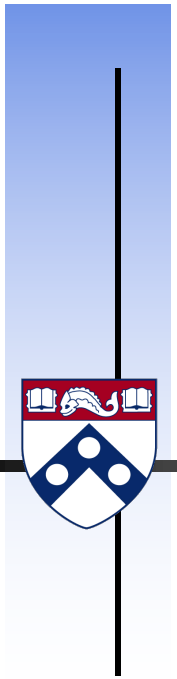


CIS 5530: Networked Systems

Transport Basics

March 15, 2023

Exam 1 Grades

- You all did well!
  - Max: 75.5 / 80 (94.4%)
  - Mean: 60.4 / 80 (75.6%)
  - Stddev: 8.08 (10.1%)

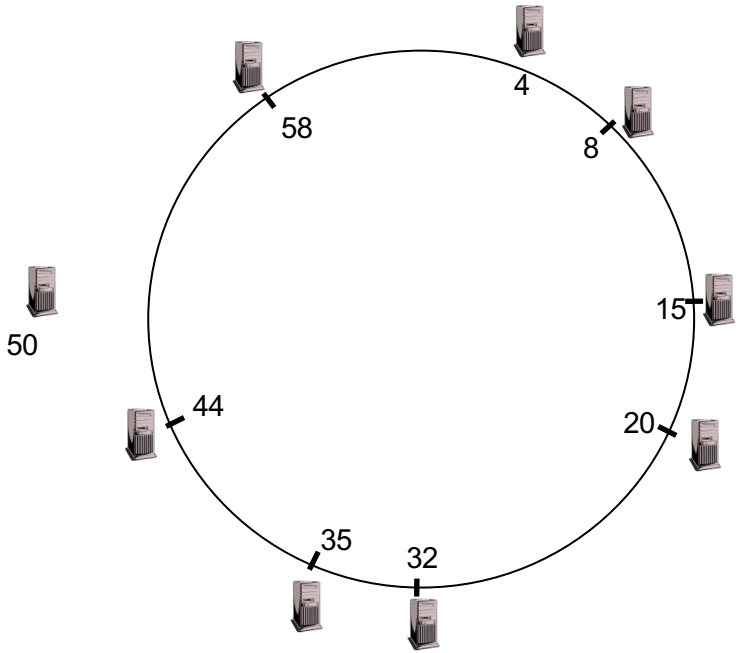
- Curved = x + (30% * [80 – x]) + (5.75)

Agenda

- Overlay Networks ✓
  - Chord ✓
  - PennSearch ✓
- Transport Layer
  - UDP
  - TCP

Review

- What information do nodes need to store?
- What is filled in after *node50*`.join(node`*15*`)`?
- What is filled in if *node50*`.stabilize()` is called immediately afterwards?



Agenda

- Transport Layer 

  - UDP
  - TCP

Layer 4: Transport layer

- On every host, typically not in the network
- Name of message
  - UDP: Datagrams
  - TCP: Segments
- Lower interface: A packet of data
- Upper interface: Two options
  - UDP: Chunks of data
  - TCP: A stream of bytes

What is the purpose of L4?

- Transport layer is where we "pay the piper"
  - Provide applications with good abstractions
  - Mostly without support or feedback from the network

Role of the transport layer

- Communication between sockets
  - Mux and demux from/to application-level sockets
  - Using L4 ports (NOT the same as L1 ports)

Role of the transport layer

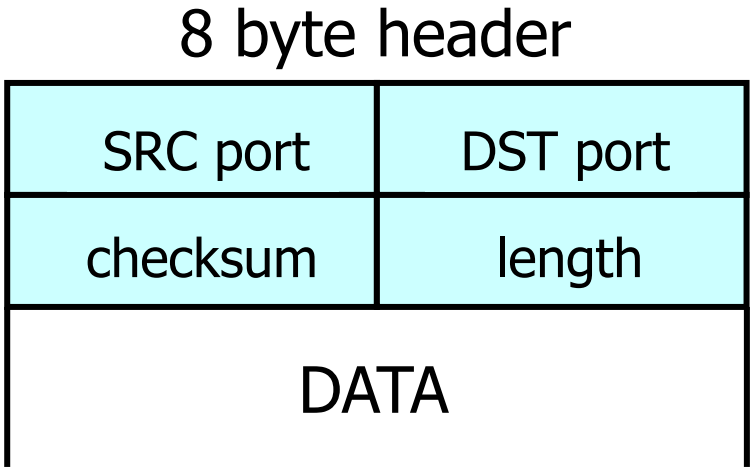
- Communication between processes
- Provide common end-to-end services for application layer [optional]
  - Reliable, in-order data delivery
  - Well-paced data delivery
    - Too fast may overwhelm the network
    - Too slow is not efficient

Role of the transport layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- UDP and TCP are the common transport protocols
  - Also SCTP, MPTCP, SST, RDP, DCCP, …

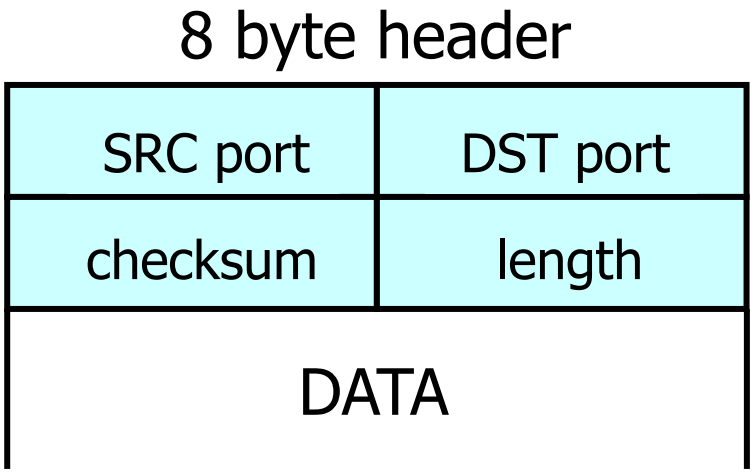
User Datagram Protocol (UDP)

- Lightweight communication between processes
  - Send and receive messages
  - Just a simple wrapper around IP
- Used by popular apps
  - Query/response for DNS
  - Real-time data in VoIP



UDP (cont'd)

- Optional error checking on the packet contents
  - (checksum field = 0 means "don't verify checksum")
- Source port is also optional
  - Useful to respond back to the sender in some cases



Why Use UDP?

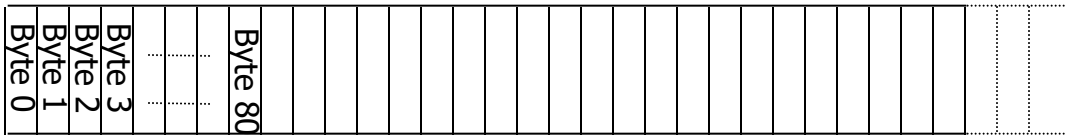
- Fine-grained control
  - UDP sends as soon as the application writes
- No connection set-up delay
  - UDP sends without establishing a connection
- No connection state
  - No buffers, parameters, sequence #s, etc.
- Small header overhead
  - UDP header is only eight-bytes long

Transmission Control Protocol (TCP)

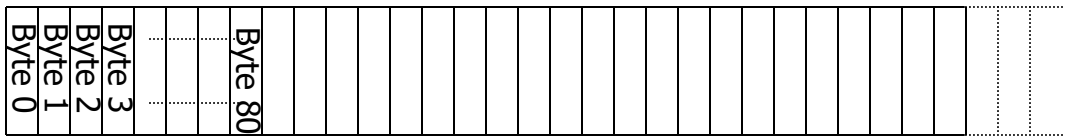
1. Stream-of-bytes service
   - Sends and receives a stream of bytes
2. Reliable, in-order delivery
   - Detect corruption, loss, and reordering
   - Reliable delivery: acknowledgments and retransmissions
3. Connection-oriented
   - Explicit set-up and tear-down of TCP connection
4. Flow control
   - Prevent overflow of the receiver's buffer space
5. Congestion control
   - Adapt to network congestion for the greater good

