CSCI-1680
Transport Layer IV

Data over TCP

Nick DeMarinis

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

Warmup

- Sender wants to send “abcdef”
- Max segment size (MSS) = 1
- Receiver’s window = 4

How many packets are sent

before the first ACK?

Warmup: Sliding Window

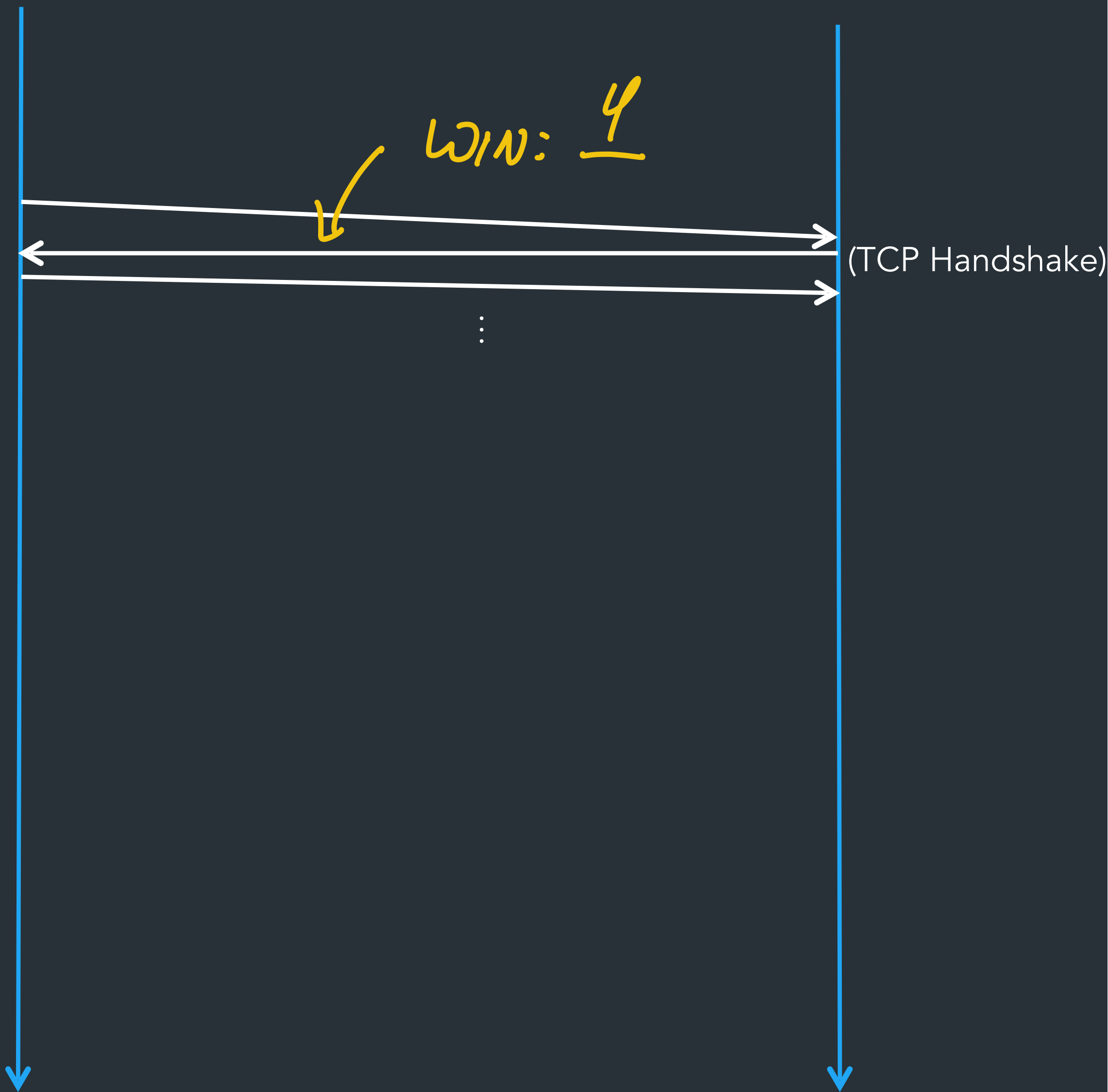
- Max segment size (MSS) = 1
- Receiver's window = 4

