

Congestion Control

Amin Vahdat
CSE 123b
April 20, 2006

Annoucements

- First assignment available
 - HTTP Server: due Apr 21, 5 pm
- Midterm: May 9
- Email Calvin Hubble (chubble@cs.ucsd.edu) your project 2 group makeup

End-to-End Protocols

Host-to-Host vs. Process-to-Process Communication

- Until now, we have focused on delivering packets between arbitrary *hosts* connected to Internet
 - Routing protocols
 - IP best effort delivery model
 - Scalability and robustness through hierarchy and soft state
- Transition to arbitrary *processes* communicating together
 - One goal: provide illusion that all processes located on one large computer
 - Can address (*name*) and reliably communicate with any process

UDP

■ User Datagram Protocol (UDP)

- Simple demultiplexing

No guarantees about reliability, in-order delivery

- Thin veneer on top of IP adds src/dest port numbers

16 bit port number allows for identification of 65536 unique communication endpoints per host

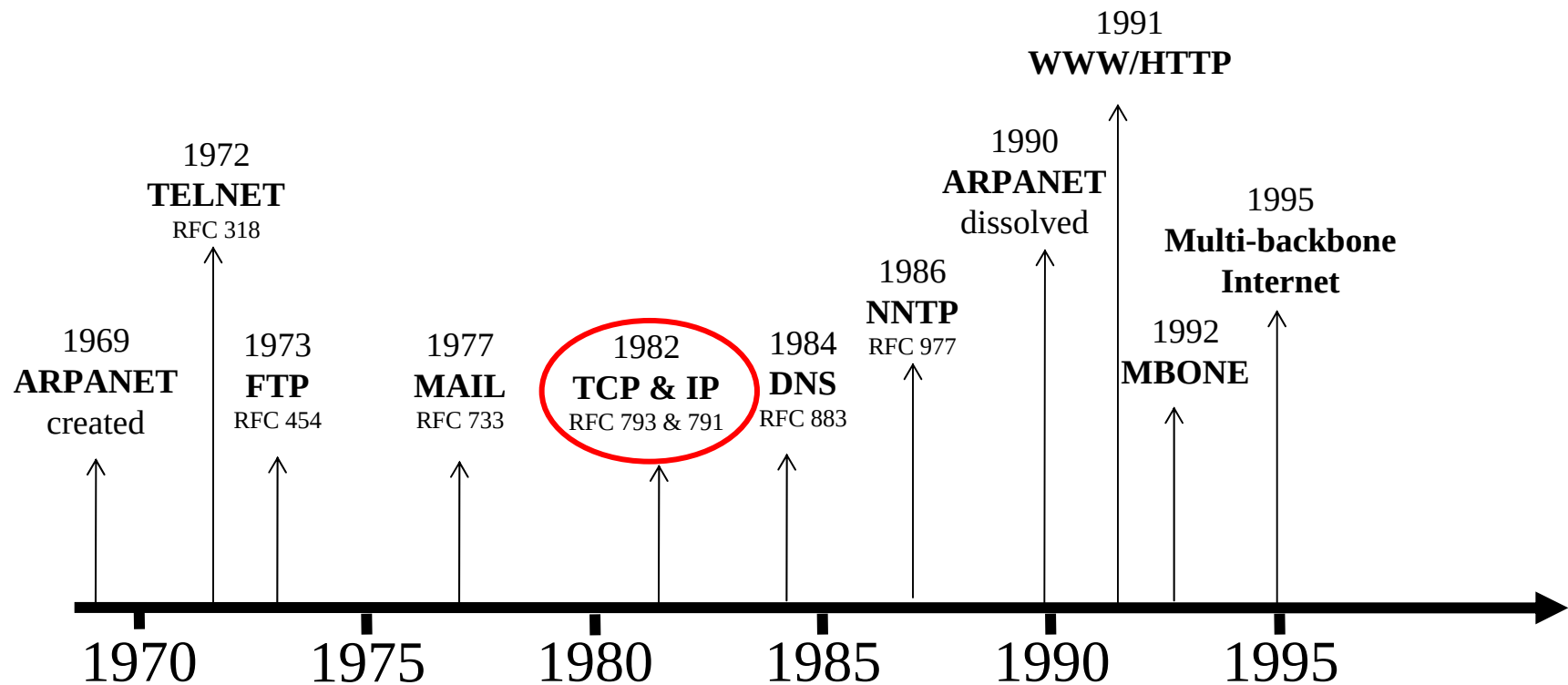
Note that a single process can utilize multiple ports

IP addr + port number uniquely identifies all Internet endpoints

- UDP Packet



A Brief Internet History



TCP

- Transmission Control Protocol (TCP)
 - Reliable in-order delivery of byte stream
 - Full duplex (endpoints simultaneously send/receive)
 - e.g., single socket for web browser talking to web server
 - Flow-control
 - To ensure that sender does not overrun *receiver*
 - Fast server talking to slow client
 - Congestion control
 - Keep the sender from overrunning the *network*
 - Many simultaneous connections across routers (cross traffic)

TCP Flow Control and Congestion Control

- Sender must determine maximum amount of data in transit that will not overrun *either* receiver or network
- Solutions?