TCP's "stream of bytes" model

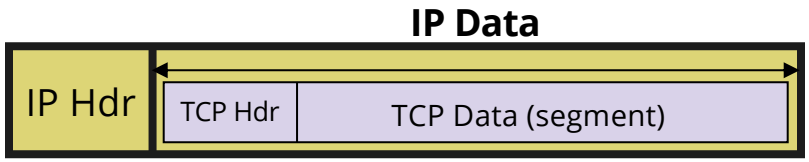
Host A



Host B



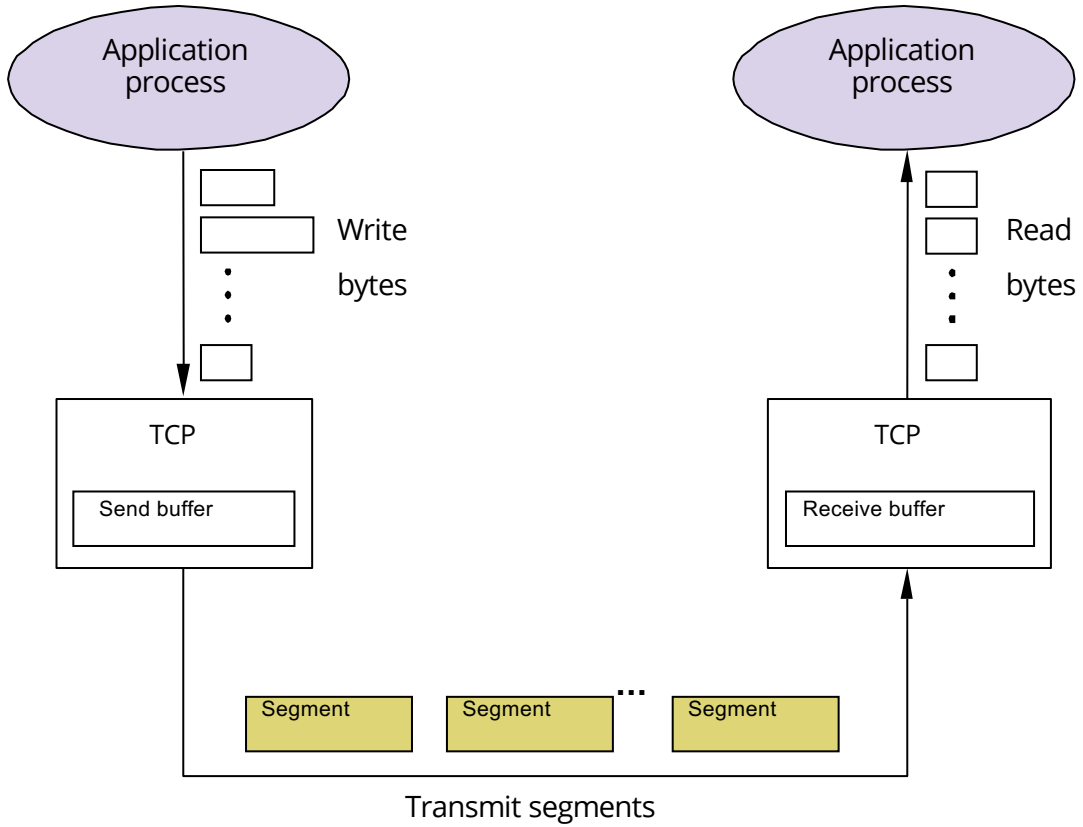
TCP Segment



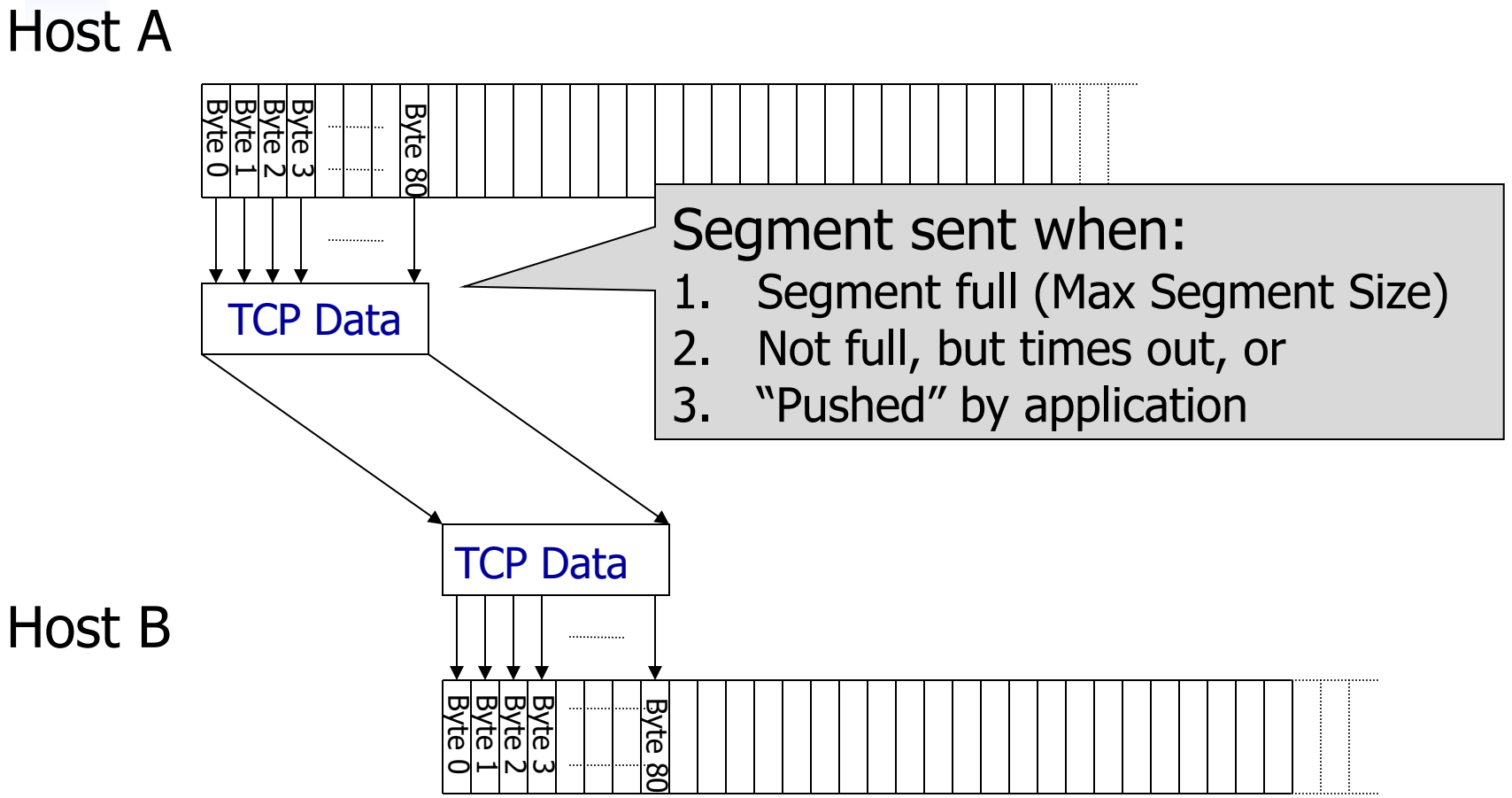
- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

TCP Byte Stream

IP	TCP/UDP	data



…Emulated Using TCP "Segments"



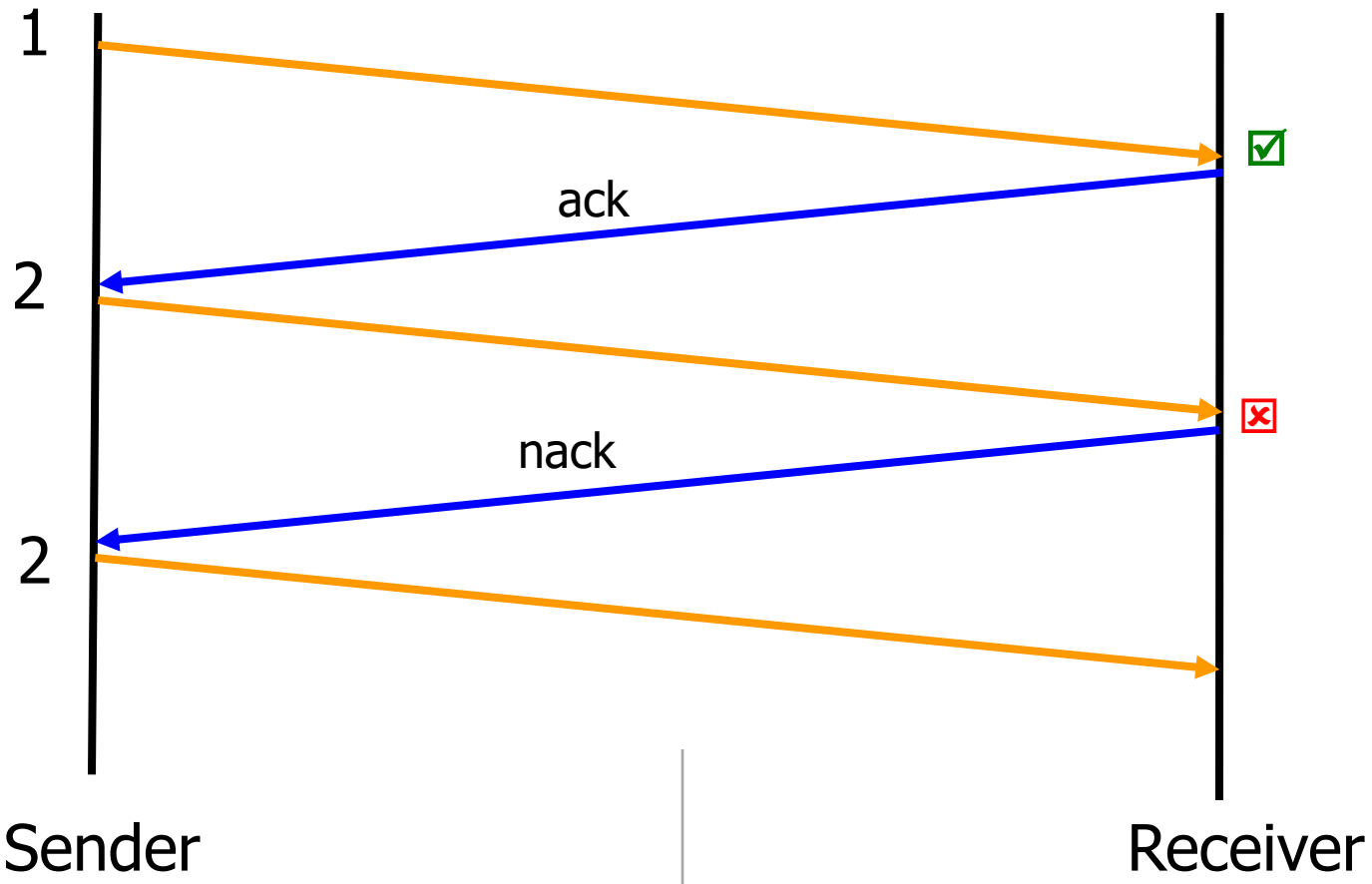
Transmission Control Protocol (TCP)