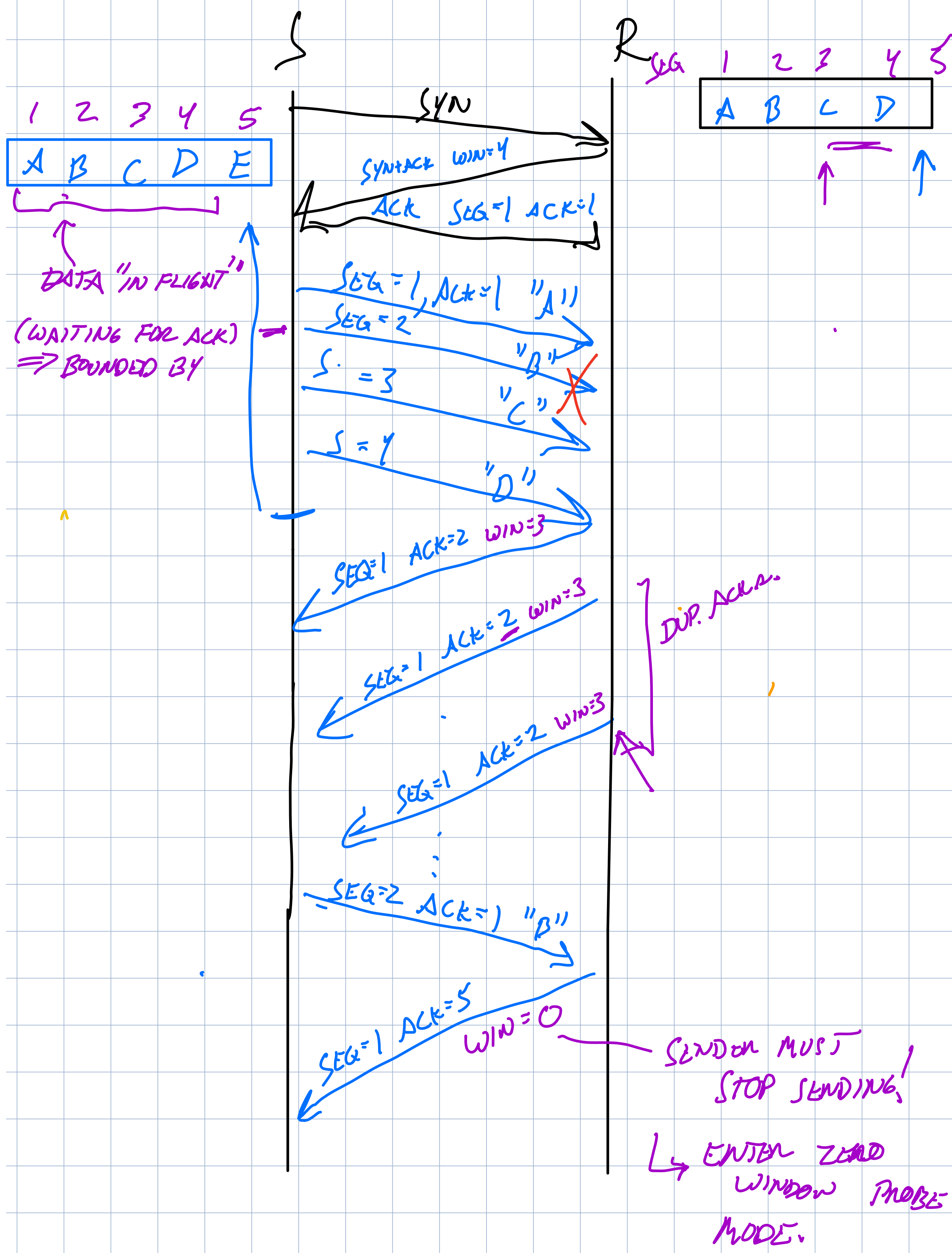
Sender sends "abcdef"

How many packets are sent before the first ACK?

(and what's in them?)

```
conn.Write("abcdef")
```




S
R
SEQ 1 2 3 4 5
A B C D
1 2 3 4 5
A B C D E
DATA "IN FLIGHT"
(WAITING FOR ACK)
⇒ BOUNDED BY
SYN
SYN+ACK WIN=4
ACK SEQ=1 ACK=1
SEQ=1, ACK=1 "A"
SEQ=2 "B"
S=3 "C"
S=4 "D"
SEQ=1 ACK=2 WIN=3
SEQ=1 ACK=2 WIN=3
SEQ=1 ACK=2 WIN=3
DUP. ACKS.
SEQ=2 ACK=1 "B"
SEQ=1 ACK=5 WIN=0
SENDER MUST STOP SENDING!
ENTER ZERO WINDOW PROBE MODE.

What happens when you have a timeout on the sending side?
=> If you have multiple packets in flight, what do you retransmit?

