Today
- TCP sending
- Sliding window (part 1)

Adminstrivia
- HW2 due tonight
    - Problem 3 is good practice for TCP
-HW3: remember for after break
- TCP milestone 1 signups: out today
    - Some meetings before break, some after break
    - Pick what's most useful for getting feedback
    - Be prepared to move to milestone 2 by end of milestone meetings (Tuesday after break)

Lecture 14: TCP III: Sending and sliding window

Warmup: say you have the following socket table:

Proto	Local (yours)		Remote (theirs)		STATE
	IP	Port	IP	Port	
tcp	1.2.3.4	22	5.6.7.8	12453	ESTABLISHED
tcp	1.2.3.4	22	100.3.15.7	66452	ESTABLISHED
tcp	*	22	*	*	LISTEN

... and you receive the following packets:

Pkt#	Proto	Source		Destination	
		IP	Port	IP	Port
P1	TCP	5.6.7.8	12453	1.2.3.4	22
P2	TCP	5.6.7.8	55444	1.2.3.4	22
P3	TCP	10.5.4.23	44344	1.2.3.4	22
P4	TCP	10.0.0.1	43233	1.2.3.4	80

When receiving, destination IP is "your" IP => local IP

→ MATCH ON S_1!

NO EXACT MATCH, BUT MAPS TO LISTEN SOCKET S_3

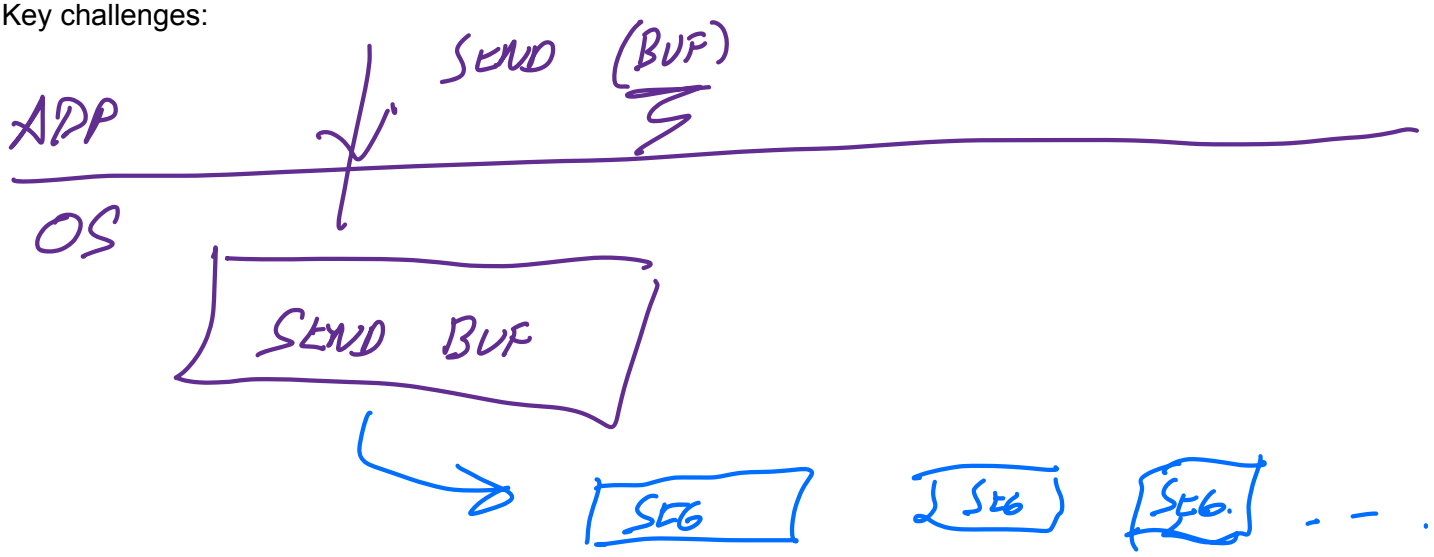
NO MATCH ON ANY SOCKET → DROP

How do you match these packets to sockets in the socket table?

Idea:
- App calls send(), loads send buffer in OS
- TCP stack divides data up into segments

Sending data (ESTABLISHED state)

Key challenges:

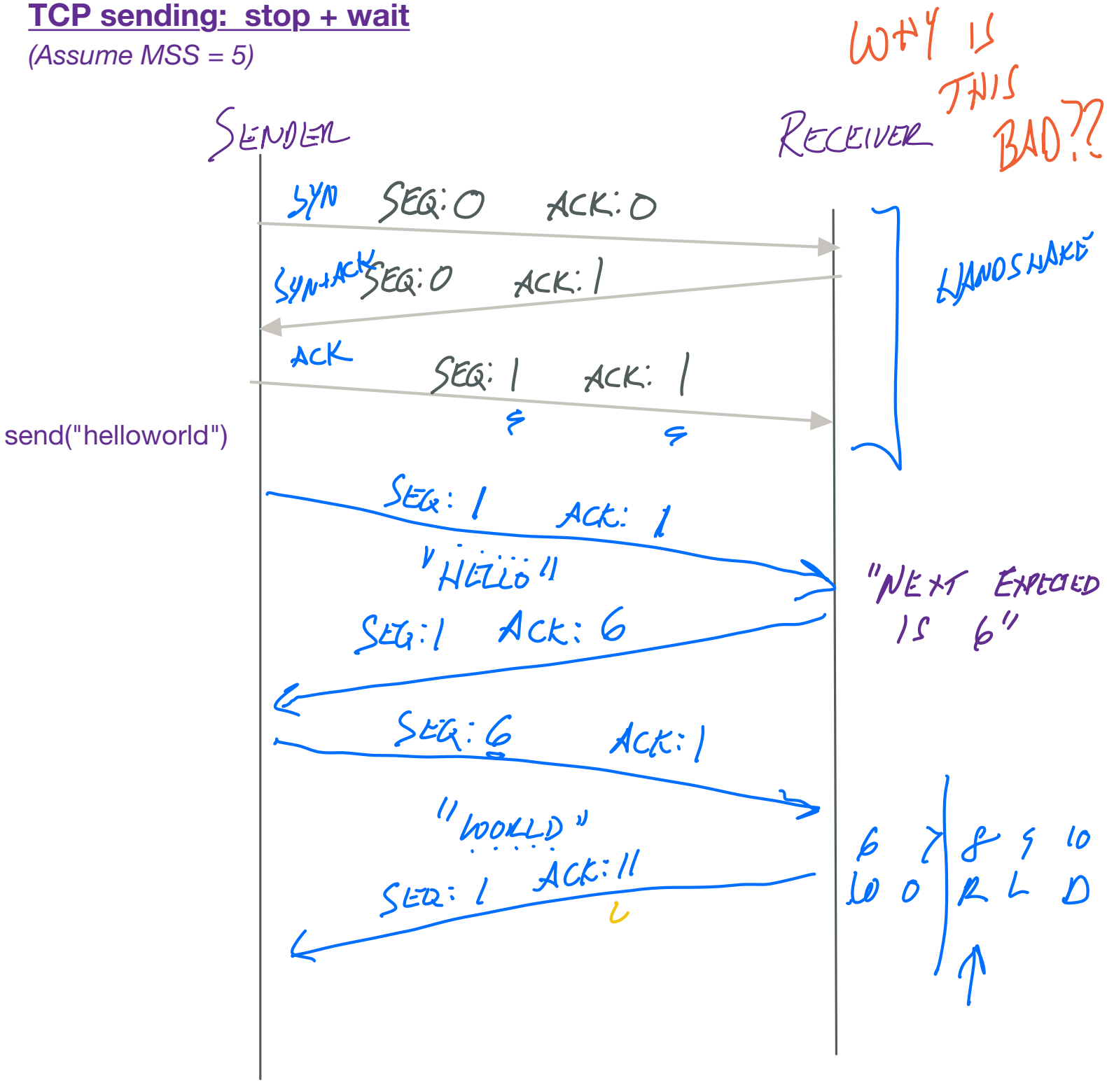


Terminology: MSS

Maximum segment size (MSS)
=> no segment may be lager than this

TCP sending: stop + wait

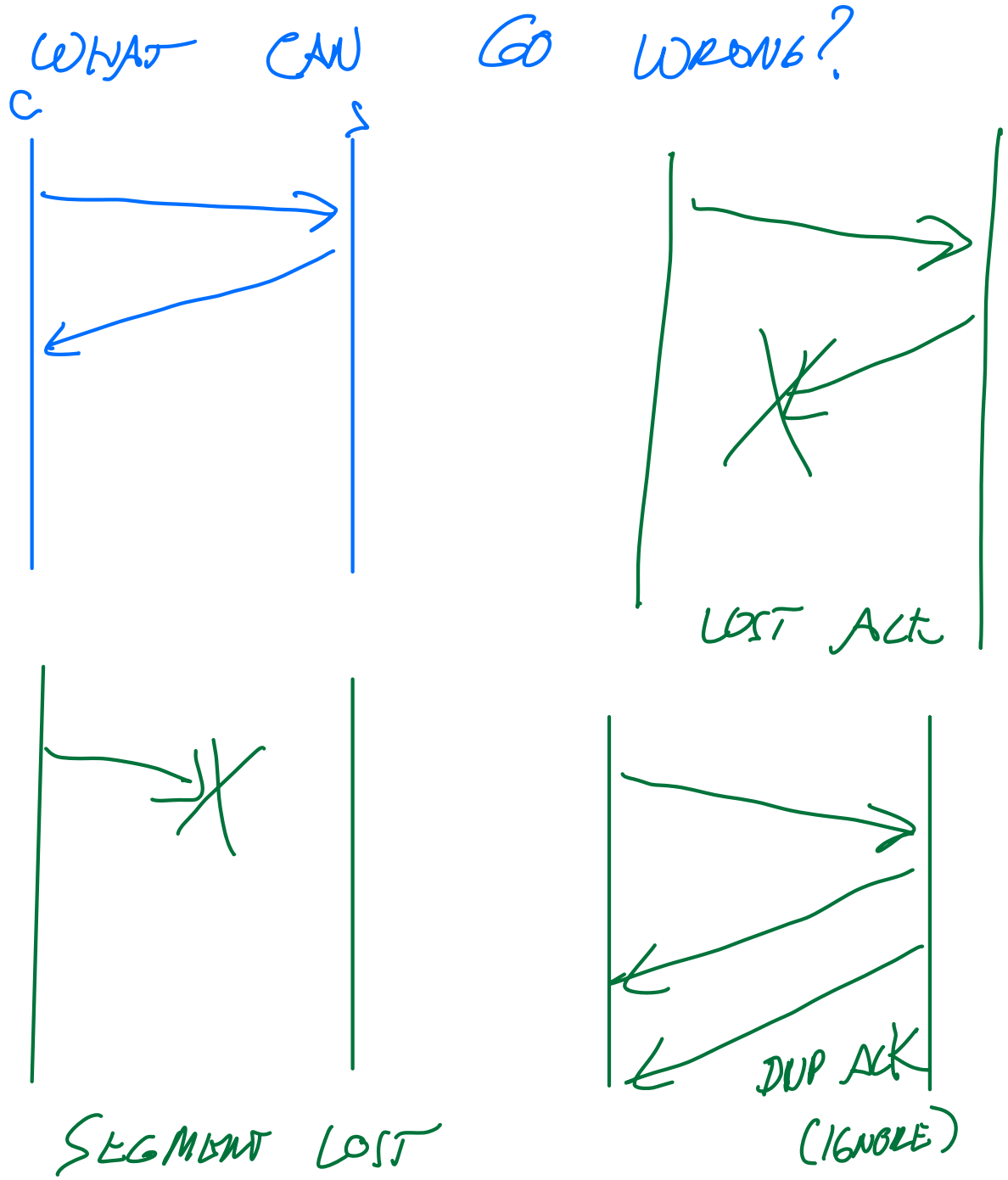
(Assume MSS = 5)



Key fields:

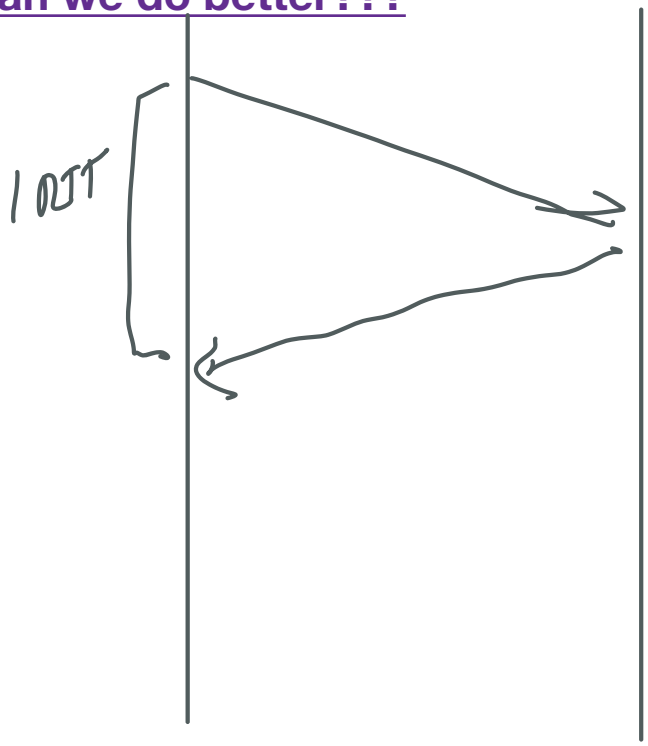
- SEQ: segment starts that this position in data stream"
- ACK: "I have up to byte (B - 1), give me byte B next"

How can we do better????



