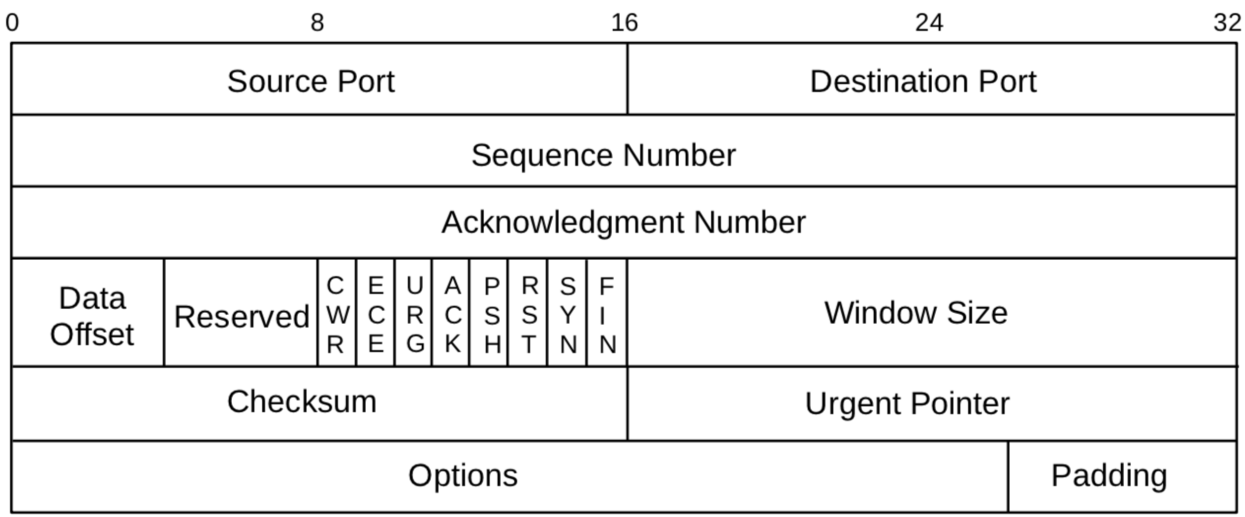
Computer Networking

COMP 177 | Fall 2020 | University of the Pacific | Jeff Shafer

TCP (4)

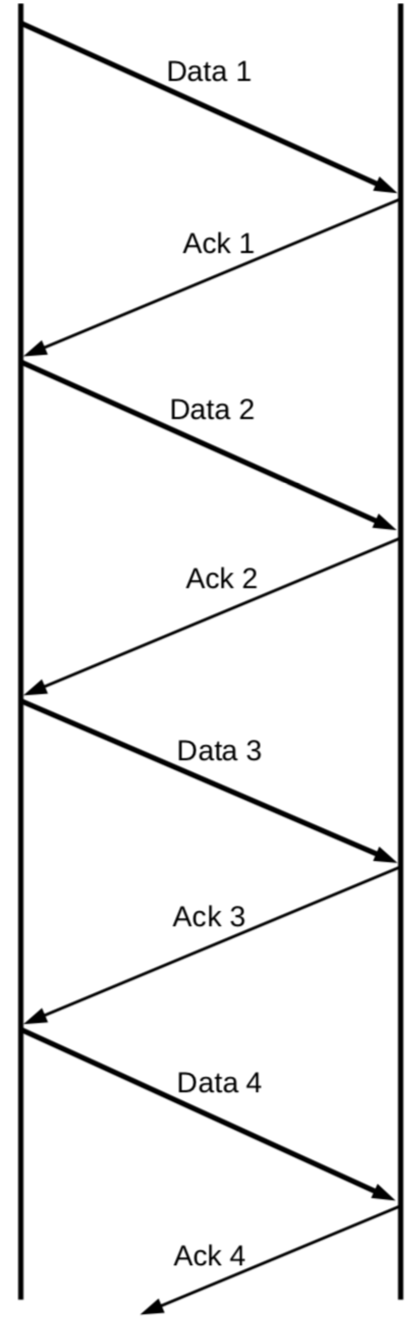
Transmission Control Protocol

Transmission Control Protocol (TCP)