TCP Flow Control and Congestion Control

- Sender must determine maximum amount of data in transit that will not overrun *either* receiver or network
- Solutions:
 - Maintain “sliding window” to track data in transit
 - Size of window determined by minimum of “flow window” and “congestion window”
 - Receiver ACKs “slide” left side of window forward

Opening up another “slot” at right side of window for transmission



TCP Sliding Window Protocol Issues

- Need for connection establishment (no dedicated cable)
- Varying round trip times over life of connection
 - Different paths, different levels of congestion
- Ready for *very* old packets
- Delay-bandwidth product highly variable
 - Amount of available buffer space at receivers also variable
- Sender has no idea what links will be traversed to receiver
 - Must dynamically estimate changing end-to-end characteristics

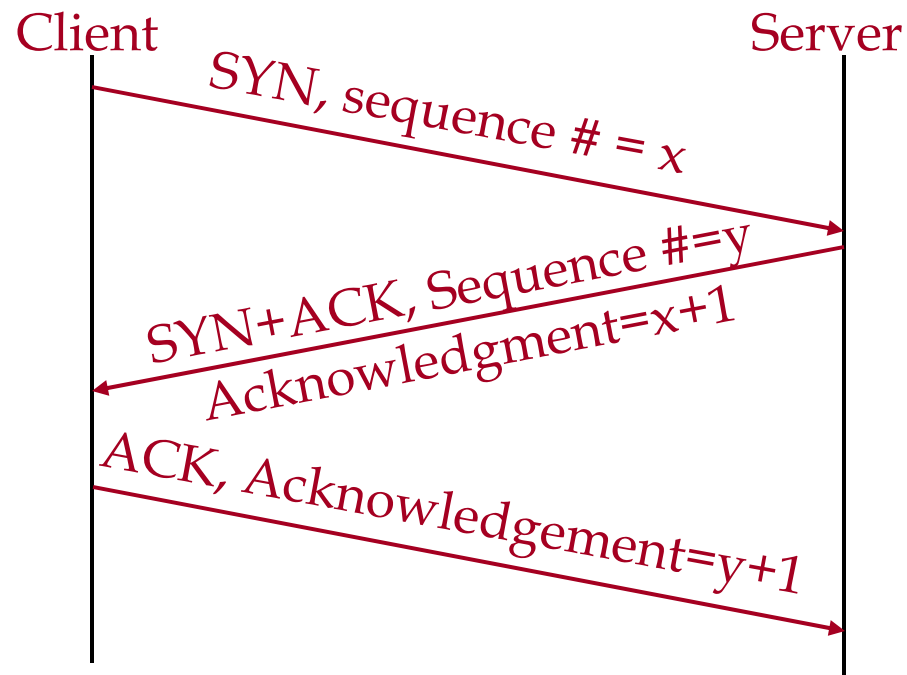
TCP Header Format

0	4	10	16	31
SrcPort		DestPort		
SequenceNum				
Acknowledgment				
HdrLen	0	Flags	AdvertisedWindow	
Checksum			UrgPtr	
Options (variable)				
Data				

- Without options, TCP header 20 bytes
 - Thus, typical Internet packet min of 40 bytes (+link header)

TCP Connection Establishment

- Exchange necessary information to begin communication
- Three-way handshake
 - E.g., server listening on socket

