CS 43: Computer Networks

Reliable Transport and TCP

October 29, 2024

SWARTHMORE COLLEGE

Transport Layer

Today

- Principles of reliability
- Class of protocols: Automatic Repeat Requests

Moving down a layer!

Application Layer

Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium
(copper, the air, fiber)

Transport Layer perspective

Application is the boss

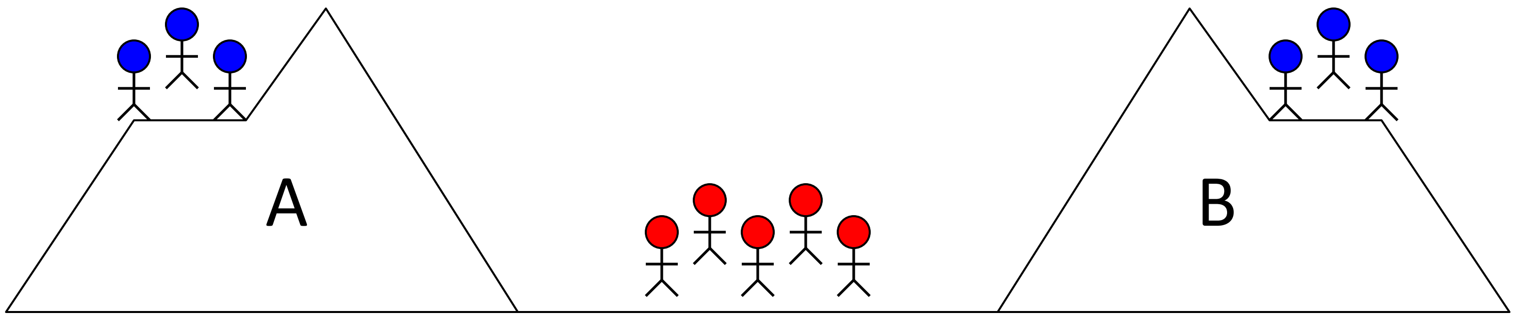
Transport: executing within the OS kernel

Network: ours to command!

Today

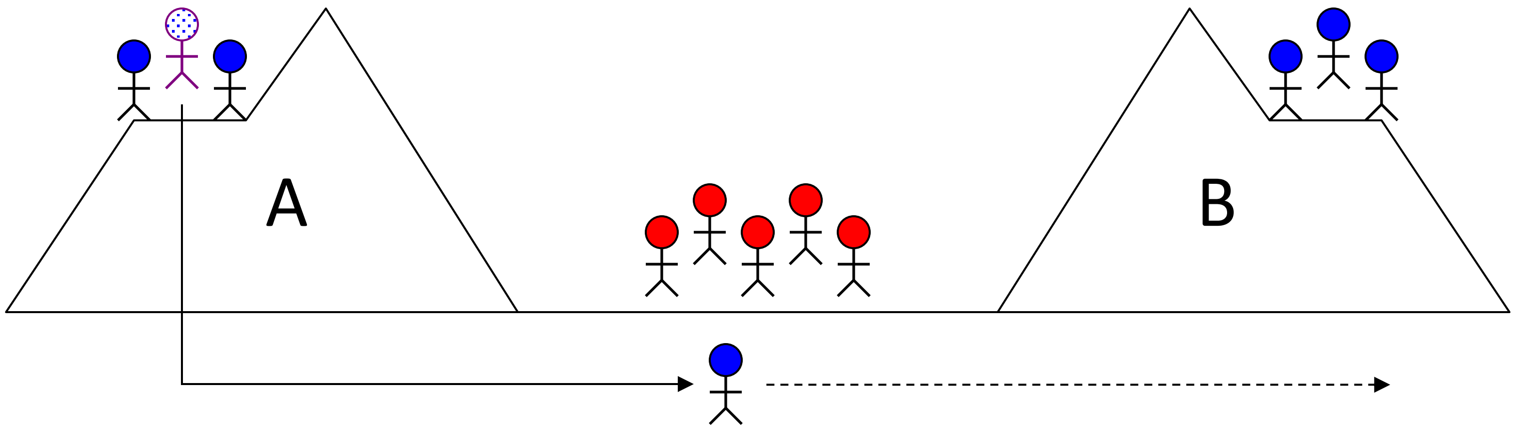
- Principles of reliability
  - The Two Generals Problem
- Automatic Repeat Requests
  - Stop and Wait
  - Timeouts and Losses
  - Pipelined Transmission

The Two Generals Problem



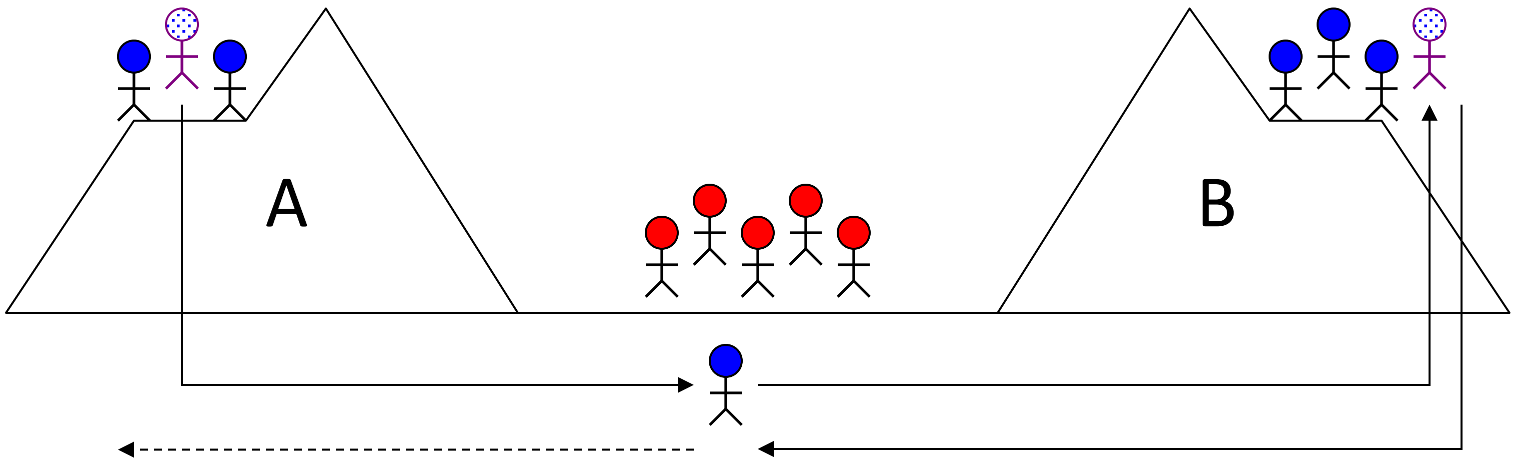
- Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

The Two Generals Problem



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

The Two Generals Problem

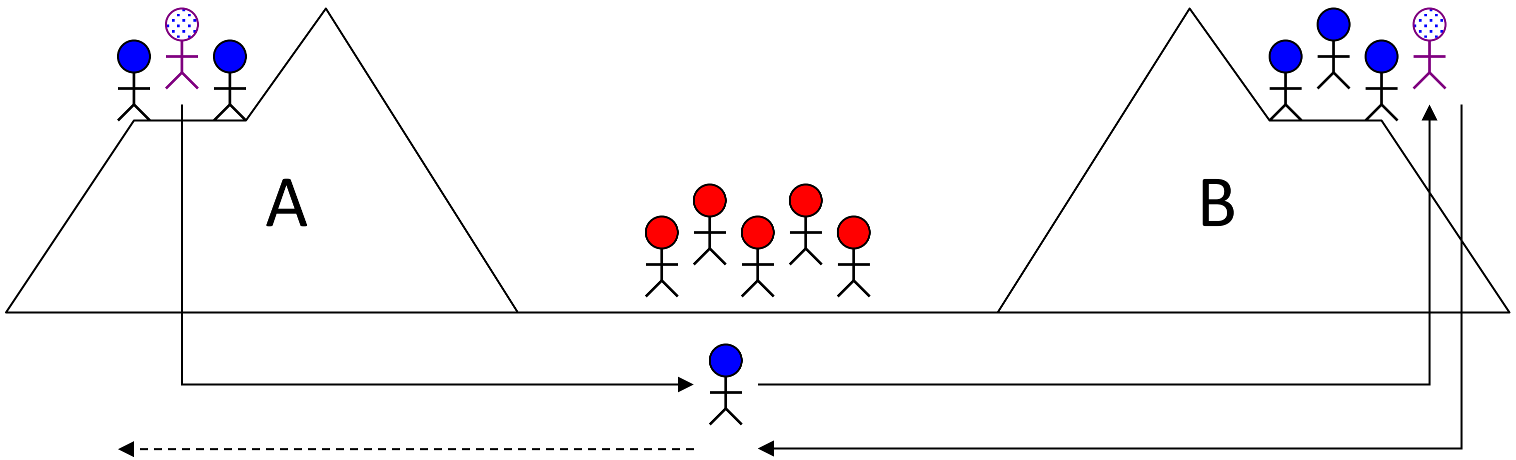


- How to be sure messenger made it?
  - Send acknowledgment: “I delivered message”