- Connection oriented
- Byte streaming
- Full duplex
- Reliable data transport
- Congestion control
- Flow control

Stop-and-Wait



Sliding Windows

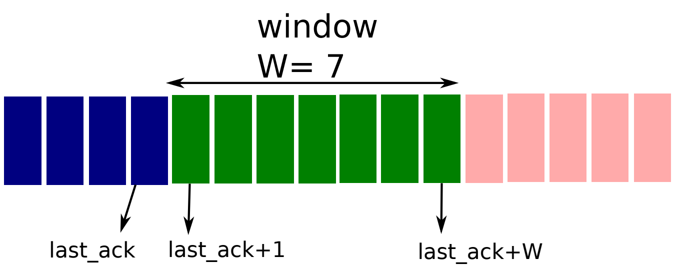
- Stop-and-wait protocols are not efficient as the network is idle most of the time
- Solution: Let the sender send packets back-to-back before waiting for their acknowledgements
  - Constraint: we cannot let the sender get too far ahead before receiving the acknowledgements
  - The limit defines a window size, **W**
  - A window is a queue of packets that can be sent back-to-back without waiting for the acknowledgements
- **Sliding windows**
  - The sender is allowed to send *window size* number of packets before waiting for the acknowledgment of the first packet in the window

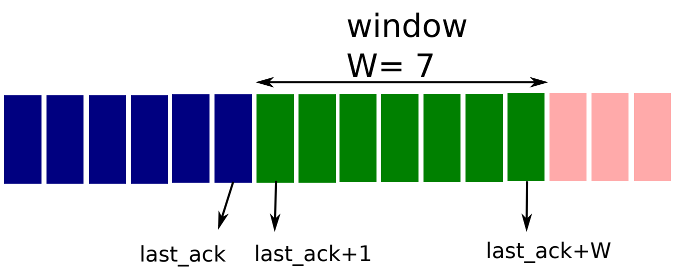
Sliding Windows

- Color scheme used for presentation of sliding windows
  - **Blue: region of buffer that *precedes* the sliding window**
  - **Green: region of buffer that refers to the sliding window**
  - **Pink: region of buffer that comes *after* the sliding window**
  - **Red: region of buffer within the sliding window that has been sent/received**

Sliding Windows: Sender Window

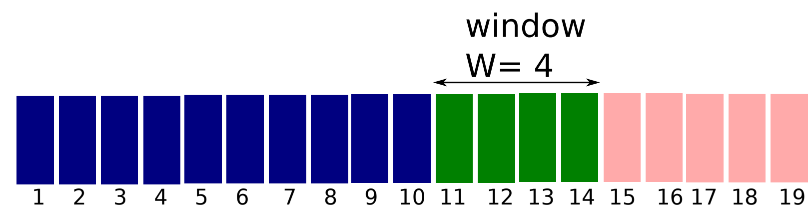
- The sender keeps a state variable `last_ack` representing the last packet for which it has received an acknowledgement
- The sender then sends packets with sequence numbers `last_ack+1` up to `last_ack+W`
  - This range defines the *sender window*
- If Ack[N] arrives and N > `last_ack` then `last_ack` becomes N
  - Acknowledgements are cumulative
  - The window "slides forward", and the sender may send more packets



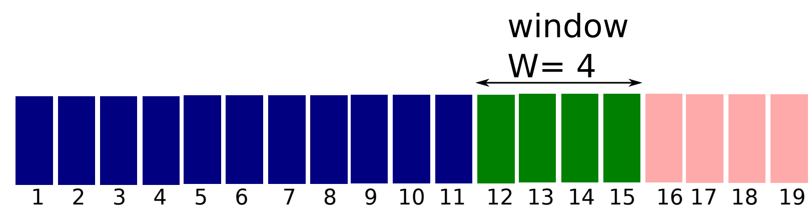


Sender Window: Example

- Assume that at the sender side,
  - Window size is 4, i.e., W = 4
  - Last acknowledged packet is 10 (i.e., `last_ack` = 10)
  - Sender window →