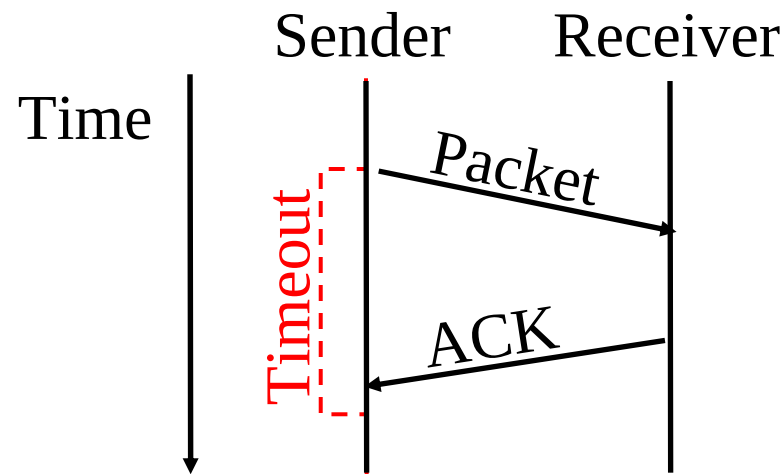


TCP Connection Teardown

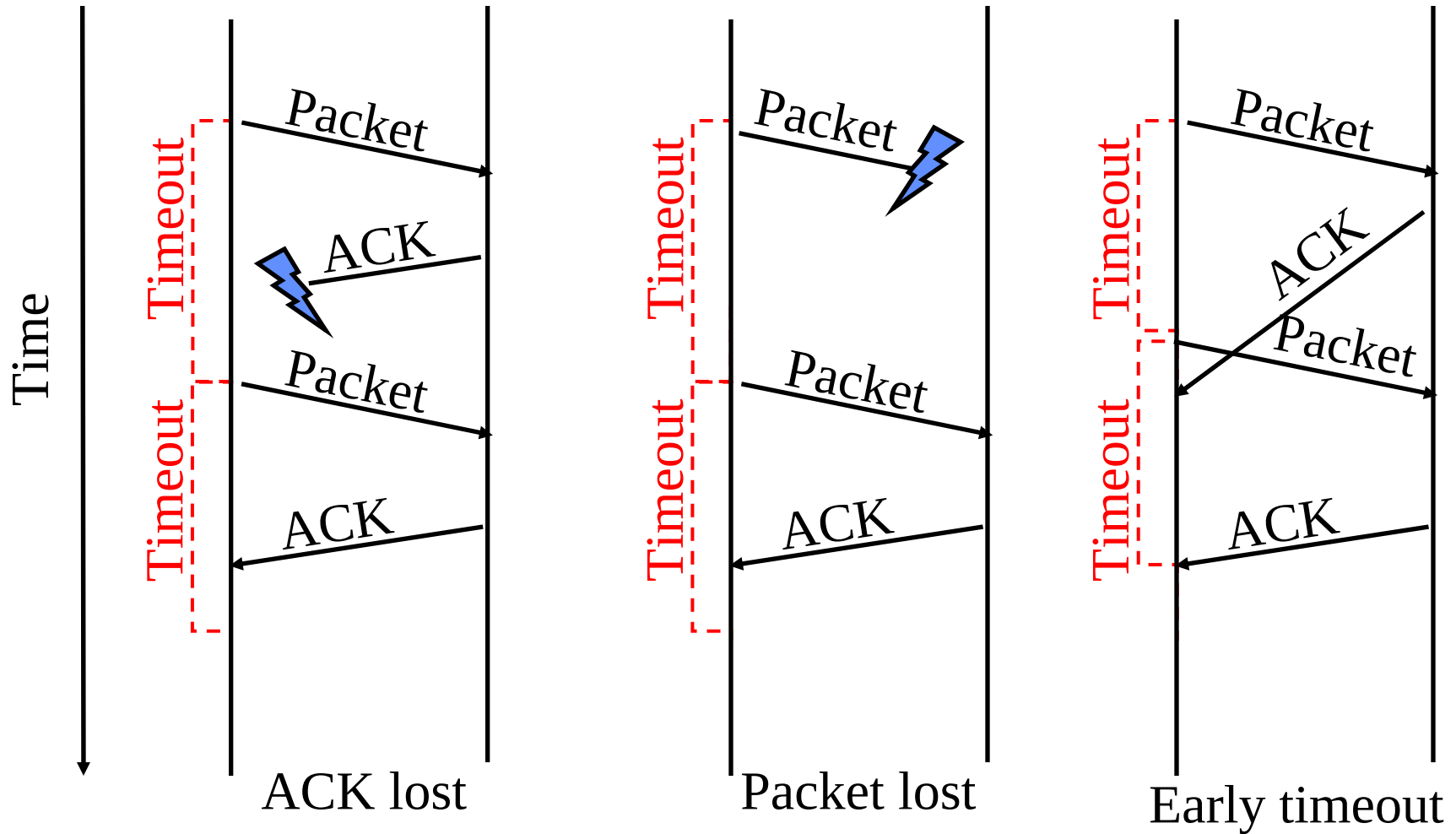
- Each side of a TCP connection can independently close the connection
 - Thus, possible to have a half duplex connection
- Closing process sends a FIN message
 - Waits for ACK of FIN to come back
 - This side of the connection is now closed

Reliability, First Cut: Stop and Wait

- Reliability, two principal mechanisms: ACKs and timeouts
- Send a packet, stop and wait until acknowledgement arrives before sending next packet



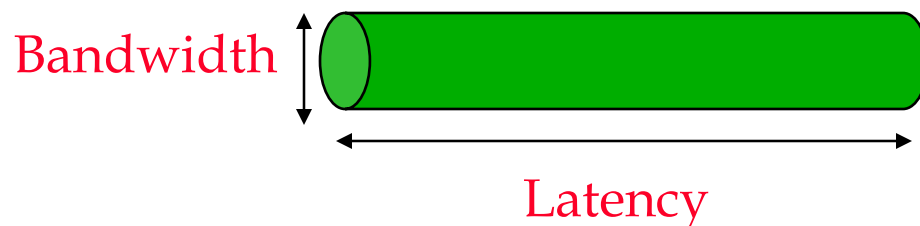
Recovering From Error



Problems with Stop and Wait

- How to recognize a duplicate transmission?
 - Solution: put sequence number in packet
- Performance
 - Unless Latency-Bandwidth product is very small, sender cannot fill the pipe
 - Solution: sliding window protocols

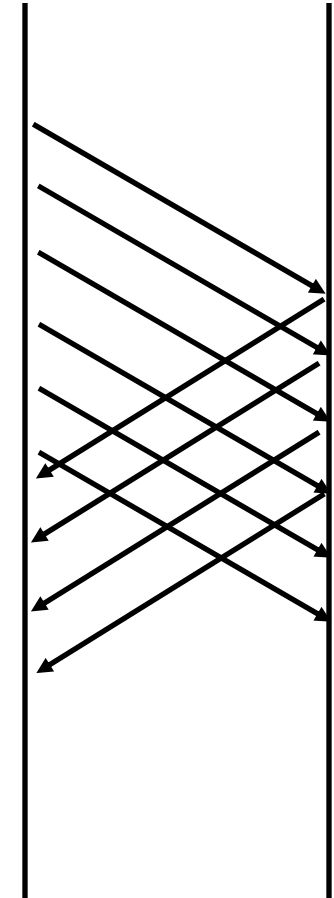
Keeping the Pipe Full



- Bandwidth-Delay product measures network capacity
- How much data can you put into the network before the first byte reaches receiver
- Stop and Wait: 1 data packet per RTT
 - Ex. 1.5-Mbps link with 45-ms RTT
 - Stop-and-wait: 182 Kbps
- Ideally, send enough packets to fill the pipe before requiring first ACK

How Do We Keep the Pipe Full?