1. Stream-of-bytes service
   - Sends and receives a stream of bytes
2. Reliable, in-order delivery
   - Detect corruption, loss, and reordering
   - Reliable delivery: acknowledgments and retransmissions
3. Connection-oriented
   - Explicit set-up and tear-down of TCP connection
4. Flow control
   - Prevent overflow of the receiver's buffer space
5. Congestion control
   - Adapt to network congestion for the greater good

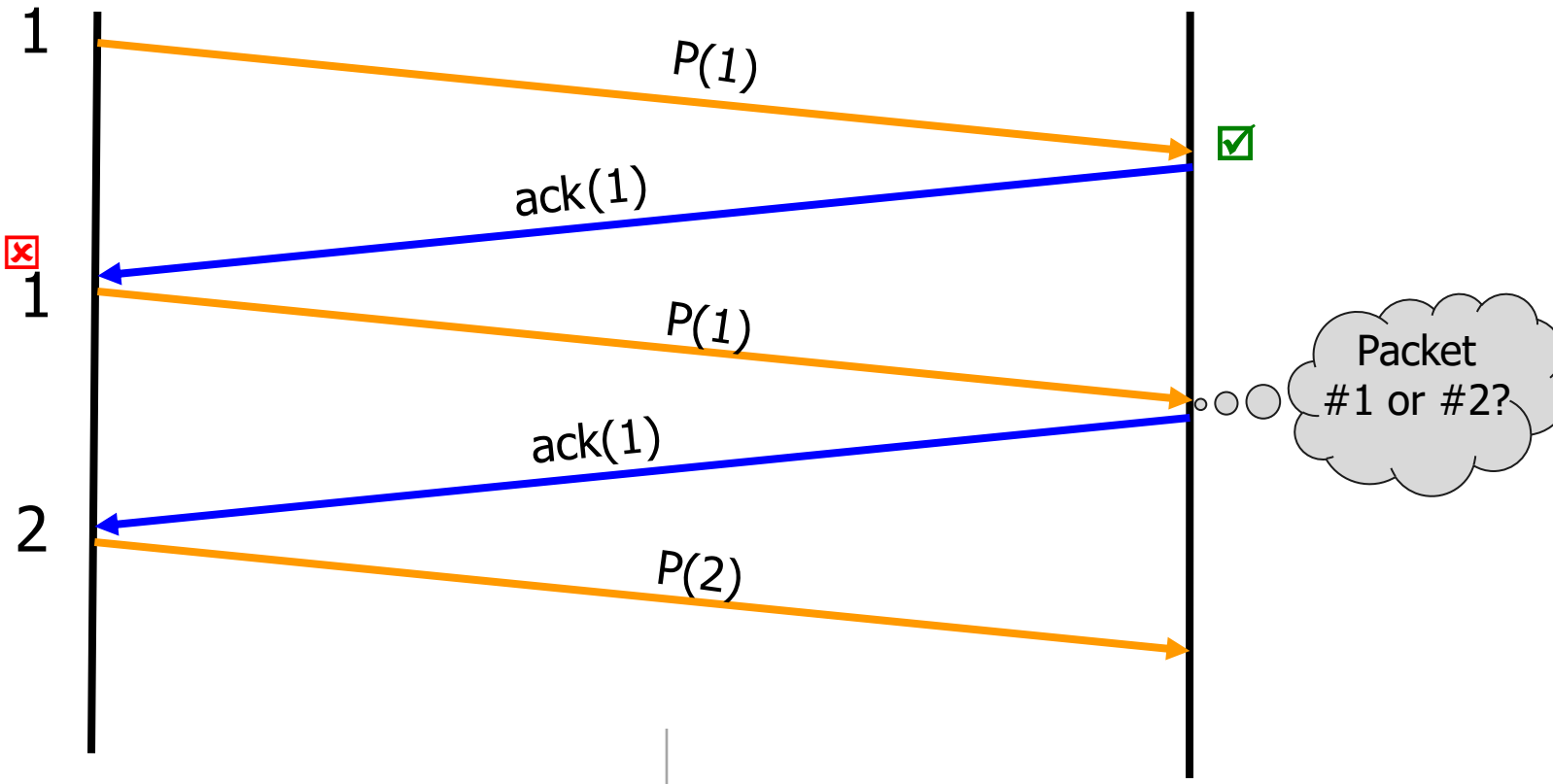
Challenges of Reliable Data Transfer

- Over a perfectly reliable channel? Done!
- What if packets can experience bit errors?
- What if packets can be lost?
- What if packets can be reordered?