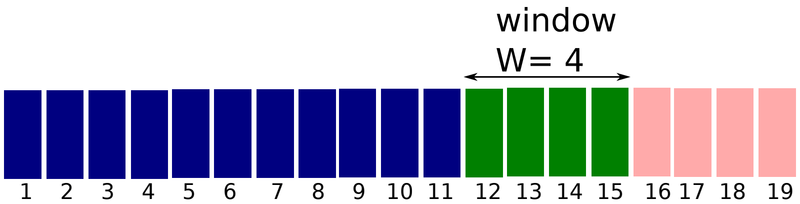


- If `Ack[11]` arrives
  - `Last_ack` gets updated to 11
  - Window slides forward by 1
  - Sender window →
  - `Data[15]` can be sent since it is in the window

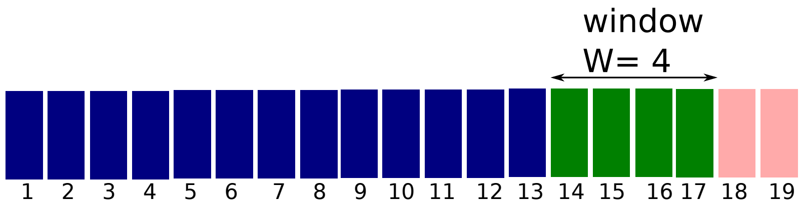


Sender Window: Example

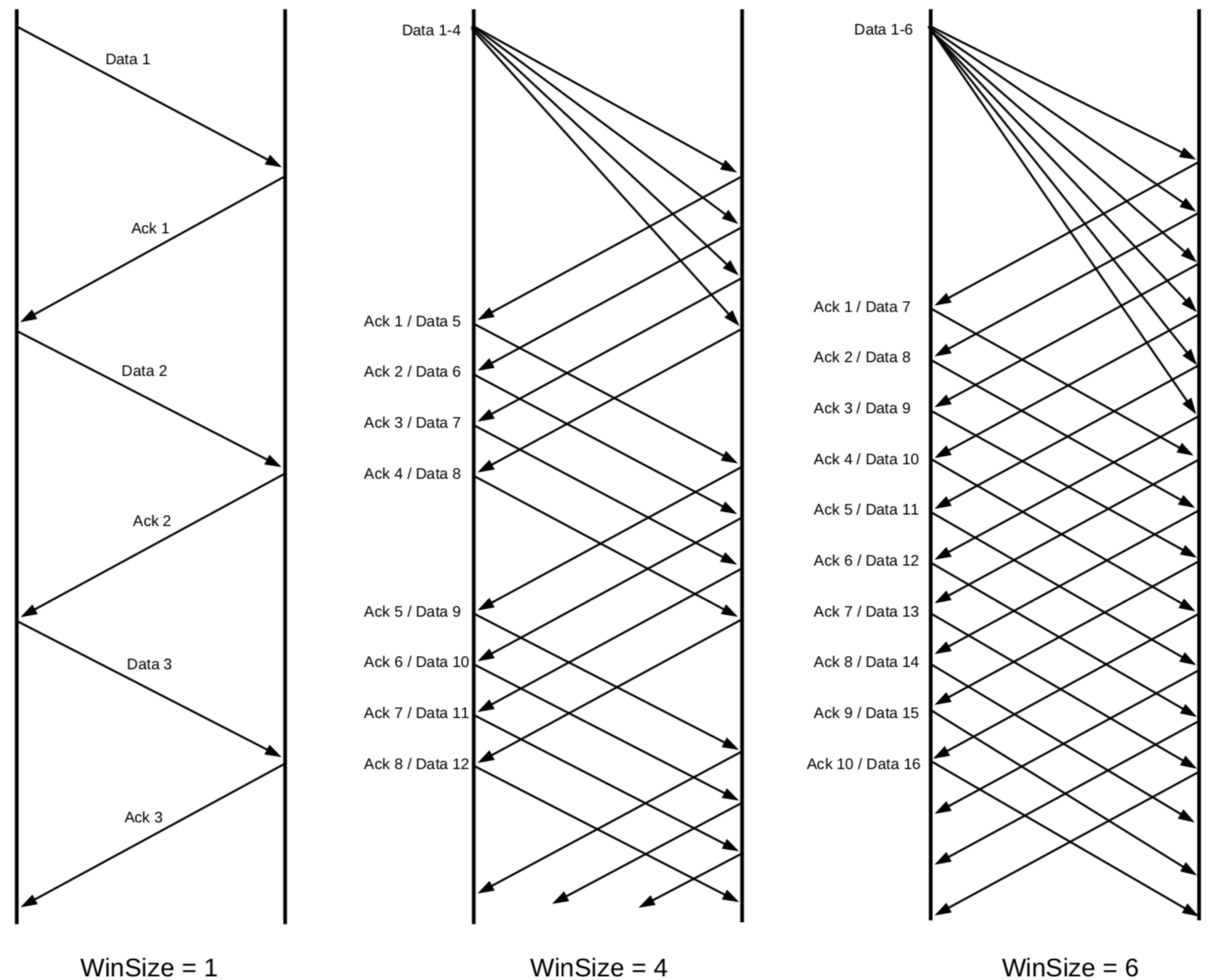
- ...