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

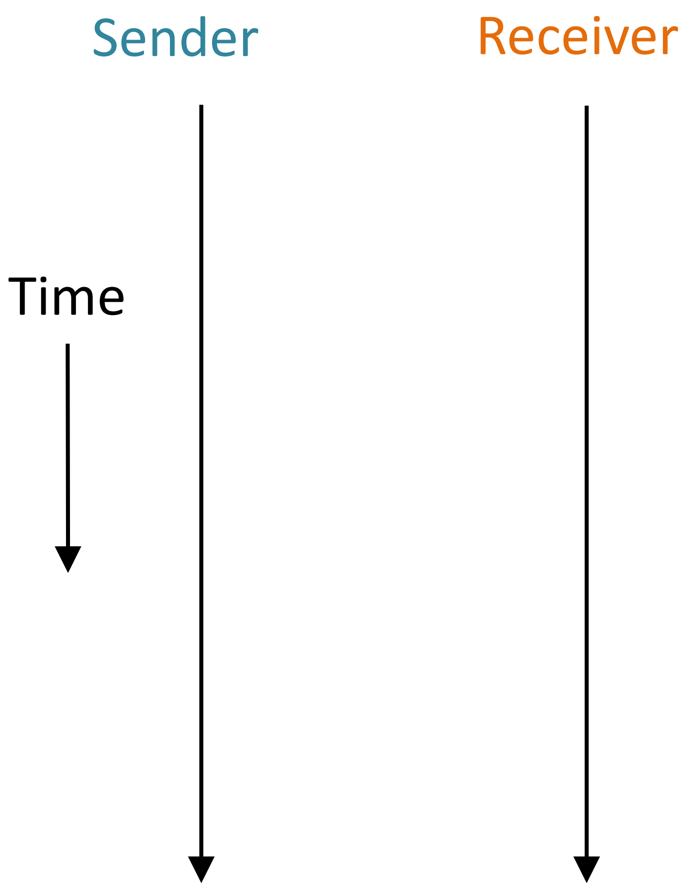
ARQ Broad Classifications

1. Stop-and-wait

Stop and Wait

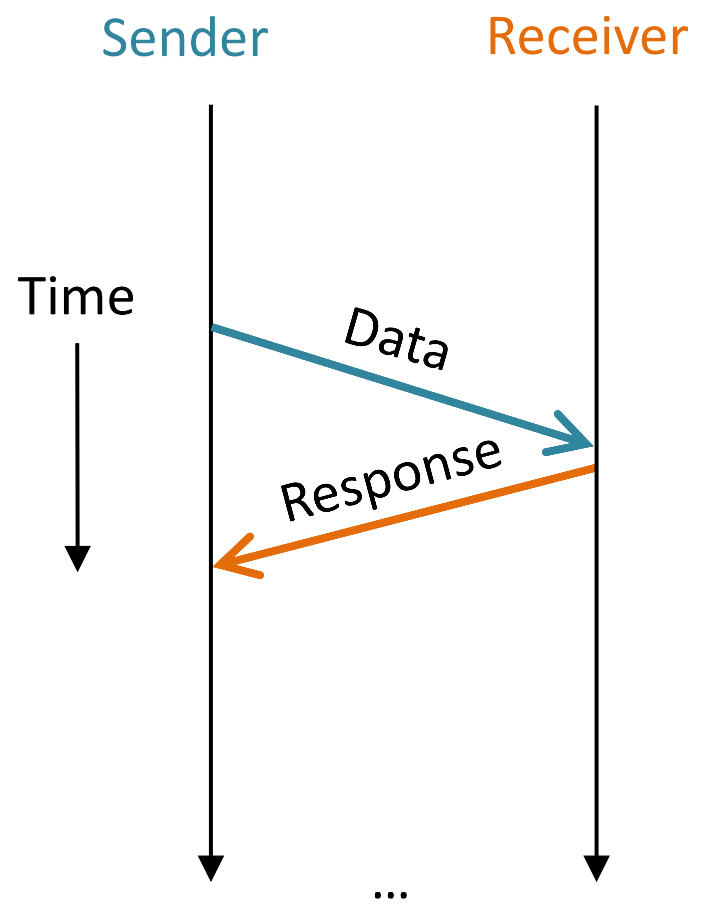
We have:

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



Stop and Wait

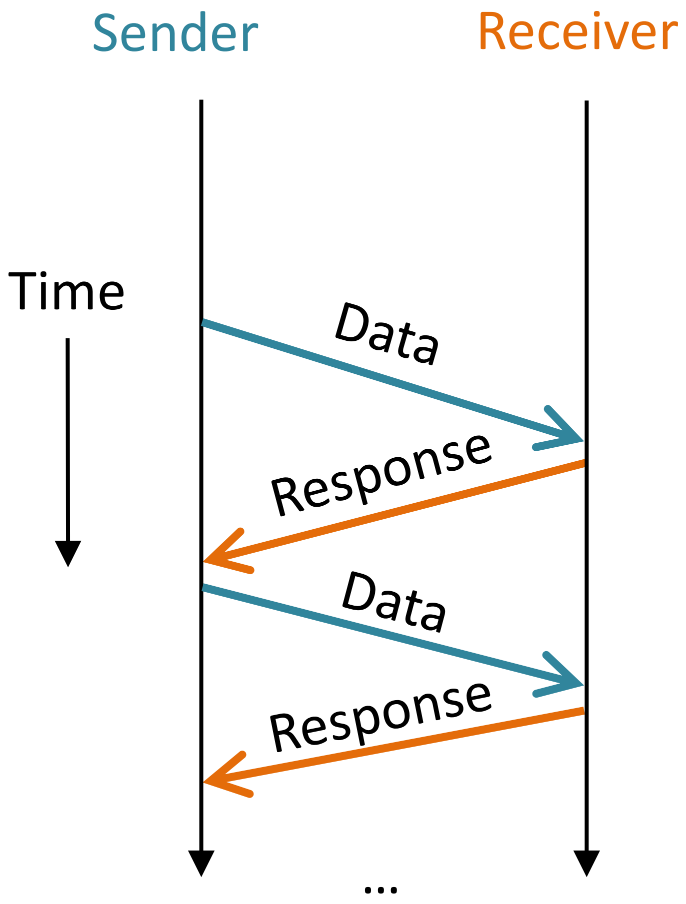
Sender sends data and waits till they get the response message from the receiver.



Buffer data, and don't send till response received

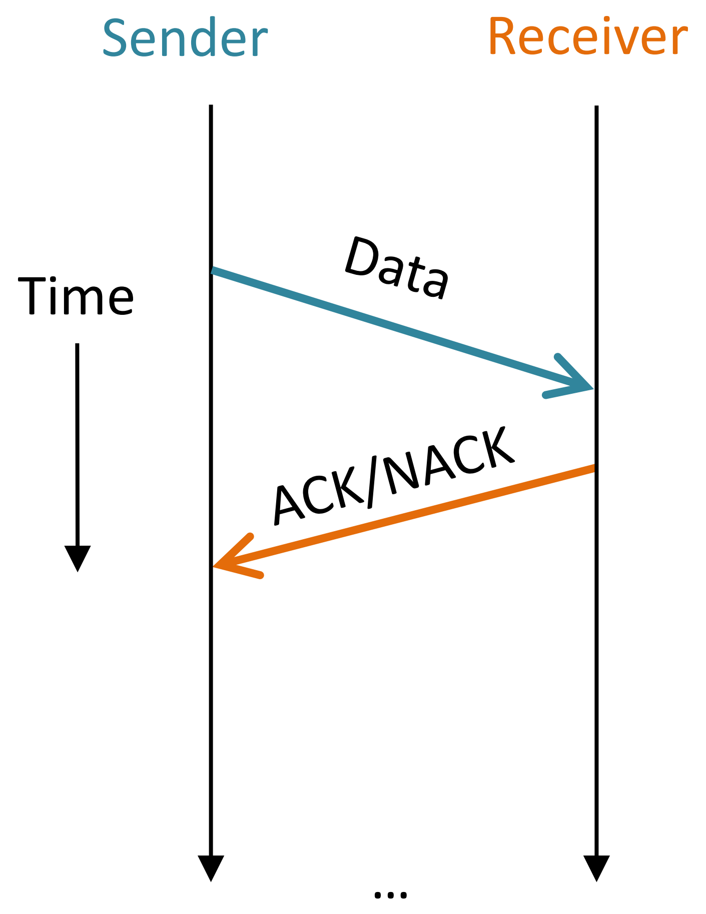
Stop and Wait

- 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

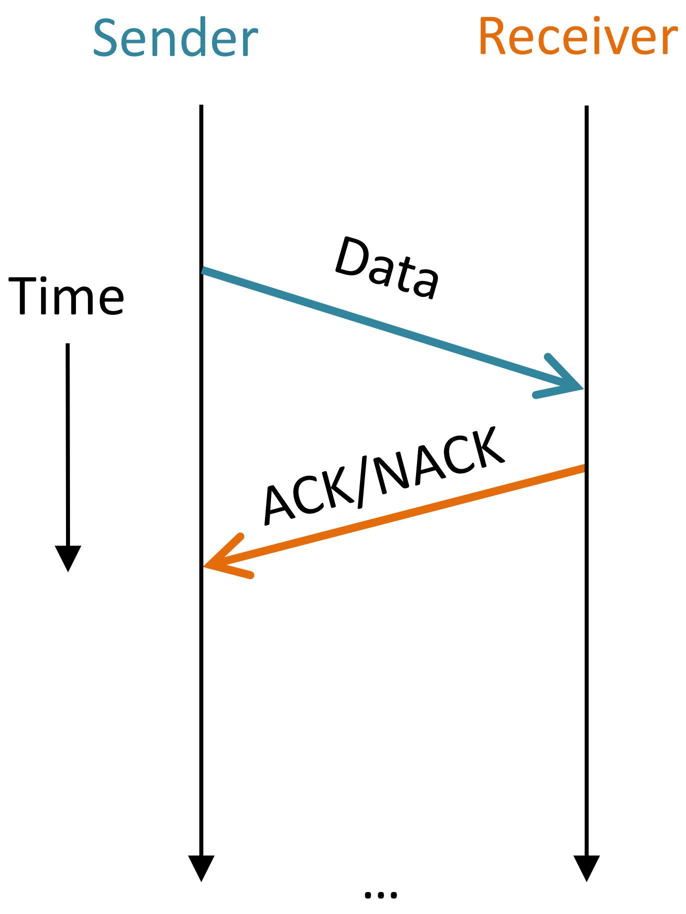


Could we do this with just ACKs or just NACKs?

Error detection mechanism: checksum

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

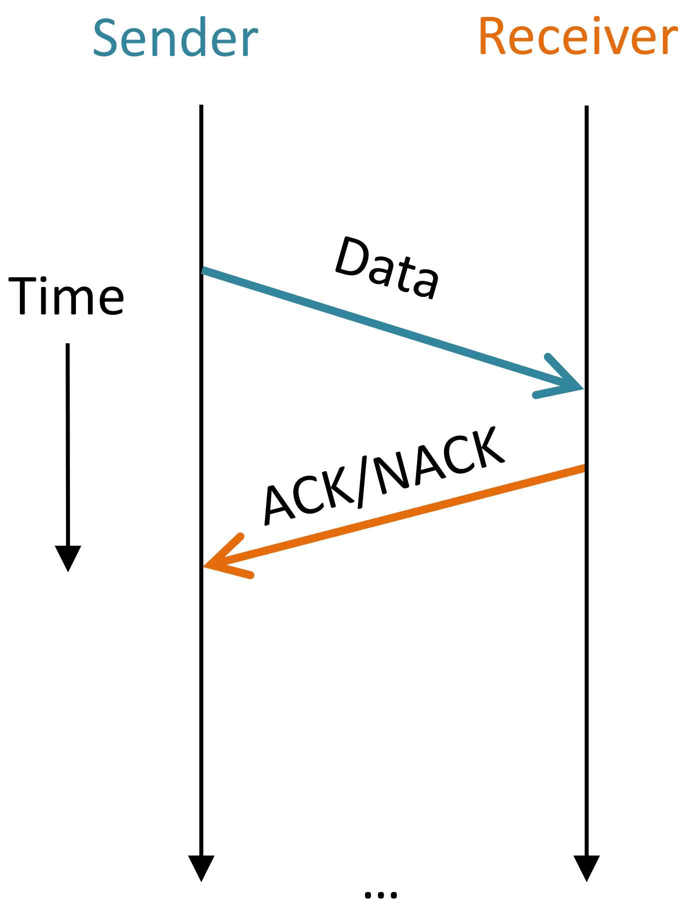
A. No, we need them both.
B. Yes, we could do without one of them, but we'd need some other mechanism.
C. Yes, we could get by without one of them.



