Lecture 16: TCP Sliding Window II

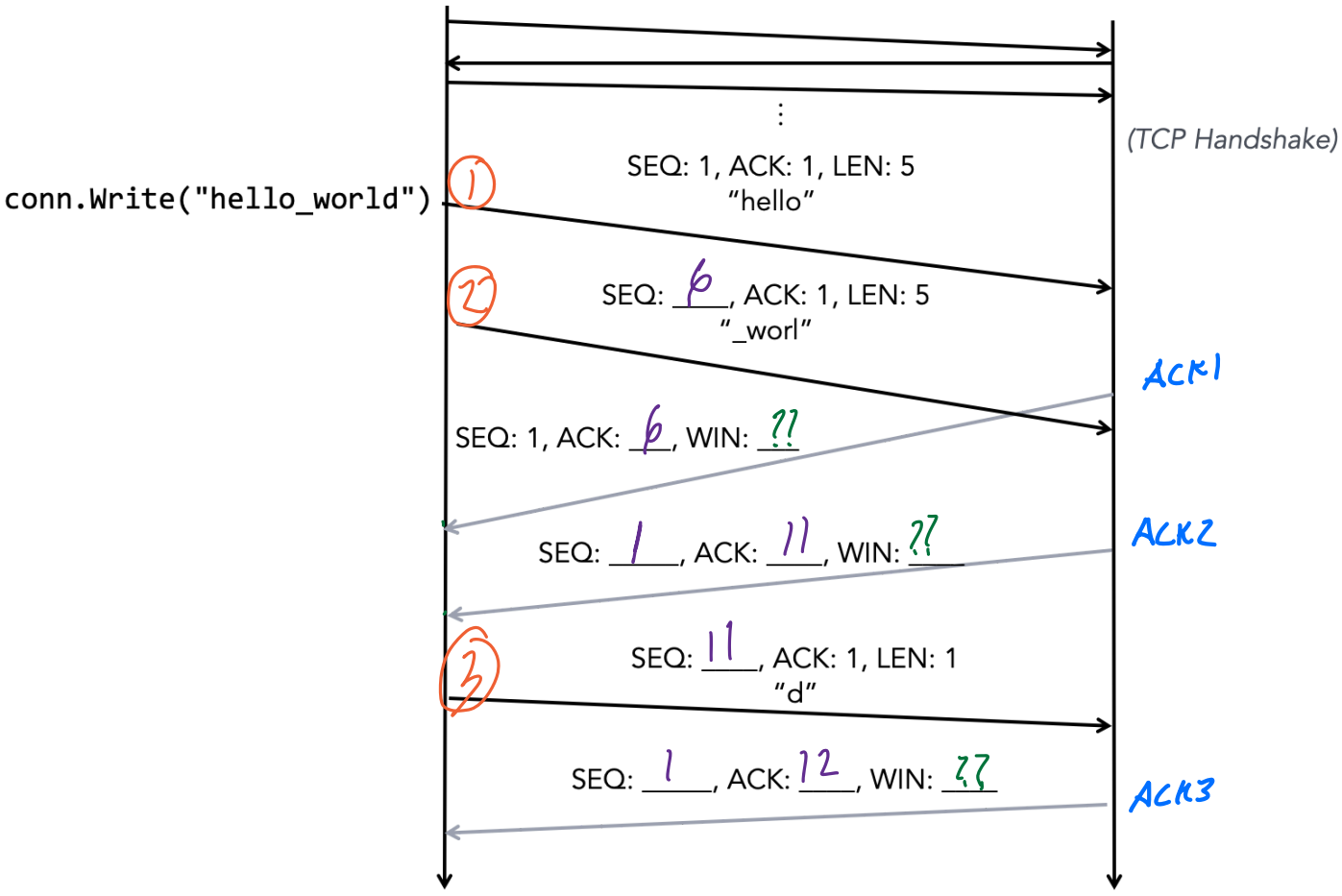
Today
 - Sliding window: receiving side
 - Zero-window probing
 - RTO / Timeouts

Administrivia
 - TCP milestone 1: sign up for meeting by Friday (even if your meeting is after spring break)
 - HW3: out later today (Good practice for TCP!!)

Lecture 16: TCP Sliding Window II

Warmup: Sliding window: What are the SEQ and ACK numbers on the packets? What if the first data segment were dropped?

```
Sender:  conn.Write("hello_world")
```