Dealing with packet corruption



Dealing with packet corruption

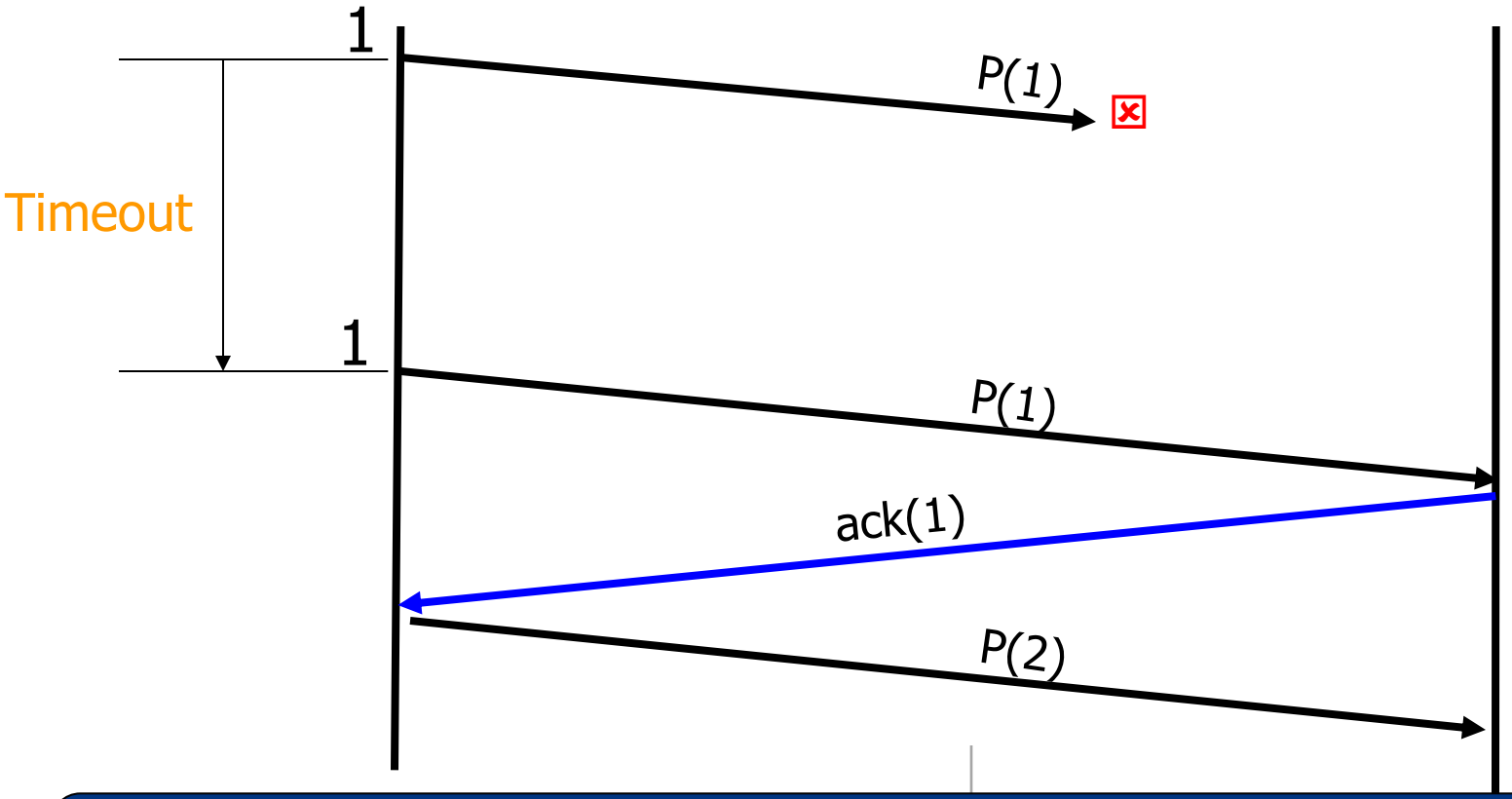


What if the ACK/NACK is corrupted?

Components of a solution

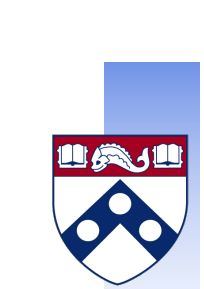
- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
- Sequence numbers (to deal with duplicates)

Dealing with packet loss

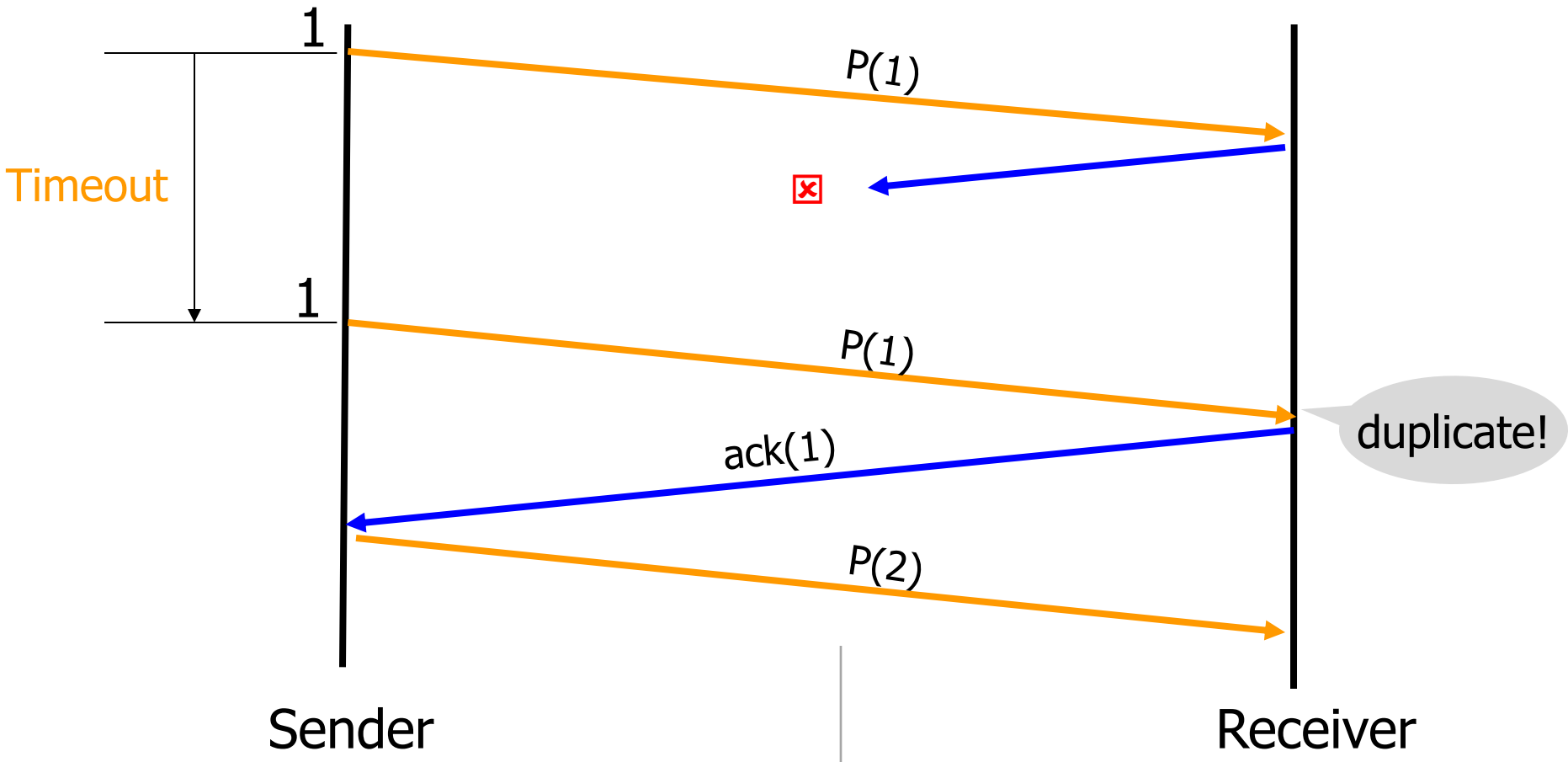


Timer-driven loss detection

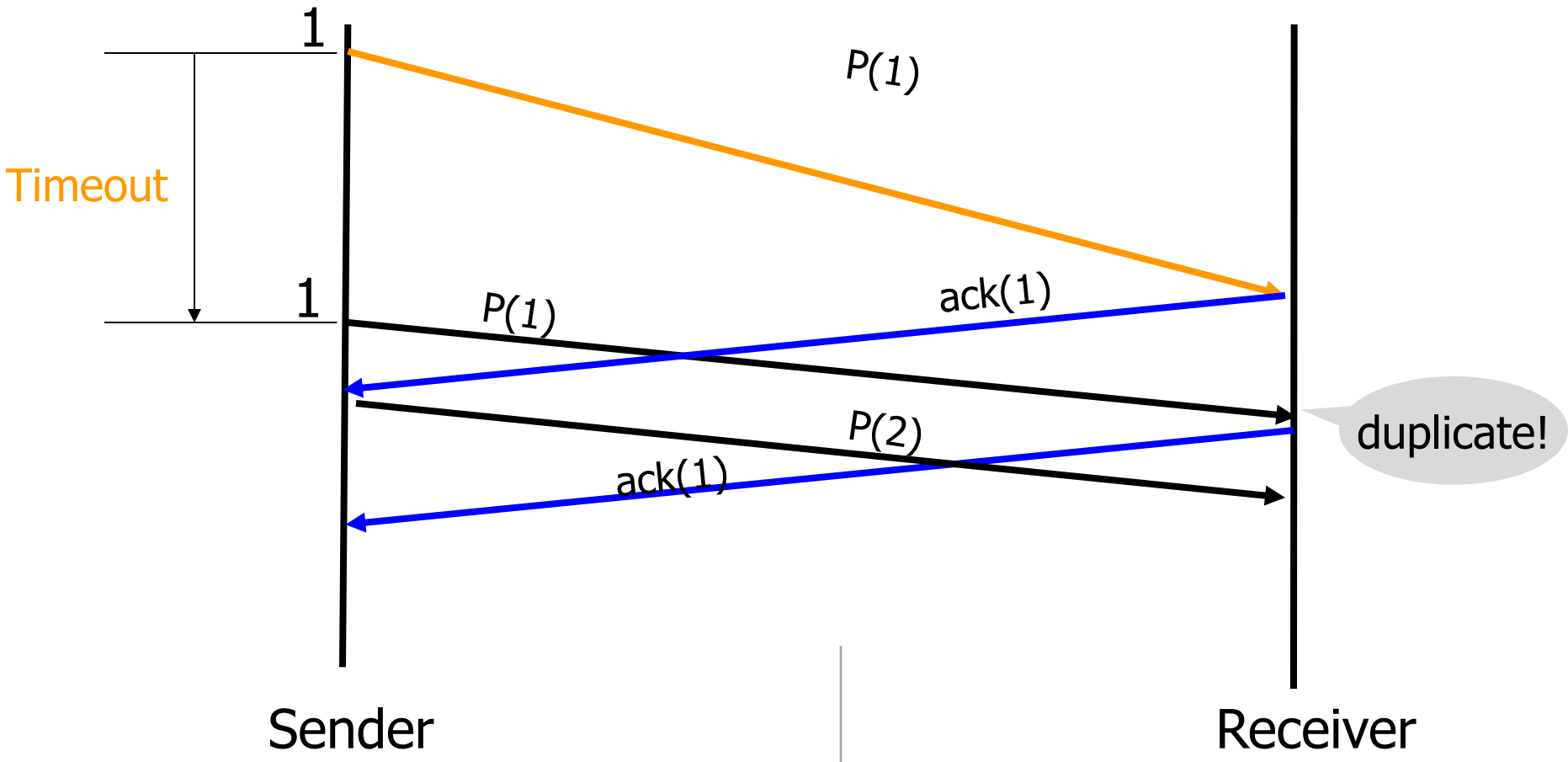
Set timer when packet is sent; retransmit on timeout



