

# CSE160: Computer Networks

## Lecture #13 – Sliding Windows and ARQ

**2020-10-08**



**Professor  
Alberto E. Cerpa**



# Last Time

---

- We finished up the Network layer
  - Internetworks (IP)
  - Intra-domain routing (DV/RIP, LS/OSPF)
  - Inter-domain routing (BGP and Ases)
  - IP Addressing (CIDR, Subnets and Aggregation)
- It was all about routing: how to provide end-to-end delivery of packets

Application
Presentation
Session
Transport
Network
Data Link
Physical



# This Time

---

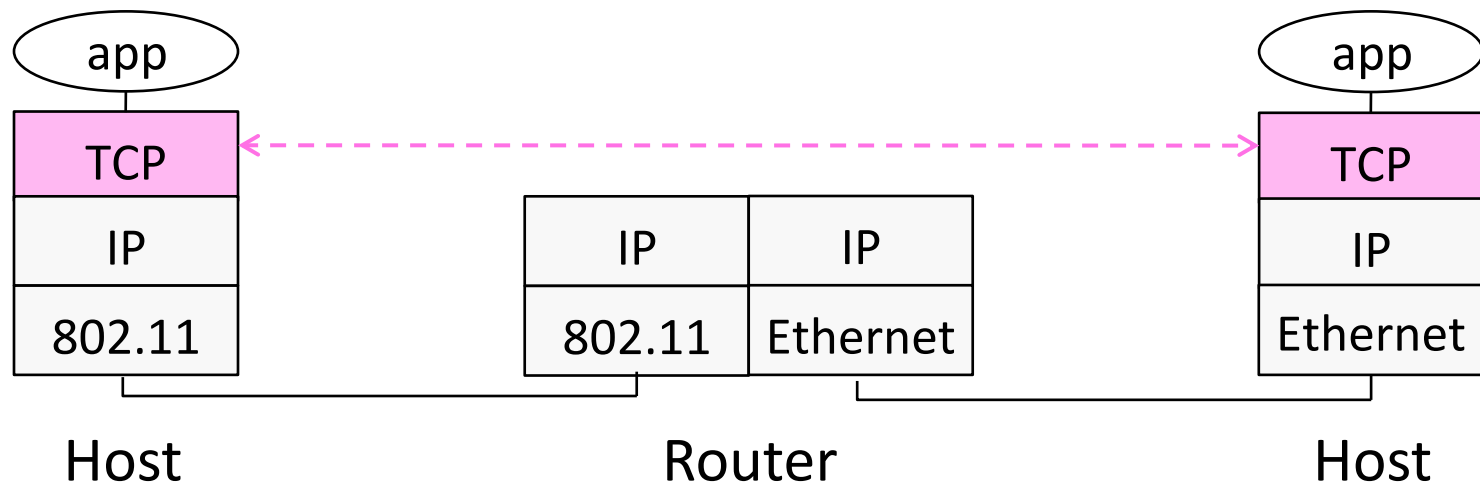
- We begin on the Transport layer (albeit this is also in the Data Link layer)
- Focus
  - How do we send information reliably?
- Topics
  - The Transport layer
  - Acknowledgements and retransmissions (ARQ)
  - Sliding windows
  - Flow Control

Application
Presentation
Session
Transport
Network
Data Link
Physical



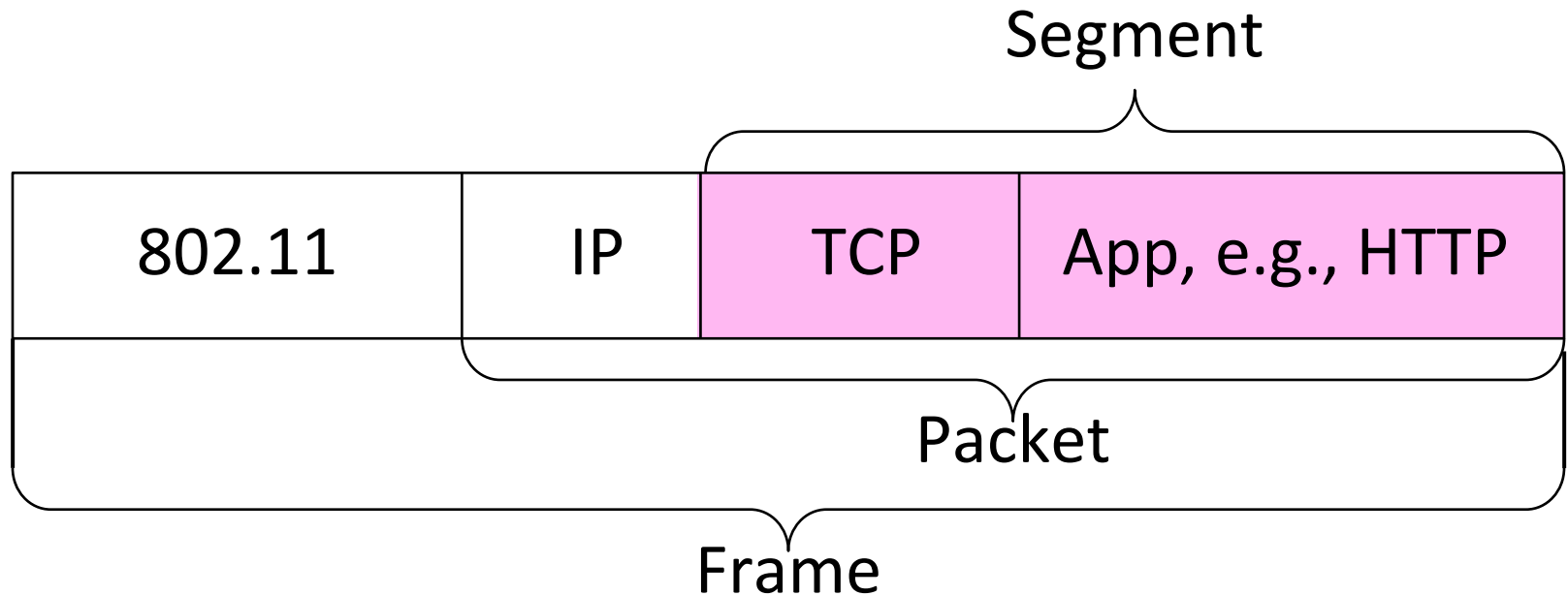
# The Transport Layer

- Builds on the services of the Network layer
- Transport layer provides end-to-end connectivity across the network
- Communication bw processes running on hosts
  - Naming/Addressing
- Stronger guarantees of message delivery
  - Reliability



# Transport Layering

- Segments carry application data across the network
- Segments are carried within packets within frames

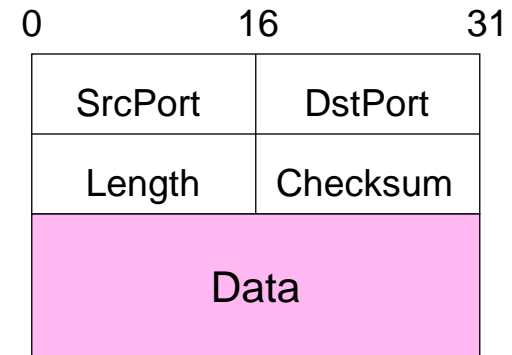


# Internet Transport Protocols

---

- UDP

- Datagram abstraction between processes
- With error detection



- TCP

- Bytestream abstraction between processes
- With **reliability**
- Plus **congestion control** (later!)



# Comparison of Internet Transports

---

- TCP is full-featured, UDP is a glorified packet

<b>TCP (Streams)</b>	<b>UDP (Datagrams)</b>
Connections	Datagrams
Bytes are delivered once, reliably, and in order	Messages may be lost, reordered, duplicated
Arbitrary length content	Limited message size
Flow control matches sender to receiver	Can send regardless of receiver state
Congestion control matches sender to network	Can send regardless of network state



# Example – Common Properties

---

## TCP

- Connection-oriented
- Multiple processes
- Reliable byte-stream delivery
  - In-order delivery
  - Single delivery
  - Arbitrarily long messages
- Synchronization
- Flow control
- Reliable delivery

## IP

- Datagram oriented
- Lost packets
- Reordered packets
- Duplicate packets
- Limited size packets





# What does it mean to be “reliable”

---

- How can a sender “know” the sent packet was received?
  - Sender receives an acknowledgement
- How can a receiver “know” a received packet is correct and not a duplicate?
  - Sender includes sequence number, checksum
- Do sender and receiver need to come to consensus on what is sent and received?
  - When is it OK for the receiver’s TCP/IP stack to deliver the data to the application?



# Automatic Repeat ReQuest (ARQ)

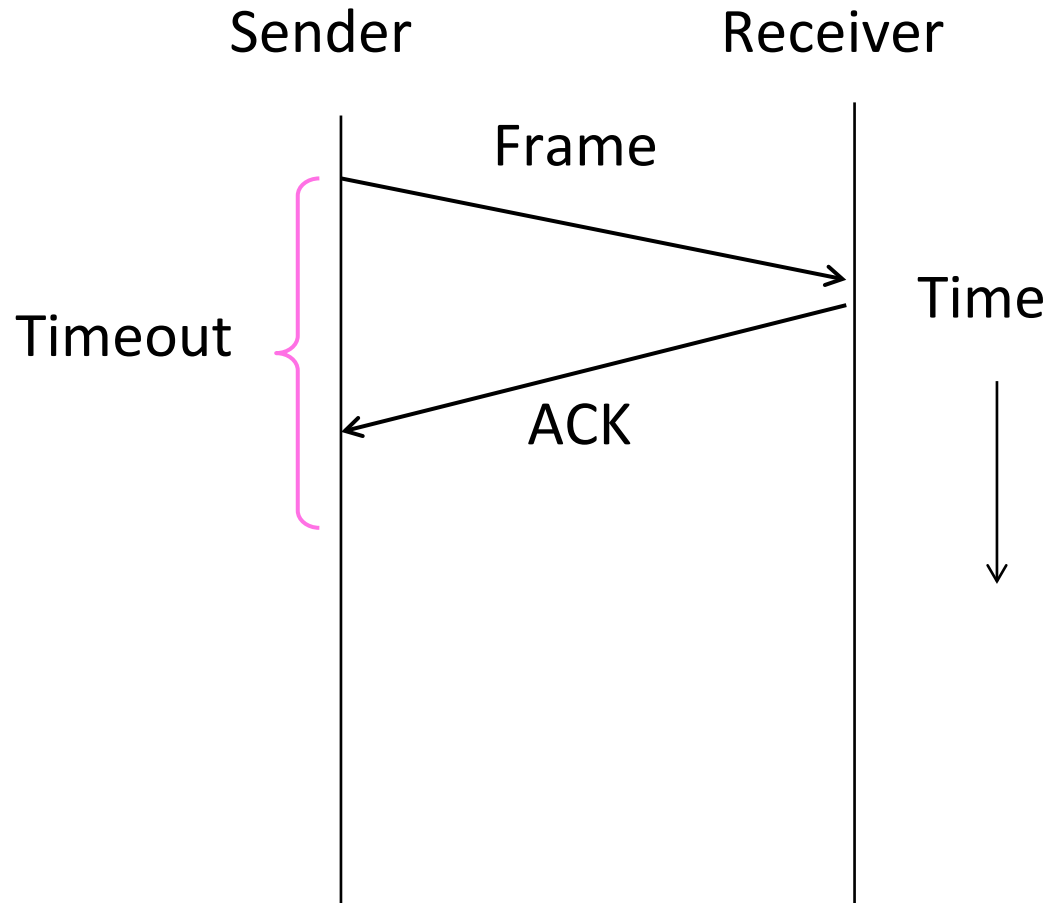
---

- ARQ often used when errors are common or must be corrected
  - E.g., WiFi (Layer 2), and TCP (Layer 4, later)
  - Remember the end-2-end principle
    - Reliability may be included at multiple layers (performance)
- Rules at sender and receiver:
  - Receiver automatically acknowledges correct frames with an ACK
  - Sender automatically resends after a timeout, until an ACK is received



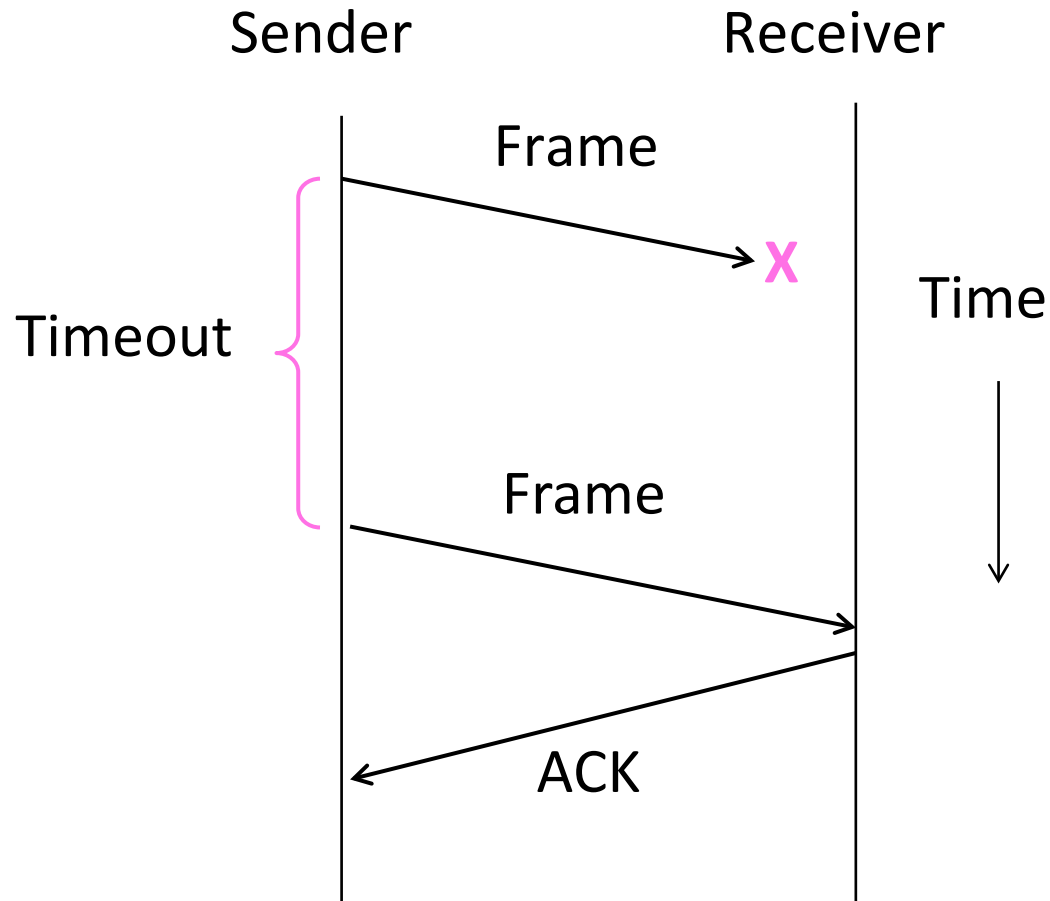
# ARQ (2)

- Normal operation (no loss)



# ARQ (3)

- Loss and retransmission



# So What's Tricky About ARQ?

---

- Two non-trivial issues:
  - How long to set the timeout?
  - How to avoid accepting duplicate frames as new frames?
- Want performance in the common case and correctness always



