

Sliding Windows

Lecture 12

<http://www.cs.rutgers.edu/~sn624/352-S22>

Srinivas Narayana

Quick recap of concepts



UDP
Connectionless

TCP
Connection-oriented

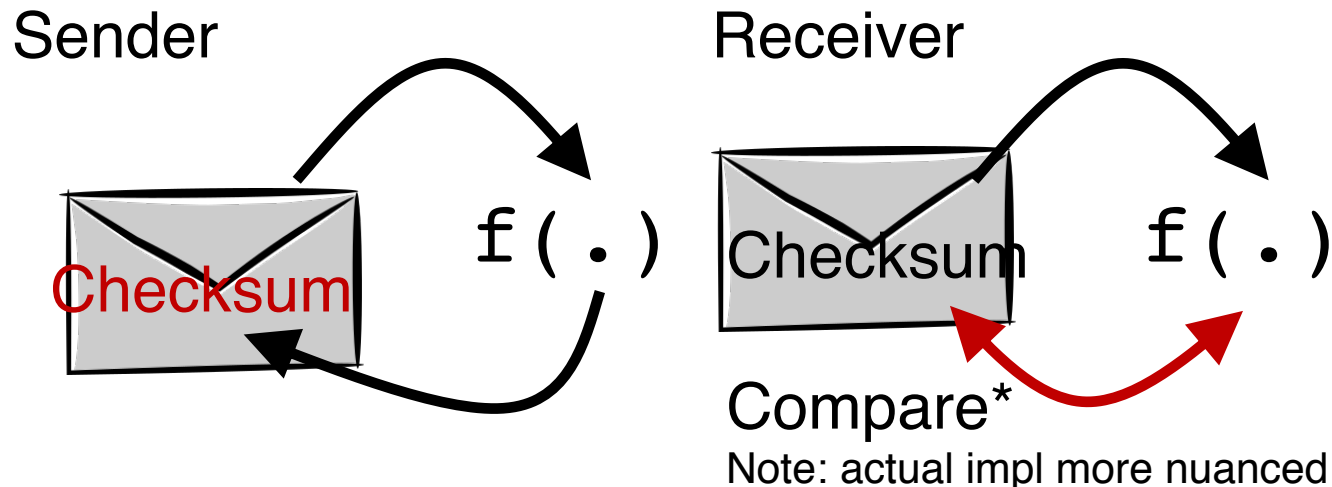
Detecting errors is insufficient.
Need to correct errors.

Also, data may simply be lost.
(checksum is also lost)

Need better mechanisms for
reliable data delivery!

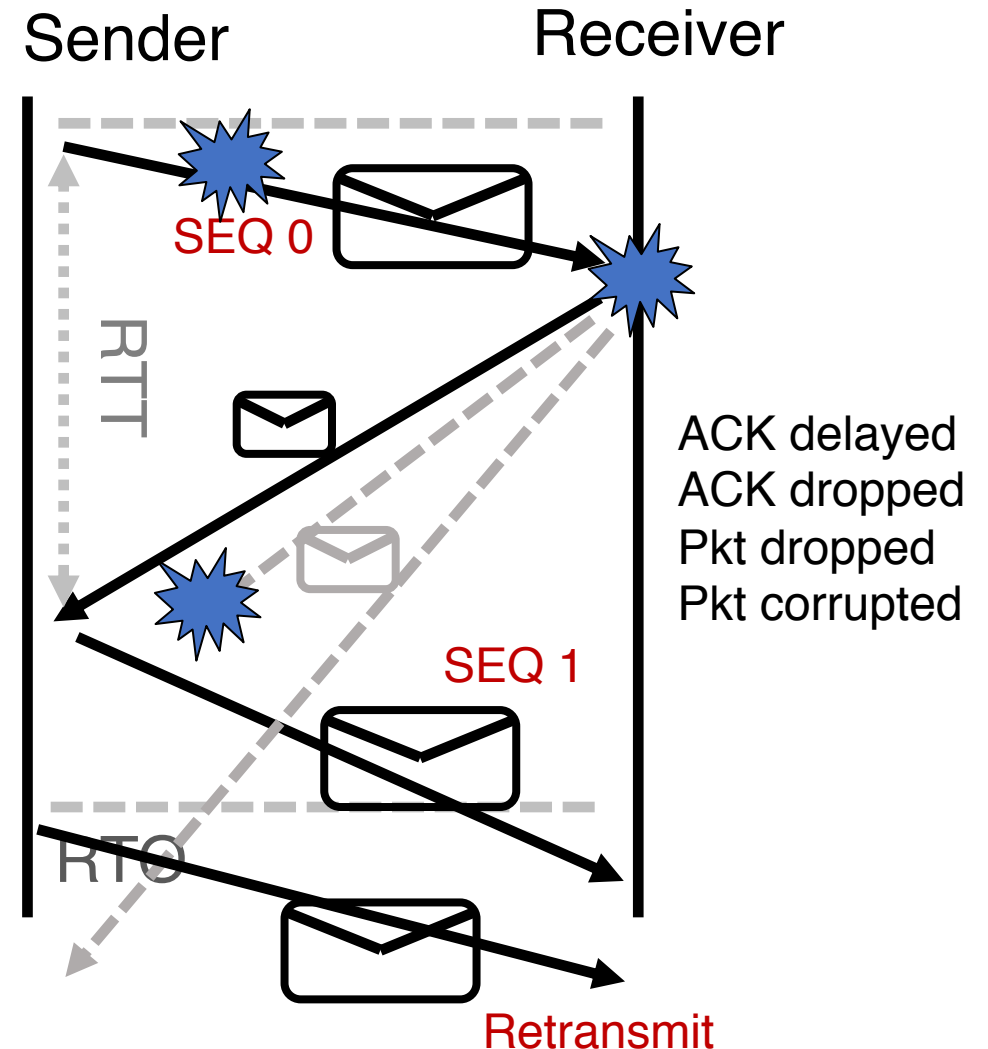
TCP uses 3 simple ideas

Error detection



Review: Stop-and-Wait Reliability

- Sender sends a **single** packet, then **waits** for an **ACK** to know the packet was successfully received. Then the sender transmits the next packet.
- If ACK is not received until a timeout (**RTO**), sender **retransmits** the packet
- Disambiguate duplicate vs. fresh packets using **sequence numbers** that change on adjacent packets

