CS 43: Computer Networks

UDP and Reliable Transport

October 27, 2025

SWARTHMORE COLLEGE

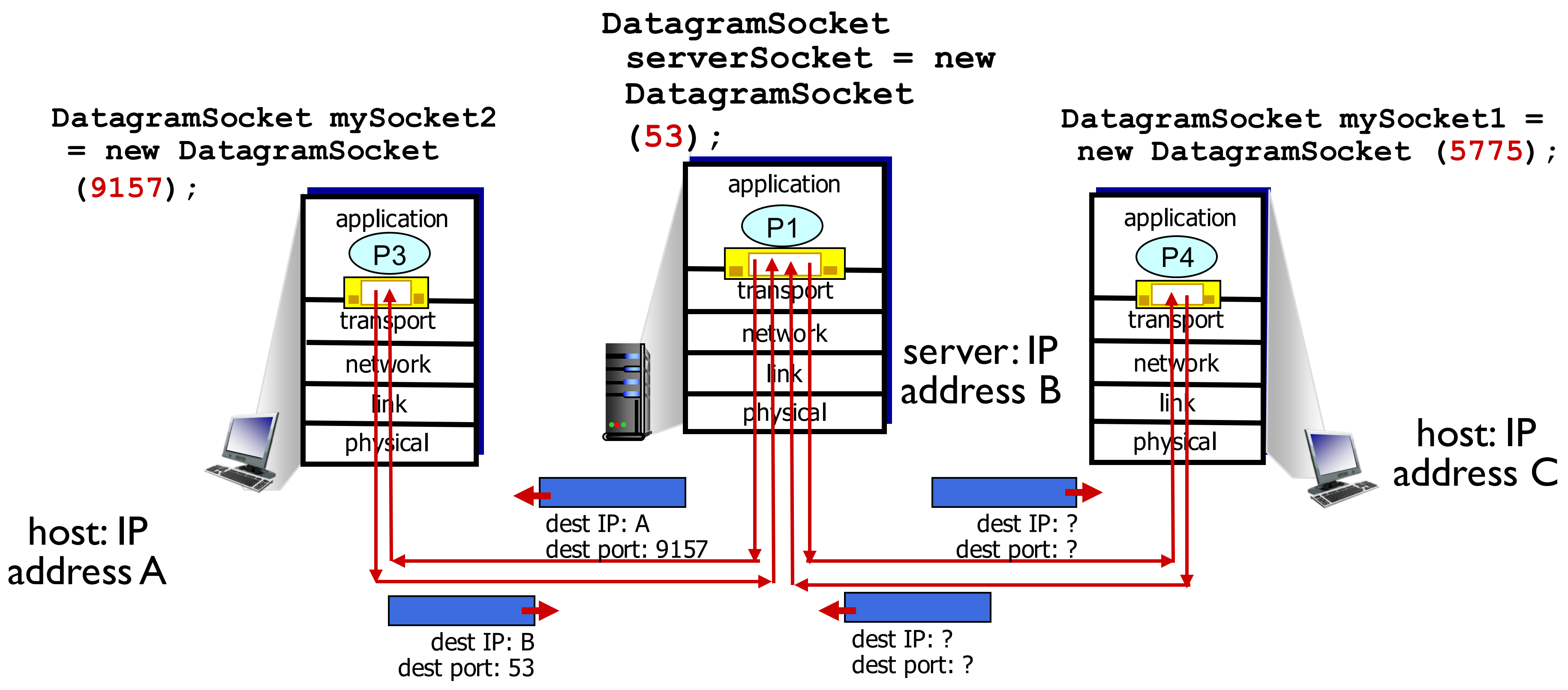
Connectionless: example

- UDP socket identified by 2-tuple:
  - dest IP address
  - dest port number
- when receiving host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

UDP datagrams with *same dest. port #,* but different source (IP/ port #) will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

A UDP socket is uniquely identified by (dest IP, dest port)



UDP – User Datagram Protocol

- Unreliable, unordered service
- Adds:
  - multiplexing,
  - checksum (error detection)

UDP: User Datagram Protocol [RFC 768]

"No frills," "Bare bones" Internet transport protocol

- RFC 768 (1980)
- Length of the document?
  https://www.rfc-editor.org/rfc/rfc768.html

UDP: User Datagram Protocol [RFC 768]

¯_(ツ)_/¯

"Best effort" service,

UDP segments may be:

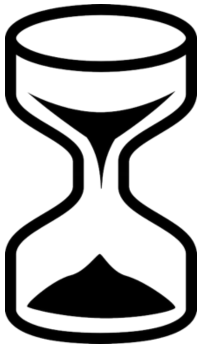
– Lost
– Delivered out of order (same as underlying network layer)

Connectionless:

- No initial state transferred between parties (no handshake)
- Each UDP segment is handled independently

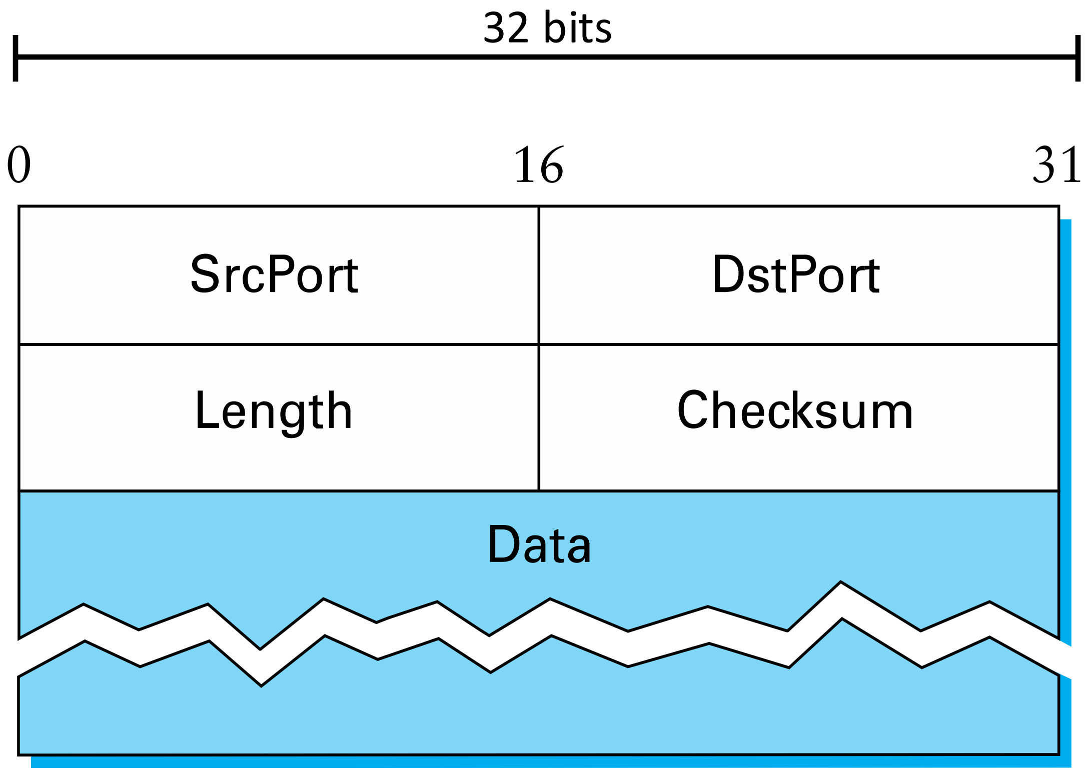
How many of the following steps does UDP implement? (which ones?)

A. exchange an initiate handshake (connection setup)
B. break up packet into segments at the source and number them
C. place segments in order at the destination
D. error-checking with checksum

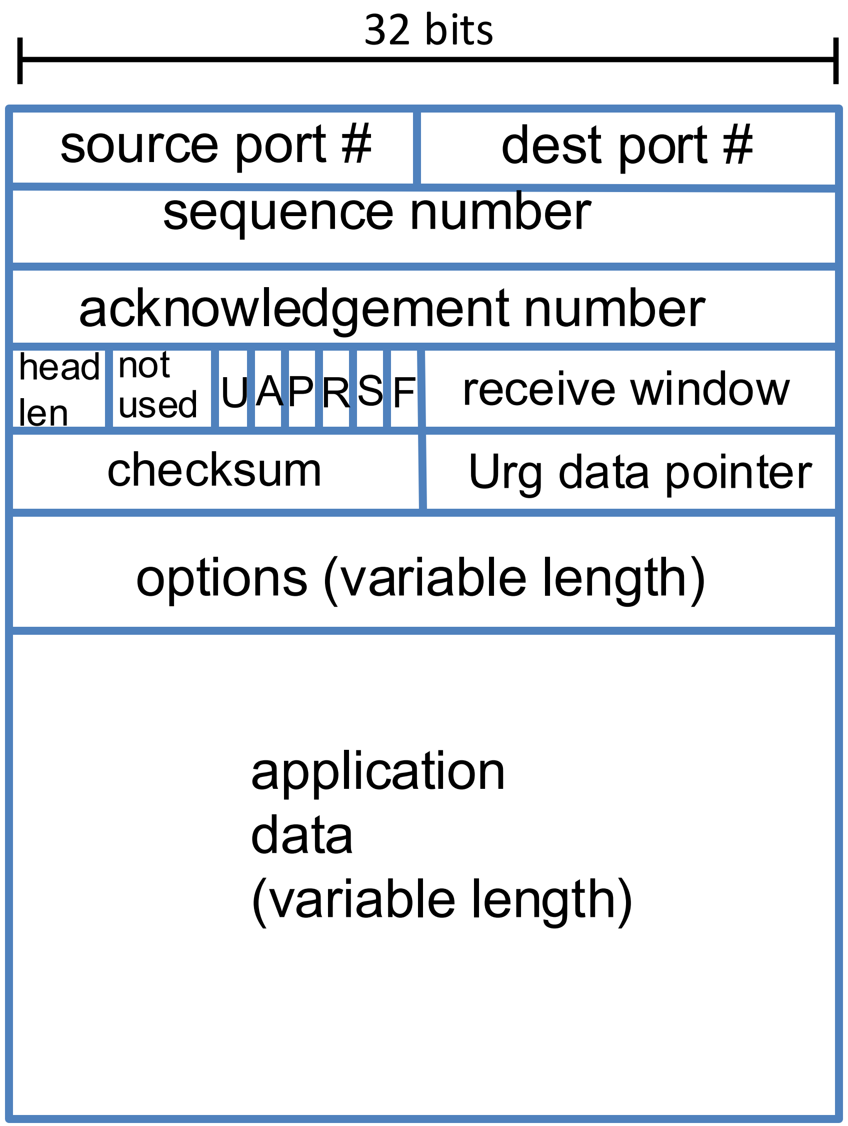


Wireshark Example

UDP Segment

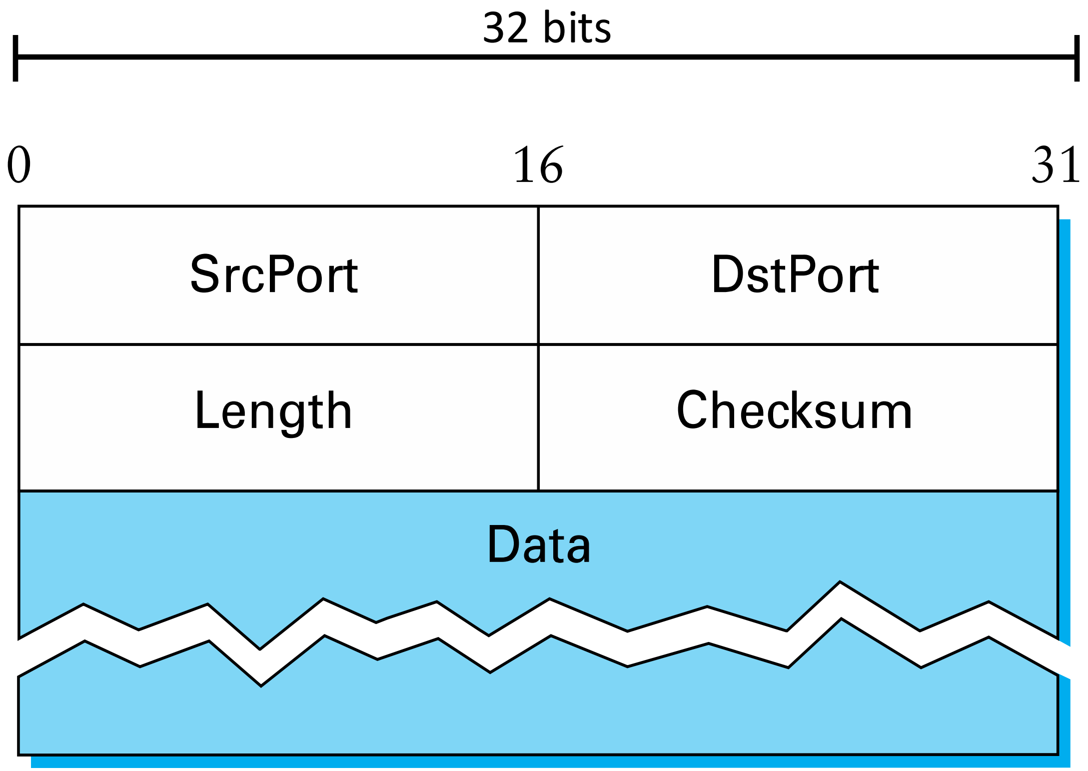


TCP Segment!



32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)

UDP Segment



UDP Checksum

- Goal: Detect transmission errors (e.g. flipped bits)
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference

UDP Checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	10

receiver-computed checksum ≠ sender-computed checksum (as received)

UDP Checksum

RFC: “Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.”

(? _ ?)