# Timeouts

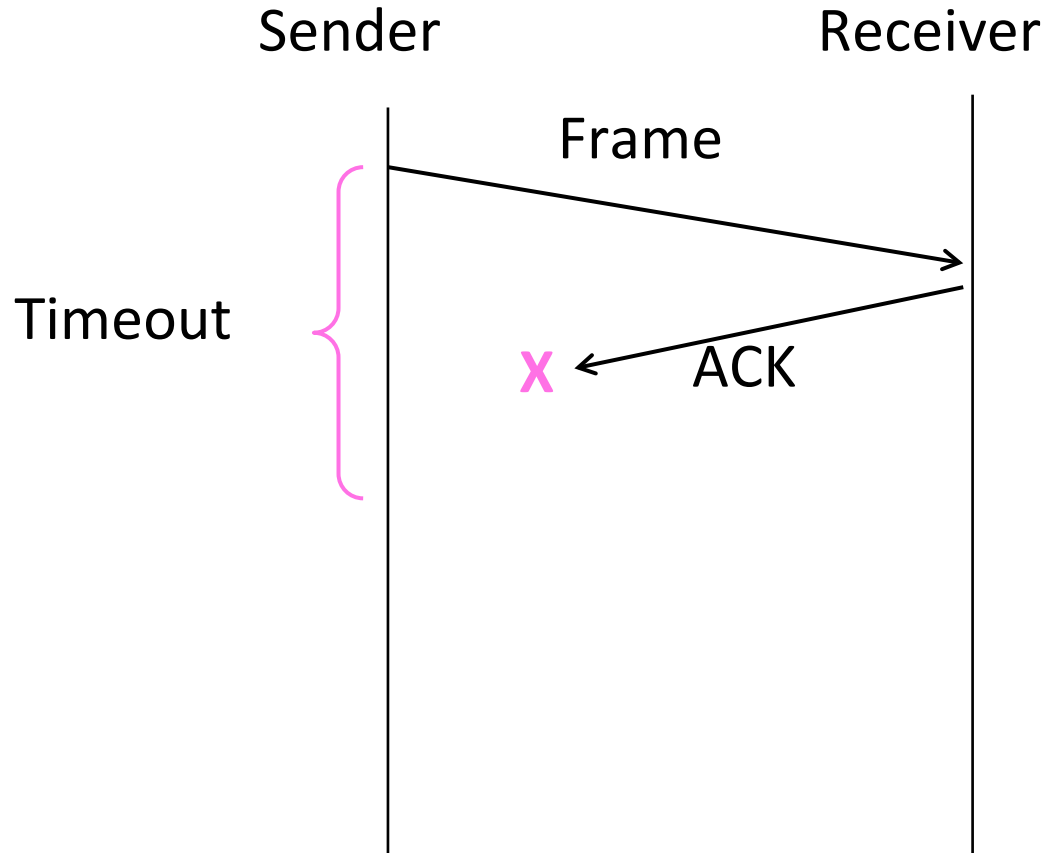
---

- Timeout should be:
  - Not too big (link goes idle)
  - Not too small (spurious resend)
- Fairly easy on a LAN
  - Clear worst case, little variation
- Fairly difficult over the Internet
  - Much variation, no obvious bound
  - We'll revisit this with TCP timeouts (later)



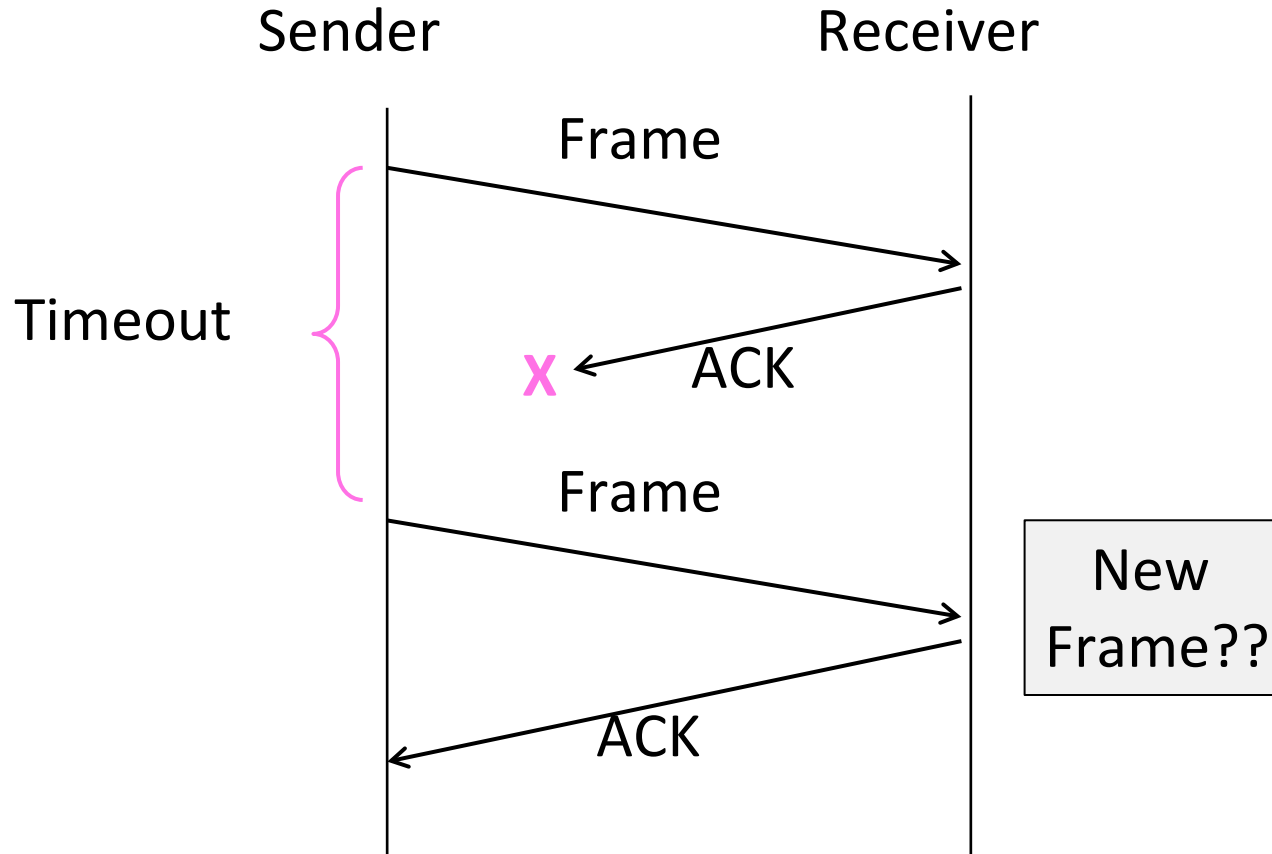
# Duplicates

- What happens if an ACK is lost?



# Duplicates (2)

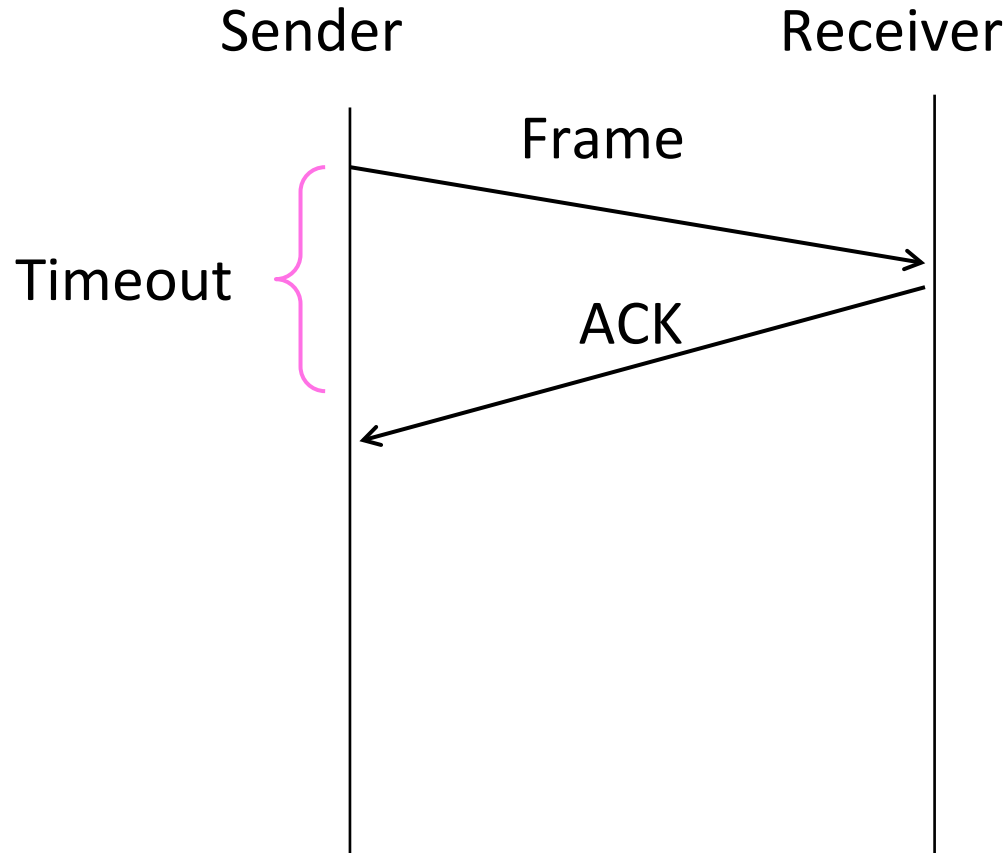
- What happens if an ACK is lost?





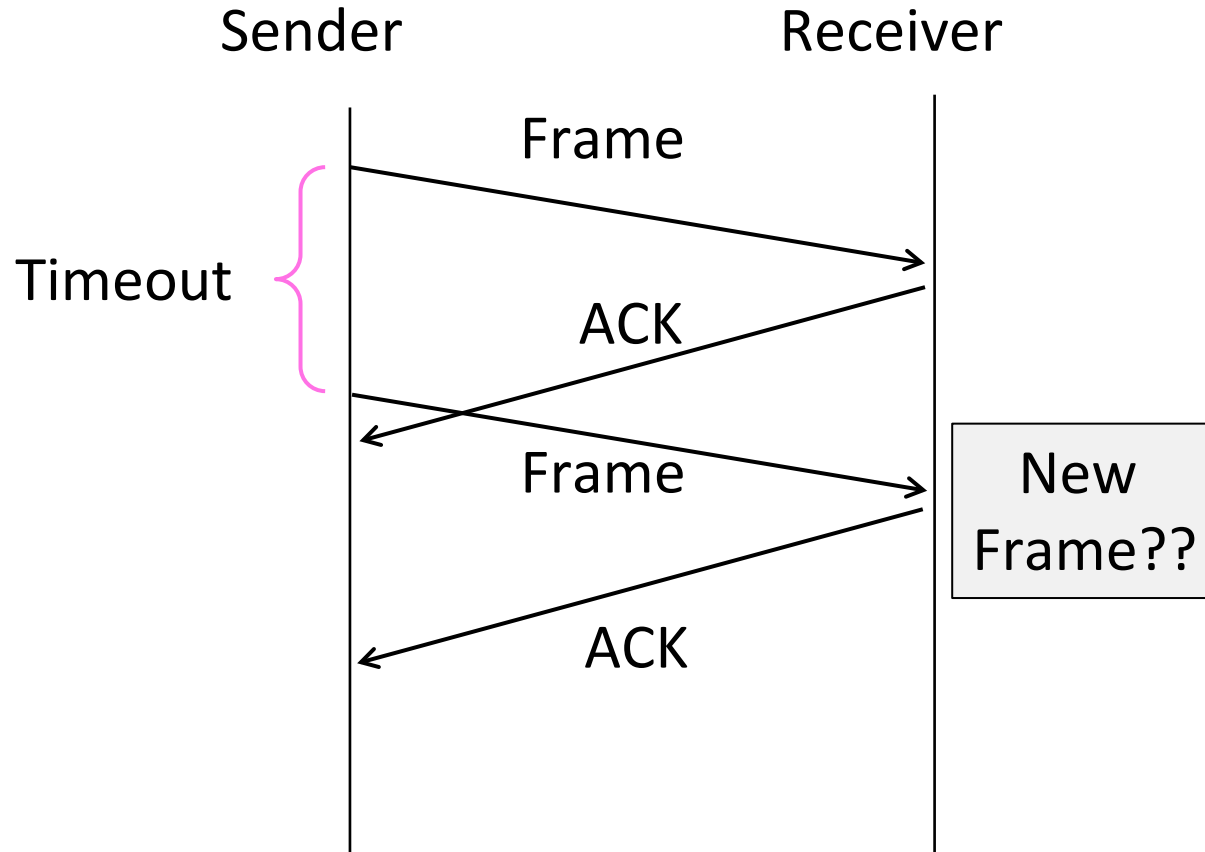
# Duplicates (3)

- Or the timeout is early?



# Duplicates (4)

- Or the timeout is early?



# Sequence Numbers

---

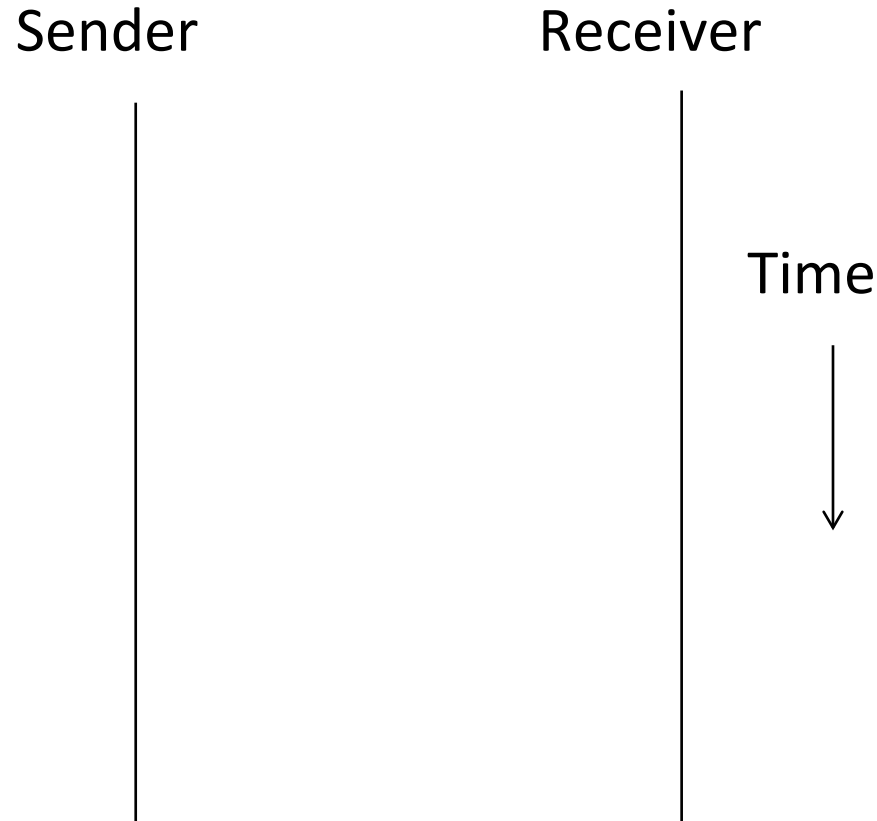
- Frames and ACKs must both carry sequence numbers for correctness
- To distinguish the current frame from the next one, a single bit (two numbers) is sufficient
  - Called Stop-and-Wait