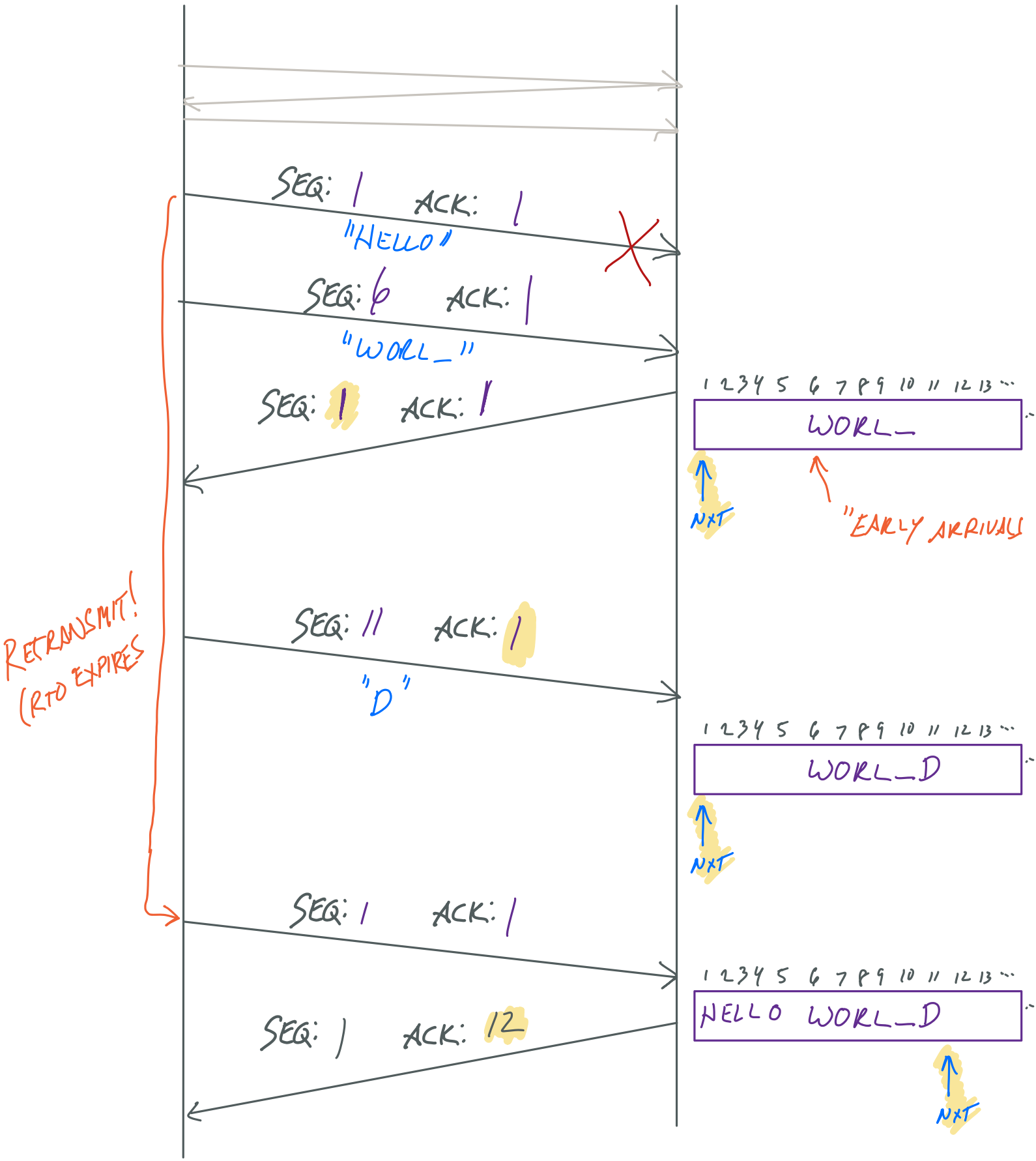


Today: How does the sender know how much data it can send at once?

⇒ WINDOW FIELD

What if the first data segment were dropped?

What if the first segment was dropped?



Sliding window: The story so far

Recap
- Sender breaks data down into segments
- Receiver: receives segments, reassembles them in order

TCP stack needs to buffer data for both parts (send buffer, receive buffer)

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

Open question: How does the sender know how much data to send at once?

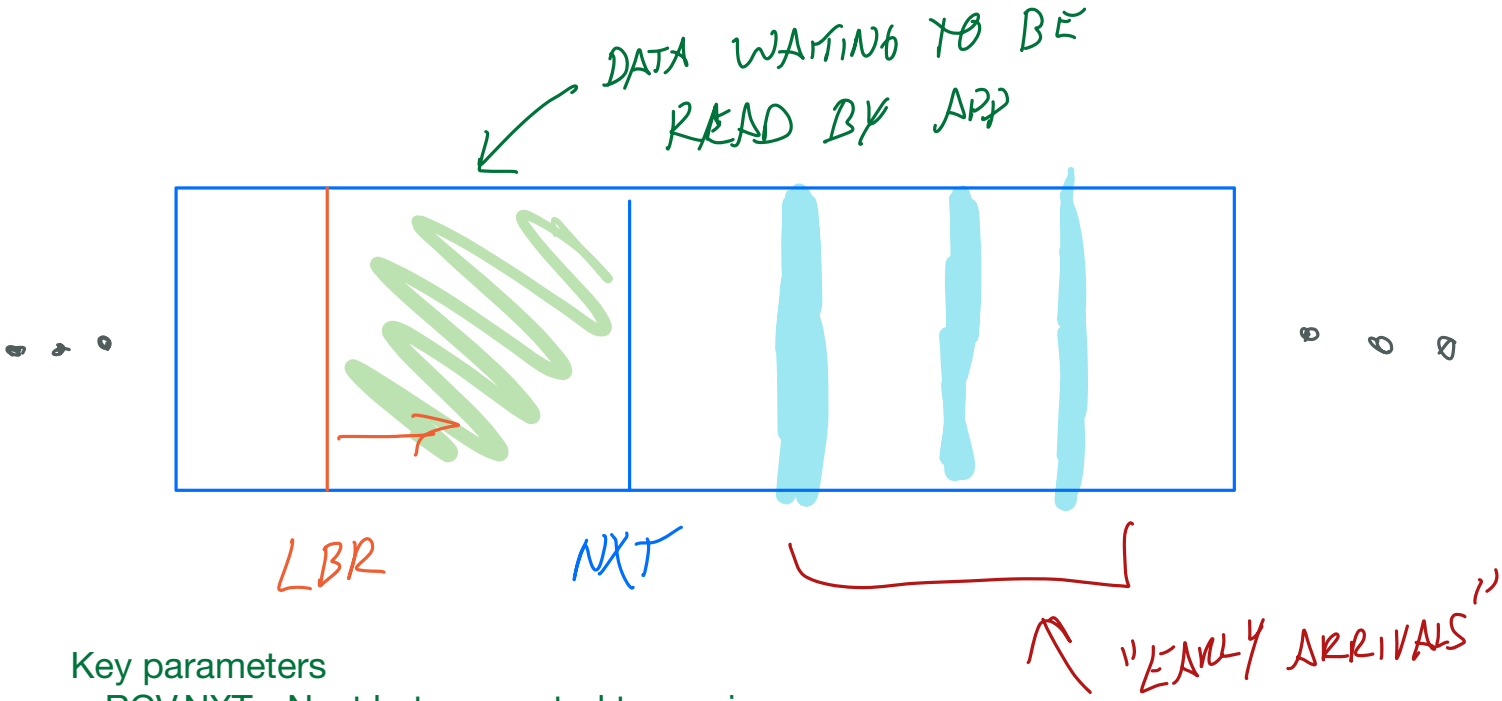
```
buf := ReadFile("all-my-files-ever.zip")
conn.Write(buf)
```