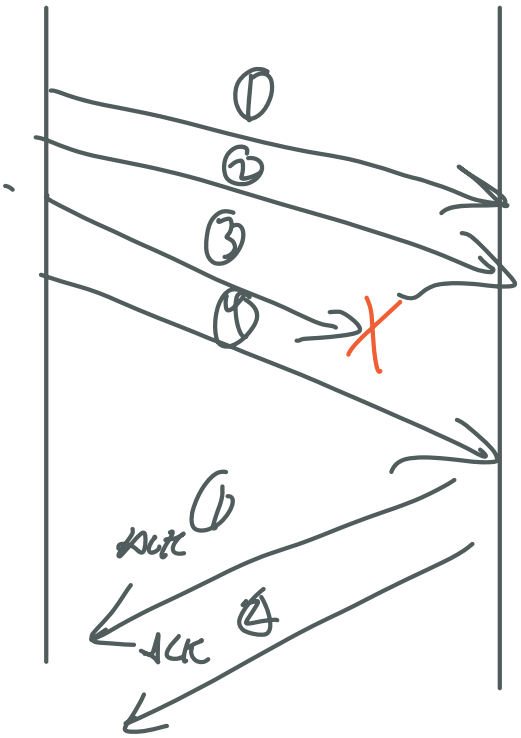
Send segment
If no ack within certain time, retransmit segment
=> Amount of time to wait is called RTO
=> If N retries without ack, give up and error

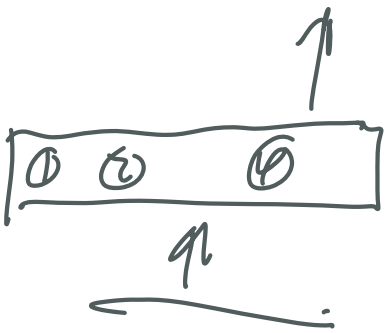
Can we do better???



Stop and wait: need to wait one RTT for each segment => very high latency

Goal: would like to have multiple segments "in flight" at one time => use more of the network bandwidth



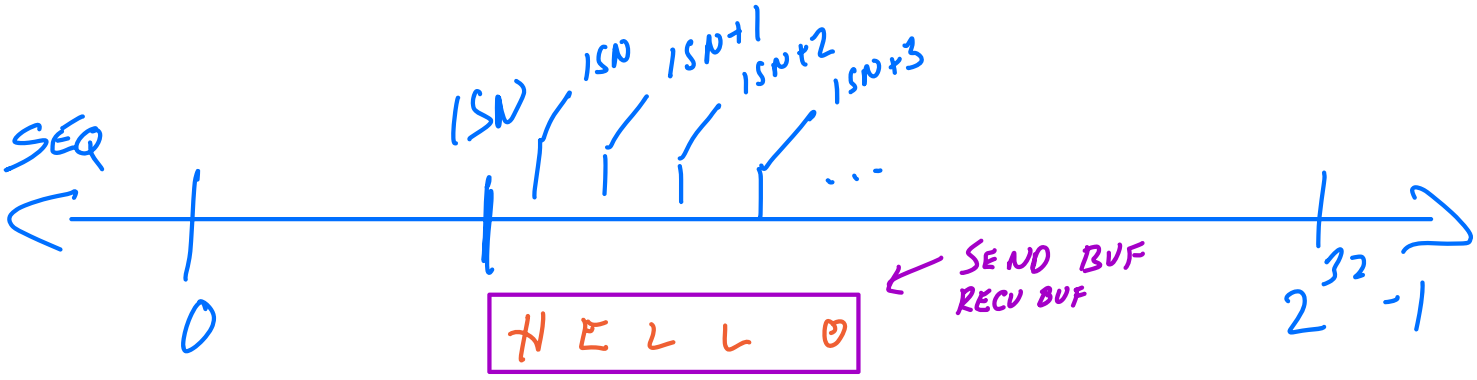


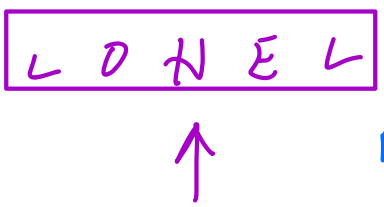
Idea: Have a "window" of packets that can be sent at one time
=> This is called **sliding window**

Challenges:
- Receiver needs to put segments in order
- Segments might arrive out of order
- Sender: needs to know how many segments to send

=> Flow control: need to make sure we don't overwhelm receiver, or network

First: how sequence numbers + buffering work (conceptually)





In practice: send/receive buffer is small (relative to the total sequence number of space), and has a fixed size