UDP Checksum at the Sender

- Treat the entire segment as 16-bit integer values
- Add them all together (sum)
- Put the 1's complement in the checksum header field

One’s Compliment

- In bitwise compliment, all of the bits in a binary number are flipped.
- So 1111000011110000 -> 0000111100001111

Checksum example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Receiver

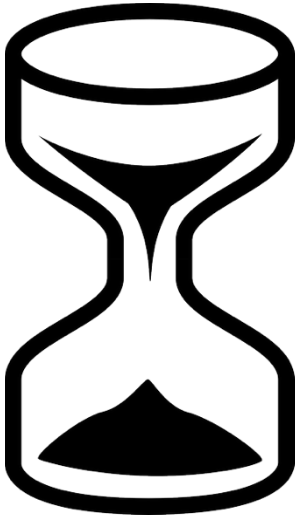
- Add all the received data together as 16-bit integers
- Add that to the checksum
- If result is not 1111 1111 1111 1111, there are errors!
- If there are errors chuck the packet.

¯_(ツ)_/¯

If our checksum addition yields all ones, are we guaranteed to be error-free?

A. Yes

B. No



Checksum example : weak protection!

example: add two 16-bit integers

1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 → 0 1

1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 → 1 0

wraparound 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0

checksum 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Even though numbers have changed (bit flips), *no* change in checksum!

UDP Applications

- Latency sensitive
  – Quick request/response (DNS)
  – Network management (SNMP, DHCP)
  – Voice/video chat
- Communicating with *lots* of others

Recall: TCP send() blocking

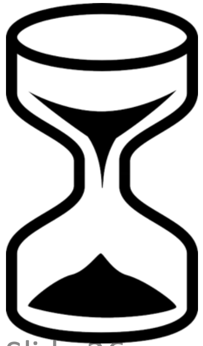
With TCP, send() blocks if buffer full.

UDP sendto() blocking

With TCP, send() blocks if buffer full.

- Does UDP need to block? Should it?

A. Yes, if buffers are full, it should.
B. It doesn't need to, but it might be useful.
C. No, it does not need to and shouldn't do so.



Summary

Transport Layer:

- Provides a logical communication between processes/ applications
- packets are called segments at the transport layer
- Transport layer protocol: responsible for adding port numbers (mux/demux segments)

Summary

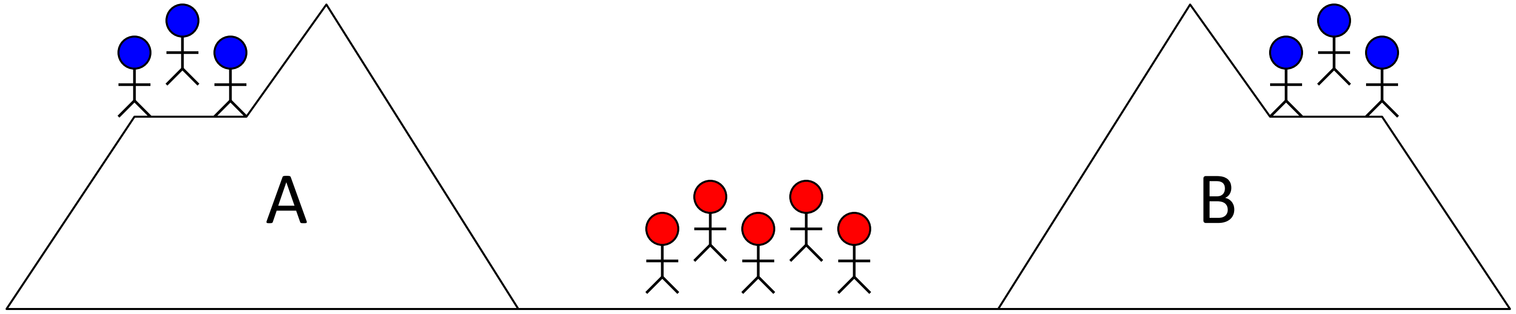
UDP:

- No “frills” protocol, No state maintained about the packet
- Checksum (1’s complement) over IP + UDP + payload.
  - can only correct for 1 bit errors.
- adds port numbers over unreliable network (best effort)
- applications:
  - latency sensitive applications: real-time audio, video
  - communicating with a lot of end-hosts (like DNS)
- UDP Sockets:
  - do not need to be implemented as blocking system calls for correctness since the only guarantee UDP makes is best-effort delivery.
  - however send/recv can be implemented as blocking system calls depending on the application

Today

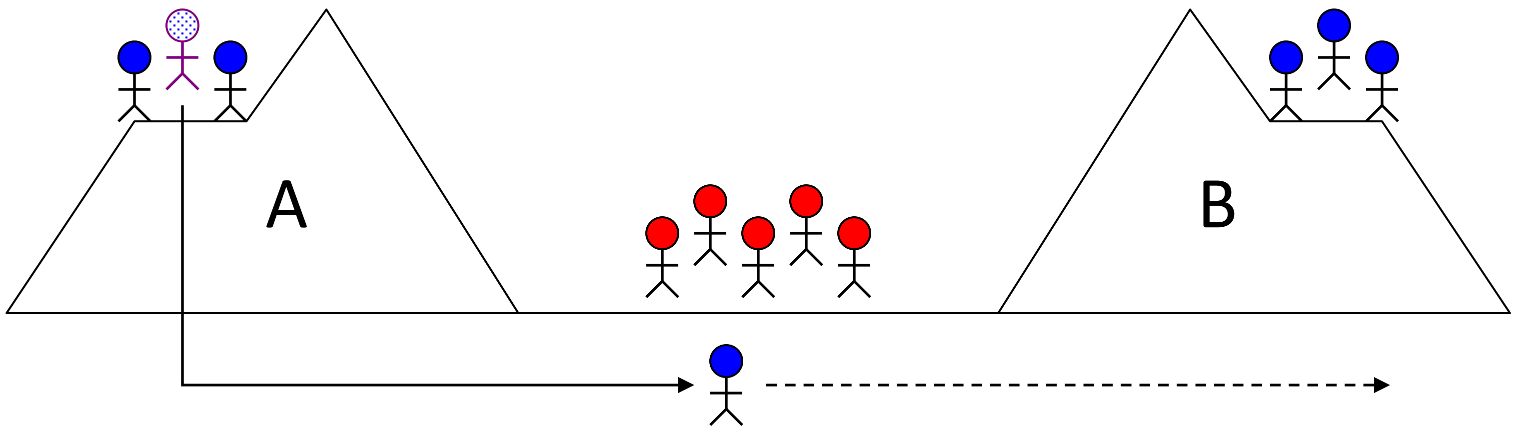
- Principles of reliability
  - The Two Generals Problem
- Automatic Repeat Requests
  - Stop and Wait
  - Timeouts and Losses
  - Pipelined Transmission

The Two Generals Problem



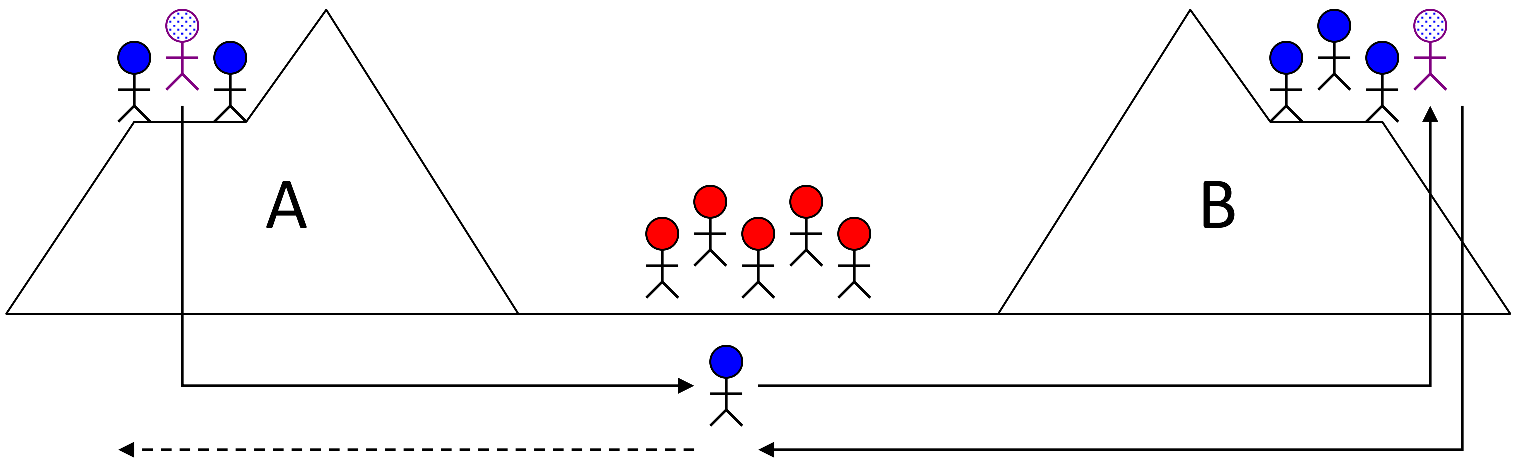
- Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

The Two Generals Problem



- How to coordinate?
  - Send messenger: “Attack at dawn”
  - What if messenger doesn’t make it?

The Two Generals Problem

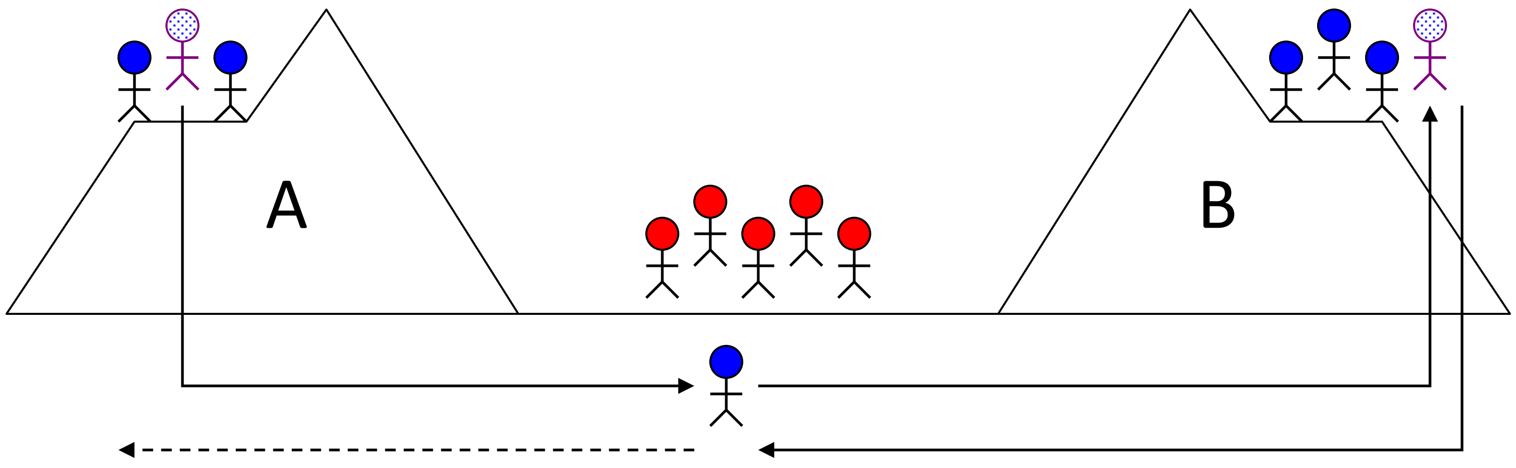


- How to be sure messenger made it?
  - Send acknowledgment: “I delivered message”

In the “two generals problem”, can the two armies reliably coordinate their attack? (using what we just discussed)

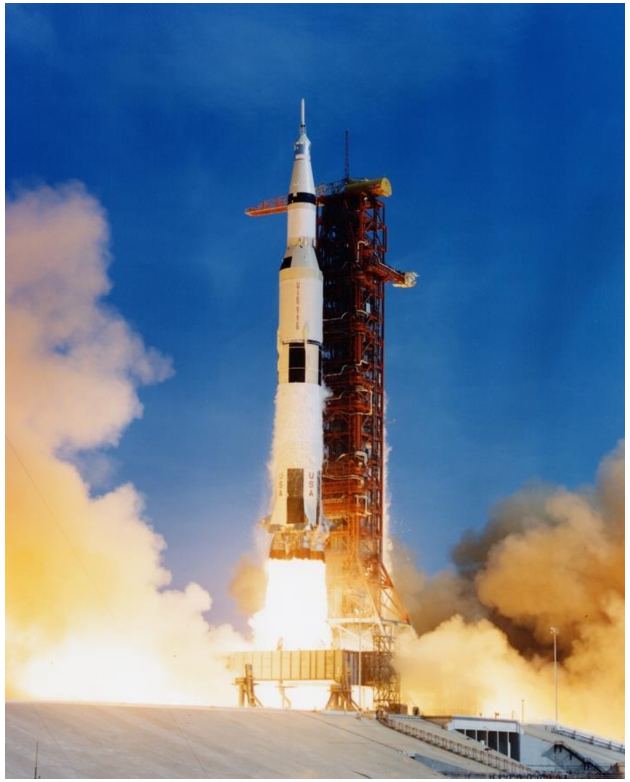
- A. Yes (explain how)
- B. No (explain why not)

The Two Generals Problem



- Result
  - Can’t create perfect channel out of faulty one
  - <u>Can only increase probability of success</u>

Give up? No way!



As humans, we like to face difficult problems.

- We can’t control oceans, but we can build canals
- We can’t fly, but we’ve landed on the moon
- We just need engineering!

What can possibly go wrong....