# Stop-and-Wait

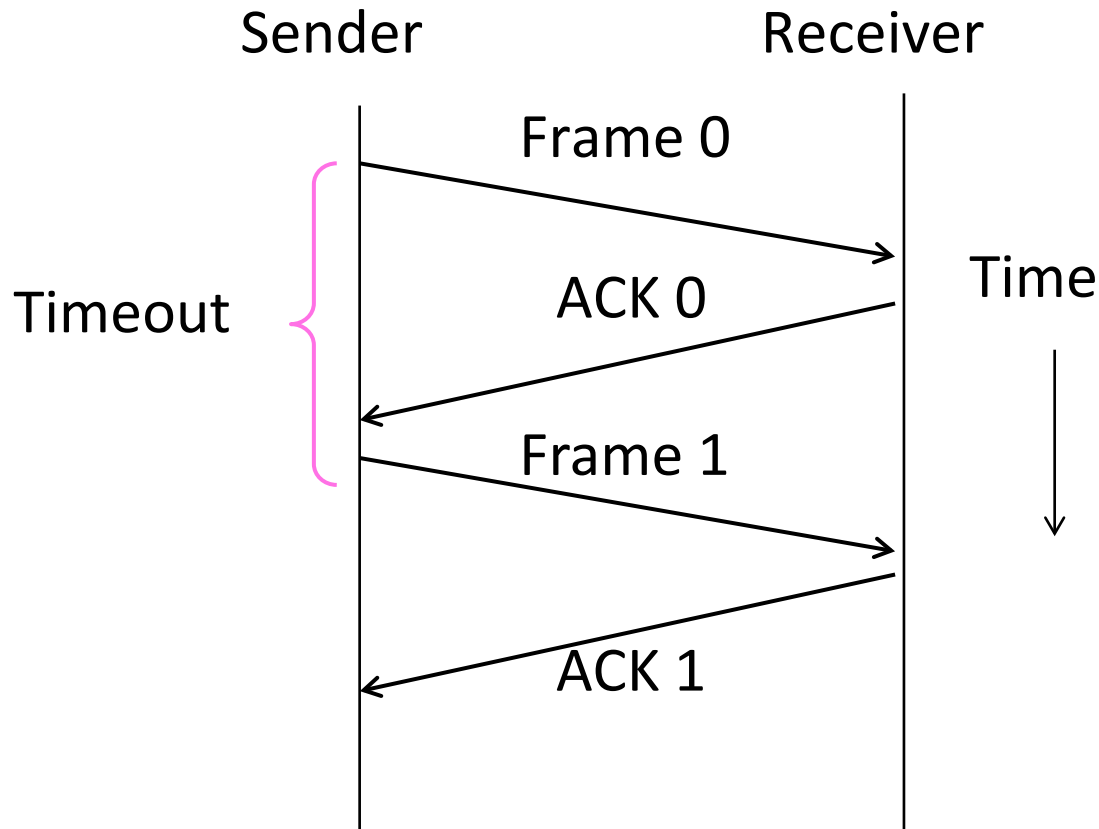
---

- In the normal case:



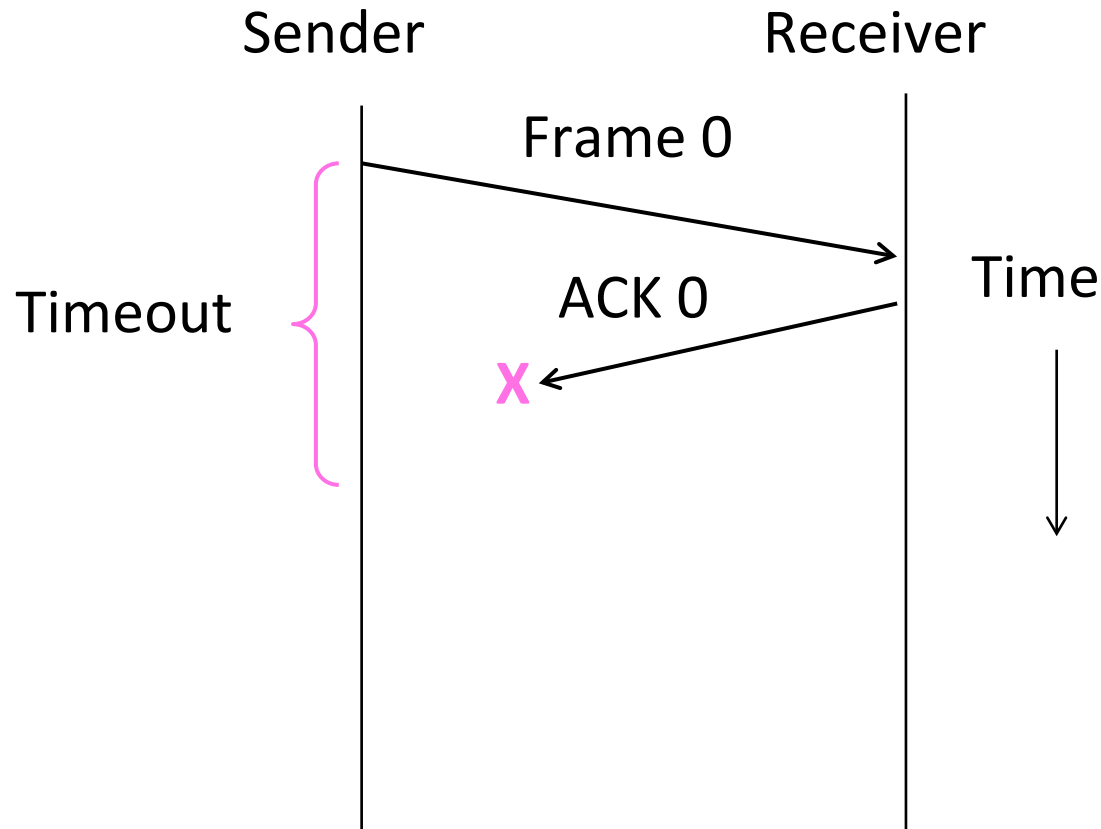
# Stop-and-Wait (2)

- In the normal case:



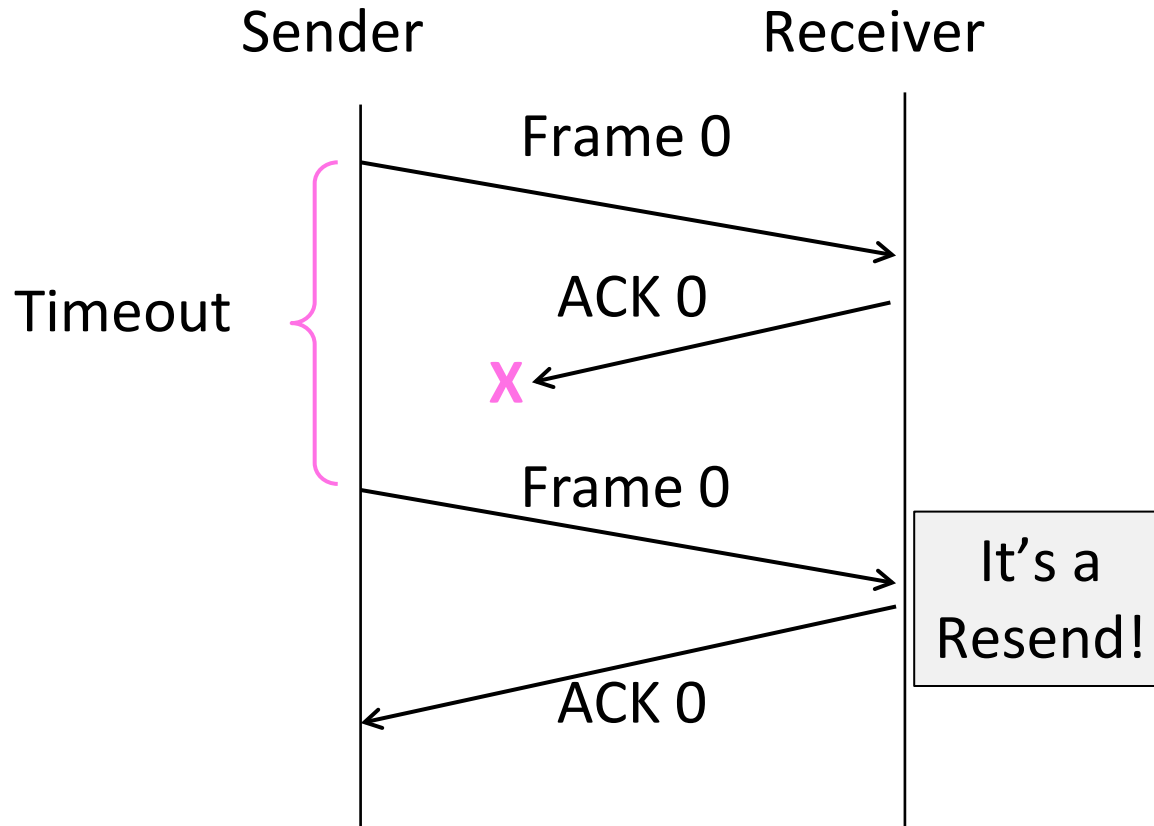
# Stop-and-Wait (3)

- With ACK loss:



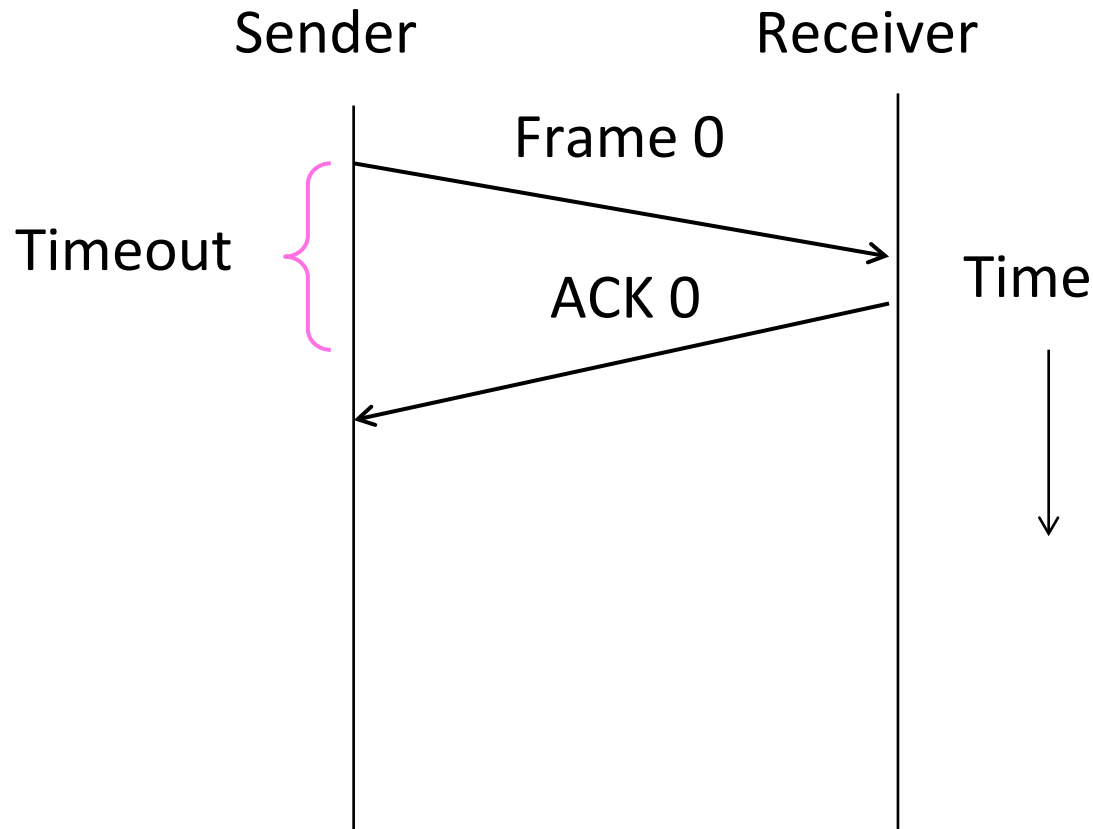
# Stop-and-Wait (4)

- With ACK loss:



# Stop-and-Wait (5)

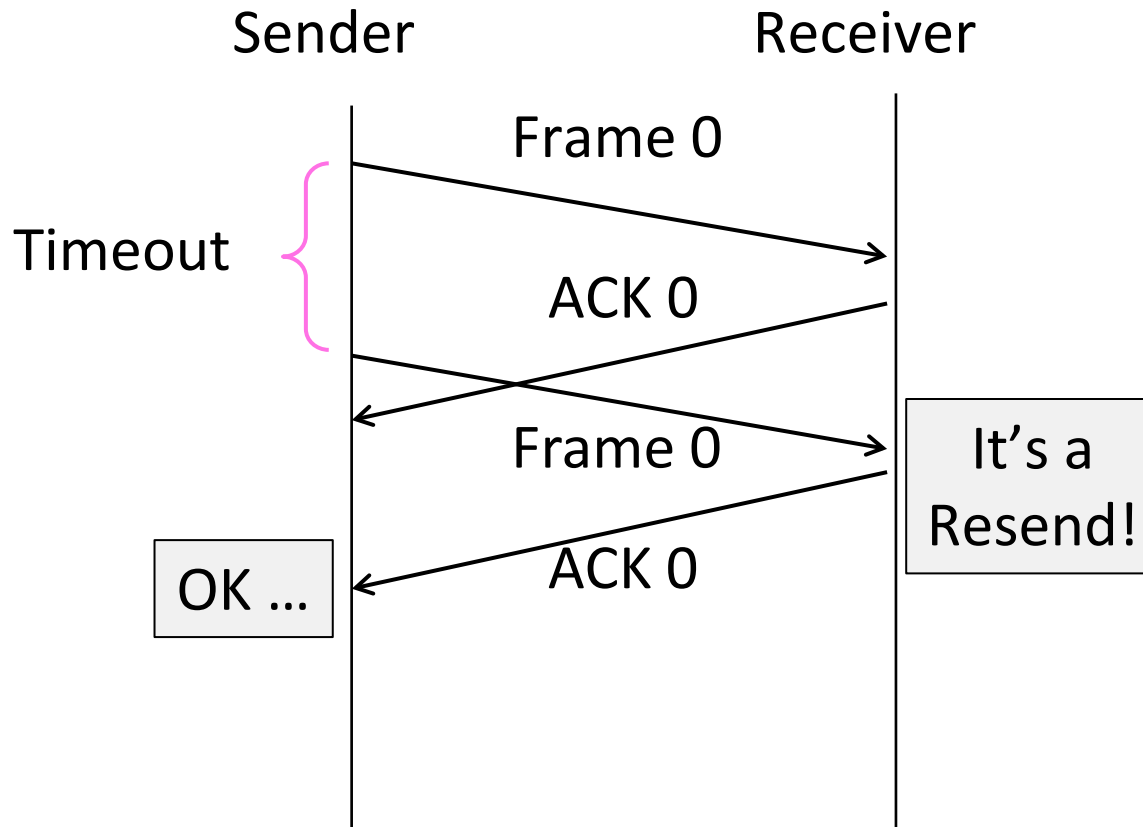
- With early timeout:





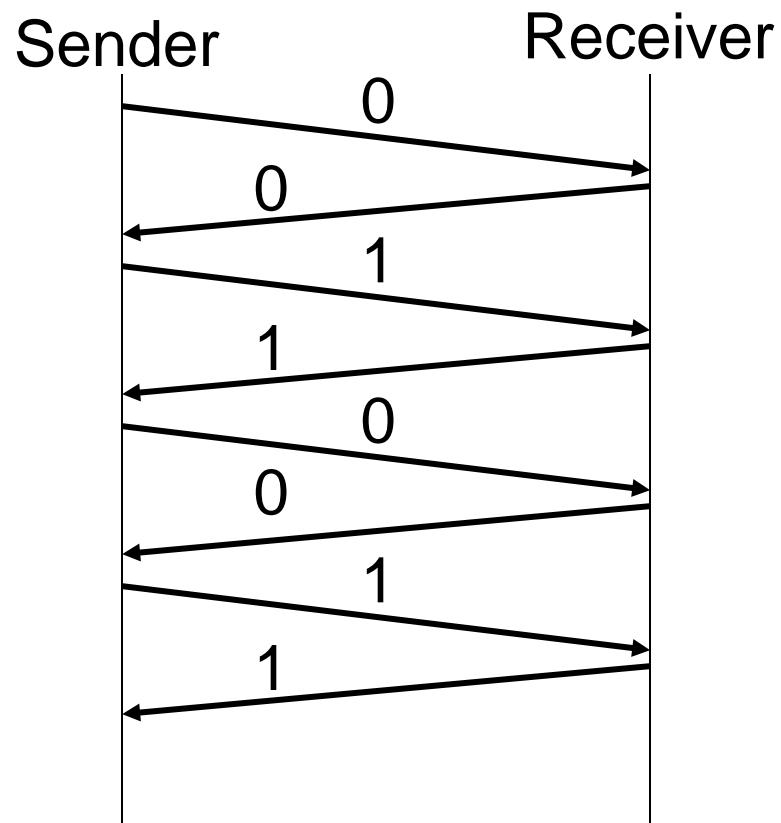
# Stop-and-Wait (6)

- With early timeout:



# Stop-and-Wait Summary

- Only one outstanding packet at a time
- Also called alternating bit protocol
- It allows only a single message to be outstanding from the sender:
  - Fine for LAN. Why?
    - Only one frame fit
  - No efficient for network paths with  $BD \gg 1$



# Limitation of Stop-and-Wait

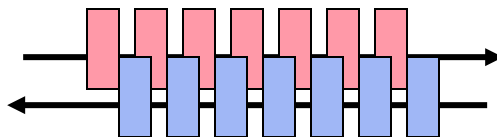


- Lousy performance if trans. delay  $\ll$  prop. delay
  - Actual BW =  $M/2D$
- Example:
  - $R=1\text{Mbps}$ ,  $D=50\text{ms}$ ,  $M=1,250\text{ Bytes}$  (10Kb)
  - RTT (Round Trip Time) =  $2D = 100\text{ms}$
  - How many bytes/sec?
    - Actual BW =  $1,250\text{ Bytes}/100\text{ ms} \rightarrow 12,500\text{ Bytes/s} \rightarrow 100\text{ Kb/s}$  (or 10 packets/sec)  $\rightarrow$  10% bandwidth use
  - What if  $R=10\text{ Mbps}$ ?
    - Actual BW =  $1,250\text{ Bytes}/100\text{ ms} \rightarrow 12,500\text{ Bytes/s} \rightarrow 100\text{ Kb/s}$  (or 10 packets/sec)  $\rightarrow$  1% bandwidth use
- How do we improve performance?



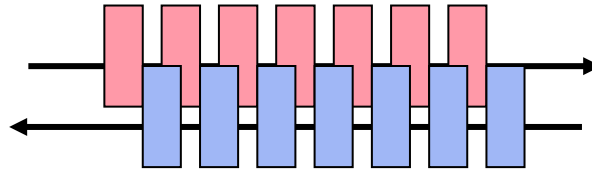
# Sliding Window

- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Remember the bandwidth-delay product?
- Leads to Sliding Window Protocol
  - Generalization of stop-and-wait
  - “Window size” says how much data can be sent without waiting for an acknowledgement
  - Allows  $W$  packets to be outstanding
  - Can send  $W$  packets per RTT ( $=2D$ )
  - Need  $W=2BD$  to fill network path



# Sliding Window Example

- What W will use the network capacity?
- Ex:  $R = 1 \text{ Mbps}$ ,  $D = 50 \text{ ms}$ ,  $M = 1250 \text{ bytes}$ 
  - $2BD = 10^6 \text{ b/sec} \times 100 \cdot 10^{-3} \text{ sec} = 100 \text{ Kbit}$
  - $W = 2BD = 10 \text{ packets of } 1250 \text{ bytes}$



- Ex: What if  $R=10 \text{ Mbps}$ ?
  - $2BD = 1000 \text{ Kbit}$
  - $W = 2BD = 100 \text{ packets of } 1250 \text{ bytes}$



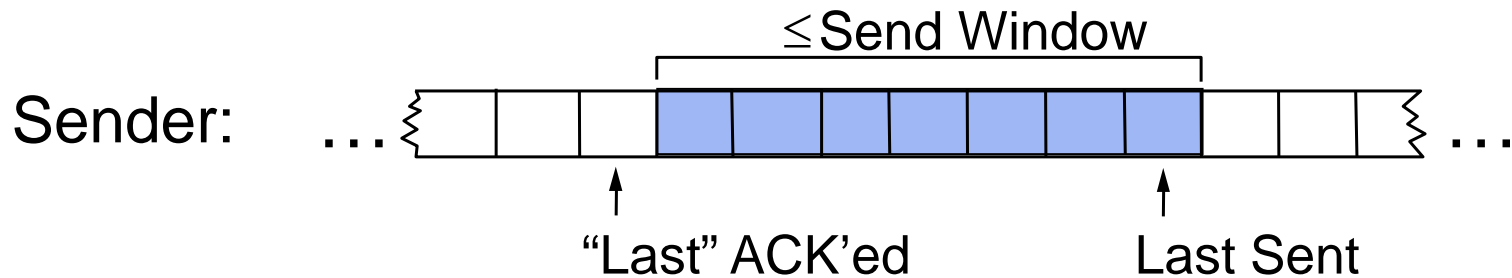
# Sliding Window Protocol

---

- Many variations, depending on how buffer, acknowledgements, and retransmissions are handled
- Go-Back-N
  - Simplest version, can be inefficient
- Selective Repeat
  - More complex, better performance



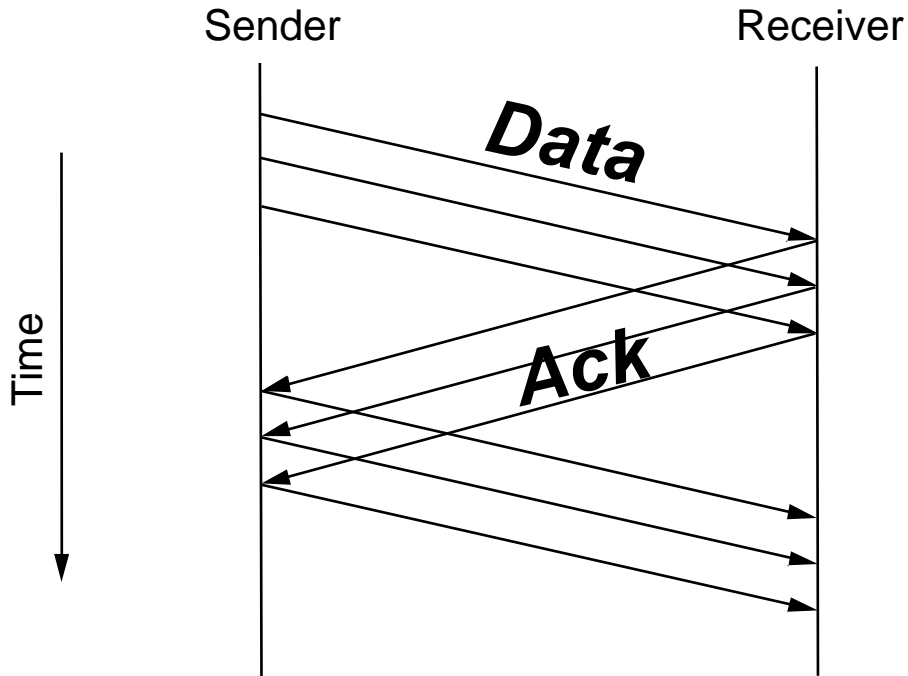
# Sliding Window – Sender



- Window bounds outstanding data
  - Implies need for buffering at sender
    - Specifically, must buffer unack’ed data
- “Last” ACK applies to in-order data
  - Need not buffer acked data
- Sender maintains timers too
  - Go-Back-N: one timer, send all unacknowledged on timeout
  - Selective Repeat: timer per packet, resend as needed



# Sliding Window – Ack Choices



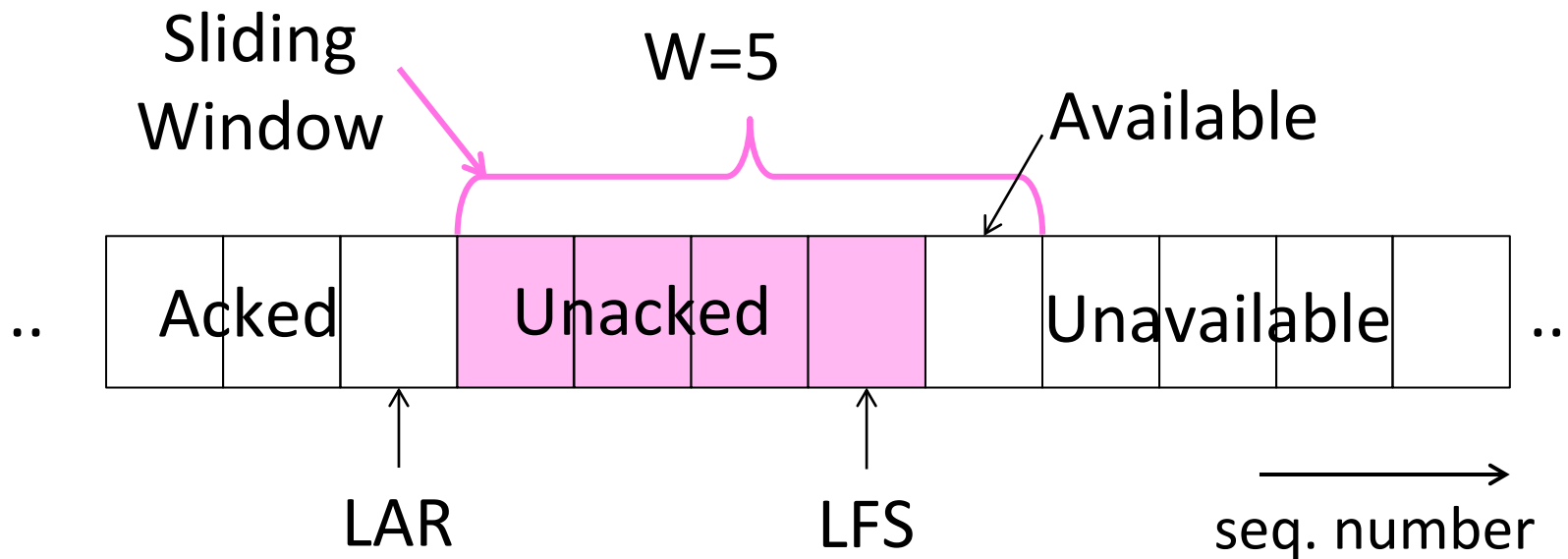
- Receiver ACK choices:
  - Individual
    - Each packet acked
  - Cumulative (TCP)
    - Ack says “got everything up to X-1...”
    - really, “my ack means that the next byte I am expecting is X”
  - Selective (newer TCP)
    - Ack says “I got X through Y”
  - Negative
    - Ack says “I did not get X”





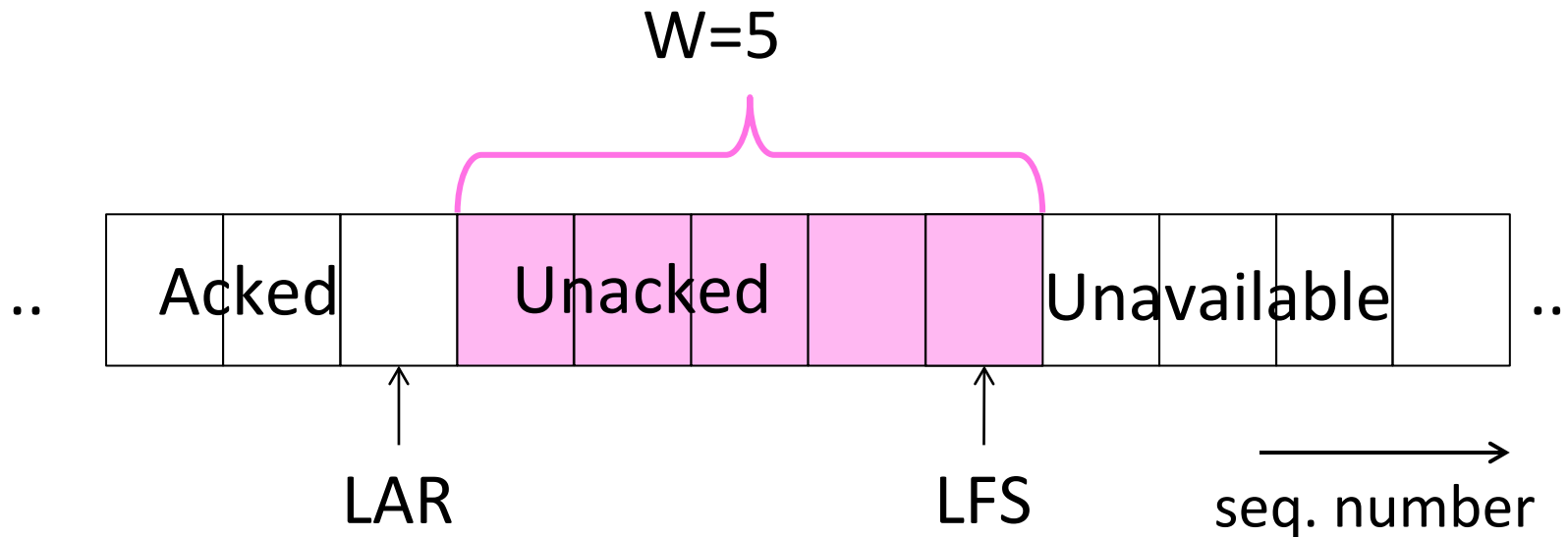
# Sliding Window – Sender

- Sender buffers up to  $W$  segments until they are acknowledged
  - LFS = Last Frame Sent, LAR = Last Ack Rec'd
  - Sends while  $LFS - LAR \leq W$