Receiving side has a "receive buffer"
=> Limit on how much data can be "in flight" based on amount of space in receive buffer
=> This is called the receiver's "[advertised] window"

Approach: with every ACK, receiver sends updates about how much space is in its window

Receiving side (RCV)

- TCP stack loads new data from packets (writer)
- Application removes data (conn.Read)



Key parameters

- RCV.NXT - Next byte expected to receive
  => Next sequence number you expect to receive

- RCV.LBR - Last byte read by the application
  => Moved by conn.read

Space left => (max size) - ((nxt - 1) - LBR)
  => Don't count early arrivals

When segment arrives

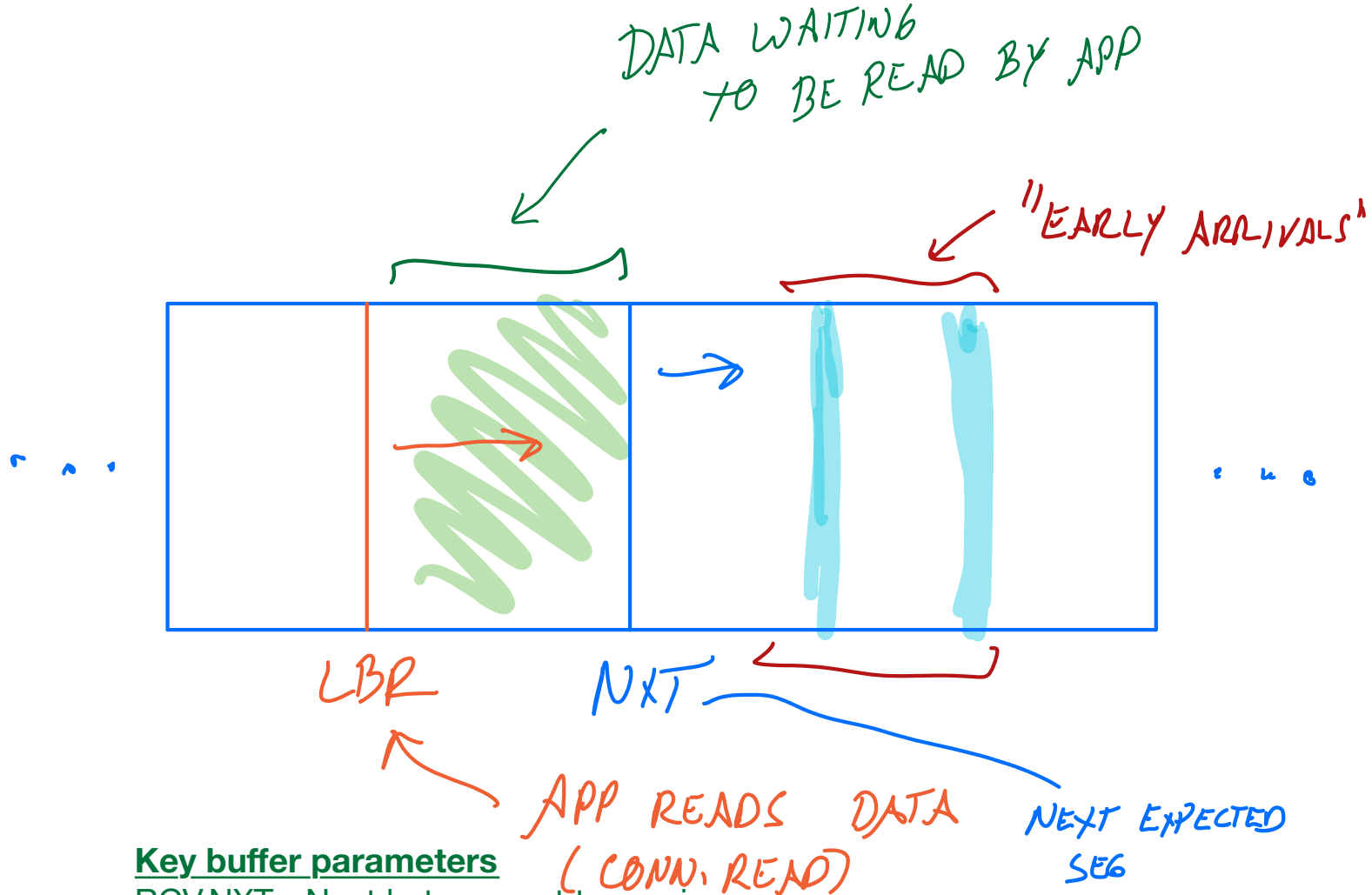
=> If segment falls within window, add to buffer
  => Check for early arrivals
  => Update NXT accordingly

Advertised window (WIN): sent with every segment
  => Tells sender how much space is left
  => Sender should have no more than WIN bytes in flight at any one time

Sliding window: receiving side (RCV)

- TCP stack loads data from received packets (writer)
- Application removes data when reading from socket (i.e., conn.Read)

For more info, see RFC 9293, Sec 3.1, 3.3.1, 3.4



Key buffer parameters

RCV.NXT - Next byte expect to receive
Next sequence number expect to receive

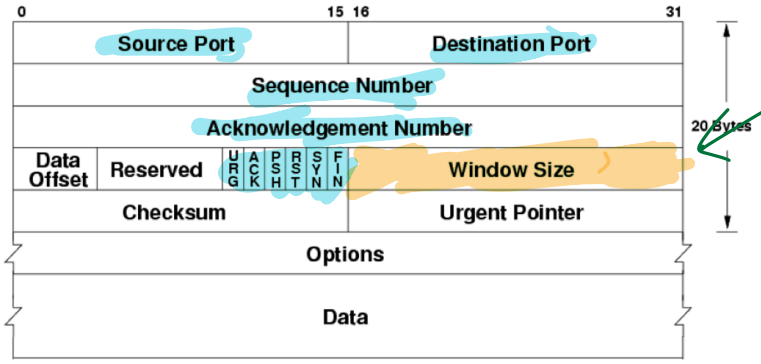
RCV.LBR - Last byte read by application (moved by conn.Read)