Engineering

- Concerns
  - Message corruption
  - Message duplication
  - Message loss
  - Message reordering
  - Performance
- Our toolbox
  - Checksums
  - Timeouts
  - Acks & Nacks
  - Sequence numbering
  - Pipelining

Engineering

- Concerns
  - Message corruption
  - Message duplication
  - Message loss
  - Message reordering
  - Performance
- Our toolbox
  - Checksums
  - Timeouts
  - Acks & Nacks
  - Sequence numbering
  - Pipelining

We use these to build Automatic Repeat Request (ARQ) protocols.

(We'll briefly talk about alternatives at the end.)

Automatic Repeat Request (ARQ)

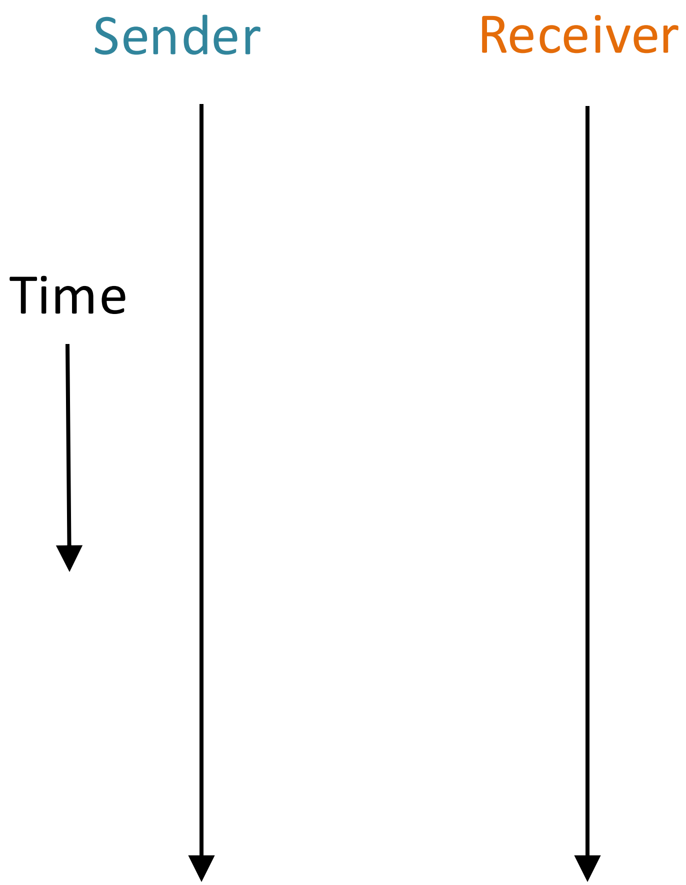
- Intuitively, ARQ protocols act like you would when using a cell phone with bad reception.
  - Receiver: Message garbled? Ask to repeat.
  - Sender: Didn’t hear a response? Speak again.
- Refer to book for building state machines.
  - We’ll look at TCP’s states soon

ARQ Broad Classifications

1. Stop-and-wait

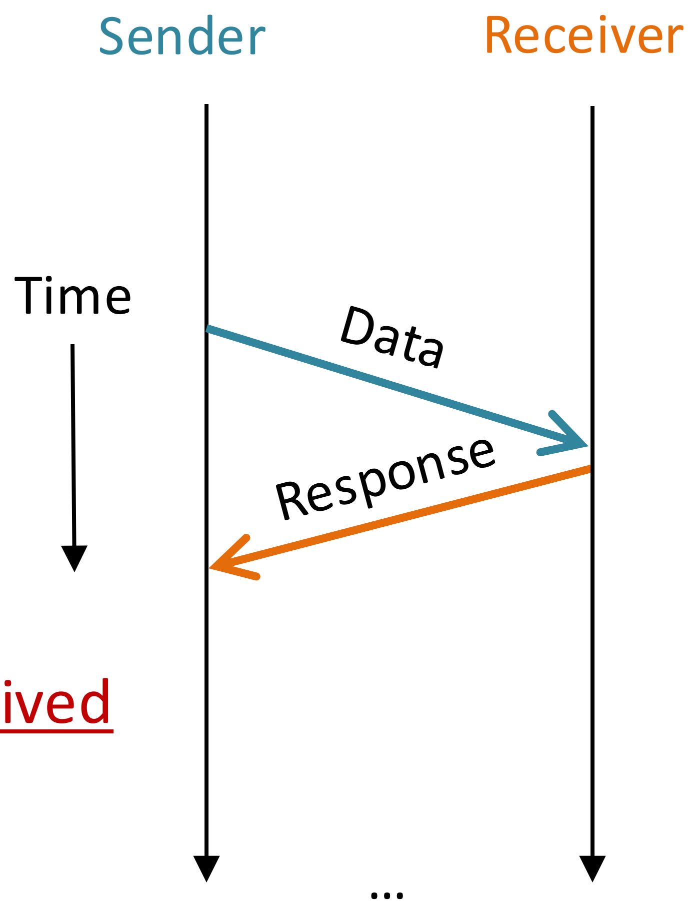
Stop and Wait

We have:
- a sender
- a receiver
- time: represented by downwards arrow



Stop and Wait

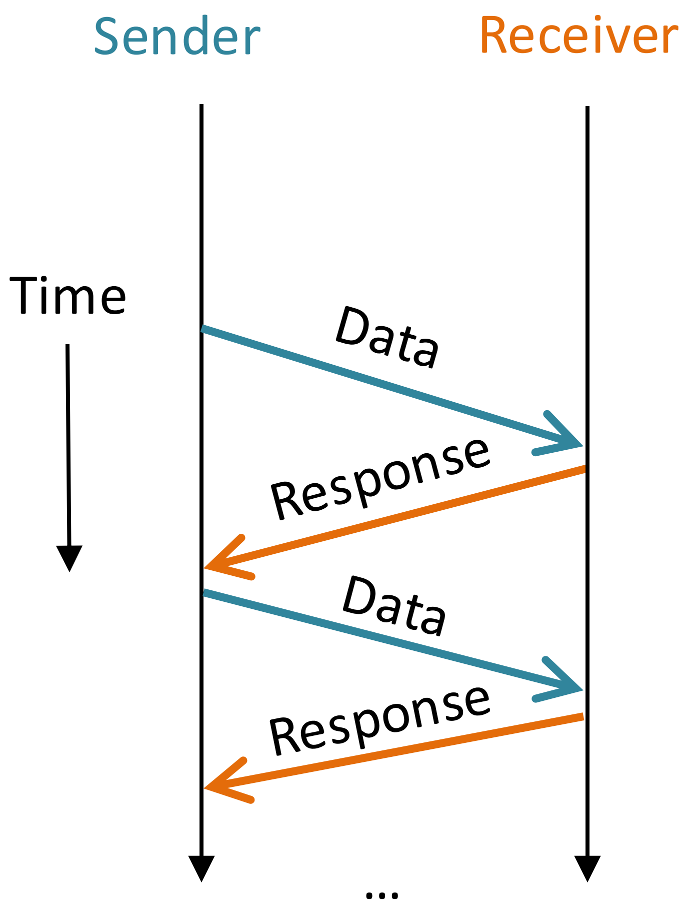
Sender sends data and waits till they get the response message from the receiver.



Buffer data, and don't send till response received

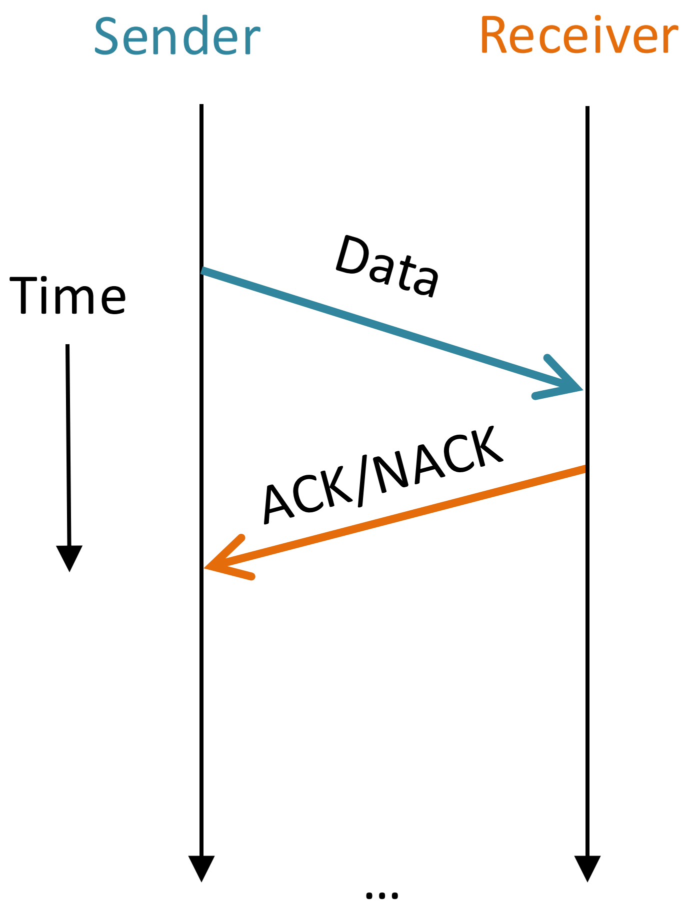
Stop and Wait

- Up next: concrete problems and mechanisms to solve them.
- These mechanisms will build upon each other
- Questions?



Corruption?

- Error detection mechanism: checksum
  - Data good – receiver sends back ACK
  - Data corrupt – receiver sends back NACK

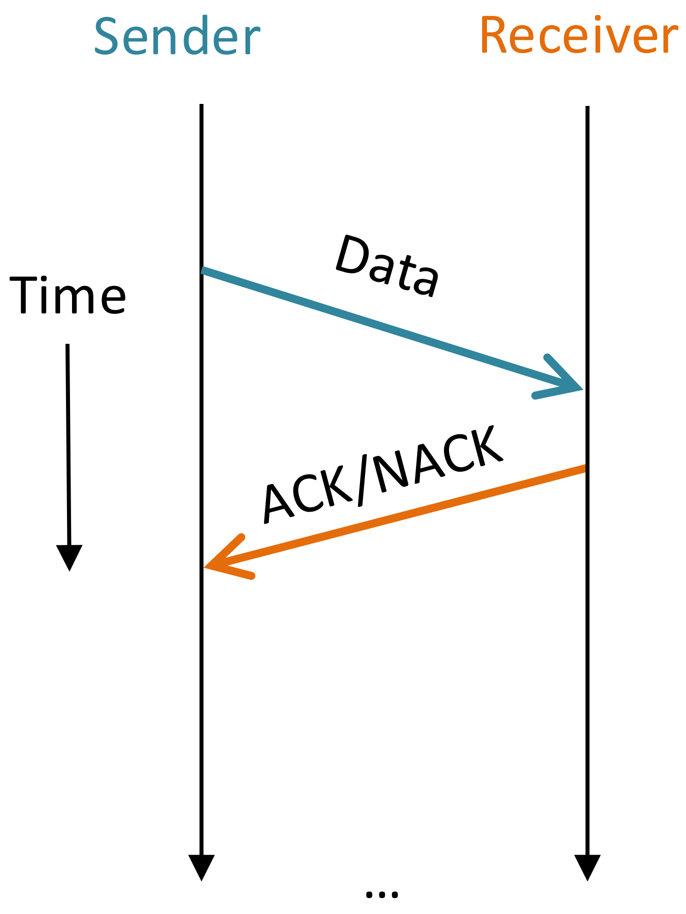


Could we do this with just ACKs or just NACKs?

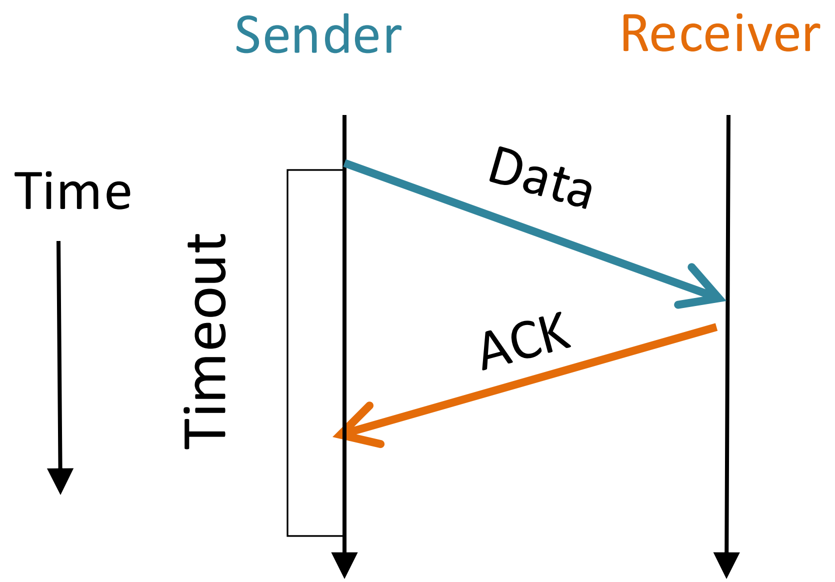
Error detection mechanism: checksum

- Data good – receiver sends back ACK
- Data corrupt – receiver sends back NACK

A. No, we need them both.
B. Yes, we could do without one of them, but we'd need some other mechanism.
C. Yes, we could get by without one of them.