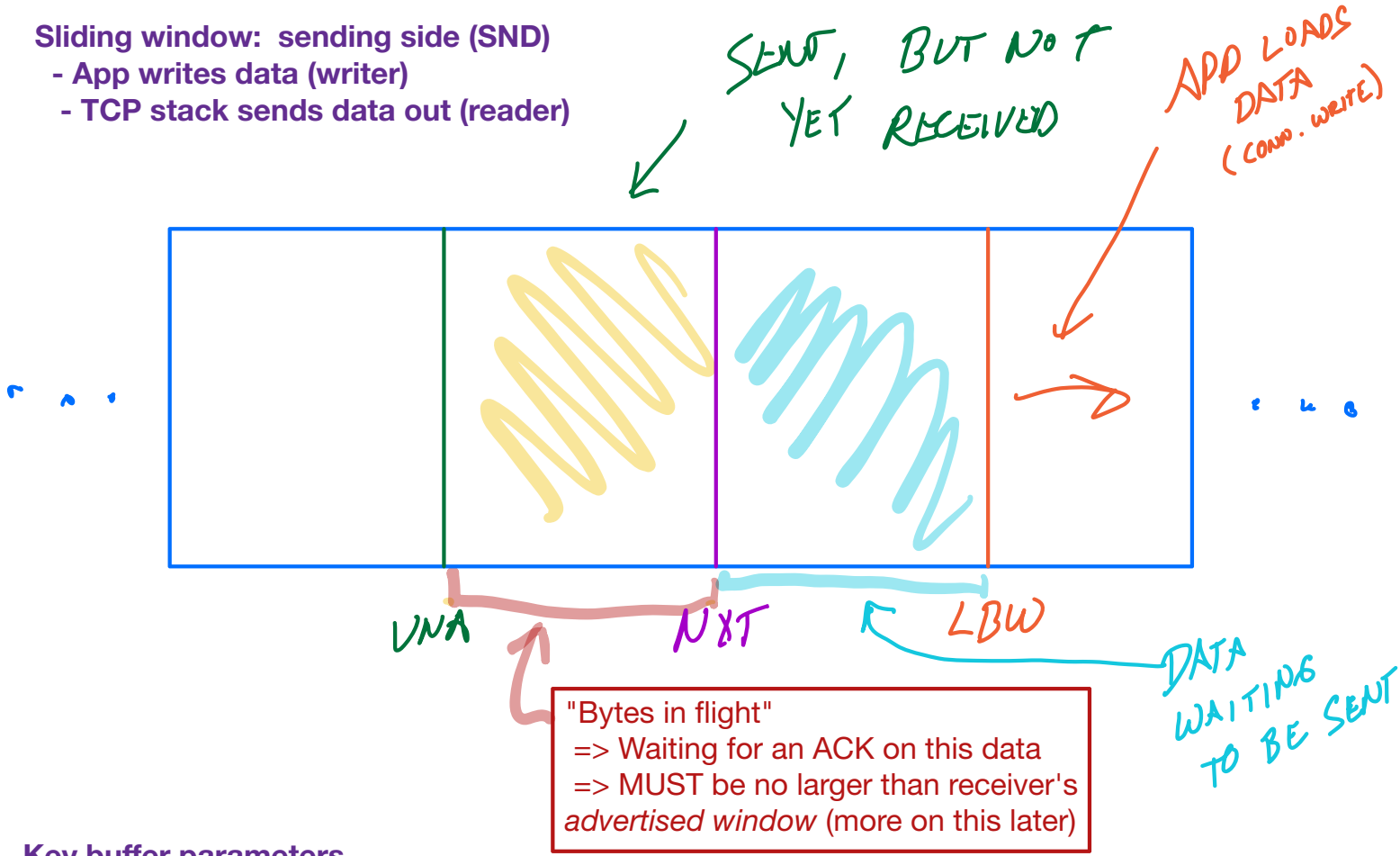
(In project, we recommend 2^16 = 65536 bytes)

Therefore: two "domains" for thinking about data

- "Sequence number space starting at ISN
  => These are the numbers in the SEQ/ACK fields

- "Buffer space": how you store data in the send/receive buffer in the TCP stack
  => Implemented as a circular buffer (or "ring buffer")

Sliding window: sending side (SND)
- App writes data (writer)
- TCP stack sends data out (reader)



Key buffer parameters

SND.UNA - Oldest unacked segment => waiting for an ACK, might need to be retransmitted

SND.NXT - Next sequence number to send out
- The next byte to send

SND.LBW - last byte written (moved by application)
=> If no more space in buffer, sender must block

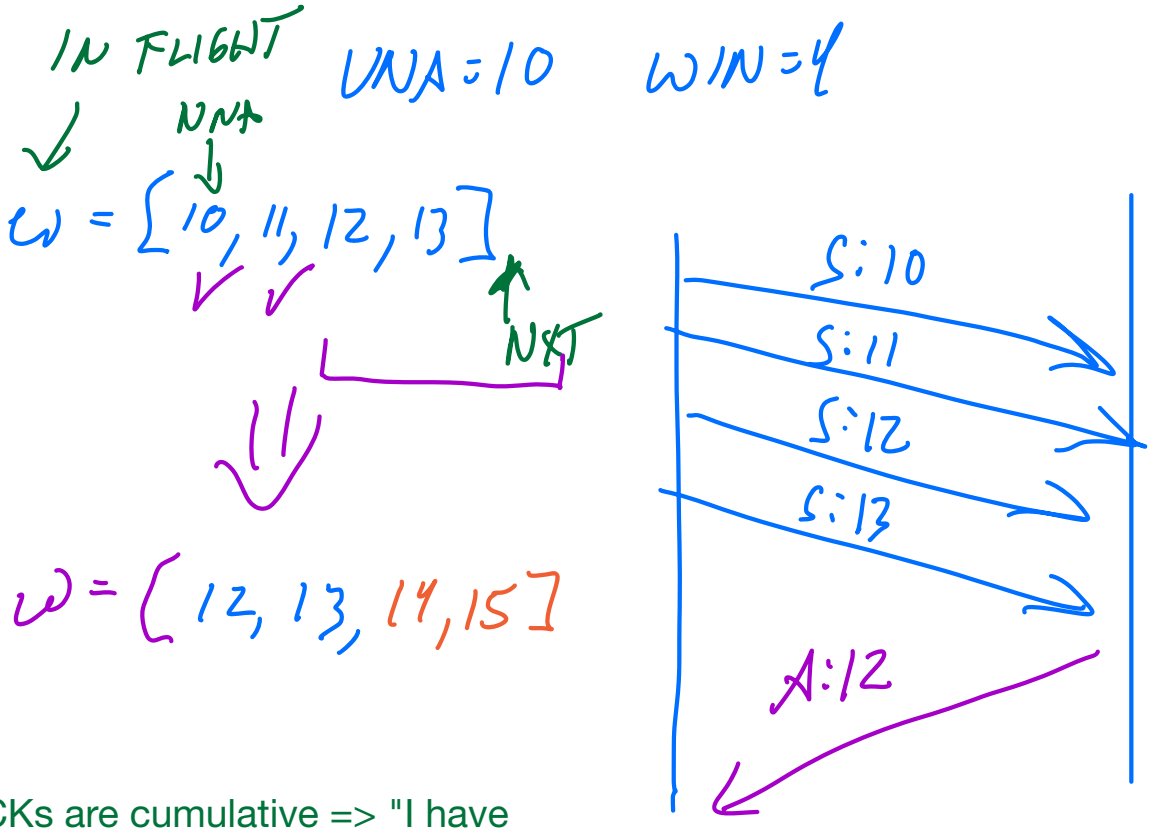
Sender operation:
=> Sender sends data up to NXT
- Keep track of "in-flight segments" UNA..NXT
=> Might need to get retransmitted ("retransmission queue")

