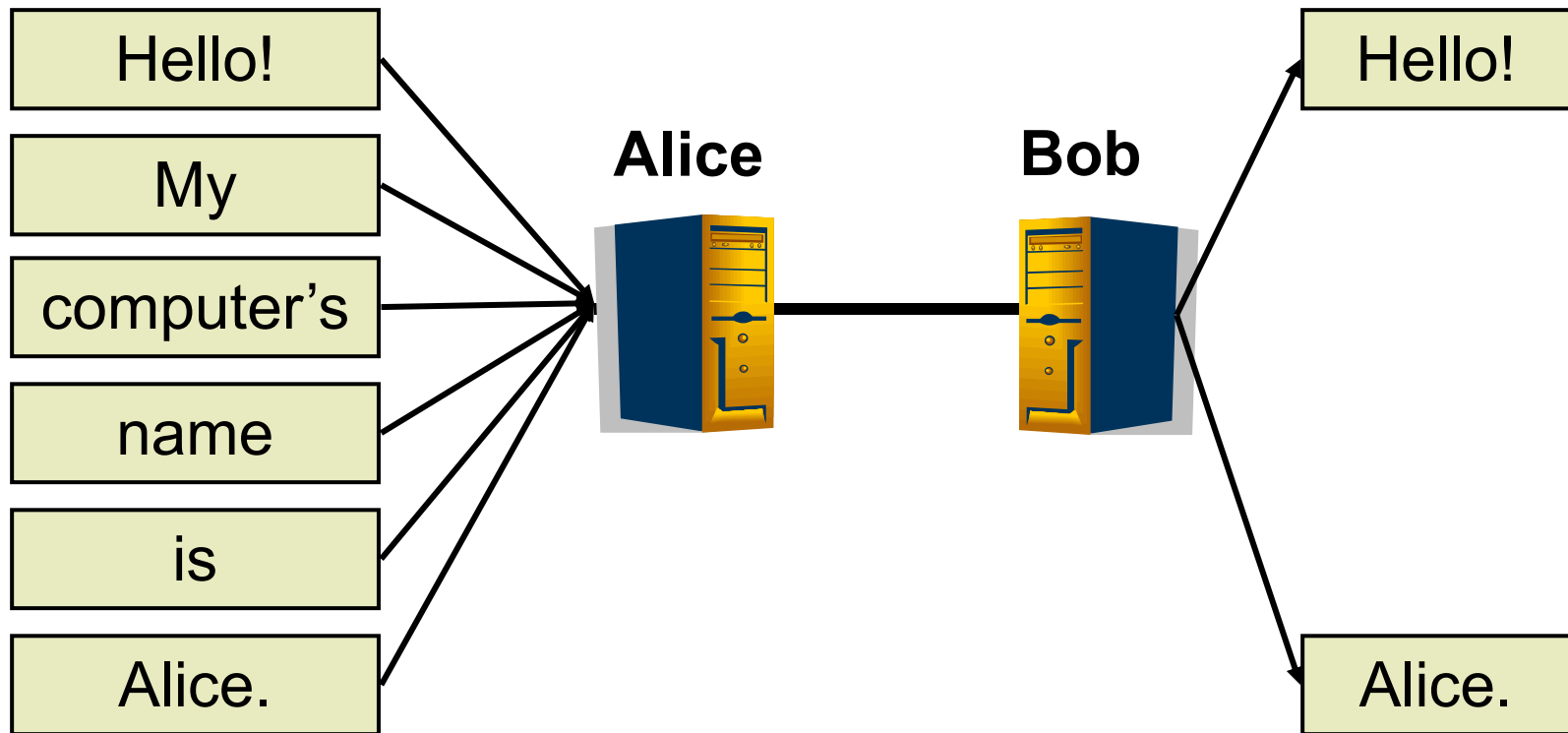


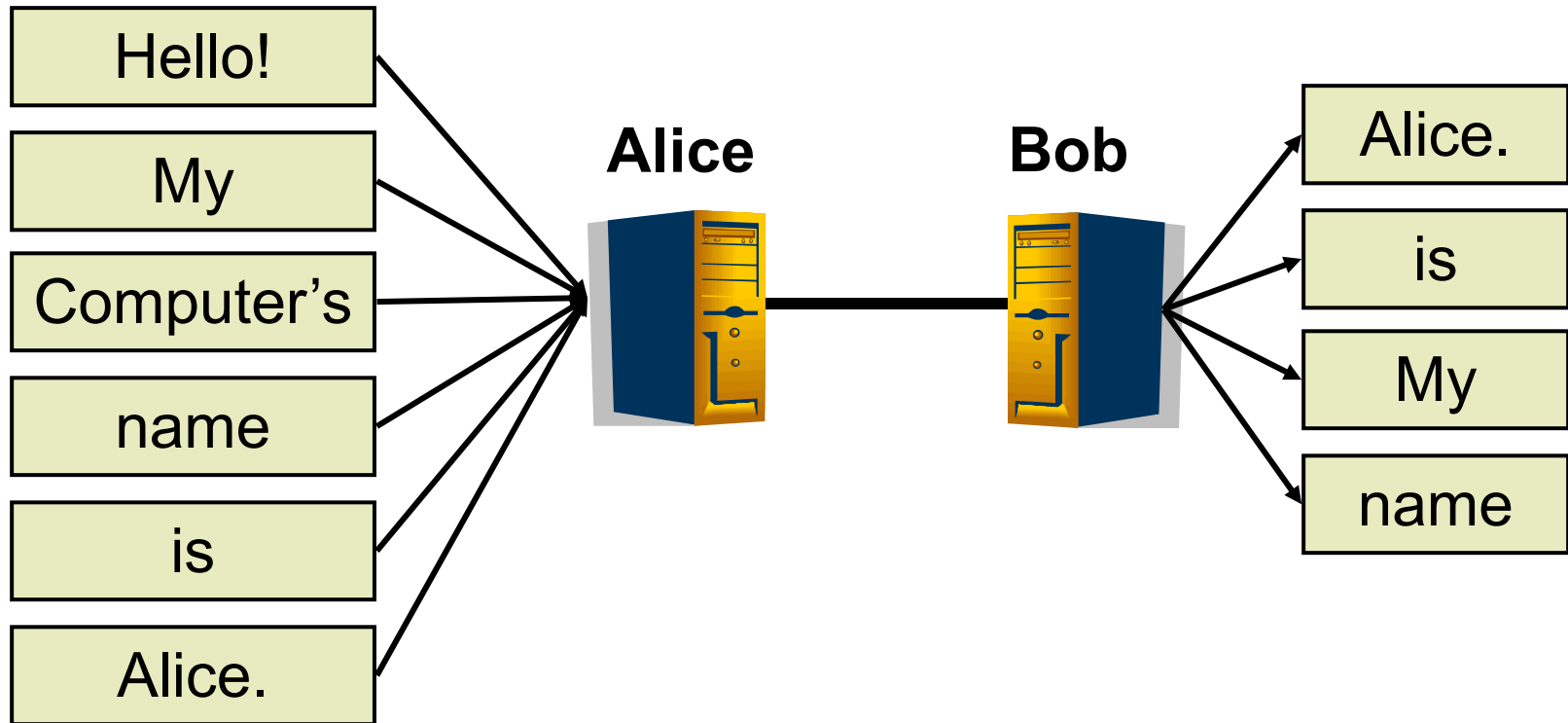
CS 439: Wireless Networking

Transport Layer – dealing with errors and unreliability

Reliable Transmission



Reliable Transmission



Reliable Transmission

- ▶ Suppose error protection identifies valid and invalid packets
 - ▶ How?
- ▶ Can we make the channel appear reliable?
 - ▶ Insure packet delivery
 - ▶ Maintain packet order
 - ▶ Provide reliability at full link capacity



Reliable Transmission Outline

- ▶ **Fundamentals of Automatic Repeat reQuest (ARQ) algorithms**
 - ▶ A family of algorithms that provide reliability through retransmission
- ▶ **ARQ algorithms (simple to complex)**
 - ▶ stop-and-wait
 - ▶ sliding window
 - ▶ go-back-n
 - ▶ selective repeat



Terminology

▶ Acknowledgement (**ACK**)

- ▶ Receiver tells the sender when a frame is received
 - ▶ Selective acknowledgement (**SACK**)
 - Specifies set of frames received
 - ▶ Cumulative acknowledgement (**ACK**)
 - Have received specified frame and all previous

▶ Timeout (**TO**)

- ▶ Sender decides the frame (or ACK) was lost
- ▶ Sender can try again



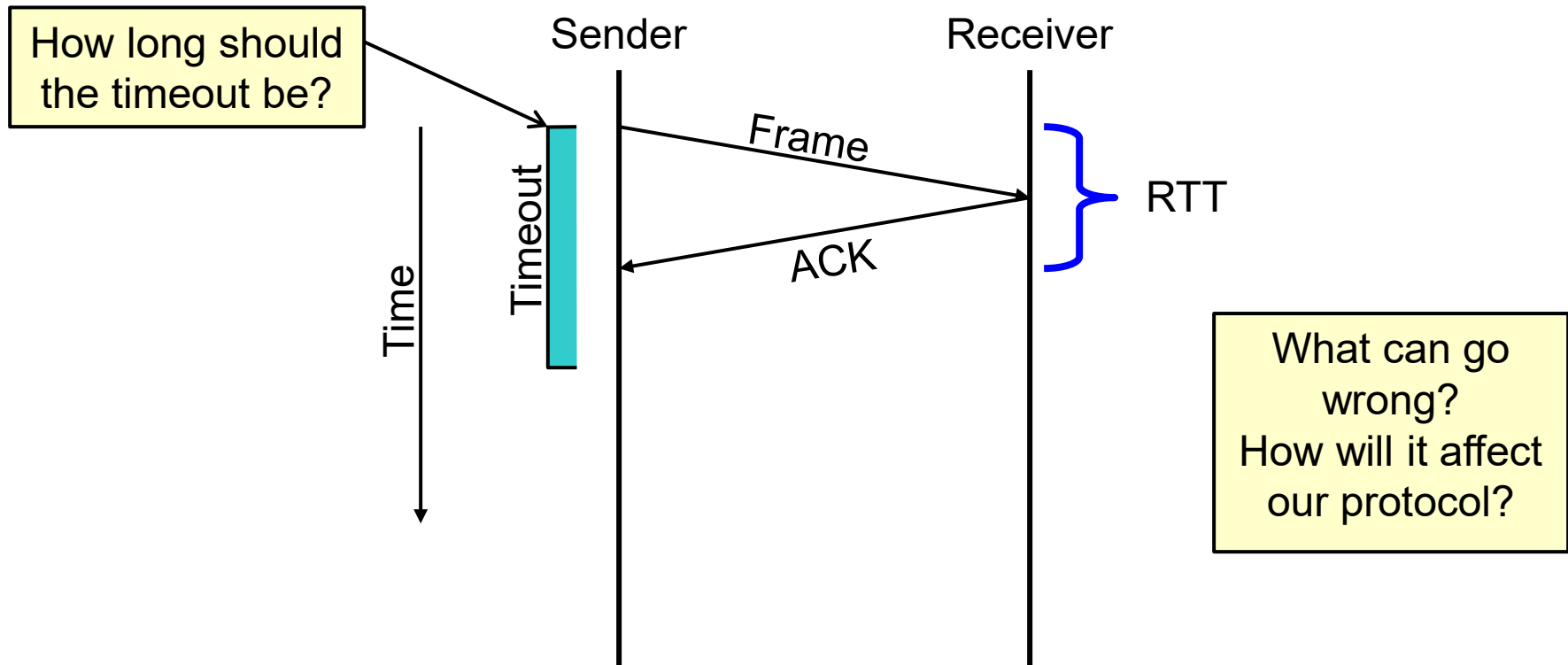
Stop-and-Wait

▶ Basic idea

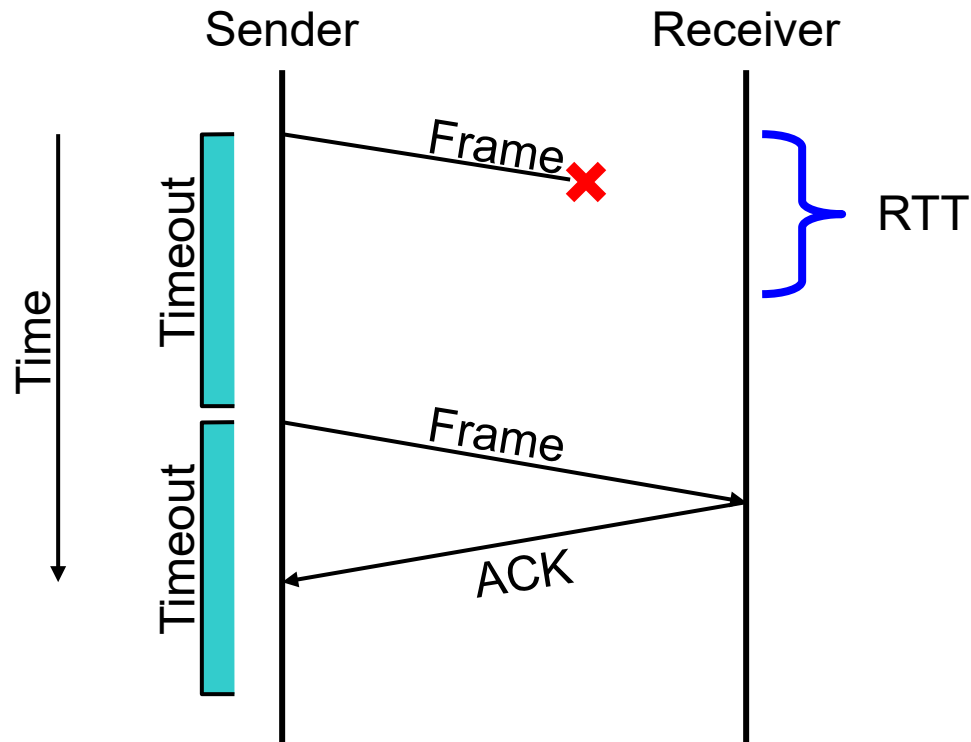
1. Send a frame
2. Wait for an ACK or TO
3. If TO, go to 1
4. If ACK, get new frame, go to 1