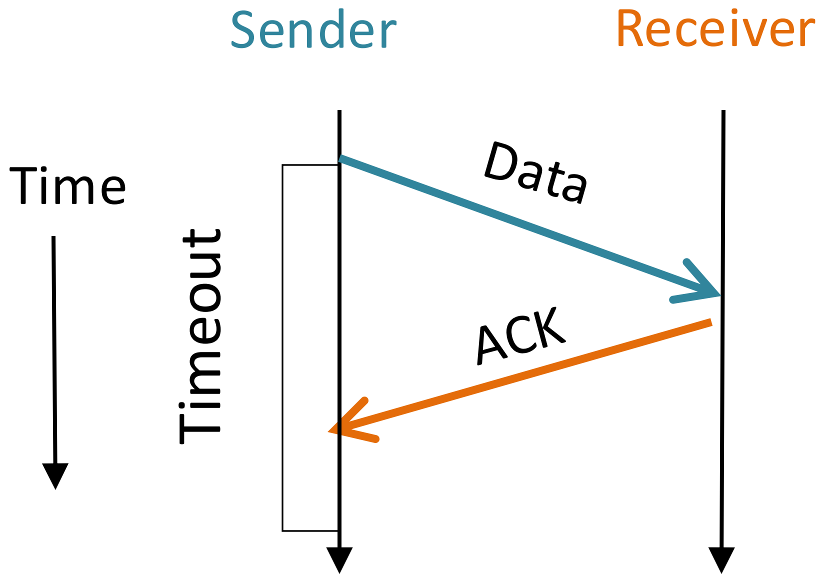


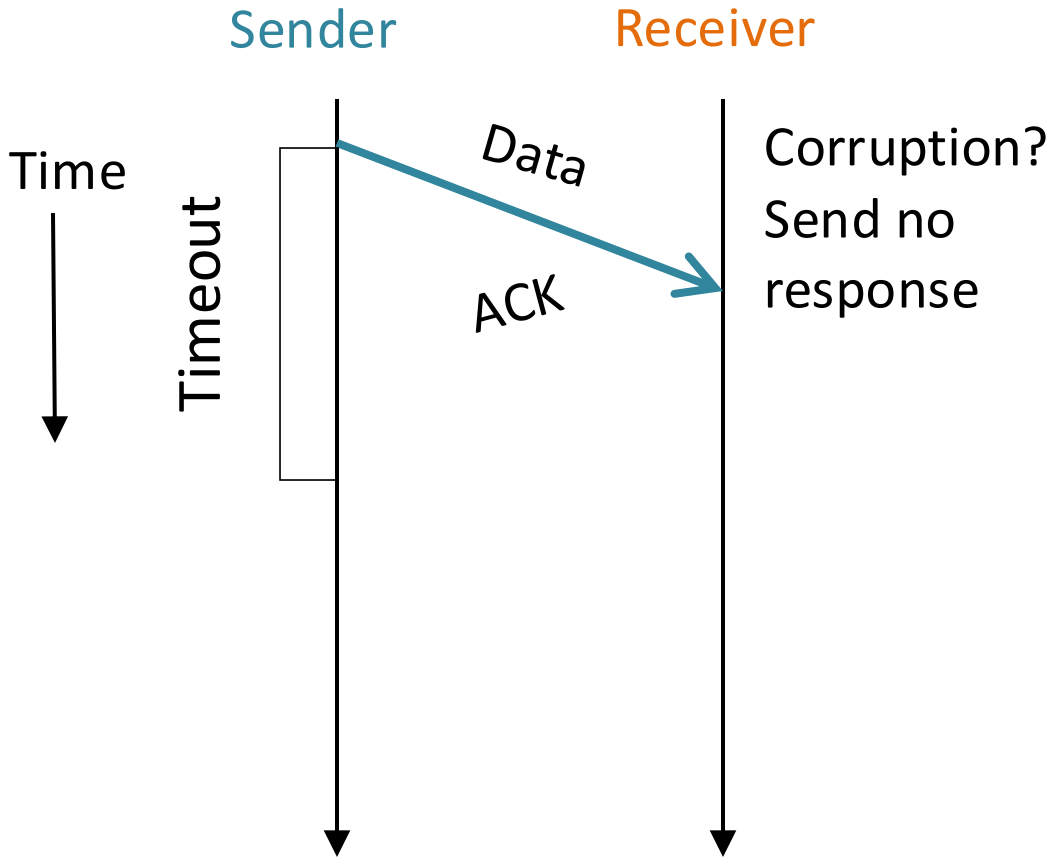
Timeouts and Losses



- Sender starts a clock. If no response, retry.

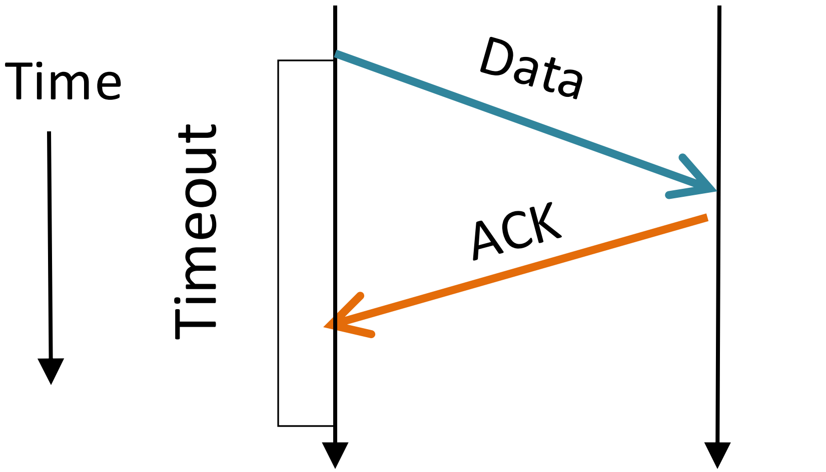
Timeouts and Losses

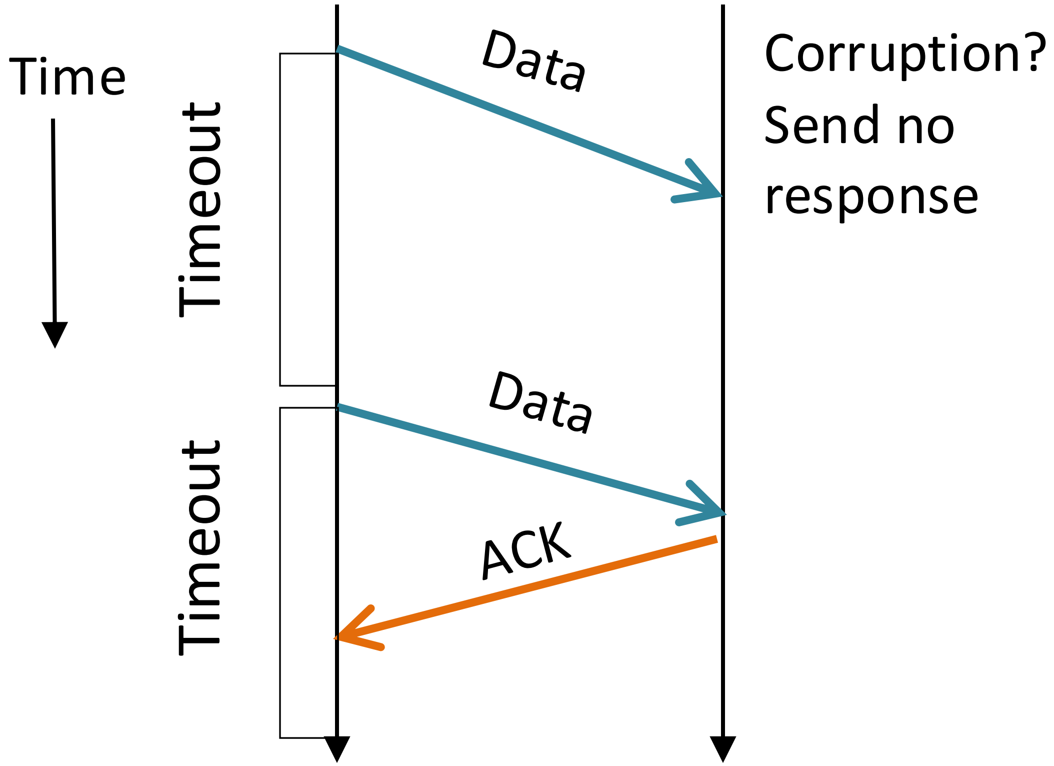




- Sender starts a clock. If no response, retry.

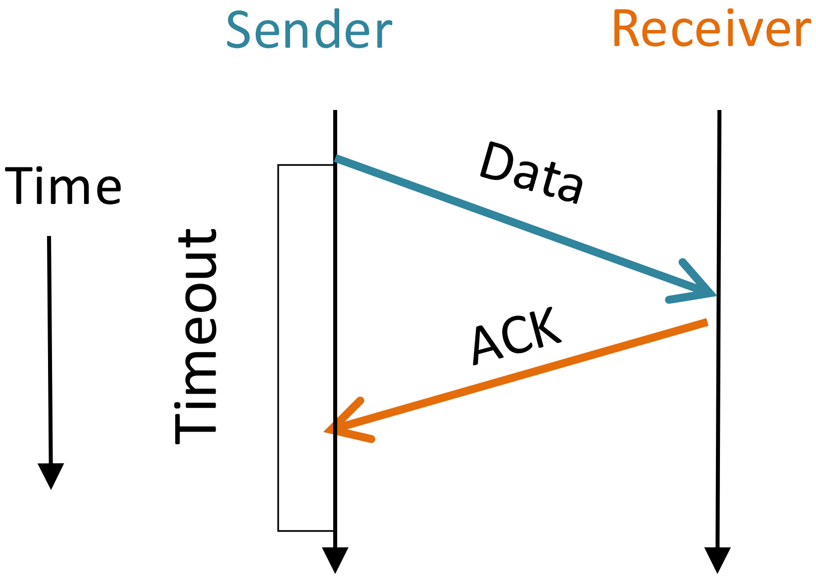
Timeouts and Losses





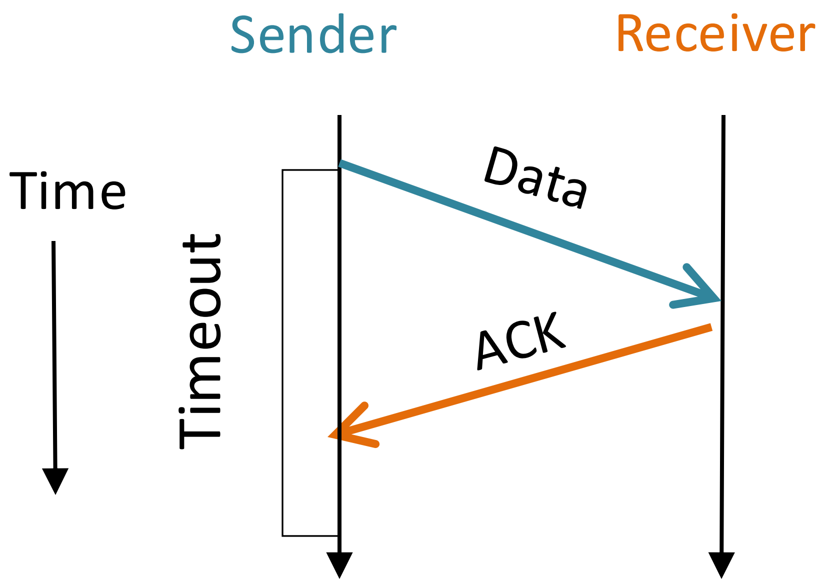
- Sender starts a clock. If no response, retry.
- Probably not a great idea for handling corruption, but it works.

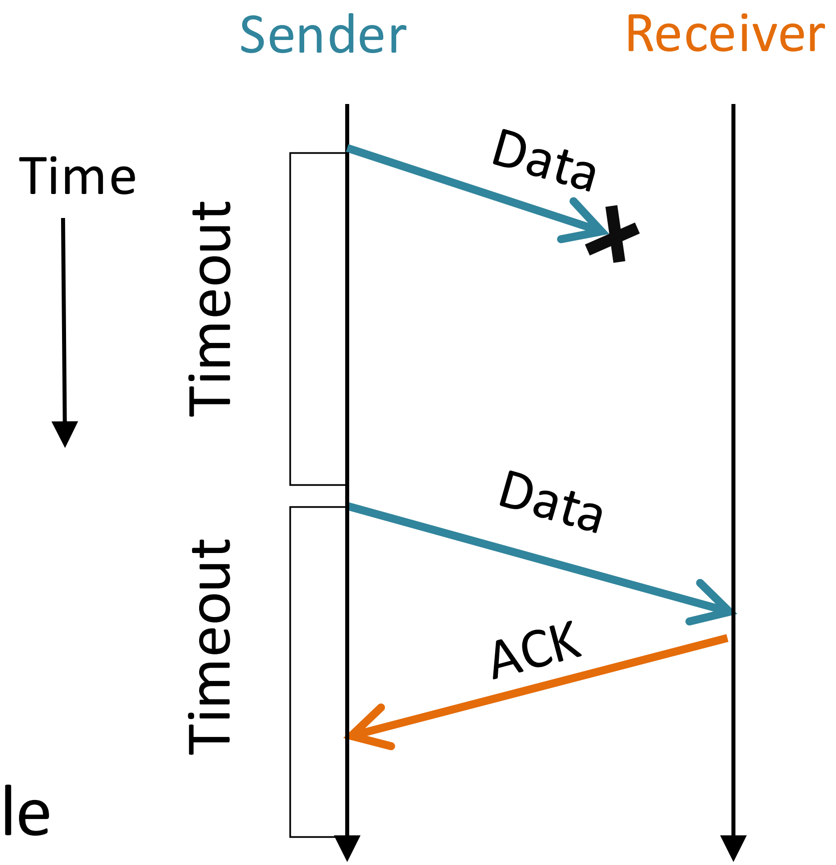
Timeouts and Losses



- Timeouts help us handle message losses too!