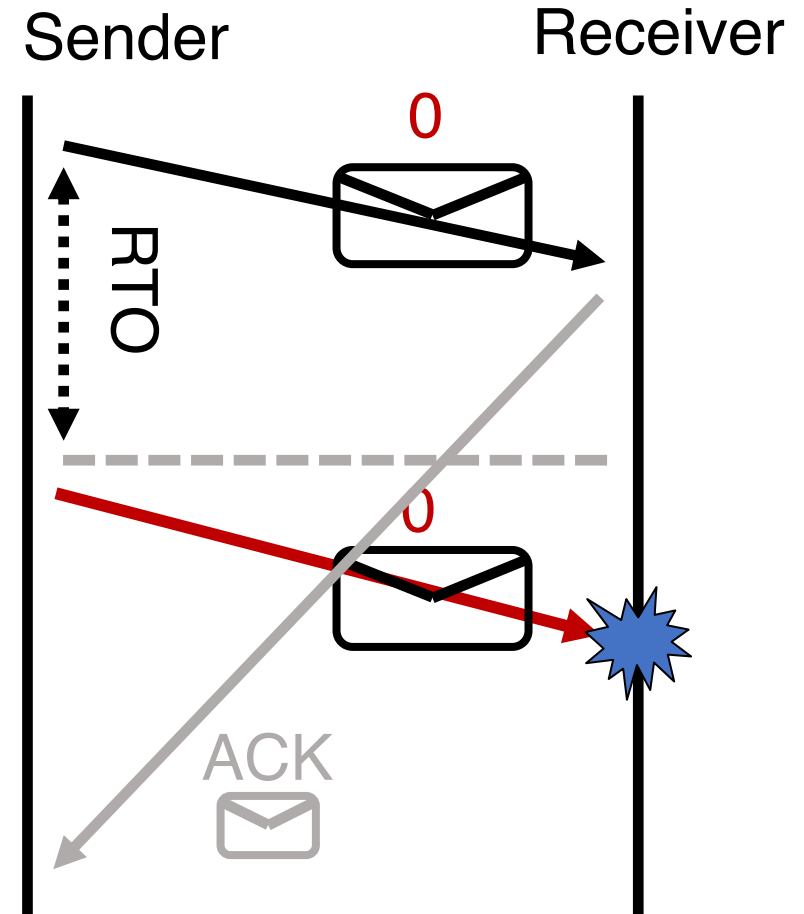


In principle, these three ideas are sufficient
to implement reliable data delivery!

Let's talk a bit more about sequence
numbers...

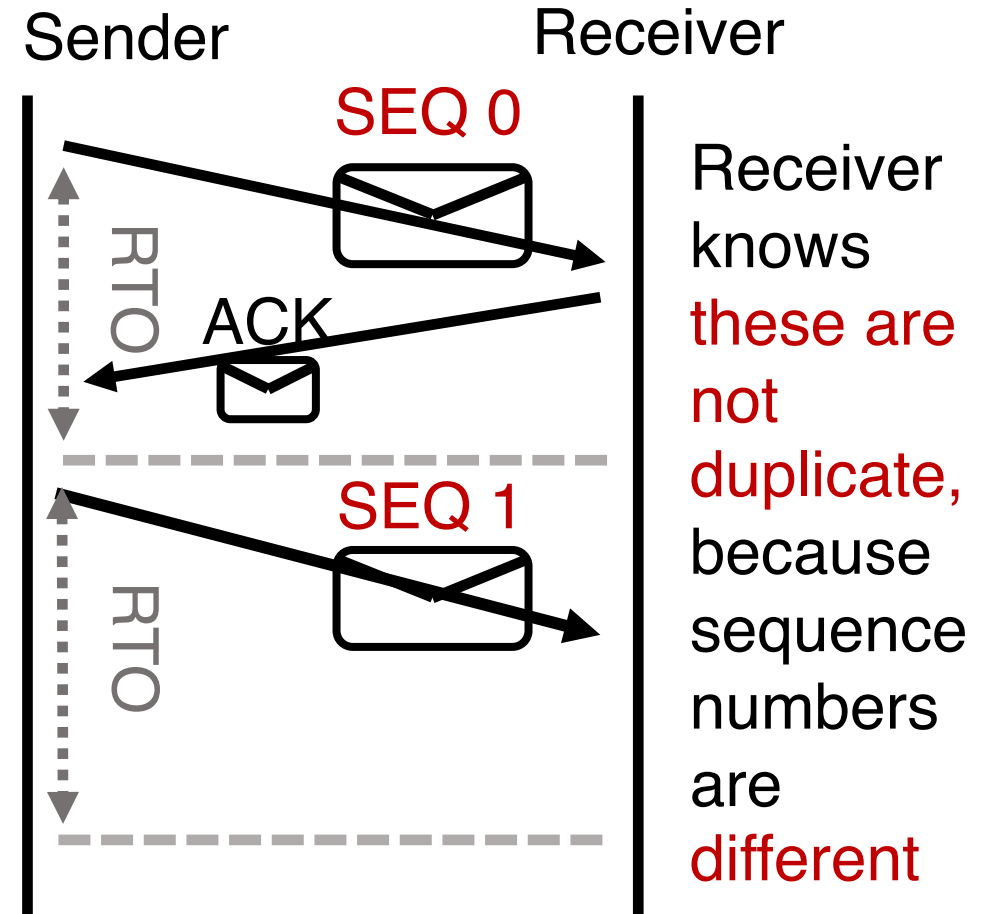
Sequence #s: Scenario 1

- A bad scenario: Suppose an ACK was delayed beyond the RTO; sender ended up retransmitting the packet.
- At the receiver: **sequence number helps disambiguate a fresh transmission from a retransmission**
 - Sequence number same as earlier: retransmission
 - Fresh sequence number: fresh data



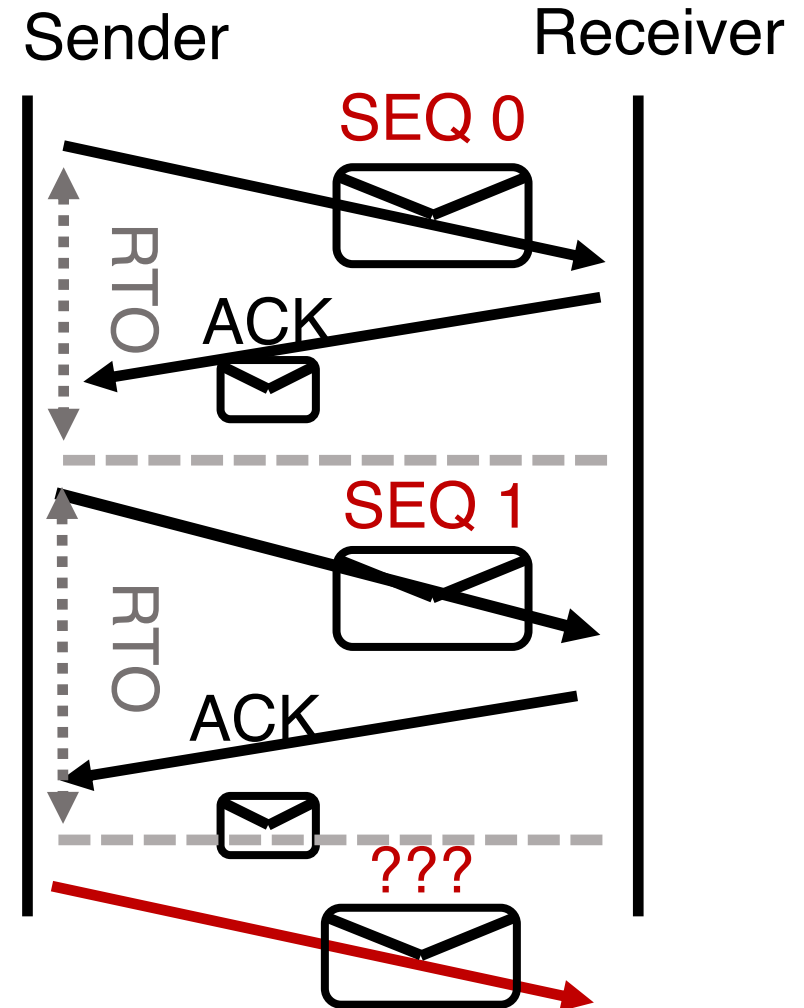
Sequence #s: Scenario 2

- A good scenario: packet successfully received and ACK returned within RTO
- Sequence numbers of successively transmitted packets are different



Q: How to number pkt sequences?