Dealing with packet loss (of ack)



Dealing with packet loss



Components of a solution

- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
- Sequence numbers (to deal with duplicates)
- Timers (to detect loss)

A Solution: "Stop and Wait"

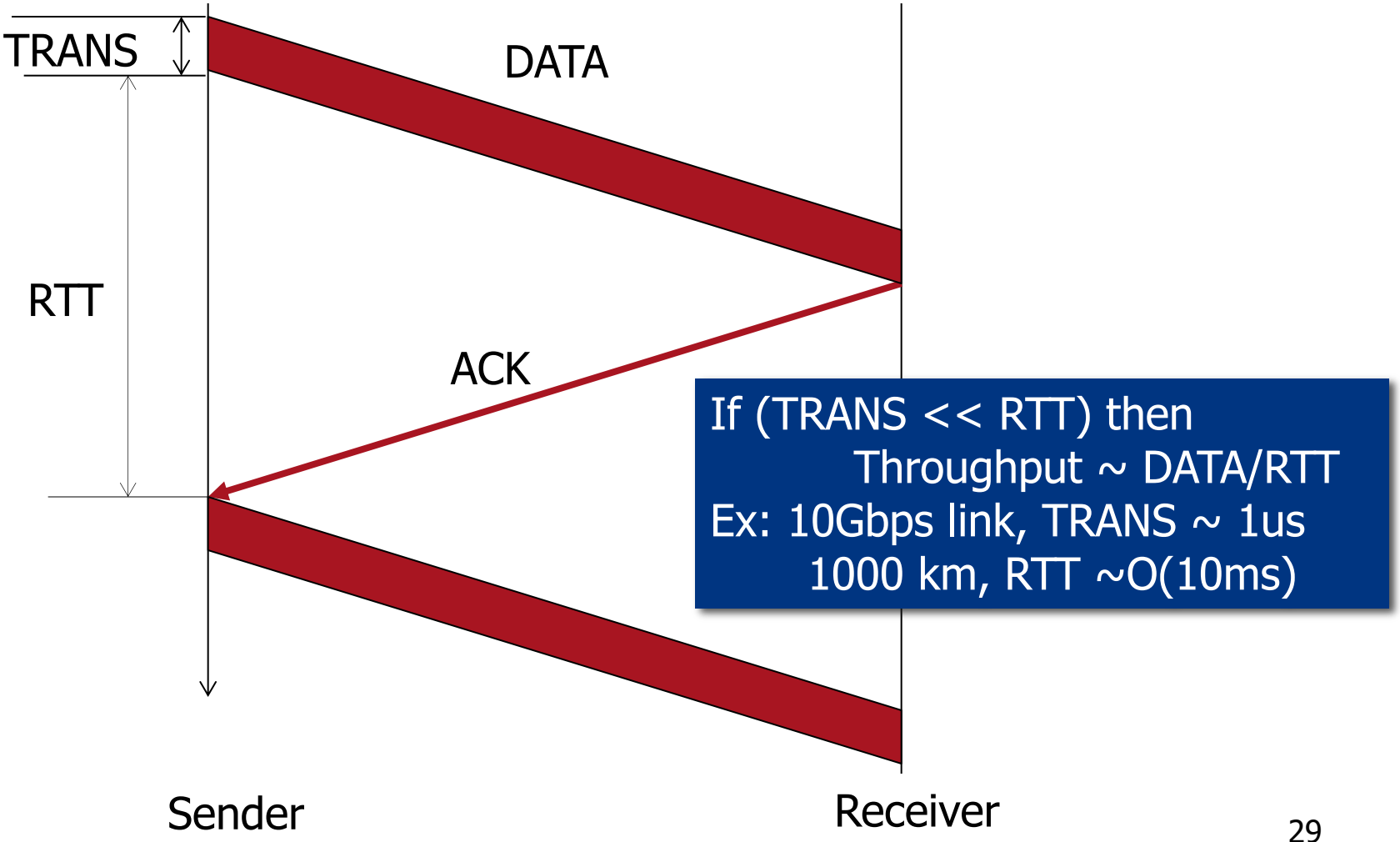
@Sender

- Send packet(I); (re)set timer; wait for ack
- If (ACK)
  - I++; repeat
- If (NACK or TIMEOUT)
  - repeat

@Receiver

- Wait for packet
- If packet is OK, send ACK
- Else, send NACK
- Repeat

Stop and Wait: correct, but inefficient



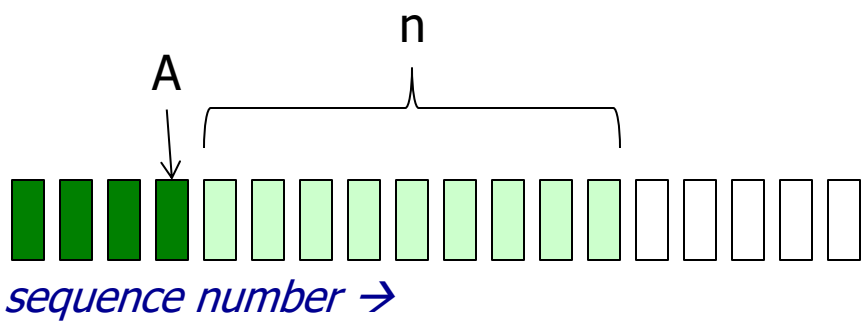
If (TRANS << RTT) then
Throughput ~ DATA/RTT
Ex: 10Gbps link, TRANS ~ 1us
1000 km, RTT ~O(10ms)

Sliding window

- Window = set of adjacent sequence numbers
  - Assume size of window is n
- General idea: send up to n packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets "slides" on successful reception/acknowledgement
  - Window contains all packets that might still be in transit
- Sliding window often called "packets in flight"

Example of sliding window

- Let A be the last ACK received;
  then sender window = {A+1, A+2, …, A+n}

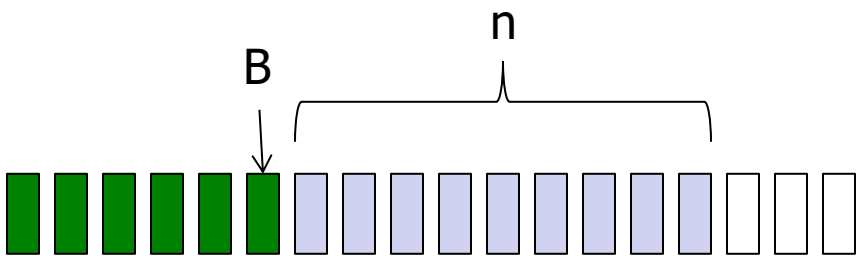


Already ACK'd

Sent but not ACK'd

Cannot be sent

- Let B be the last received packet that we ACK'd;
  then receiver window = {B+1, …, B+n}



Received and ACK'd

Acceptable but not yet received

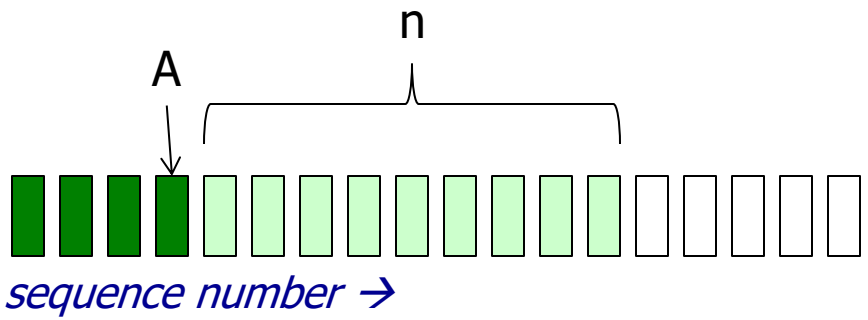
Cannot be received

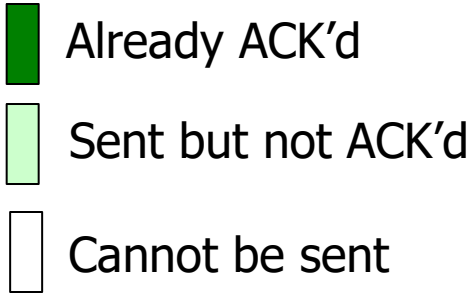
Throughput of sliding window

- If window size is n, then throughput is roughly
  - min(n*DATA/RTT, Link Bandwidth)
- Compare to Stop and Wait: Data/RTT
- Why wouldn't we set n to be very large?