When you get an ACK for segment S, should be within this window
UNA < S.ACK <= NXT

If not within the region, drop it

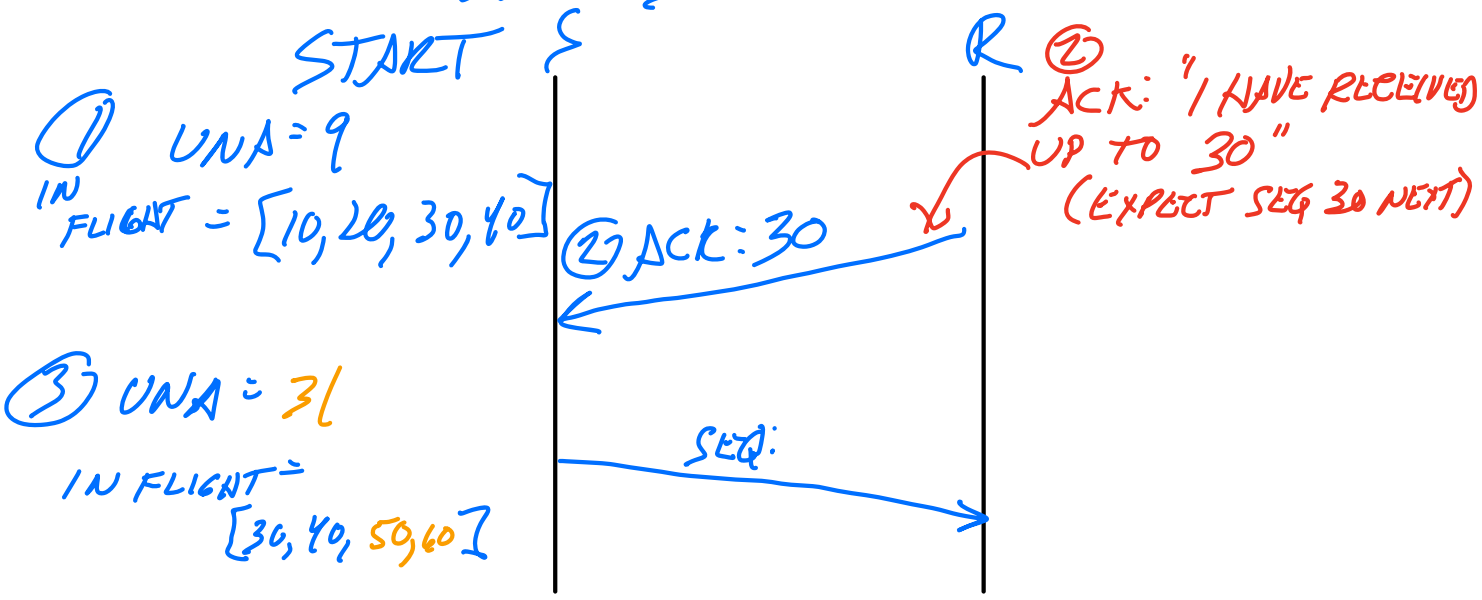
Otherwise, UNA += (Amount of data ACKed)

EXAMPLE: 1-BYTE SEGMENTS



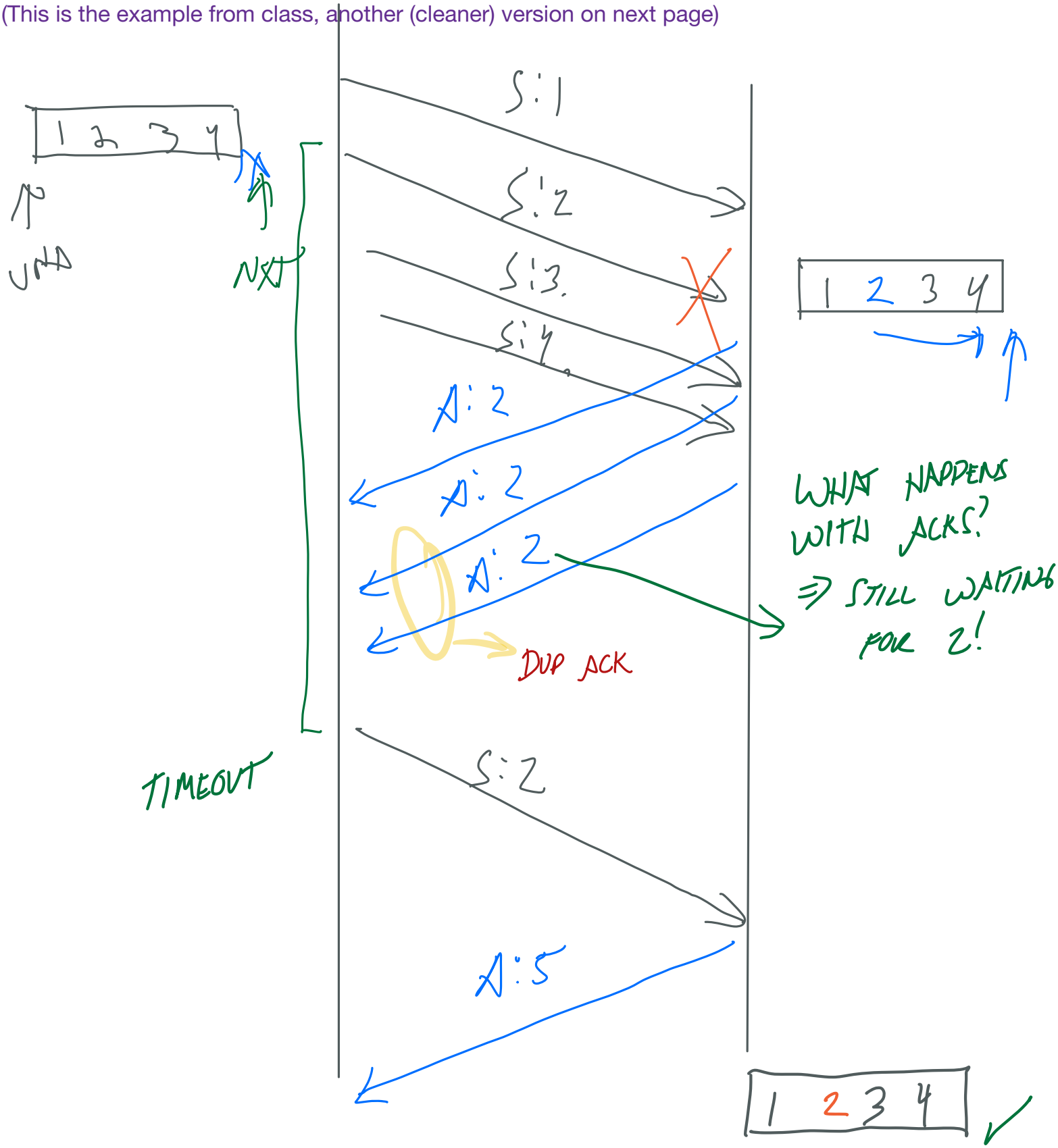
ACKs are cumulative => "I have everything up to this point"

