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

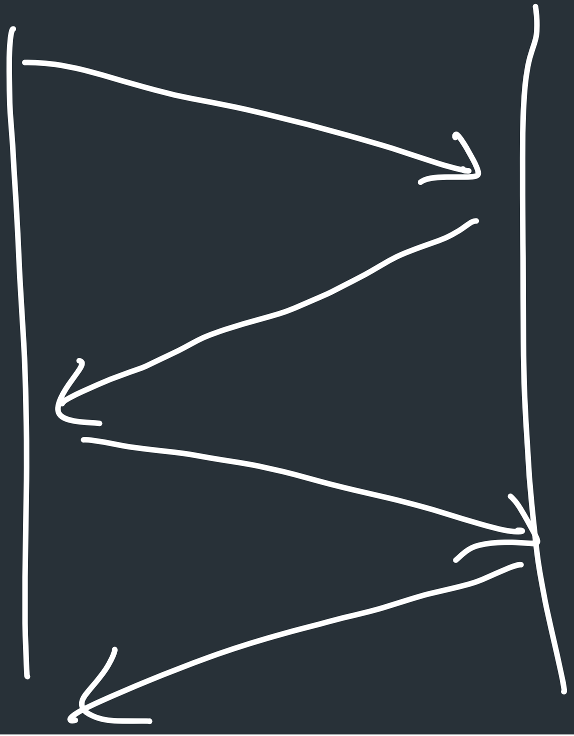
- TCP Gearup I TONIGHT (10/26) 5-7pm, CIT368 (+Zoom, +REC)
  - How the project works, how to think about sockets
  - Stuff you need for milestone 1
- TCP milestone 1: Schedule on/before Thursday, November 2
  - Email later today for signups
- HW2: Due Mon, Oct 30
  - Last problem helpful for milestone 1

Topics for today

- Flow control: Sliding window
- Computing RTO
- Connection termination

The story so far

Stop and Wait: Simplest TCP sender/receiver



Key features

- SEQ/ACK numbers denote where sender/receiver are in data stream
- Only one segment is "in flight" at a time

Warmup: Stop and Wait

What are the values for the SEQ and ACK fields?

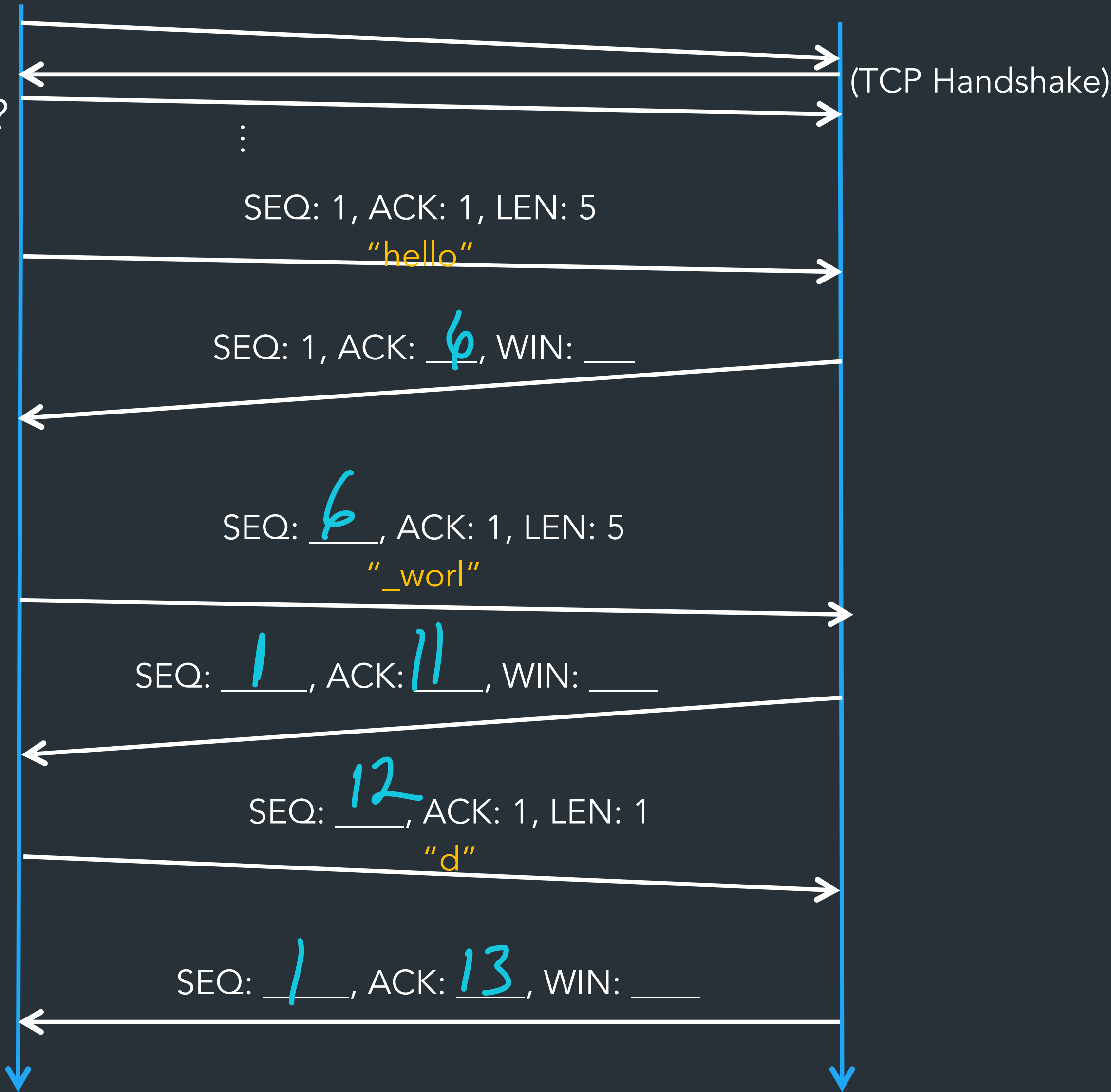
```
conn.Write(“hello_world”)
```

SEQ: where segment is in data stream

ACK: next byte the sender expects to get

eg. ACK x

“I have up to (x - 1), send me x next”



(TCP Handshake)

⋮

SEQ: 1, ACK: 1, LEN: 5
“hello”

SEQ: 1, ACK: 6, WIN: ___

SEQ: 6, ACK: 1, LEN: 5
“_worl”