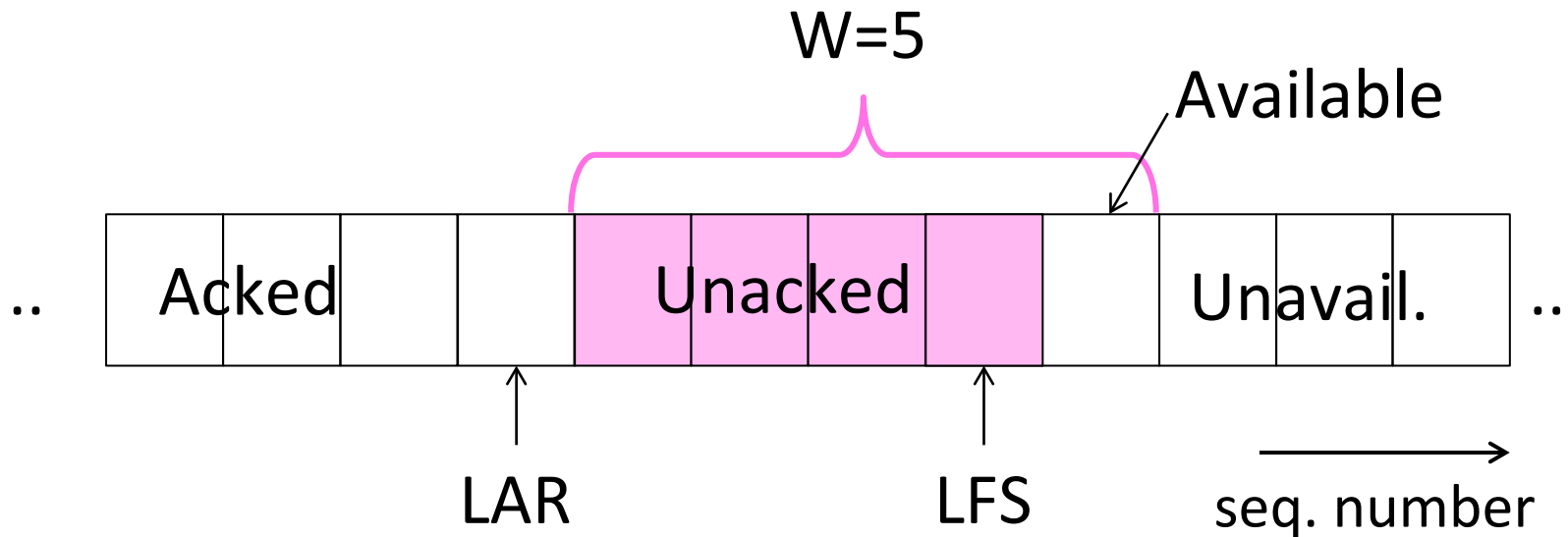
# Sliding Window – Sender (2)

- Transport accepts another segment of data from the Application ...
  - Transport sends it (as  $LFS - LAR \rightarrow 5$ )

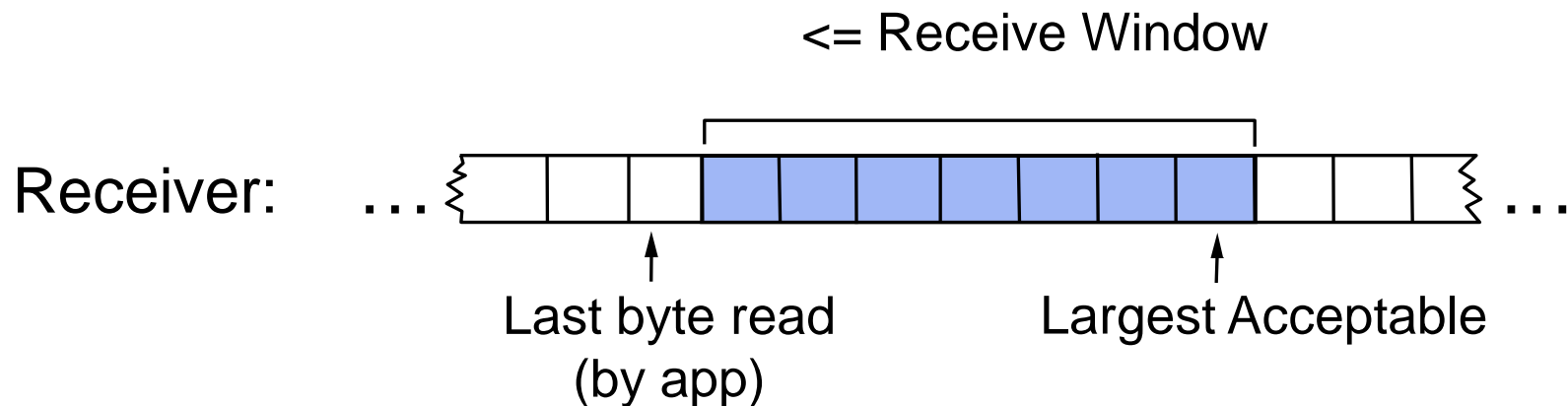


# Sliding Window – Sender (3)

- Next higher ACK arrives from peer ...
  - Window advances, buffer is freed
  - $LFS - LAR \rightarrow 4$  (can send one more)



# Sliding Window – Receiver



- Receiver buffers too:
  - Data may arrive out-of-order
  - Or faster than can be consumed by receiving process
- No sense having more data on the wire than can be buffered at the receiver
  - In other words, receiver buffer size should limit the sender's window size



# Sliding Window – Go-Back-N

---

- Receiver keeps a  $W$  buffer for incoming segments
  - State variable,  $LAS$  = Last Ack Sent
- On receive:
  - If seq. number is  $LAS+1$ , accept and pass it to app, update  $LAS$ , send ACK ( $LAS+2$ )
  - Otherwise, store it in the buffer, send ACK for  $LAS+1$



# Sliding Window – Selective Repeat

---

- Receiver passes data to app in order, and buffers out-of-order segments to reduce transmissions
- ACK conveys highest in-order segment, plus hints about out-of-order segments
- Modern TCP uses a selective repeat design; we'll see details later



# Sliding Window – Selective Repeat (2)

---

- Buffers  $W$  segments, keeps state variable,  $LAS = \text{Last Ack Sent}$
- On receive:
  - Buffer segments  $[LAS+1, LAS+W]$
  - Pass up to app in-order segments from  $LAS+1$ , and update  $LAS$
  - Send ACK for  $LAS$  regardless



# Sliding Window – Retransmissions

---

- Go-Back-N sender uses a single timer to detect losses
  - On timeout, resends buffered packets start at LAR+1
- Selective Repeat sender uses a timer per unacked segment to detect losses
  - On timeout for segment, resend it
  - Hope to resend fewer segments





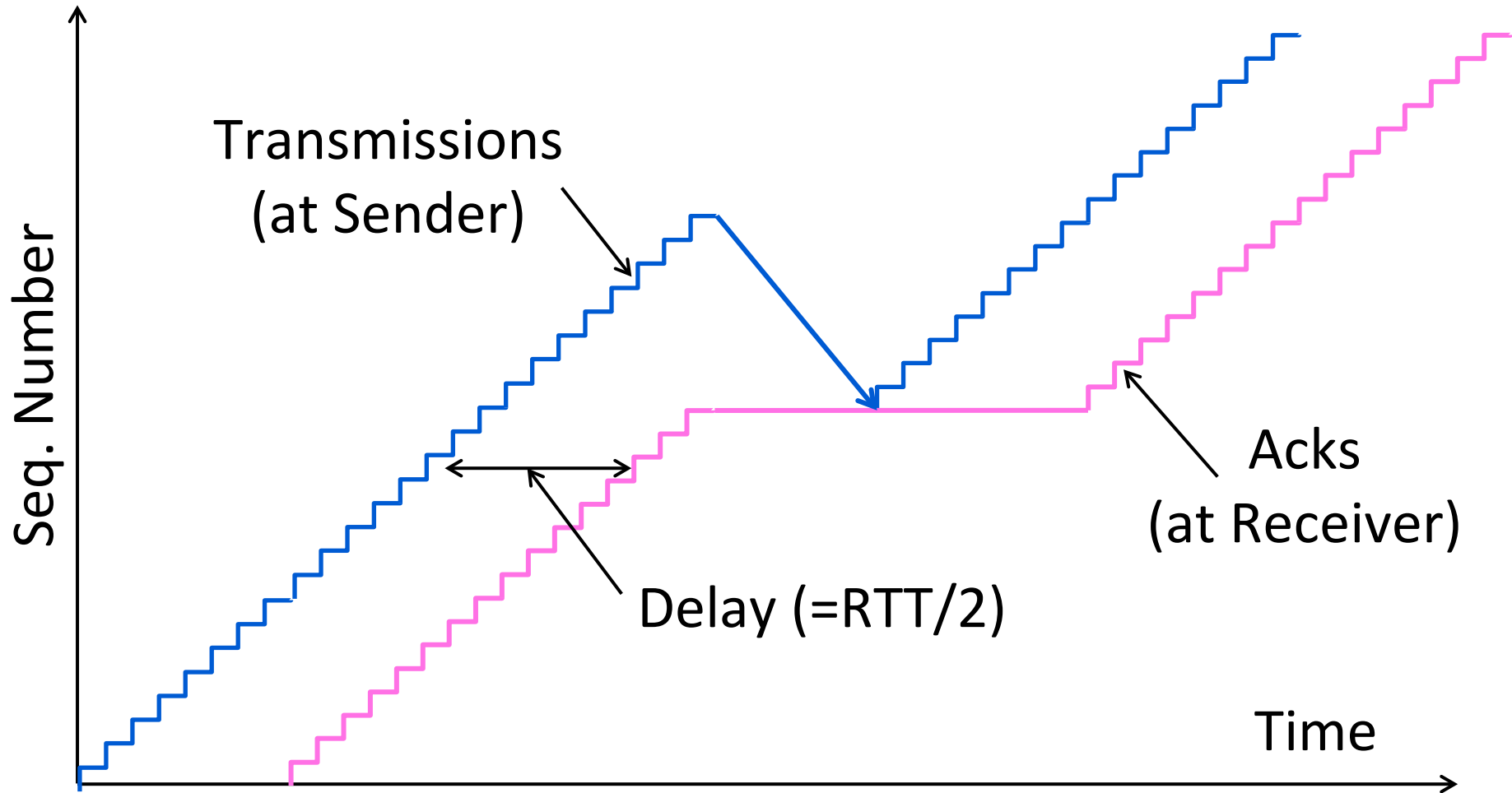
# Sequence Numbers

---

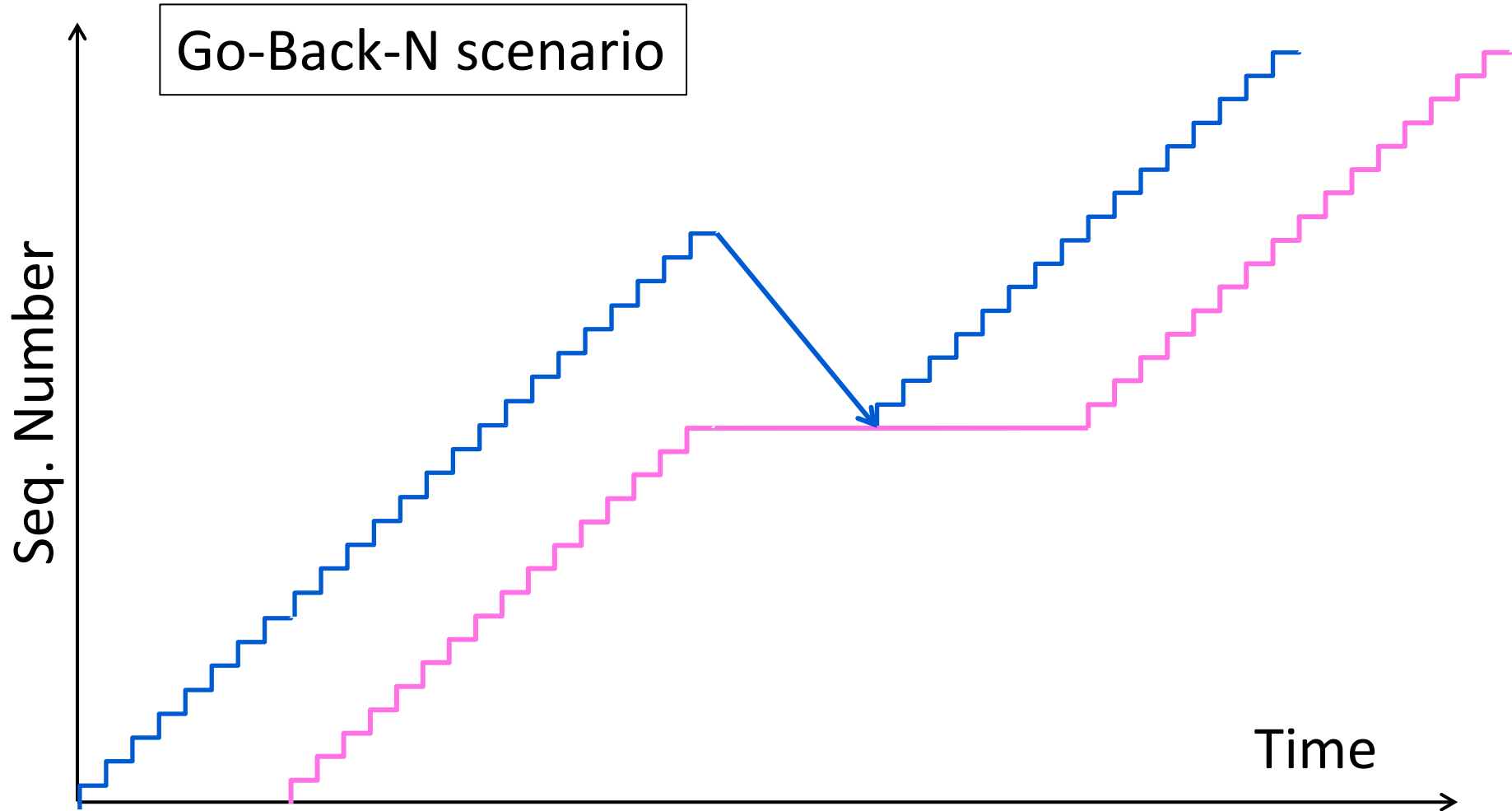
- Need more than 0/1 for Stop-and-Wait ...
  - But how many?
- For Selective Repeat, need  $W$  numbers for segments, plus  $W$  for acks of earlier segments
  - $2W$  seq. numbers
  - Fewer for Go-Back-N ( $W+1$ )
- Typically implement seq. number with an  $N$ -bit counter that wraps around at  $2^N - 1$ 
  - E.g.,  $N=8$ : ..., 253, 254, 255, 0, 1, 2, 3, ...



