CS 439: Wireless Networking

Transport Layer – dealing with errors and unreliability

© CS 439 Staff, University of Illinois Fall 2026

Reliable Transmission

- Suppose error protection identifies valid and invalid packets

- Can we make the channel appear reliable?
  - Insure packet delivery
  - Maintain packet order
  - Provide reliability at full link capacity





Reliable Transmission

- 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





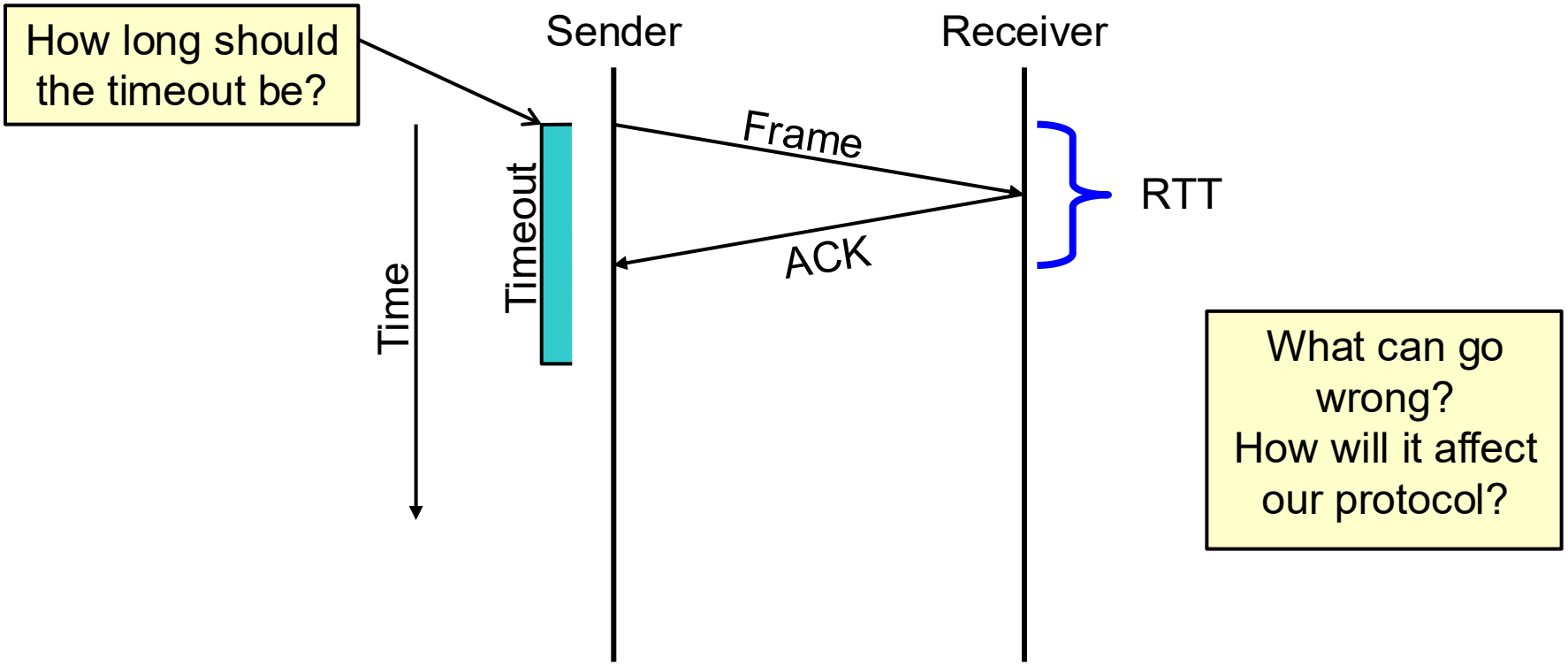
Terminology

- 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

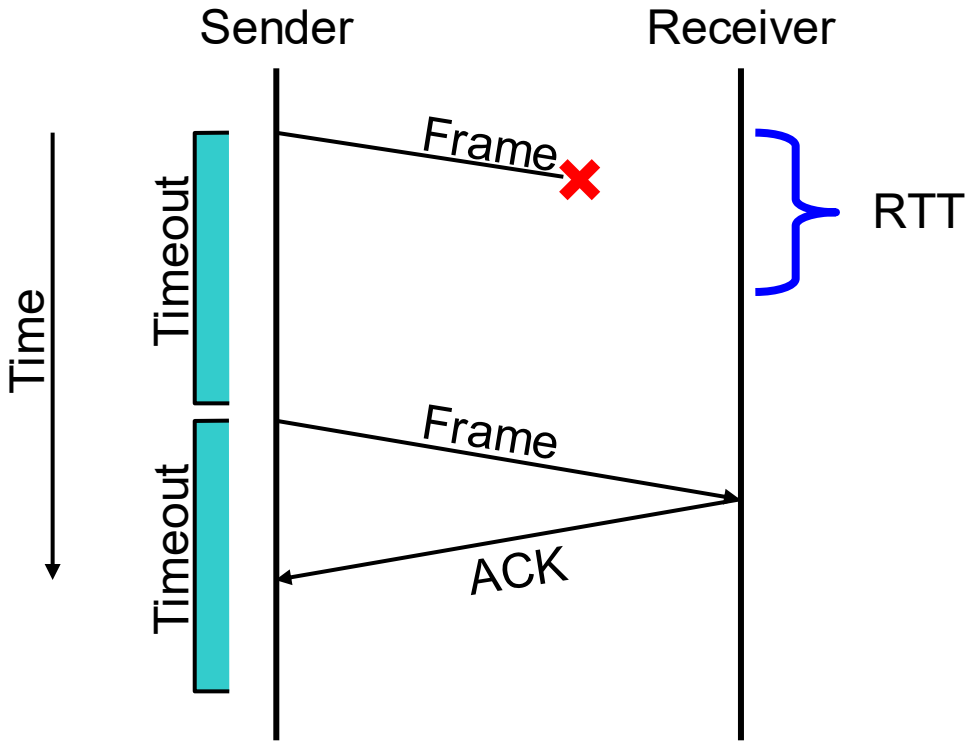
Stop-and-Wait

- 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

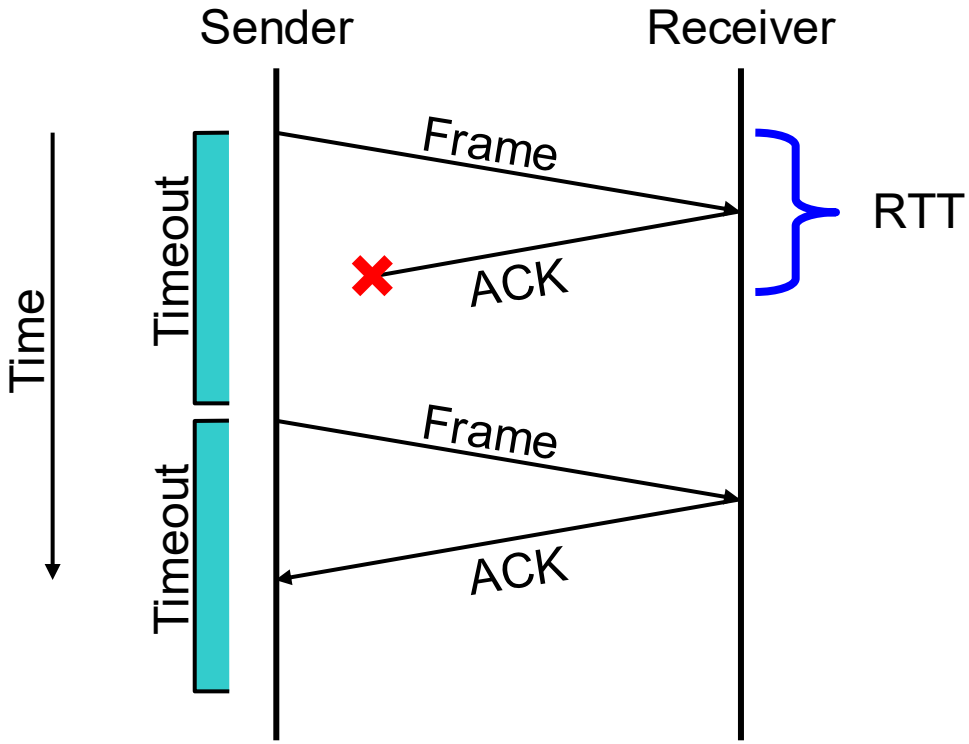
Stop-and-Wait: Success



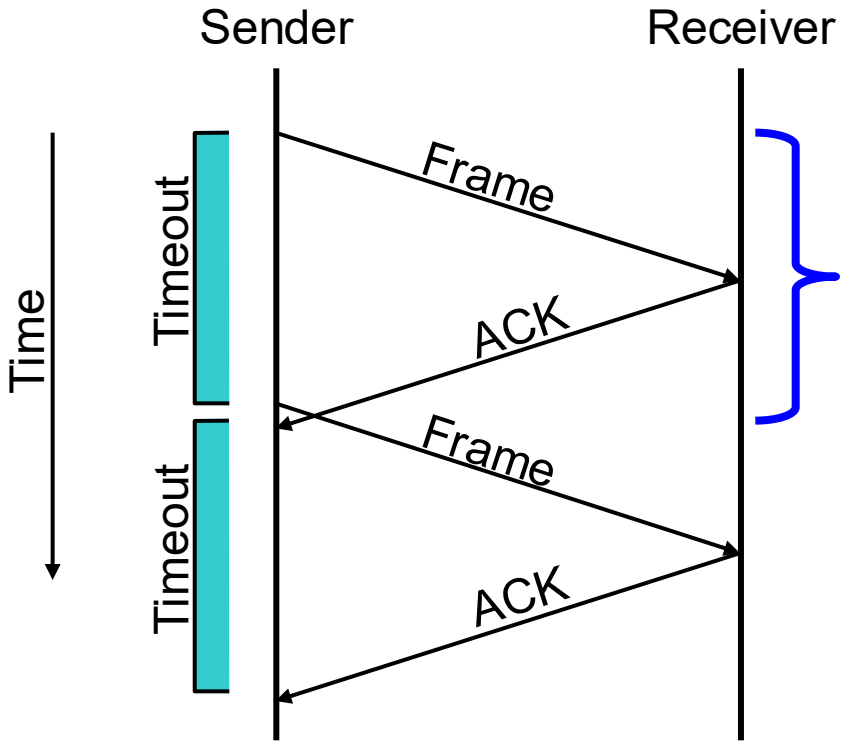
Stop-and-Wait: Lost Frame



Stop-and-Wait: Lost ACK



Stop-and-Wait: Delayed Frame



How can receiver distinguish between two frames?

How many bits do you need for sequence numbers?