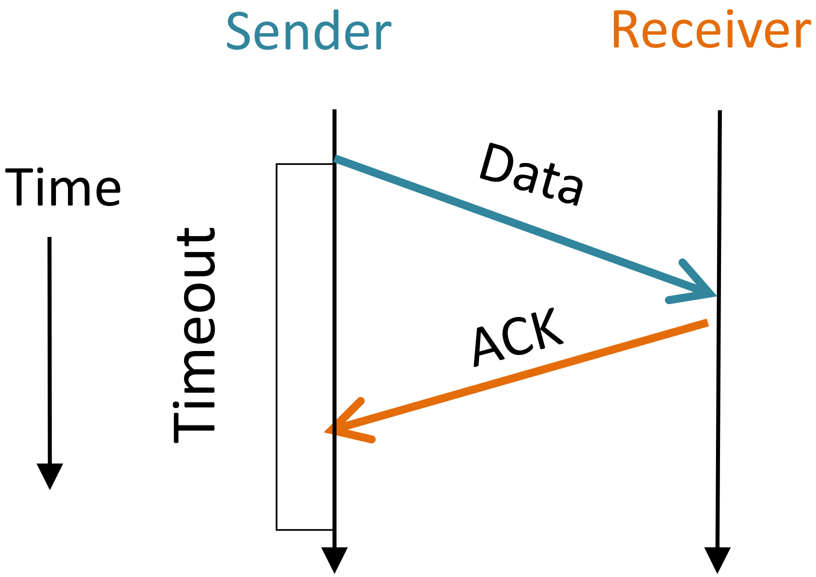
Could we do this with just ACKs or just NACKs?

- With only ACK, we could get by with a timeout.
- With only NACK, we couldn't advance (no good).

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.

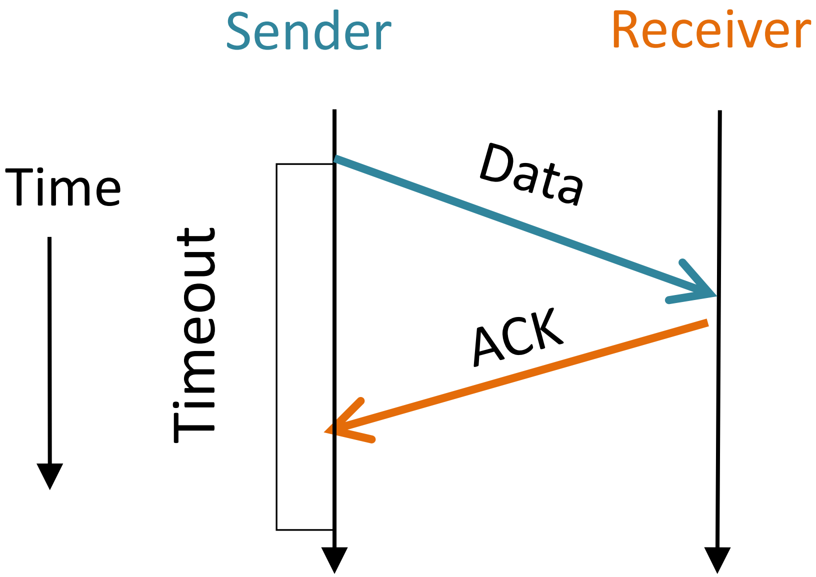


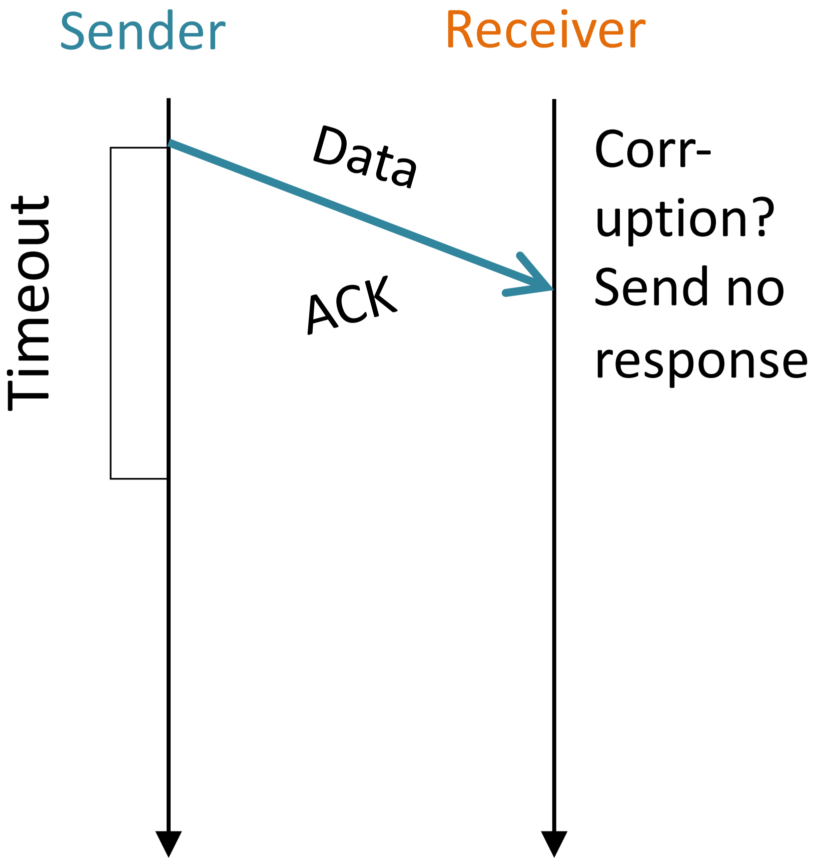
Timeouts and Losses



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

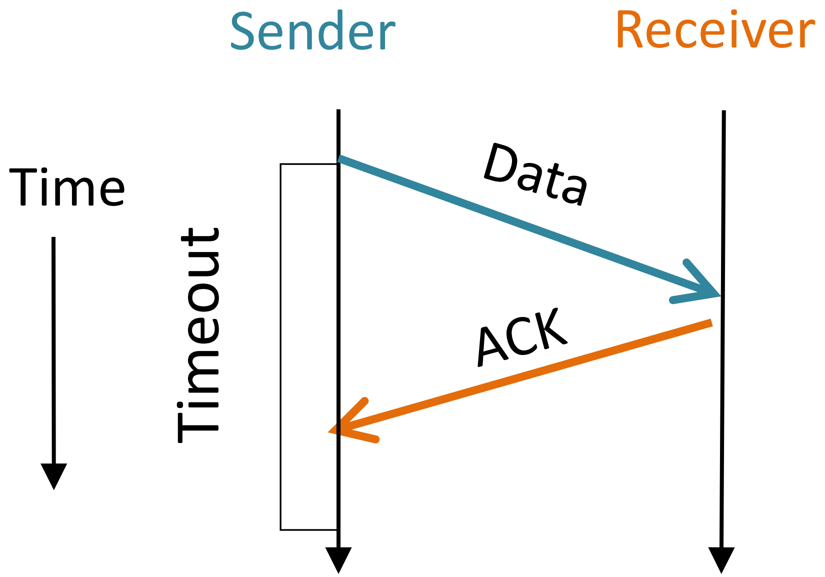
Timeouts and Losses

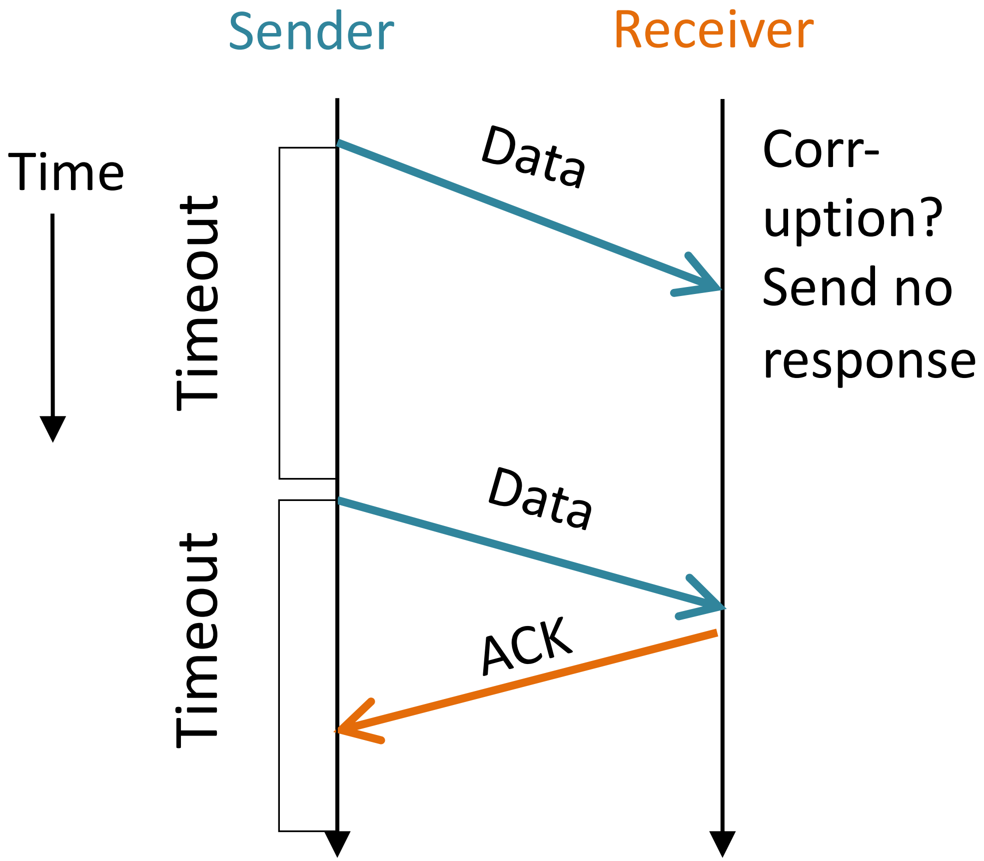




- Sender starts a clock. If no response, retry.

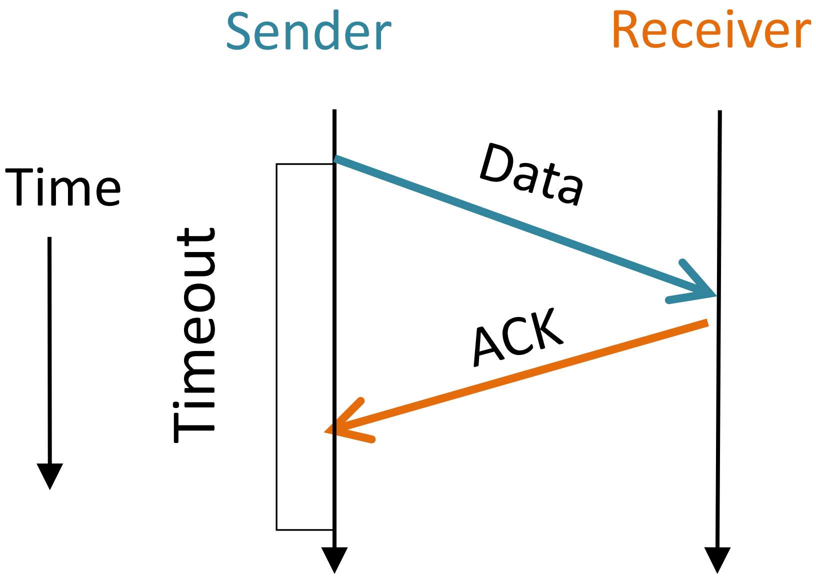
Timeouts and Losses





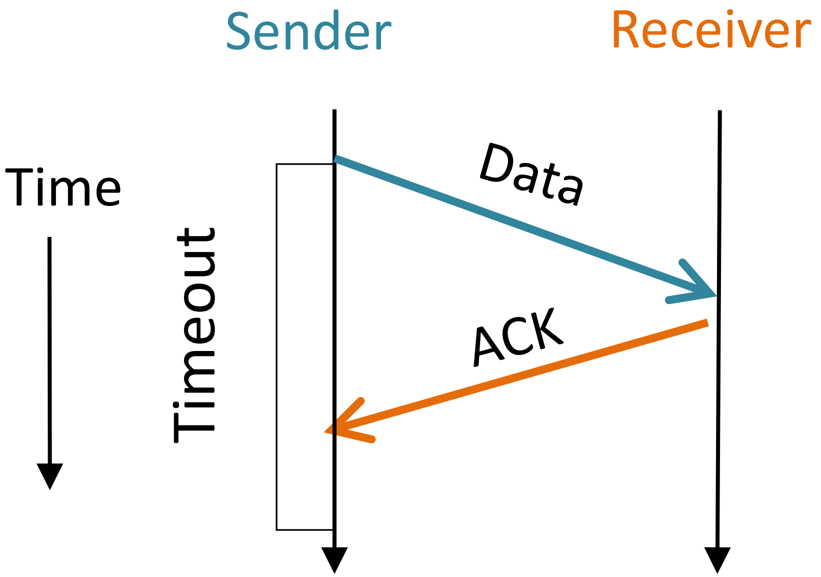
- Sender starts a clock. If no response, retry.
- Probably not a great idea for handling corruption, but it works.