- If Ack[13] arrives
  - Last_ack gets updated to 13
  - Window slides forward by 2
  - Sender window →
  - Data[16-17] can be sent since they are in the window

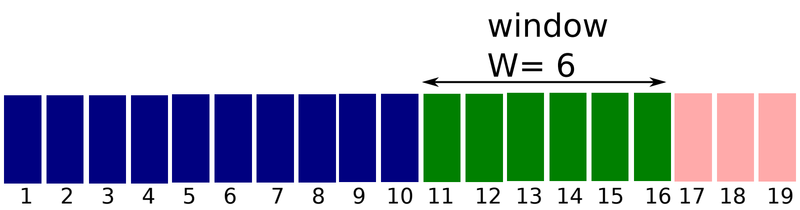


Sender Window Size Comparison



Sliding Windows: Receiver Window

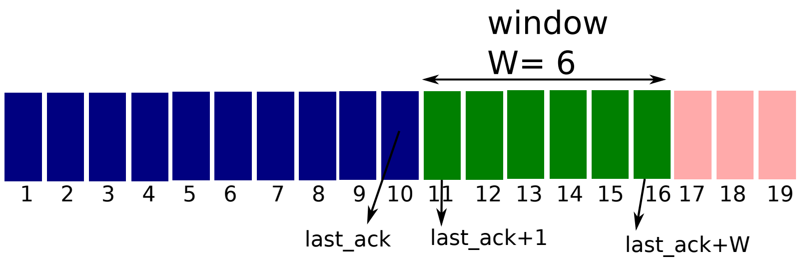
- Sliding sender window and timeout mechanism can *detect packet loss* and initiate retransmission
- However, it does not prevent *out-of-order receipt* of packets at the receiver side
  - The receiver also needs a window of potentially the same size as sender's W
  - The receiver buffers out-of-order packets whose sequence numbers are in the receiver window.
- Example: Assume that the receiver window is →



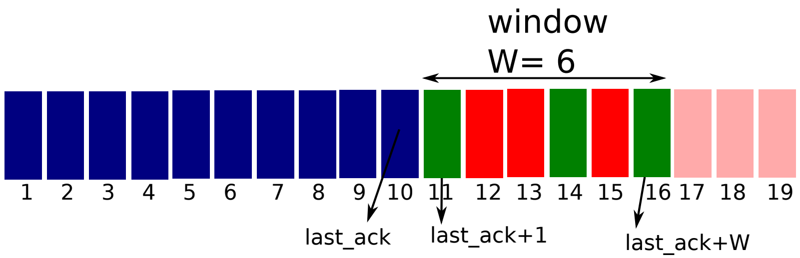
  - The receiver can receive packets 11 to 16
  - If `Data[11]` is delayed, but `Data[12]` through `Data[16]` are received, then these out-of-order packets are buffered until *Data[11]* arrives

Sliding Windows: Receiver Window

- Similar to sender, the receiver keeps the state variable `last_ack`
- Receiver's `last_ack` is not necessarily in sync with sender's `last_ack`
- At any instance, the receiver is willing to receive packets `last_ack+1` to `last_ack+W`
  - This defines the *receiver window*