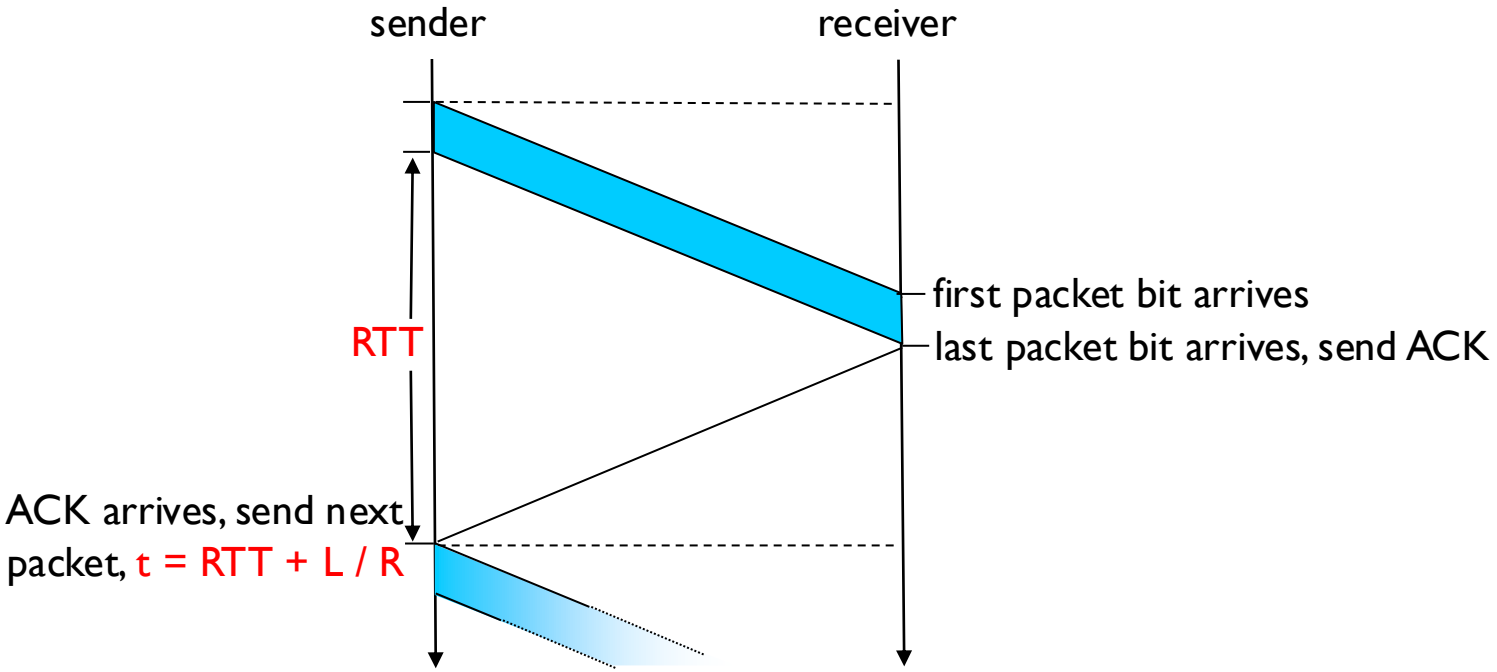
Stop-and-Wait

- 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 × 8 bits × 10 per second = 81,920 bps (~82 kbps)
    - Regardless of link bandwidth!



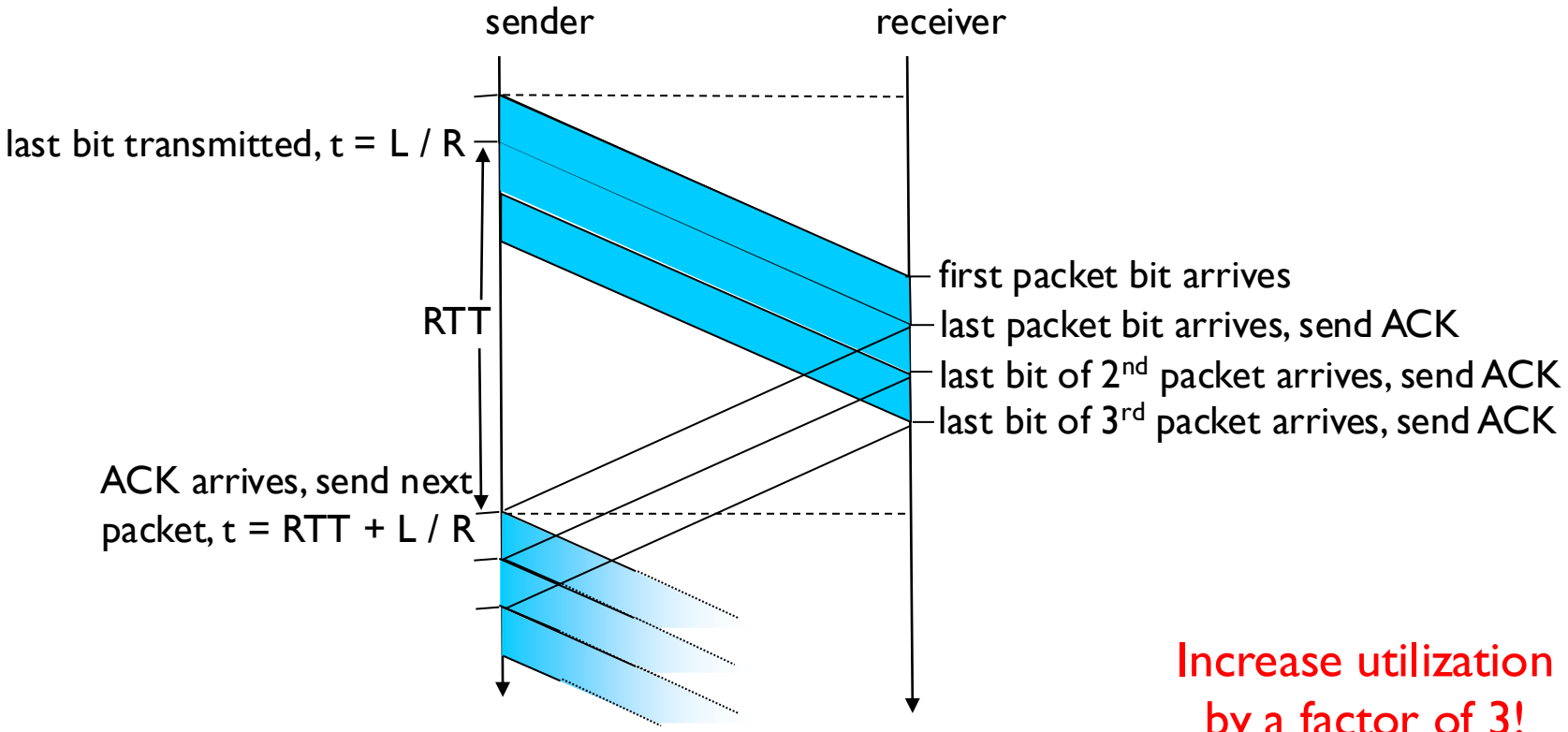


Stop-and-Wait



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

Keeping the Pipe Full

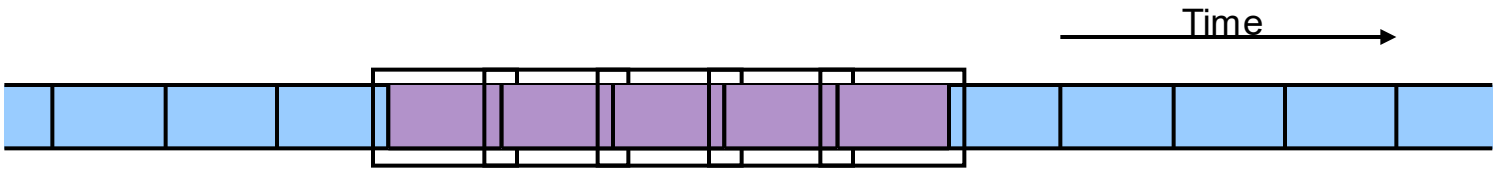


Increase utilization by a factor of 3!