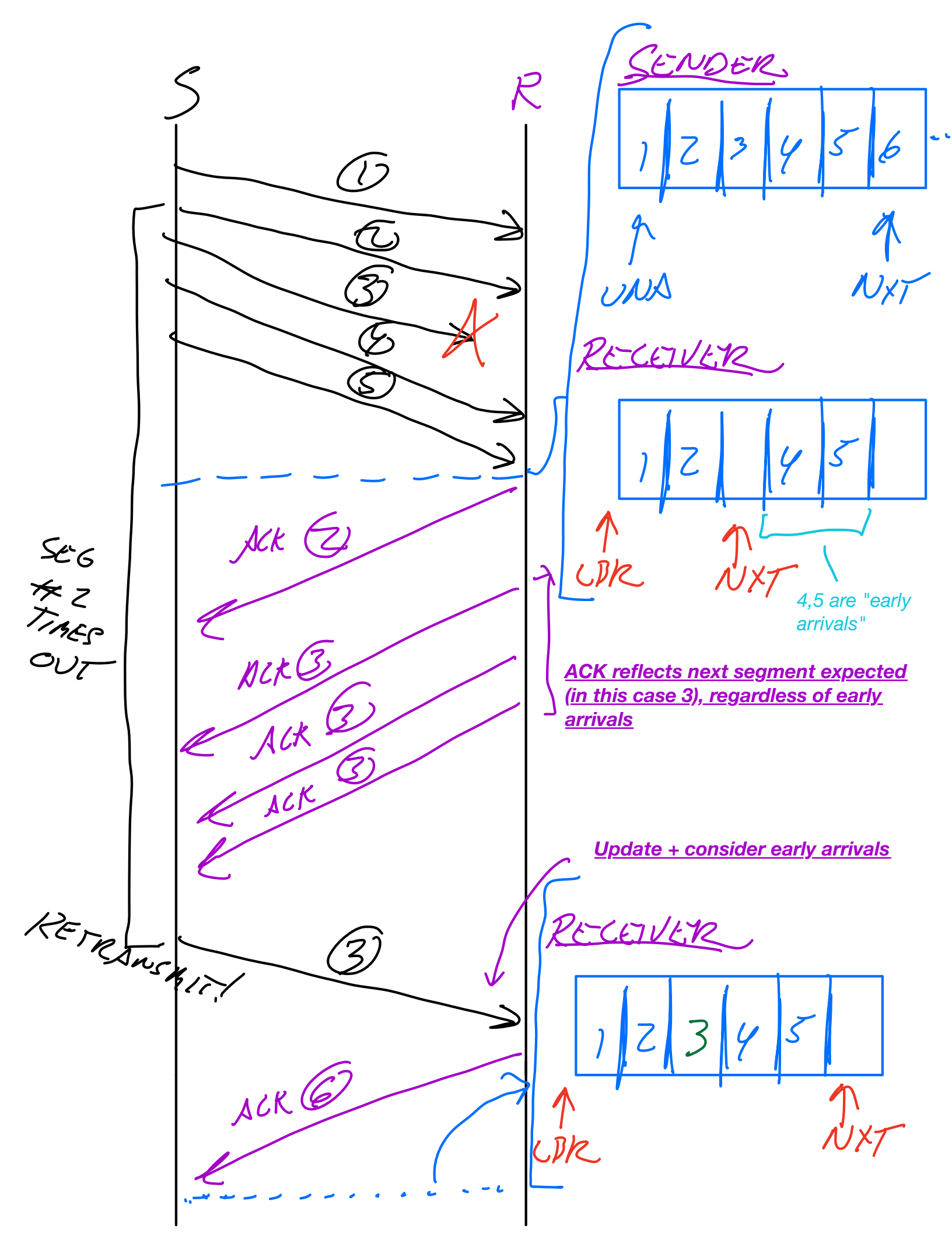
Ex. 10 BYTE SEGMENTS



What happens when a segment is lost?

(This is the example from class, another (cleaner) version on next page)




S
R
SENDER
1 2 3 4 5 6
UNA
NXT
RECEIVER
1 2 4 5
LBR
NXT
4,5 are "early arrivals"
SEG # 2 TIMES OUT
ACK 2
ACK 3
ACK 3
ACK 3
ACK 6
ACK reflects next segment expected (in this case 3), regardless of early arrivals
Update + consider early arrivals
RETRANSMIT!
3
RECEIVER
1 2 3 4 5
LBR
NXT

Where we go from here

- ACKs are a signal from the receiver about what data needs to be sent
- There are also others
    - "Advertised window": receiver communicates how much space is left in its receive buffer, sender can send no more than this
    - Duplicate ACKs: used by some congestion control algorithms as indicators of packet loss (later, not needed for project)

The rest of these notes contain:

- Some more drawings from previous years (in case you want more examples)

- Notes on the receiving side (in case you want to read ahead)