- At minimum retransmit the oldest segment ("b")
- Could also resend the whole window

=> There also exist TCP options to tell the sender specifically what segments you received (but not in minimal spec)

Administrivia

- Sign up for TCP milestone I: this meeting should be this week
- ~~HW4~~ (short!): out today, one problem, practice for TCP

  HW3
- TCP Gearup I: new video + notes—take a look if you haven't
- TCP Gearup II: Thursday (11/2) 5-7pm, CIT368
  - Sliding window, how to test/debug

Grading is in progress… we are prioritizing your milestone meetings so you get real-time feedback

Topics for today

- Connection termination
- Some sending mechanics
- Motivation for congestion control

Connection termination

<u>A 4-step process</u>

- When you have no more data to send, send a FIN
- Both sides close connection separately!

"I AM DONE SENDING"

CLOSING A CONNECTION: 4-WAY "END" PROCESS

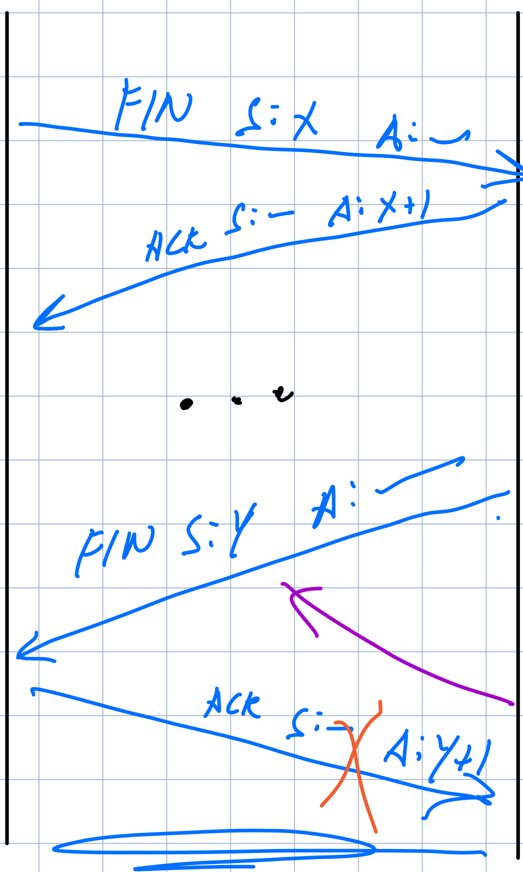
ONE SIDE STARTS => "ACTIVE CLOSE"

OTHER SIDE STARTS => "PASSIVE CLOSE"

CLOSE

FIN-WAIT1

CLOSE-WAIT



At this point, we know other side is done sending
=> But we could still send data

FIN-WAIT2

Can't send anymore from app
TCP stack could still have data to send/ack
Might still receive data

LAST_ACK

TIME_WAIT

MAY BE RETRANSMITTED!

CLOSED
(DELETE TCB)

Even after all of this, the initiating side doesn't know the final ACK was received