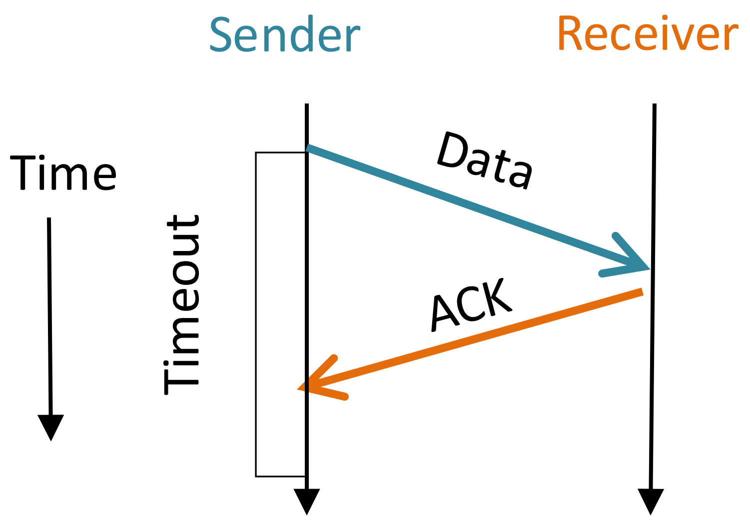
Timeouts and Losses





- Timeouts help us handle message losses too!

Adding timeouts might create new problems for us to worry about. How many? Examples?



A. No new problems (why not?)
B. One new problem (which is..)
C. Two new problems (which are..)
D. More than two new problems (which are..)

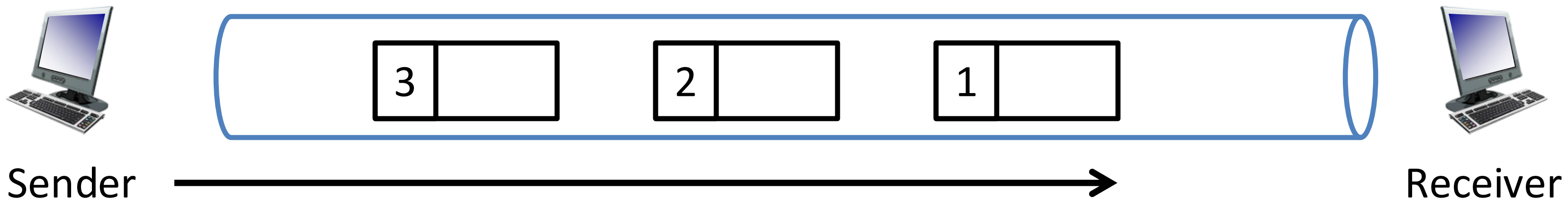
Sequence Numbering

Sender

- Add a monotonically increasing label to each msg

Receiver

- Ignore messages with numbers we've seen before
- When pipelining (a few slides from now)
  - Detect gaps in the sequence (e.g., 1,2,4,5)



What is our link utilization with a stop-and-wait protocol?

A. < 0.1 %
B. ≈ 0.1 %
C. ≈ 1 %
D. 1-10 %
E. > 10 %

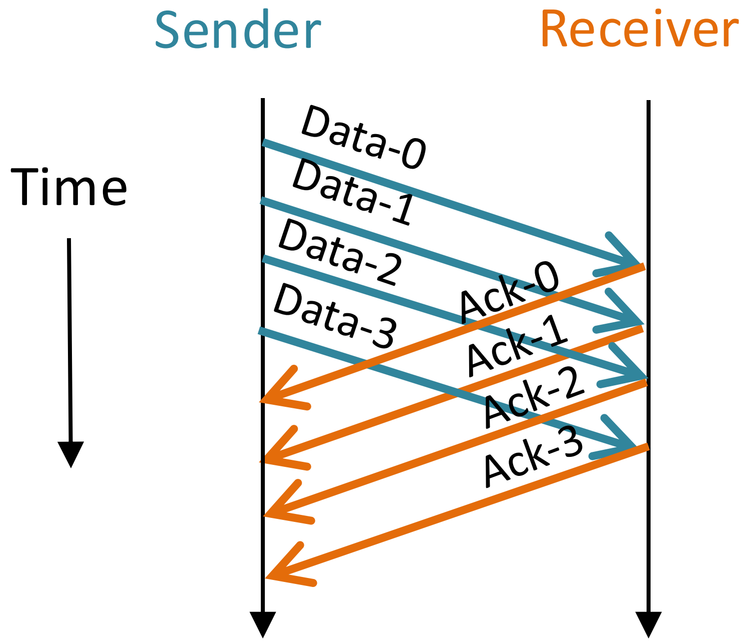
System parameters:

Link rate: 8 Mbps (one megabyte per second)

RTT: 100 milliseconds

Segment size: 1000 bytes

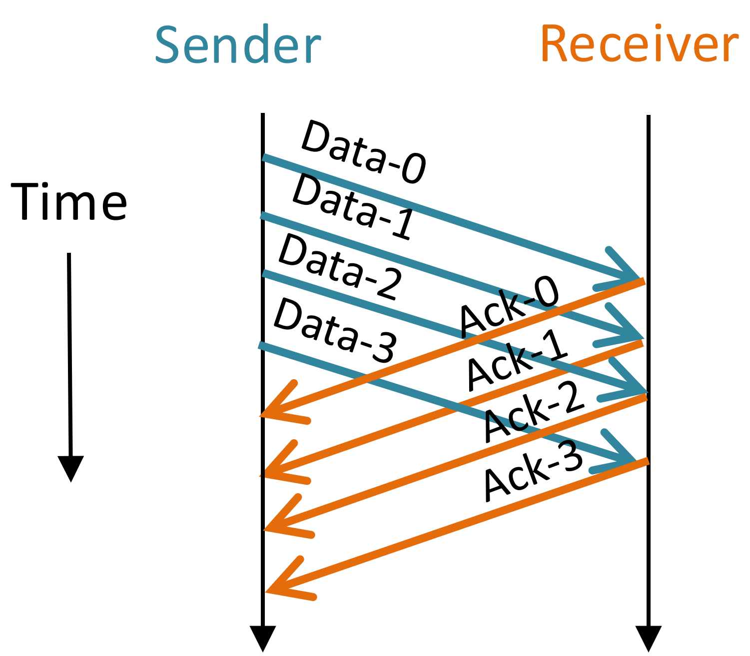
Pipelined Transmission

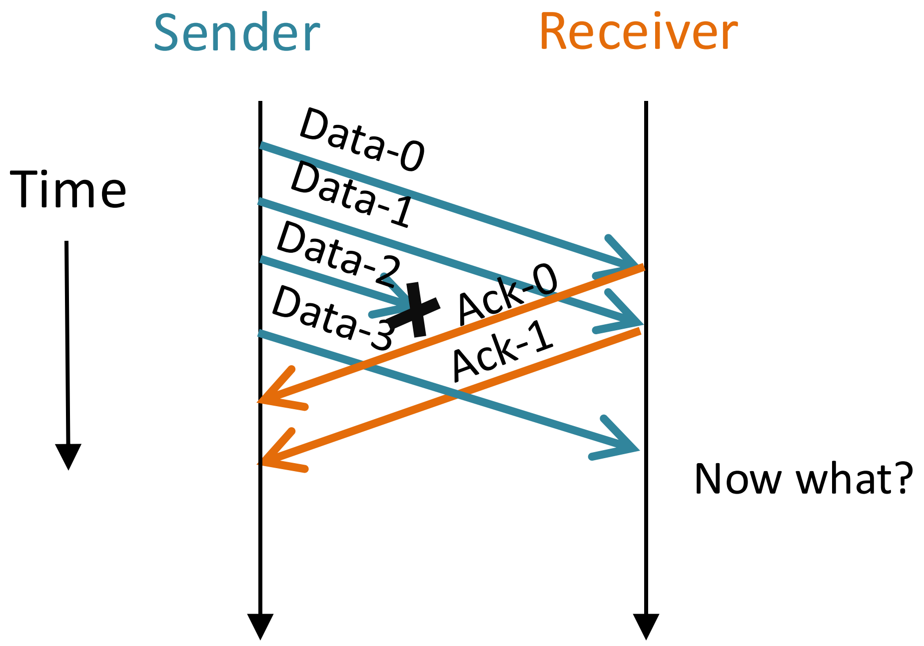


<u>Keep multiple segments "in flight"</u>

- Allows sender to make efficient use of the link
- Sequence numbers ensure receiver can distinguish segments

Pipelined Transmission

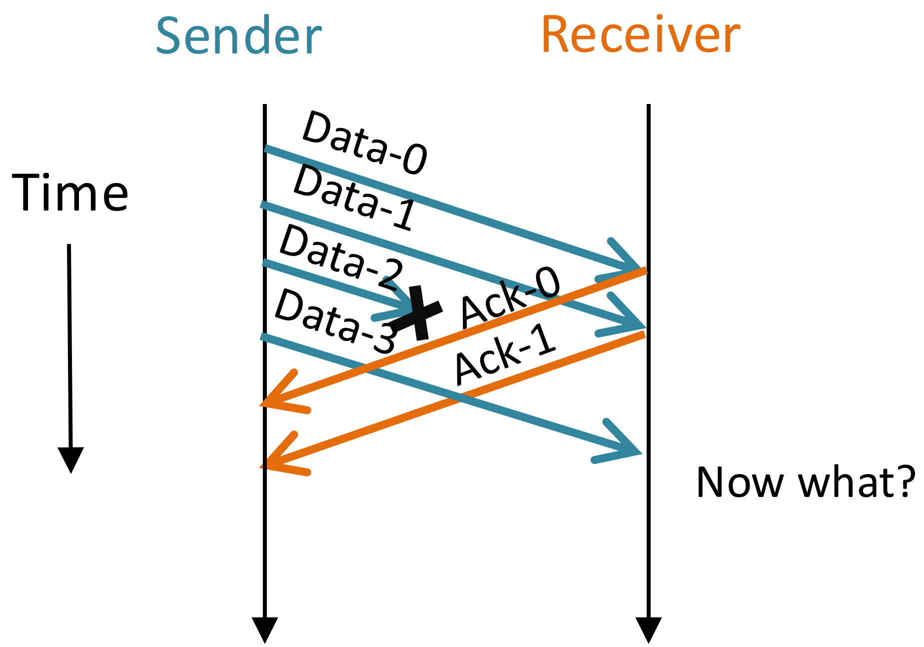




<u>Keep multiple segments "in flight"</u>

- Allows sender to make efficient use of the link
- Sequence numbers ensure receiver can distinguish segments

What should the sender do here?



What information does the sender need to make that decision?

What is required by either party to keep track?

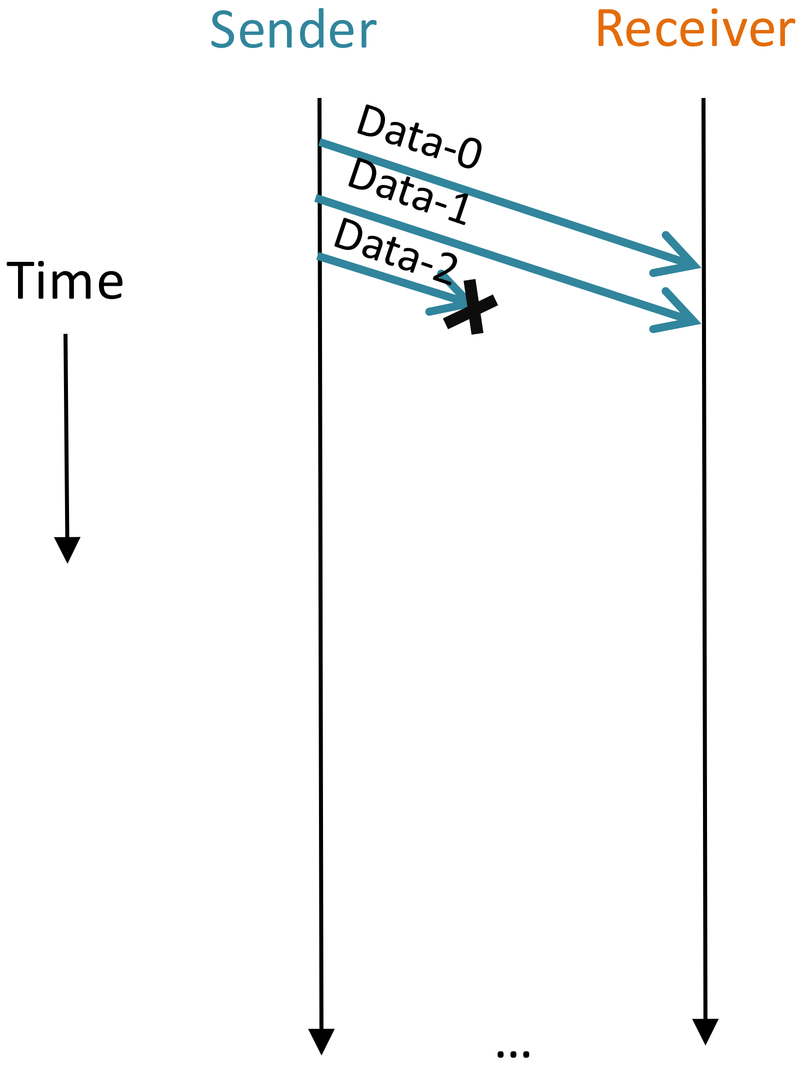
A. Start sending all data again from 0.
B. Start sending all data again from 2.
C. Resend just 2, then continue with 4 afterwards.