- Send multiple packets without waiting for first to be ACK'd
- Reliable, unordered delivery:
 - Send new packet after each ACK
 - Sender keeps list of unack'd packets; resends after timeout
- Ideally, first ACK arrives immediately after pipe is filled
 - Opens up another “slot”
- Example 10 Mbit link, 100 ms RTT:
 - Send 125 KB to keep pipe full



Sliding Window Protocols: Reliable In-Order Delivery + Flow Control

- Receivers buffer later packets until prior packets arrive
 - For out of order delivery
 - Receiver ACKs largest sequence # received in order

What happens when receive 1, 2, 3, 5, 6, 7?

- Sender must prevent buffer overflow at receiver
 - Flow control
- Solution: sliding window
 - Circular buffer at sender and receiver

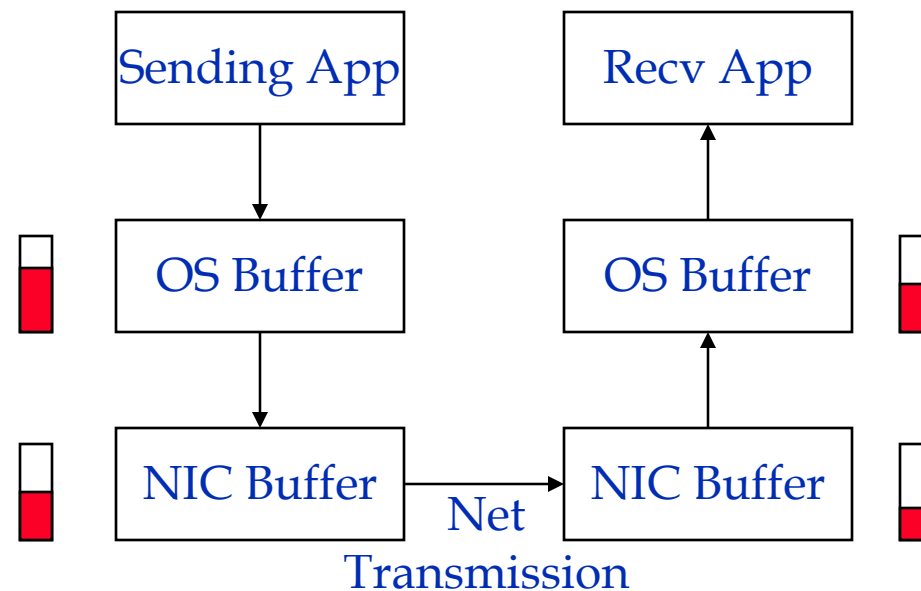
Packets in transit $\leq$ buffer size

Advance when sender and receiver agree packets at beginning have been successfully received

TCP Flow Control

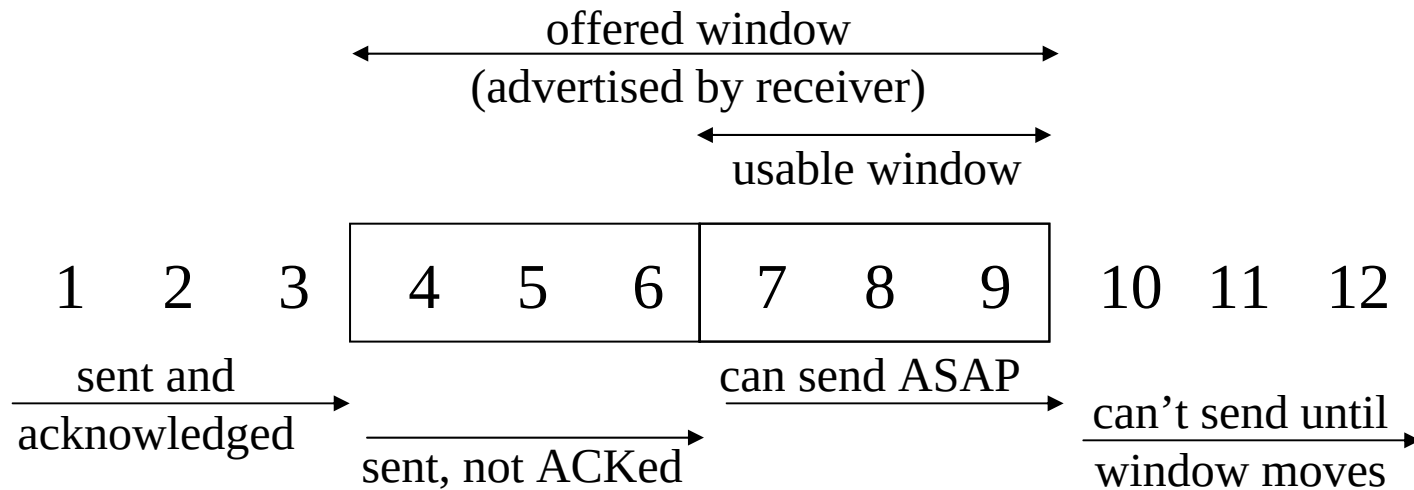
- TCP is a sliding window protocol
 - For window size n , can send up to n bytes without receiving an acknowledgement
 - When the data is acknowledged then the window slides forward
- Each packet advertises a window size in TCP header
 - Indicates number of bytes the receiver is willing to buffer
- Original TCP always sent entire window immediately
 - Too bursty?