Advertised window: amount of space that remains in the buffer
=> (max buf size) - ((nxt - 1) - LBR)
(Doesn't count early arrivals)

When a segment arrives

=> If segment falls within expected window, add to buffer, also check for early arrivals and advance NXT accordingly

Advertised window (WIN)

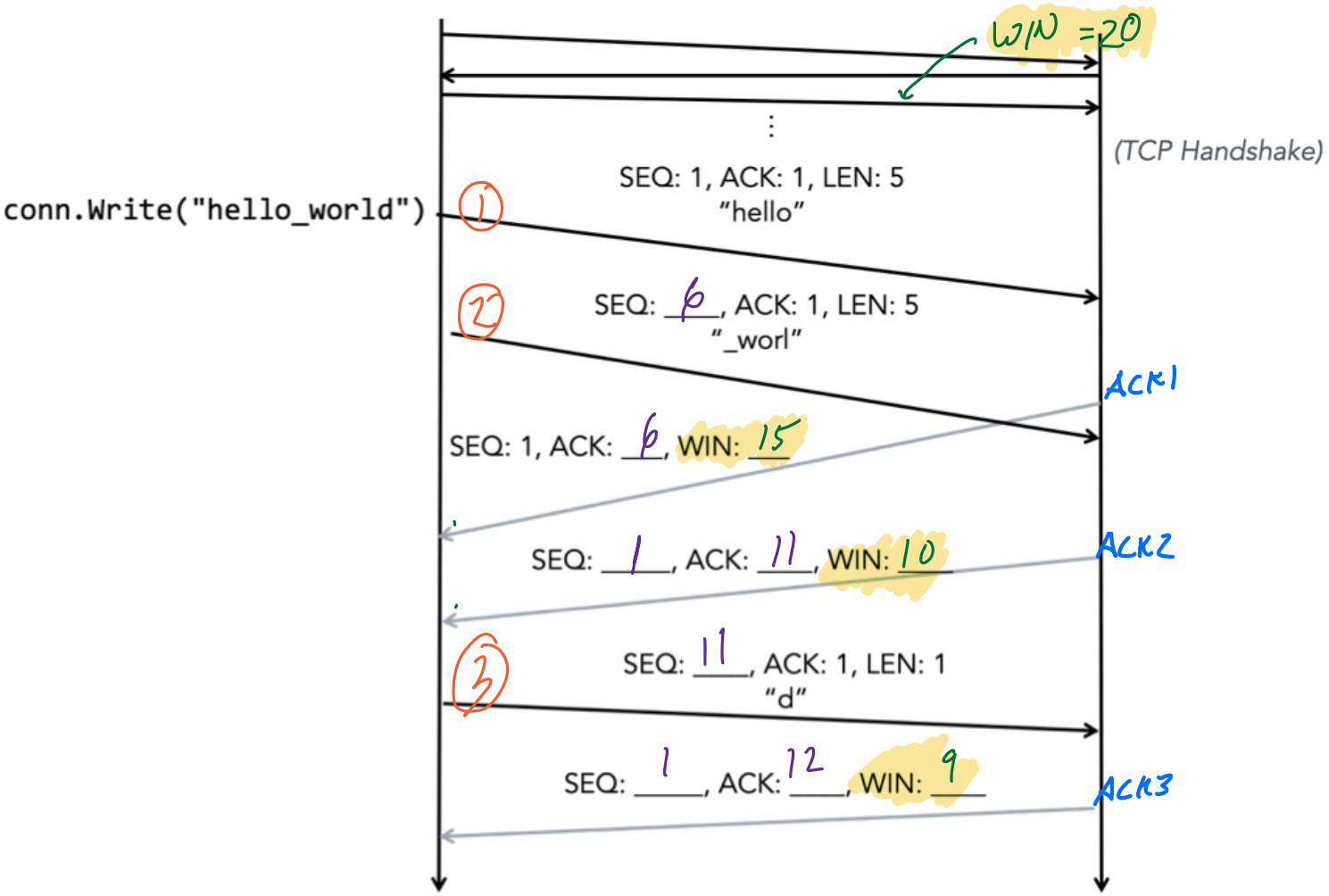


ADVERTISED WINDOW

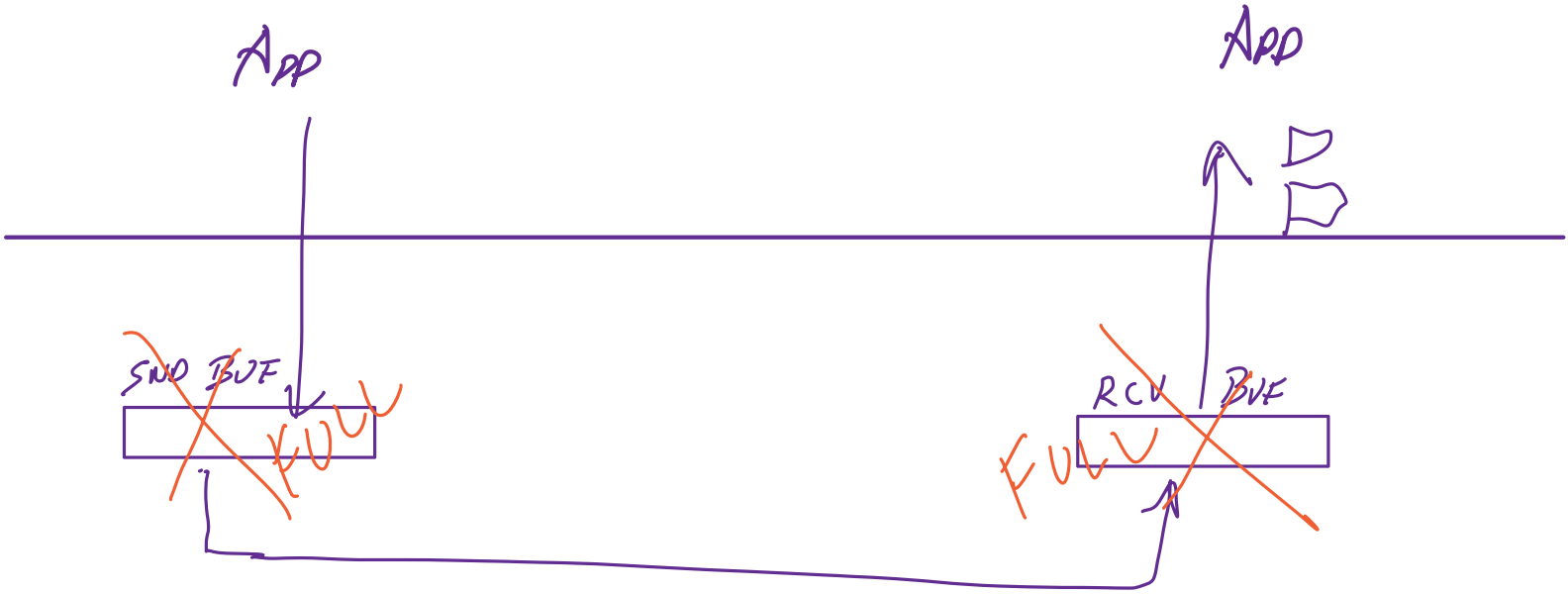
⇒ SENT WITH EVERY SEGMENT!