SEQ: 1, ACK: 11, WIN: ____

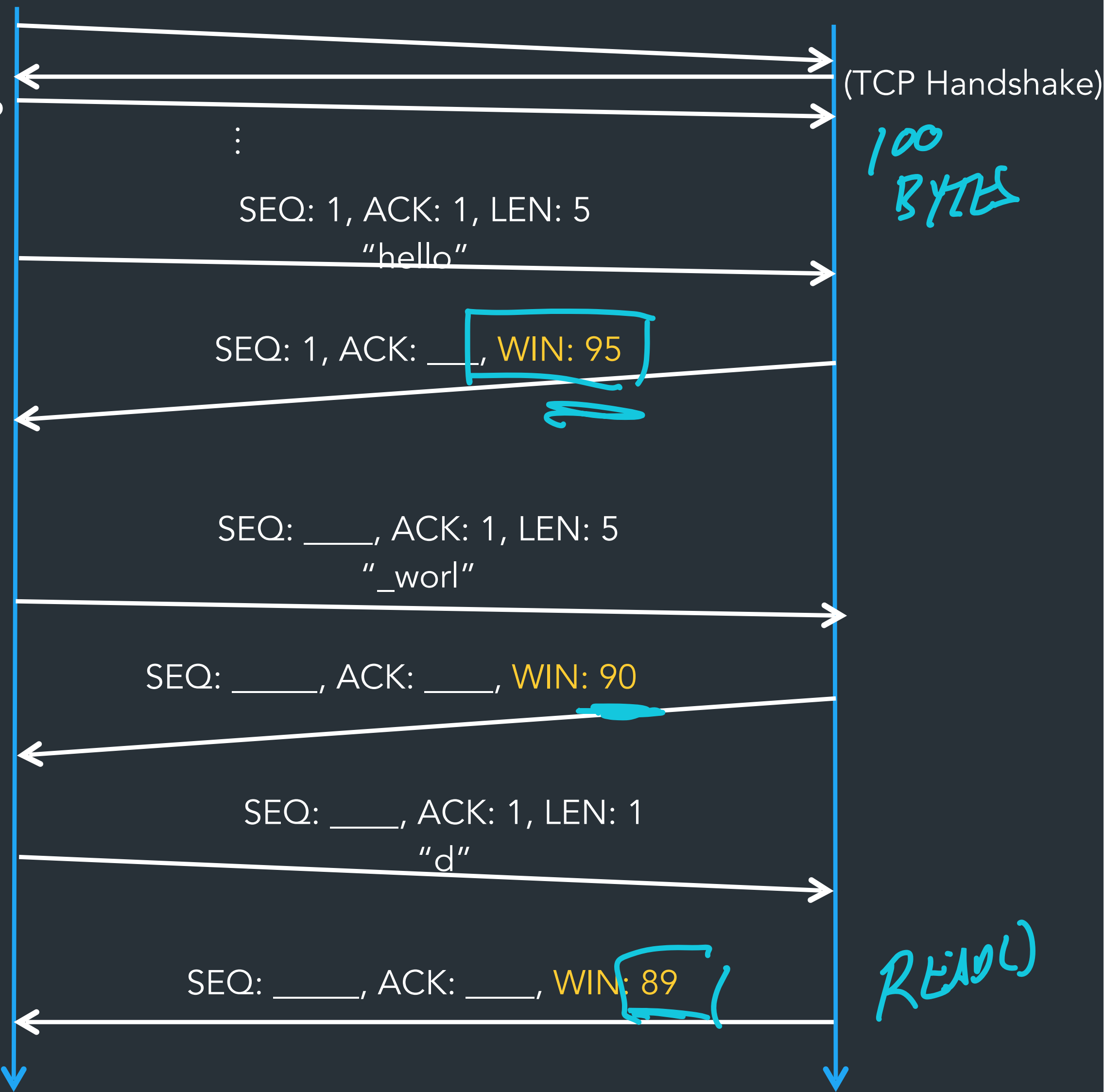
SEQ: 12, ACK: 1, LEN: 1
“d”

SEQ: 1, ACK: 13, WIN: ____

Warmup: Stop and Wait

What are the values for the SEQ and ACK fields?

conn.Write("hello_world")



(TCP Handshake)

100 BYTES

⋮

SEQ: 1, ACK: 1, LEN: 5
"hello"

SEQ: 1, ACK: ___, WIN: 95

SEQ: ____, ACK: 1, LEN: 5
"_worl"

SEQ: _____, ACK: ____, WIN: 90

SEQ: ____, ACK: 1, LEN: 1
"d"

SEQ: _____, ACK: ____, WIN: 89

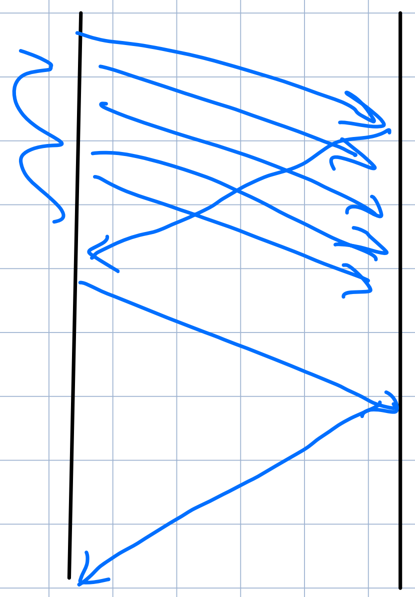
READ()

Key features

- SEQ: Position of this segment in the data stream
- ACK: Next sequence number the receiver expects to receive (ACK N == "I have up to (N – 1)")

Advertised window: how much space the receiver has left in its receive buffer
=> Window (WIN) field in TCP header

IN EVERY PACKET.



STOP + WAIT:
LOTS OF UNUSED BW

WANT: MORE DATA "IN FLIGHT"
AT A TIME.

⇒ SLIDING WINDOW

TCP and buffering

Recall: TCP stack responsibilities

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

TCP stack needs to buffer data for both parts

- Sender: data waiting to be sent, not yet ACK'd
- Receiver: data not yet read by app, out-of-order segments