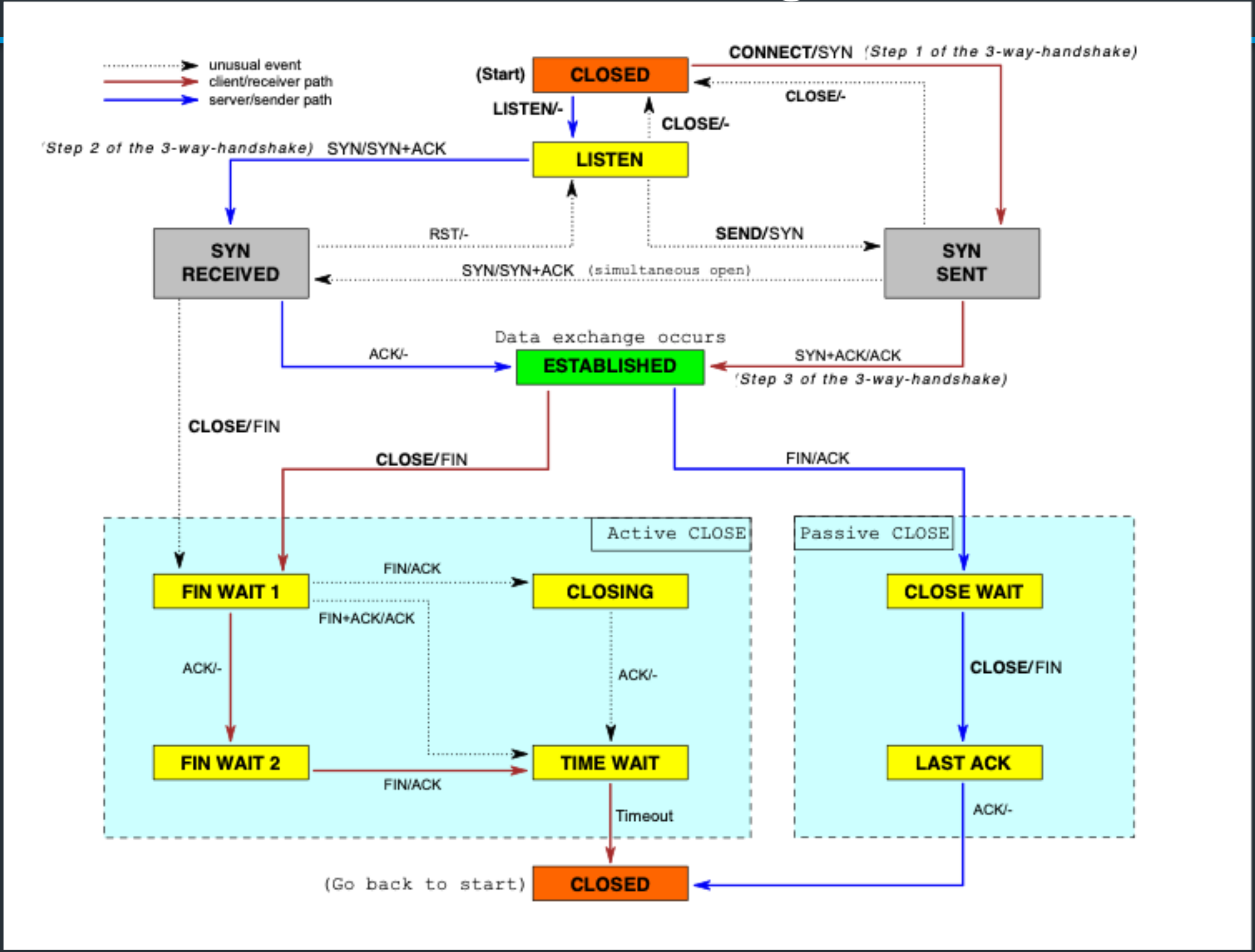
If the ACK was lost, we might need to retransmit, so we can't delete the TCB yet

Solution: need to wait a while before we can delete the TCB (purges TCP state for this connection)
=> How long to wait? 2*MSL (longest time a segment might be delayed) ~2 minutes, configurable

In practice, when we close a connection, it means we're done reading and writing
=> BUT, TCP allows you to close one side at a time (and this process is what lets us do it)

IF YOU ARE A BIG SERVER: PROBLEM:

TCP State Diagram



Connection termination

<u>A 4-step process</u>

- When you have no more data to send, send a FIN
- Both sides close connection separately!

- How to know when last ACK received?
- Initiating side must wait for 2*MSL before deleting TCB

  => MSL = Longest time a segment might be delayed
  (configurable, ~<u>1min</u>)

Connection termination

<u>A 4-step process</u>

- When you have no more data to send, send a FIN
- Both sides close connection separately!

- How to know when last ACK received?
- Initiating side must wait for 2*MSL before deleting TCB

  => MSL = Longest time a segment might be delayed
  (configurable, ~<u>1min</u>)

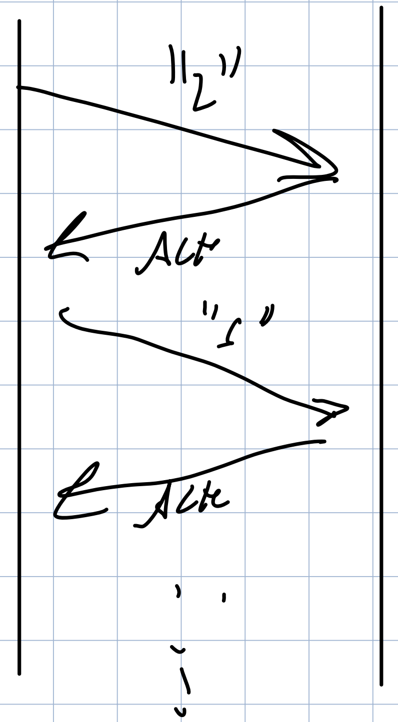
Why do we need to wait this long?

Other mechanics for sending packets
(used in modern TCPs, not required for project)

Example: telnet/SSH

Terminal input <=> TCP connection

EXAMPLE: TELNET (OLD SSH)



"LS -LA"

PROBLEM: LOTS OF NETWORK OVERHEAD FOR SMALL SEGMENTS!