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

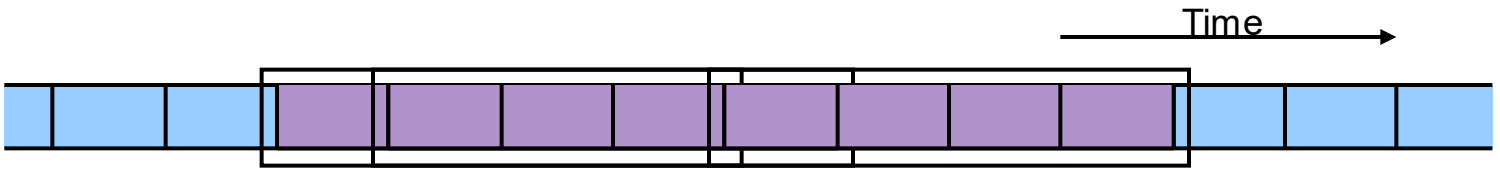
Reliability

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



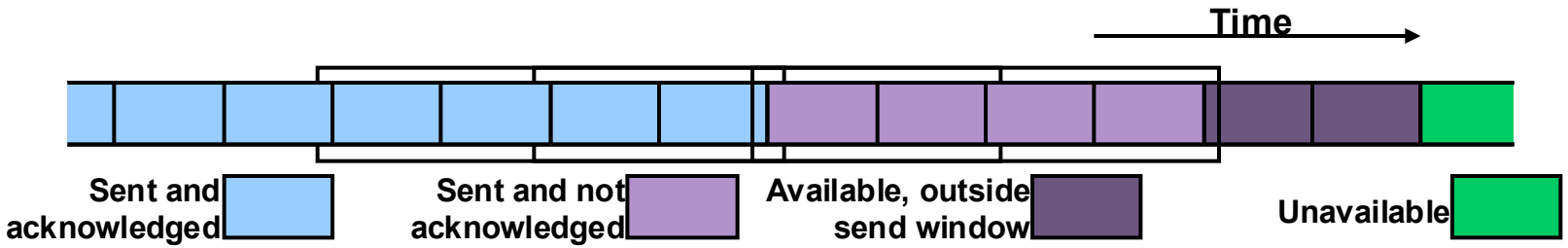
Reliability

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



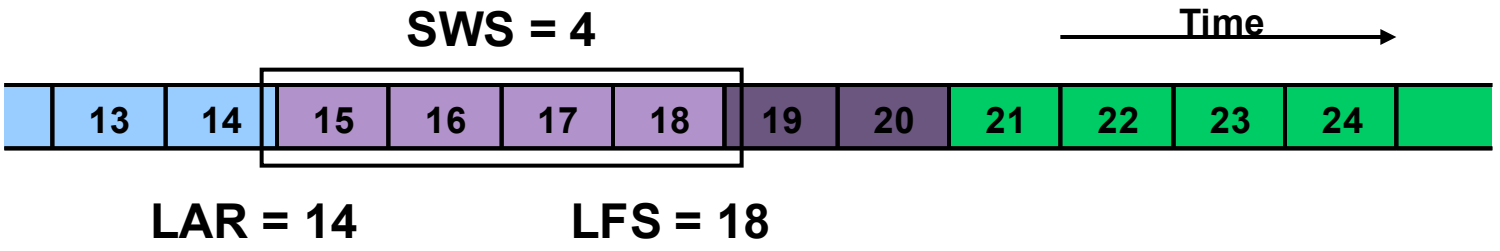
Sliding Window

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



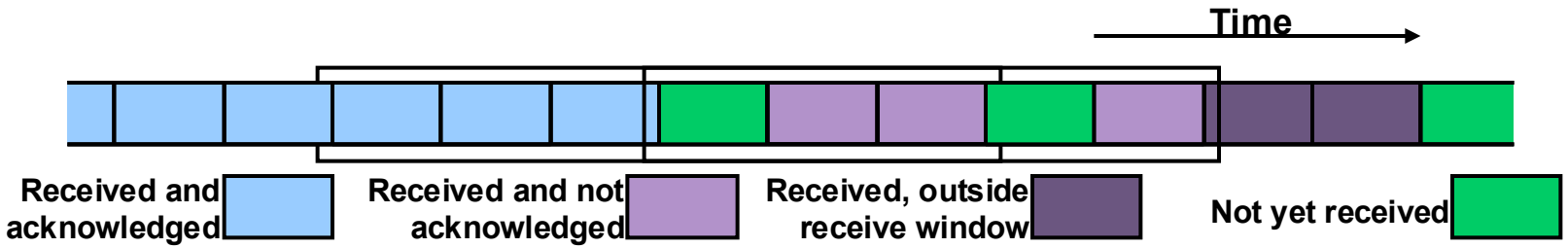
Sliding Window Terminology

- Sender Parameters
  - Send Window Size (SWS)
  - Last Acknowledgement Received (LAR)
  - Last Frame Sent (LFS)



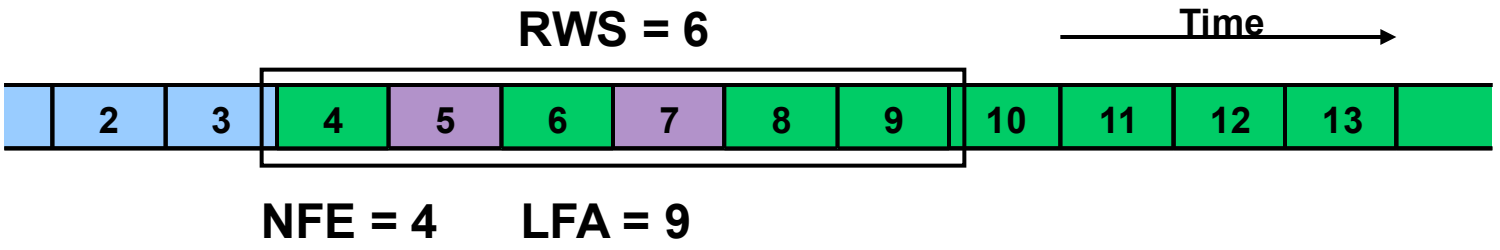
Sliding Window

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



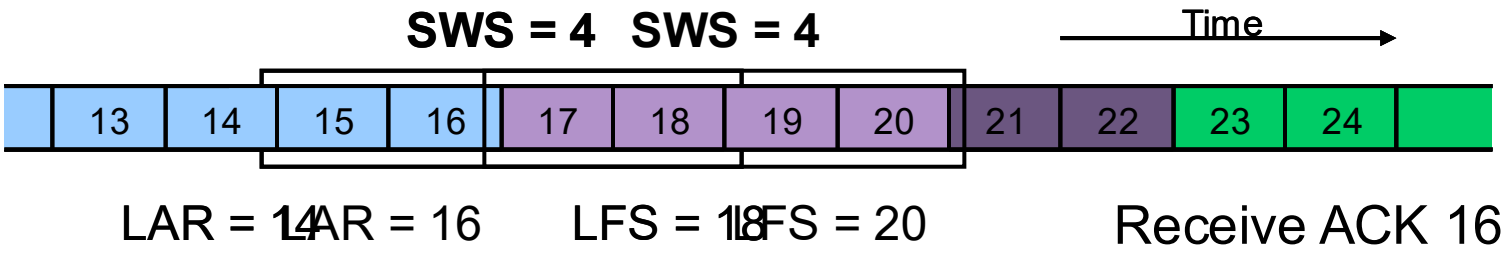
Sliding Window Terminology

- Receiver Parameters
  - Receive Window Size (RWS)
  - Next Frame Expected (NFE)
  - Last Frame Acceptable (LFA)



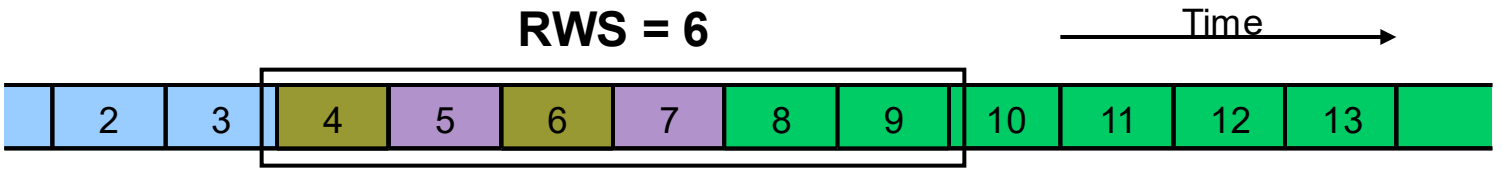
Sliding Window Details

- 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 >= NFE + RWS
    - Send cumulative ACK if within window



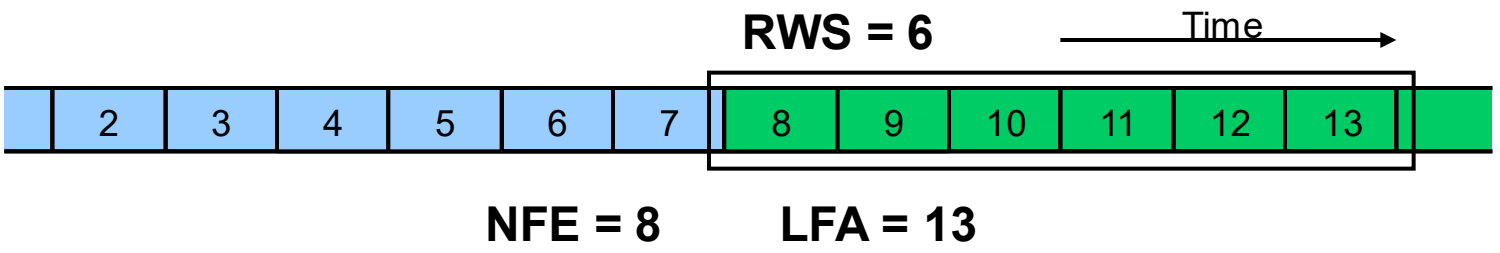
Receive Frame 4

Send ACK 7

© CS 439 Staff, University of Illinois

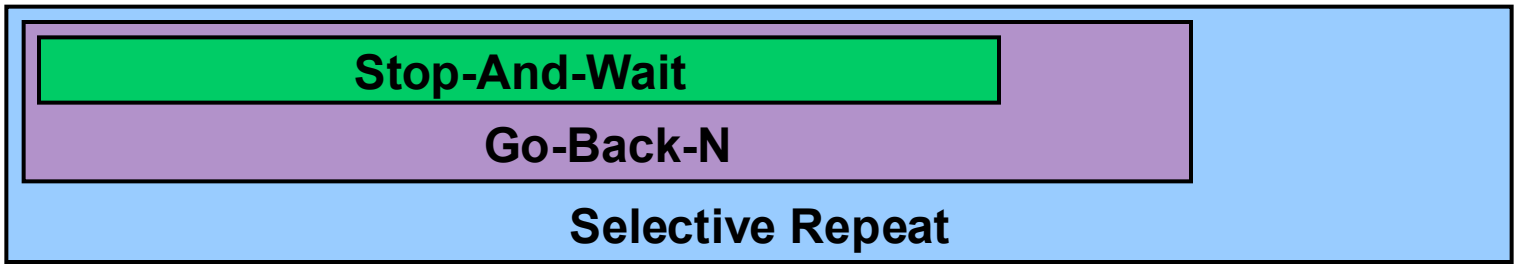
Sliding Window Details

- Receiver Tasks
  - On Frame Arrival (N)
    - Silently discard if outside of window
      - N < NFE (NACK possible, too)
      - N >= NFE + RWS
    - Send cumulative ACK if within window