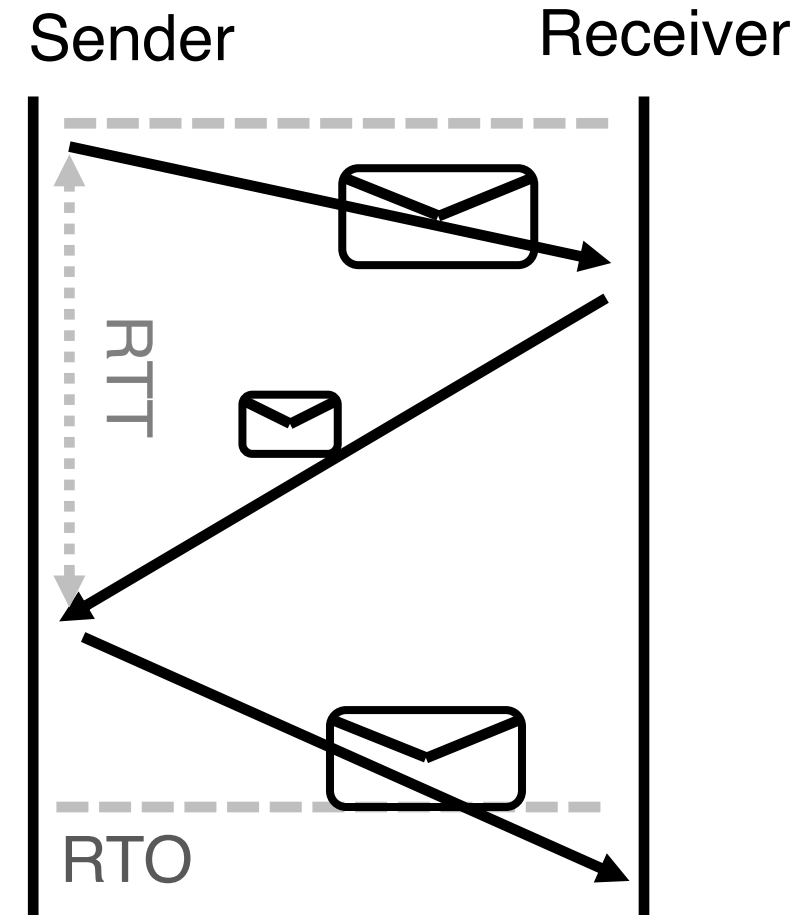
- One possibility: keep incrementing the seq #: 2, 3, ...
- Alternative: since seq # 0 was successfully ACK'ed earlier, it is OK to reuse seq #0 for next transmission.
 - No chance of confusion with a previous packet with the same sequence number
- Principle: Seq #s can be reused if older packets with those seq #s have surely been delivered



Making reliable data transfer
efficient

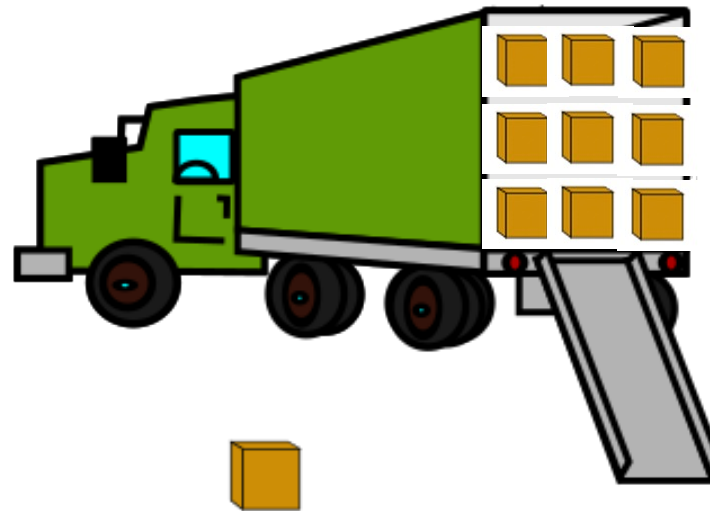
Efficiency problem with stop-and-wait

- Sender sends **one packet**, waits for an ACK (or RTO) before transmitting next one
 - Unfortunately, too slow ☹
- Suppose $RTO = RTT = 100$ milliseconds
- Packet size (bytes in 1 packet) = 12,000 bits
- Bandwidth of links from sender to receiver = 12 Mbit/s ($1\text{ M} = 10^6$)
- Rate of data transfer = data size / time



120 Kilobit/s == 1% of bw!

Sending one packet per RTT makes the data transfer rate limited by the **time** between the endpoints, rather than the **bandwidth**.



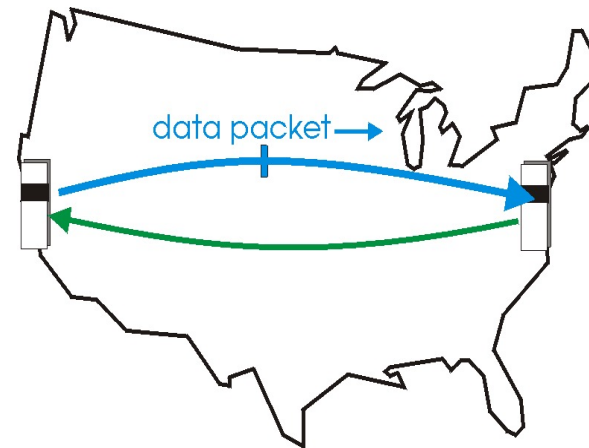
Ensure you got the (one)
box safely; make N trips

Ensure you get **N** boxes
safely; make **just 1 trip!**

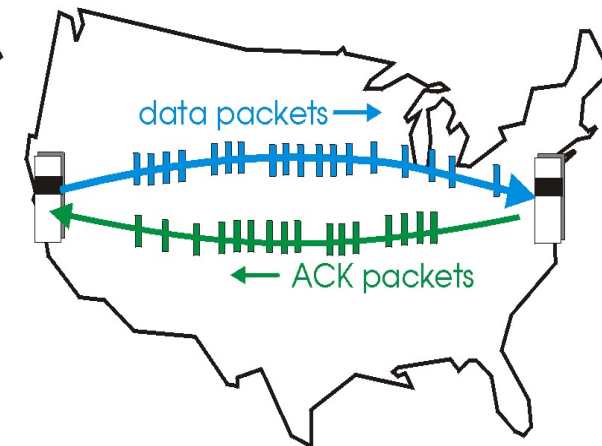
Keep many packets in flight

Pipelined reliability

- **Data in flight:** data that has been sent, but sender hasn't yet received ACKs from the receiver
 - Note: can refer to packets in flight or bytes in flight
- New packets sent at the same time as older ones still in flight
- New packets sent at the same time as ACKs are returning
- More data moving in same time!
- Improves **throughput**
 - Rate of data transfer