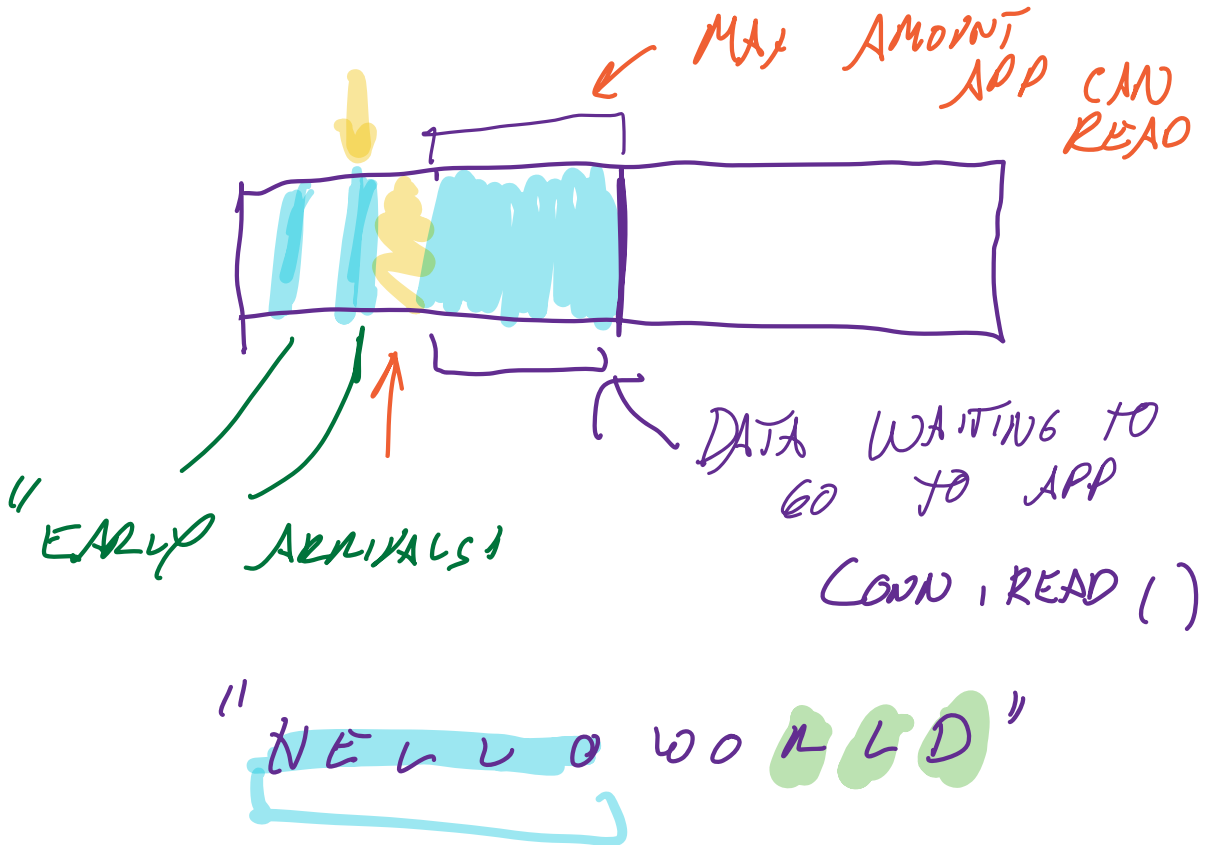
Preview: TCP sliding window (from an old lecture)

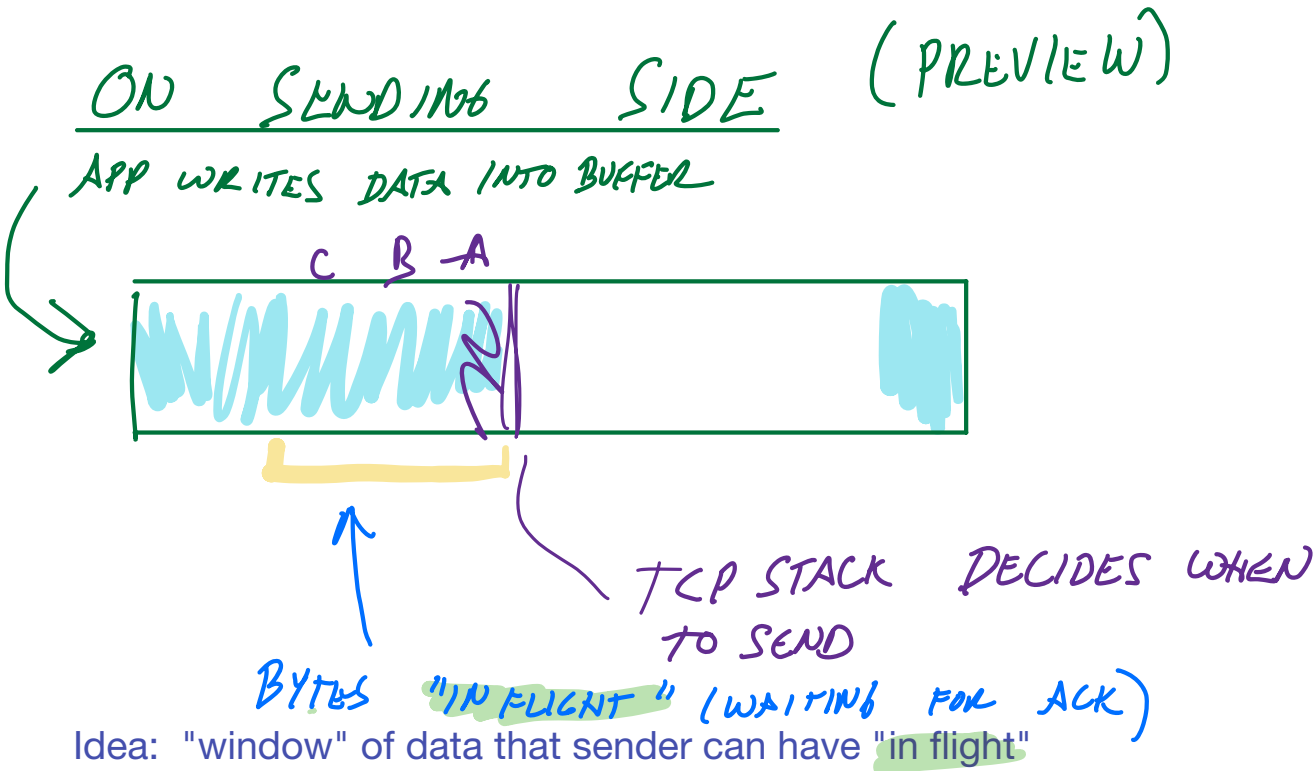
On both sender and receiver: buffer of packets

- Sending side ("Send buffer"): packets waiting to go out
  => Data can be removed once it's ACKed by receiver

- Receiving side: packets waiting to go to application
  => Need to have data up to certain point in order before it can go to application