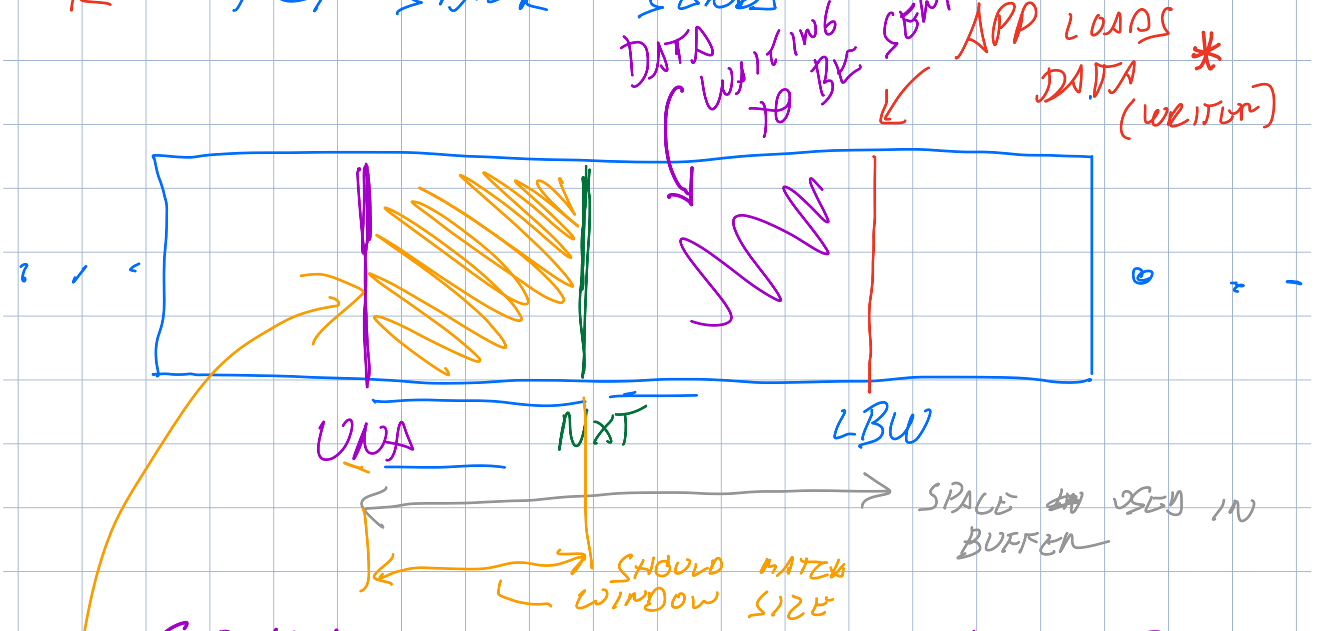
Remember: in reality, both sides can send and receive!
=> All sockets have both a send and receive buffer

RFC 9293: Sec 3.1, 3.3.1, 3.4

SLIDING WINDOW: SENDING SIDE
(SND)

W - APP LOADS DATA INTO BUFFER (CONN. WRITE)

R - TCP STACK SENDS



SND. UNA - OLDEST UNACKED SEGMENT

SND. NXT - NEXT SEQUENCE NUMBER TO BE SENT
- NEXT BYTE TO BE SENT

LBW - LAST BYTE WRITTEN

<u>"BYTES IN FLIGHT"</u> - DATA THAT HAS BEEN SENT OUT, BUT NOT ACK'D YET.

* NOTE: IF BUFFER BECOMES FULL, WRITE FROM APP SHOULD BLOCK UNTIL DATA AVAILABLE.

SENDER OPERATION

- SEND UP TO WINDOW (ADVANCES NXT)

- BYTES IN FLIGHT < ADVERTISED WINDOW

- KEEP TRACK OF "IN FLIGHT" SEGMENTS, RETRANSMIT ON TIMEOUT ("RETRANSMIT QUEUE")

- ON ACK FOR SOME SEGMENT S,

  - ACK MUST FALL WITHIN WINDOW

$$UNA < S.ACK \leq NXT$$

↘ WITHIN "BYTES IN FLIGHT"

  - IF NOT, ACK IS INVALID/OLD => DROP.

OTHERWISE

  - $UNA \mathrel{+}= \left(\text{HOW MUCH DATA WAS ACK'D}\right)$

  - IF ACK FULLY COVERED A SEGMENT, REMOVE FROM RETRANSMIT QUEUE

EXAMPLE: 10 1-BYTE SEGMENTS

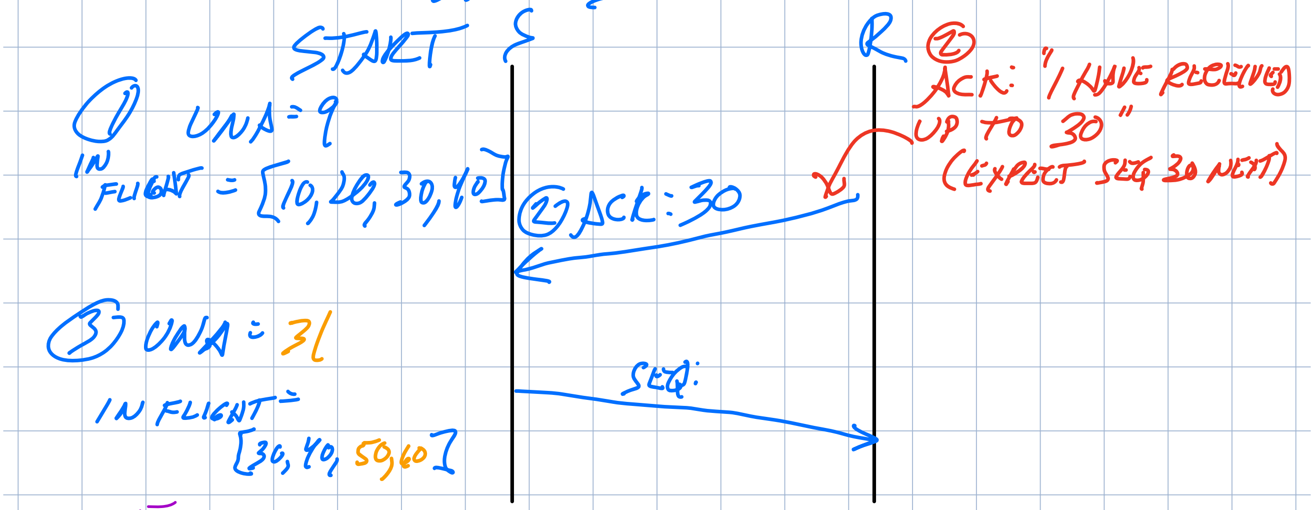
UNA = 10   W = 4   W = [10, 11, 12, 13]

IN FLIGHT

IF YOU GET ACK = 12, W = [12, 13, 14, 15]

CAN SEND.

