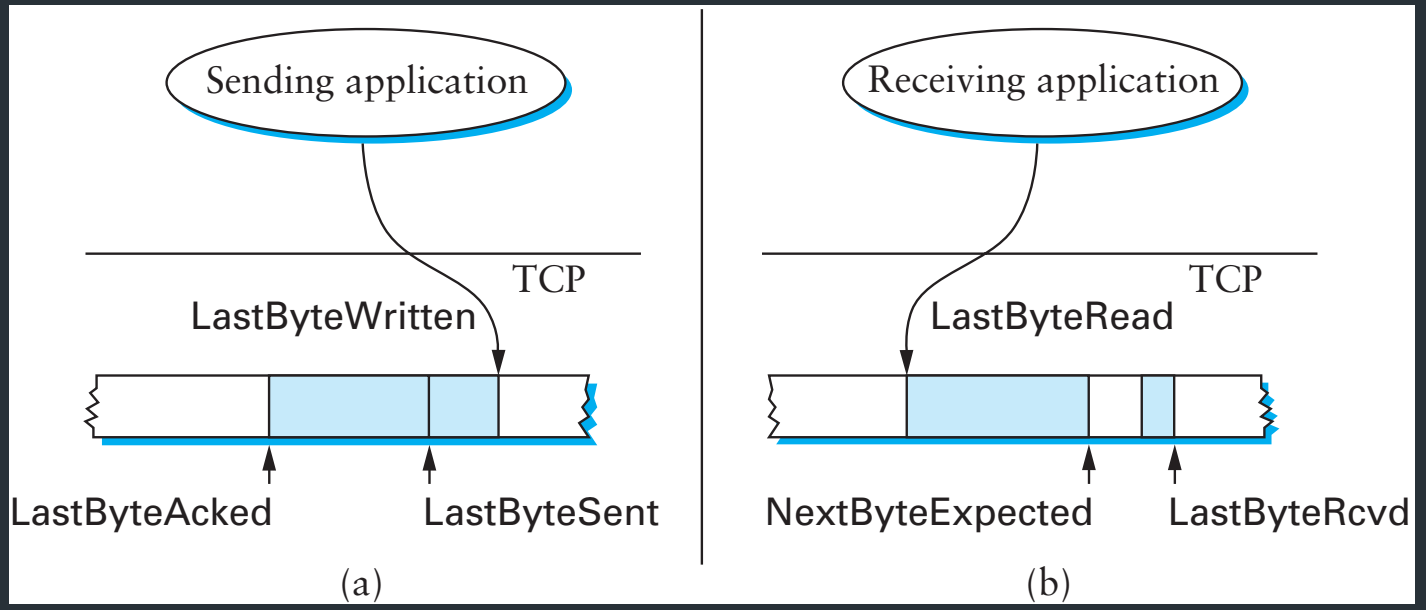
AdvertisedWindow

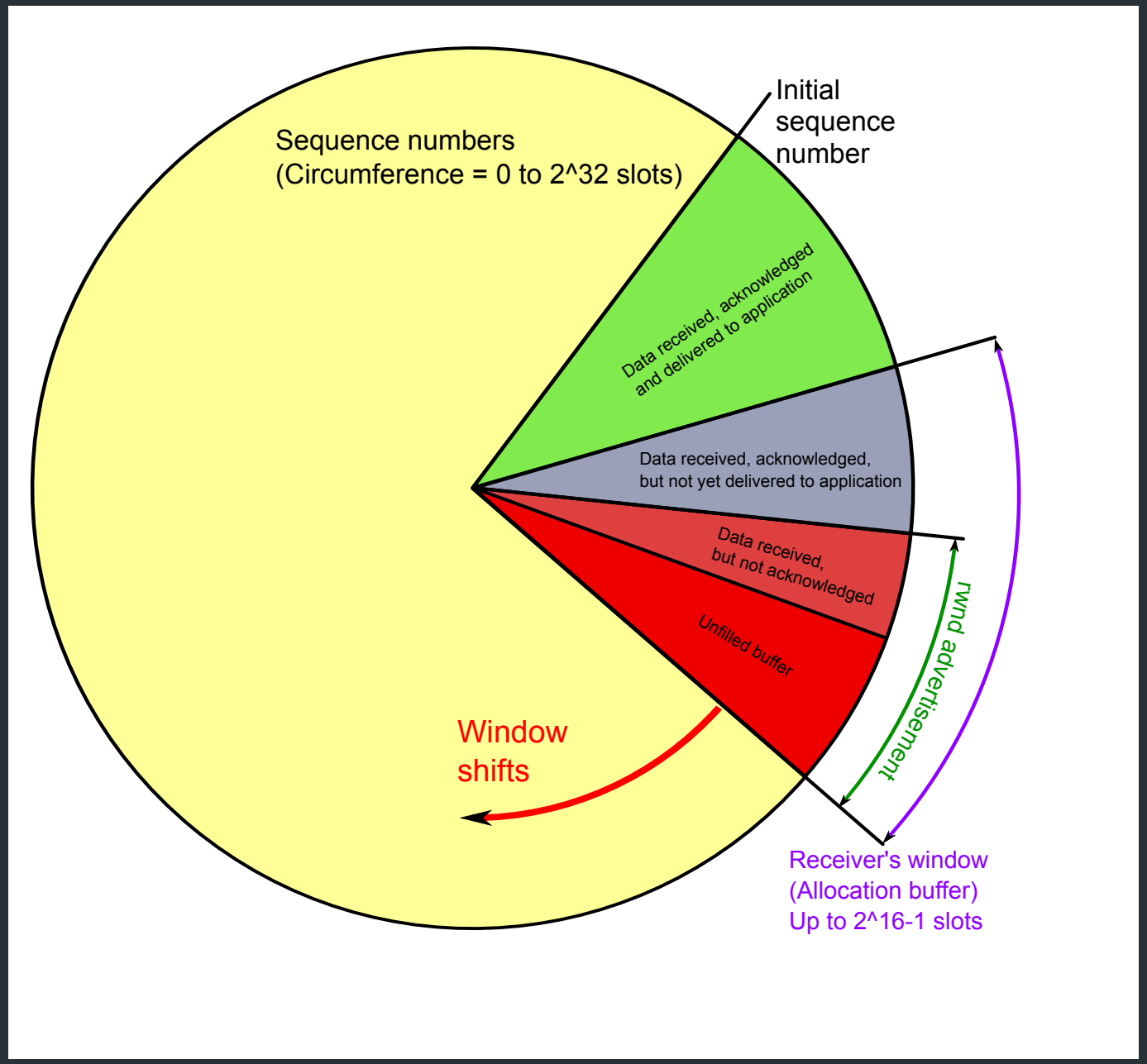
= MaxRcvBuffer – ((NextByteExpected-1) - LastByteRead)

Useful Sliding Window Terminology:
RFC 793, Sec 3.3

Flow Control

- Advertised window can fall to 0
  - How?
  - Sender eventually stops sending, blocks application
- Sender keeps sending 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
- If receiver gets 3 dup ACKs, retransmit

- How to know when to retransmit? Compute based on observed RTT, more on this later

When to Transmit?

- Nagle's algorithm
- Goal: reduce the overhead of small packets

```
if (there is data to send) and (window >= MSS)
  Send a MSS segment
else
  if there is unAcked data in flight
        buffer the new data until ACK arrives
  else
        send all the new data now
```

- Receiver should avoid advertising a window <= MSS after advertising a window of 0

Delayed Acknowledgments

- Goal: Piggy-back ACKs on data
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)

- Warning: can interact badly with Nagle for some applications
  - Nagle waits for ACK until send => Temporary deadlock
  - App can disable Nagle with `TCP_NODELAY`
  - App should also avoid many small writes

Limitations of Flow Control

- Network may be the bottleneck
  - Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides *correctness*
- Need more for performance: congestion control

Second goal

- We should not send more data than the network can take: *congestion control*