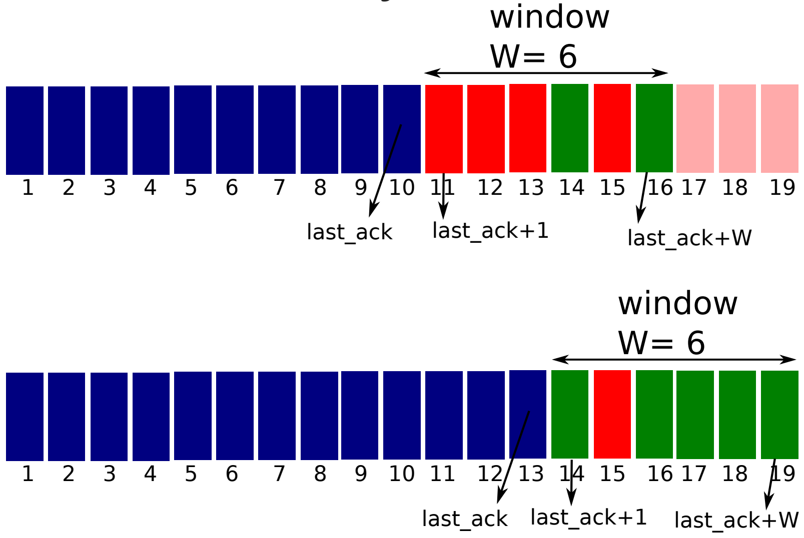


- If the receiver receives packets in the range `last_ack+2` to `last_ack+W`, the packets would be buffered



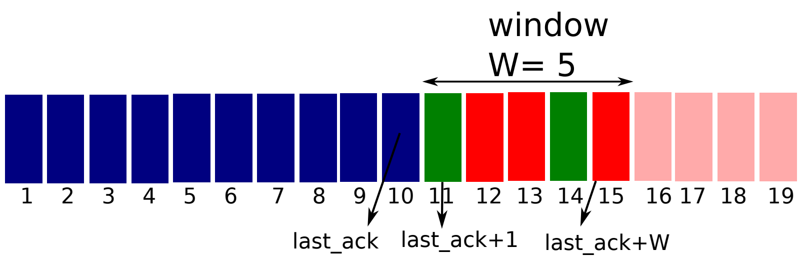
Sliding Windows: Receiver Window

- Upon receiving packet `last_ack+1`, the receiver
  - Examines the buffered packets,
  - Sends the largest cumulative acknowledgement to the sender
  - Updates `last_ack` to the largest cumulative acknowledgement number
  - This *slides the receiver window forward*

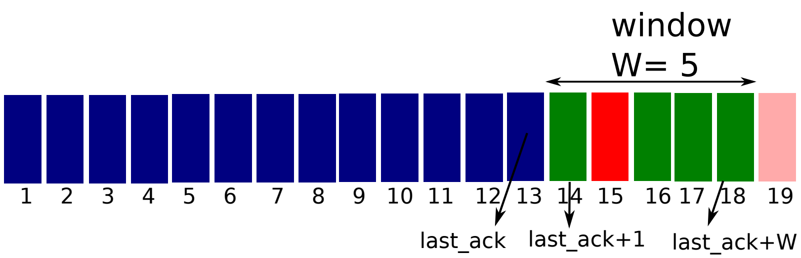


Receiver Window: Example

- Assume that at the receiver
  - Window size is 5, i.e., W=5
  - `last_ack=10`
  - Packets Data[12], Data[13], and Data[15] are already received and buffered
  - Receiver window→

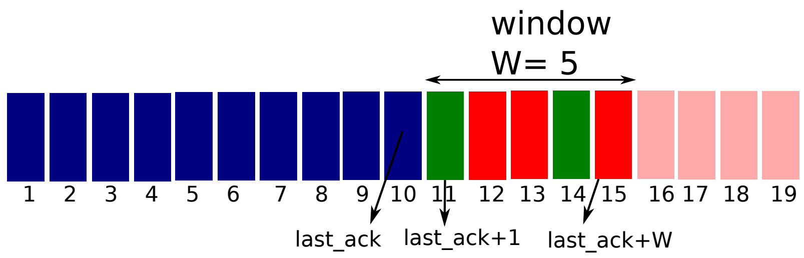


- If Data[11] arrives, then
  - Ack[13] is sent
  - `last_ack` becomes 13
  - Receiver window slides forward →