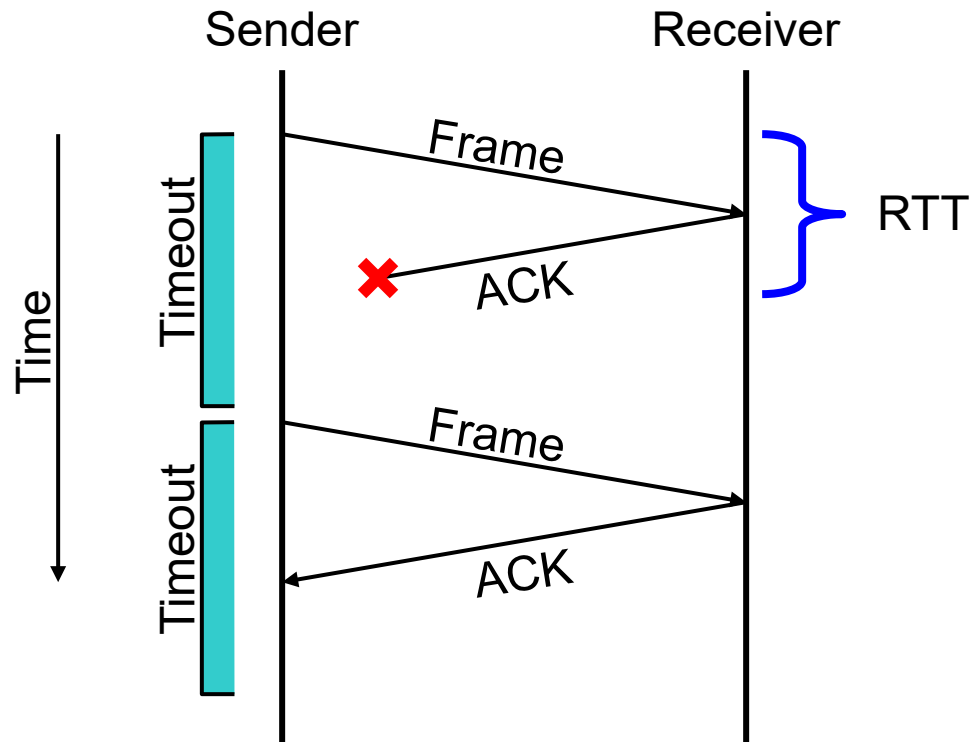
Stop-and-Wait: Success



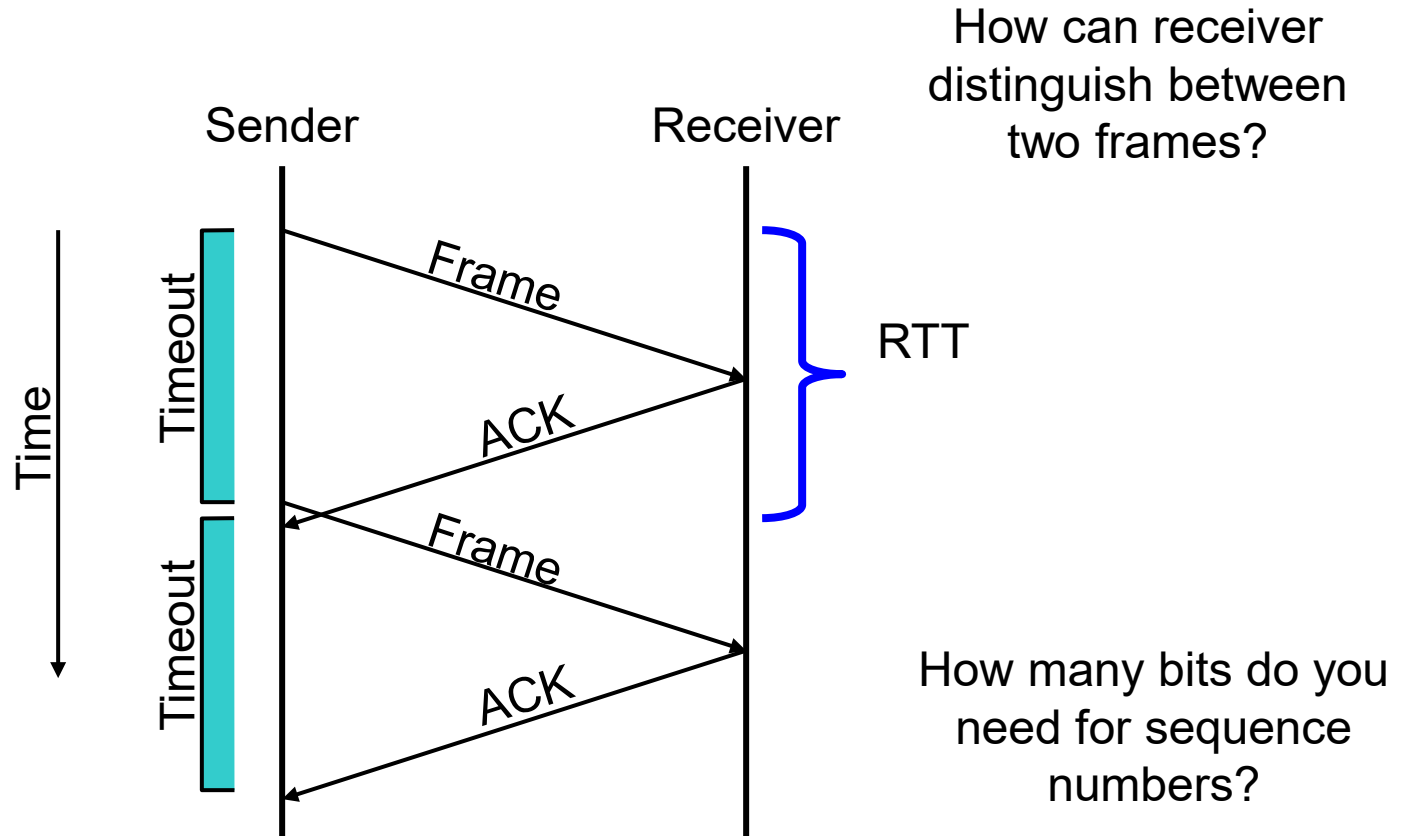
Stop-and-Wait: Lost Frame



Stop-and-Wait: Lost ACK



Stop-and-Wait: Delayed Frame



Stop-and-Wait

- ▶ **Goal**
 - ▶ Guaranteed, at-most-once delivery
- ▶ **Protocol Challenges**
 - ▶ Dropped frame/ACK
 - ▶ Duplicate frame/ACK
- ▶ **Requirements**
 - ▶ 1-bit sequence numbers (if physical network maintains order)
 - ▶ sender tracks frame ID to send
 - ▶ receiver tracks next frame ID expected

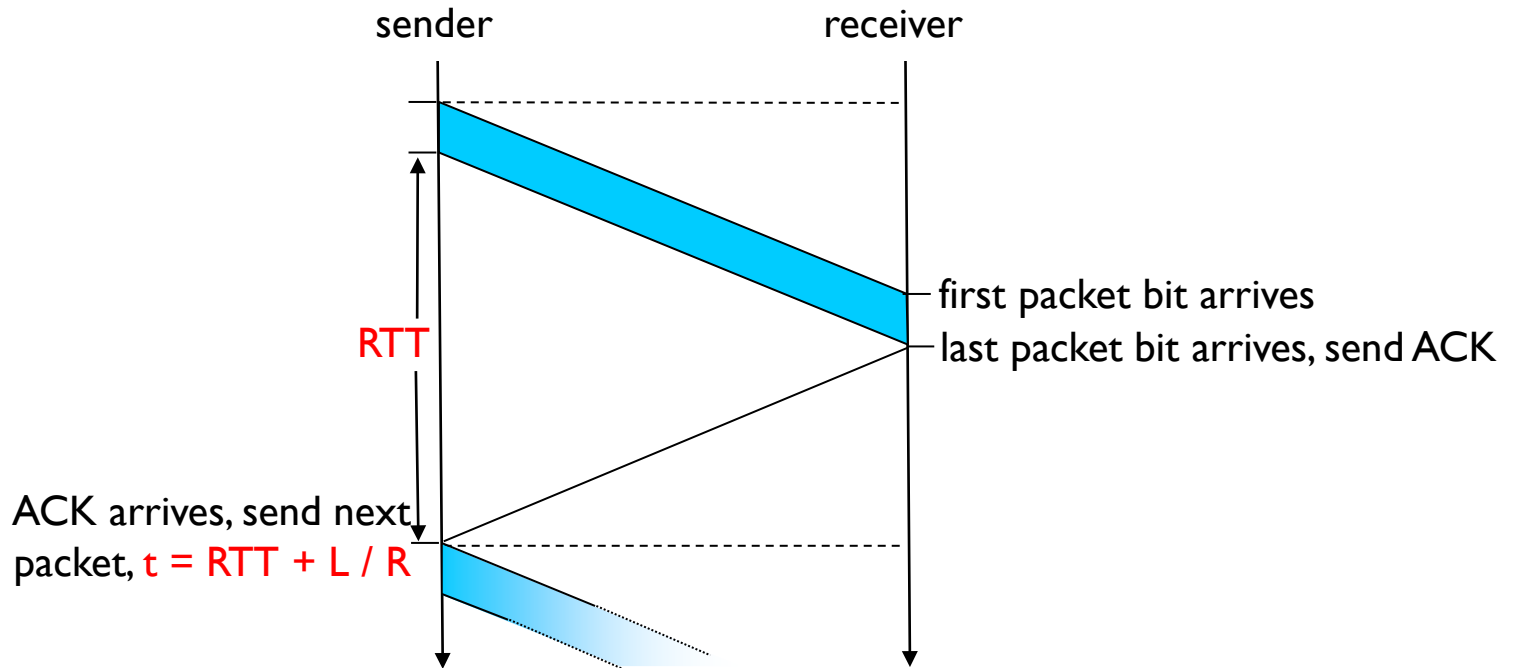


Stop-and-Wait

- ▶ **We have achieved**
 - ▶ Frames delivered reliably and in order
 - ▶ Is that enough?
- ▶ **Problem**
 - ▶ Only allows one outstanding frame
 - ▶ Does not keep the pipe full
 - ▶ **Example**
 - ▶ 100ms RTT
 - ▶ One frame per RTT = 1KB
 - ▶ $1024 \times 8 \times 10 = 81920$ kbps
 - ▶ Regardless of link bandwidth!