Buffering and Flow Control



- Buffering taking place at multiple places
 - Only finite space available at each location
 - System will eventually block (through backpressure)
 - How to apply backpressure across the Internet?

Visualizing the Window

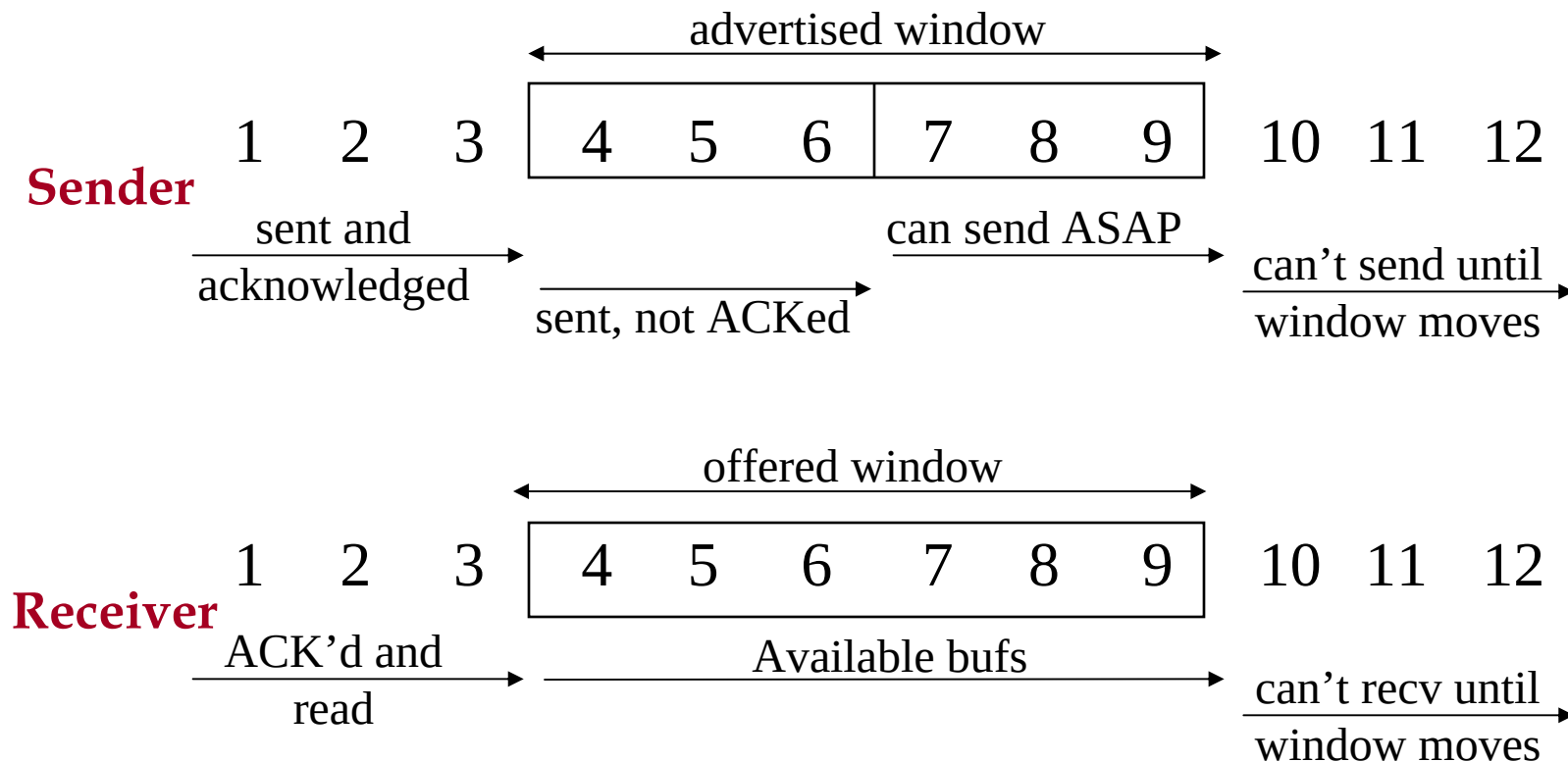


Left side of window advances when data is acknowledged.
Right side controlled by size of window advertisement.