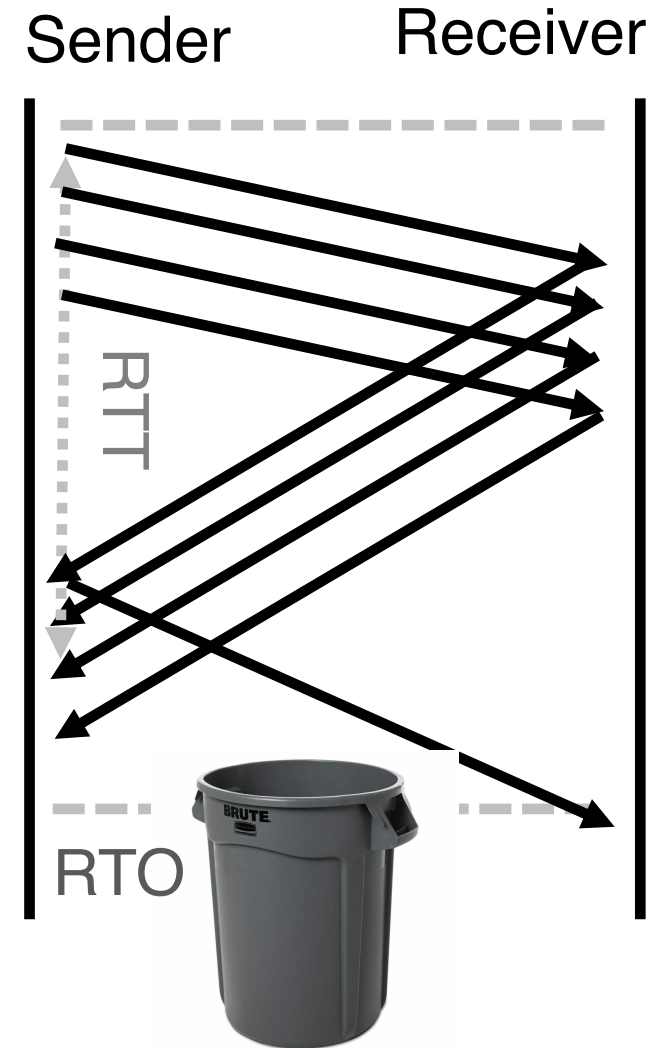
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

Pipelined reliability

- Stop and wait: send 1 packet per RTT
- Pipelined: send **N** packets per RTT
- If there are N packets in flight, throughput improves by **N times** compared to stop-and-wait!



Pipelining makes reliable data transfer efficient.

However, pipelining also makes it more complex.

Which packets are currently in flight?

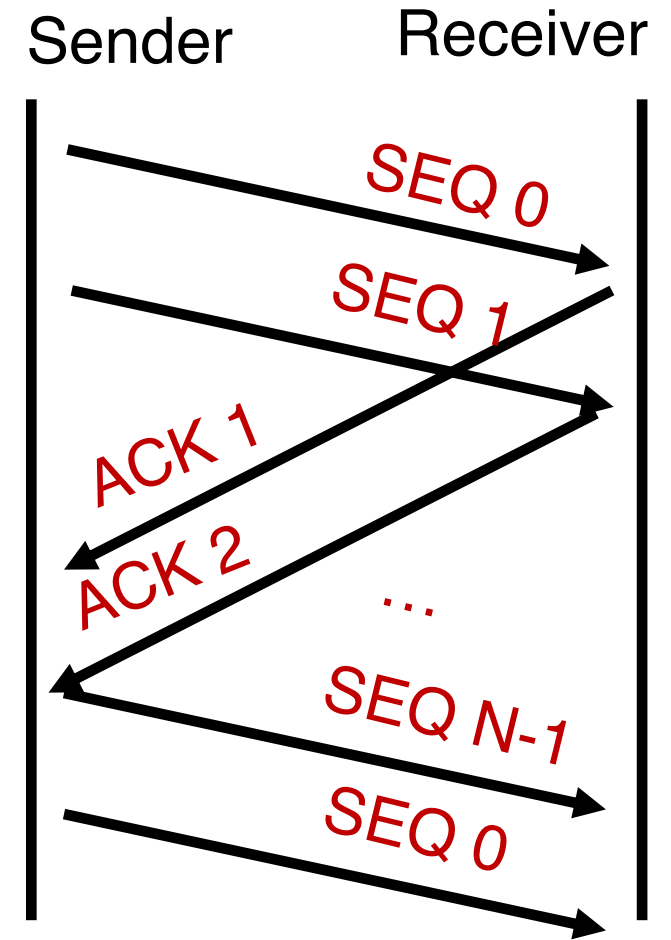
Which packets were successfully delivered?

Which packets should the sender retransmit?

Sliding Windows

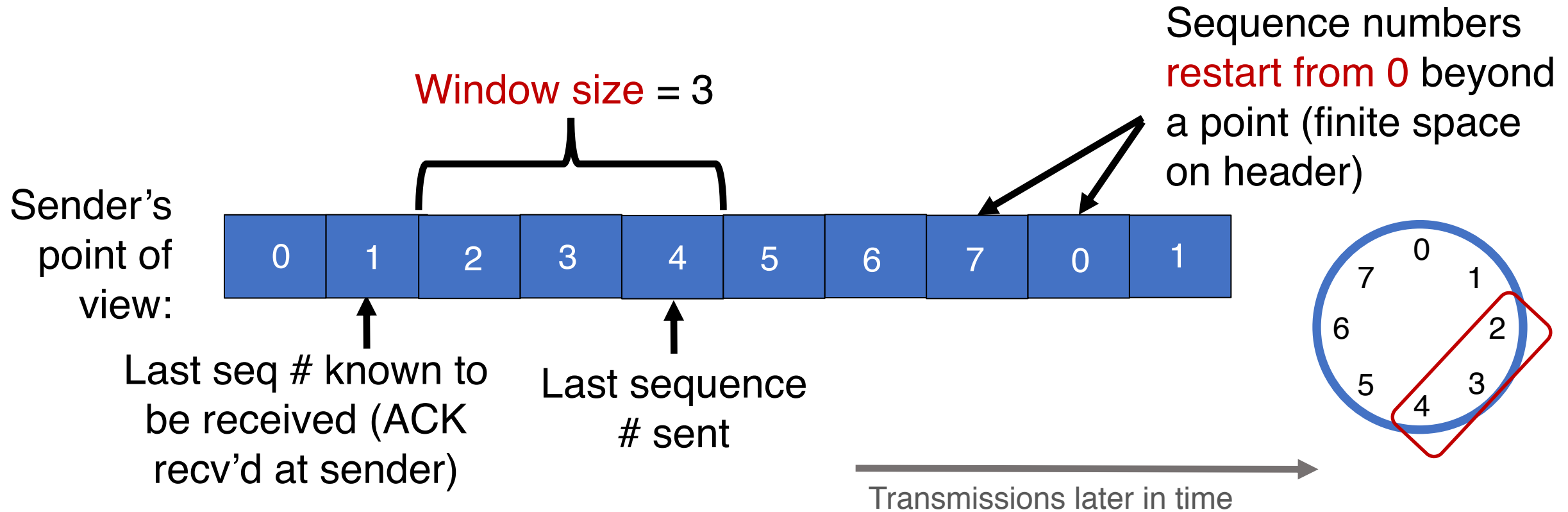
Setup

- Assume packets are labeled by sequence numbers
 - Increasing from 0, ..., N-1, then roll back to 0
- Assume ACKs indicate the sequence numbers of data that was received
 - Note: Didn't need this for stop-and-wait
- Convention: ACK#s carry the **next sequence** number expected
 - Used in TCP.