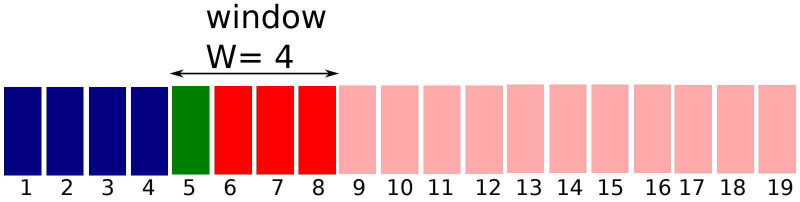
Cumulative Acknowledgements

- Note that acknowledgements are cumulative.
- Receiving out-of-order packets generates Ack[`last_ack`]
- For instance, in the previous example, upon receiving each of Data[12], Data[13], and Data[15], the receiver sends back Ack[10]



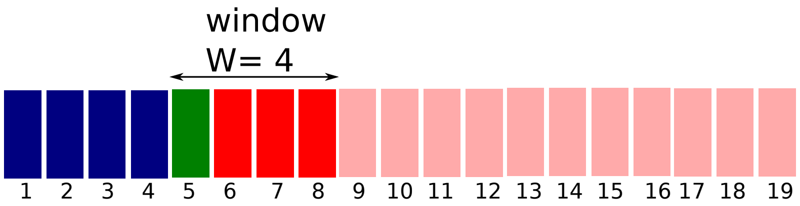
Example: Loss Recovery

- Assume scenario
  - W=4 at both ends
  - Sender has sent packets Data[5] through Data[8]
  - Data[5] is lost and all others have been received
- Result
  - Upon the arrival of each of packets Data[6], Data[7], and Data[8], the receiver has sent Ack[4]
  - Packets Data[6], Data[7], and Data[8] are buffered at the receiver
  - Receiver window →

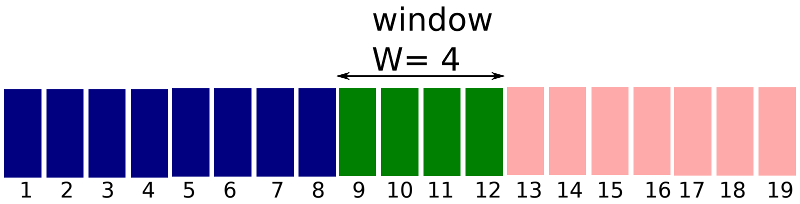


Example: Loss Recovery

- ...