Example: from the warmup problem

Suppose receiver's buffer is 20 bytes

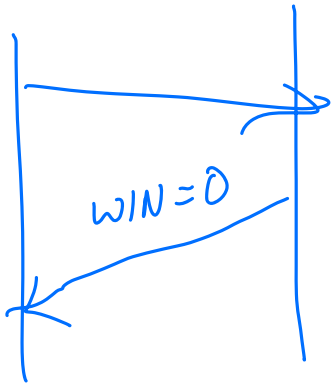


What happens if receiver never reads from receive buffer??



What happens

- **Receive buffer fills up**
- **Advertised window goes to zero**
- **Sender must stop sending since win = 0**
  => Bytes in flight => 0
- **Send buffer fills up**
- **When send buffer is full, sending app can't send anymore**
  => Conn.write will block



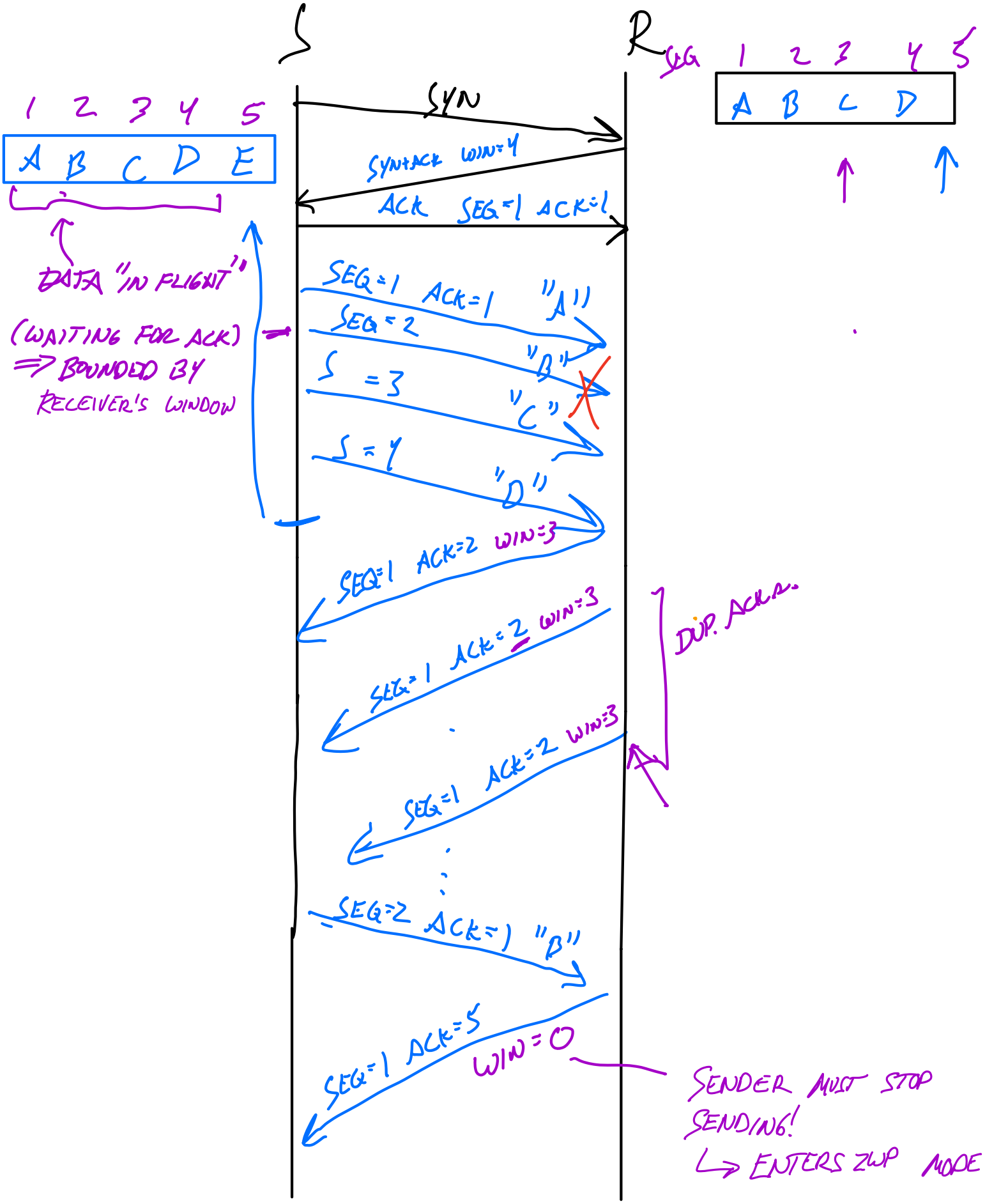
Problem: Now we have a situation like a deadlock--need a way for sender to know when space is available in the buffer again!