ARQ Broad Classifications

1. Stop-and-wait

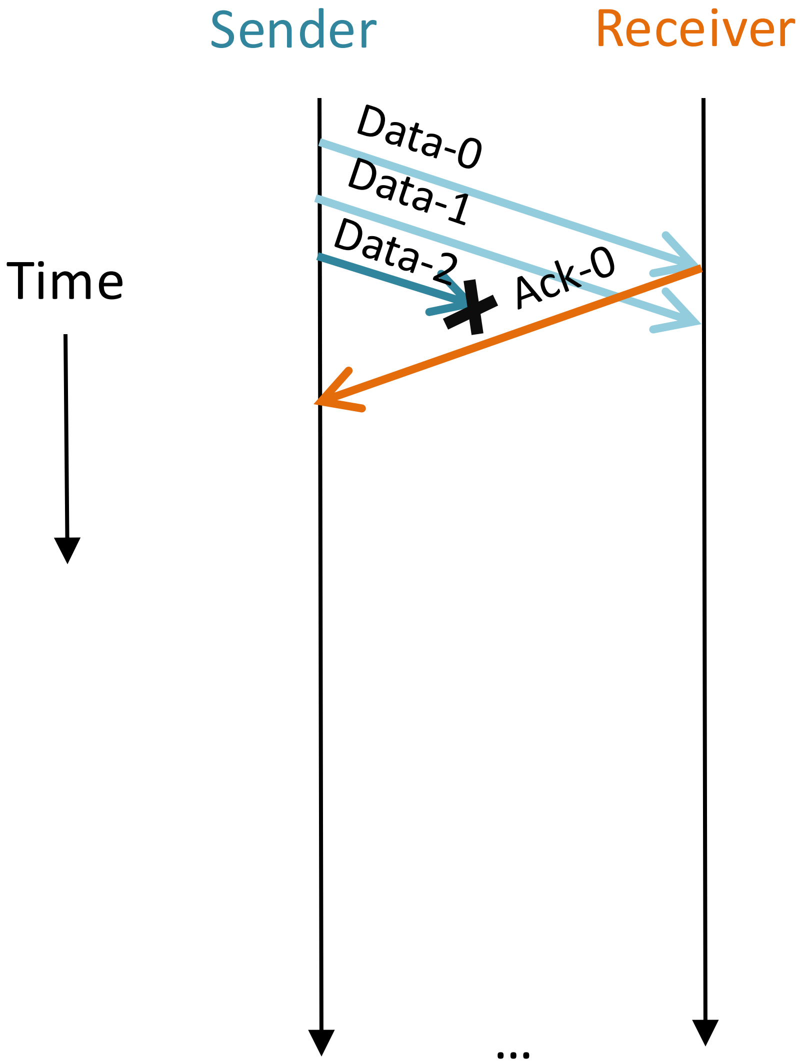
2. Go-back-N

Go-Back-N

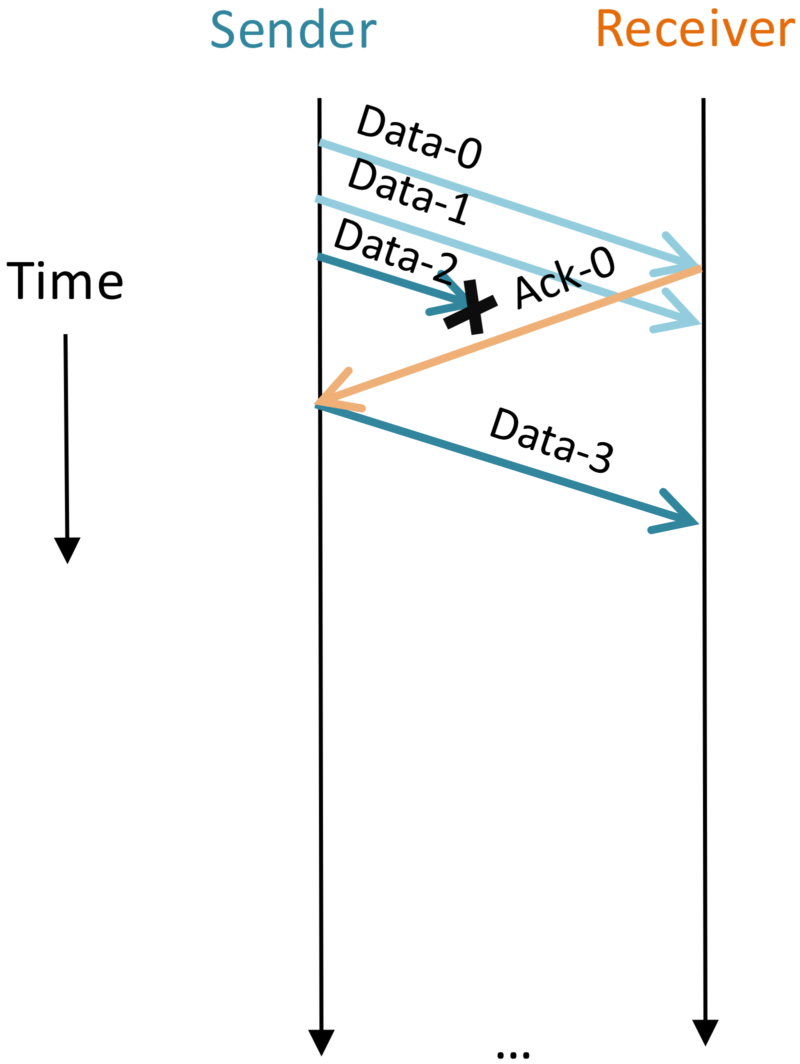


- Retransmit from point of loss
  - Segments between loss event and retransmission are ignored
  - “Go-back-N” if a timeout event occurs

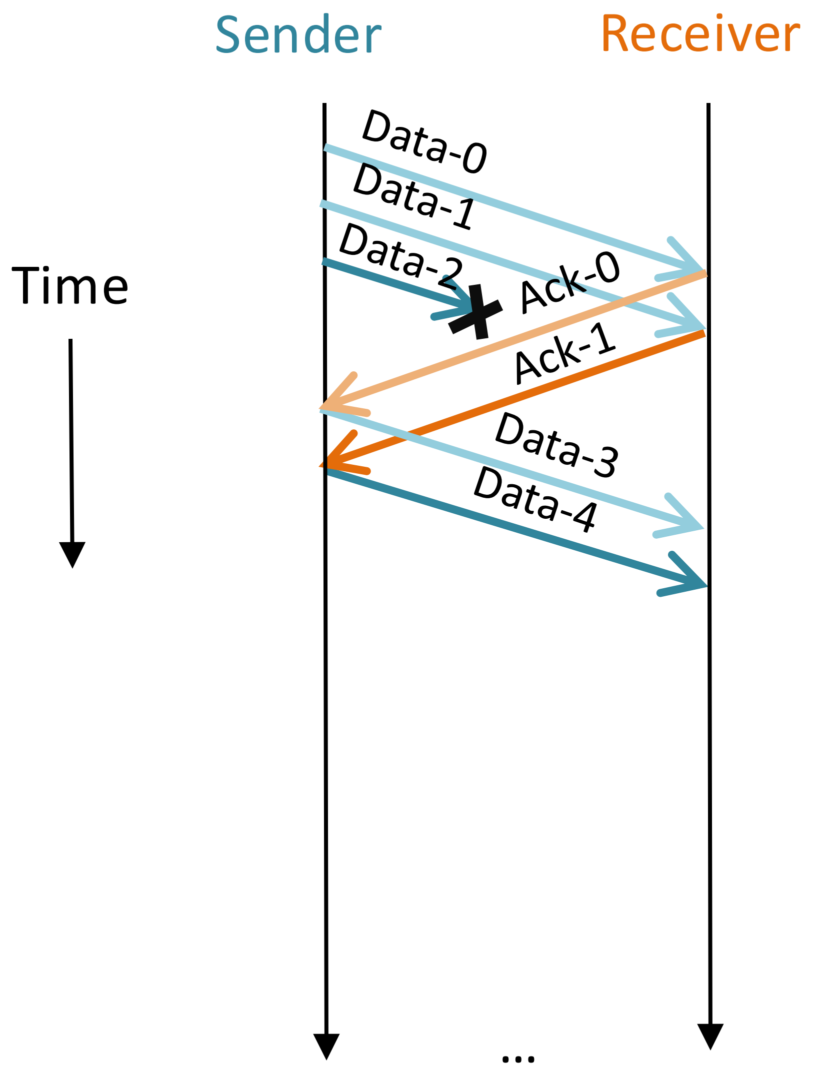
Go-Back-N



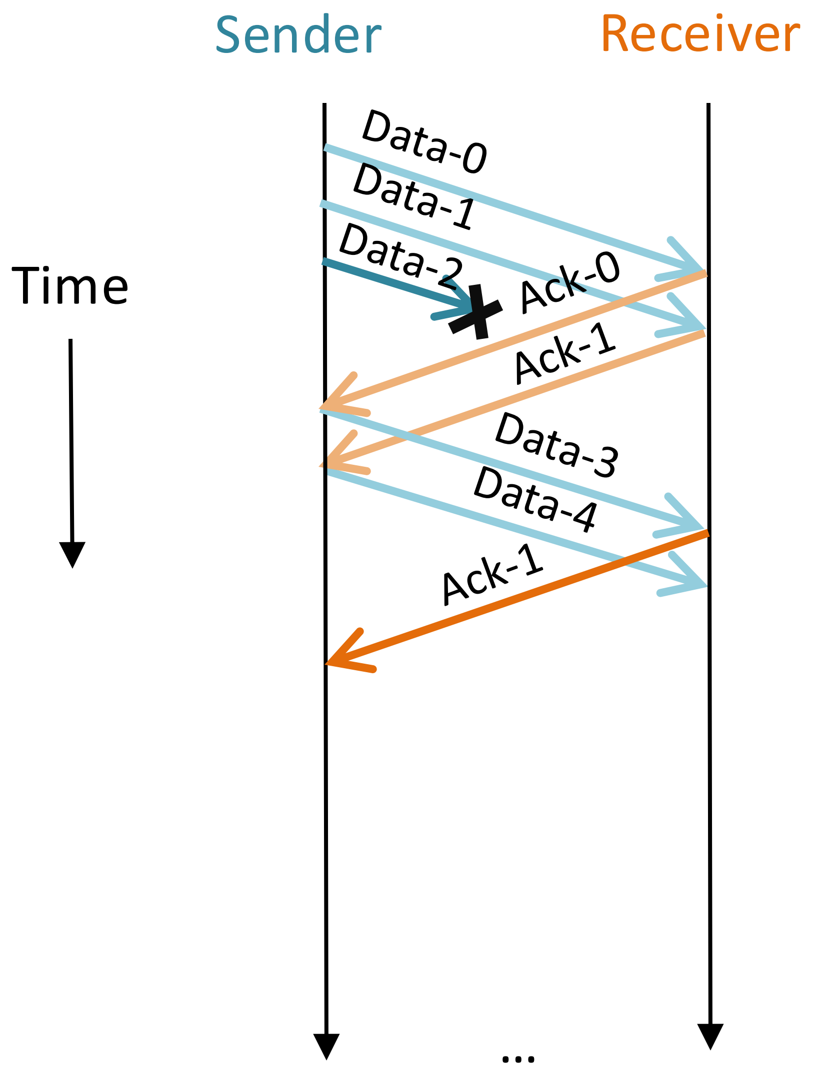
Go-Back-N



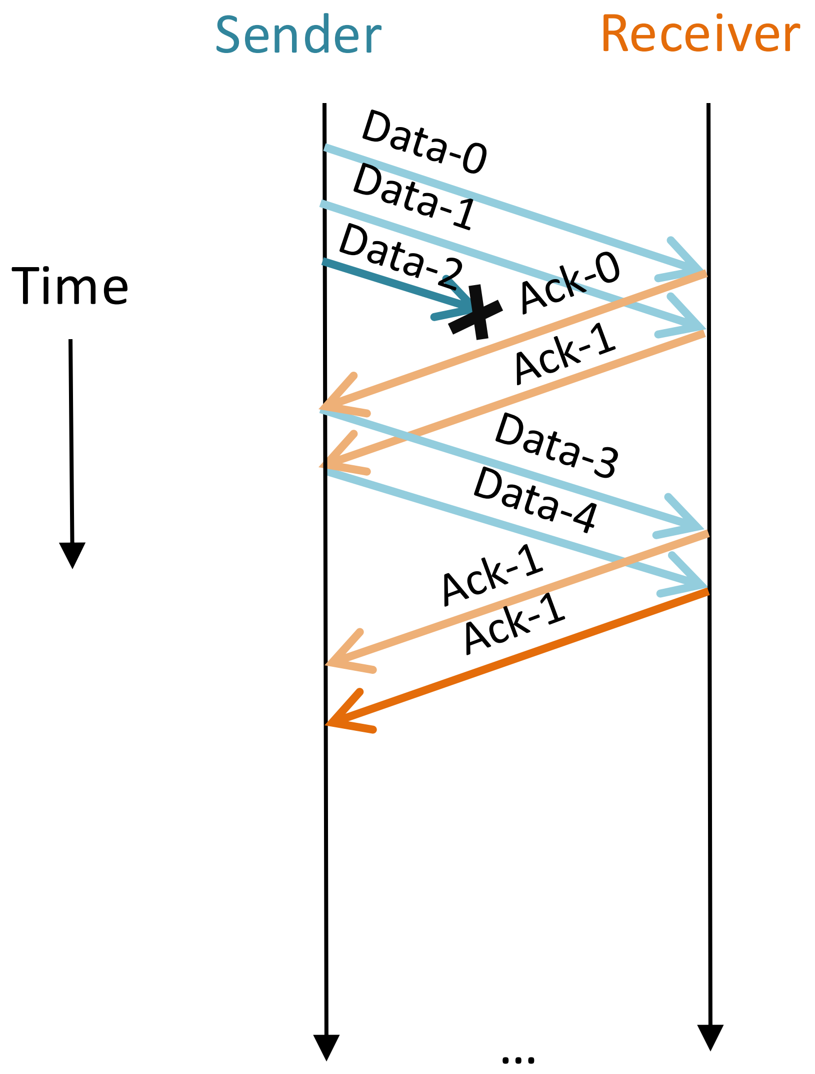
Go-Back-N


Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
...