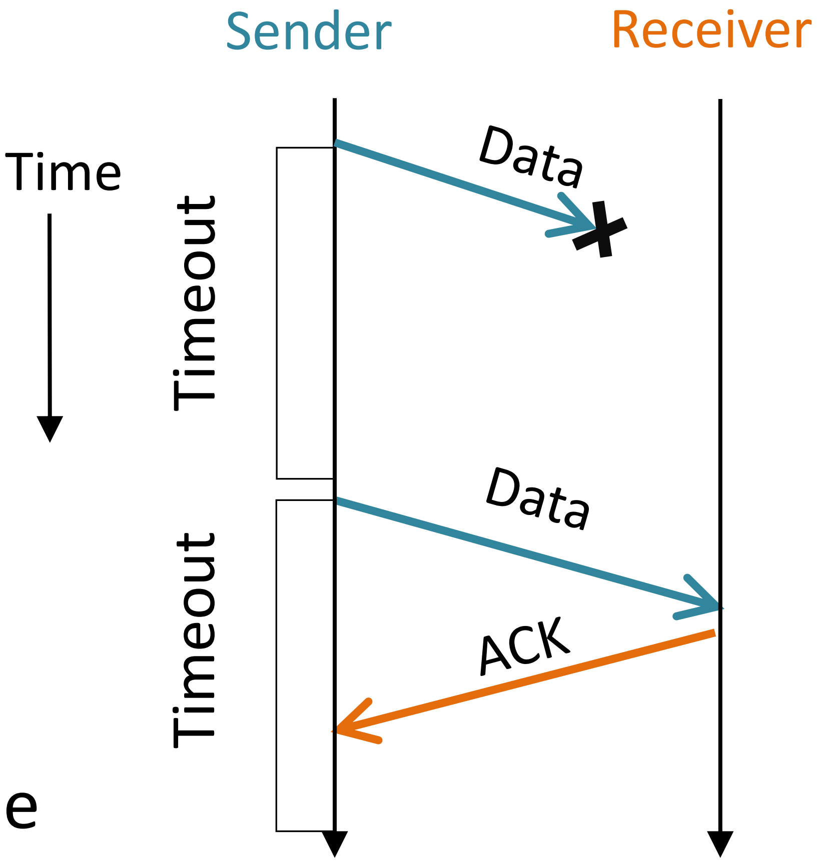
Timeouts and Losses



- Timeouts help us handle message losses too!

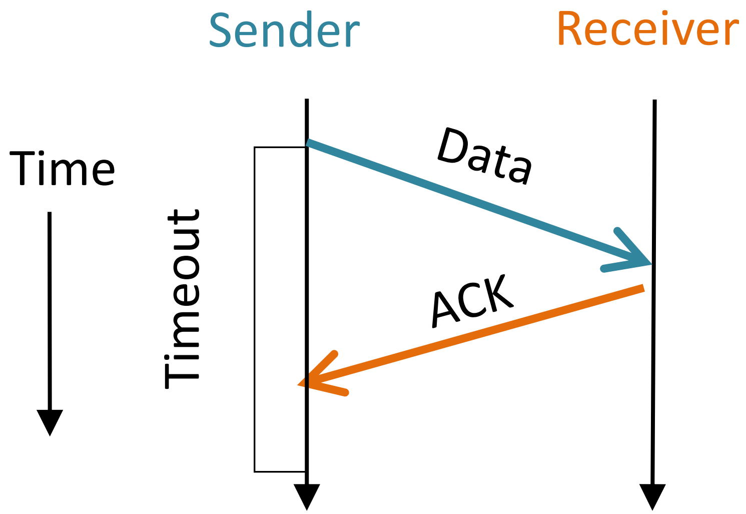
Timeouts and Losses





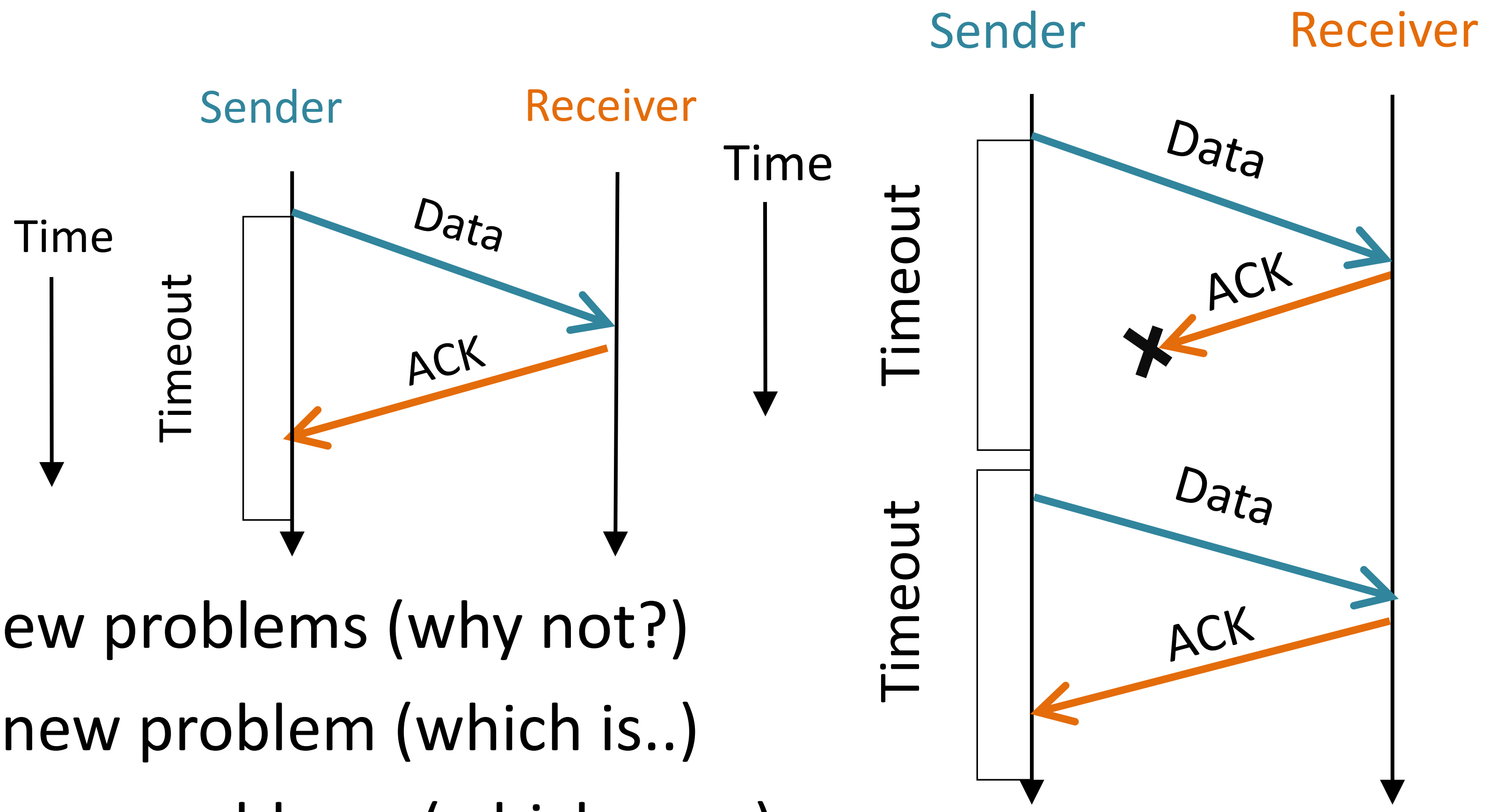
- Timeouts help us handle message losses too!

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..)

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

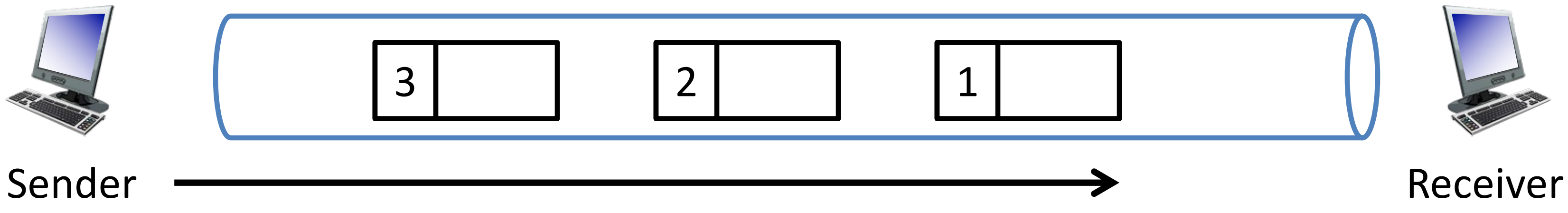
Sender

- 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: 1024 bytes

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:

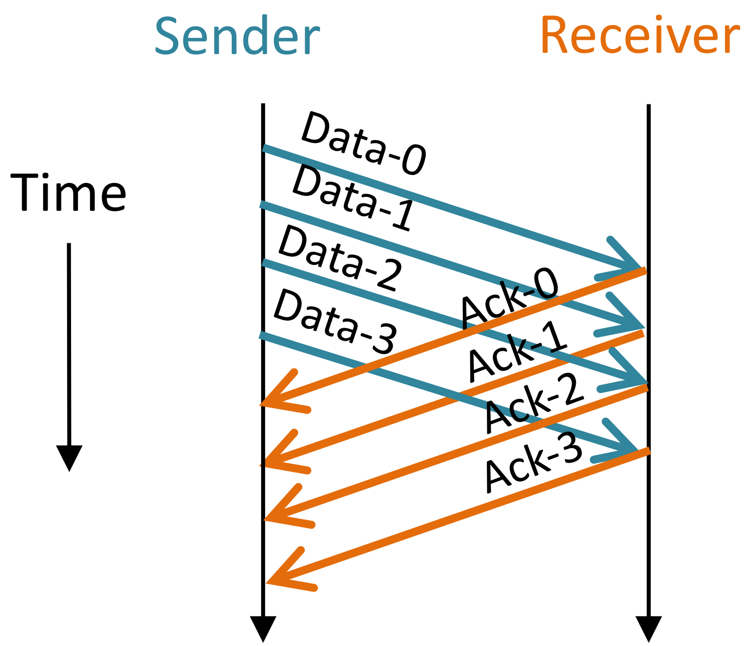
Link rate: 8 Mbps (one megabyte per second)

RTT: 100 milliseconds

Segment size: 1024 bytes

Big Problem: Performance is determined by RTT, not channel capacity!