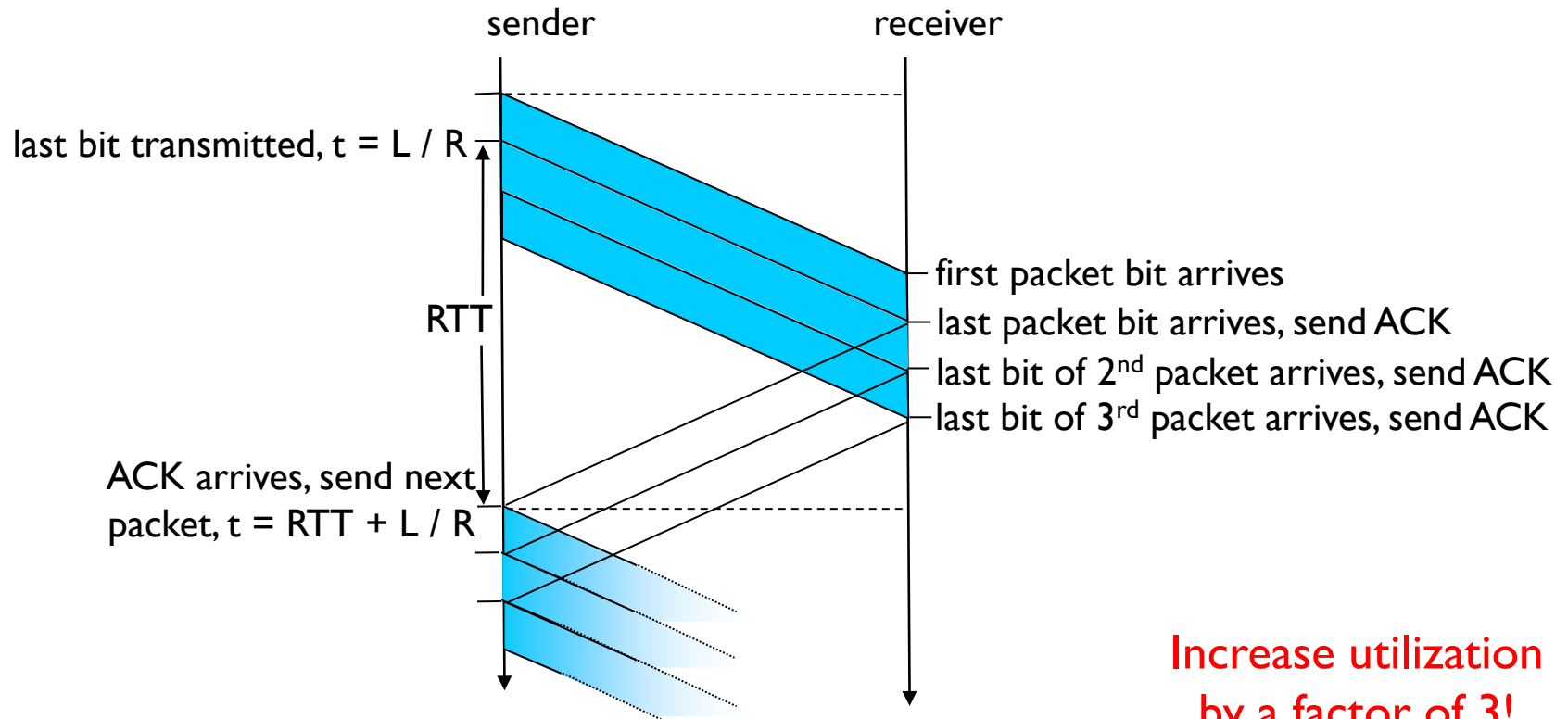
Stop-and-Wait



$$U_{\text{sender}} = \frac{L / R}{RTT + L / R}$$



Keeping the Pipe Full



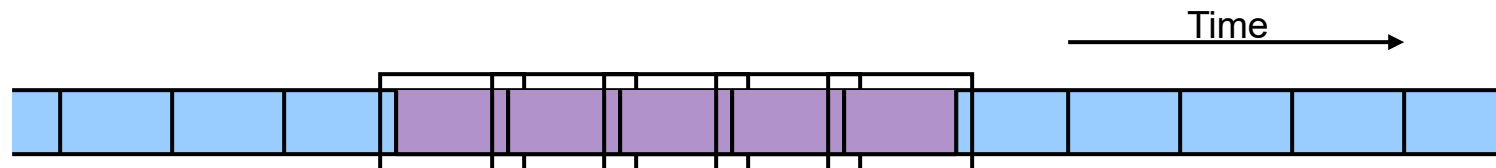
**Increase utilization
by a factor of 3!**

$$U_{\text{sender}} = \frac{3 * L / R}{RTT + L / R}$$



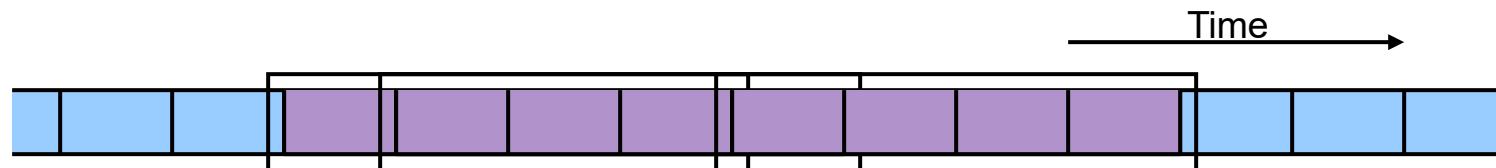
Concepts

- ▶ Consider an ordered stream of data frames
- ▶ Stop-and-Wait
 - ▶ Window of one frame
 - ▶ Slides along stream over time



Concepts

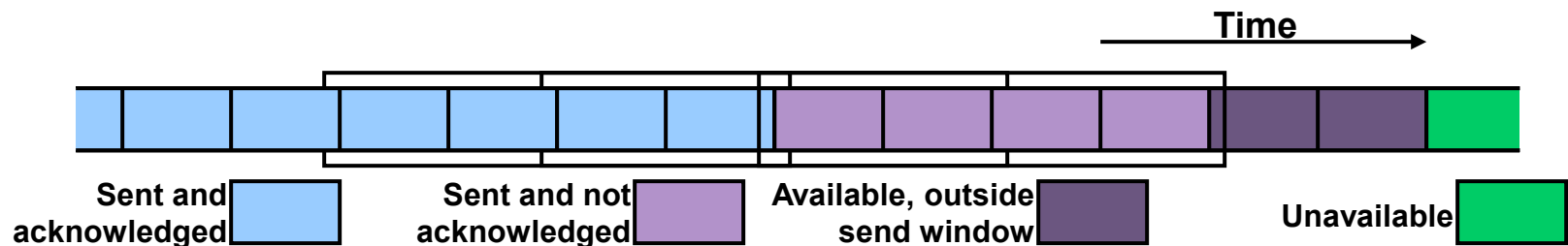
- ▶ Sliding Window Protocol
 - ▶ Multiple-frame send window
 - ▶ Multiple frame receive window



Sliding Window

▶ Send Window

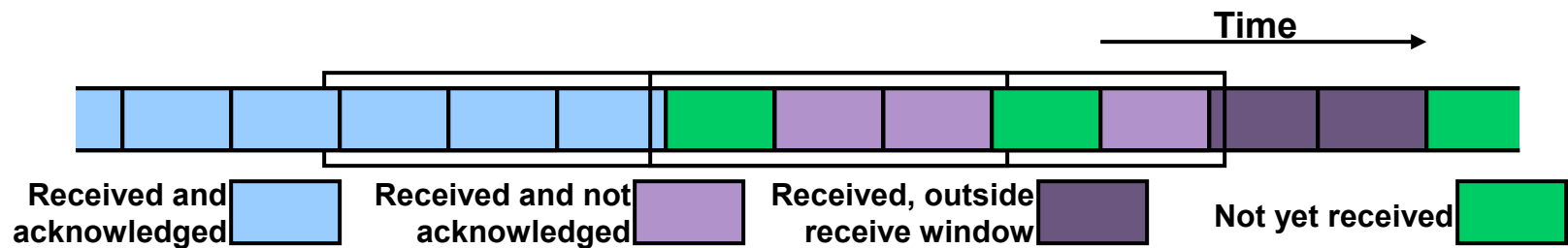
- ▶ Fixed length
- ▶ Starts at earliest unacknowledged frame
- ▶ Only frames in window are active



Sliding Window

▶ Receive Window

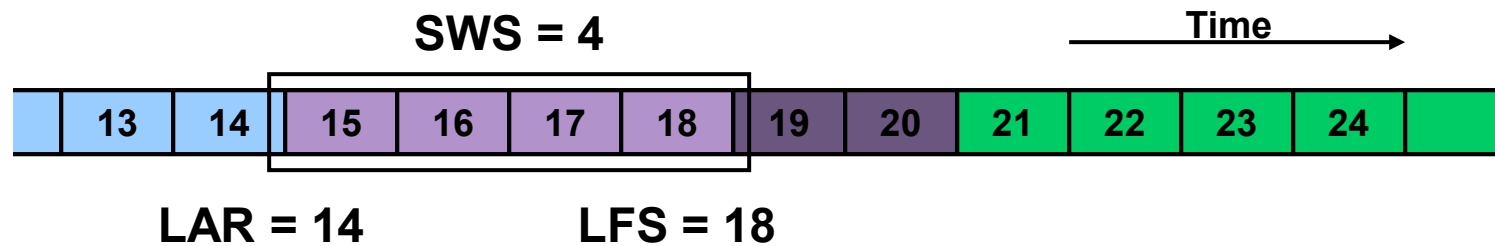
- ▶ Fixed length (unrelated to send window)
- ▶ Starts at earliest frame not received
- ▶ Only frames in window accepted



Sliding Window Terminology

▶ Sender Parameters

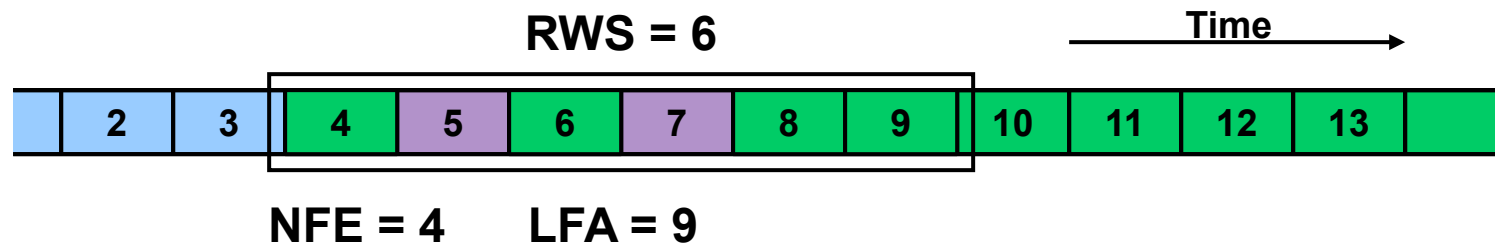
- ▶ Send Window Size (**SWS**)
- ▶ Last Acknowledgement Received (**LAR**)
- ▶ Last Frame Sent (**LFS**)



Sliding Window Terminology

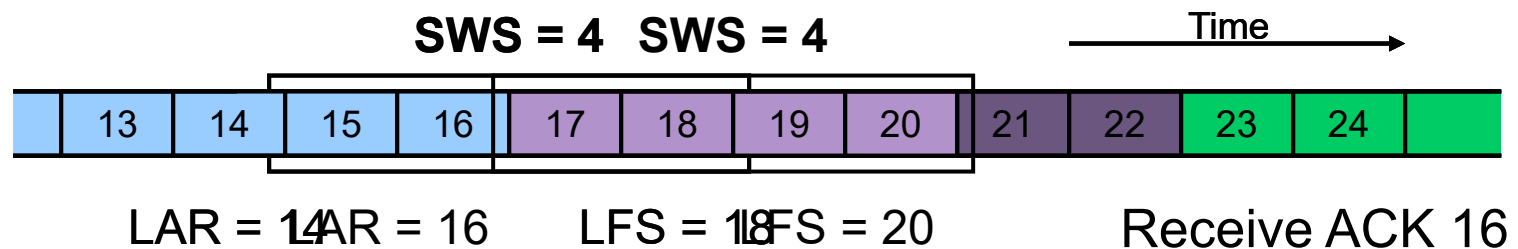
▶ Receiver Parameters

- ▶ Receive Window Size (**RWS**)
- ▶ Next Frame Expected (**NFE**)
- ▶ Last Frame Acceptable (**LFA**)



Sliding Window Details

- ▶ Sender Tasks
 - ▶ Assign sequence numbers
 - ▶ On ACK Arrival
 - ▶ Advance LAR
 - ▶ Slide window

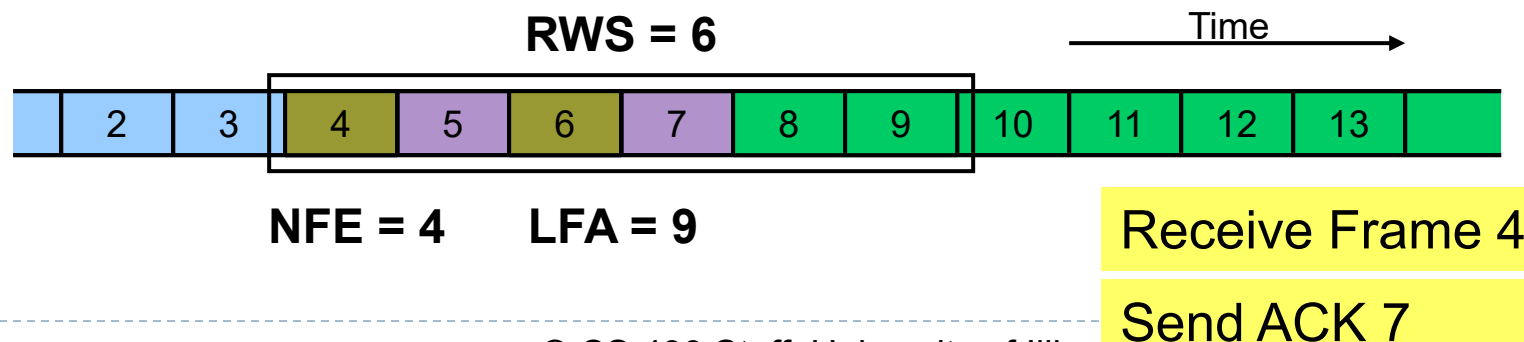


Sliding Window Details

▶ Receiver Tasks

▶ On Frame Arrival (N)