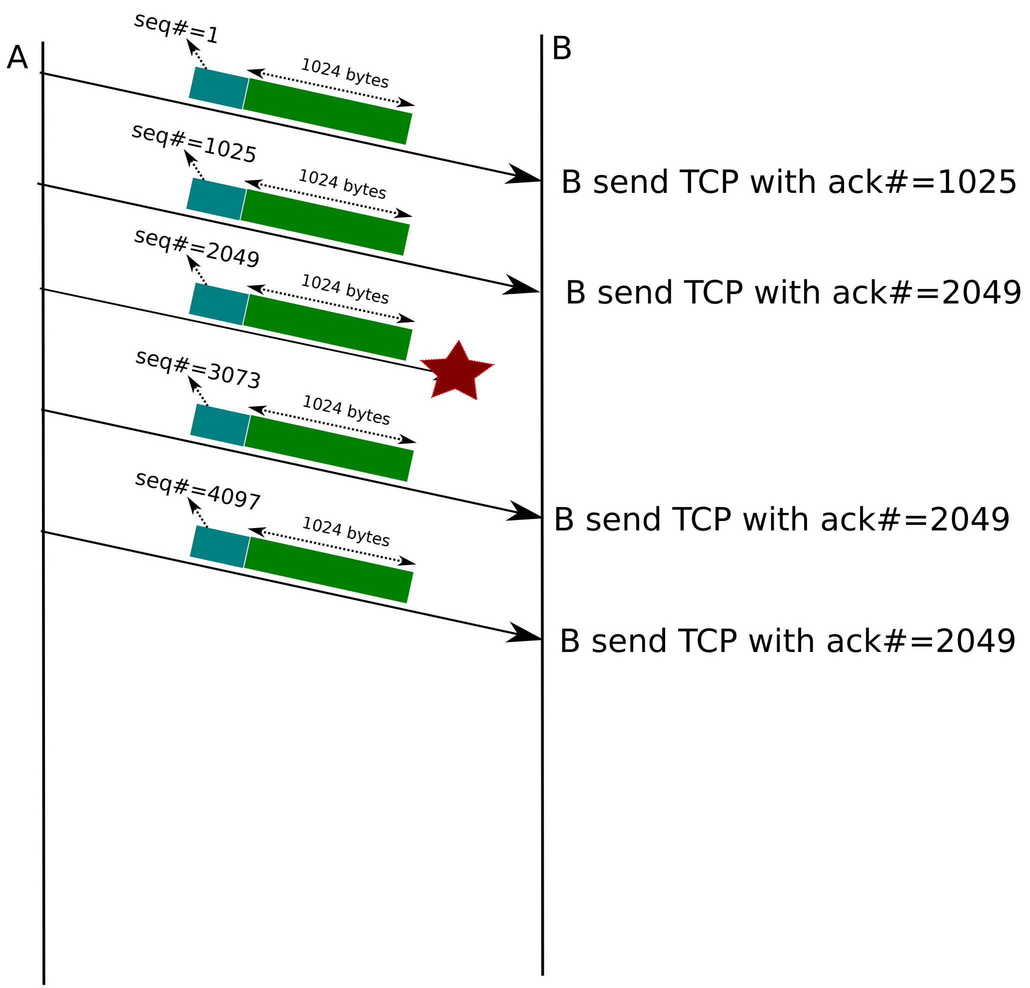
- When the sender times out, it only sends Data[5]
  - If it arrives at the receiver, Ack[8] will be sent
  - Receiver window →