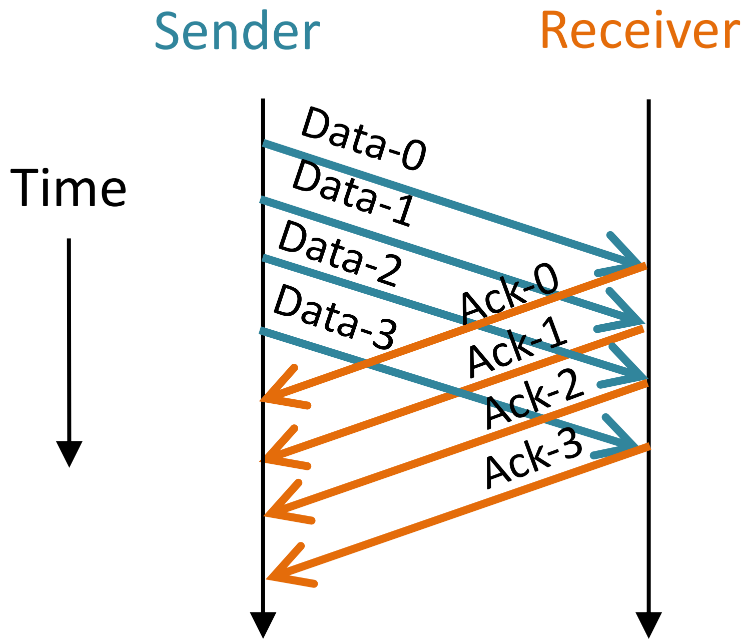
Pipelined Transmission

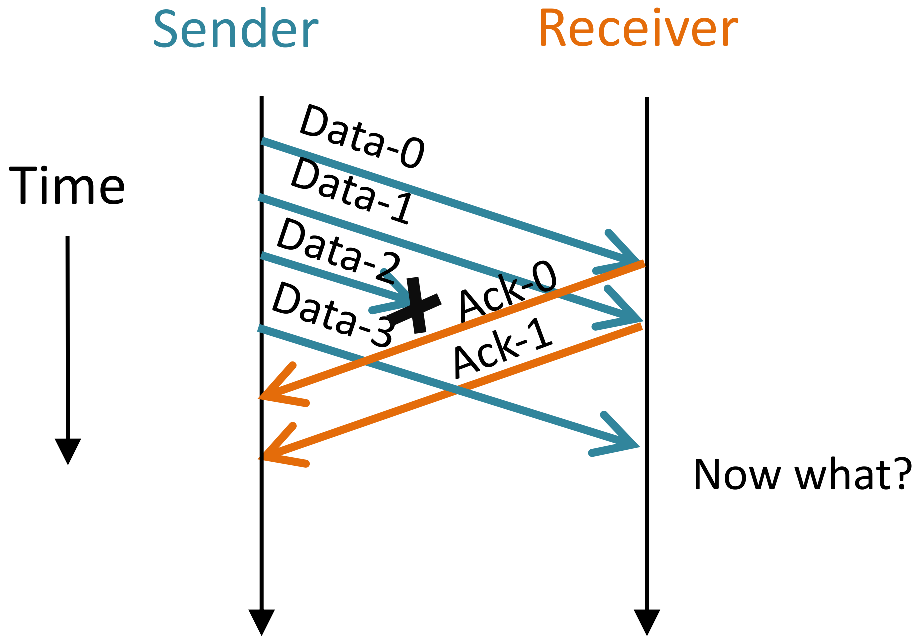


<u>Keep multiple segments "in flight"</u>

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

Pipelined Transmission

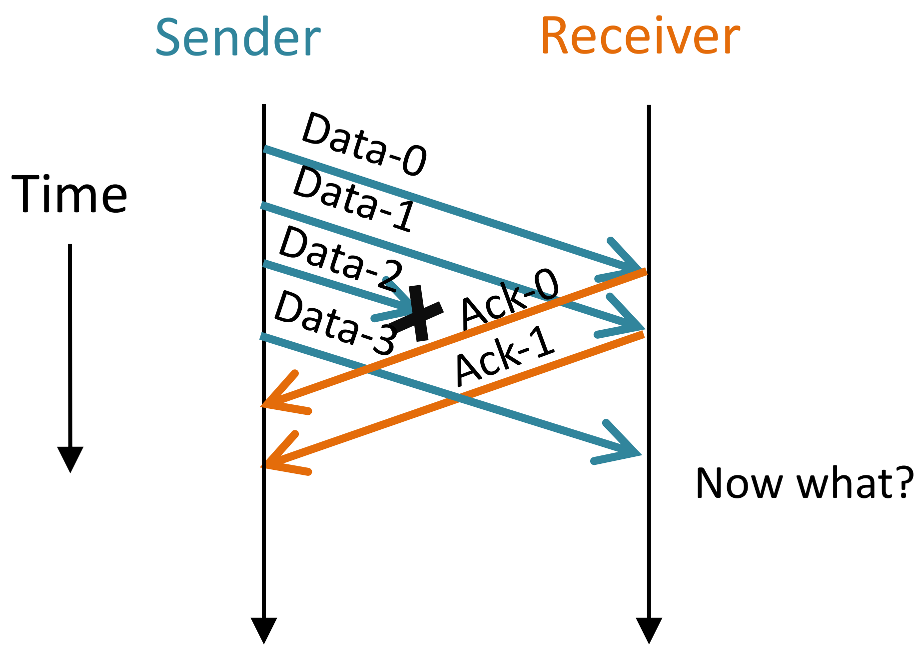




<u>Keep multiple segments "in flight"</u>

- 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?

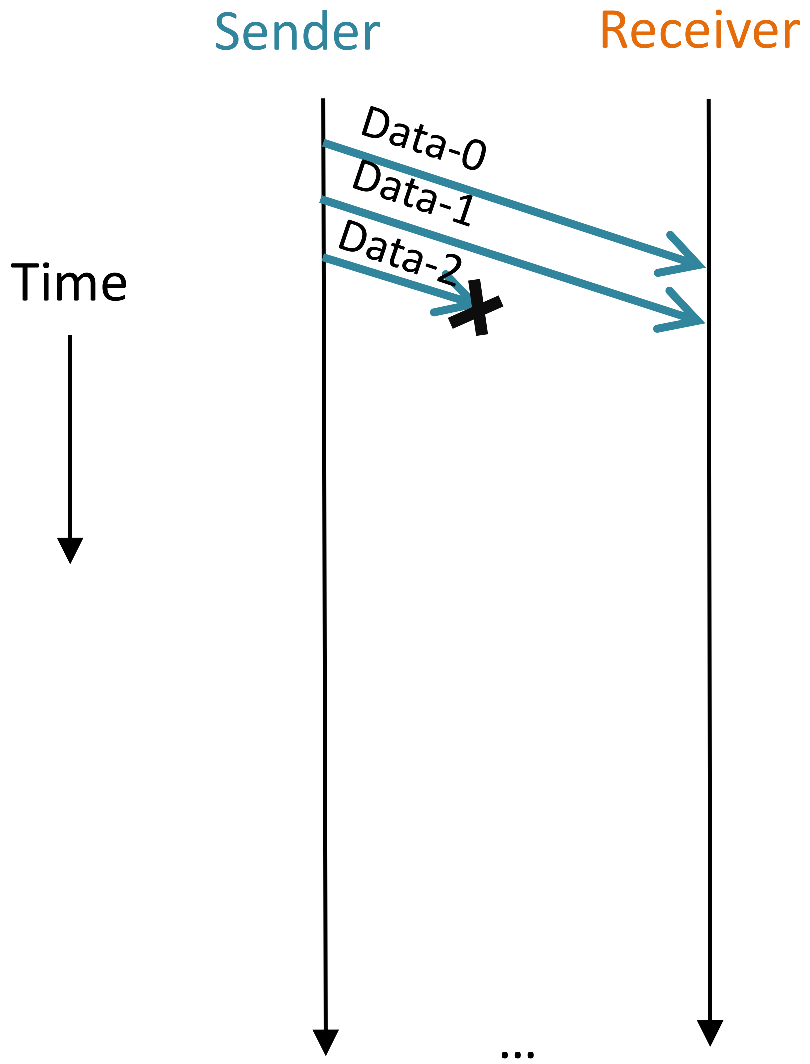
A. Start sending all data again from 0.
B. Start sending all data again from 2.
C. Resend just 2, then continue with 4 afterwards.