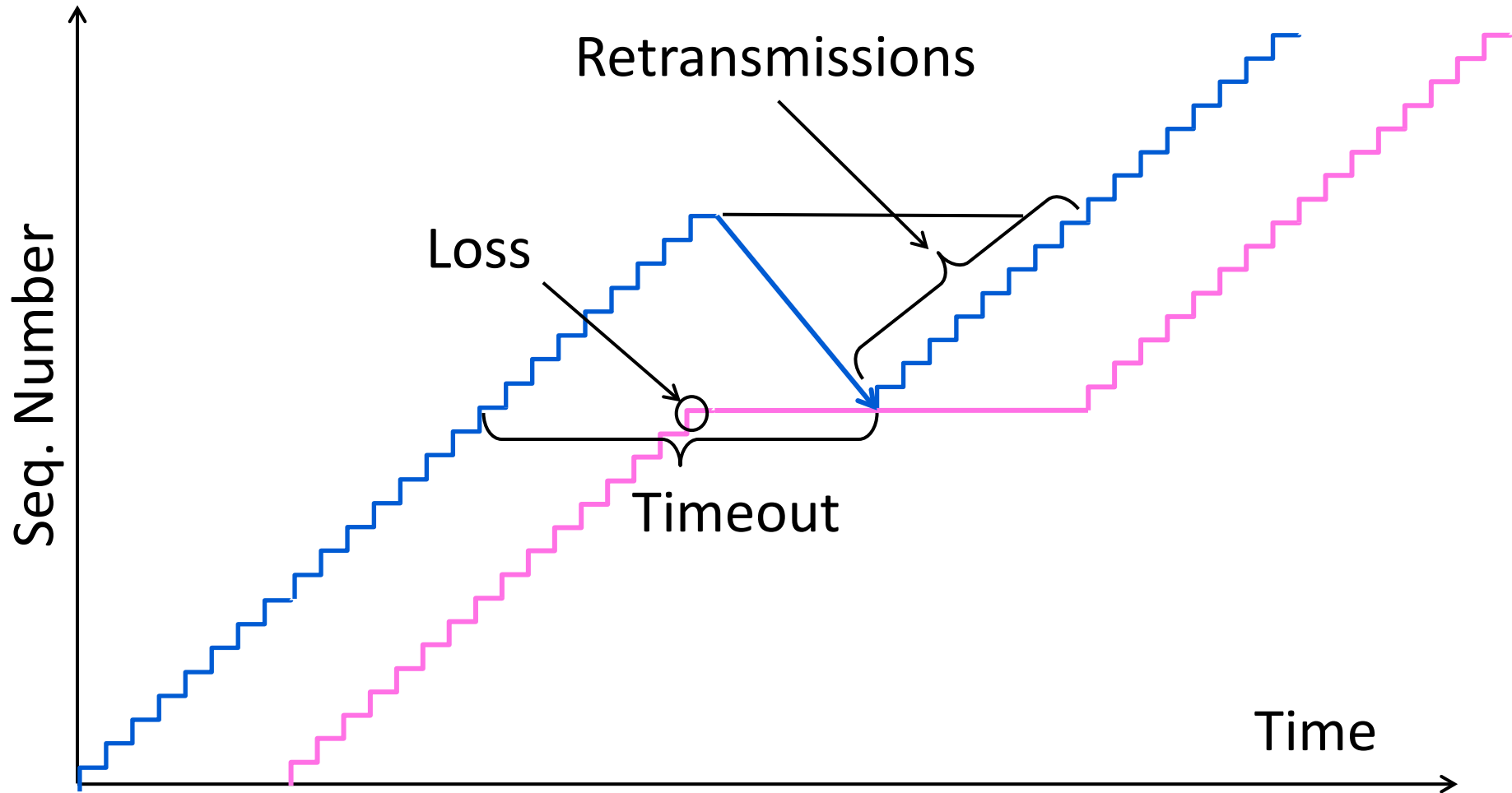
# Sequence Time Plot



# Sequence Time Plot (2)

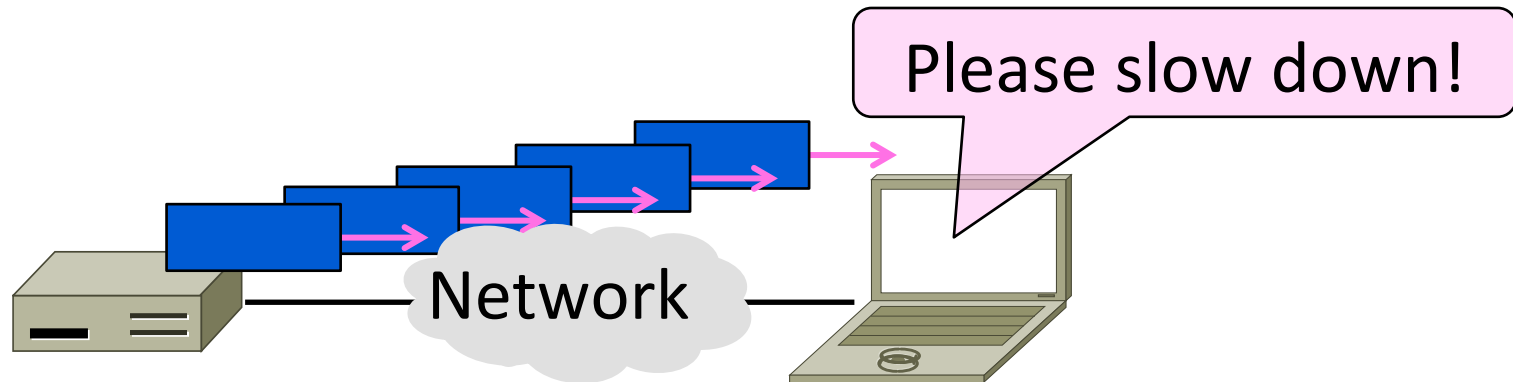


# Sequence Time Plot (3)



# Flow Control

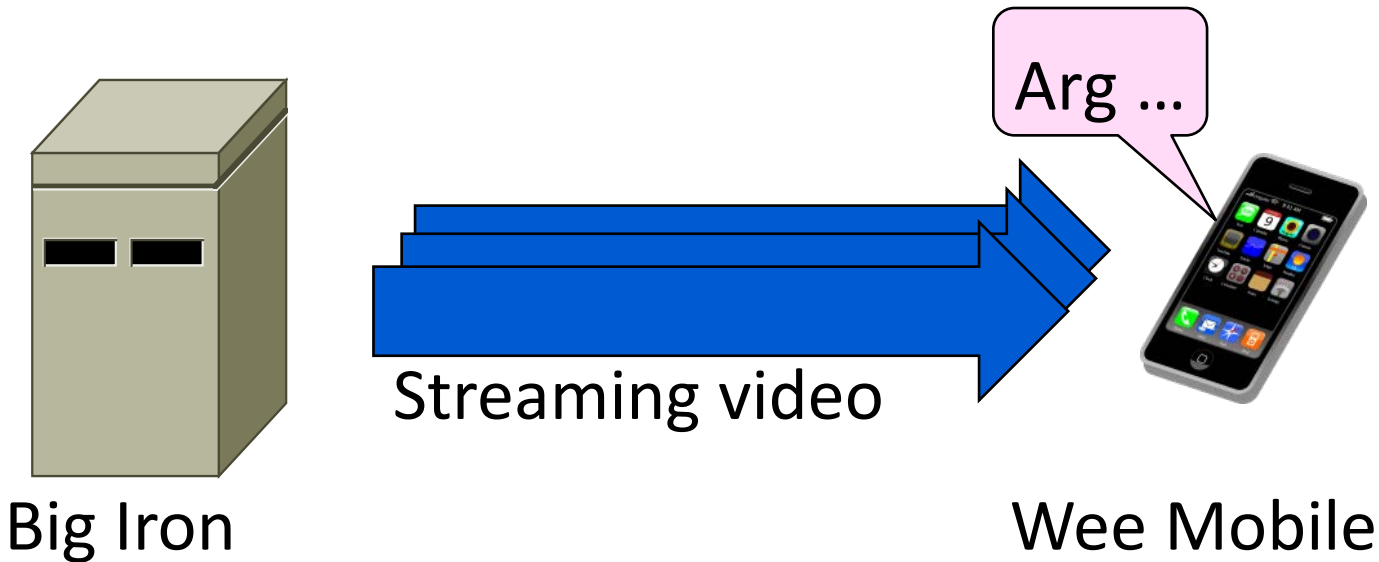
- Adding flow control to the sliding window algorithm
  - To slow down the over-enthusiastic sender



# Problem

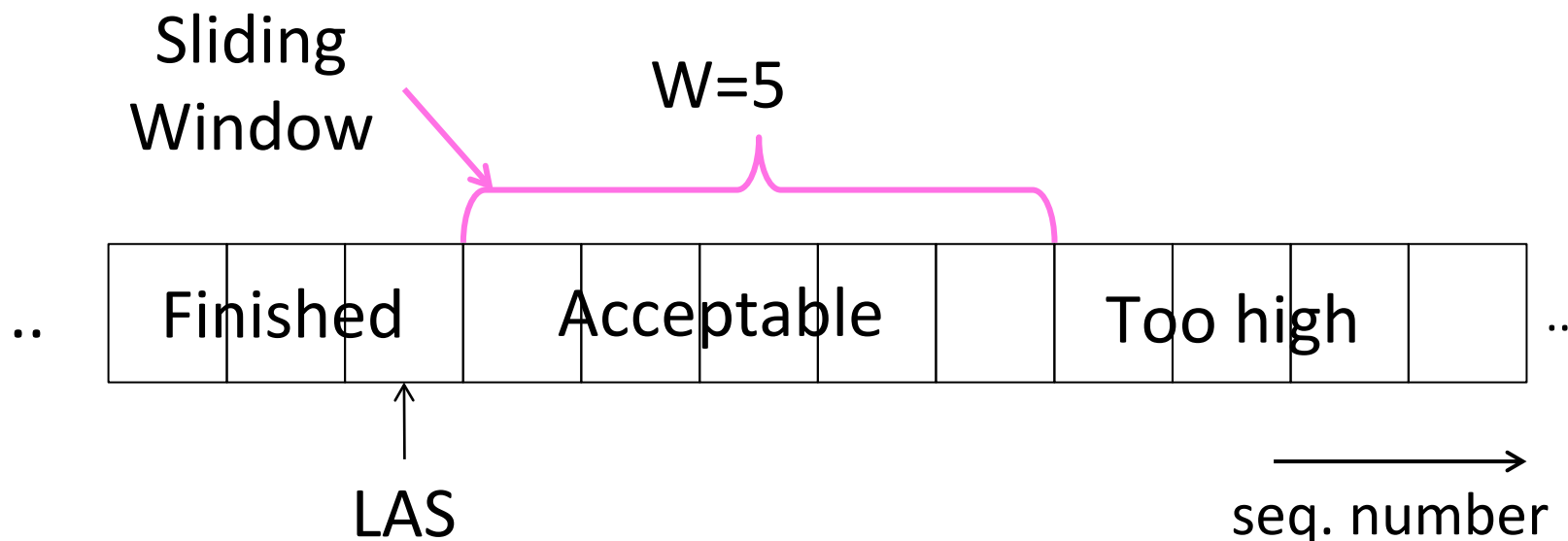
---

- Sliding window uses pipelining to keep the network busy
  - What if the receiver is overloaded?



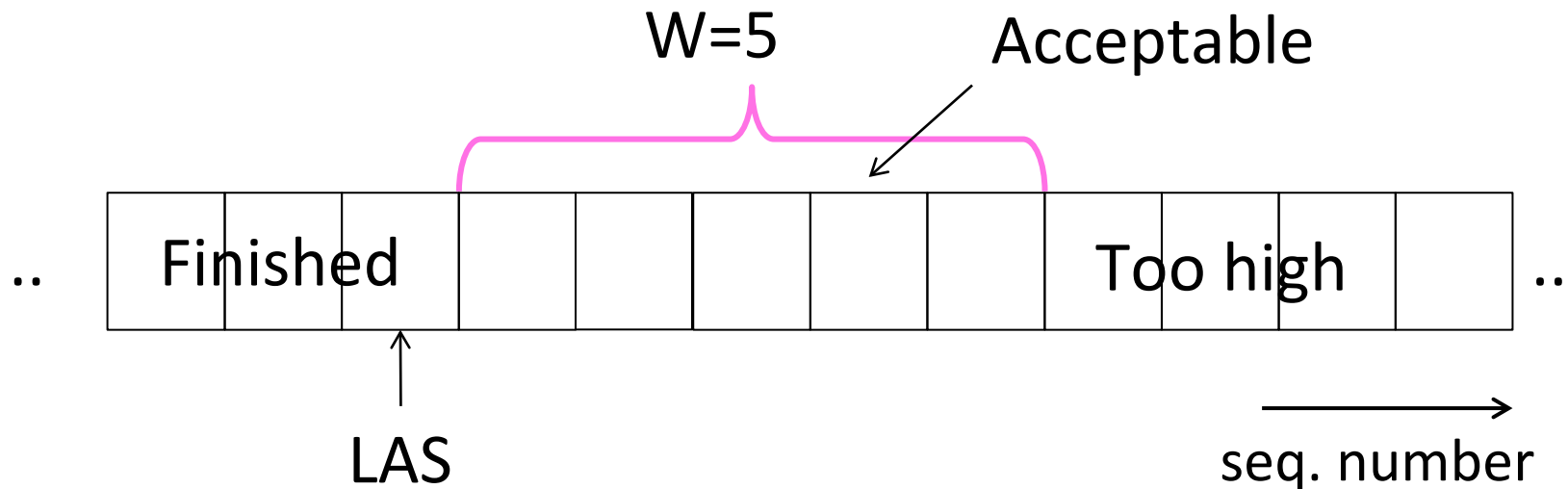
# Sliding Window – Receiver

- Consider receiver with  $W$  buffers
  - LAS = LAST ACK SENT, app pulls in-order data from buffer with `recv()` call (more on this later)



# Sliding Window – Receiver (2)

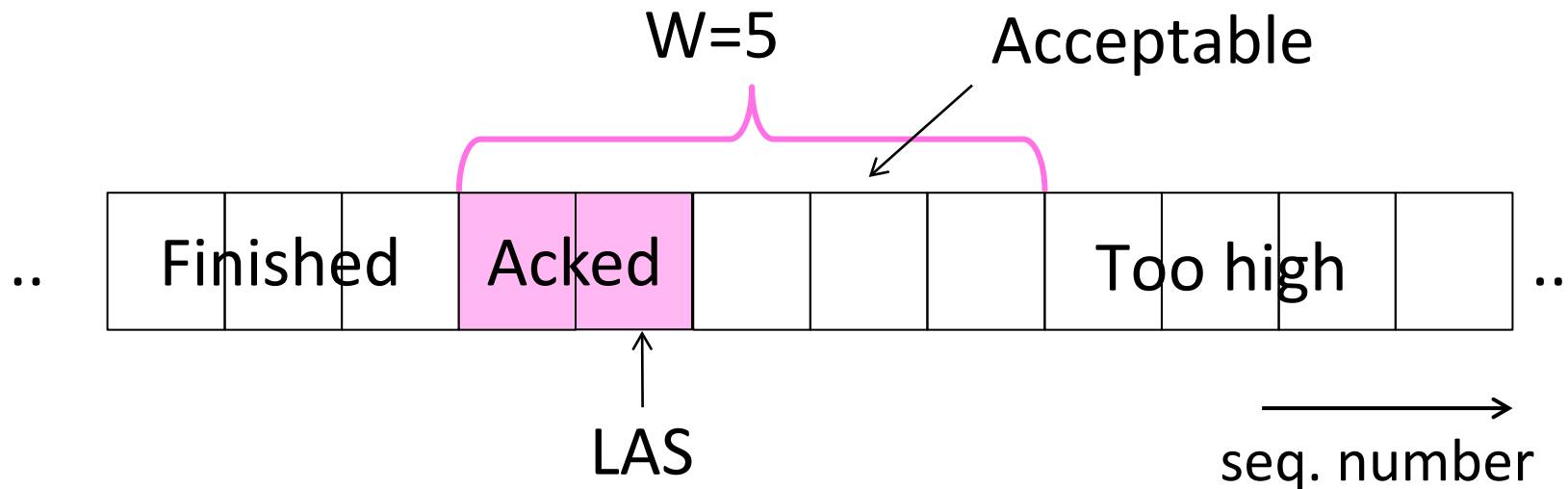
- Suppose the next two segments arrive but app does not call `recv()`