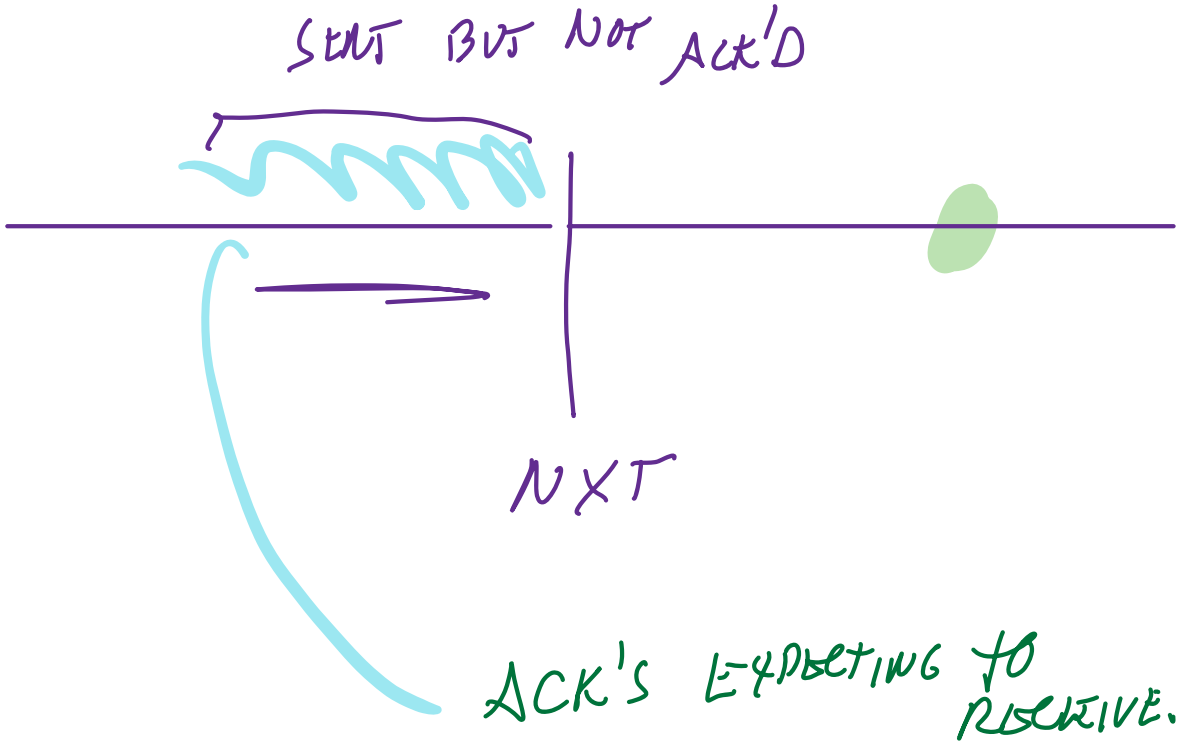
PREVIEW: RECEIVING SIDE



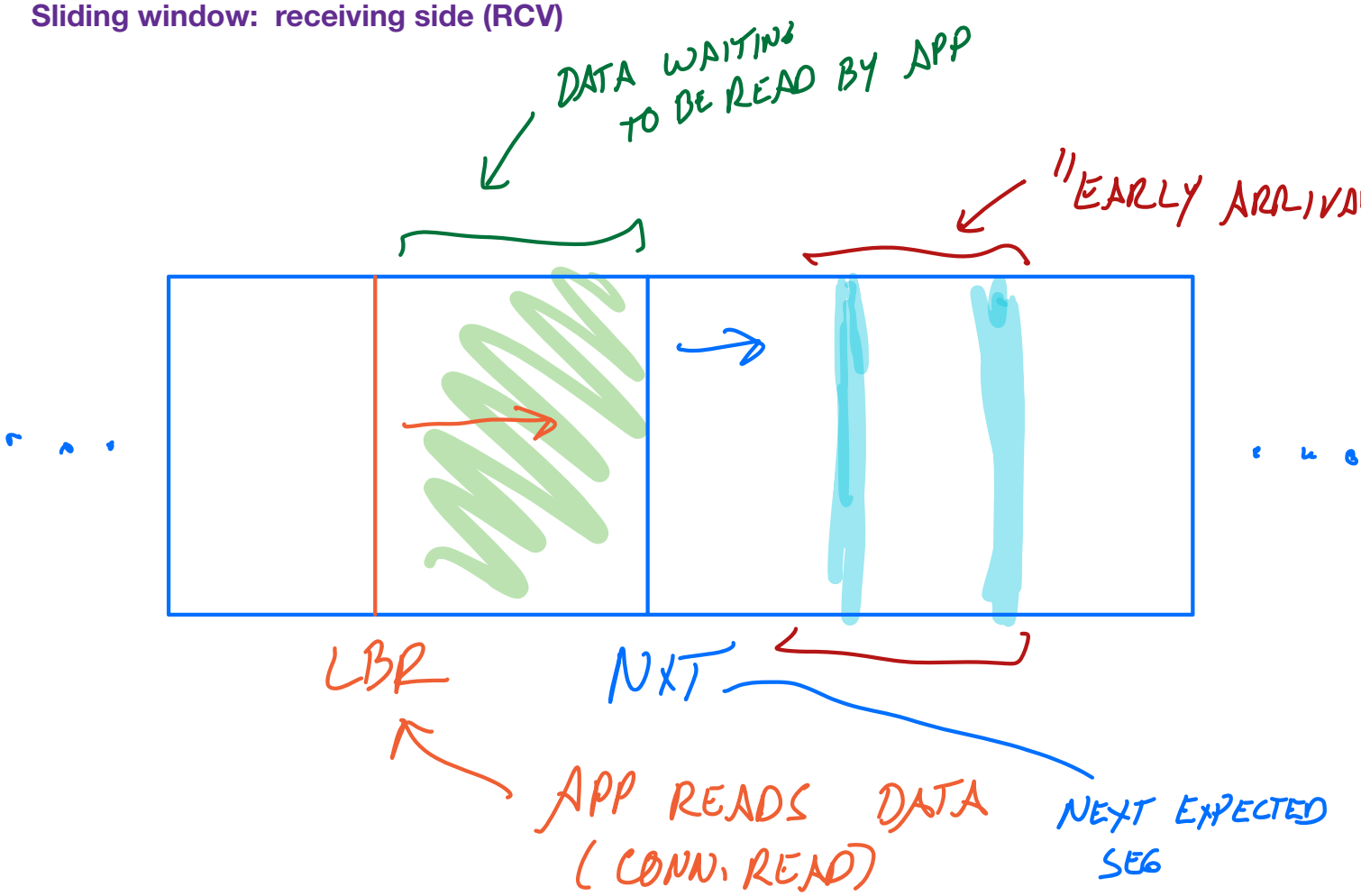


Idea: "window" of data that sender can have "in flight" waiting for receiver to ACK => as ACKs received, can send more data (up to window size)

Typically: these buffers are "circular buffer" or "ring buffers"
=> Finite chunk of memory, has "head" and a "tail"



Sliding window: receiving side (RCV)



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

"ACK clocking"