Visualizing the Window: Example

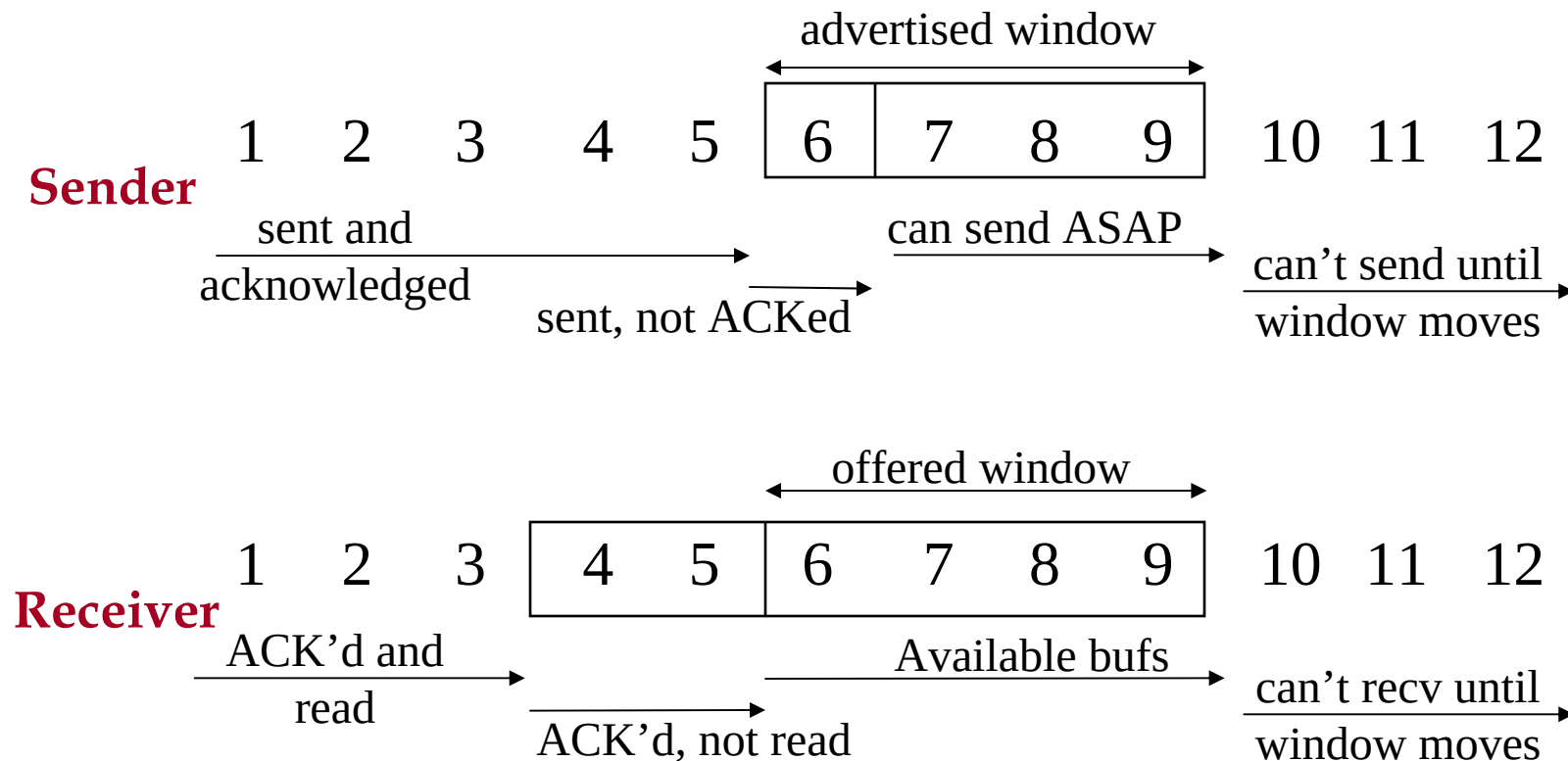
Initial State, Receiver has 6 slots to buffer packets