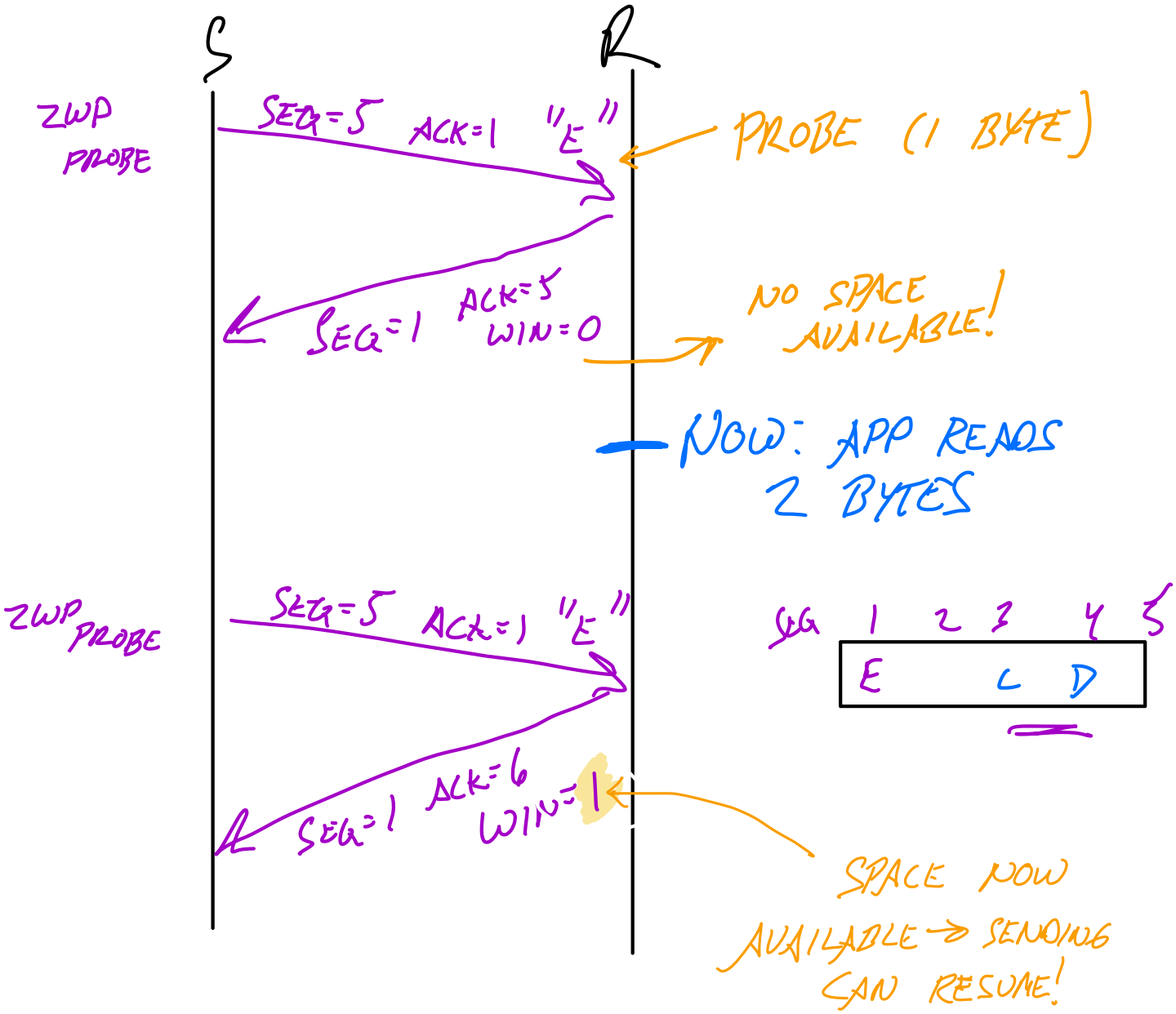
=> Zero window probing (ZWP)

When receiver advertises a zero window:

- Periodically, send 1 byte segments ("probes"),using whatever data is in send buffer,
- Receiver sends back ACK, with advertised window (even if it has no room)
- Sender can resume sending when win != 0
  - (preferably, resume when win >= MSS) => avoids "Silly Window Syndrome"
    => Can resolve with SWS avoidance algorithm (not required for project)

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



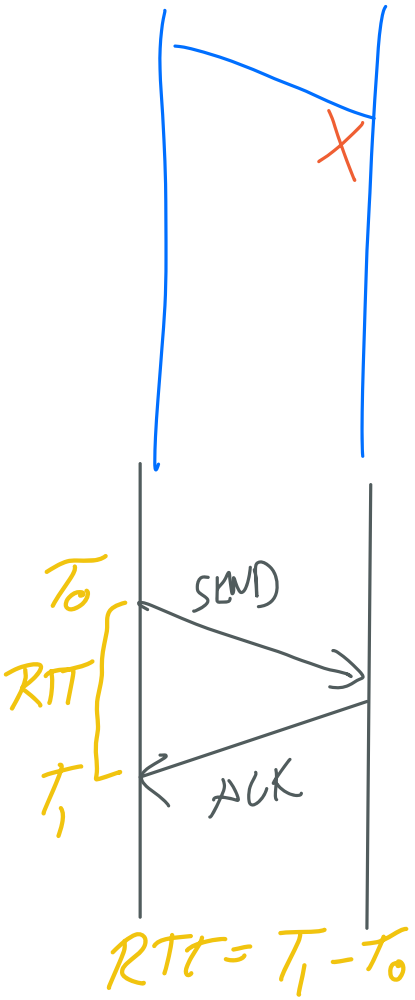

S
R
ZWP PROBE
SEQ=5 ACK=1 "E"
PROBE (1 BYTE)
SEQ=1 ACK=5 WIN=0
NO SPACE AVAILABLE!
NOW: APP READS 2 BYTES
ZWP PROBE
SEQ=5 ACK=1 "E"
SEQ 1 2 3 4 5
E C D
SEQ=1 ACK=6 WIN=1
SPACE NOW AVAILABLE → SENDING CAN RESUME!