- ▶ Silently discard if outside of window
 - $N < NFE$ (NACK possible, too)
 - $N \geq NFE + RWS$
- ▶ Send cumulative ACK if within window

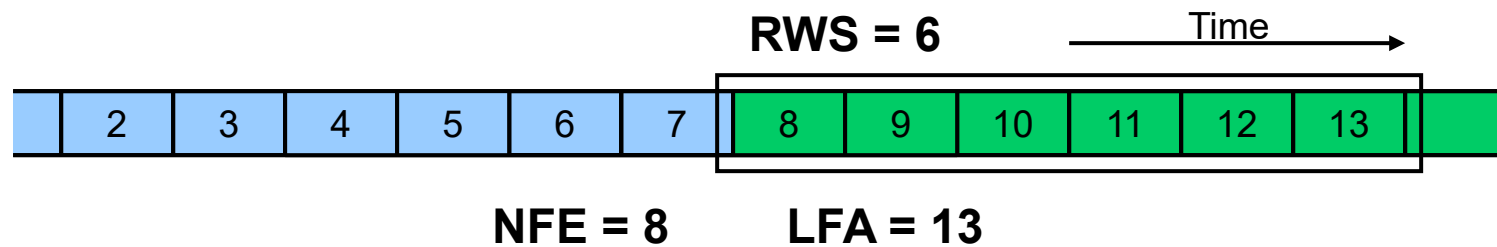


Sliding Window Details

▶ Receiver Tasks

▶ On Frame Arrival (N)

- ▶ Silently discard if outside of window
 - $N < NFE$ (NACK possible, too)
 - $N \geq NFE + RWS$
- ▶ Send cumulative ACK if within window



Sliding Window Details

- ▶ **Sequence number space**
 - ▶ Finite number, so wrap around
 - ▶ Need space larger than SWS (outstanding frames)
 - ▶ In fact, need twice as large



Window Sizes

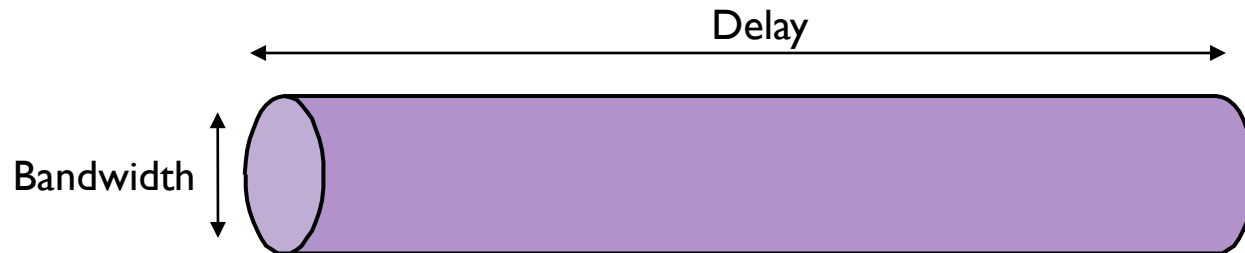
- ▶ **How big should we make SWS?**
 - ▶ Compute from delay \times bandwidth

- ▶ **How big should we make RWS?**
 - ▶ Depends on buffer capacity of receiver



Delay x Bandwidth Product - Revisited

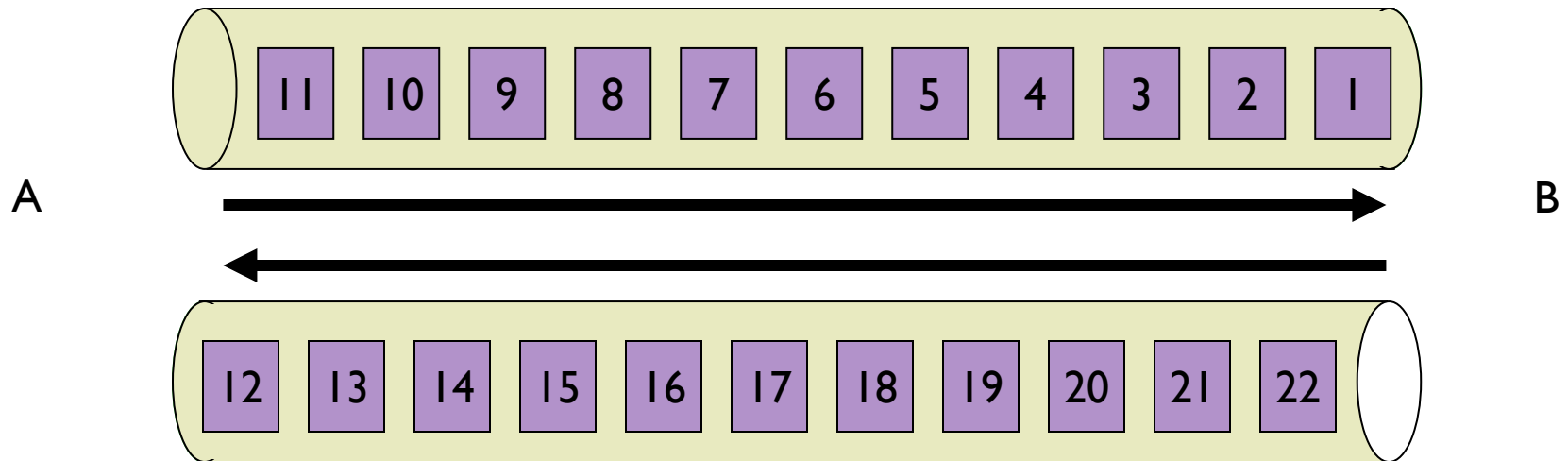
- ▶ Amount of data in “pipe”
 - ▶ channel = pipe
 - ▶ delay = length
 - ▶ bandwidth = area of a cross section
 - ▶ bandwidth x delay product = volume



Delay x Bandwidth Product

▶ Bandwidth x delay product

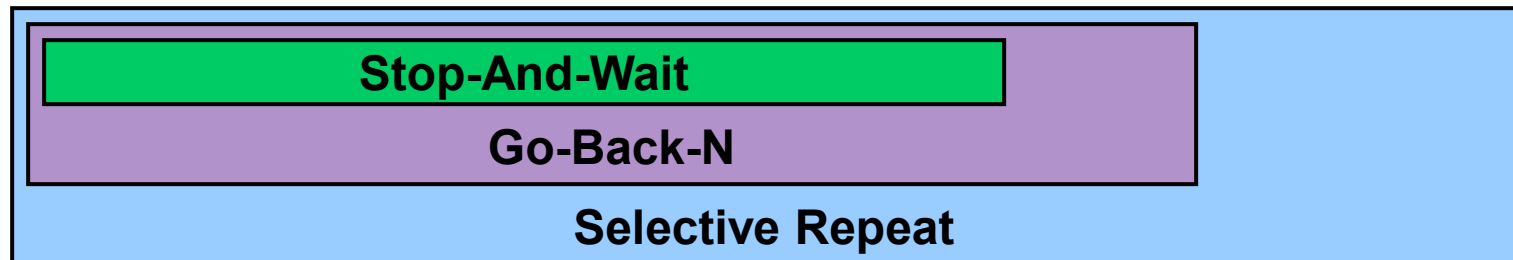
- ▶ How many bits the sender must transmit before the first bit arrives at the receiver if the sender keeps the pipe full
- ▶ Takes another one-way latency to receive a response from the receiver



ARQ Algorithm Classification

▶ Three Types:

- ▶ Stop-and-Wait: $SWS = 1$ $RWS = 1$
- ▶ Go-Back-N: $SWS = N$ $RWS = 1$
- ▶ Selective Repeat: $SWS = N$ $RWS = M$
 - ▶ Usually $M = N$

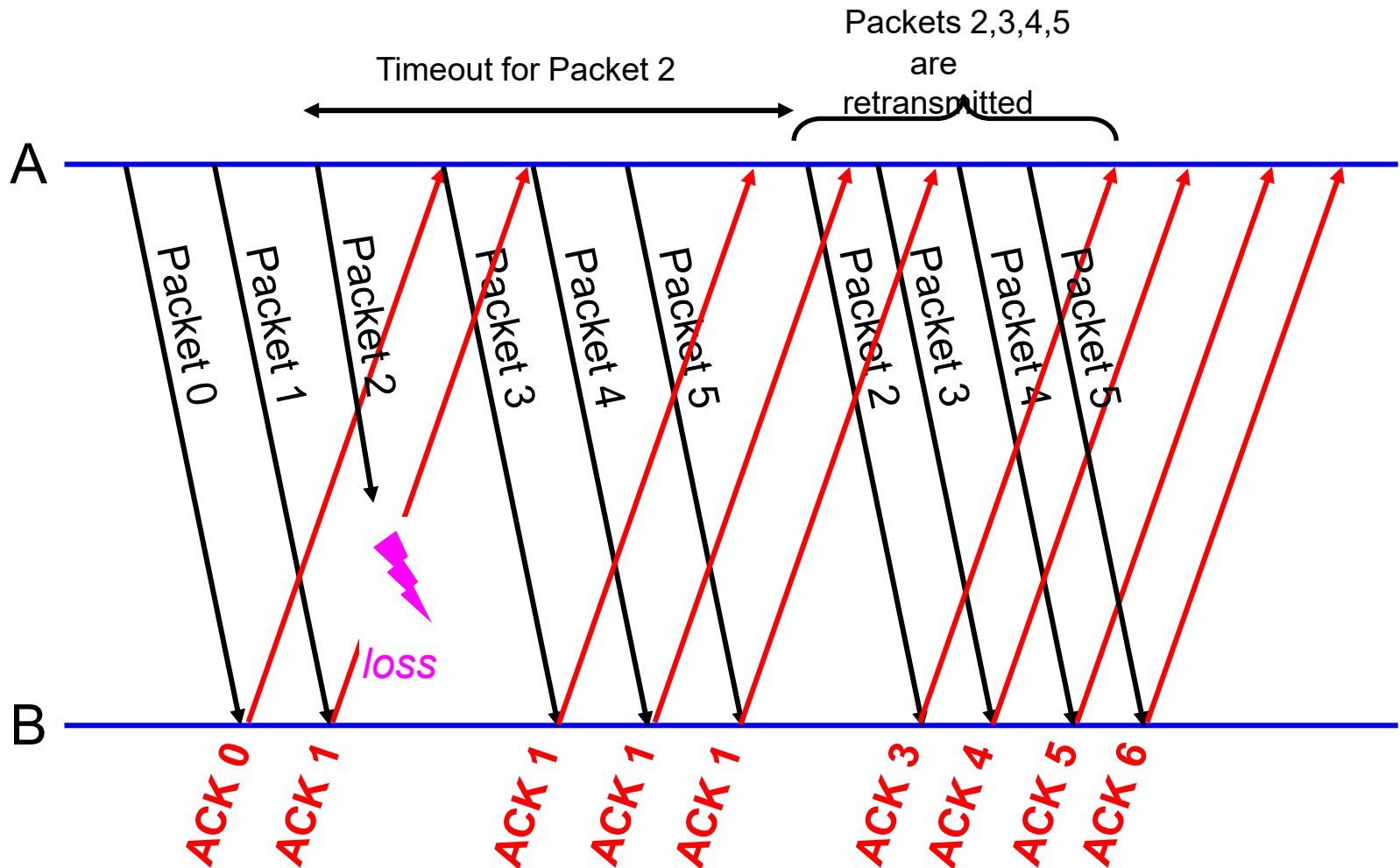


Sliding Window Variations: Go-Back-N

- ▶ $SWS = N, RWS = 1$
- ▶ Receiver only buffers one frame
- ▶ If a frame is lost, the sender may need to retransmit up to N frames
 - ▶ i.e., sender “goes back” N frames
- ▶ Variations
 - ▶ How long is the frame timeout?
 - ▶ Does receiver send NACK for out-of-sequence frame?