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



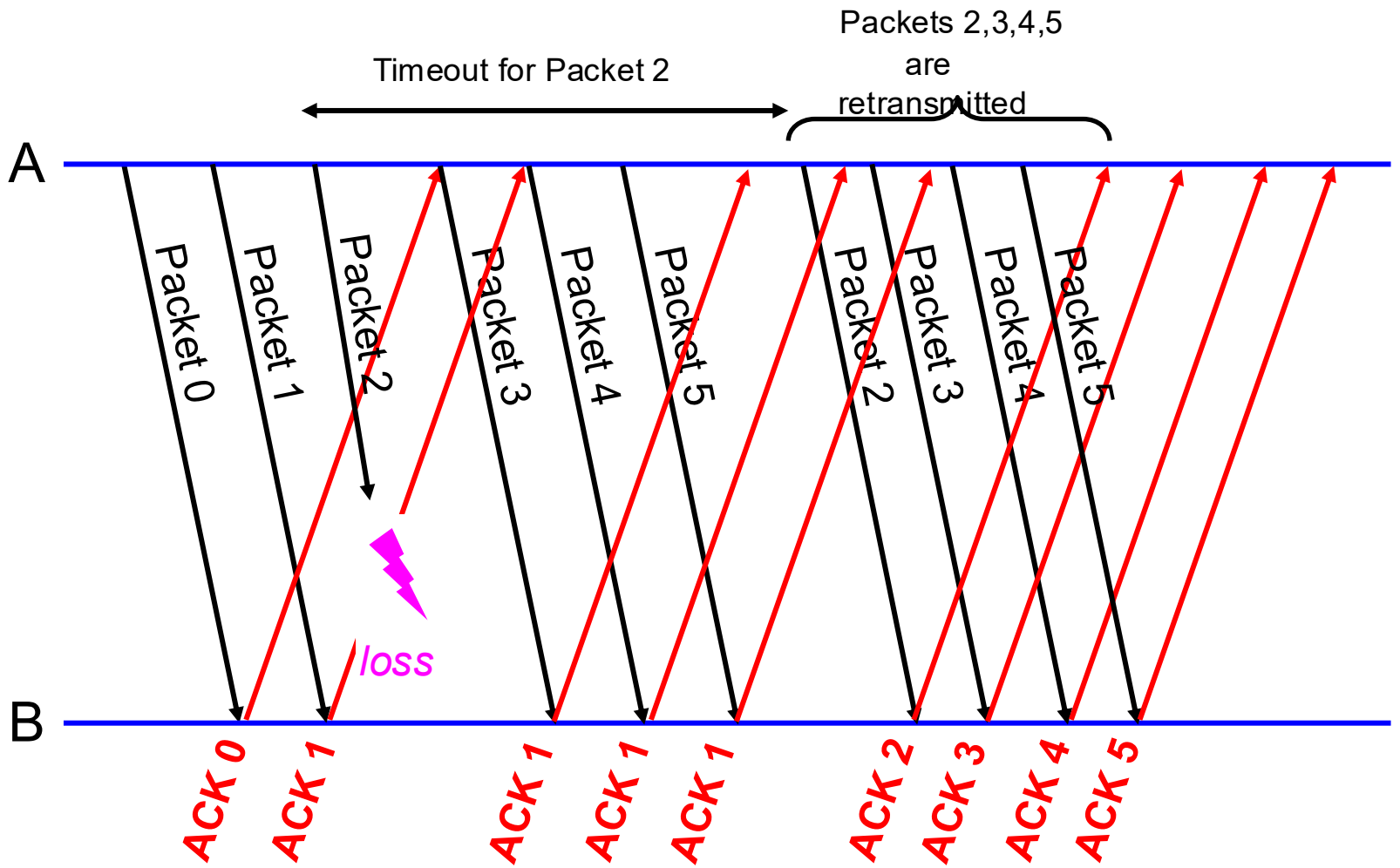
Sliding Window Variations: Go-Back-N

- 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





Go-Back-N: Cumulative ACKs

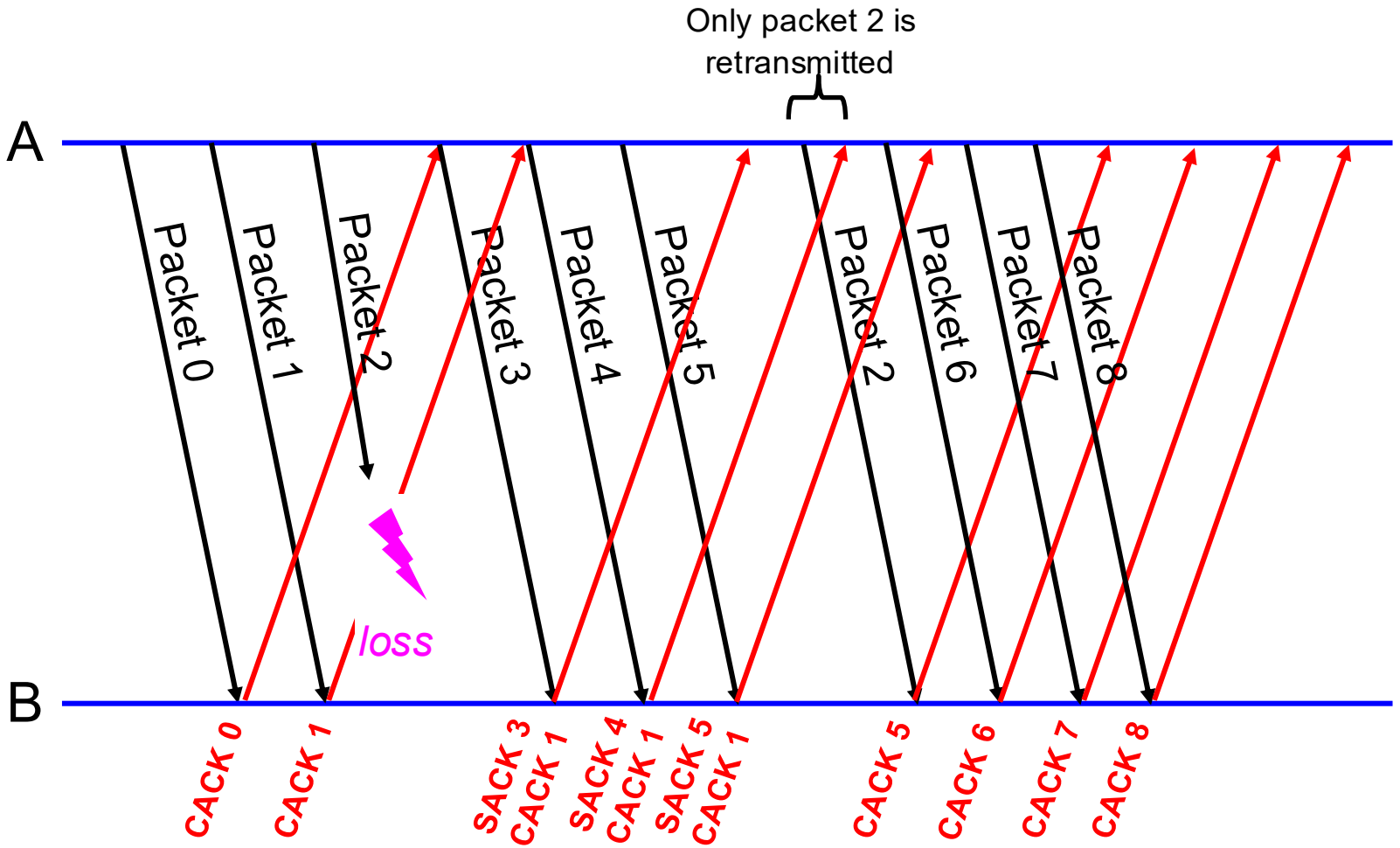


Sliding Window Variations: Selective Repeat

- 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
- AKA Selective Acknowledgements (SACK)
  - SACK option (RFC 2018)



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

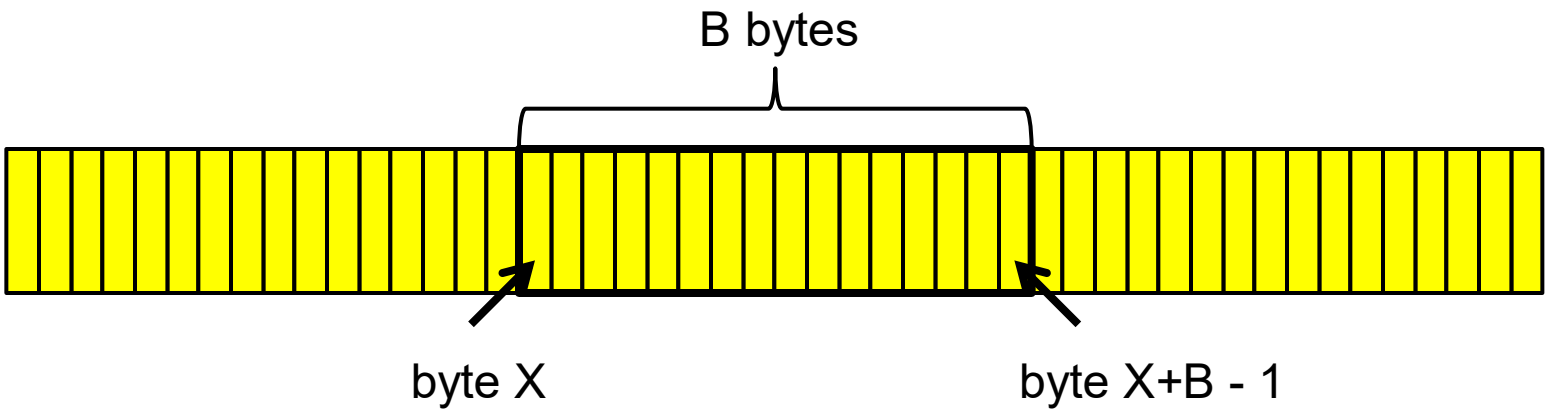
- Separation of Concerns
  - Must be able to distinguish between different functions that are sometimes rolled into one mechanism





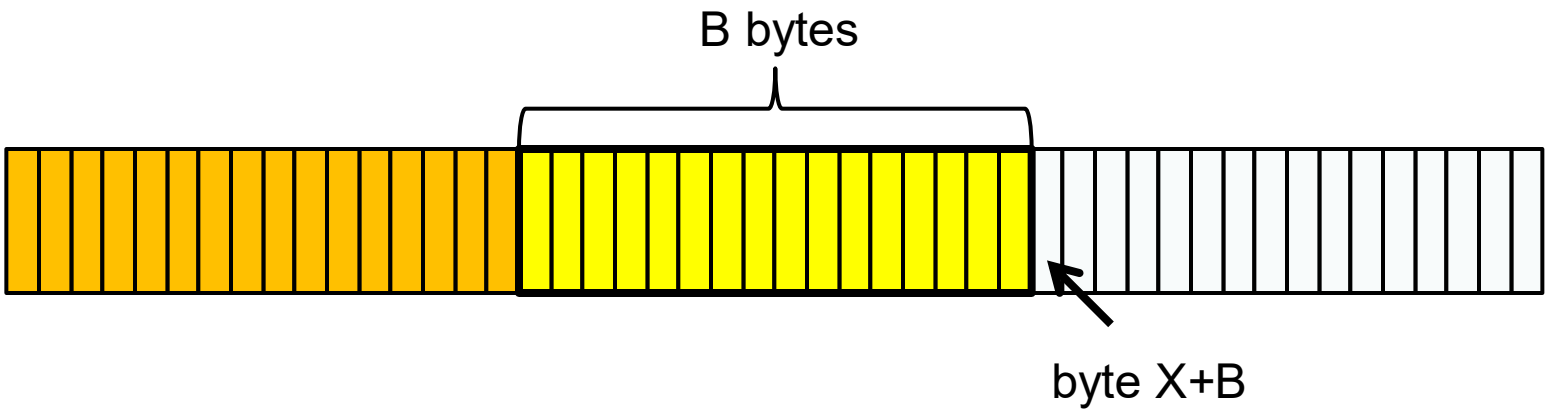
TCP ACKing and Sequence Numbers

- TCP sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes
    - X, X+1, X+2, ….X+B-1