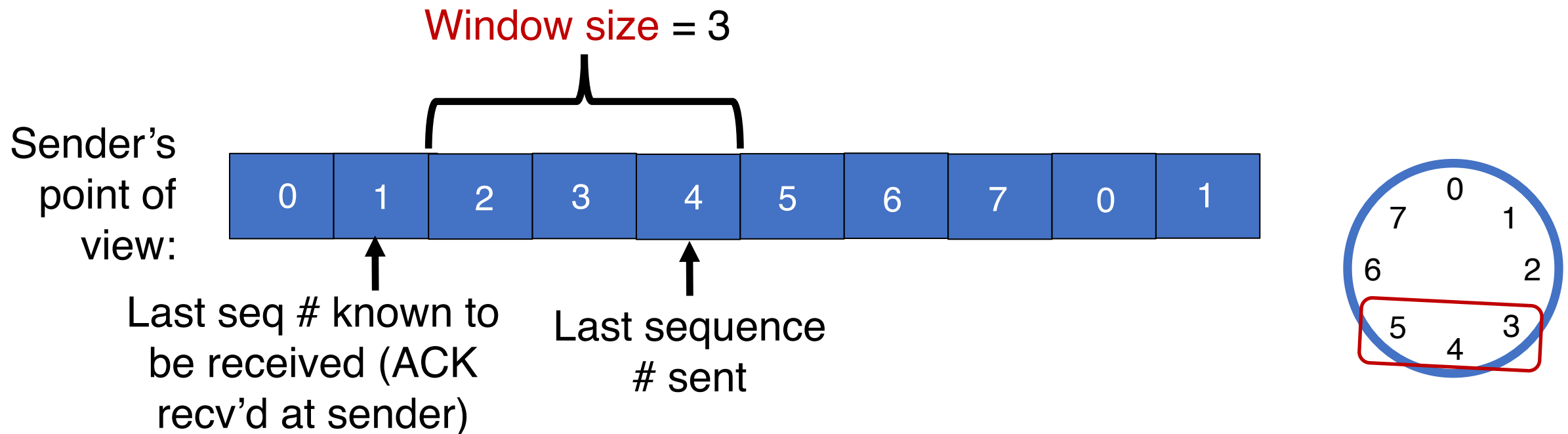
Sliding window (sender side)

- **Window**: Sequence numbers of in-flight data
- **Window size**: The amount of in-flight data (unACKed)



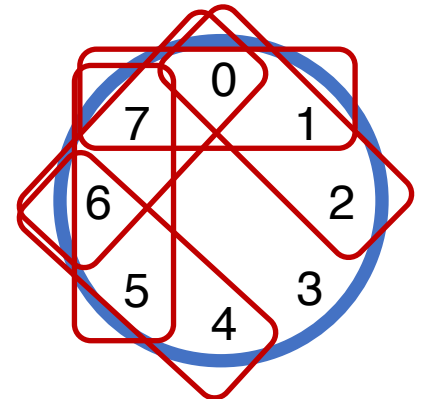
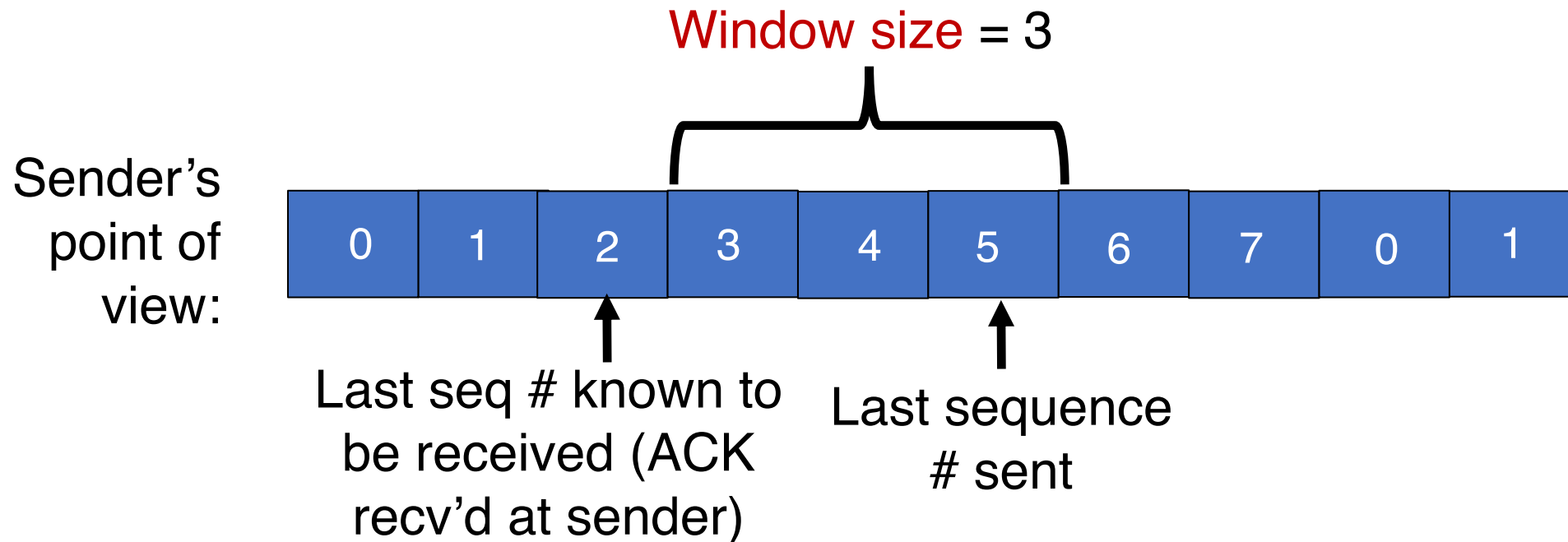
Sliding window (sender side)

- Suppose sequence number 2 is acknowledged by the receiver
 - Sender receives the ACK. Sender can transmit sequence # 5
 - The window “slides” forward



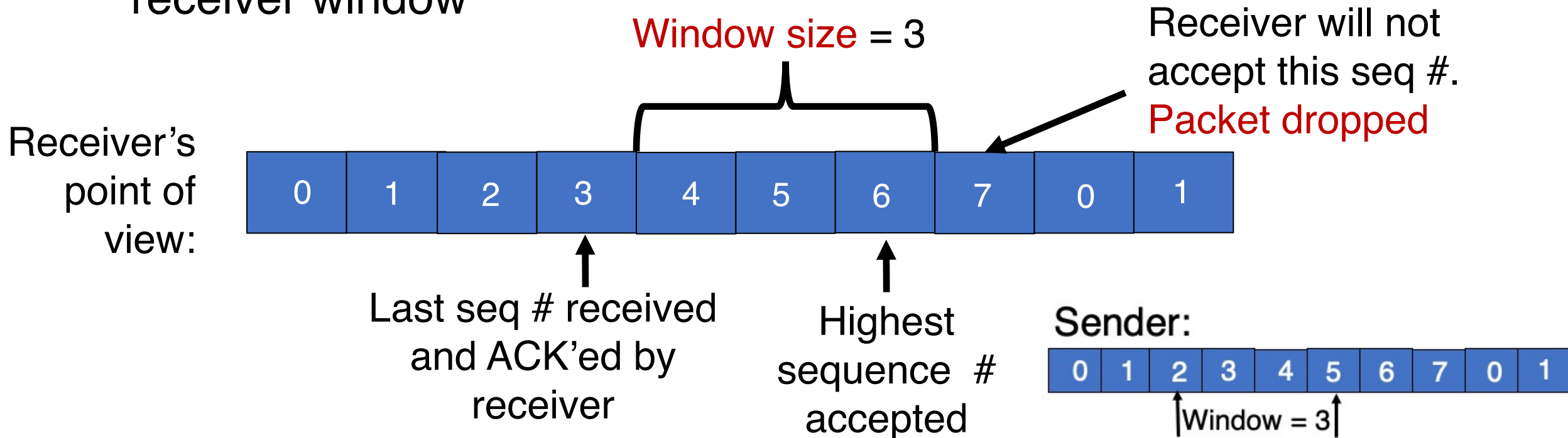
Sliding window (sender side)

- Suppose sequence number 2 is acknowledged by the receiver
 - Sender receives the ACK. Sender can transmit sequence # 5
 - The window “slides” forward