Go-Back-N: Cumulative ACKs

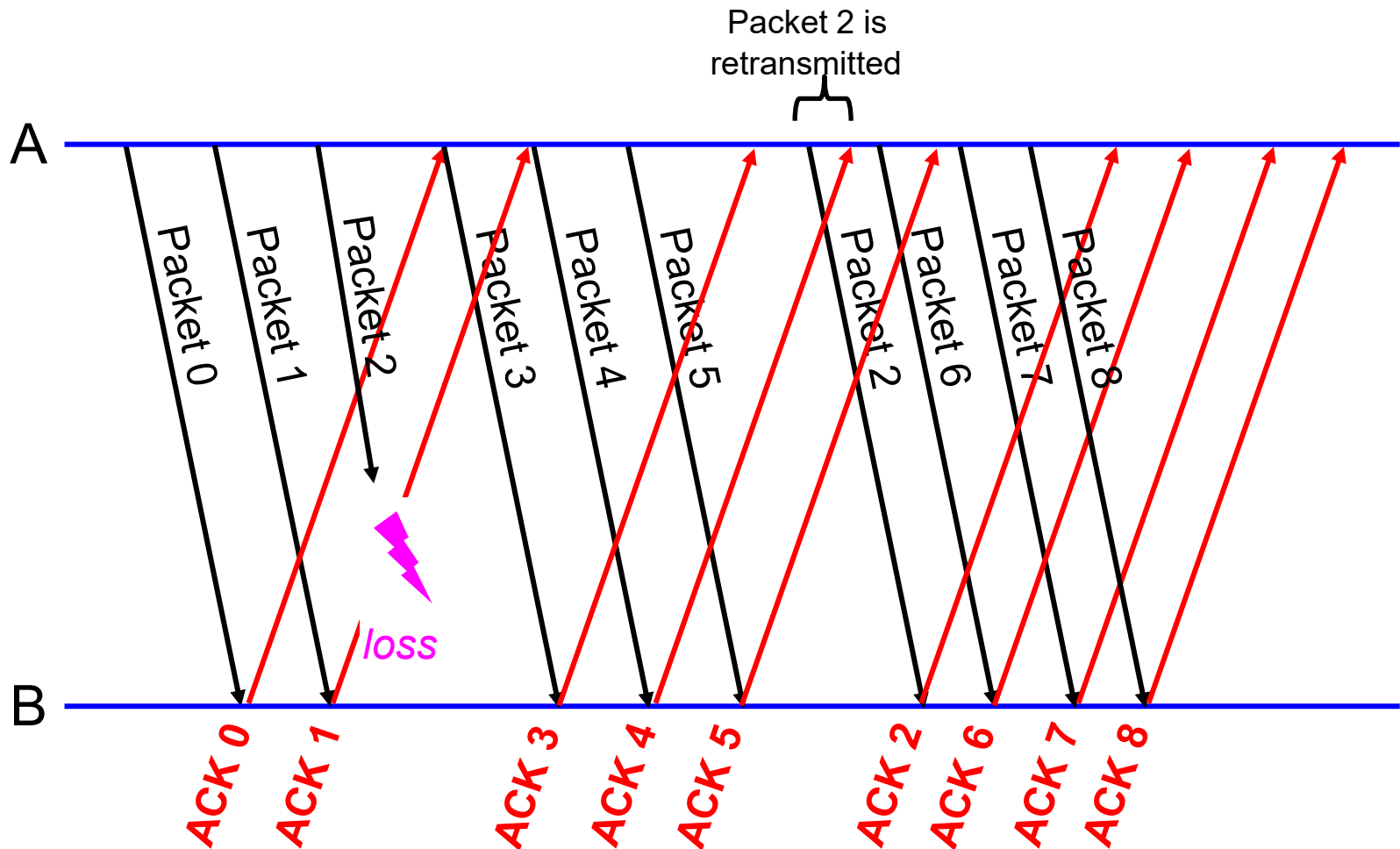


Sliding Window Variations: Selective Repeat

- ▶ $SWS = N, RWS = M$
- ▶ Receiver individually acknowledges all correctly received frames
 - ▶ Buffers up to M frames, as needed, for eventual in-order delivery to upper layer
- ▶ If a frame is lost, sender must only resend
 - ▶ Frames lost within the receive window
- ▶ Variations
 - ▶ How long is the frame timeout?
 - ▶ Use cumulative or per-frame ACK?
 - ▶ Does protocol adapt timeouts?
 - ▶ Does protocol adapt SWS and/or RWS?



Selective Repeat



Roles of a Sliding Window Protocol

- ▶ **Reliable delivery on an unreliable link**
 - ▶ Core function
- ▶ **Preserve delivery order**
 - ▶ Controlled by the receiver
- ▶ **Flow control**
 - ▶ Allow receiver to throttle sender
- ▶ **Separation of Concerns**
 - ▶ Must be able to distinguish between different functions that are sometimes rolled into one mechanism

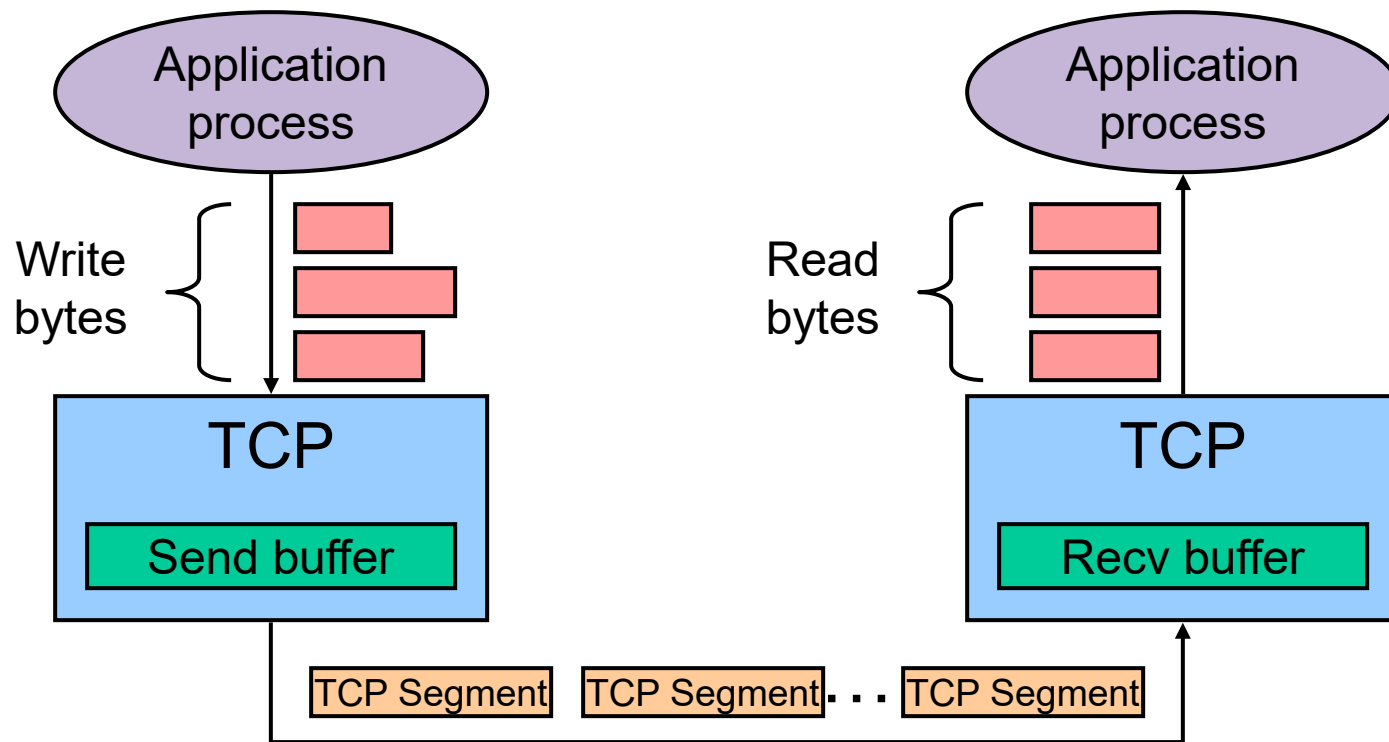


TCP Data Transport

- ▶ **Data broken into segments**
 - ▶ Limited by maximum segment size (MSS)
 - ▶ Defaults to 352 bytes
 - ▶ Negotiable during connection setup
 - ▶ Typically set to
 - ▶ MTU of directly connected network – size of TCP and IP headers
- ▶ **Three events cause a segment to be sent**
 - ▶ \geq MSS bytes of data ready to be sent
 - ▶ Explicit PUSH operation by application
 - ▶ Periodic timeout

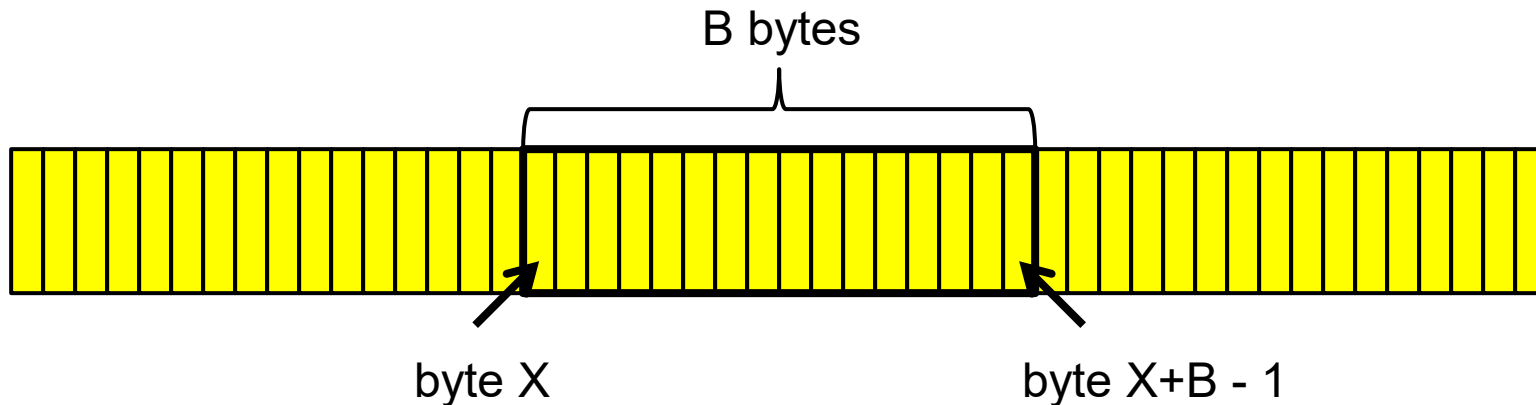


TCP Byte Stream



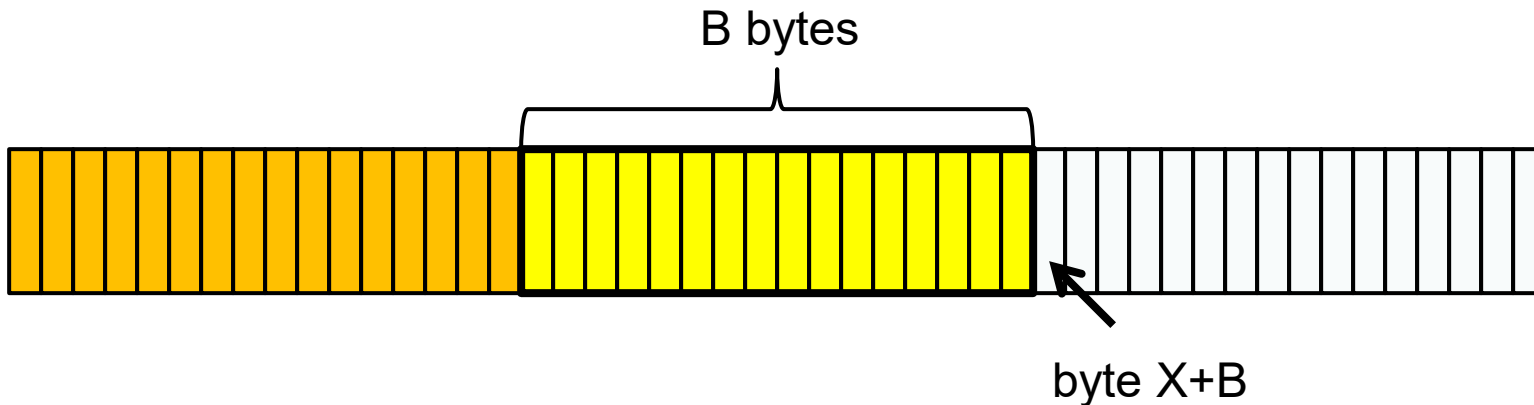
ACKing and Sequence Numbers

- ▶ Sender sends packet
 - ▶ Data starts with sequence number X
 - ▶ Packet contains B bytes
 - ▶ $X, X+1, X+2, \dots, X+B-1$



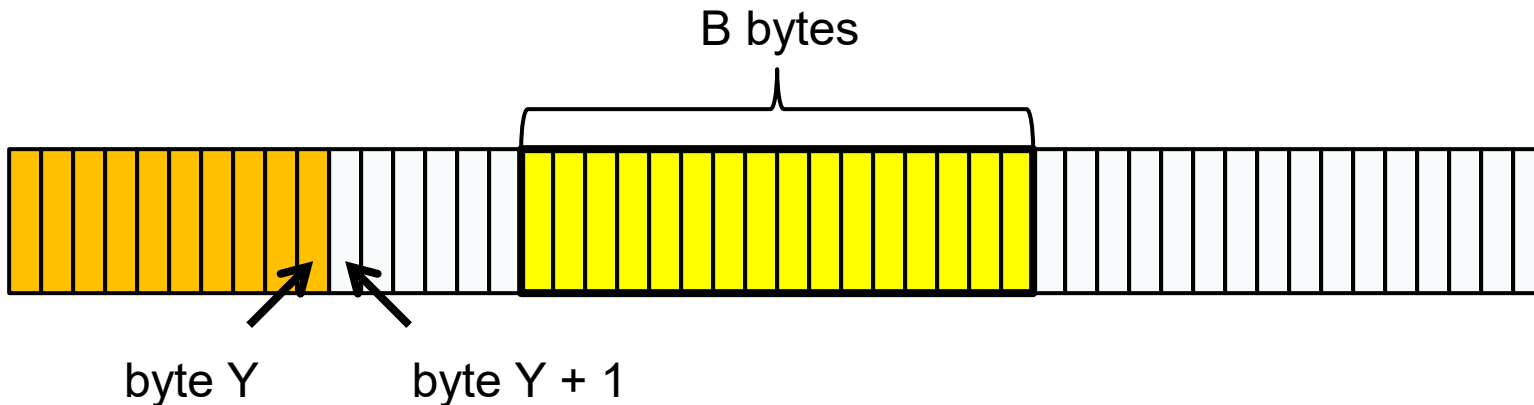
ACKing and Sequence Numbers

- ▶ Upon receipt of packet, receiver sends an ACK
 - ▶ If all data prior to X already received:
 - ▶ ACK acknowledges $X+B$ (because that is next expected byte)