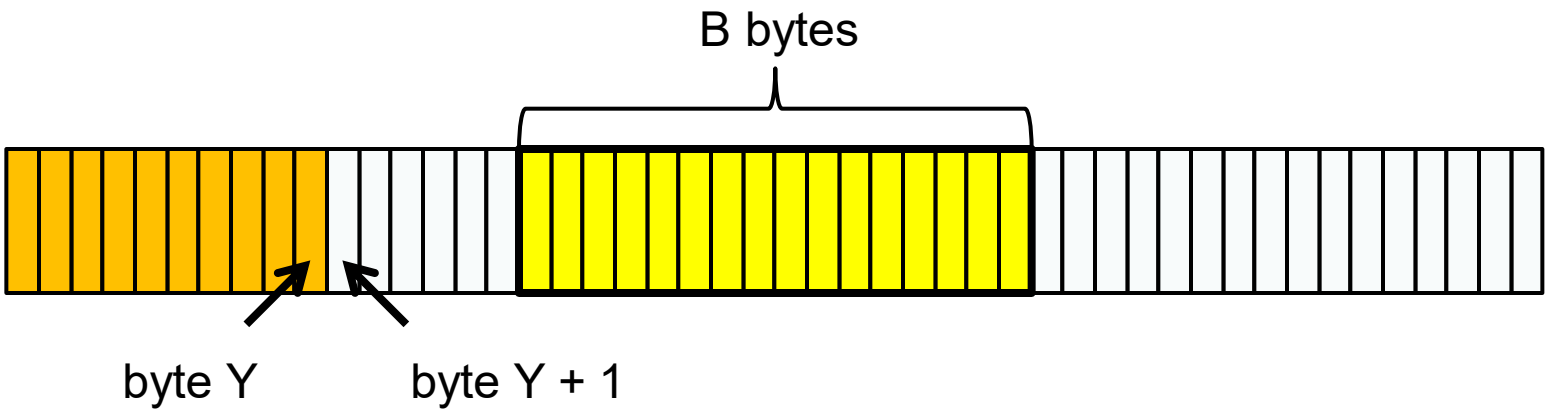
TCP ACKing and Sequence Numbers

- 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)



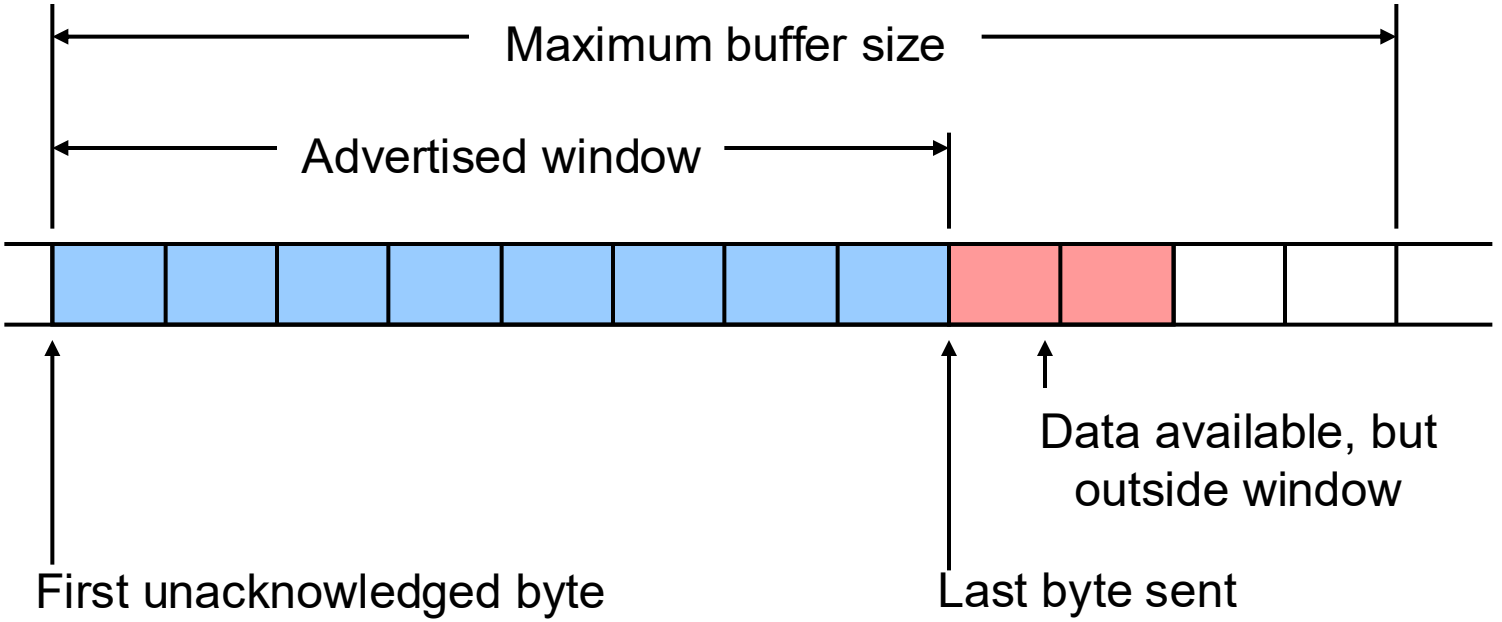
ACKing and Sequence Numbers

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



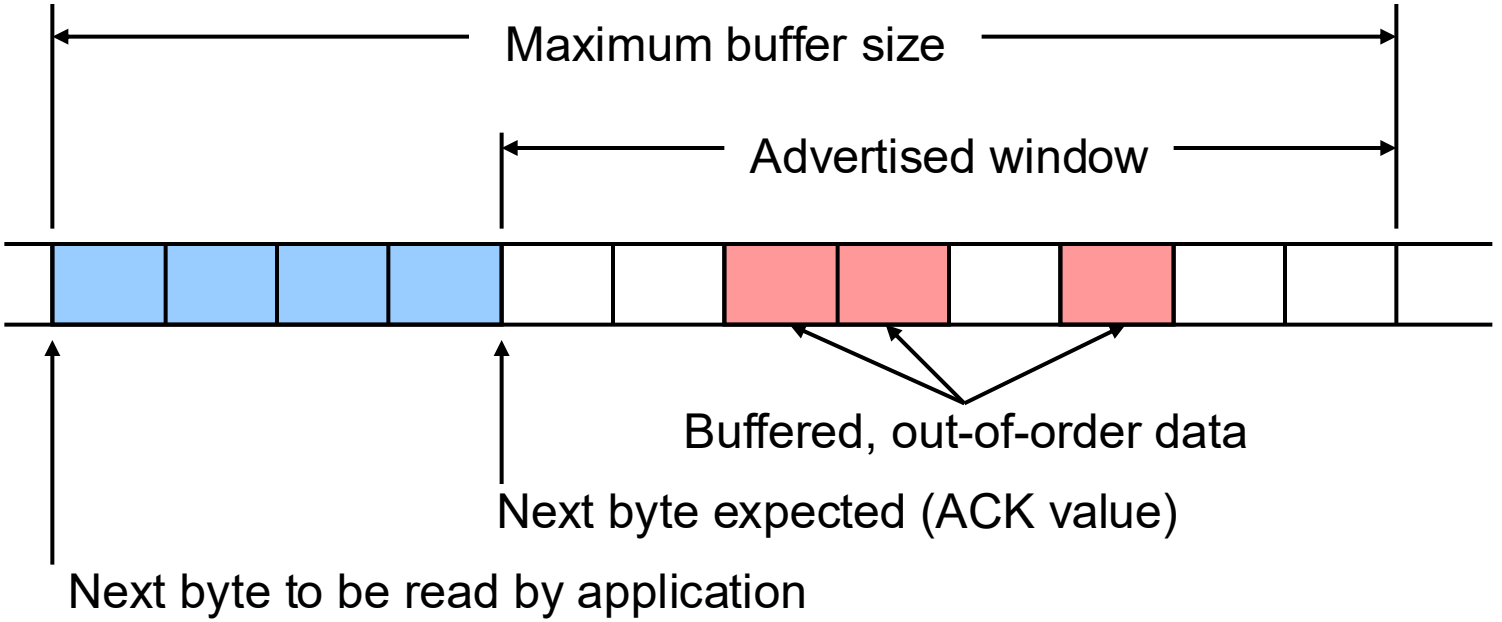
TCP Sliding Window Protocol – Sender Side

- Advertised window
  - Enables dynamic receive window size



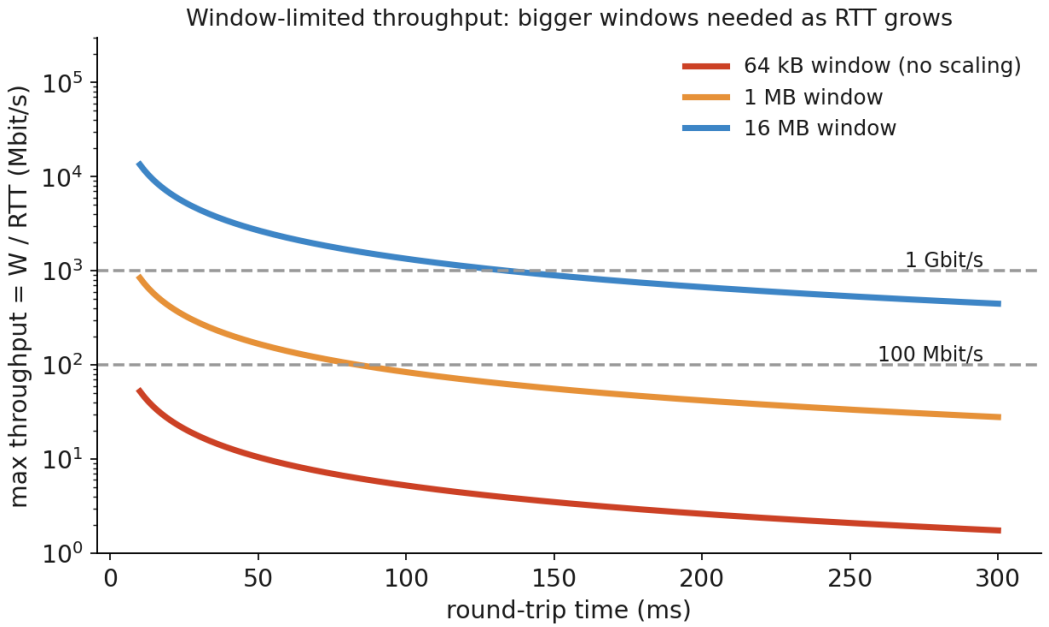
TCP Sliding Window Protocol – Receiver Side

- Receive buffers
  - Data ready for delivery to application until requested
  - Out-of-order data to maximum buffer capacity