ARQ Broad Classifications

1. Stop-and-wait

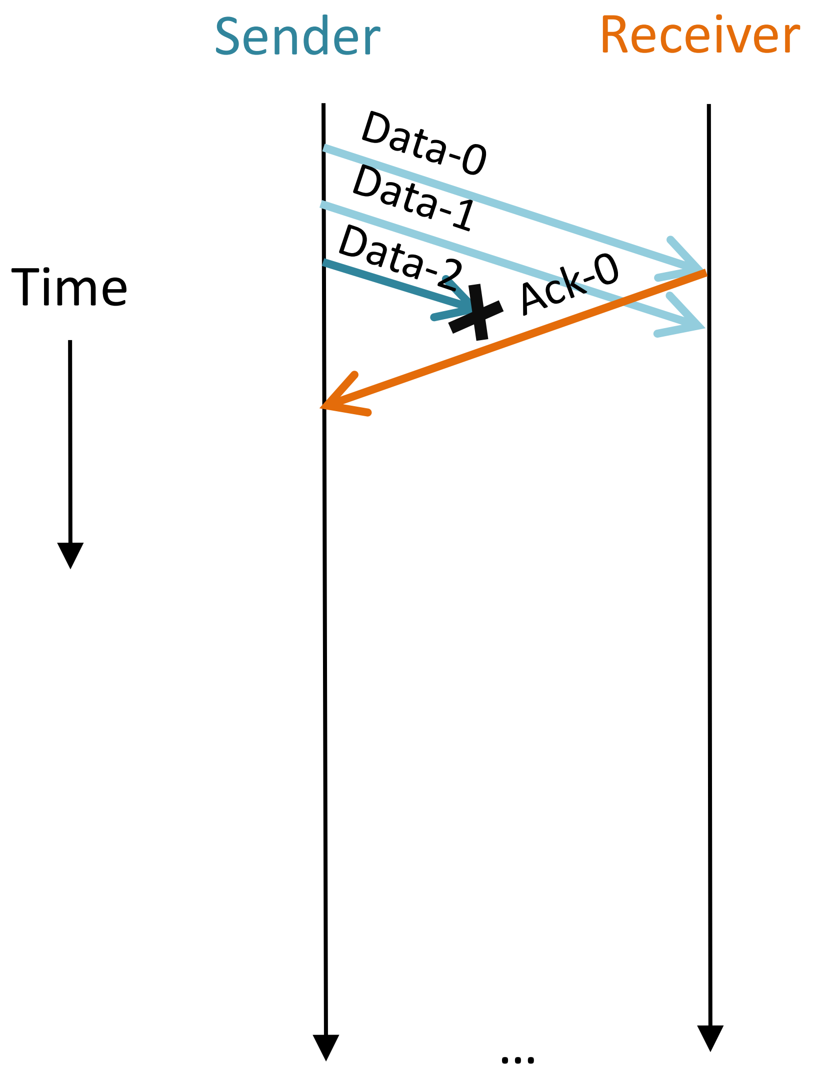
2. Go-back-N

Go-Back-N

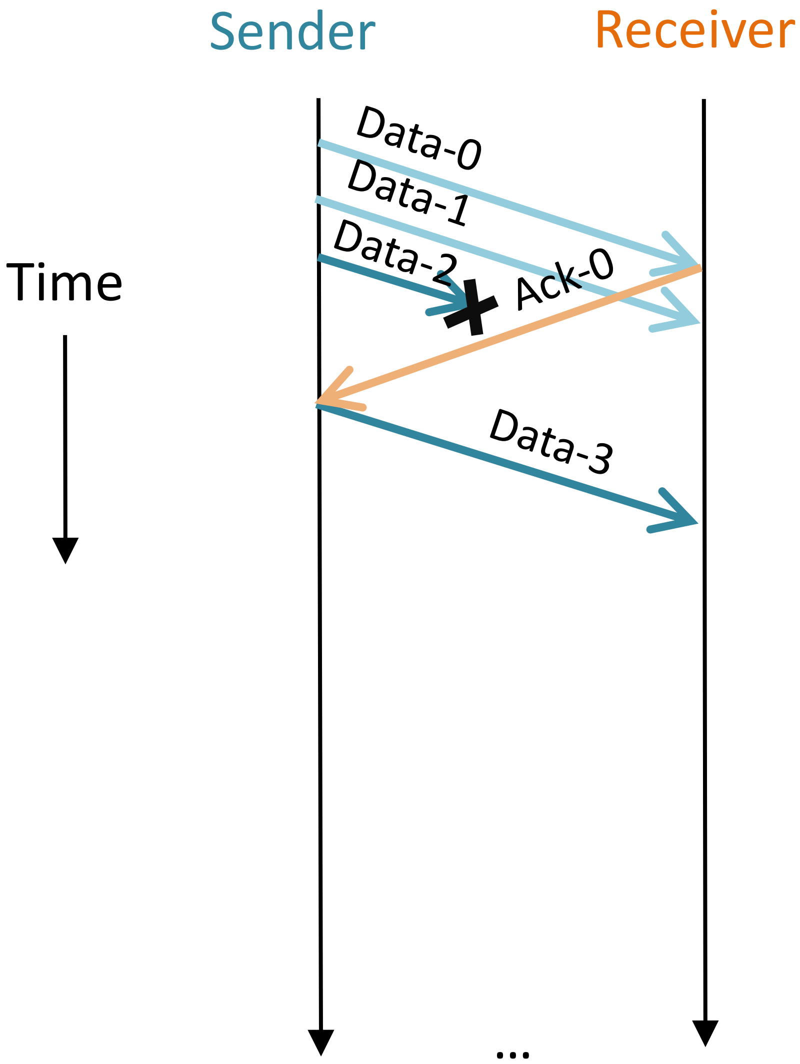


- Retransmit from point of loss
  - Segments between loss event and retransmission are ignored
  - “Go-back-N” if a timeout event occurs

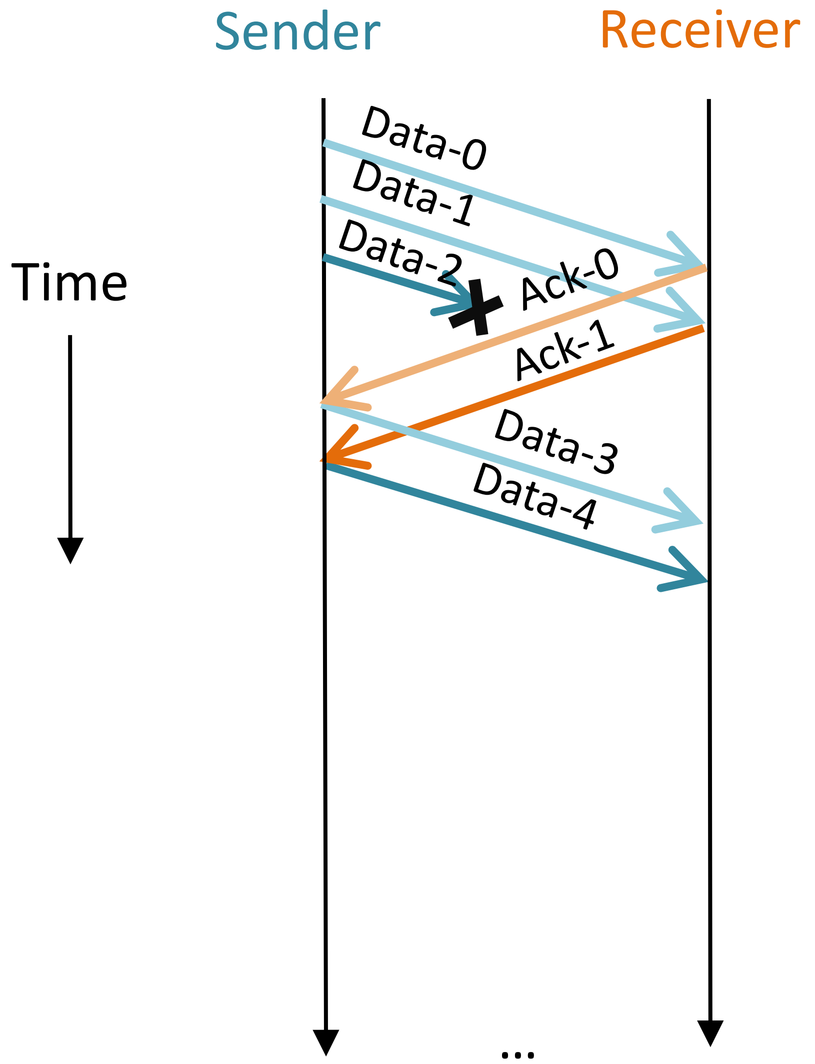
Go-Back-N



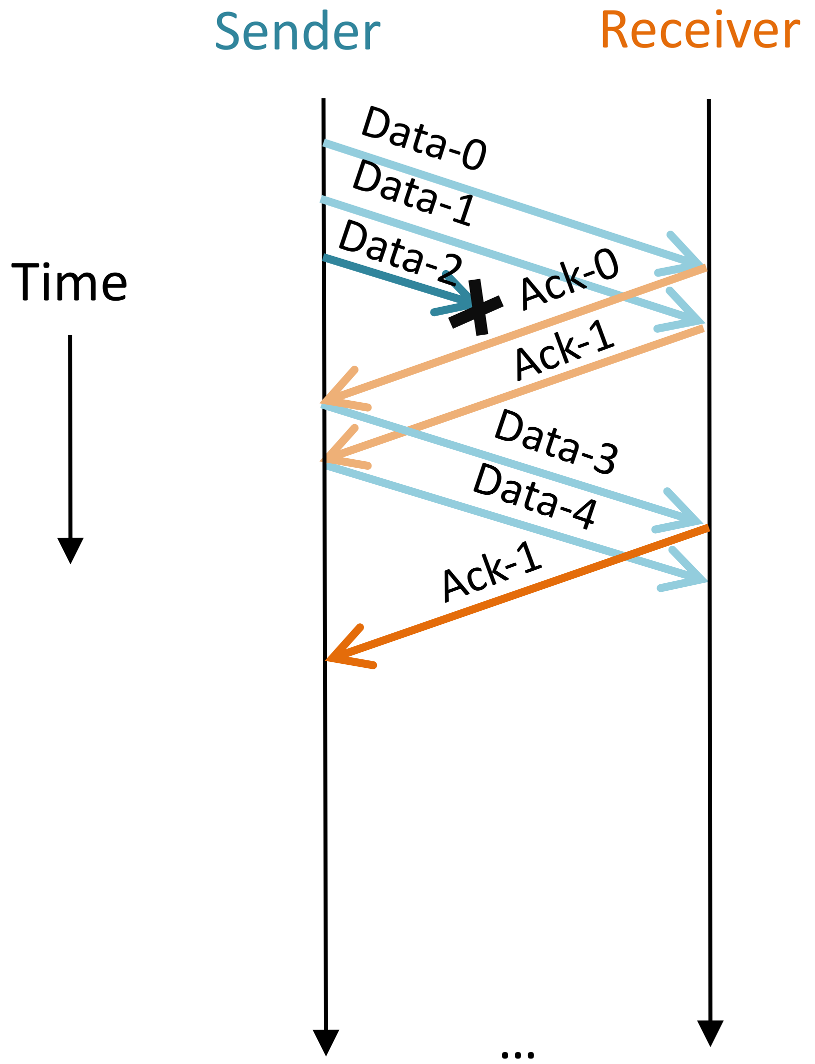
Go-Back-N



Go-Back-N

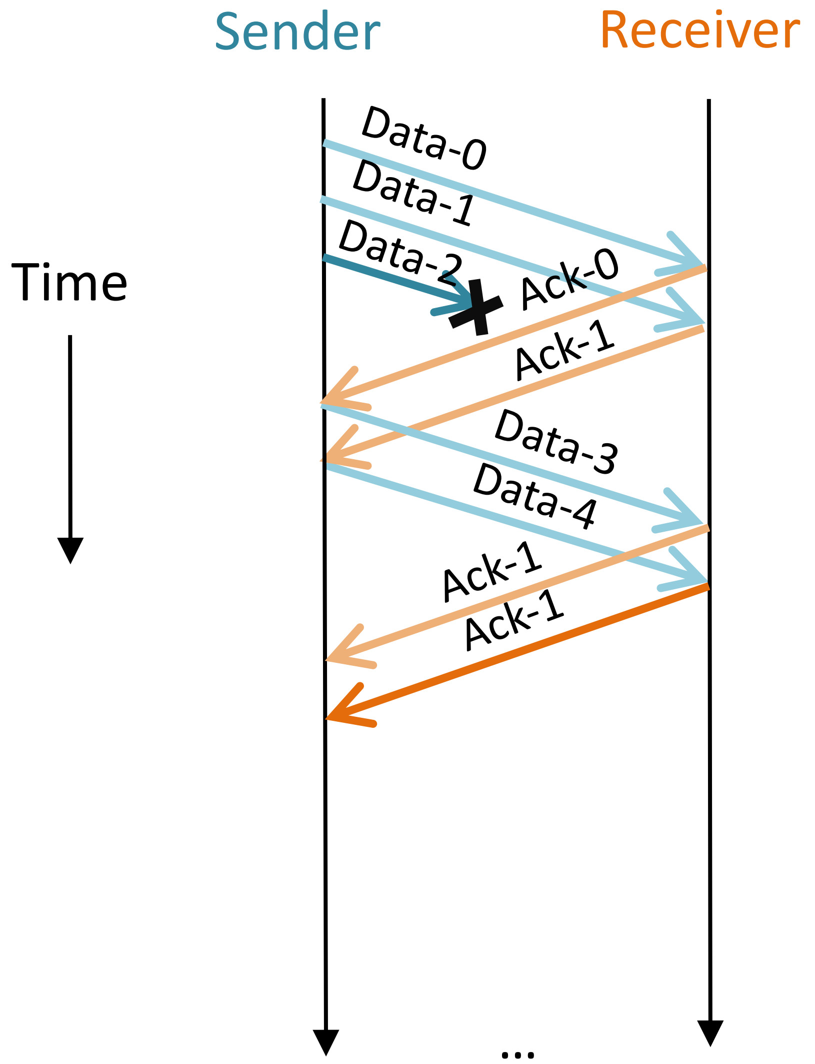

Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
...