Sliding window (**receiver side**)

- Window of in-flight packets can look different between sender and the receiver
- Receiver only accepts sequence #s allowed by the current receiver window

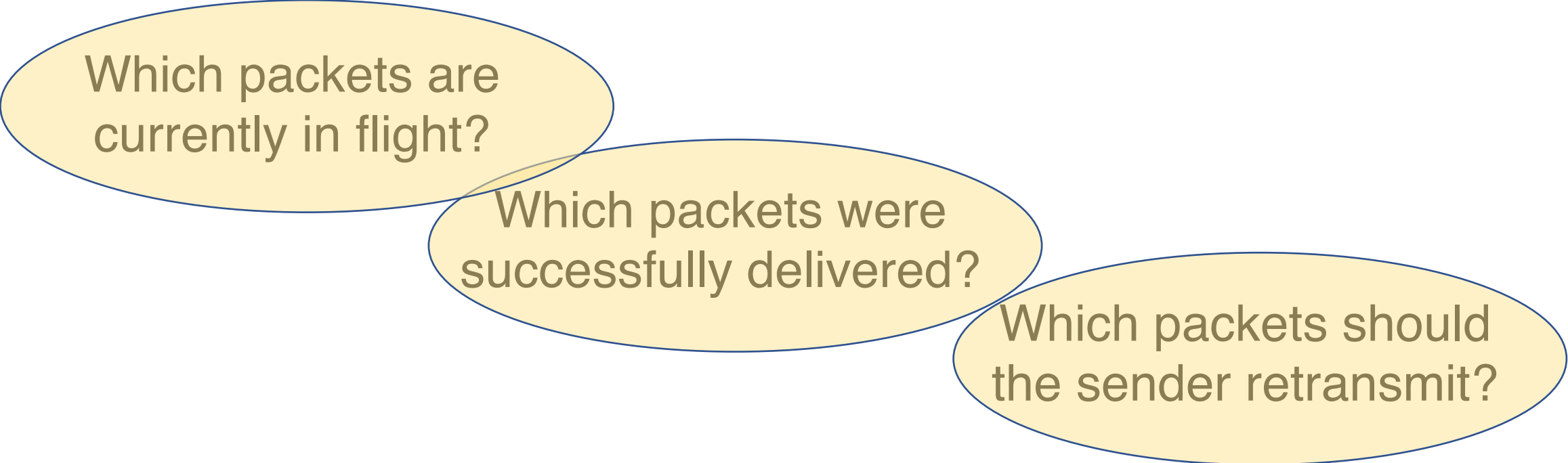


Summary of sliding windows

- Sender and receiver can keep several packets of in-flight data
 - Book-keep the sequence numbers using the window
- Windows **slide forward** as packets are ACKed (at receiver) and ACKs are received (at sender)
- Common case: Improve throughput by sending and ACKing more packets in the same duration

Pipelining makes reliable data transfer efficient.

However, pipelining also makes it more complex.



Which packets are currently in flight?

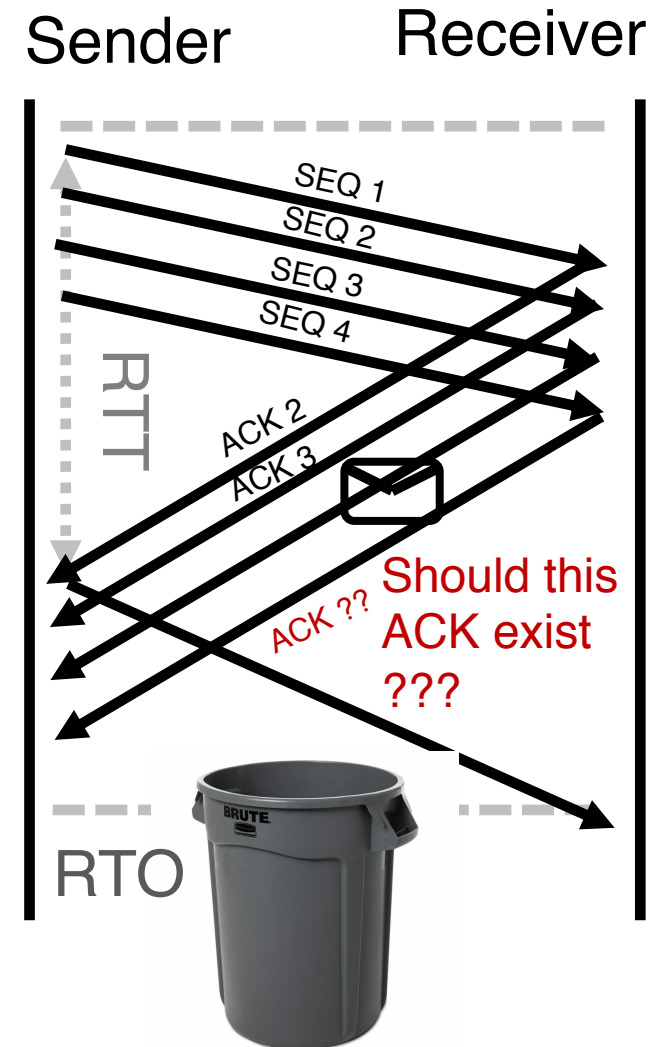
Which packets were successfully delivered?

Which packets should the sender retransmit?

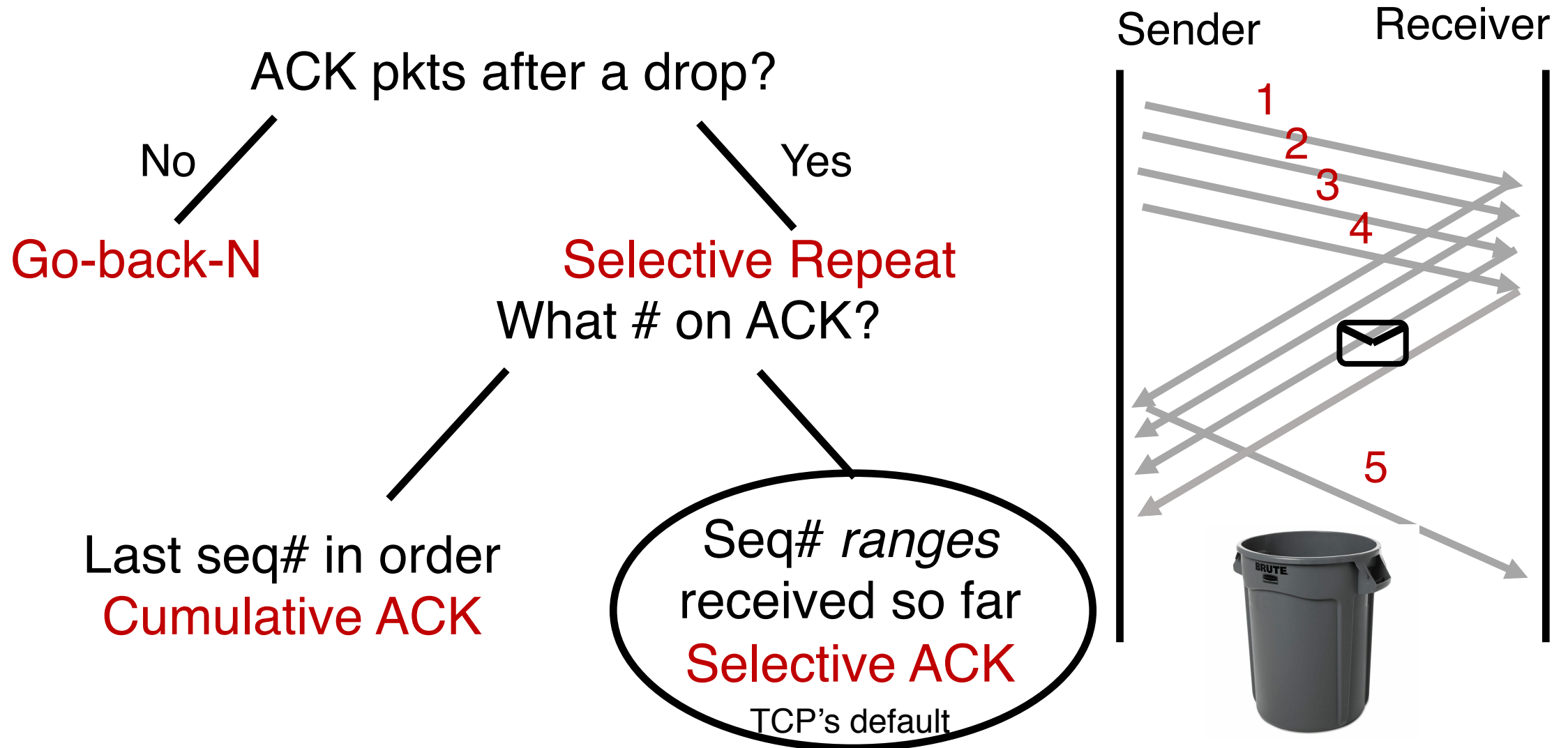
Which packets to retransmit?