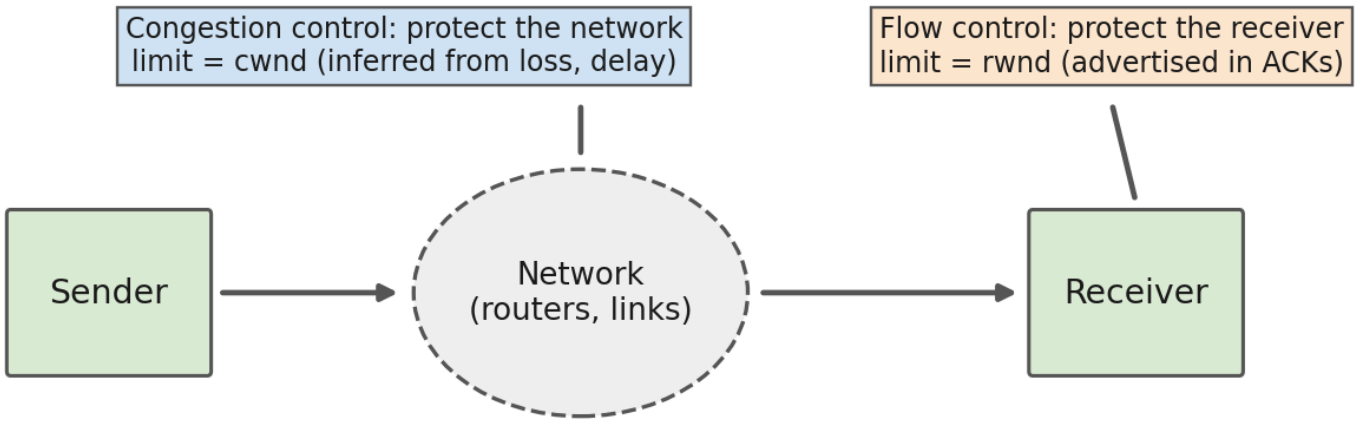
Advertised Window Limits Rate

- 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



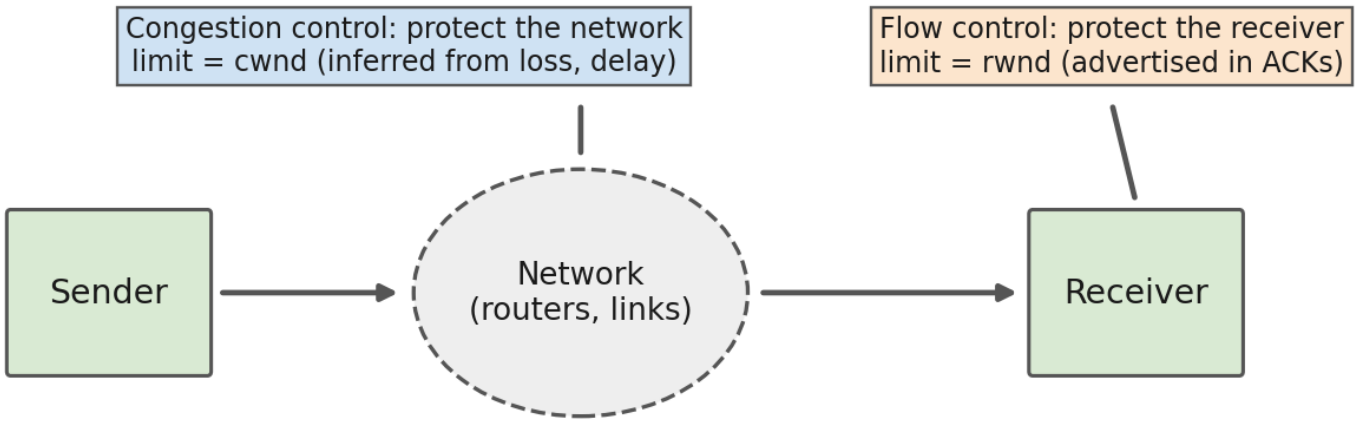
Flow Control vs. Congestion Control

- Flow control
  - Preventing senders from overrunning the capacity of the receivers



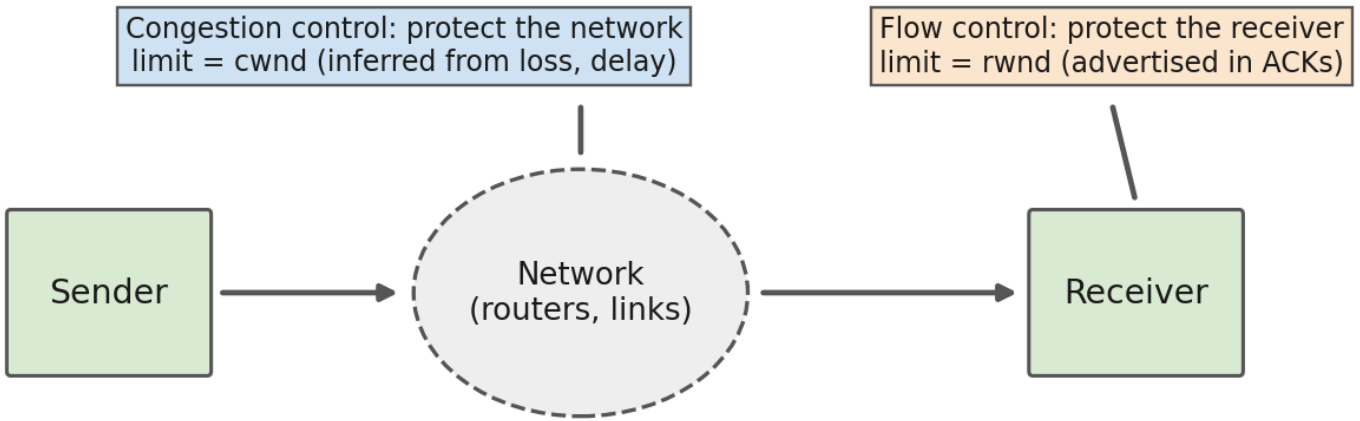
Flow Control vs. Congestion Control

- Congestion control
  - Preventing too much data from being injected into the network, causing switches or links to become overloaded

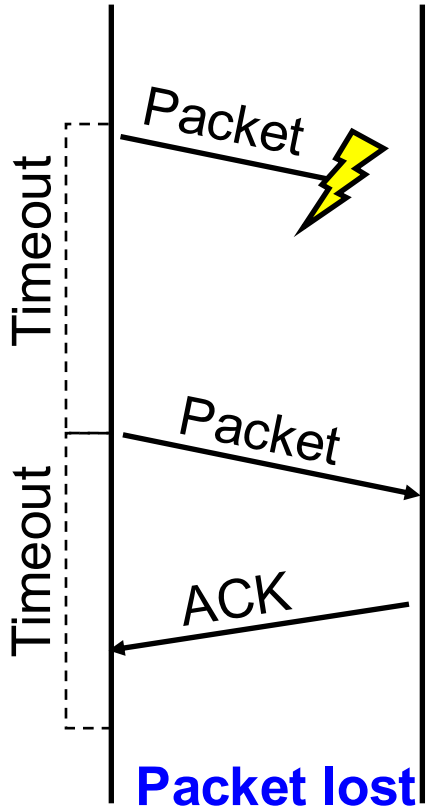


Flow Control vs. Congestion Control

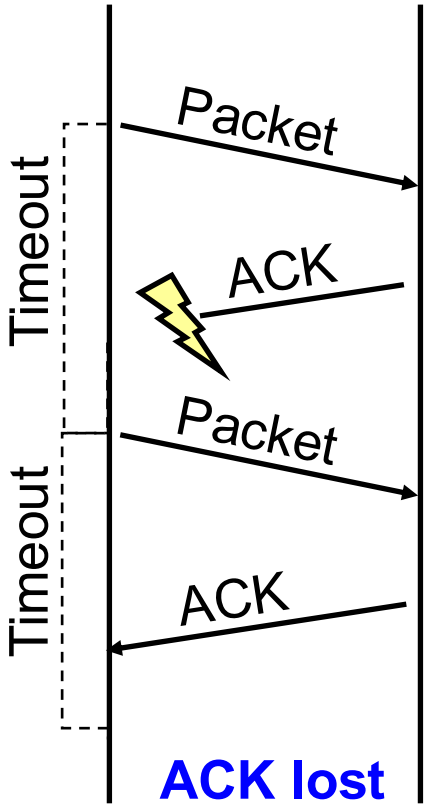
- Which one does TCP provide?
- TCP provides both
  - Flow control based on advertised window
  - Congestion control comes next



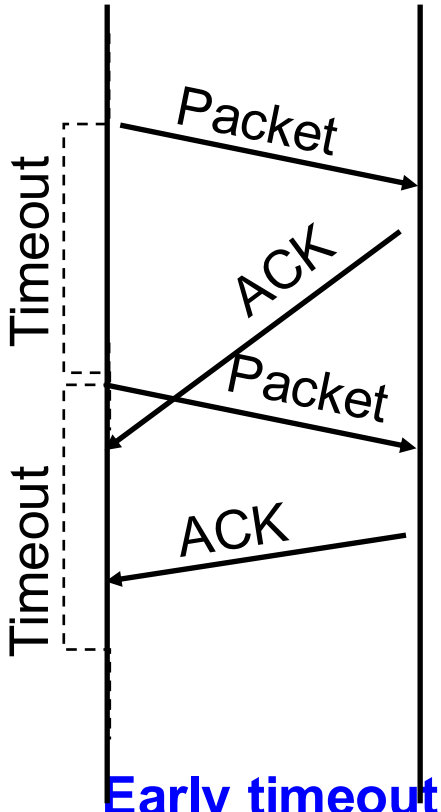
Reasons for Retransmission



Packet lost



ACK lost
DUPLICATE PACKET



Early timeout
DUPLICATE PACKETS

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
- Karn's rule
  - Omit RTT samples from retransmitted segments





TCP Congestion Control

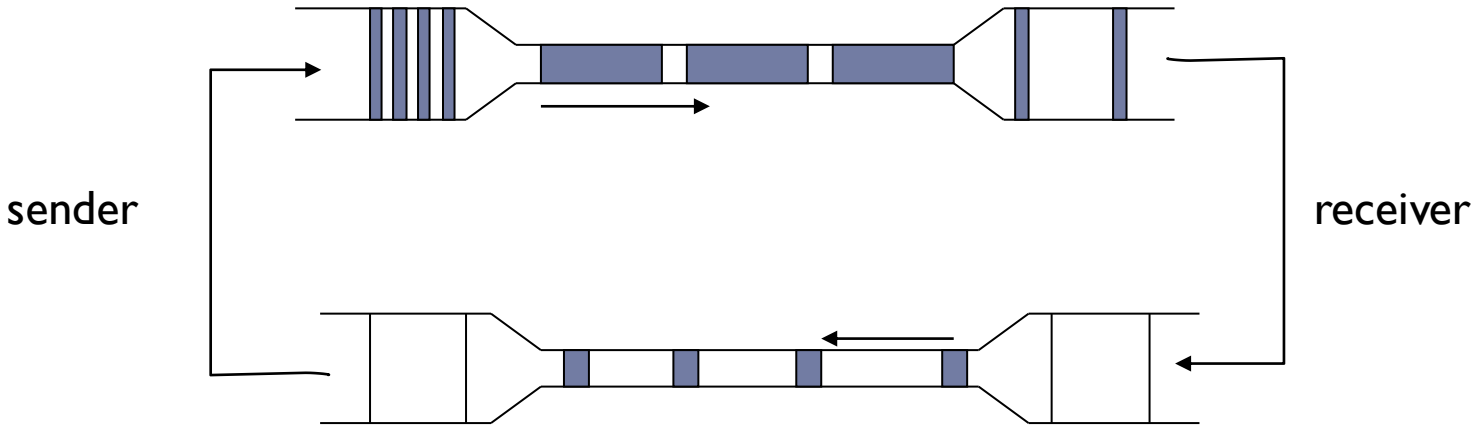
- Challenge
  - Determining initial available capacity
  - Adjusting to changes in capacity in a timely manner
- Basic idea
  - Add notion of congestion window
  - Effective window is smaller of
    - Advertised window (flow control)
    - Congestion window (congestion control)
  - Changes in congestion 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