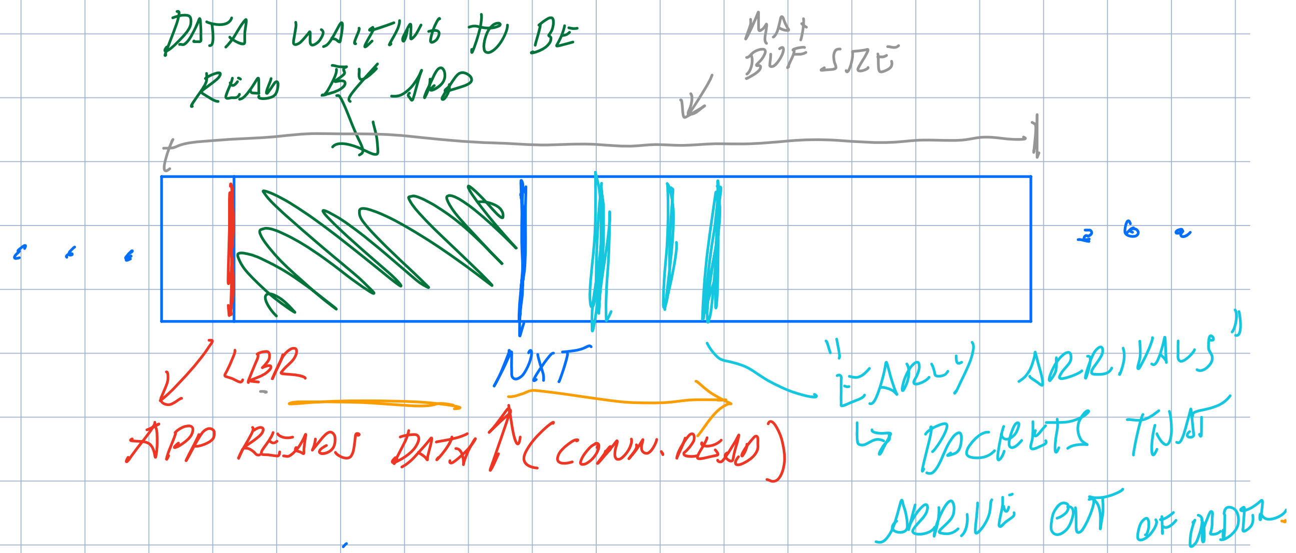
Ex. 10 BYTE SEGMENTS



FOR EACH SEGMENT

- KEEP TIMESTAMP OF LAST SENT TIME
- RETRANSMIT IF IT EXPIRES.

RECEIVING SIDE (RCV)


DATA WAITING TO BE READ BY APP
MAX BUF SIZE
LBR
NXT
APP READS DATA (CONN.READ)
"EARLY ARRIVALS"
↳ PACKETS THAT ARRIVE OUT OF ORDER.

RCV.NXT = NEXT BYTE EXPECT TO RECEIVE

= NEXT SEQ NUM EXPECT TO RCV

LBR - LAST BYTE READ BY APP

ADVERTISED WINDOW - AMOUNT OF SPACE REMAINING IN BUFFER (CAN BE 0)

= MAXBUF - ((NXT-1) - LBR)

↳ THIS IS WHAT IS SENT IN WINDOW FIELD

PROBLEM: OUT OF ORDER PACKETS

SOLUTION: "EARLY ARRIVAL QUEUE"

- TRACKS SEGMENTS ARRIVING AFTER NXT (BUT WIN BOUND)

WHEN RECEIVER GETS A SEGMENT S MUST CHECK IT FITS IN WINDOW:

$S.SEQ < RCV.NXT$ AND $S.SEQ < RCV.NXT + RCV+WND$

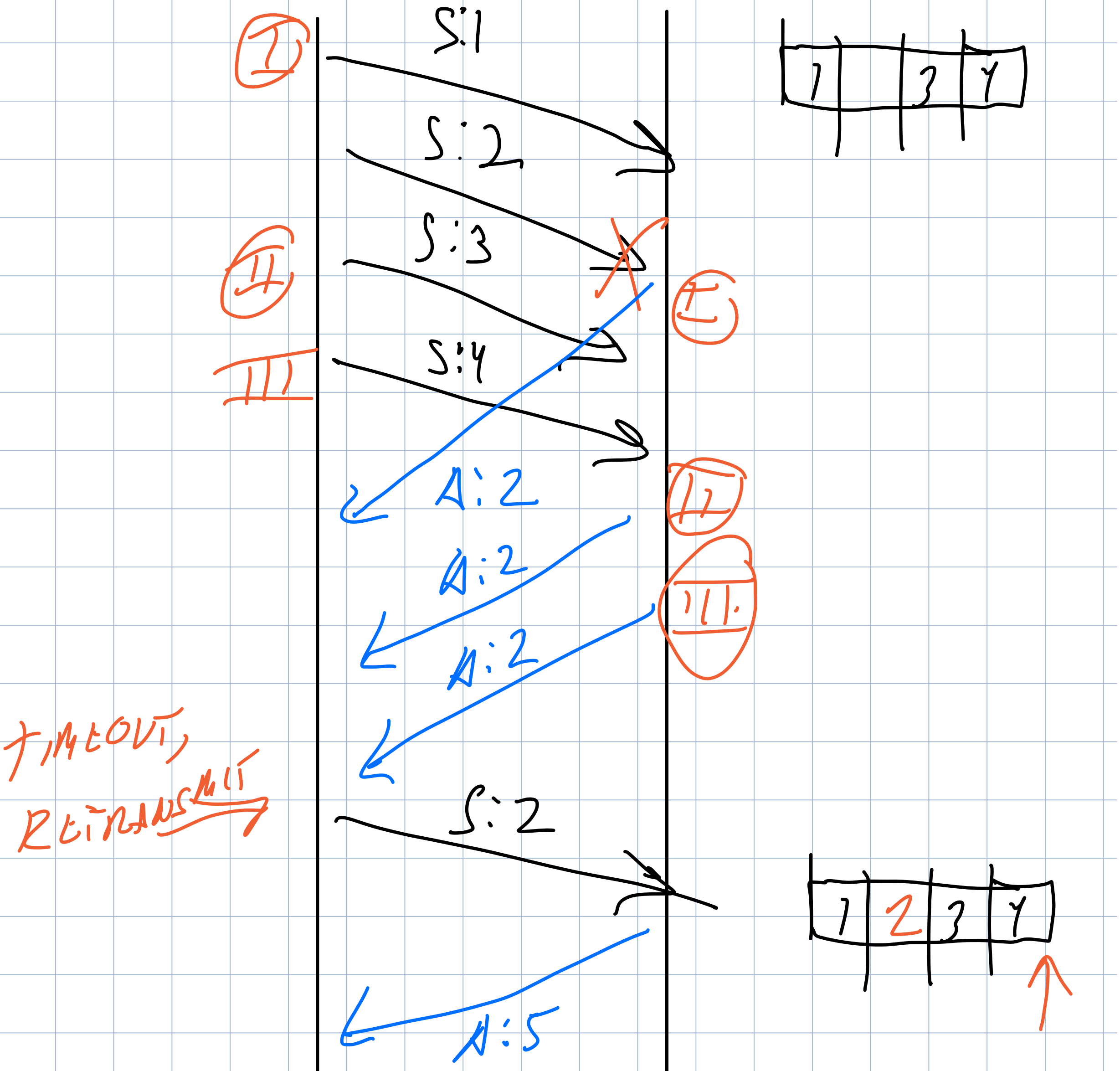
OR

(SIMILAR CHECK FOR END OF WINDOW)