ACKing and Sequence Numbers

- ▶ Upon receipt of packet, receiver sends an ACK
 - ▶ If highest byte already received is some smaller value Y
 - ▶ ACK acknowledges $Y+1$
 - ▶ Even if this has been ACKed before



TCP Sliding Window Protocol

- ▶ **Sequence numbers**
 - ▶ Indices into byte stream
- ▶ **ACK sequence number**
 - ▶ Actually next byte expected as opposed to last byte received



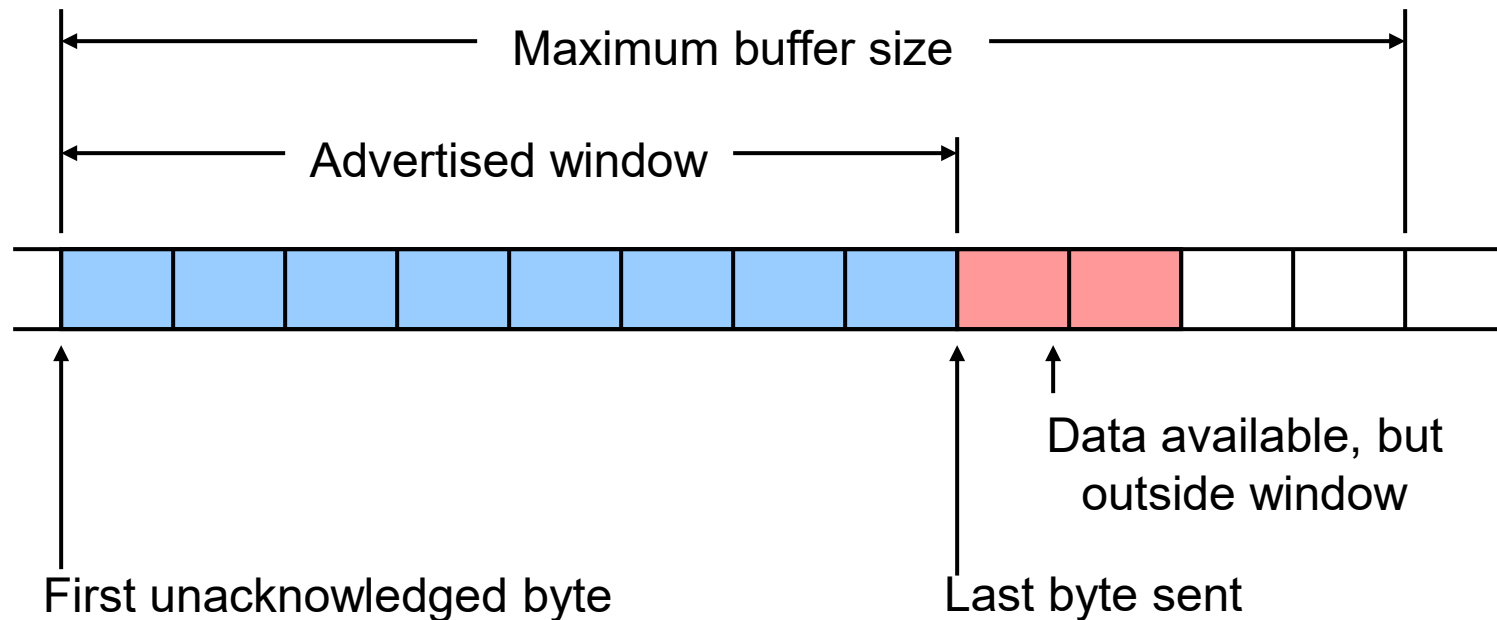
TCP Sliding Window Protocol

- ▶ **Advertised window**
 - ▶ Enables dynamic receive window size
- ▶ **Receive buffers**
 - ▶ Data ready for delivery to application until requested
 - ▶ Out-of-order data to maximum buffer capacity
- ▶ **Sender buffers**
 - ▶ Unacknowledged data
 - ▶ Unsent data out to maximum buffer capacity



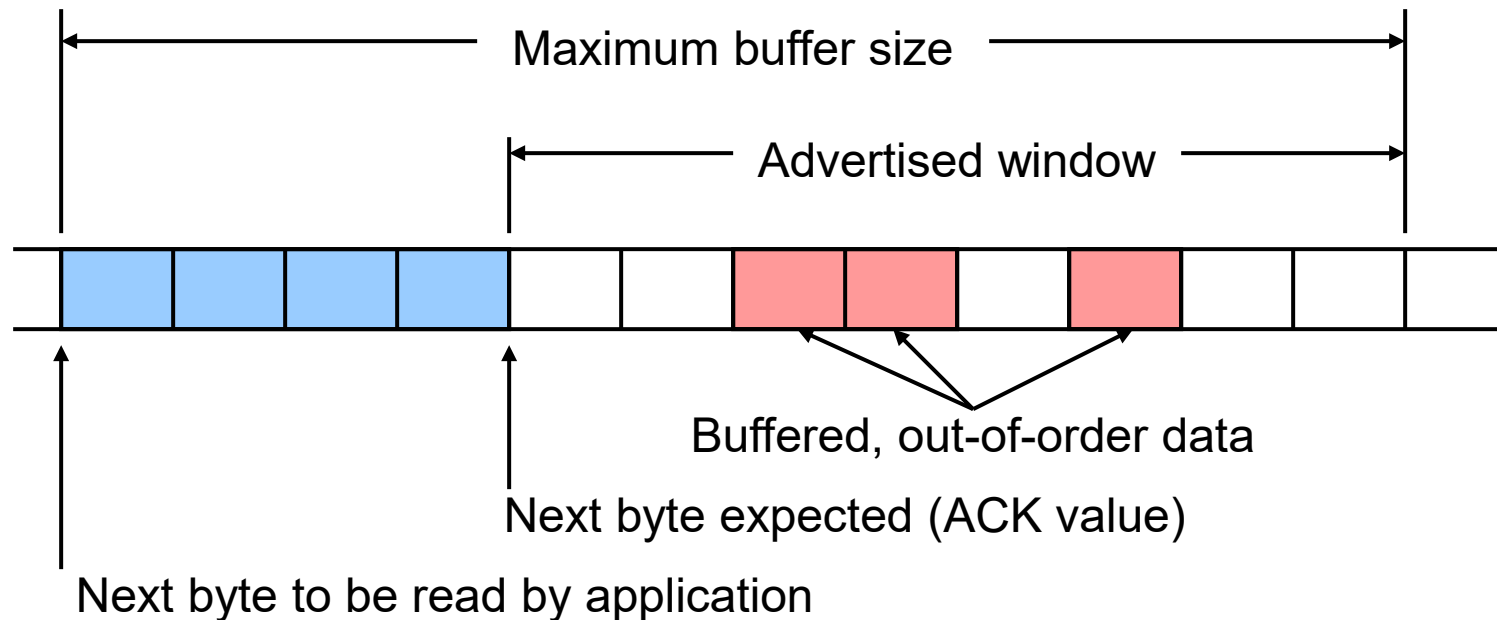
TCP Sliding Window Protocol – Sender Side

- ▶ `LastByteAcked` \leq `LastByteSent`
- ▶ `LastByteSent` \leq `LastByteWritten`
- ▶ Buffer bytes between `LastByteAcked` and `LastByteWritten`



TCP Sliding Window Protocol – Receiver Side

- ▶ `LastByteRead < NextByteExpected`
- ▶ `NextByteExpected <= LastByteRcvd + 1`
- ▶ Buffer bytes between `NextByteRead` and `LastByteRcvd`



Flow Control vs. Congestion Control

- ▶ **Flow control**
 - ▶ Preventing senders from overrunning the capacity of the receivers
- ▶ **Congestion control**
 - ▶ Preventing too much data from being injected into the network, causing switches or links to become overloaded
- ▶ **Which one does TCP provide?**
- ▶ **TCP provides both**
 - ▶ Flow control based on advertised window
 - ▶ Congestion control discussed later in class

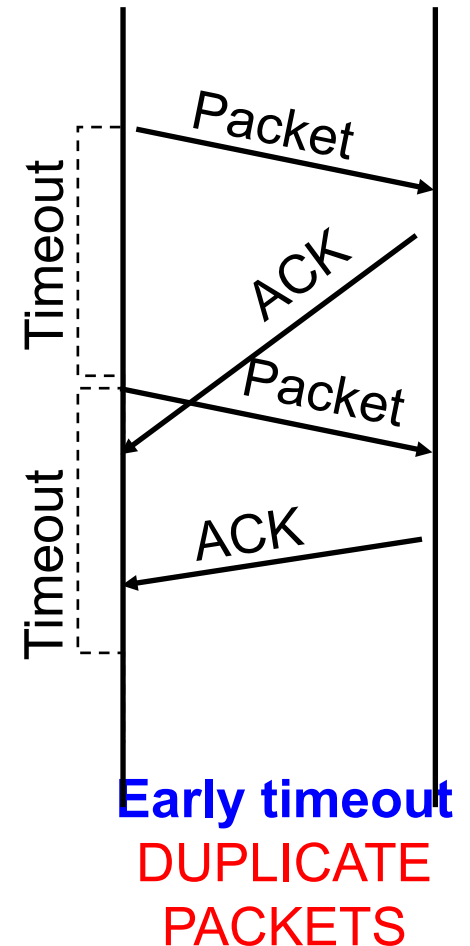
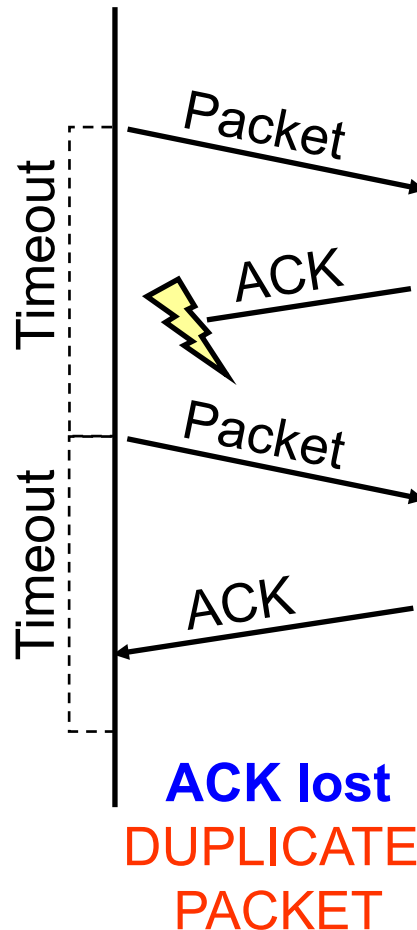
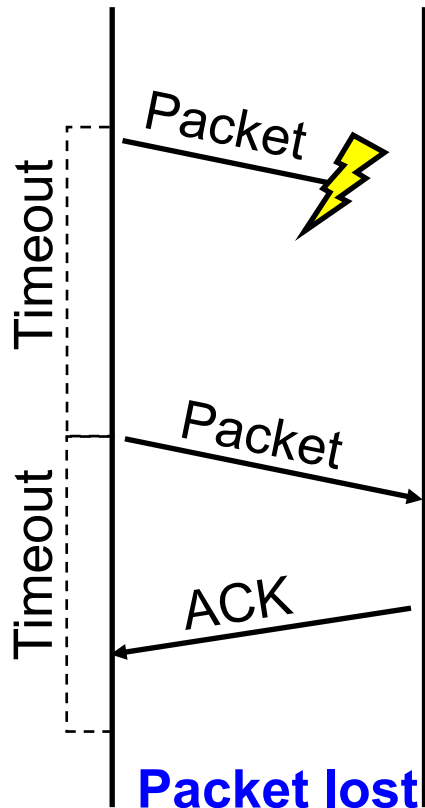


Advertised Window Limits Rate

- ▶ **W = window size**
 - ▶ Sender can send no faster than W/RTT bytes/sec
 - ▶ Receiver implicitly limits sender to rate that receiver can sustain
 - ▶ If sender is going too fast, window advertisements get smaller & smaller



Reasons for Retransmission



How Long Should Sender Wait?