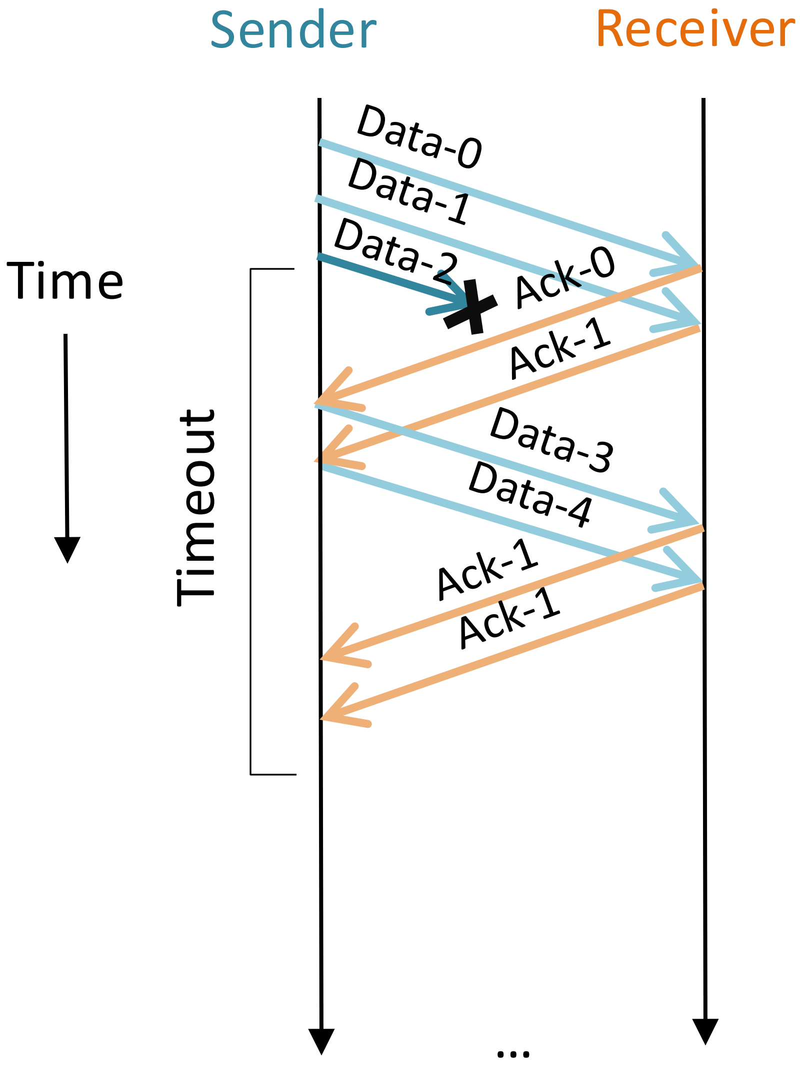
Go-Back-N


Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
…

Go-Back-N

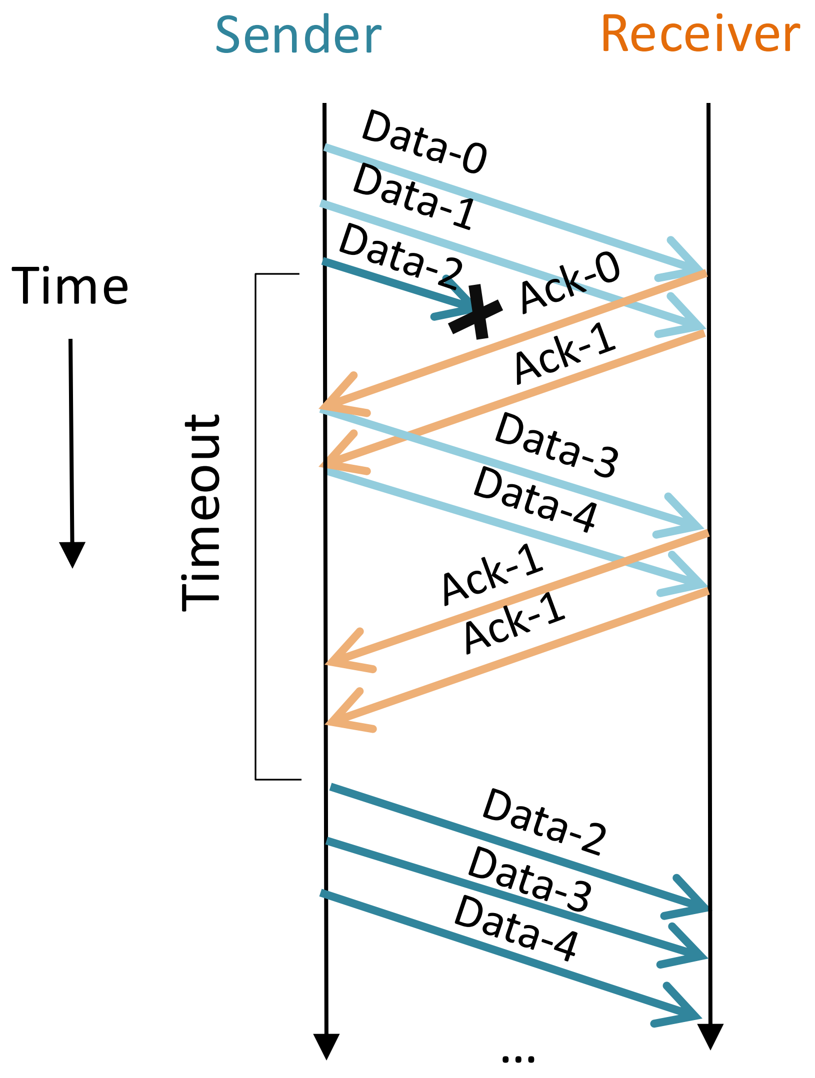

Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
...

Go-Back-N

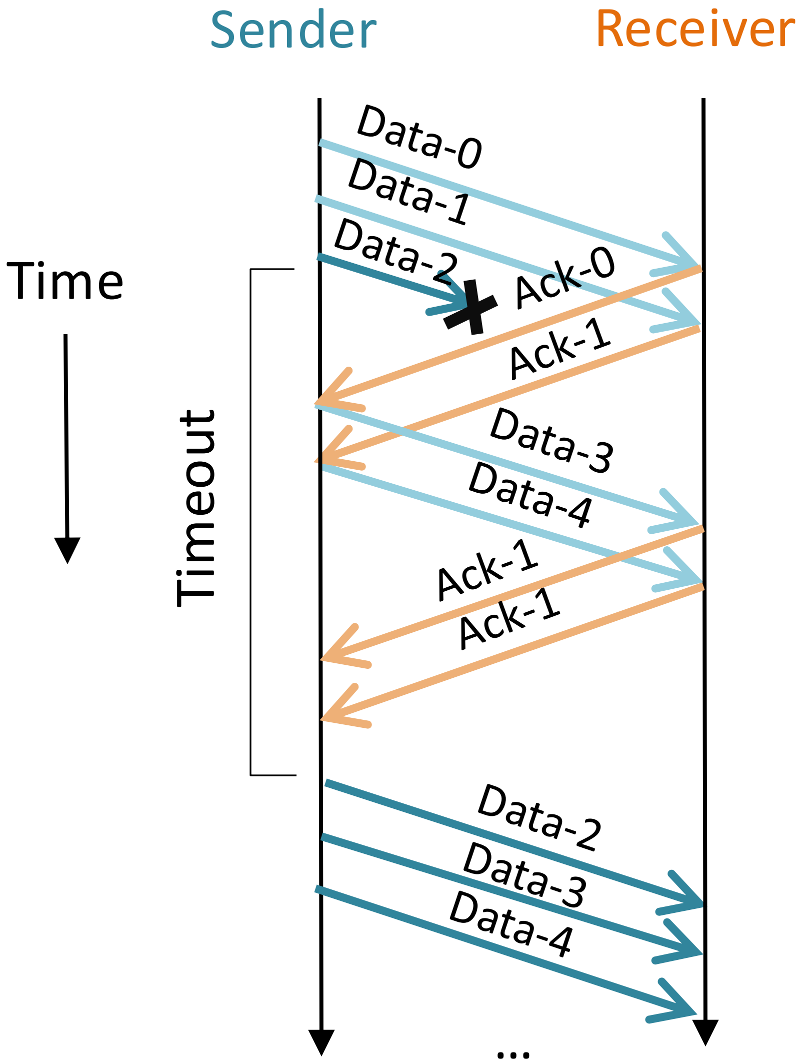


Sender
Receiver
Time
Timeout
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
...

Go-Back-N

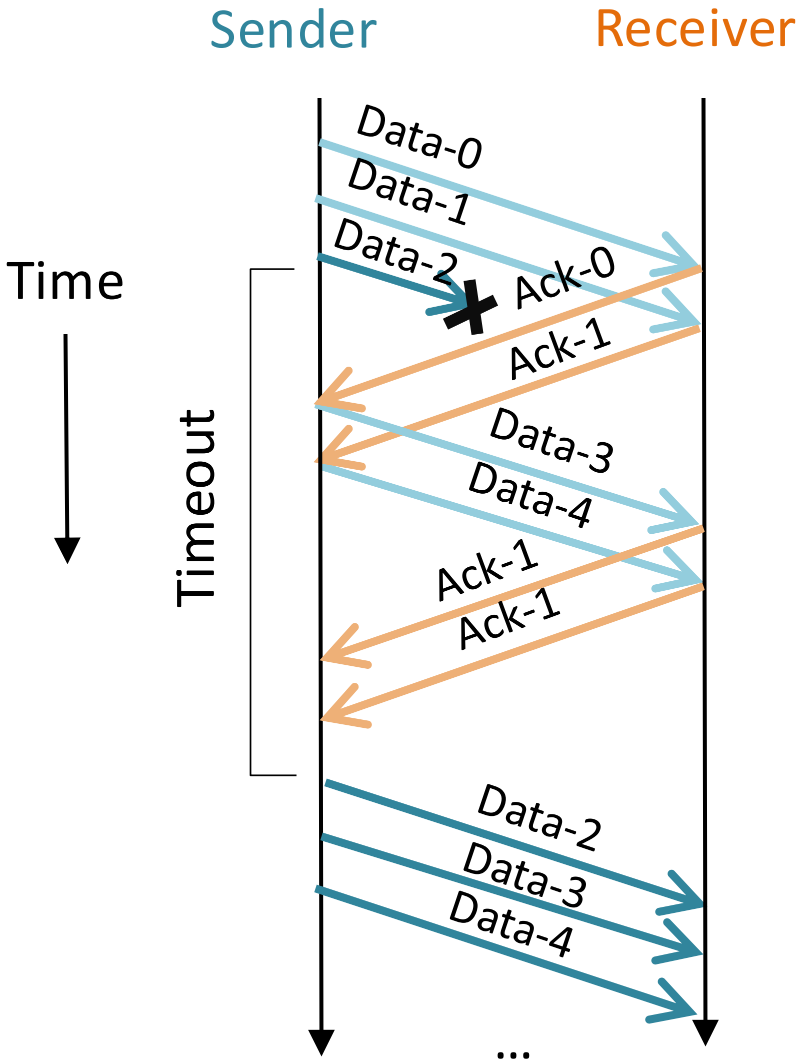

Sender
Receiver
Time
Timeout
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
Data-2
Data-3
Data-4
...

Go-Back-N



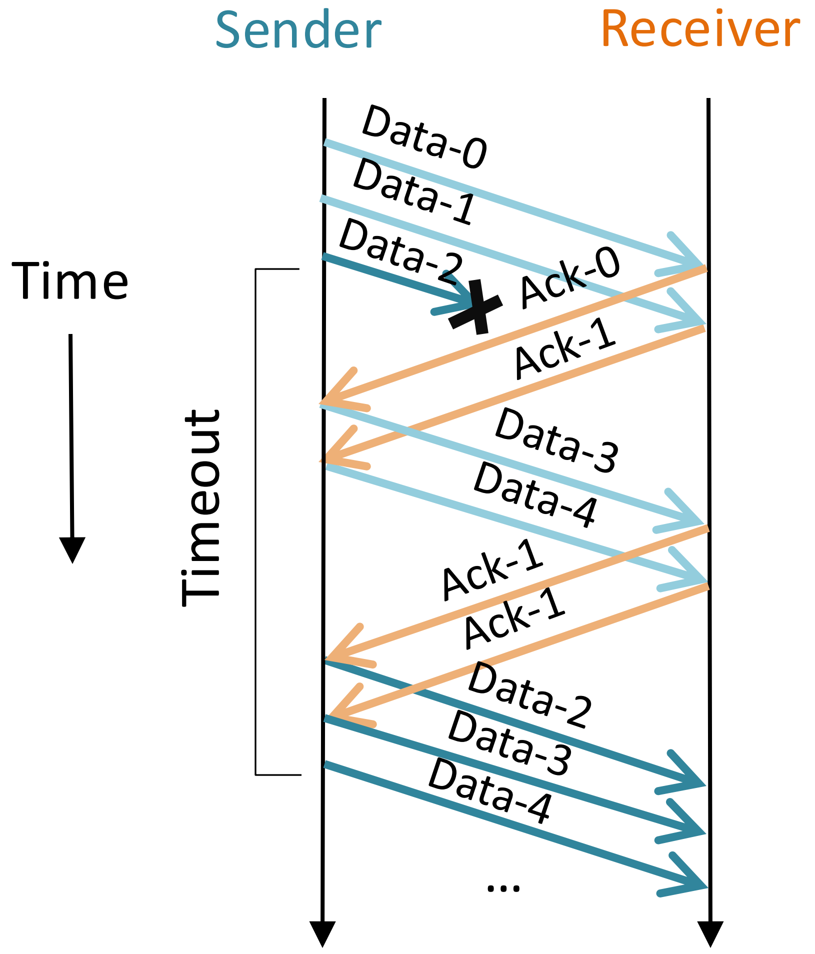
- Retransmit from point of loss
  - Segments between loss event and retransmission are ignored
  - "Go-back-N" if a timeout event occurs

Go-Back-N Performance Optimization



- Can we optimize the performance? Yes!