Timeouts and Retransmissions

What's a good timeout value before retransmitting?
0.1s, 1s, 0.001s?

Why is this a bad question?
=> If timeout is too small, packet might not have arrived yet (latency)
=> If timeout is too big, will affect throughput (because you're not sending)

Takeaway: Retransmission timeout (RTO) should be based on RTT to destination



Strategy: measure RTT based on ACKs received, use this to set time

=> Timeout called RTO

Sketch: when receiving an ACK, if ACK was not for a retransmitted packet, you can measure RTT

TCP does this using a weighted moving average:

SRTT = (alpha)(SRTT_old) + (1 - alpha)*(SRTT_new)
- Set RTO based on some multiple of RTT
- Also add some upper and lower bound

Computing RTO

(Copying in an old slide since the formulas didn't reproduce well in the handout)

Strategy: *measure* expected RTT based on ACKs received

Use exponentially weighted moving average (EWMA)

- RFC793 version ("smoothed RTT"):

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

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

UPDATE BASED ON NEW + OLD VALUE

UPPER + LOWER BOUND

α = "Smoothing factor": .8-.9

β = "Delay variance factor": 1.3—2.0

RTO_{Min} = 1 second

OK FOR INTERNET, BUT WE'LL USE A LOWER VALUE FOR PROJECT

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

Bonus content: Implementing the retransmission timer efficiently

Here's what RFC6298 recommends managing the timer:
- Maintain ONE timer per connection
- When segment is sent => set timer to expire after t_RTO
- When an ACK is received with new info, reset the timer

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

In practice, this is only the beginning...

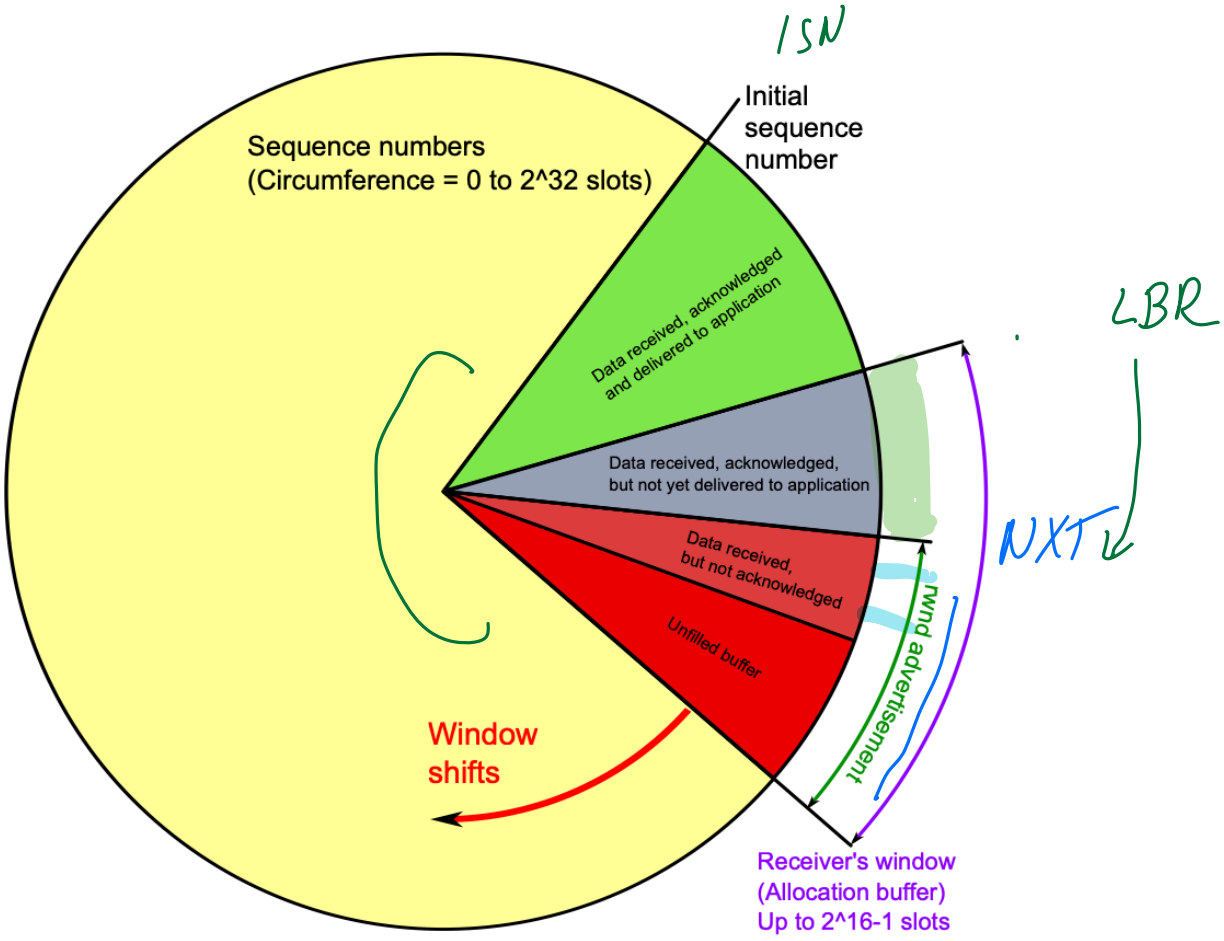
Problem: RTT can have high variance!