- ▶ **Sender sets a timeout to wait for an ACK**
 - ▶ Too short
 - ▶ wasted retransmissions
 - ▶ Too long
 - ▶ excessive delays when packet lost



TCP Round Trip Time and Timeout

- ▶ How should TCP set its timeout value?
 - ▶ Longer than RTT
 - ▶ But RTT varies
 - ▶ Too short
 - ▶ Premature timeout
 - ▶ Unnecessary retransmissions
 - ▶ Too long
 - ▶ Slow reaction to segment loss

- ▶ Estimating RTT
 - ▶ SampleRTT
 - ▶ Measured time from segment transmission until ACK receipt
 - ▶ Will vary
 - ▶ Want smoother estimated RTT
 - ▶ Average several recent measurements
 - ▶ Not just current SampleRTT



TCP Congestion Control

▶ Idea

- ▶ Assumes best-effort network
- ▶ Each source determines network capacity for itself
- ▶ Implicit feedback
- ▶ ACKs pace transmission (self-clocking)

▶ Challenge

- ▶ Determining initial available capacity
- ▶ Adjusting to changes in capacity in a timely manner



TCP Congestion Control

▶ Basic idea

- ▶ Add notion of congestion window
- ▶ Effective window is smaller of
 - ▶ Advertised window (flow control)
 - ▶ Congestion window (congestion control)
- ▶ Changes in congestion window size
 - ▶ Slow increases to absorb new bandwidth
 - ▶ Quick decreases to eliminate congestion

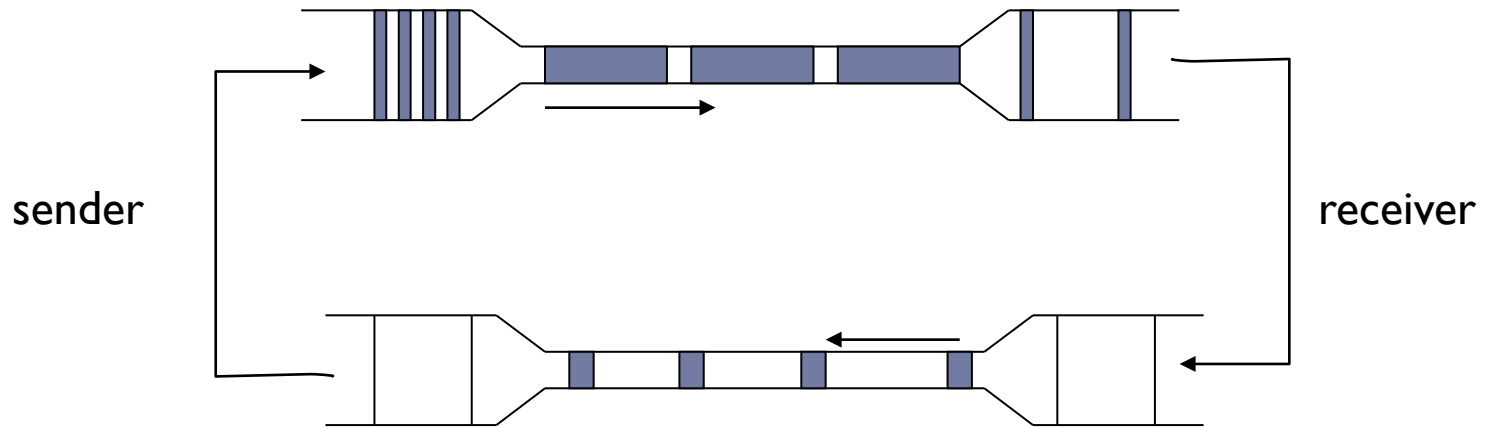


TCP Congestion Control

- ▶ **Specific strategy**

- ▶ Self-clocking

- ▶ Send data only when outstanding data ACK'd
 - ▶ Equivalent to send window limitation mentioned



TCP Congestion Control

- ▶ **Specific strategy**

- ▶ **Self-clocking**

- ▶ Send data only when outstanding data ACK'd
 - ▶ Equivalent to send window limitation mentioned

- ▶ **Growth**

- ▶ Add one maximum segment size (MSS) per congestion window of data ACK'd
 - ▶ It's really done this way, at least in Linux:
 - see `tcp_cong_avoid` in `tcp_input.c`.
 - Actually, every ack for new data is treated as an MSS ACK'd
 - ▶ Known as additive increase



TCP Congestion Control

- ▶ **Specific strategy (continued)**
 - ▶ Decrease
 - ▶ Cut window in half when timeout occurs
 - ▶ In practice, set window = window / 2
 - ▶ Known as multiplicative decrease
 - ▶ Additive increase, multiplicative decrease (AIMD)



Additive Increase/ Multiplicative Decrease

▶ Tools

- ▶ React to observance of congestion
- ▶ Probe channel to detect more resources

▶ Observation

- ▶ On notice of congestion
 - ▶ Decreasing too slowly will not be reactive enough
- ▶ On probe of network
 - ▶ Increasing too quickly will overshoot limits



Additive Increase/ Multiplicative Decrease

- ▶ **New TCP state variable**
 - ▶ **CongestionWindow**
 - ▶ Similar to **AdvertisedWindow** for flow control
 - ▶ Limits how much data source can have in transit
 - ▶ **MaxWin = MIN(CongestionWindow, AdvertisedWindow)**
 - ▶ **EffWin = MaxWin - (LastByteSent - LastByteAcked)**
 - ▶ TCP can send no faster than the slowest component, network or destination
- ▶ **Idea**
 - ▶ Increase **CongestionWindow** when congestion goes down
 - ▶ Decrease **CongestionWindow** when congestion goes up



Additive Increase/ Multiplicative Decrease

▶ Question

- ▶ How does the source determine whether or not the network is congested?

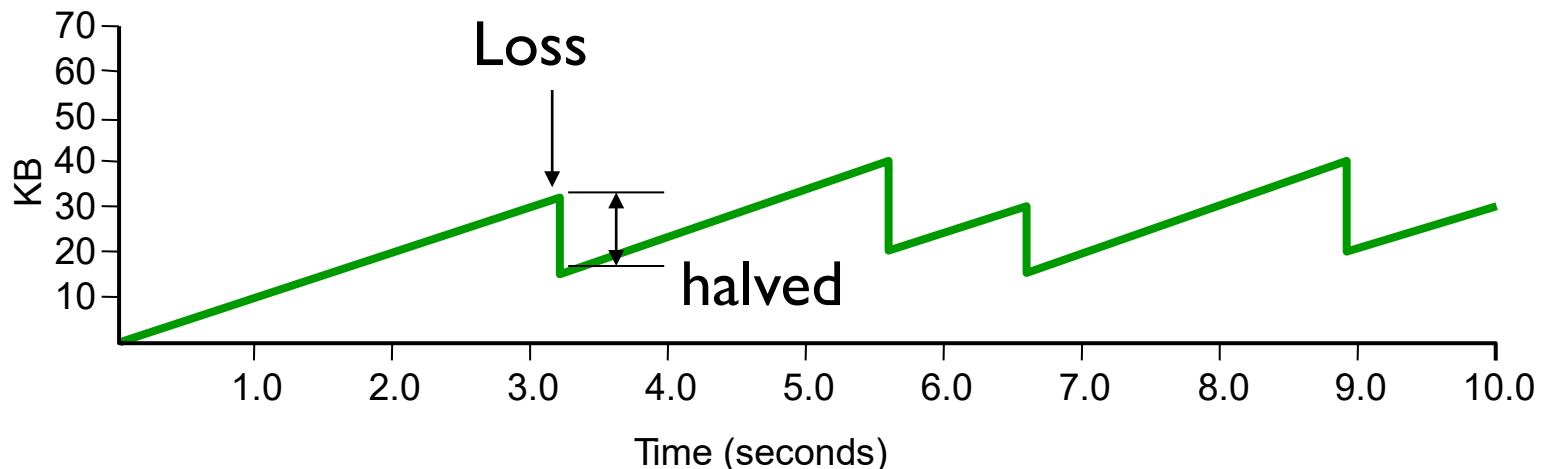
▶ Answer

- ▶ Timeout signals packet loss
- ▶ Packet loss is rarely due to transmission error (on wired lines)
- ▶ Lost packet implies congestion!



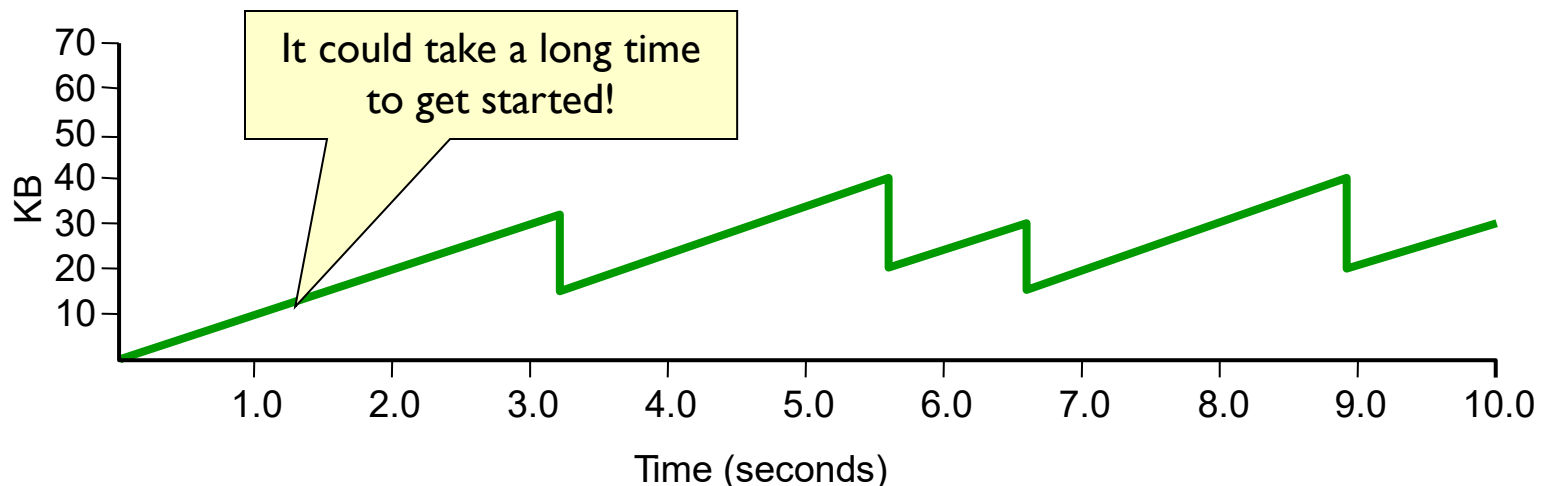
AIMD – Sawtooth Trace

- ▶ Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
 - ▶ Factor of 2
- ▶ TCP periodically probes for available bandwidth by increasing its rate



TCP Start Up Behavior

- ▶ How should TCP start sending data?
 - ▶ AIMD is good for channels operating at capacity
 - ▶ AIMD can take a long time to ramp up to full capacity from scratch



TCP Start Up Behavior

- ▶ How should TCP start sending data?
 - ▶ AIMD is good for channels operating at capacity
 - ▶ AIMD can take a long time to ramp up to full capacity from scratch
 - ▶ Use Slow Start to increase window rapidly from a cold start



TCP Start Up Behavior: Slow Start

- ▶ Initialization of the congestion window
 - ▶ Congestion window should start small
 - ▶ Avoid congestion due to new connections
 - ▶ Start at 1 MSS,
 - ▶ Initially, CWND is 1 MSS
 - ▶ Initial sending rate is MSS/RTT
 - ▶ Reset to 1 MSS with each timeout
 - ▶ timeouts are coarse-grained, $\sim 1/2$ sec

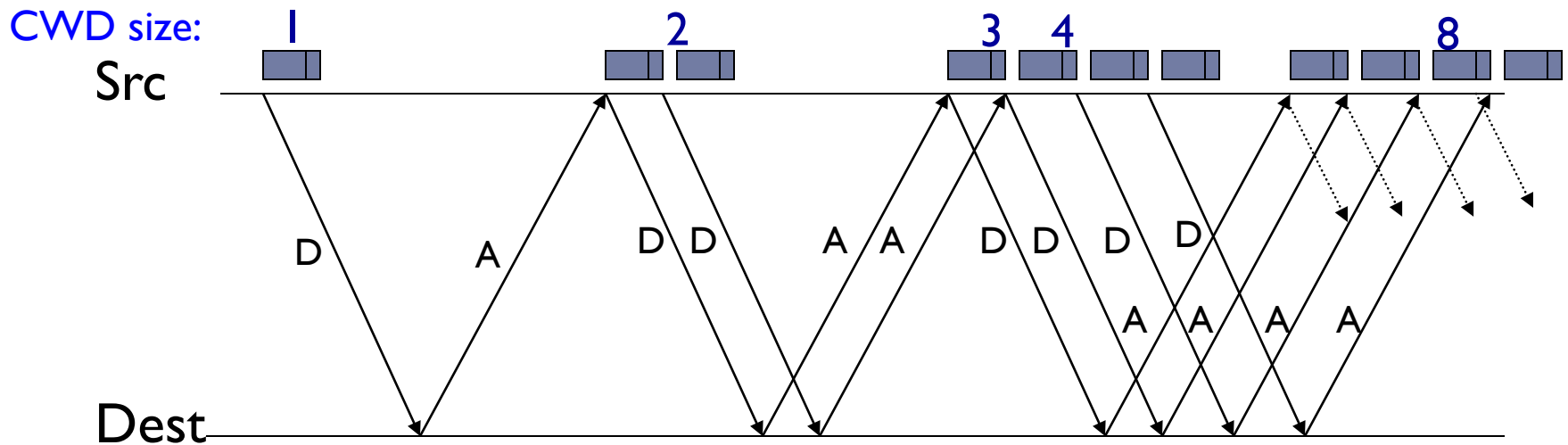


TCP Start Up Behavior: Slow Start

- ▶ Growth of the congestion window
- ▶ Linear growth could be pretty wasteful
 - ▶ Might be much less than the actual bandwidth
 - ▶ Linear increase takes a long time to accelerate
- ▶ Start slow but then grow fast
 - ▶ Sender starts at a slow rate
 - ▶ Increase the rate exponentially
 - ▶ Until the first loss event