How to identify dropped packets?

- Suppose 4 packets sent, but 1 dropped. How does sender know which one(s) dropped?
- Recall: Receiver writes **sequence numbers** on the ACK indicating successful reception
- Key idea: Sender can infer which data was received successfully using the ACK #s!
 - Hence, sender can know which data to retransmit
- Q1: Should receivers ACK subsequent packets upon detecting data loss?
- Q2: If so, what sequence number should receiver put on the ACK?



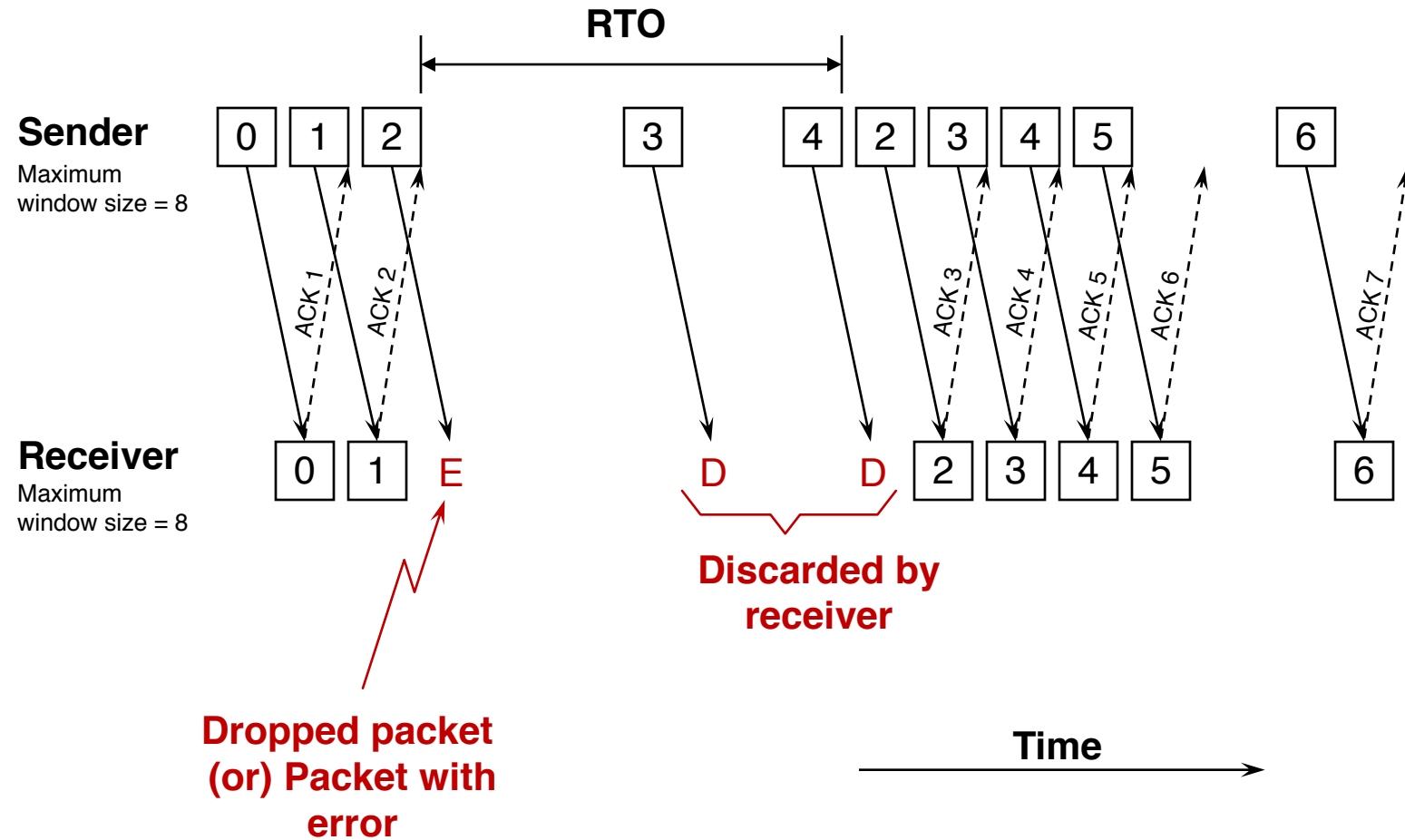
Receiver strategies upon packet loss



Sliding Window with Go Back N

- When the receiver notices missing data:
- It simply **discards** all data with greater sequence numbers
 - i.e.: the receiver will send no further ACKs
- The sender will eventually time out (RTO) and retransmit all the data in its sending window
- Subtle: conceptually, **separate timer per byte** to infer RTO