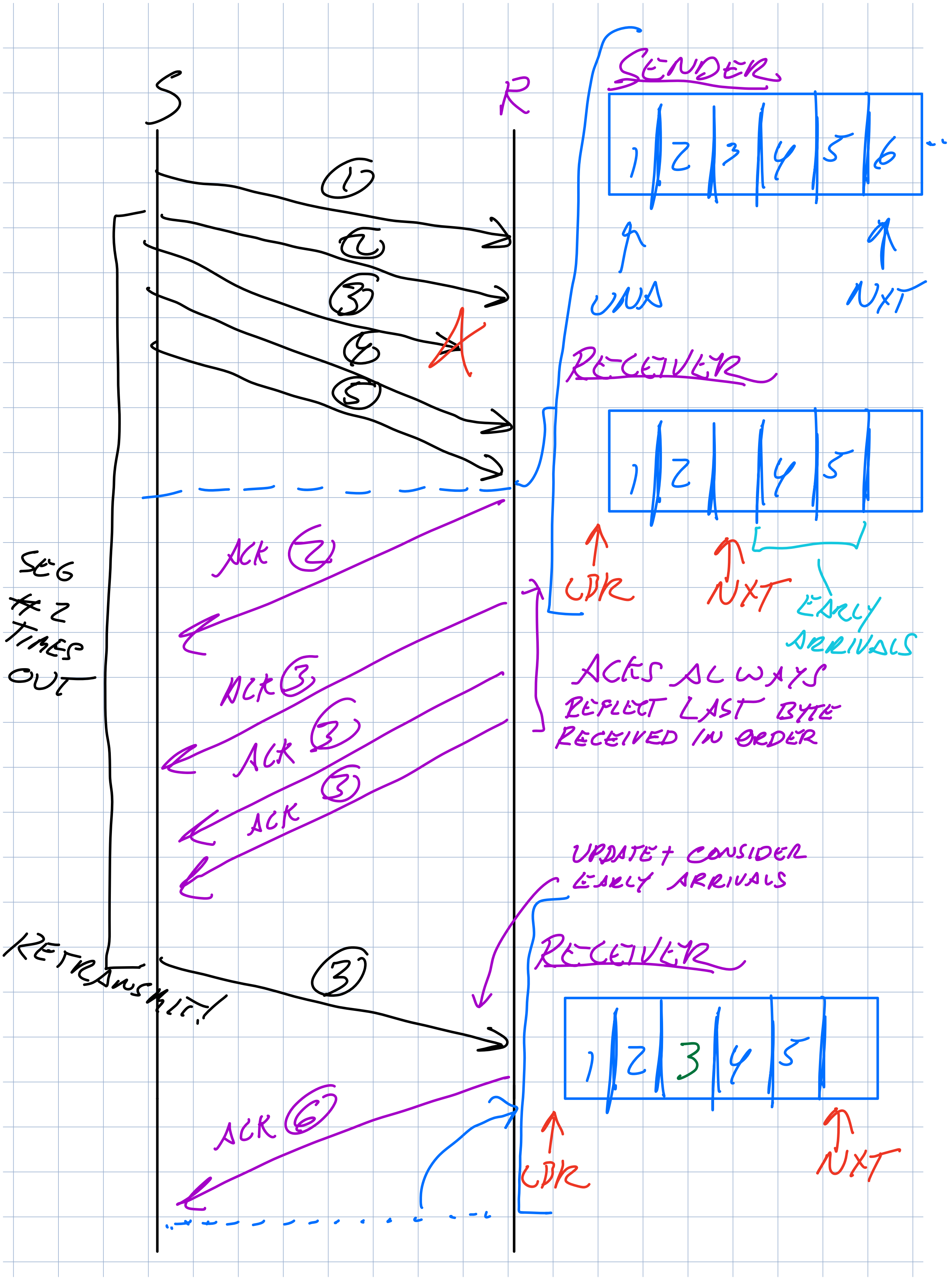
(RFC 9293, Sec 3.4)

- ADD AT POSITION S.SEQ
  - NXT += SEGMENT SIZE
  - CHECK EARLY ARRIVAL QUEUE - MOVE UP TO NEXT CONTIGUOUS PART

Sender example from class (cleaner version on next page)



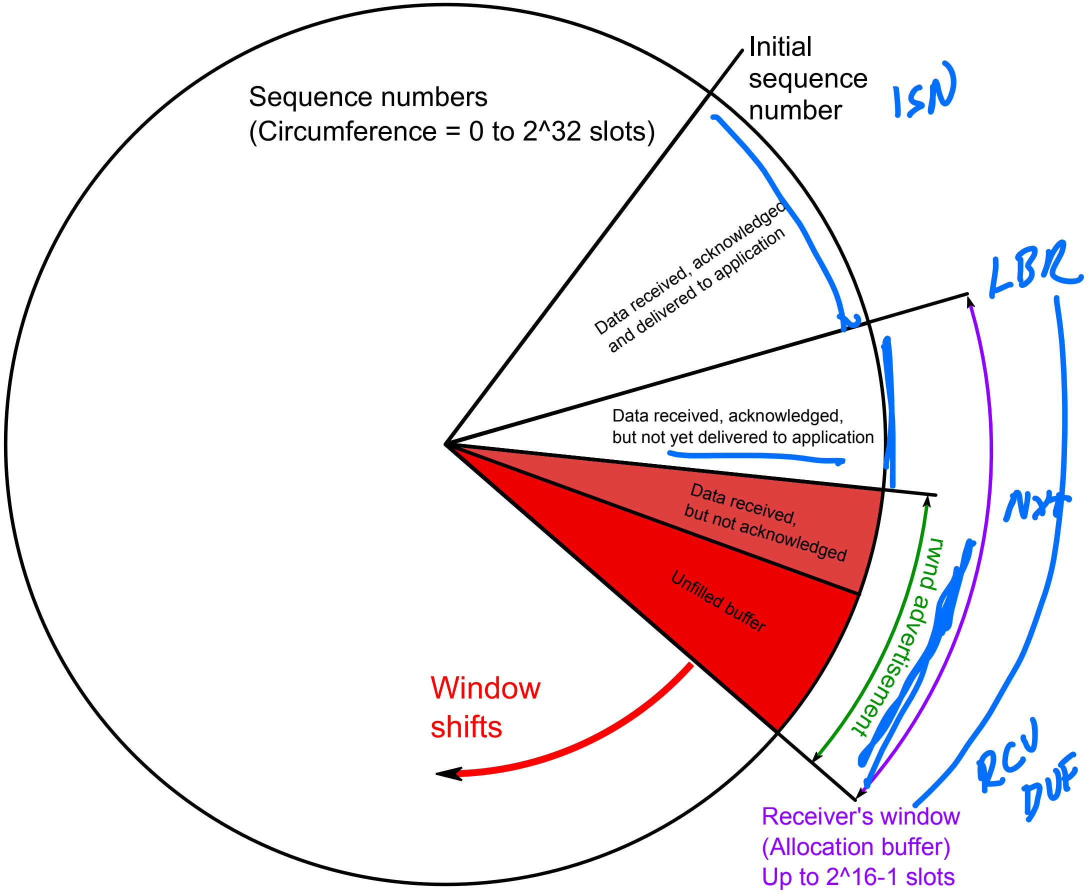
ACK number: last segment the receiver has in order