Go-Back-N



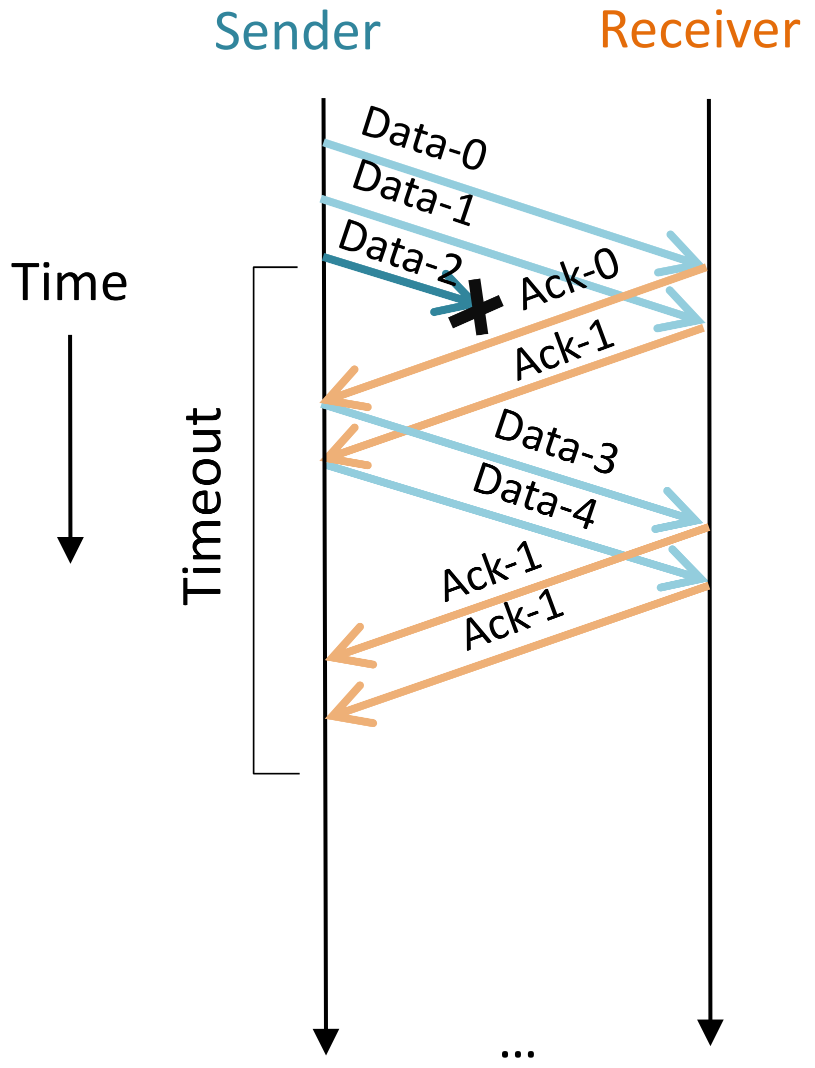
Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
...

Go-Back-N



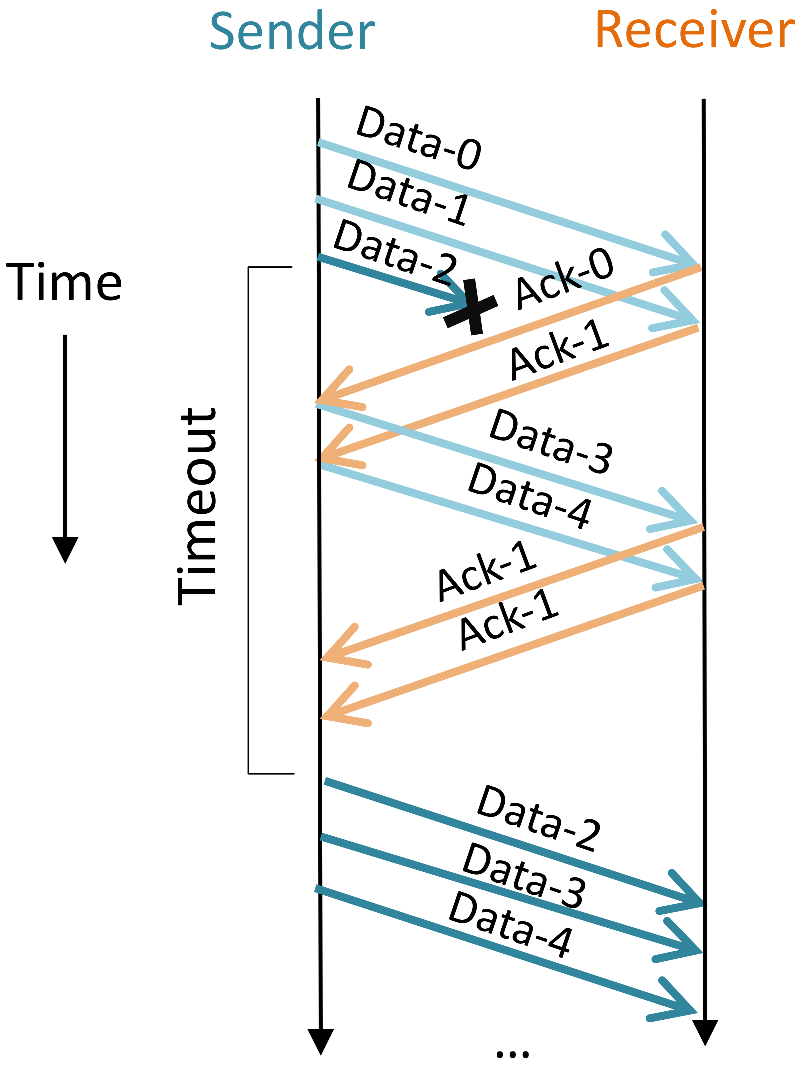
Sender
Receiver
Time
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
...

Go-Back-N

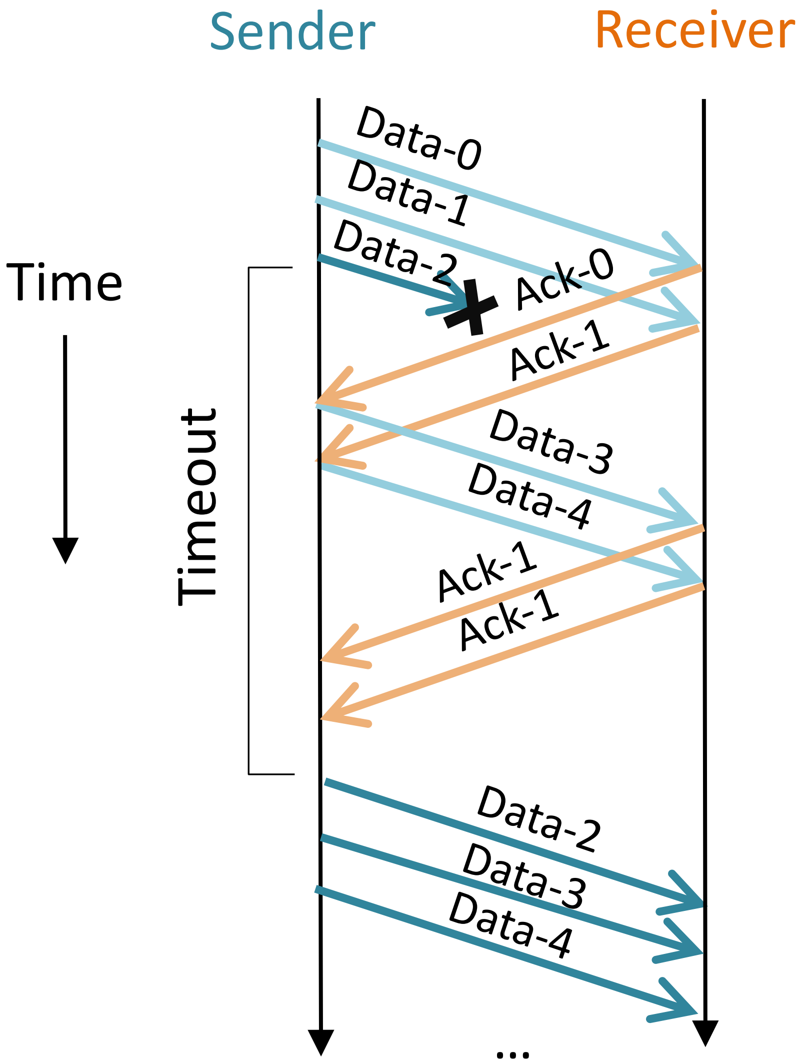


Sender
Receiver
Time
Timeout
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
...

Go-Back-N

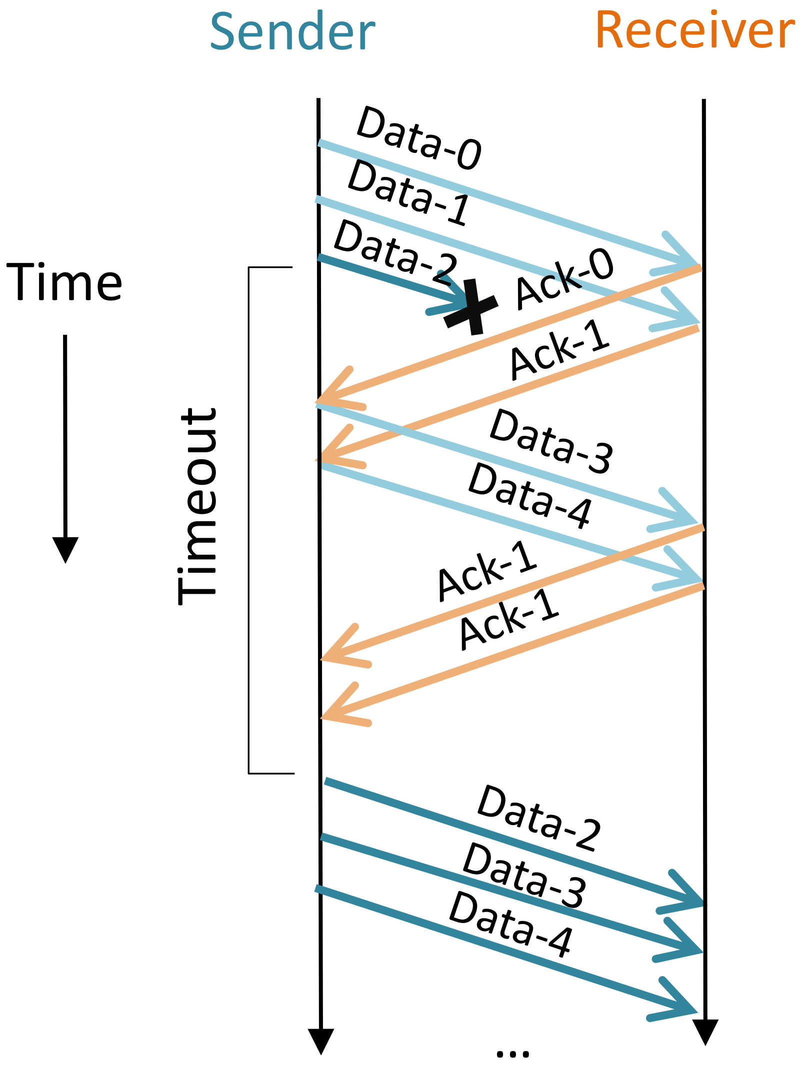

Sender
Receiver
Time
Timeout
Data-0
Data-1
Data-2
Ack-0
Ack-1
Data-3
Data-4
Ack-1
Ack-1
Data-2
Data-3
Data-4
...