Example: telnet/SSH

Terminal input <=> TCP connection

Problems

=> Tiny packets means high overhead!

=> But also don't want to add latency

=> How to decide when to send? Multiple strategies.

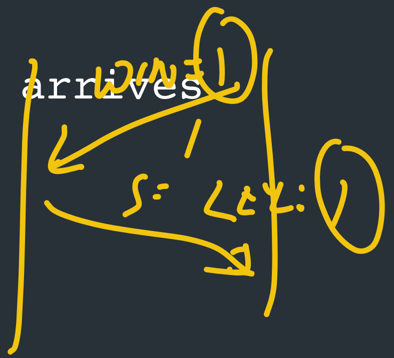
One way: add some more logic to the sender

Nagle's algorithm

TCP_NODELAY

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
```



One way: add some more logic to the sender

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
```

$\Longrightarrow$ AVOIDS TINY SEGMENTS, AT EXPENSE OF LATENCY

One way: add some more logic to the sender

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
```

Recommended in some cases, but waiting to send not always a great idea
=> Configurable on socket creation

Another way: change the receiver

What if receiving app only reads 1 byte at a time?

Silly Window Syndrome (SWS) Avoidance: when window is zero, wait until 1MSS of receive buffer space is available before advertising nonzero window

Yet another way: receiver could delay sending ACK for short time (400ms), in case it has data to send

=> All data segments are ACKs, so why send packet again?

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)

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)

N6

- 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