S
R
1
2
3
4
5
SENDER
1
2
3
4
5
6
UNA
NXT
RECEIVER
1
2
4
5
CDR
NXT
EARLY ARRIVALS
ACK 2
ACK 3
ACK 3
ACK 3
SEG #2 TIMES OUT
ACKS ALWAYS REFLECT LAST BYTE RECEIVED IN ORDER
UPDATE + CONSIDER EARLY ARRIVALS
RETRANSMIT!
3
RECEIVER
1
2
3
4
5
CDR
NXT
ACK 6

Some Visualizations

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

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



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

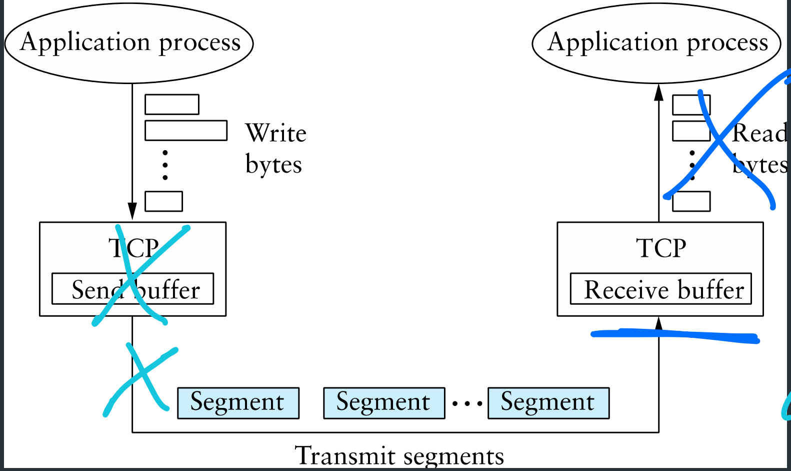
Unfilled buffer

rwnd advertisement

Window
shifts

Receiver's window
(Allocation buffer)
Up to 2^16-1 slots

What happens if the receiving app never reads from its buffer?



FULL

~~WIN~~ ADVERTISED WINDOW DECREASES

- Receive buffer fills up
- Advertised window goes to zero WIN=0
- When WIN=0, sender must stop sending
- Send buffer will fill up (if app keeps sending)
 - If send buffer is full, sender's Write() will block

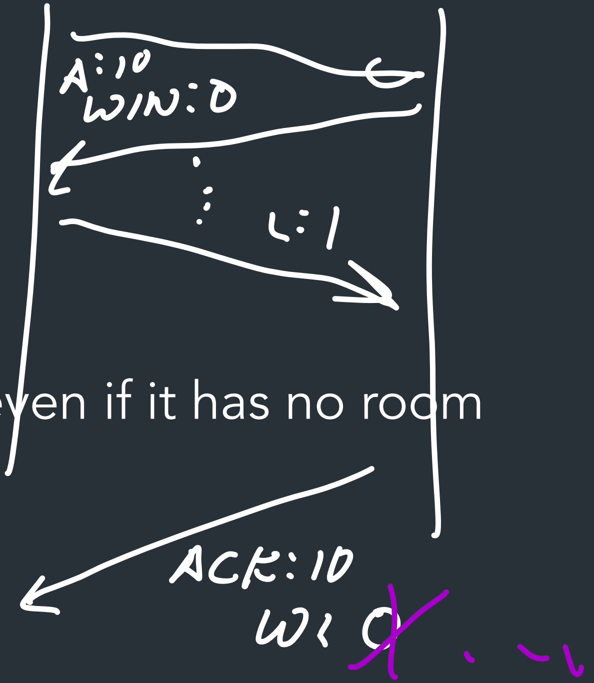
⇒ FLOW CONTROL.

What happens if the receiving app never reads from its buffer?

Problem: need a way for sender to know when space is available again!

Resolution: <u>zero window probing</u>

- Sender periodically sends 1-byte segments
- Receiver sends back ACK with advertised window (even if it has no room for segment



What happens if the receiving app never reads from its buffer?

Problem: need a way for sender to know when space is available again!