Packets 4, 5, 6 sent, but not yet received



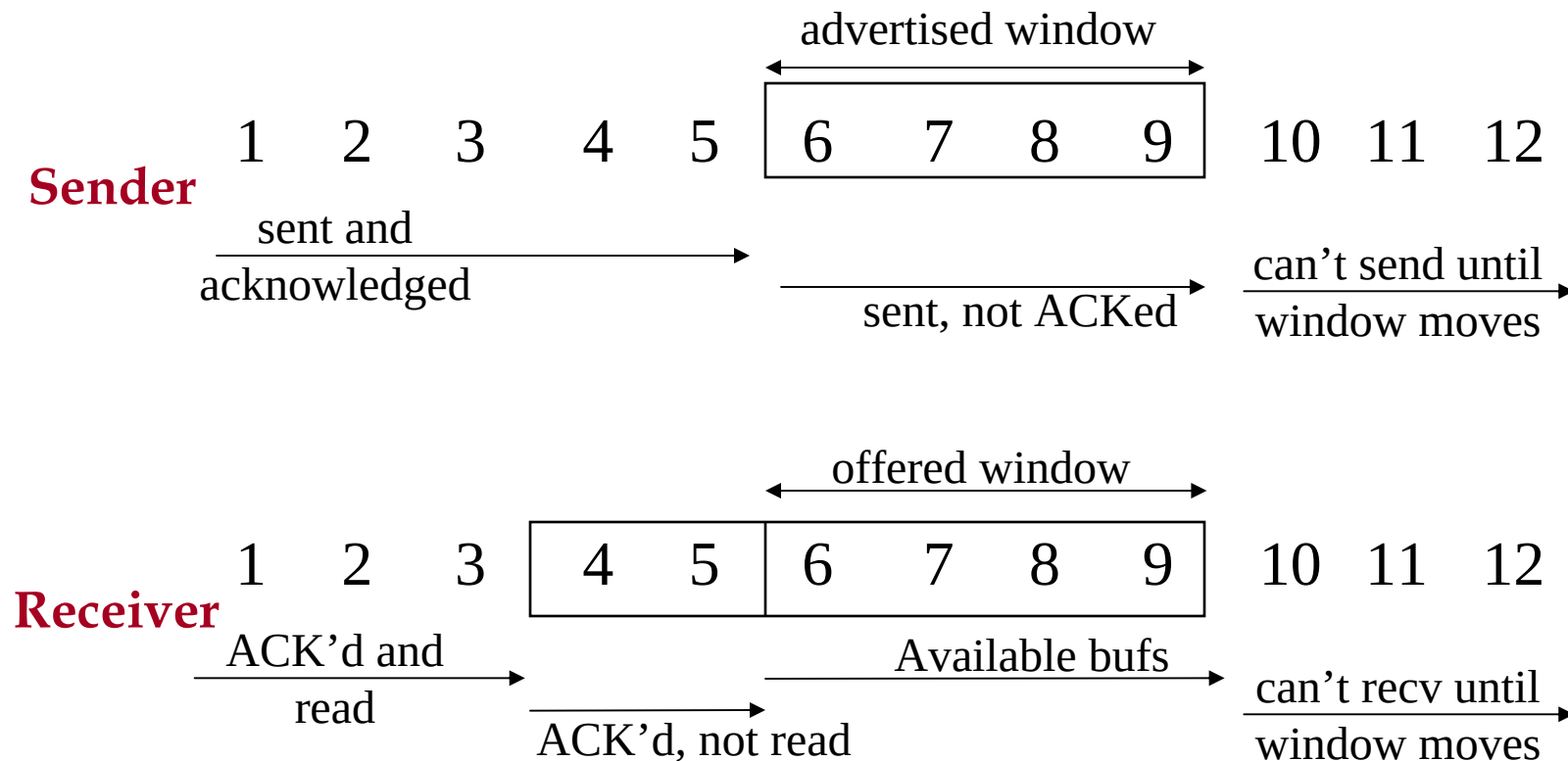
Visualizing the Window: Example

Receiver to Sender → ACK 5, Window 4



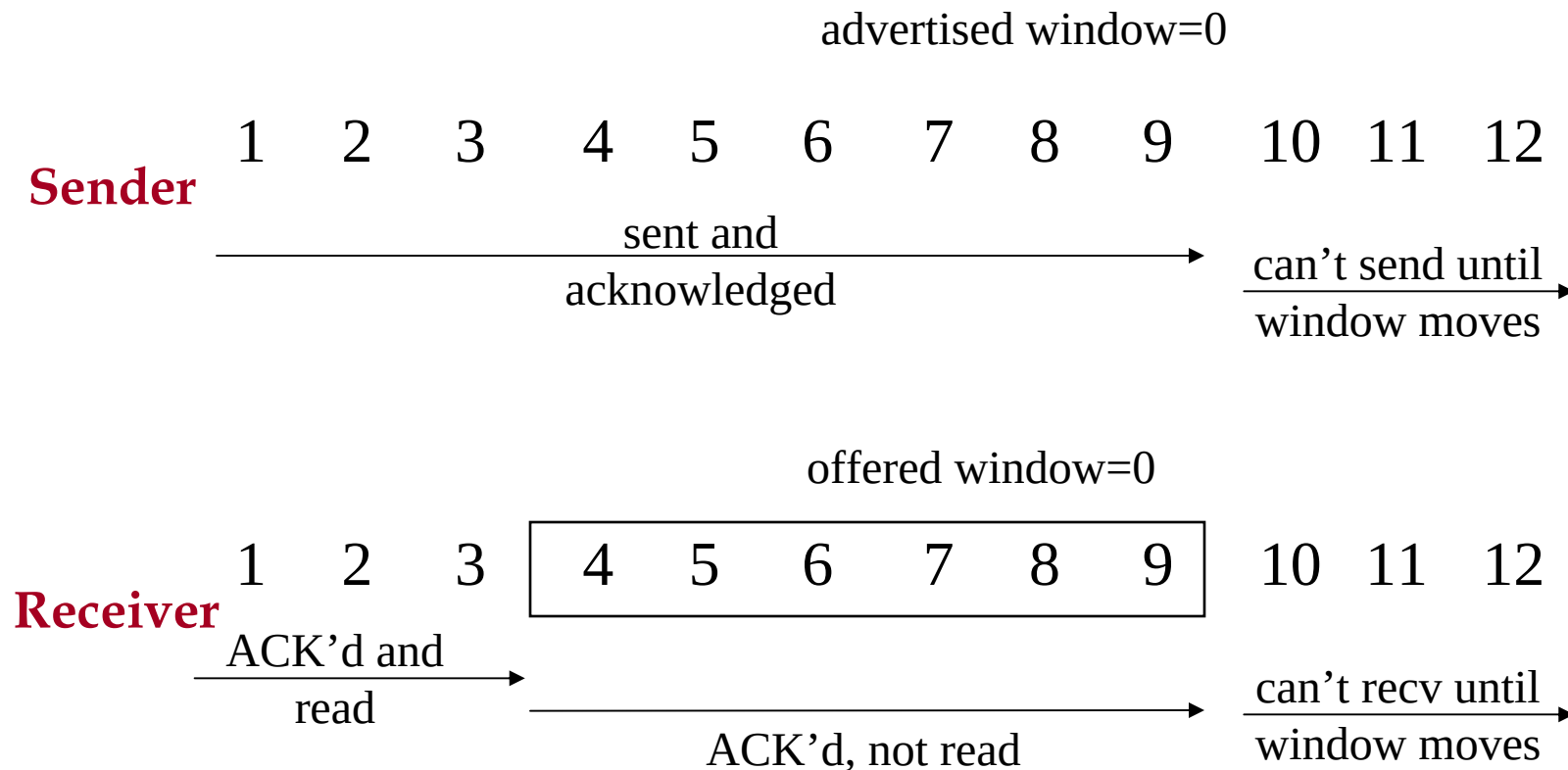
Visualizing the Window: Example

Sender to Receiver → Send 7, 8, 9



Visualizing the Window: Example

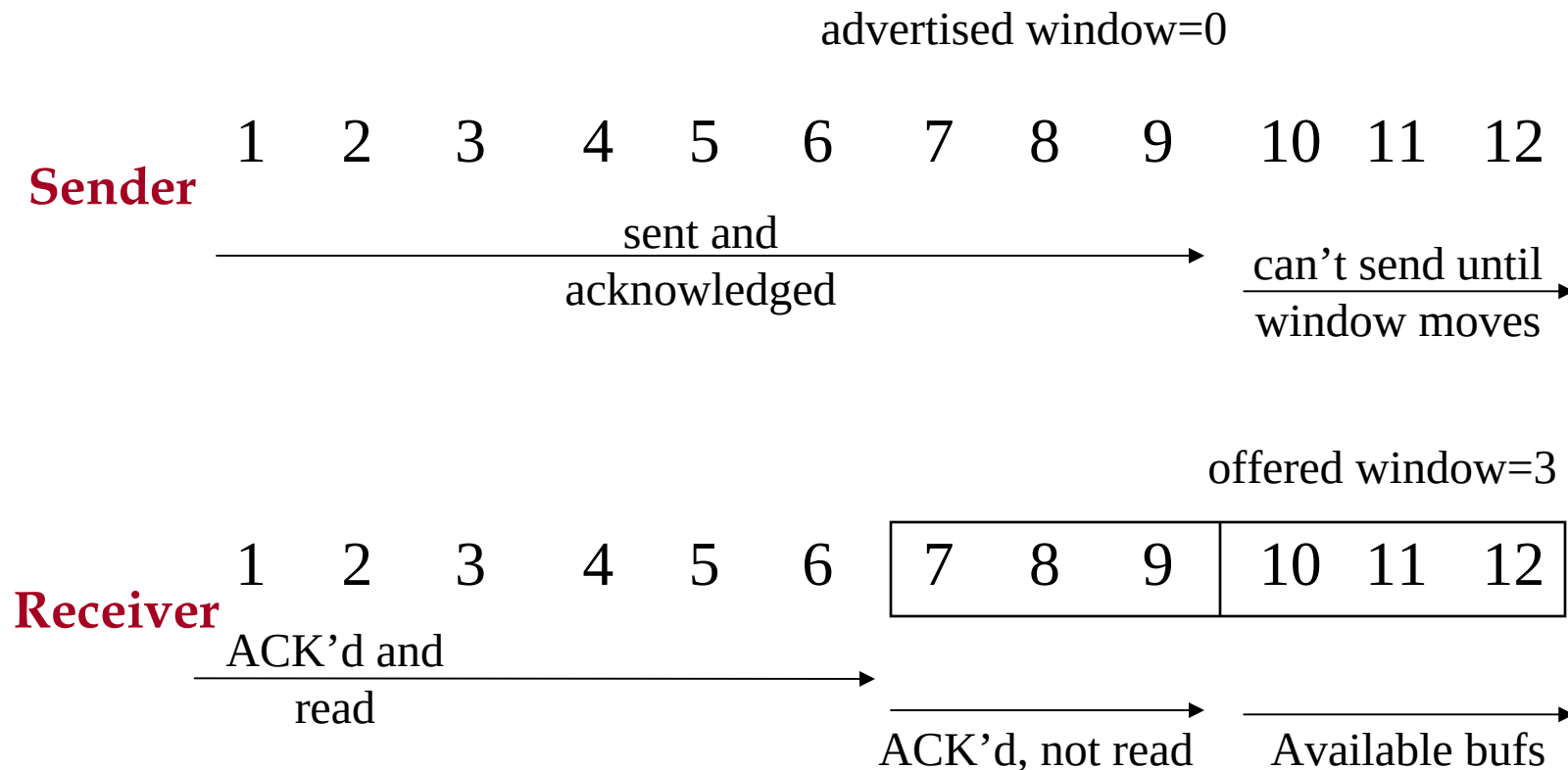
Receiver to Sender → ACK 9, Window 0



Visualizing the Window: Example

Receiver App reads packets 4, 5, 6

But sender has no way of knowing that more room is available!



Options for Sender Discovery of Increased Advertised Window

- Receiver sends duplicate ACK with a larger advertised window
 - Complicates receiver design
 - TCP design philosophy: keep receiver simple

Also explains slow deployment of SACK, NACK, etc.
- Sender periodically transmits a 1-byte packet
 - If no space available at receiver → packet dropped, no ACK
 - If additional space became available → ACK contains new advertised window
- NOTE: advertised window in bytes, not packets