Congestion control: the start

The story so far

Flow control provides reliable, in-order delivery

Goal: send as much data as receiver can handle

- Receiver's <u>advertised window</u>: sent with every ACK
- Sliding window: increase throughput by having multiple packets in flight

The story so far

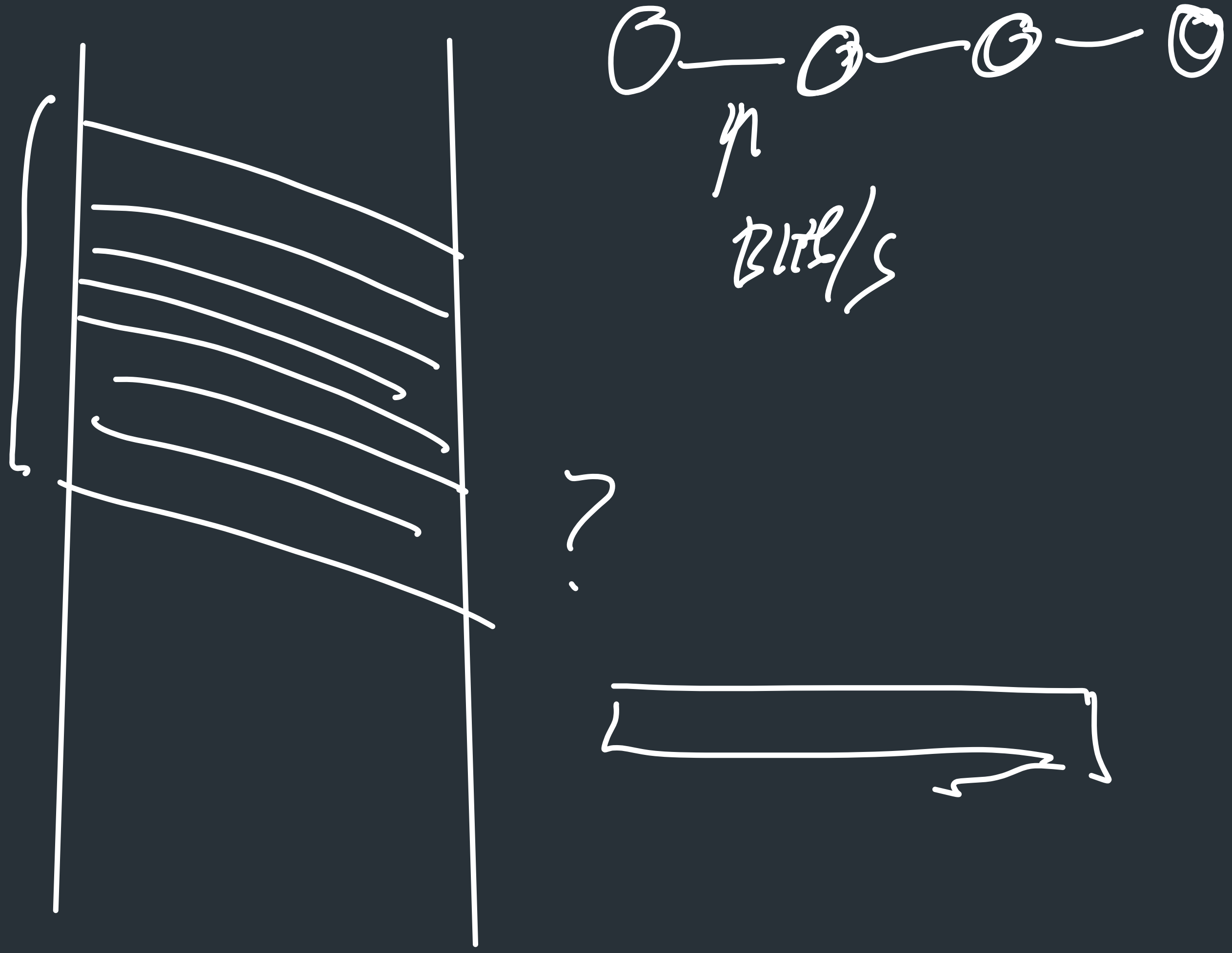
Flow control provides reliable, in-order delivery

Goal: send as much data as receiver can handle

- Receiver's <u>advertised window</u>: sent with every ACK
- Sliding window: increase throughput by having multiple packets in flight

<u>Problems?</u>

What would happen with our current sliding window implementation?



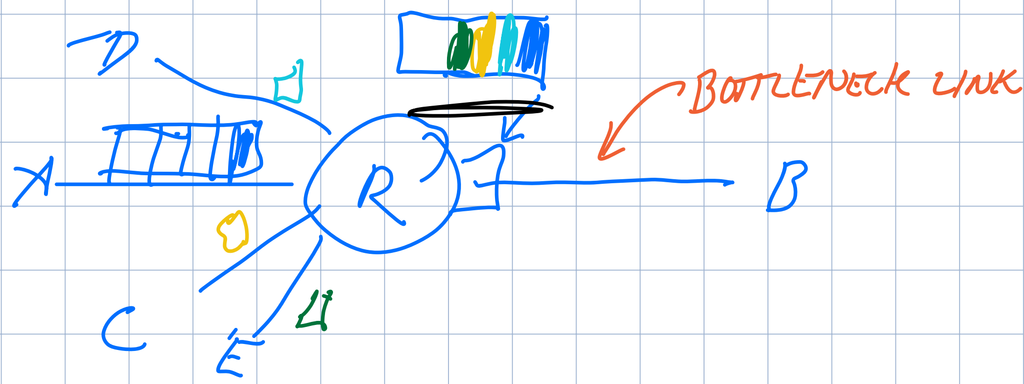
What else do we need?

- Flow control provides correctness: reliable, in order delivery
- Need more for performance
  - What if the network is the bottleneck?

How do we know when <u>the network</u> is overloaded?

CONGESTION: WHAT CAN GO WRONG?

- INCREASED LATENCY
- DROPPED PACKETS

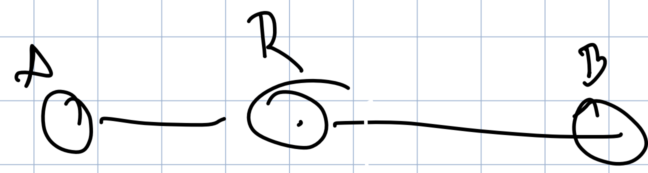


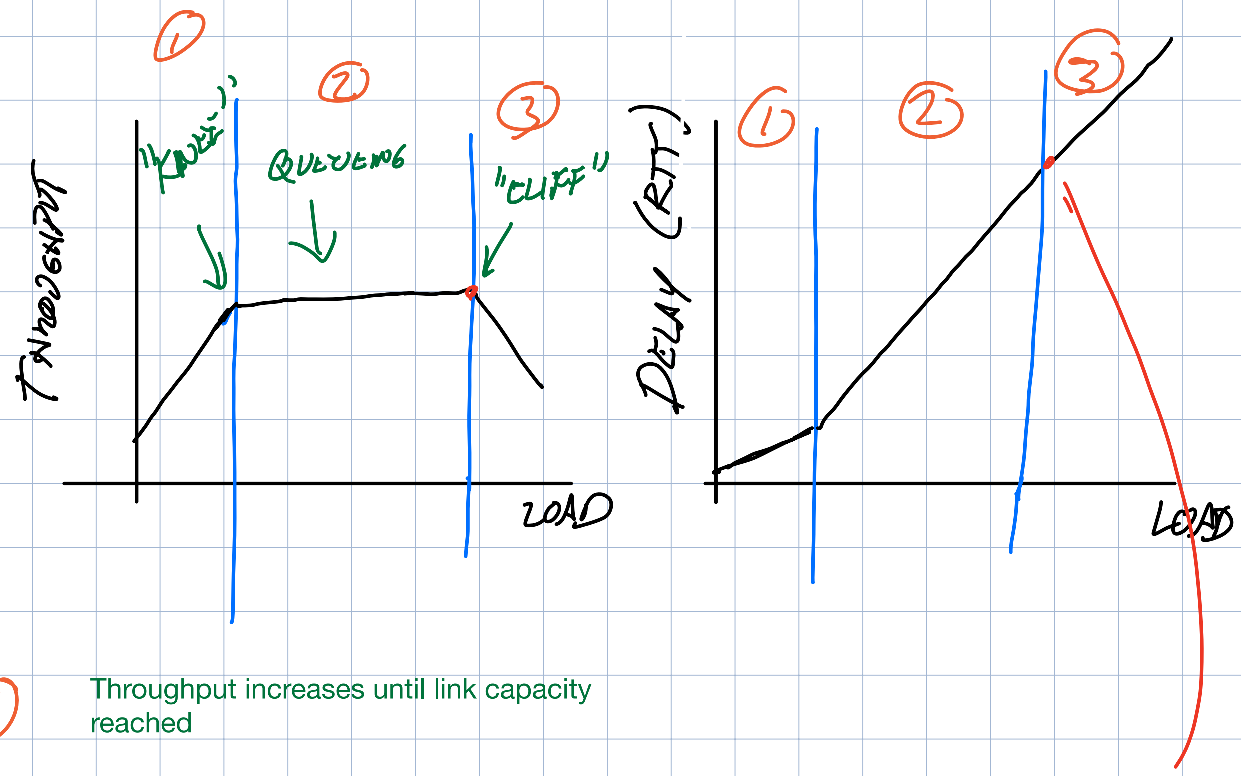
Inside every network device is a queue of packets waiting to be send out (usually as part of the destination port for the packet)

Buffer can fill up if
- Lots of senders trying to use the same link