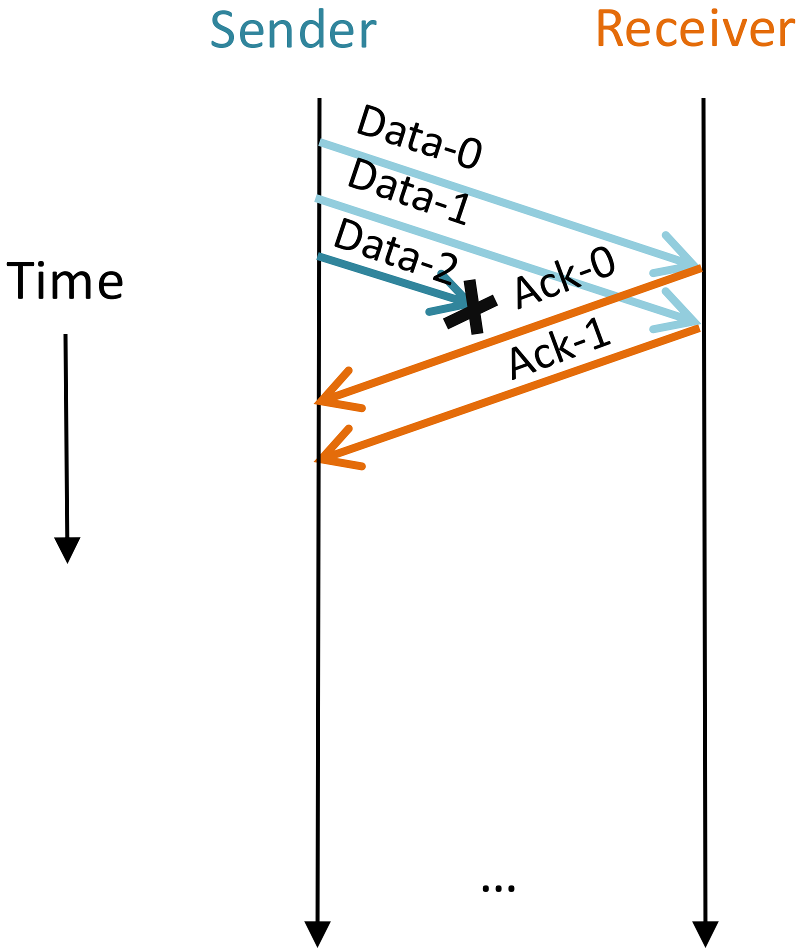
Go-Back-N: Performance Optimization



Animation

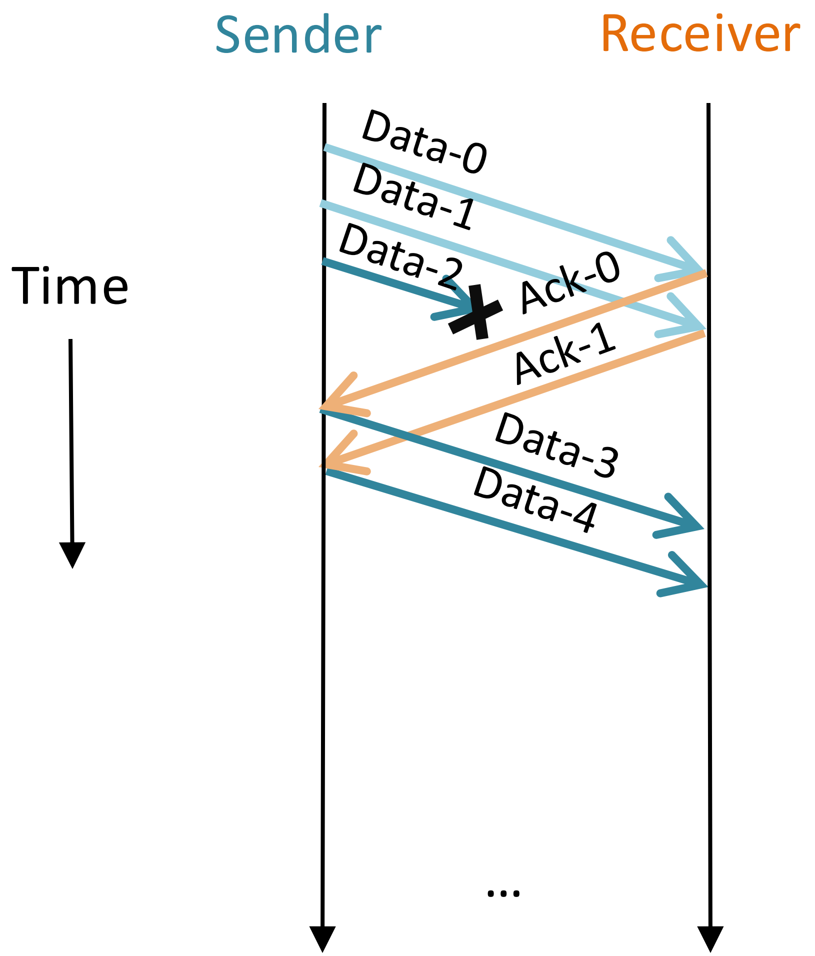
- https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_8/cw/content/interactiveanimations/go-back-n-protocol/index.html

Selective Repeat

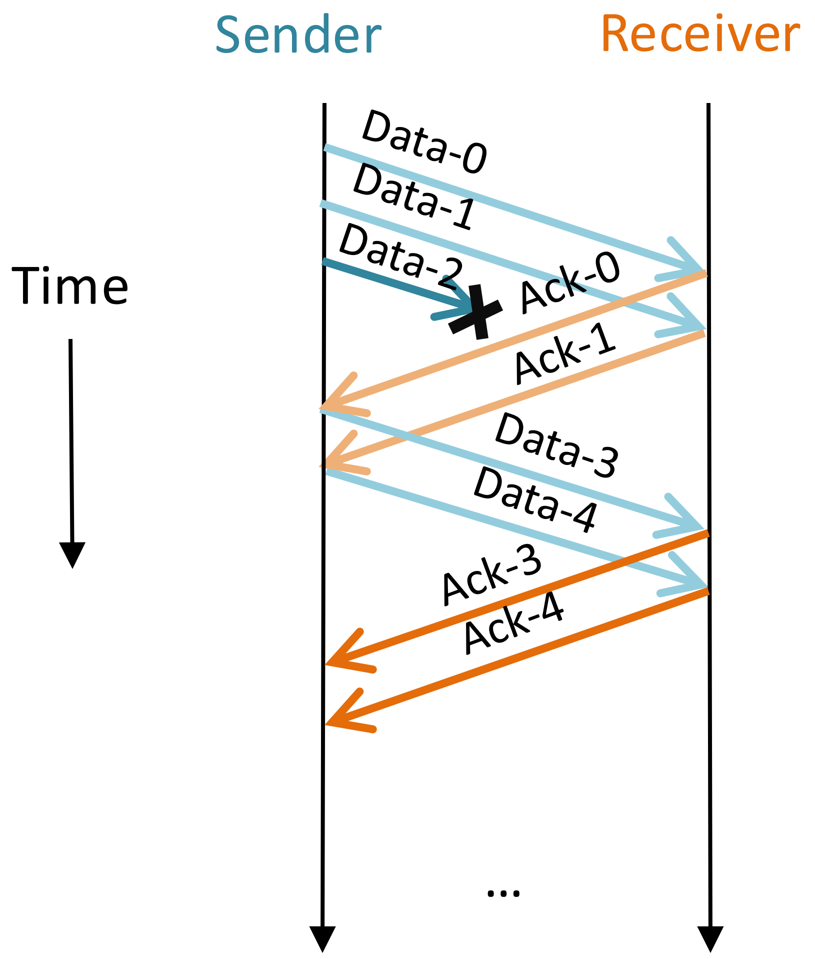


- Receiver ACKs each segment individually (not cumulative)
- Sender only resends those not ACKed

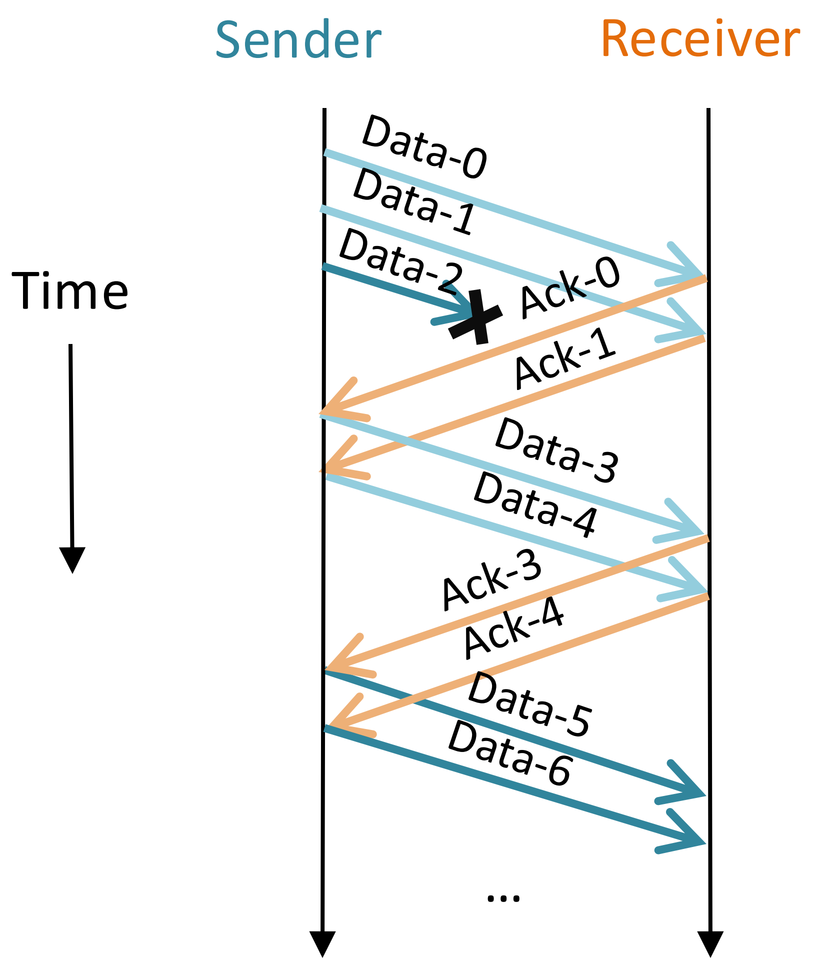
Selective Repeat



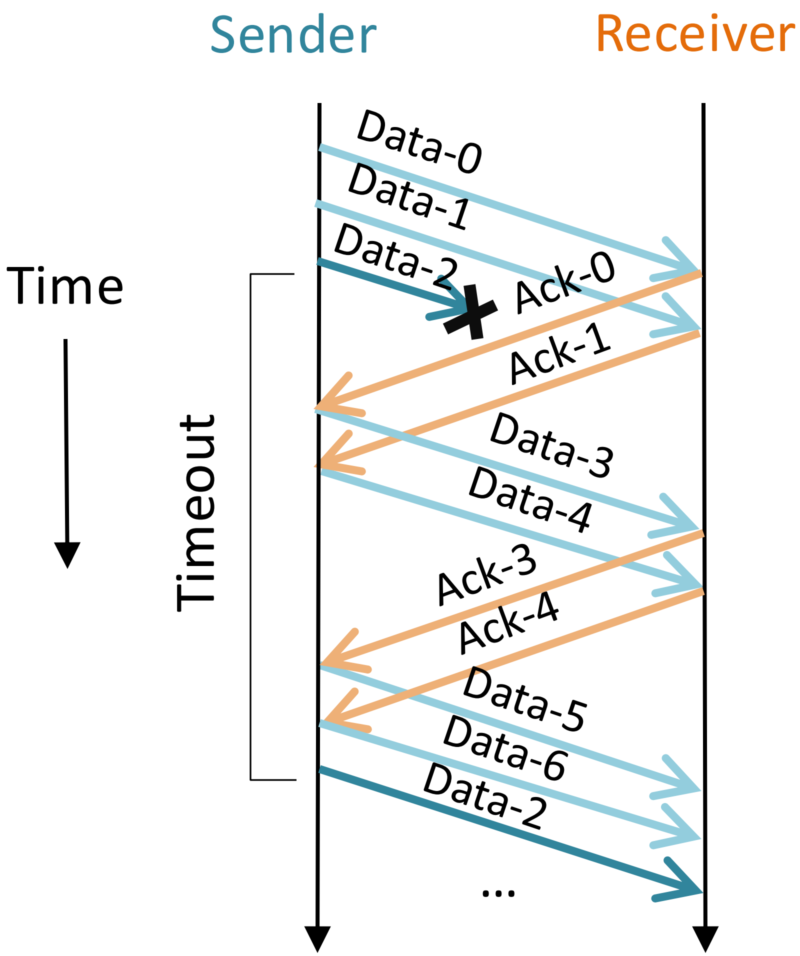
Selective Repeat



Selective Repeat



Selective Repeat



- Receiver ACKs each segment individually (not cumulative)
- Sender only resends those not ACKed

Animation

- https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_8/cw/content/interactiveanimations/selective-repeat-protocol/index.html

ARQ Alternatives

- Can’t afford the RTT’s or timeouts?
- When?
  - Broadcasting, with lots of receivers
  - Very lossy or long-delay channels (e.g., space)
- Use redundancy – send more data
  - Simple form: send the same message N times
  - More efficient: use “erasure coding”
  - For example, encode your data in 10 pieces such that the receiver can piece it together with any subset of size 8.

Practical Reliability Questions

- What does connection establishment look like?
- How do we choose sequence numbers?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How should we choose timeout values?
- How many segments should be pipelined?