Resolution: zero window probing

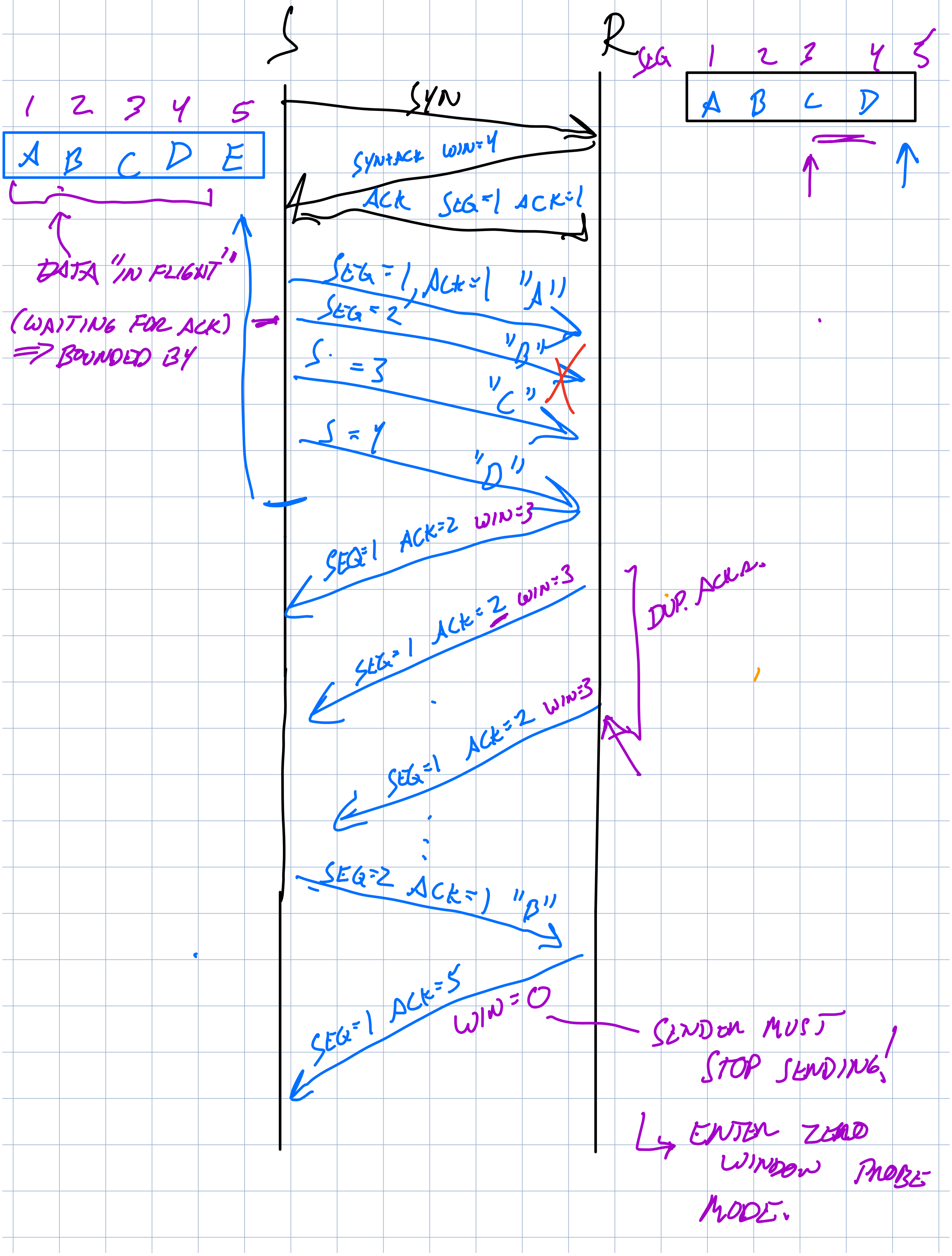
- Sender periodically sends 1-byte segments

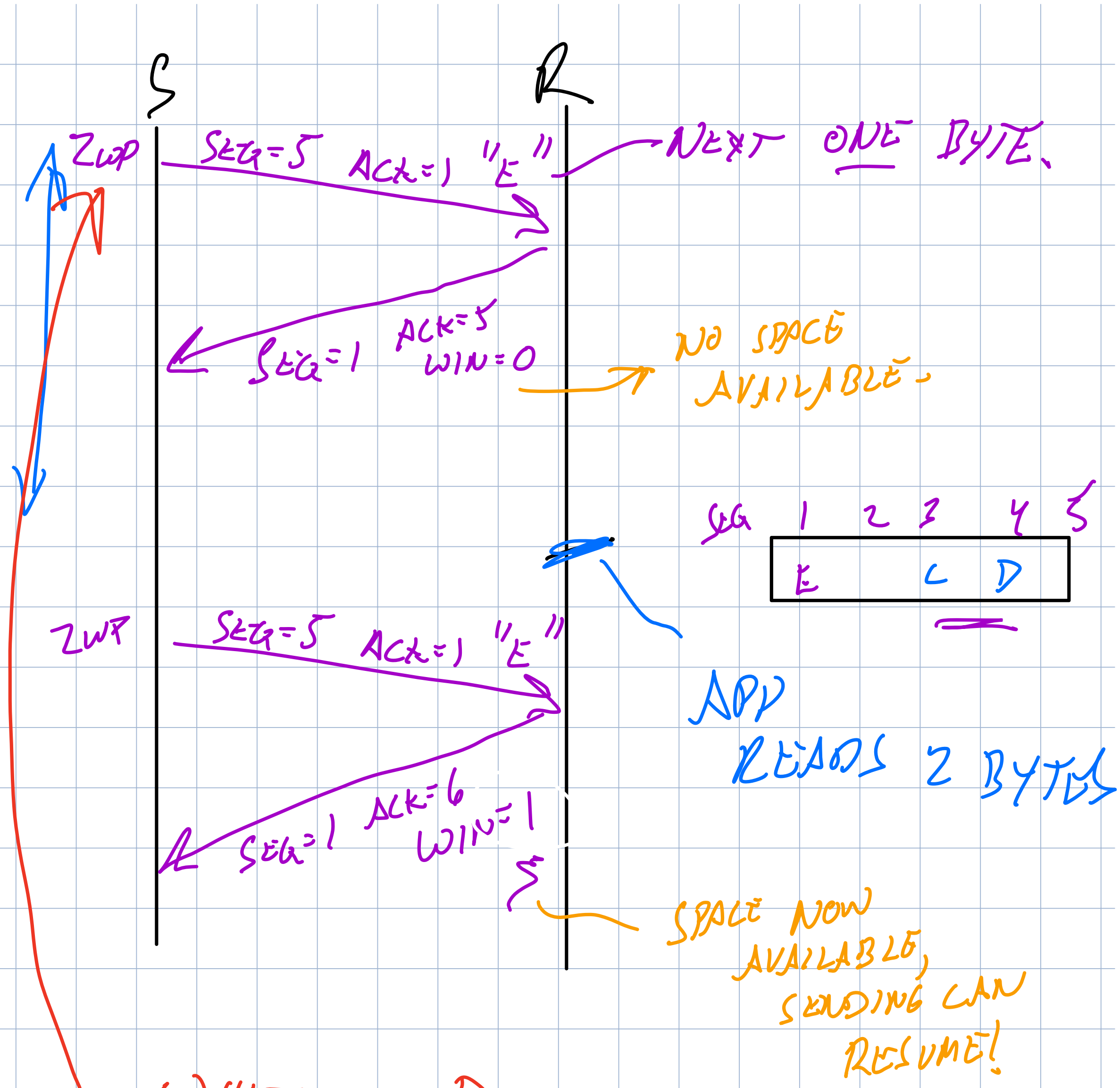
PURPOSELY OUT OF WINDOW.

- Receiver sends back ACK with advertised window (even if it has no room for segment
- Sender can resume sending when win != 0 (preferably when win >= MSS)

SILLY WINDOW SYNDROME AVOIDANCE.