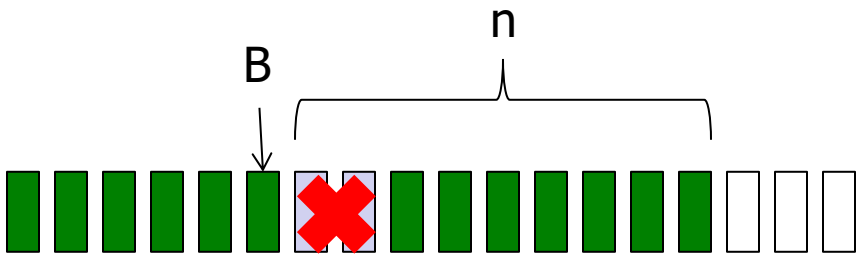
What happens when there is a loss?

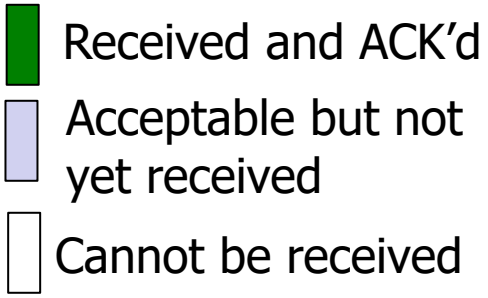
- Let A be the last ACK received;
  then sender window = {A+1, A+2, …, A+n}





- Let B be the last received packet that we ACK'd;
  then receiver window = {B+1, …, B+n}





Go-Back-N (GBN)

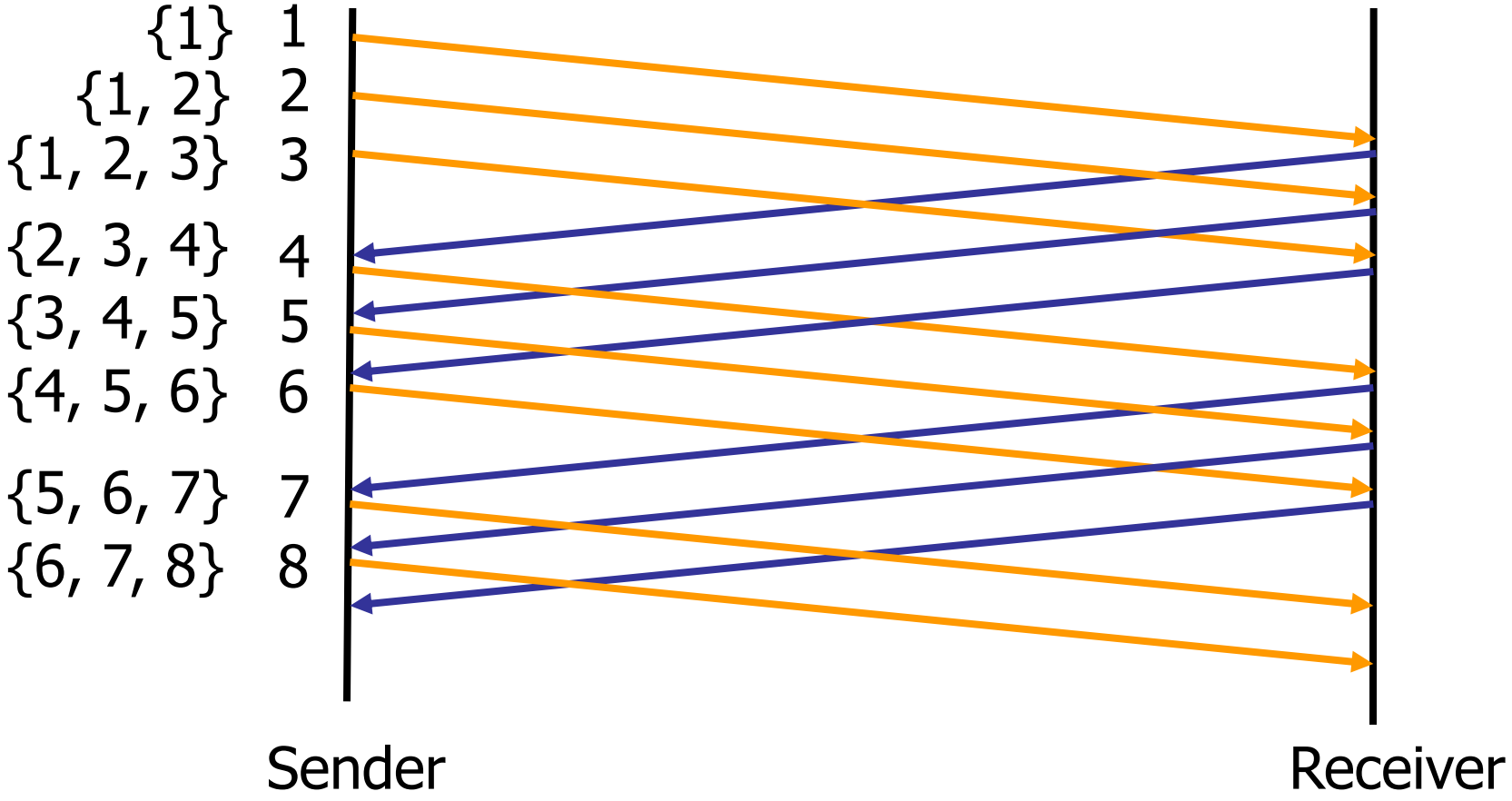
- Sender transmits up to n unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses cumulative acknowledgements
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ack (A+1)
- If timeout, retransmit A+1, … , A+n