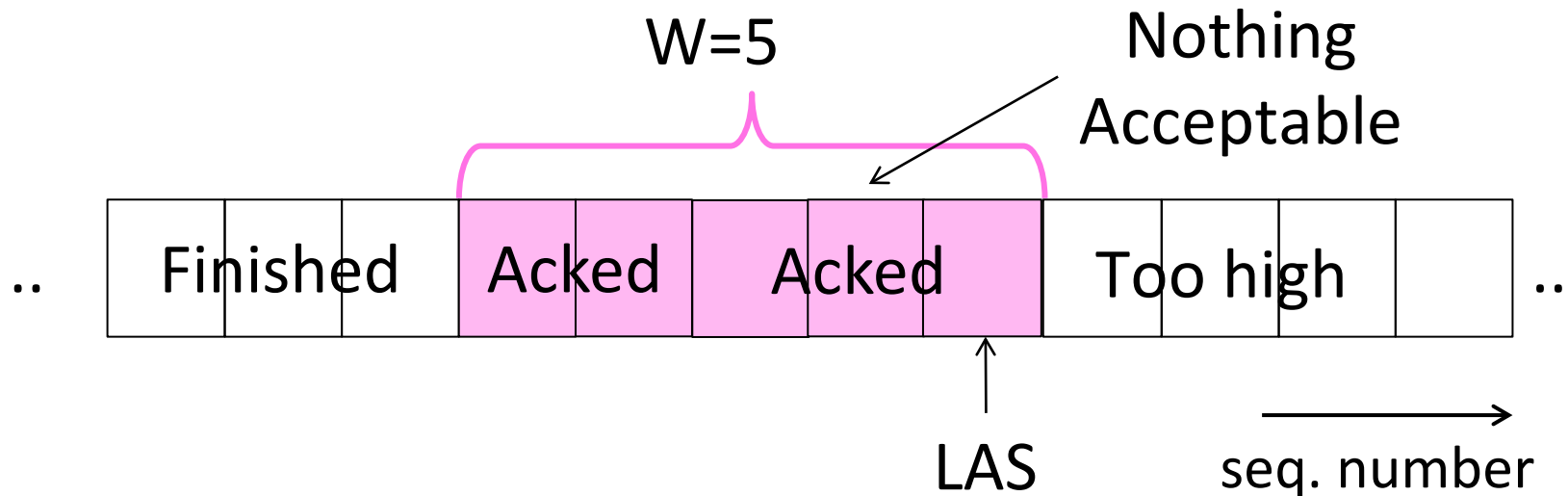
# Sliding Window – Receiver (3)

- Suppose the next two segments arrive but app does not call `recv()`
  - LAS rises, but we can't slide window!



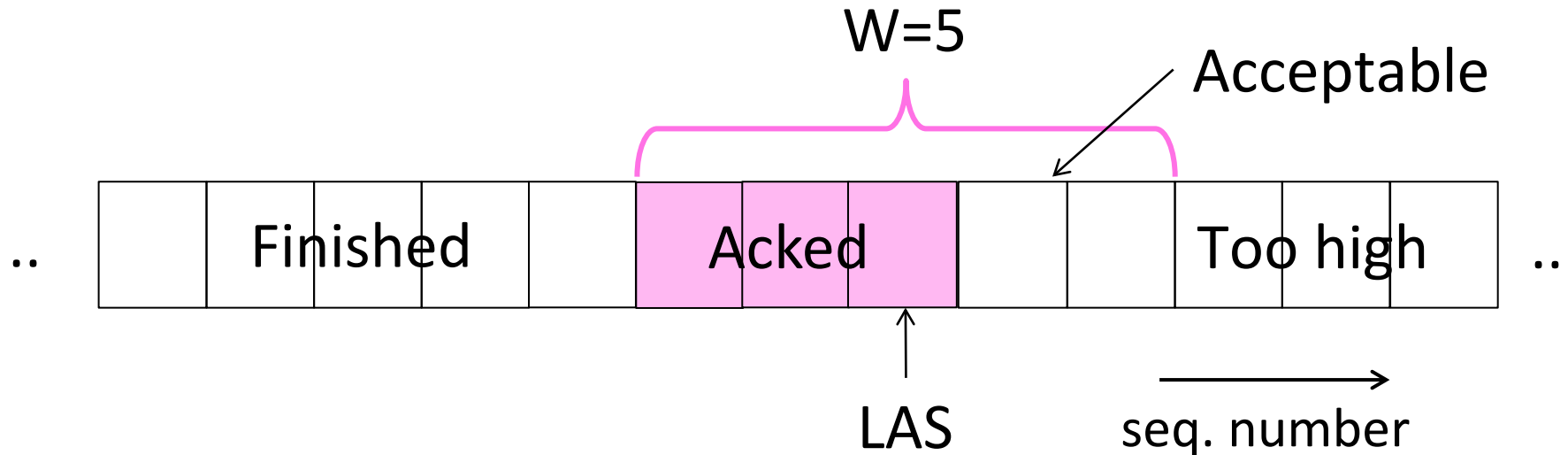
# Sliding Window – Receiver (4)

- If further segments arrive (even in order) we can fill the buffer
  - Must drop segments until app recvs!



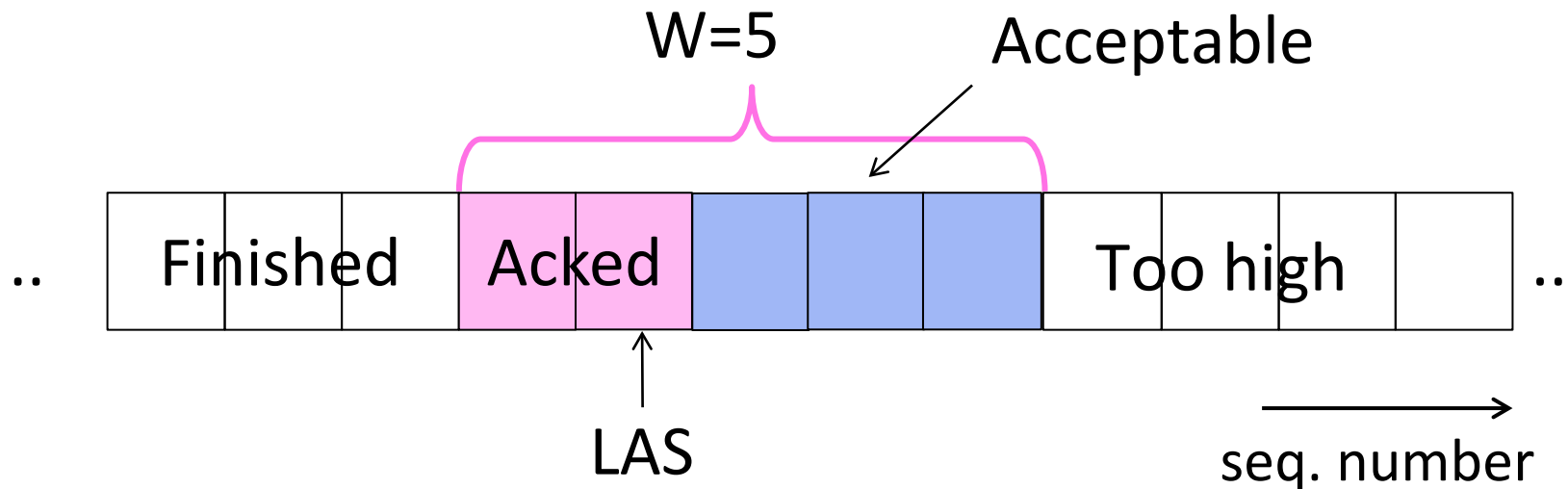
# Sliding Window – Receiver (5)

- App recv() takes two segments
  - Window slides (phew!)



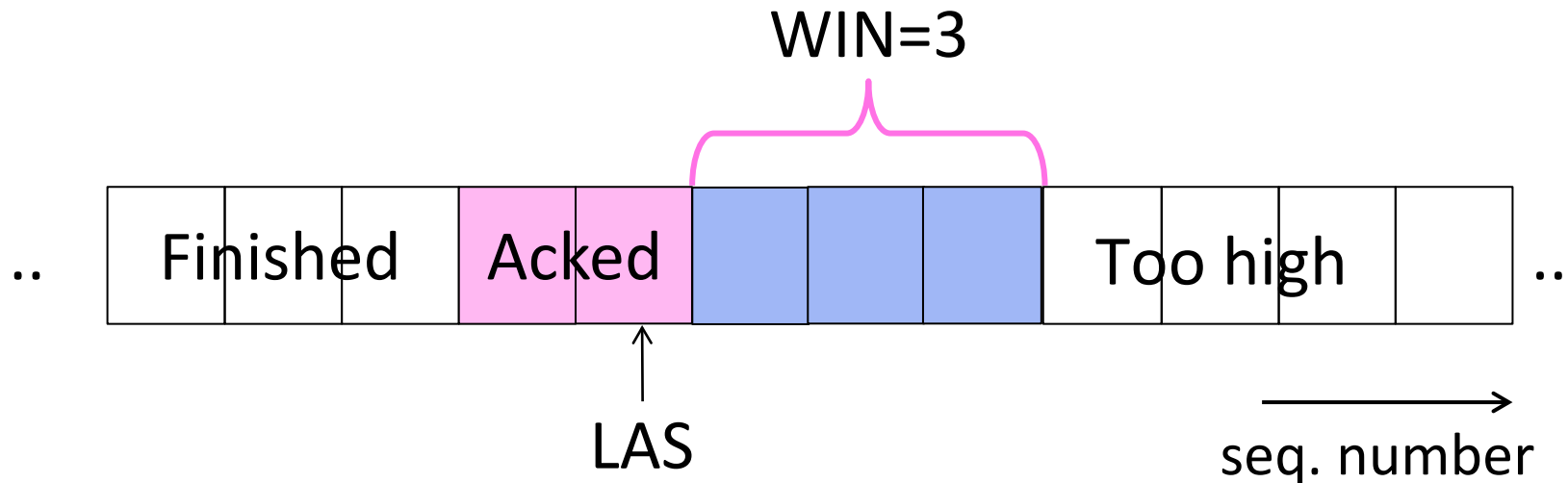
# Flow Control

- Avoid loss at the receiver by telling sender the available buffer space
  - $WIN = \#Acceptable$ , not  $W$  (from LAS)



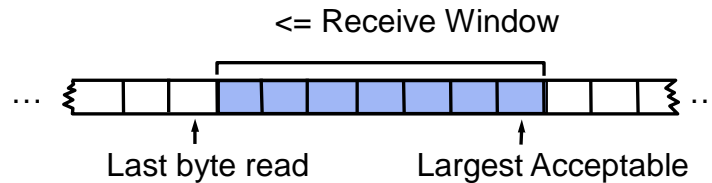
## Flow Control (2)

- Sender uses the lower of the sliding window and flow control window (WIN) as the effective window size



# Flow Control Summary

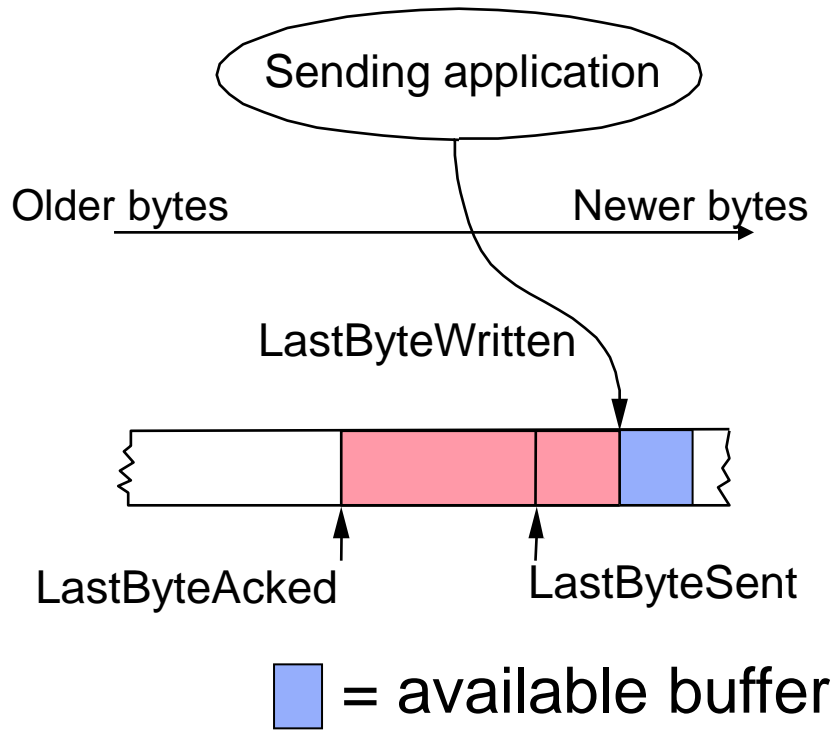
- Sender must transmit data no faster than it can be consumed by receiver
  - Receiver might be a slow machine
  - App might consume data slowly



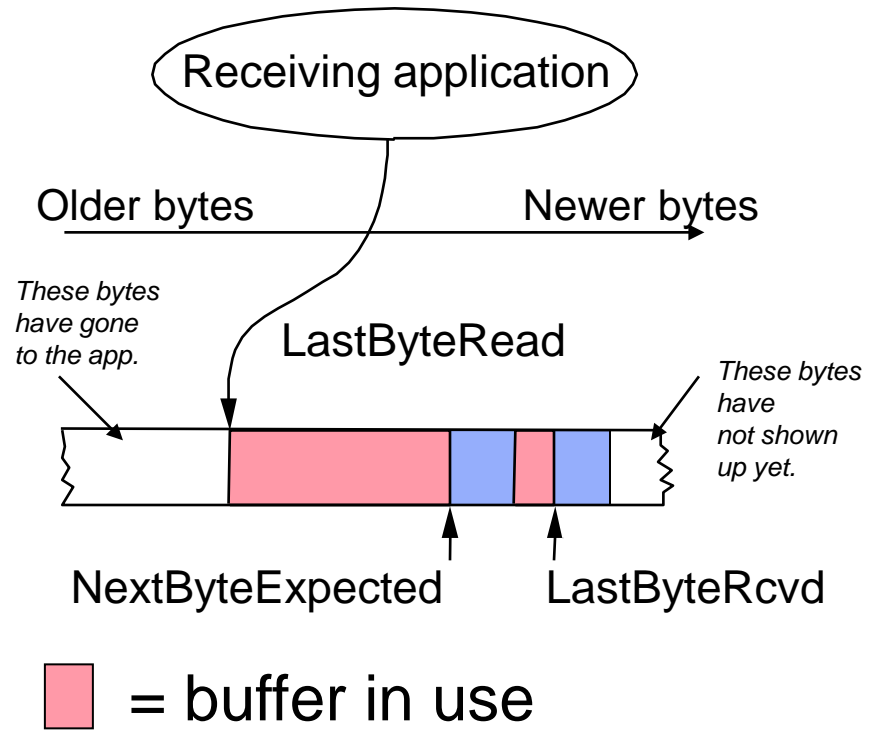
- Accomplish by adjusting the size of sliding window used at the sender
  - sender adjusts based on receiver's feedback about available buffer space
  - the receiver tells the sender an "Advertised Window"



# Sender and Receiver Buffering



$\text{LastByteAked} \leq \text{LastByteSent}$   
 $\text{LastByteSent} \leq \text{LastByteWritten}$

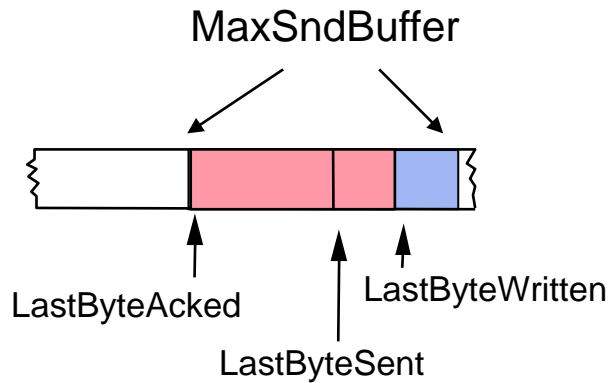


$\text{LastByteRead} < \text{NextByteExpected}$   
 $\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$   
== if data arrives in order  
else start of first gap.