Slow Start Example



Slow Start

- ▶ **Used**

- ▶ When first starting connection
- ▶ When connection times out

- ▶ **Why is it called slow-start?**

- ▶ Because TCP originally had no congestion control mechanism
- ▶ The source would just start by sending a whole window's worth of data



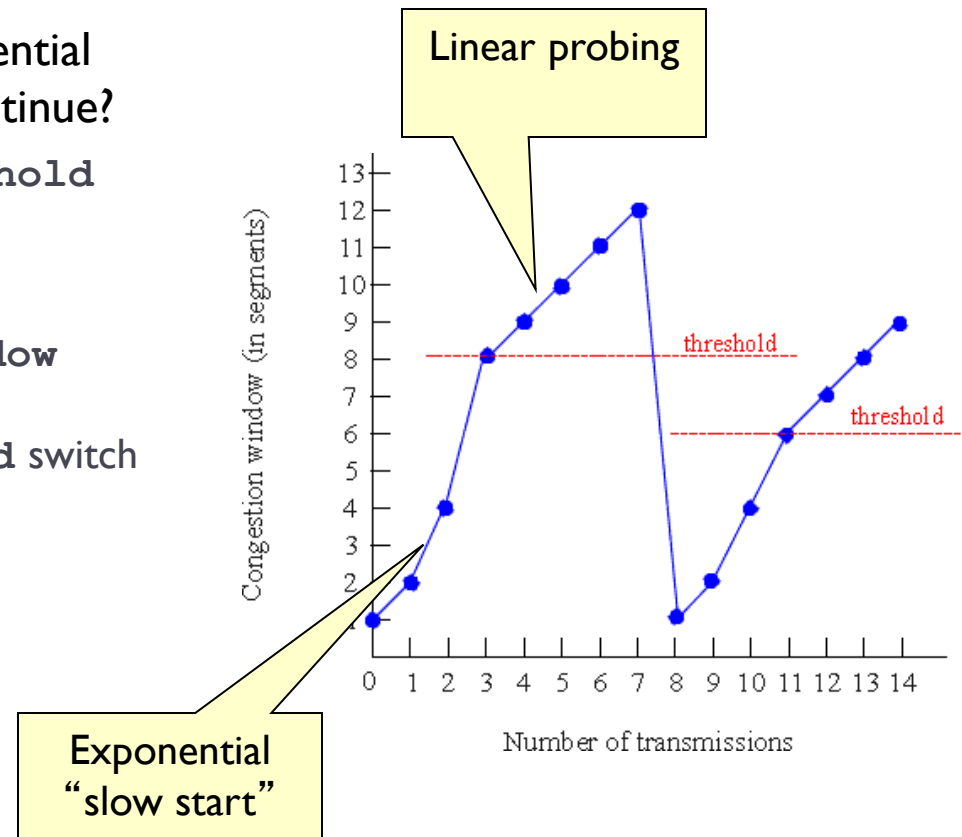
TCP Congestion Control

- ▶ **Maintain threshold window size**
 - ▶ Threshold value
 - ▶ Initially set to maximum window size
 - ▶ Set to 1/2 of current window on timeout
 - ▶ Use multiplicative increase
 - ▶ When congestion window smaller than threshold
 - ▶ Double window for each window ACK'd
- ▶ **In practice**
 - ▶ Increase congestion window by one MSS for each ACK of new data (or N bytes for N bytes)



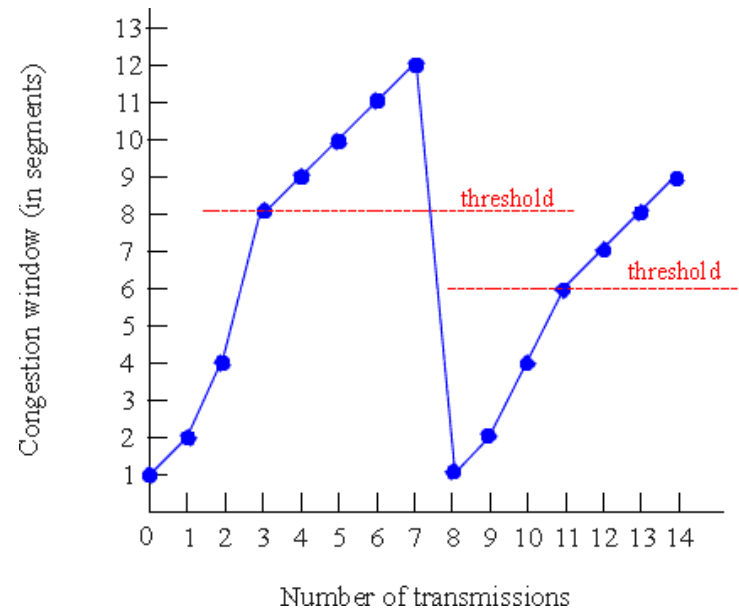
Slow Start

- ▶ How long should the exponential increase from slow start continue?
 - ▶ Use `CongestionThreshold` as target window size
 - ▶ Estimates network capacity
 - ▶ When `CongestionWindow` reaches `CongestionThreshold` switch to additive increase



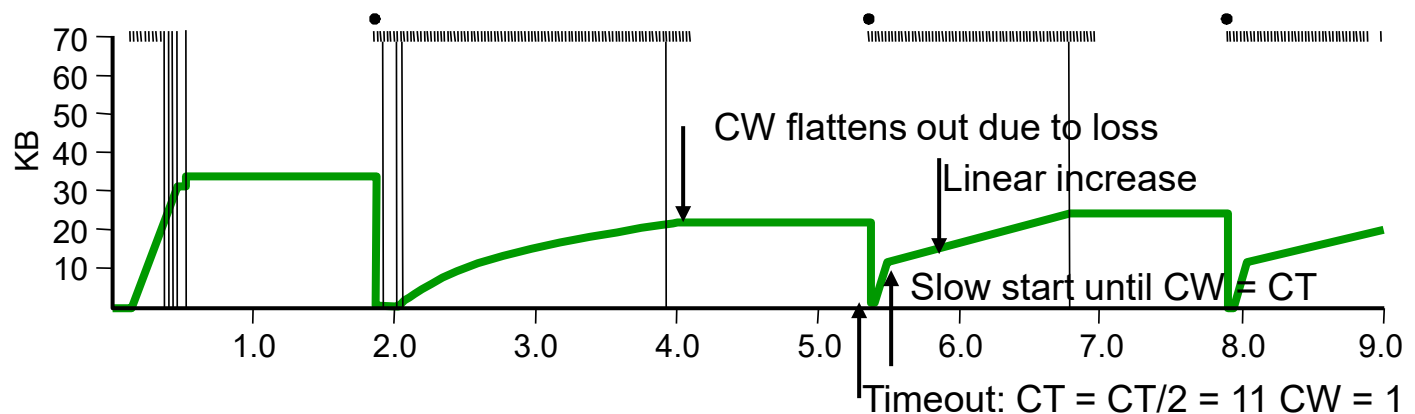
Slow Start

- ▶ Initial values
 - ▶ `CongestionThreshold = 8`
 - ▶ `CongestionWindow = 1`
- ▶ Loss after transmission 7
 - ▶ `CongestionWindow` currently 12
 - ▶ Set `CongestionThreshold = CongestionWindow/2`
 - ▶ Set `CongestionWindow = 1`



Slow Start

▶ Example trace of **CongestionWindow**



■ Problem

- Have to wait for timeout
- Can lose half **CongestionWindow** of data



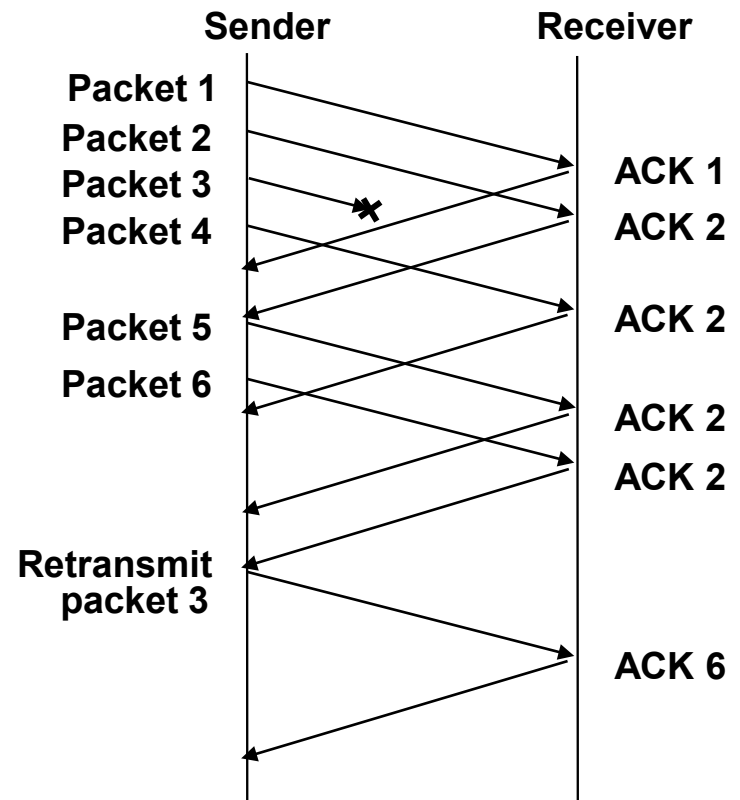
Fast Retransmit and Fast Recovery

▶ Problem

- ▶ Coarse-grain TCP timeouts lead to idle periods

▶ Solution

- ▶ Fast retransmit: use duplicate ACKs to trigger retransmission



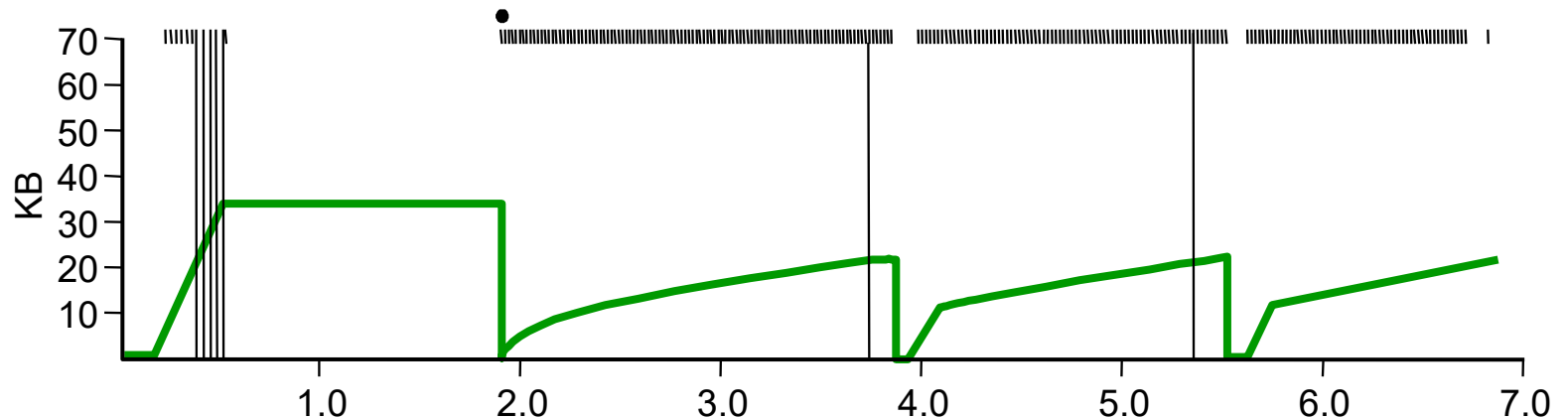
Fast Retransmit and Fast Recovery

- ▶ Send ACK for each segment received
- ▶ When duplicate ACK's received
 - ▶ Resend lost segment immediately
 - ▶ Do not wait for timeout
 - ▶ In practice, retransmit on 3rd duplicate
- ▶ **Fast recovery**
 - ▶ When fast retransmission occurs, skip slow start
 - ▶ Congestion window becomes $1/2$ previous
 - ▶ Start additive increase immediately



Fast Retransmit and Fast Recovery

► Results



- Fast Recovery
 - Bypass slow start phase
 - Increase immediately to one half last successful **CongestionWindow (ssthresh)**



TCP Congestion Window Trace