Complete send/recv example with zero-window probing (ZWP part on next page)



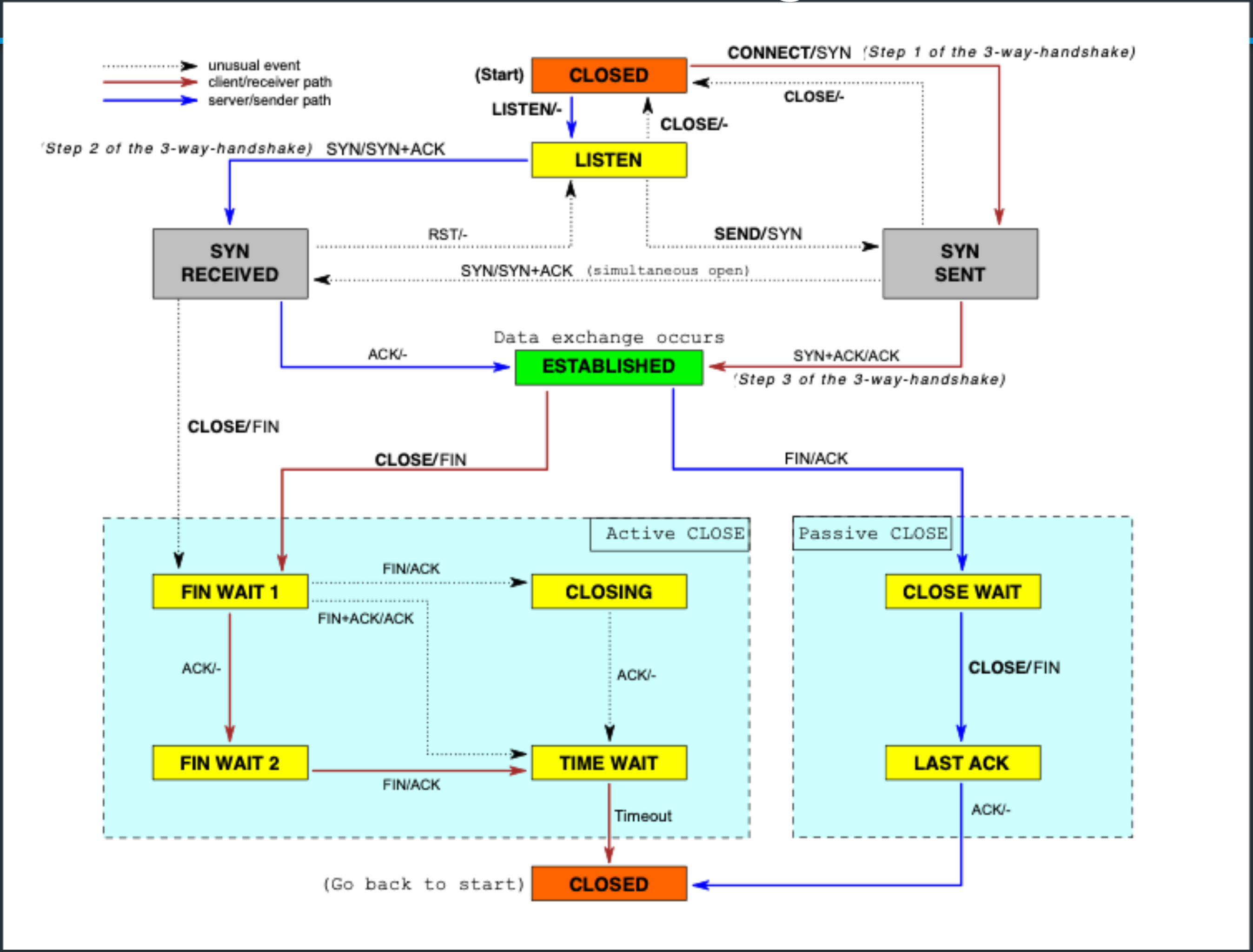


WHAT TO DO WHEN WINDOW IS FULL?

⇒ ZERO WINDOW PROBING

- SENDER SENDS 1-BYTE SEGMENT PERIODICALLY
- RECEIVER WILL ACK, WHICH WILL INDICATE IF ITS WINDOW HAS CHANGED.

TCP State Diagram



How do ACKs work?

- ACK contains next expected sequence number
- Sender: if one segment is missed but new ones received, send duplicate ACK

How do ACKs work?

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

- How to set RTO?

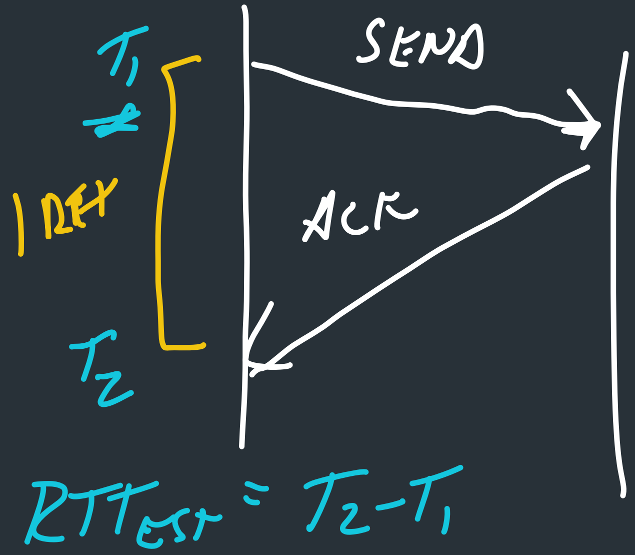
What’s a good timeout value?

- 0.5s? 1s? 0.01s?

Thoughts?

- If timeout is too small, packet might have not arrived (latency)
- If timeout is too long, will affect throughput

=> Can’t just pick a fixed timeout value



$RTT_{EST} = T_2 - T_1$

Strategy: measure RTT based on ACKs received, use this to set a timeout value

=> Timeout time is called RTO

Computing RTO

Strategy: measure expected RTT based on ACKs received

Use exponentially weighted moving average (EWMA)

- RFC793 version ("smoothed RTT"):

$$\mathrm{SRTT} = (\alpha * \overset{\text{PREV EST}}{\underline{\mathrm{SRTT}_{\mathrm{Last}}}}) + (1 - \alpha) * \overset{\text{NEW EACH SEGMENT}}{\underline{\mathrm{RTT}_{\mathrm{Measured}}}}$$

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

UPPER + LOWER BOUND

α = "Smoothing factor": .8-.9

β = "Delay variance factor": 1.3—2.0

$\mathrm{RTO}_{\mathrm{Min}}$ = 1 second

RFC793, Sec 3.7
RFC6298 (slightly more complicated, also measures variance)

Using the RTO timer

Recommended by RFC6298

- Maintain ONE timer per connection
- When segment is sent => set timer to expire after t_{RTO}
- When ACK is received with new data, reset the timer

When the timer expires:

- Retransmit earliest unacknowledged segment
- RTO = 2 * RTO (up to some max)
- If no data after N retransmissions => give up, terminate connection

This is only the beginning…

- Problem 1: what if ACK is for a retransmitted segment?
  - Solution: don't update RTT if segment was retransmitted

- Problem 2: RTT can have high variance
  - Initial implementation doesn't account for this (modern version, RFC6298)
  - Congestion control: modeling network load