- Original version (in these notes) doesn't account for this. More versions (RFC6298, RFC9293) propose versions based on more modern research about variance of typical RTTs on the Internet

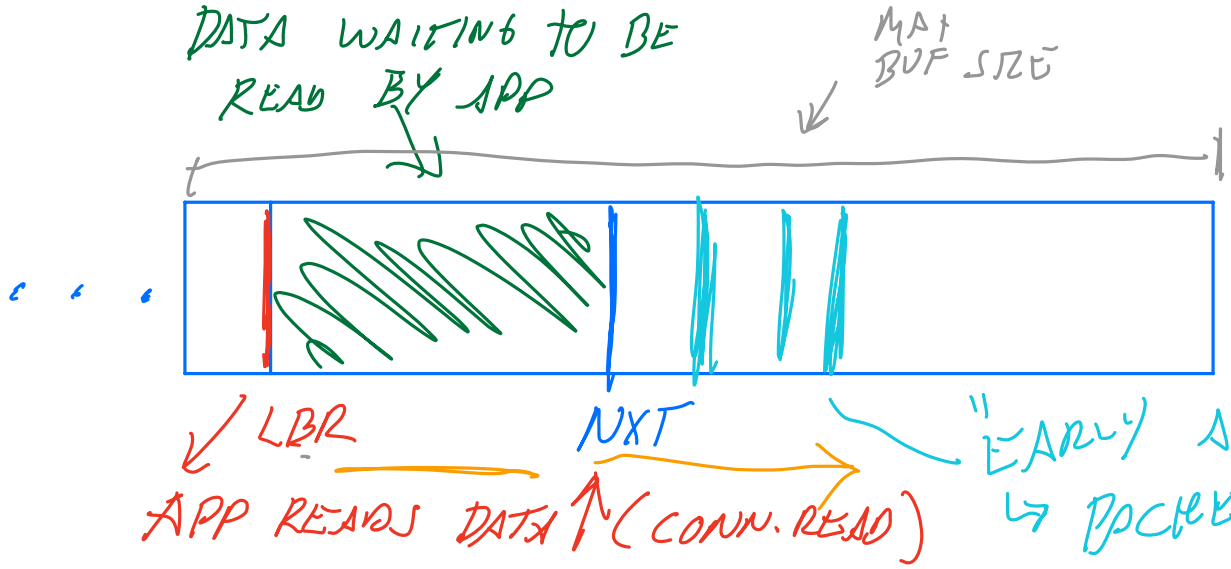
=> In particular, RTT can be affected by congestion => need better ways to model network load (more on this later, when we discuss congestion control)

Older version of the notes / more pictures after this (in case these are clearer)

"ACK clocking"



RECEIVING SIDE (RCV)



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 W/IN BOUND)

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

S.SEQ < RCV.NXT AND S.SEQ < RCV.NXT + RCVWND

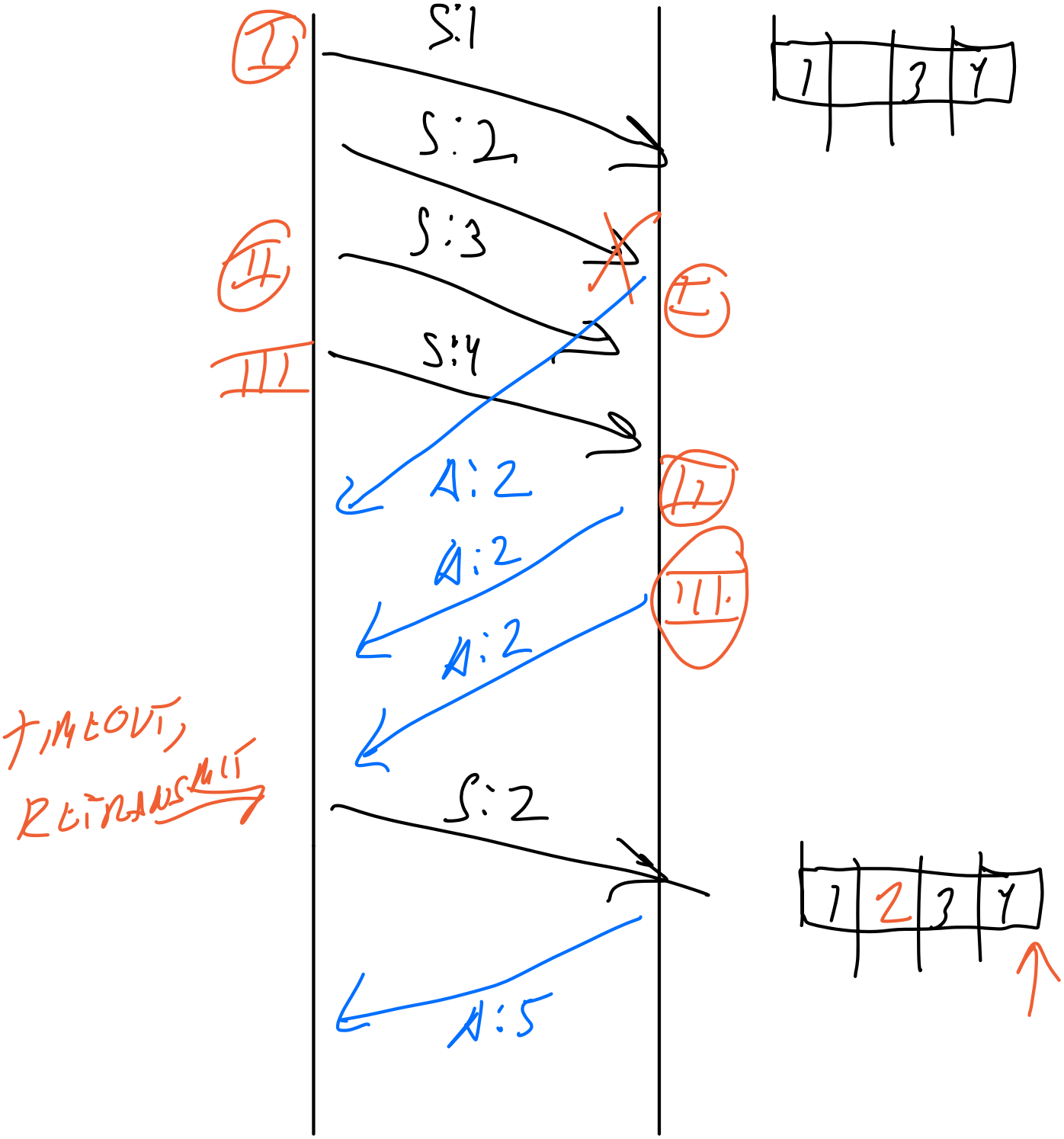
OR

(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

Q from class: why can't we recompute RTO on retransmission?
=> No way to know if ACK is for initial send, or retransmission, so estimate could be wrong