- Output has a lower bandwidth than the input

If you fill the buffer, newest packets get dropped

THINKING ABOUT CONGESTION CONTROL





FULL BUFFERS
=> PACKET LOSS

1. Throughput increases until link capacity reached

2. Once the bottleneck starts buffering packets, throughput stops increasing, RTT increases (more queuing => more time spent in the buffer)

3. After queue fills, packets are dropped/retransmitted => RTT increases, useful throughput decreases

The problem

- https://witestlab.poly.edu/respond/sites/genitutorial/files/tcp-aimd.ogv

Congestion control

We must not send more data than the network can handle

What happens if we do?

A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer congestion collapses
  - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (TCP Tahoe)
- 1990: Fast transmit and fast recovery added

  (TCP Reno)

* Van Jacobson and Michael Karels. Congestion avoidance and control. SIGCOMM ’88

Congestion Collapse

Nagle, rfc896, 1984

- Mid 1980's: Problem with the protocol implementations, not the protocol!
- What was happening?
- If close to capacity, and, e.g., a large flow arrives suddenly…
  - RTT estimates become too short
  - Lots of retransmissions → increase in queue size
  - Eventually many drops happen (full queues)
  - Fraction of useful packets (not copies) decreases

Congestion control: the main idea

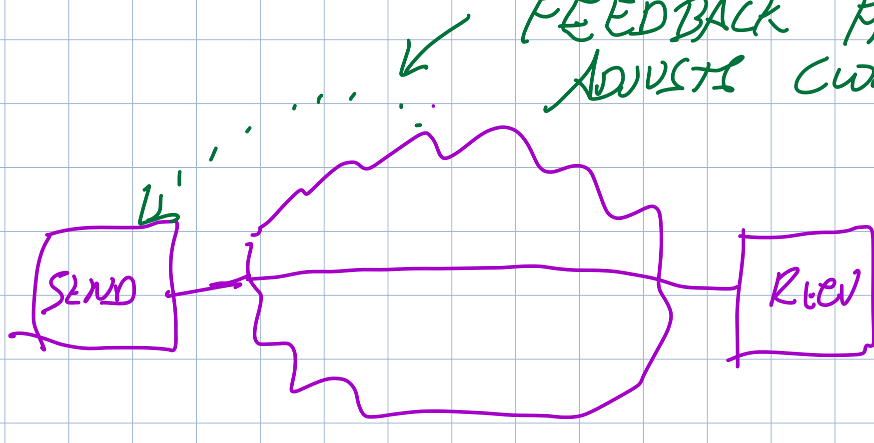
- Determine the initial capacity of the network
- Adjust the sending rate as the capacity changes over time
  (continually monitor something that gives indication about the network

How to do this? A modern TCP has two “windows”
- Advertised window from receiver (WIN in the TCP header0
- Congestion window (cwnd)

Amount of data you can send = min(advertised window, cwnd)

Lots of different ways that this control process happens (ways to signal the network is congested):
- Loss-based congestion control (TCP Tahoe, …) => packet loss === congestion
- Monitor packet delay
- (if network cooperates) routers can mark packets (ECN)

FEEDBACK FROM NETWORK
ADJUSTS CWND.



TCP Congestion Control

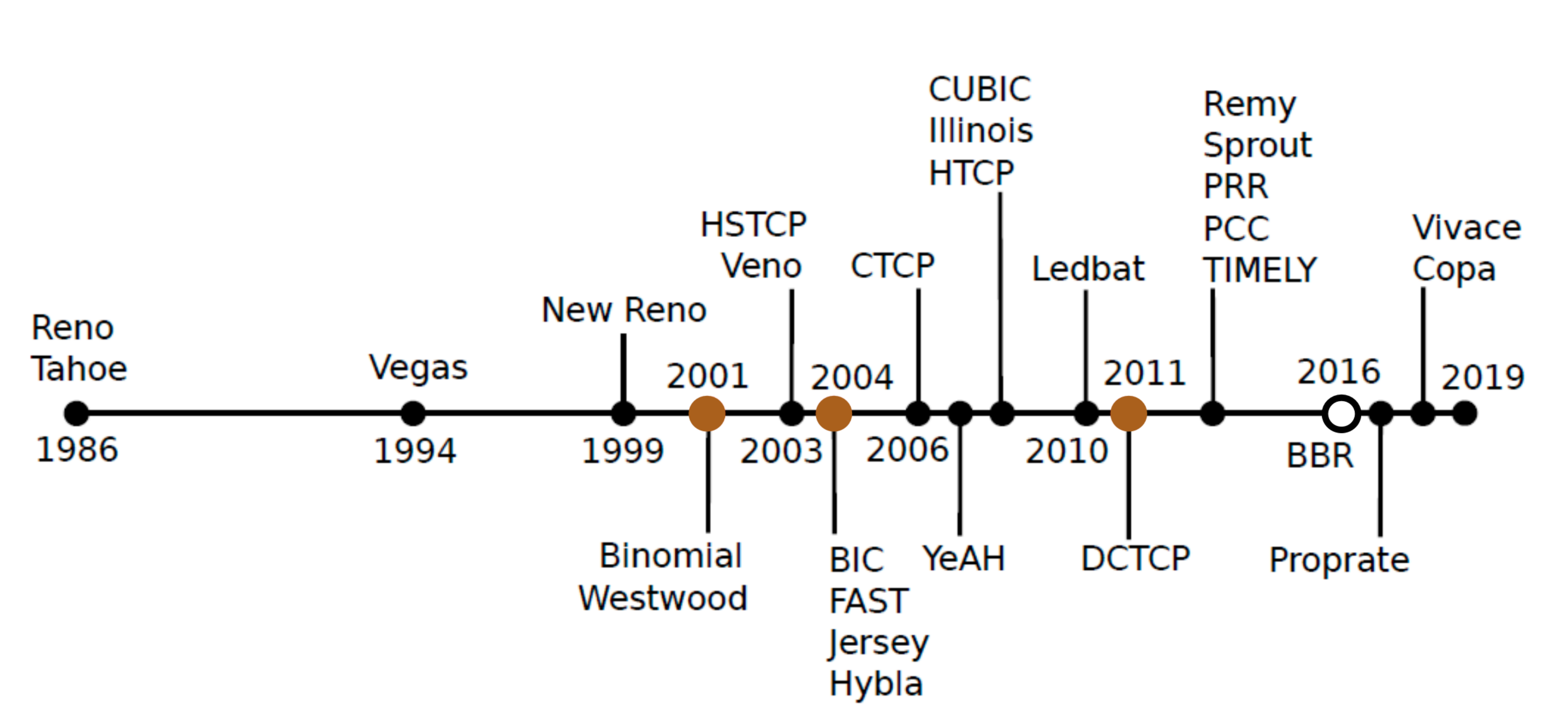
- 3 Key Challenges
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows

- Idea
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs pace transmission (self-clocking)

Congestion control has a long history

- Active research area for ~40 years
- I am <u>nowhere close</u> to being an expert
- My hope is to get you to understand the problems involved

Timeline of (some!) congestion control implementations



[“The great Internet congestion control census” (2019)]