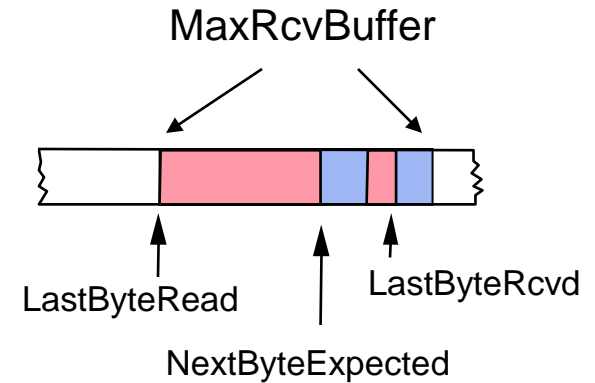


# Flow Control

**Sender:**



**Receiver:**



Receiver's goal: always ensure that  $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$

- in other words, ensure it never needs to buffer more than MaxRcvBuffer data

To accomplish this, receiver advertises the following window size:

- $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$
- *"All the buffer space minus the buffer space that's in use."*





# Flow Control on the Receiver

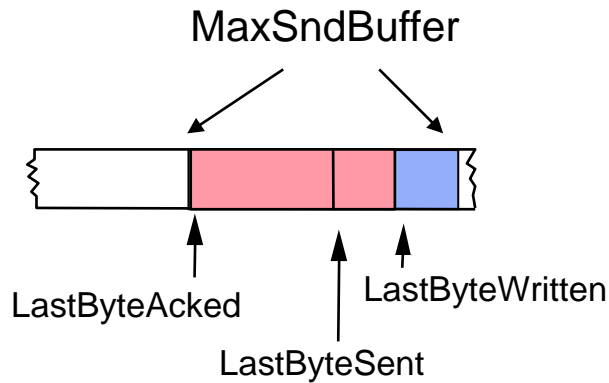
---

- As data arrives:
  - Receiver acknowledges it so long as all preceding bytes have also arrived
  - ACKs also carry a piggybacked AdvertisedWindow
  - So, an ACK tells the sender:
    1. All data up to the ACK'ed seqno has been received
    2. How much more data fits in the receiver's buffer, as of receiving the ACK'ed data
- AdvertisedWindow:
  - Shrinks as data is received
  - Grows as receiving app. reads the data from the buffer

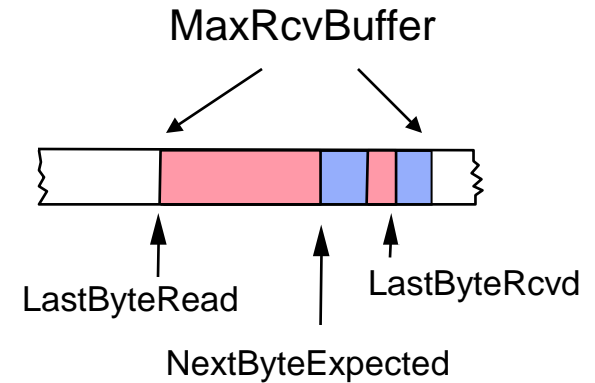


# Flow Control on the Sender

**Sender:**



**Receiver:**



Sender's goal: always ensure that  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$

- In other words, don't send that which is unwanted

Notion of "EffectiveWindow": how much new data it is OK for sender to currently send

- $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$

*OK to send that which there is room for, which is that which was advertised (AdvertisedWindow) minus that which I've already sent since receiving the last advertisement.*



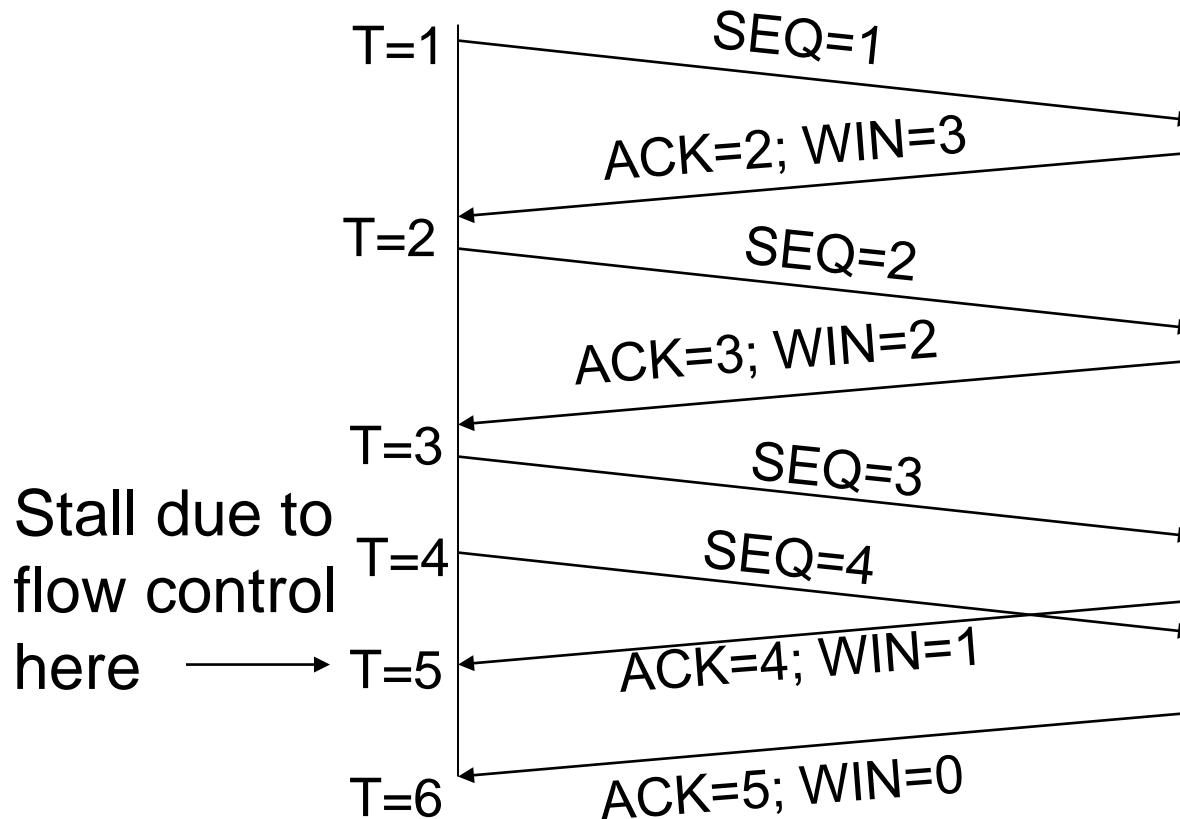
# Sending Side

---

- As acknowledgements arrive:
  - Advance LastByteAcked
  - Update AdvertisedWindow
  - Calculate new EffectiveWindow
    - If  $\text{EffectiveWindow} > 0$ , it is OK to send more data
- One last detail on the sender:
  - Sender has finite buffer space as well
    - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
  - OS needs to block application writes if buffer fills
    - i.e., block `write(y)` if
$$(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer}$$



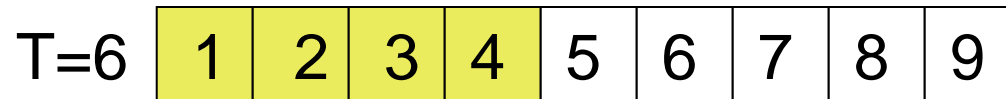
# Example – Exchange of Packets






Receiver has buffer of size 4 and application doesn't read



# Example – Buffer at Sender

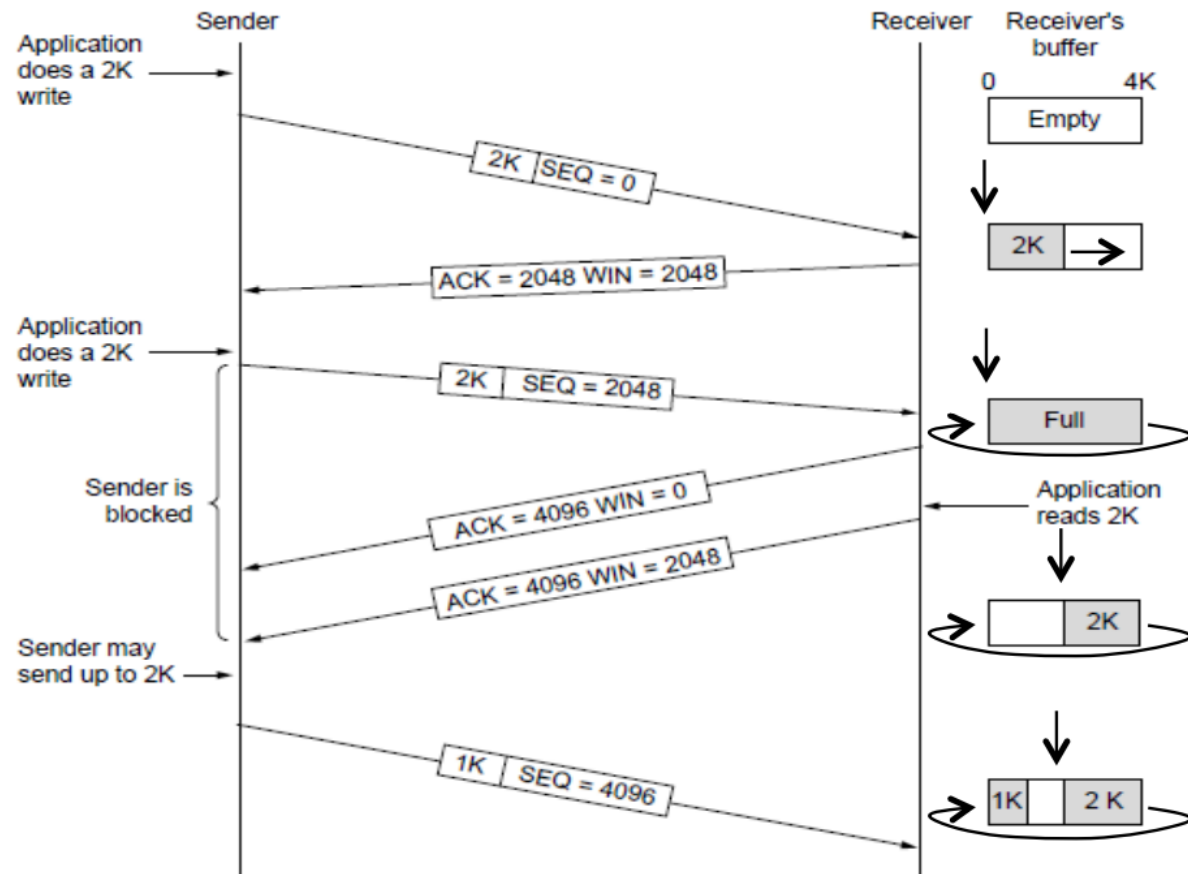


 =acked  
 =sent  
 =advertised



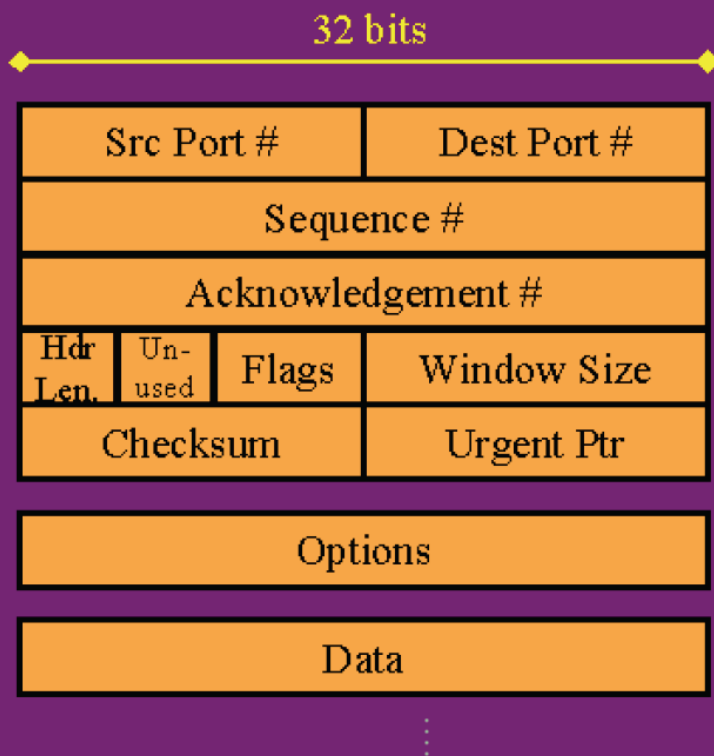
# Flow Control Full Example

- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - $\text{SEQ} + \text{length} < \text{ACK} + \text{WIN}$
  - 4KB buffer at receiver
  - Circular buffer of bytes



# TCP Packet Format (NOT in Midterm!)

## TCP Packet Format



16-bit window size gets  
Cramped with large  
Bandwidth x delay

16 bits --> 64K  
BD ethernet: 122KB  
STS24 (1.2Gb/s): 14.8MB

32-bit sequence number  
must not wrap around  
faster than the maximum  
packet lifetime. (120  
seconds)  
-- 622Mb/s link: 55 secs



# Sliding Window Functions

---

- Sliding window is a mechanism
- It supports multiple functions:
  - Reliable delivery
    - *If I hear you got it, I know you got it.*
    - ACK (Ack # is “next byte expected”)
  - In-order delivery
    - *If you get it, you get it in the right order.*
    - SEQ # (Seq # is “the byte this is in the sequence”)
  - Flow control
    - *If you don't have room for it, I won't send it.*
    - Advertised Receiver Window
    - AdvertisedWindow is amount of free space in buffer





# Key Concepts

---

- Transport layer allows processes to communicate with stronger guarantees, e.g., reliability
- Basic reliability is provided by ARQ mechanisms
  - Stop-and-Wait through Sliding Window plus retransmissions