Go back N



Go back N

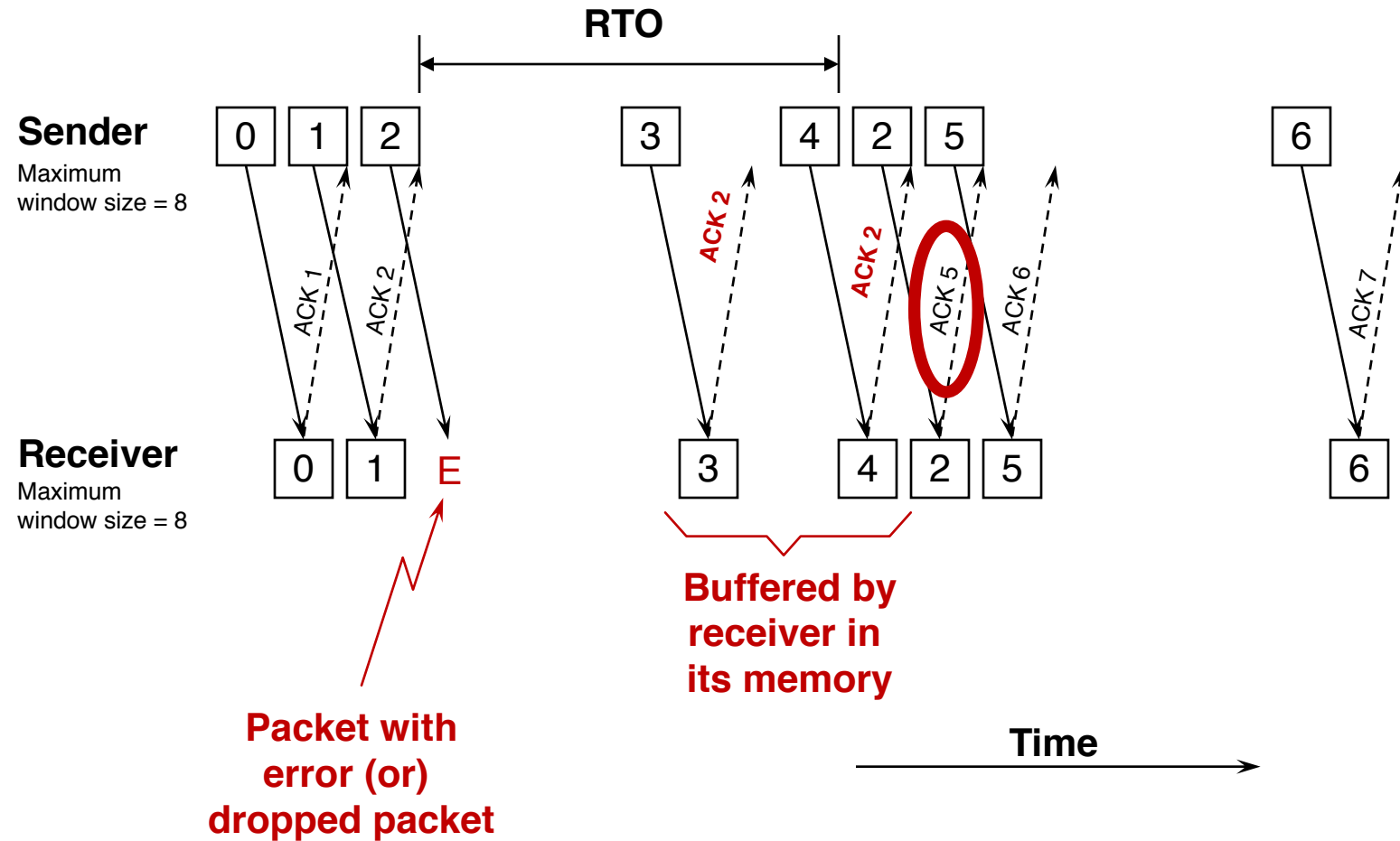
- Go Back N can recover from erroneous or missing packets.
- But it is wasteful.
- If there are errors, the sender will spend time and network bandwidth retransmitting **data the receiver has already seen.**

Selective repeat with cumulative ACK

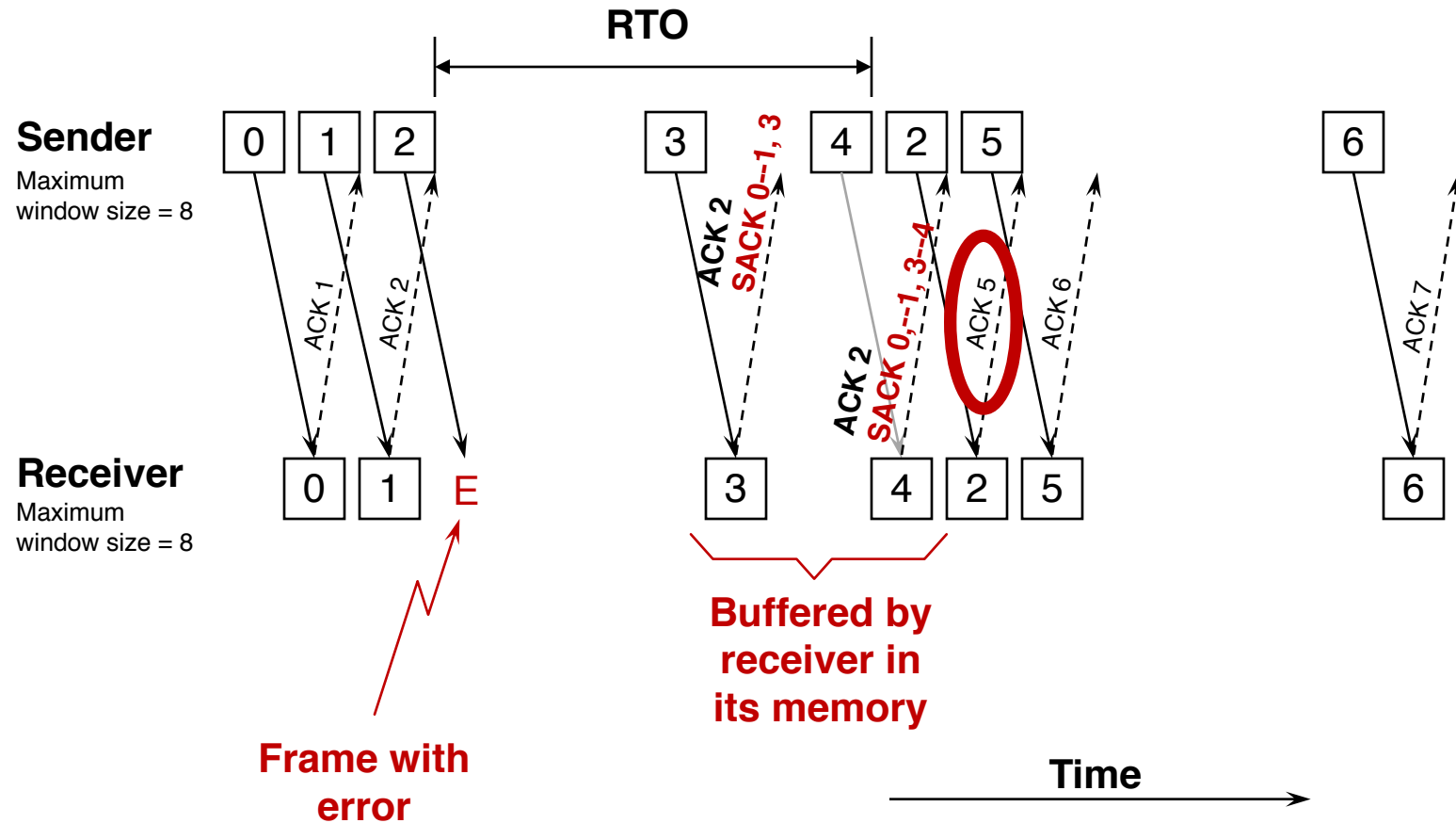
Idea: sender should only retransmit dropped/corrupted data.

- The receiver **stores** all the correct frames that arrive following the bad one. (Note that the receiver requires **memory to hold data** for each sequence number in the receiver window.)
- When the receiver notices a skipped sequence number, it keeps acknowledging the **first in-order sequence number it wants to receive**. This is termed **cumulative ACK**.
- When the sender times out waiting for an acknowledgement, it **just retransmits the first unacknowledged data**, not all its successors.
- Recall that RTO applies independently to each sequence #

Selective repeat with cumulative ACK



Selective repeat with selective ACK



TCP: Cumulative & Selective ACKs

- Sender retransmits the seq #s it thinks aren't received successfully yet
- Pros & cons: selective vs. cumulative ACKs
 - Precision of info available to sender
 - Redundancy of retransmissions
 - Packet header space
 - Complexity (and bugs) in transport software
- On modern Linux, TCP uses selective ACKs by default