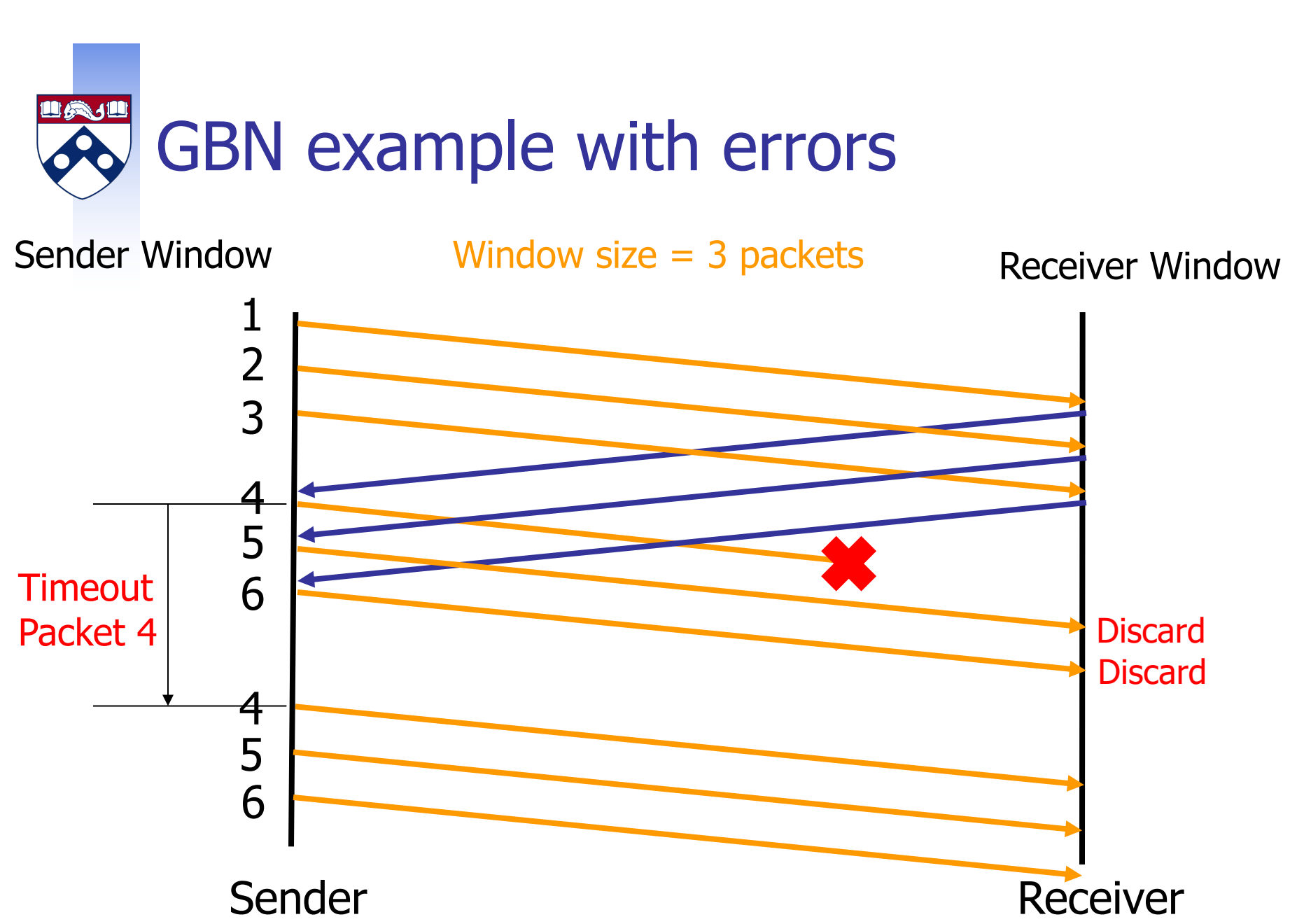
GBN example w/o errors

Sender Window

Window size = 3 packets

Receiver Window

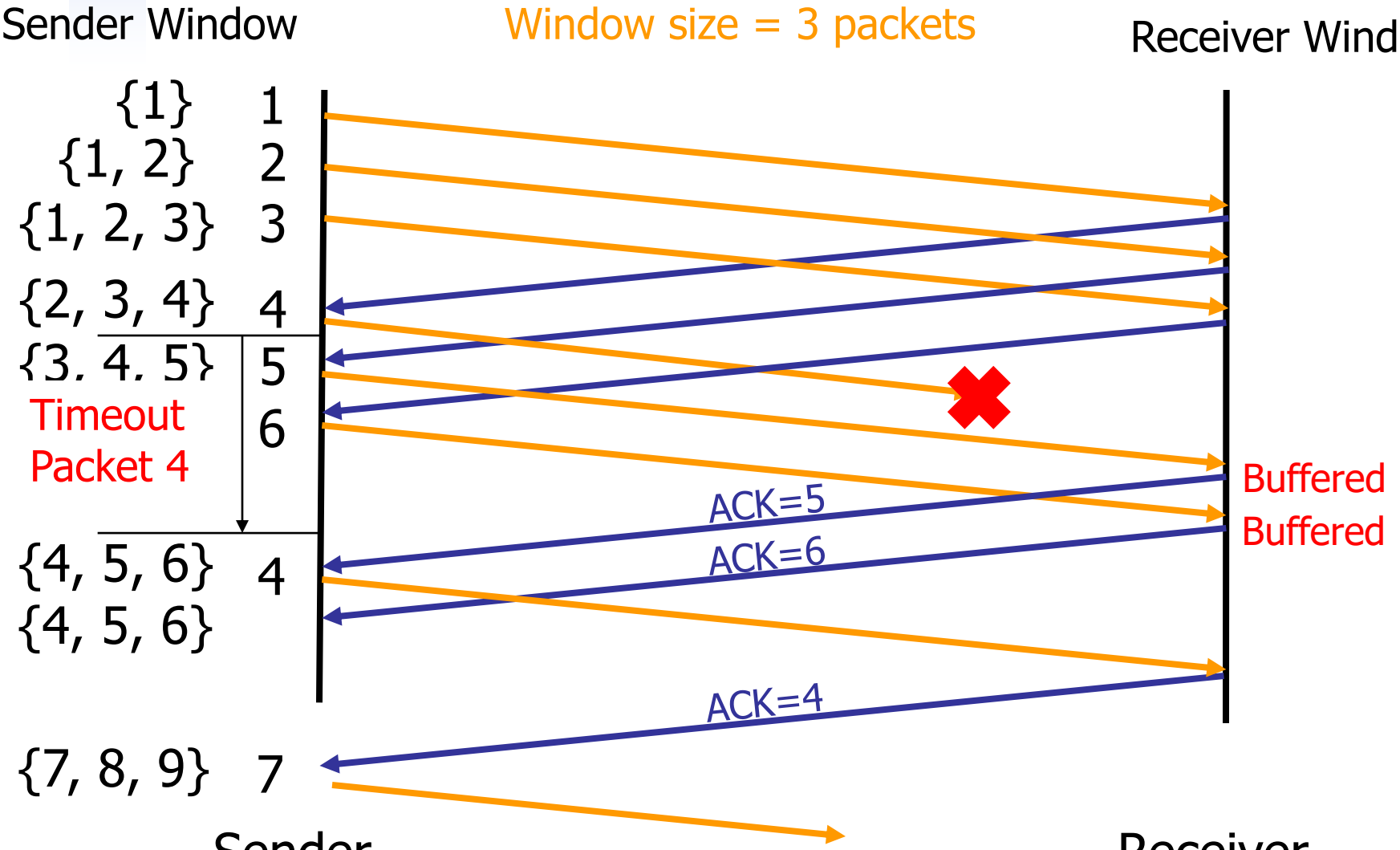



GBN example with errors
Sender Window
Window size = 3 packets
Receiver Window
1
2
3
4
5
6
Timeout
Packet 4
Discard
Discard
4
5
6
Sender
Receiver

Selective Repeat (SR)

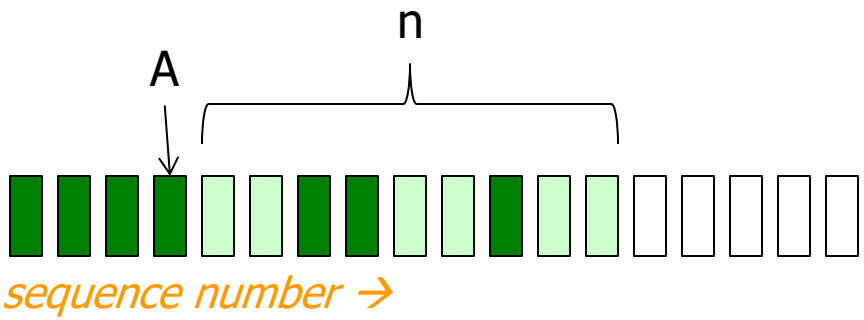
- Sender: transmit up to n unacknowledged packets
- Assume packet k is lost, k+1 is not
  - Receiver: indicates packet k+1 correctly received
  - Sender: retransmit only packet k on timeout
- Efficient in retransmissions, but…
  - Needs complex book-keeping, e.g., timer per packet
  - Needs

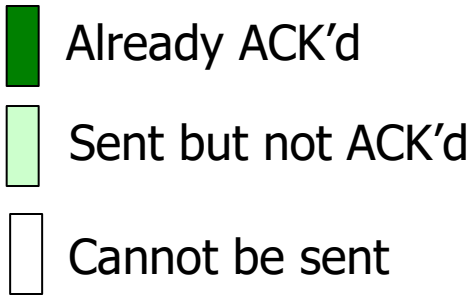
SR example with errors



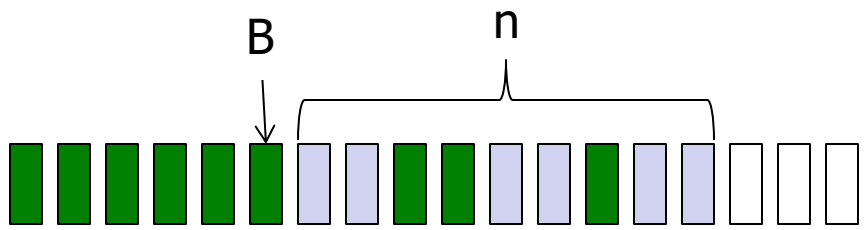
Sliding window w/ SR

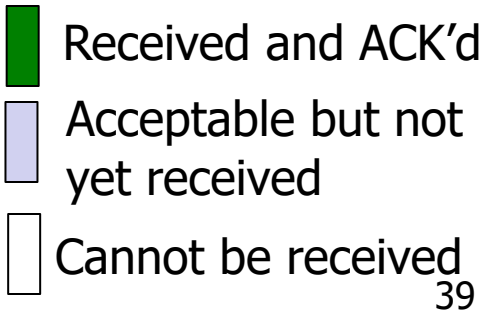
- Let A be the last ACK'd packet without gap; then sender window = {A+1, A+2, …, A+n}





- Let B be the last received packet without gap; then receiver window = {B+1, …, B+n}





GBN vs. Selective Repeat

- When would GBN be better?
  - When error rate is low; wastes bandwidth otherwise
- When would SR be better?
  - When error rate is high; wastes memory and adds complexity otherwise

Many more optimizations

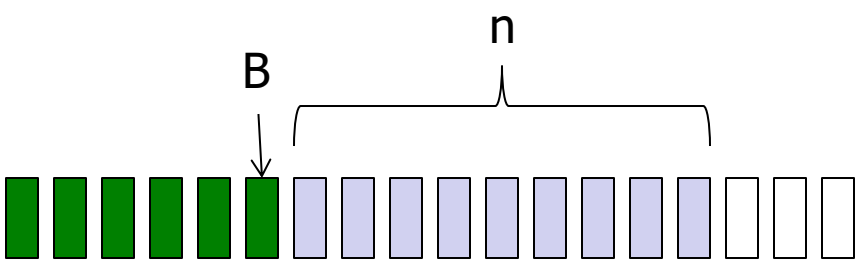
- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
  - Can we avoid using a timeout per packet?
- Sequence numbers (to deal with duplicates)
  - Can we defend against stale packets?
- Timers (to detect loss)
  - How do we set the timer?