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

- Additive increase
  - Add one maximum segment size (MSS) per congestion window of data ACK'd
  - In practice
    - Each ACK of new data grows cwnd by MSS x MSS/cwnd
- Multiplicative decrease
  - Cut window in half on loss signaled by duplicate ACKs (a timeout resets it to 1 MSS)
  - In practice
    - ssthresh = cwnd / 2





Additive Increase/ Multiplicative Decrease

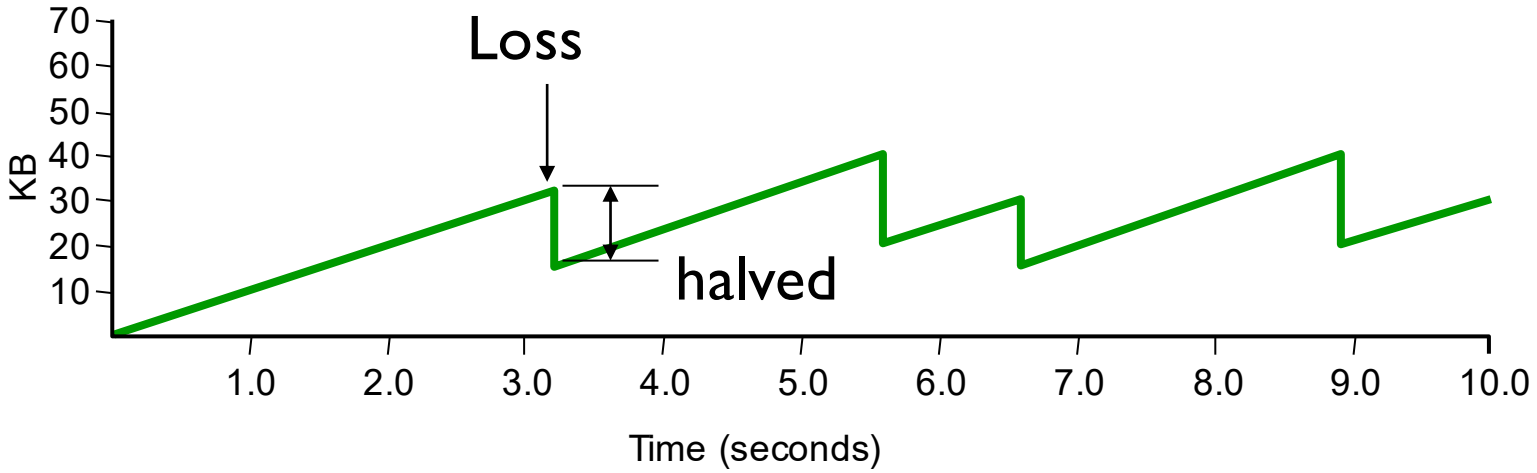
- Question
  - How does the source determine whether or not the network is congested?
- Answer
  - Timeout signals packet loss
  - Packet loss is rarely due to transmission error (on wired lines)
  - Lost packet implies congestion! (on wired lines)





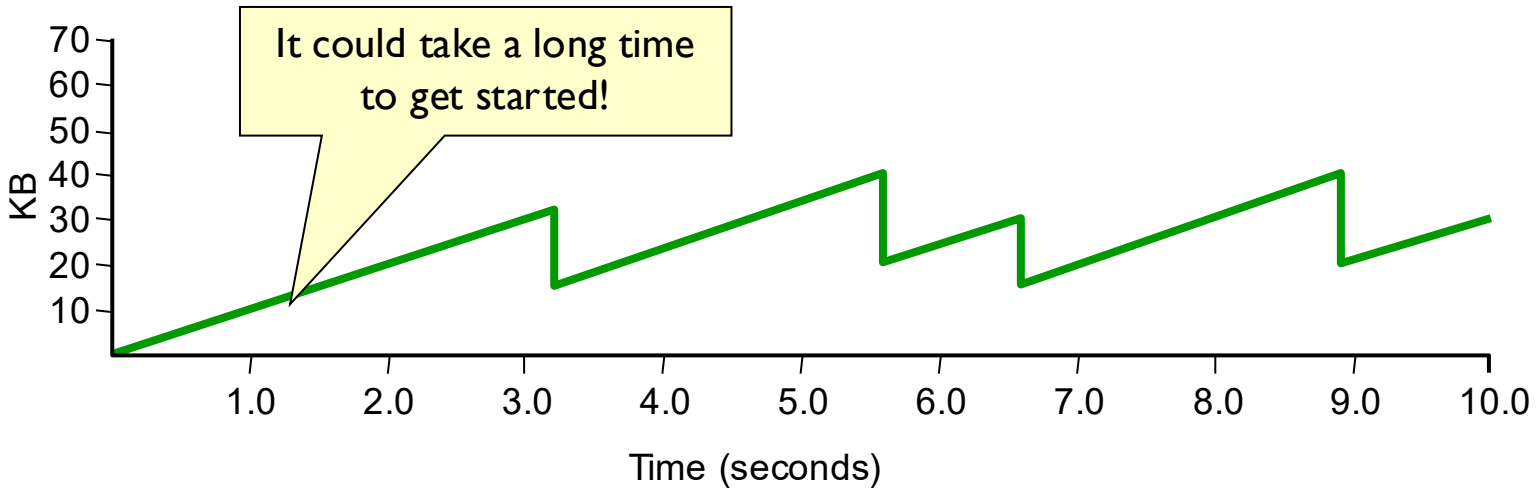
AIMD – Sawtooth Trace

- 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 (modern stacks start at 10 MSS, RFC 6928)
    - Initial sending rate is MSS/RTT
  - Reset to 1 MSS with each timeout
    - timeouts are coarse: RFC 6298 minimum RTO is 1 s (Linux uses 200 ms)



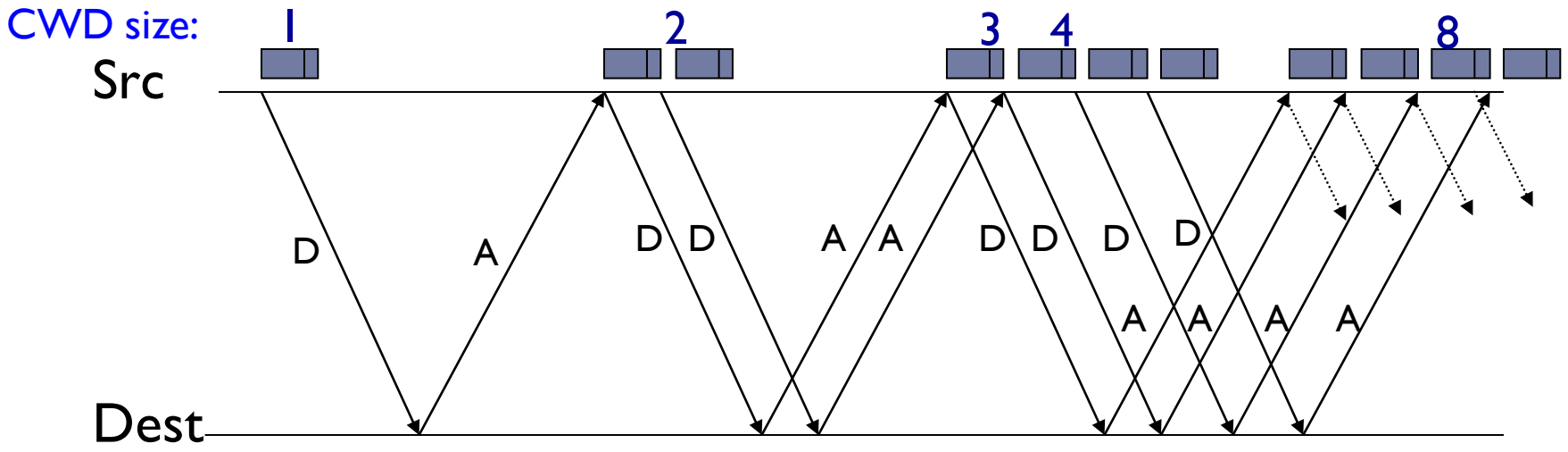


TCP Start Up Behavior: Slow Start

- 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

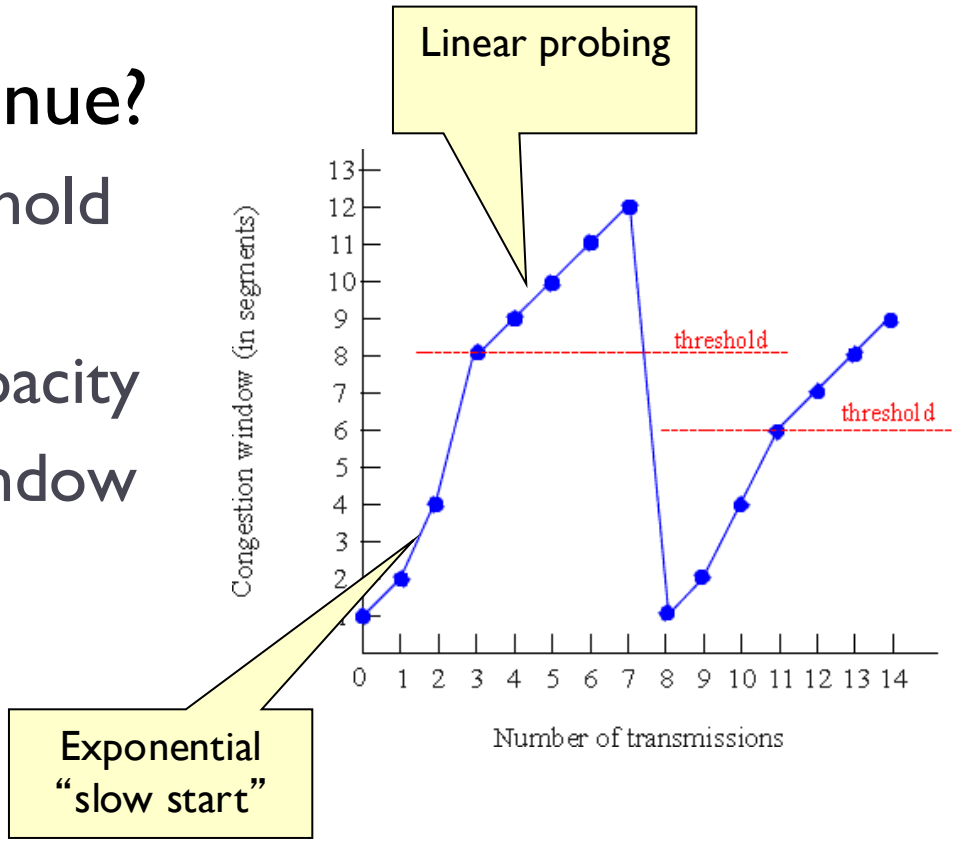
- Maintain threshold window size
  - Threshold value
    - Initially set to maximum window size
    - Set to half the data in flight (at least 2 MSS) on loss
  - Use multiplicative increase
    - When congestion window smaller than threshold
    - Double window for each window ACK'd
- In practice
  - Increase congestion window by one MSS for each ACK of new data (or N bytes for N bytes)