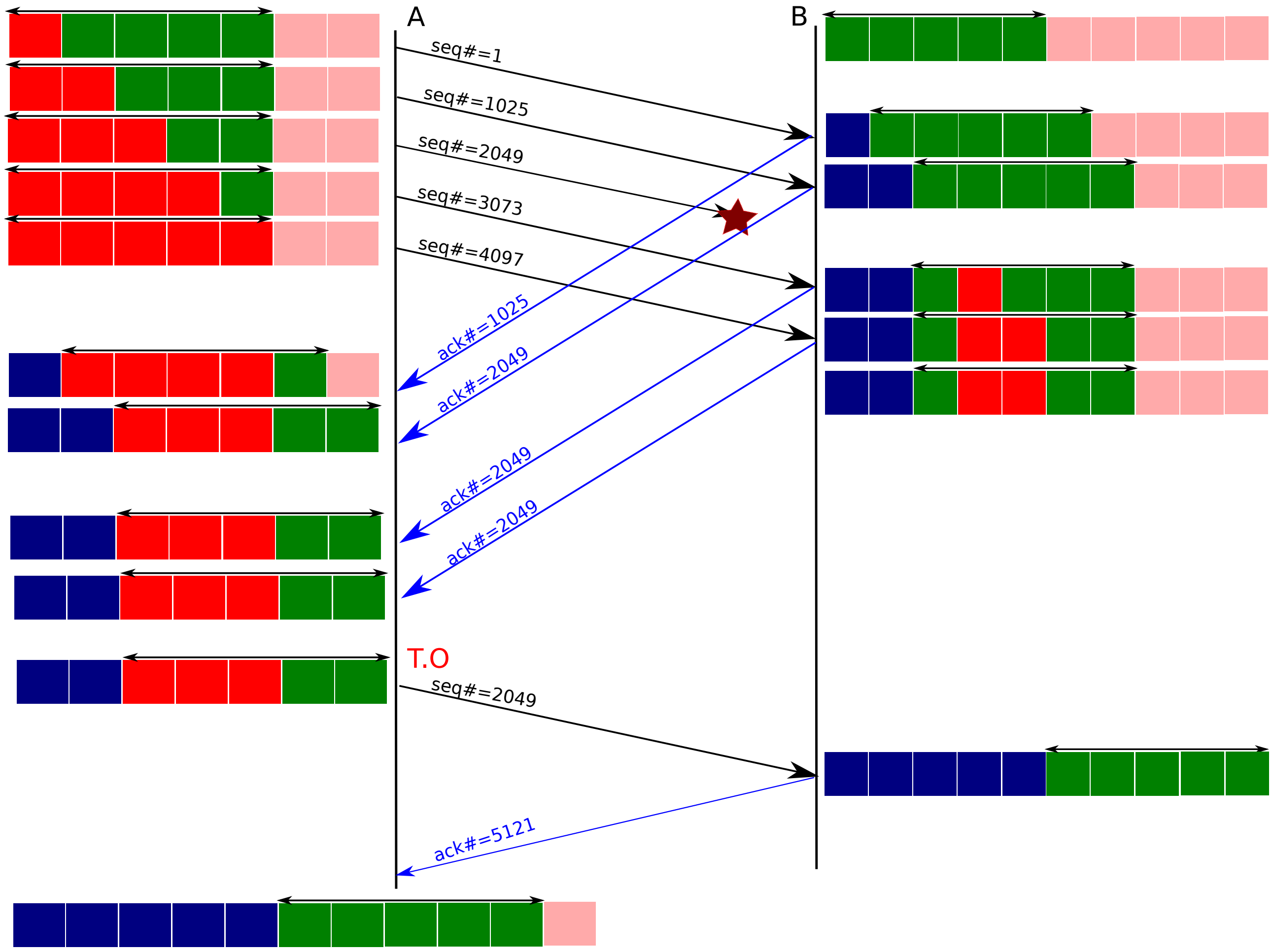
Sliding Windows in TCP

- TCP uses sliding windows to improve *throughput*
- However, TCP is a *byte stream* protocol
  - Sequence numbers and acknowledgement numbers refers to *bytes*, not packets.
  - Window sizes are measured in terms of bytes, not packets
- Each side announces its initial window size in the 3-way handshake using "Window Size" field in TCP
  - *Window Scale* option may be used to scale the window size.
- Example: Let's revisit an earlier TCP communication
  - Suppose A sends 5 TCP packets to B.
  - 3^{rd} packet is lost in the path, so B doesn't receive it
  - Upon receiving the 4^{th} and 5^{th} packets, B still acknowledges the receipt of every byte up to the last byte in the 2nd packet
  - In addition let's assume that window size for both sender and receiver is $5 \times 1024 = 5120$ bytes

TCP Sliding Window Example



TCP Sliding Window Example



TCP Flow Control

- It is possible that the sender sends faster than the *receiver* can process
- In order to avoid flooding the receiver, TCP provides *flow control*
- Window size determines the amount of bytes that are in flight at any time
- This parameter can be adjusted for flow control
- Flow control:
  - If the receiver is processing slower than it receives packets, it advertises a reduced window size in the acknowledgement
    - Using Window Size header field in TCP
  - The sender then reduces its own window size accordingly
- If receiver advertises window size as 0, the sender stops sending packets until the receiver advertises a larger window size later within an acknowledgement

TCP Congestion Control

- It is possible that the sender sends faster than the *intermediary nodes* can process
- Similar to flow control, TCP congestion control is window based.
- TCP computes *congestion window size* (cwnd) according to the level of perceived congestion in the network
- The actual window size is minimum of
  - Advertised window size in “Window Size” header field, and
  - Congestion window size (cwnd)
- When a packet *times out* at the sender side, i.e., not acknowledged before timeout, the sender interprets it as a packet loss
- Packet loss is interpreted as congestion in the network

TCP Congestion Control

- Another measure used by TCP is *three duplicate acknowledgements*
  - Rather than waiting for timeouts, if a packet receives three duplicate acknowledgements, it is considered lost at that point
- Upon sensing congestion (either by timeout or three duplicate acknowledgements), TCP adjusts cwnd
  - If congestion is not sensed, cwnd grows gradually.
  - The growth continues until packet loss happens.
  - Then, cwnd drops to smaller size

Closing Thoughts

Recap

- Today we discussed
- Sliding windows
  - Sender window
  - Receiver window
- TCP sliding window
- TCP flow control
- TCP congestion control

Next Class

- Project Work Day (Tuesday)
- Port Scanning (Thursday)

Project 4

Due Nov 18th

Presentation

Due Nov 23rd