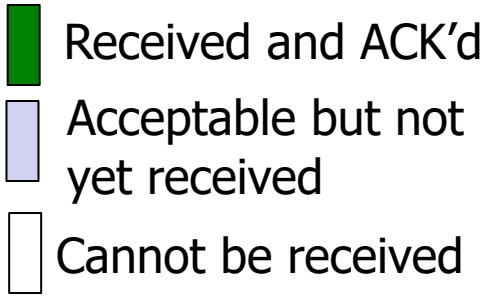
Acknowledgements

- Problem: timeouts are very slow
- Idea: ACKs can carry information beyond just the current packet
  - Cumulative ACKs: ACK carries next in-order sequence number that the receiver expects
  - Selective ACKs: ACK individually acknowledges correctly received packets

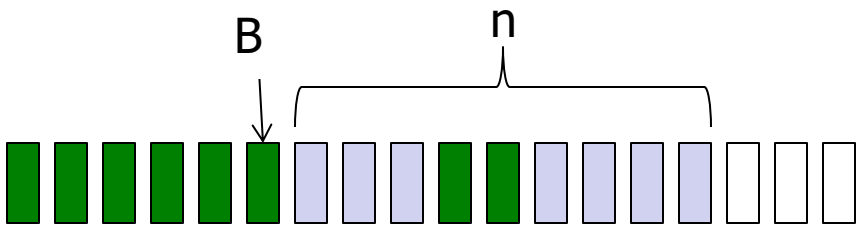
Cumulative acknowledgements (cont'd)

- At receiver





- After receiving B+4, B+5



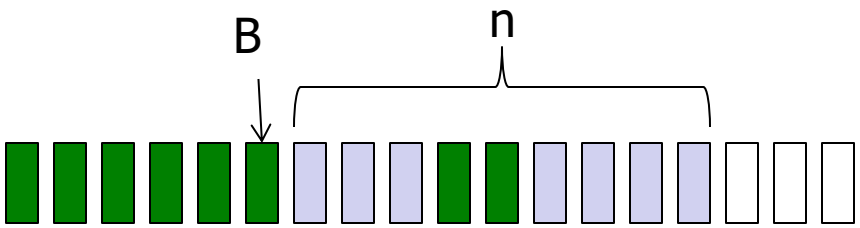
- Receiver sends ACK(B+1)

Taking advantage of cumulative ACKs

- When packet n is lost…
  - … packets n+1, n+2, and so on may get through
- Exploit the ACKs of these packets
  - ACK says receiver is still awaiting nth packet
  - Duplicate ACKs suggest later packets arrived
  - Sender uses "duplicate ACKs" as a hint
- Fast retransmission
  - Retransmit after "triple duplicate ACK"

Selective ACKs

- Send in the ACK the ranges of bytes received
- Example:



ACK(B+1, [B+4, B+6])

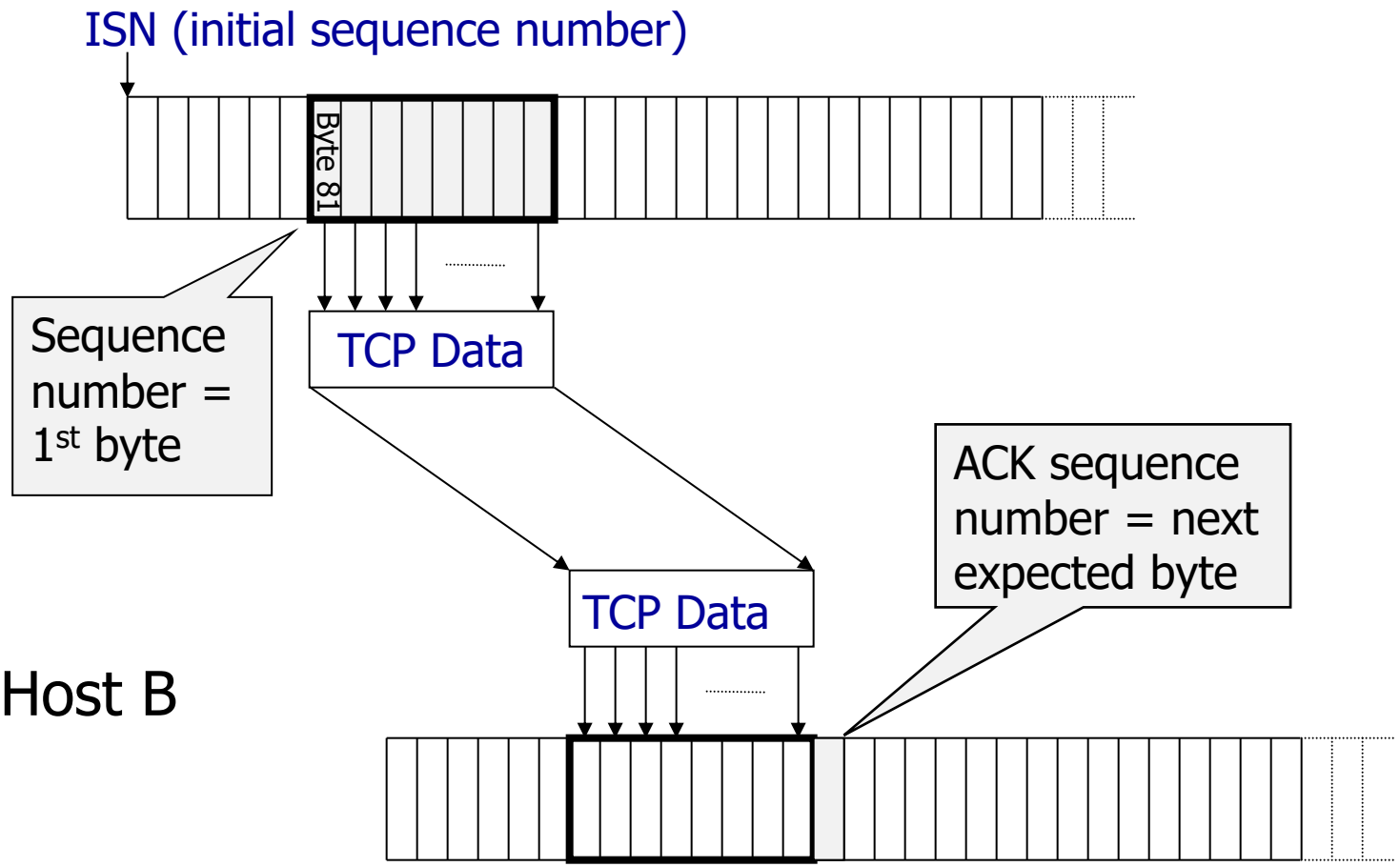
- Selective ACKs offer more precise information but require more complicated book-keeping

Many more optimizations

- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
  - Can we avoid using a timeout per packet?
- Sequence numbers (to deal with duplicates)
  - Can we defend against stale packets?
- Timers (to detect loss)
  - How do we set the timer?

Recap: Sequence Numbers

Host A



Initial Sequence Number (ISN)

- Problem: Can't always use ISN of 0
  - Why?
- Reuse of port numbers
  - Port numbers must (eventually) get used again
  - … and an old packet may still be in flight
  - … and associated with the new connection
- Idea: TCP must change the ISN over time
  - Set from a 32-bit clock that ticks every 4 microsec
  - … which wraps around once every 4.55 hours!

Many more optimizations

- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
  - Can we avoid using a timeout per packet?
- Sequence numbers (to deal with duplicates)
  - Can we defend against stale packets?
- Timers (to detect loss)
  - How do we set the timer?

Retransmission timeout

- Problem: How to set timeout for when to retransmit the first packet in the window?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
  - RTO = EstimatedRTT + 4 x StdDevRTT