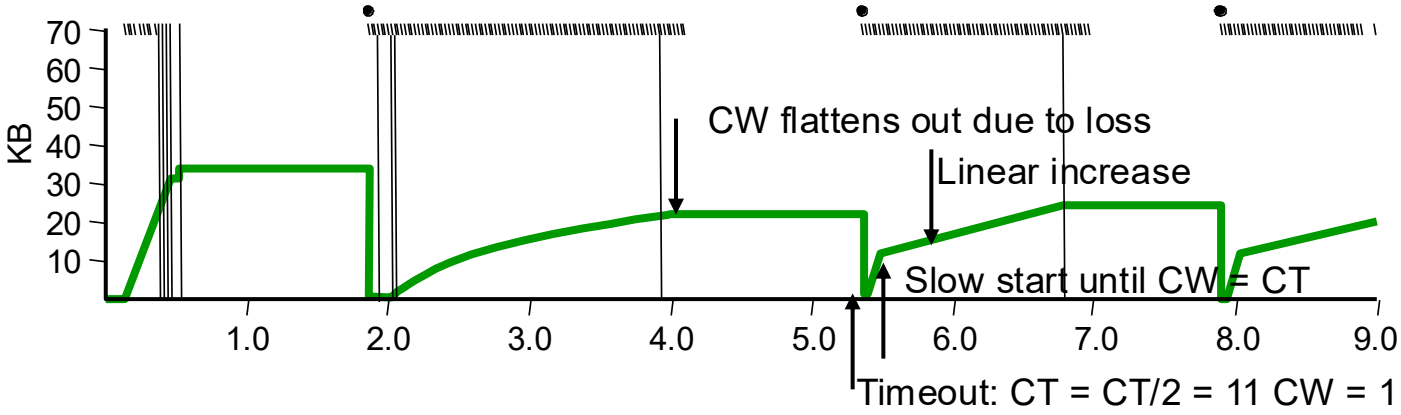
Slow Start

- 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

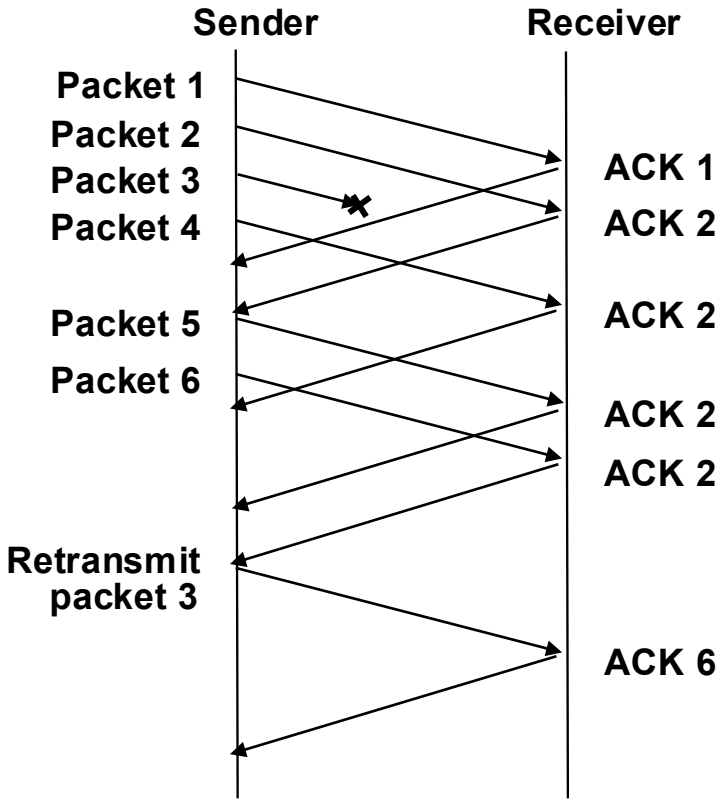
- Example trace of **`CongestionWindow`**



  - 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

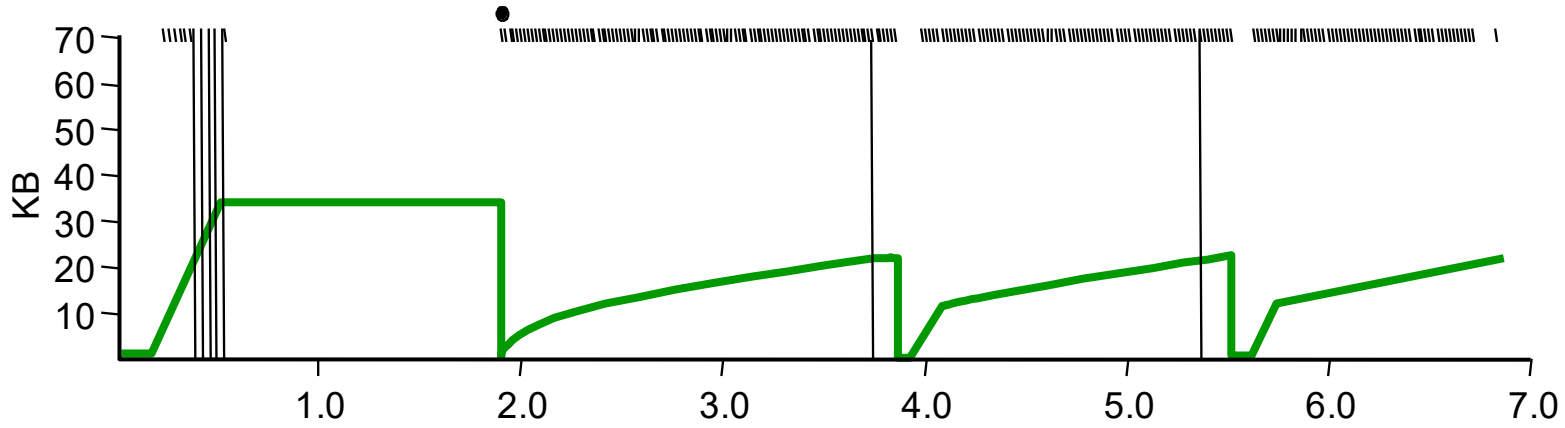
- Send ACK for each segment received
- When duplicate ACK's received
  - Resend lost segment immediately
  - Do not wait for timeout
  - In practice, retransmit on 3rd duplicate
- Fast recovery
  - When fast retransmission occurs, skip slow start
  - Congestion window becomes 1/2 previous
  - Start additive increase immediately
  - Reno recovers from one loss per window
    - Several losses often end in a timeout





Fast Retransmit and Fast Recovery

- Results



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

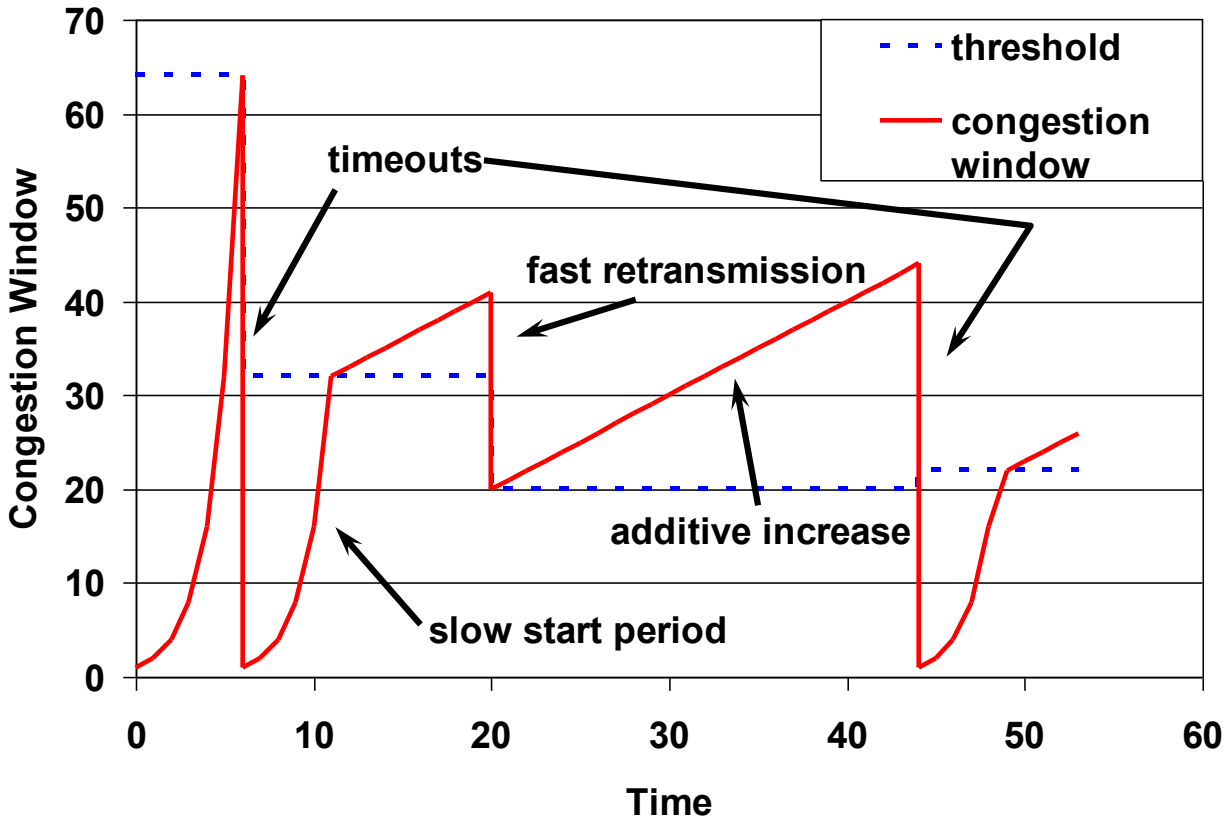
Delayed ACKs and Duplicate ACKs

- Delayed ACK
  - Receiver waits to acknowledge
  - Typically ACKs every second full segment, or after a short timer (tens of ms)
  - Halves the number of ACKs
- Duplicate ACK
  - Same ACK number repeated
  - Sent for each out-of-order segment, since the next expected byte has not arrived
  - Three duplicate ACKs trigger fast retransmit





TCP Congestion Window Trace



TCP Throughput vs. Loss Rate

- Steady-state AIMD
  - Throughput ≈ 1.22 · MSS / (RTT · $\sqrt{p}$)
  - To fill 100 Mbit/s at 100 ms RTT, loss must be about 0.0002%
- Wireless
  - Random loss is very costly for loss-based TCP