Go-Back-N



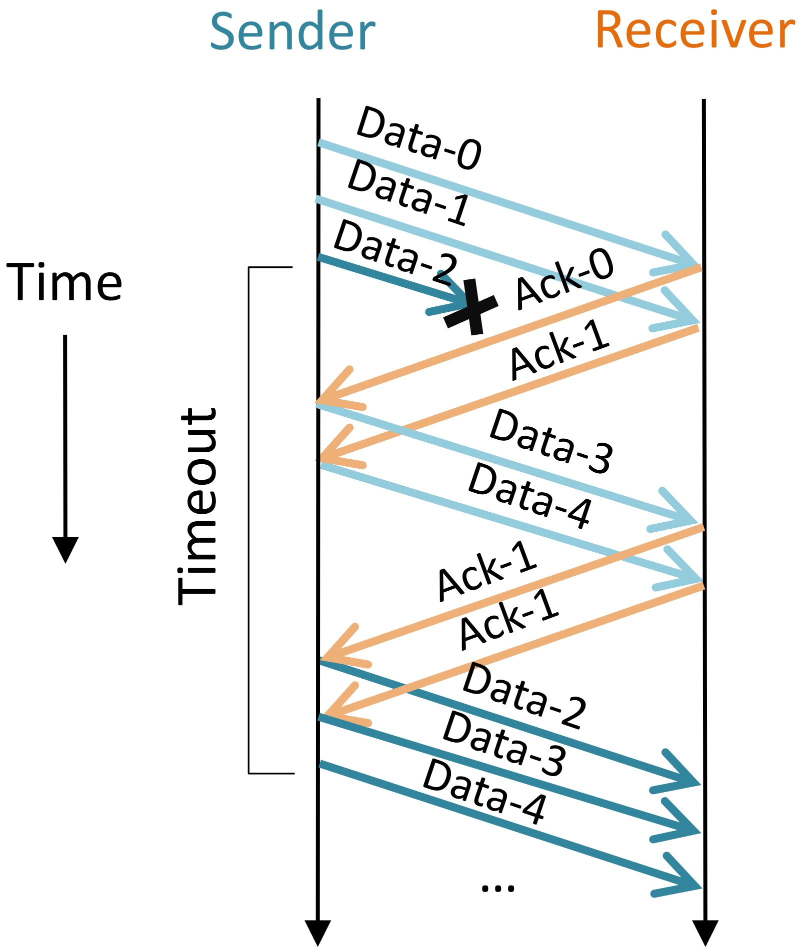
- Retransmit from point of loss
  - Segments between loss event and retransmission are ignored
  - "Go-back-N" if a timeout event occurs

Go-Back-N Performance Optimization



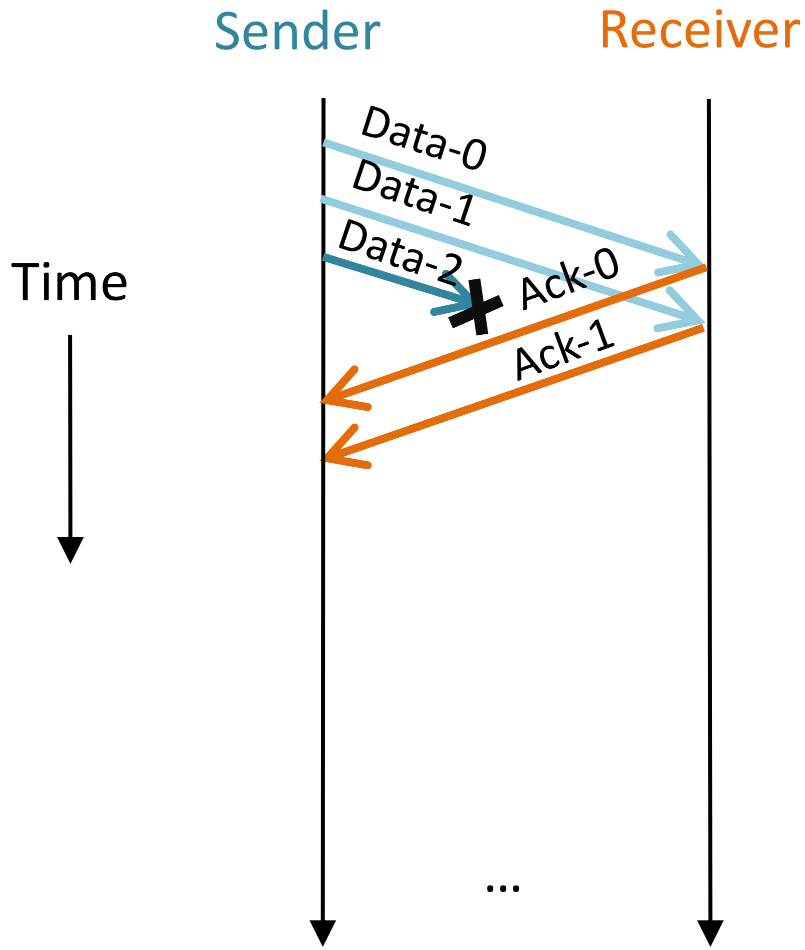
- We can optimize performance in

Go-Back-N: Performance Optimization



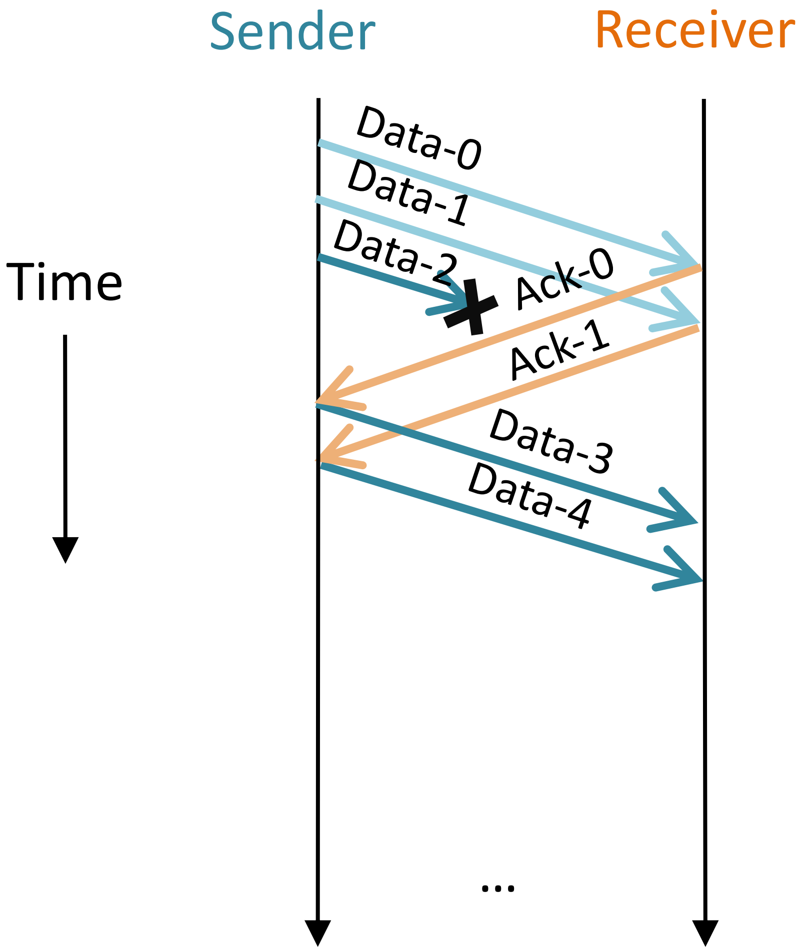
- Yes,

Selective Repeat

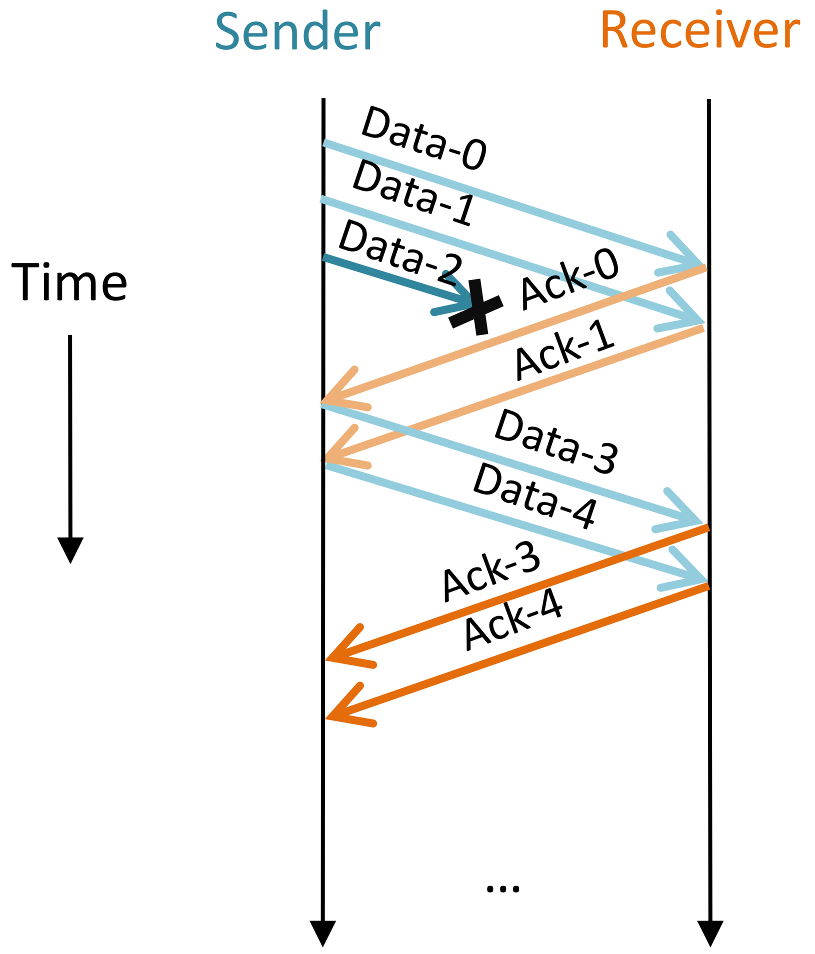


- Receiver ACKs each segment individually (not cumulative)
- Sender only resends those not ACKed

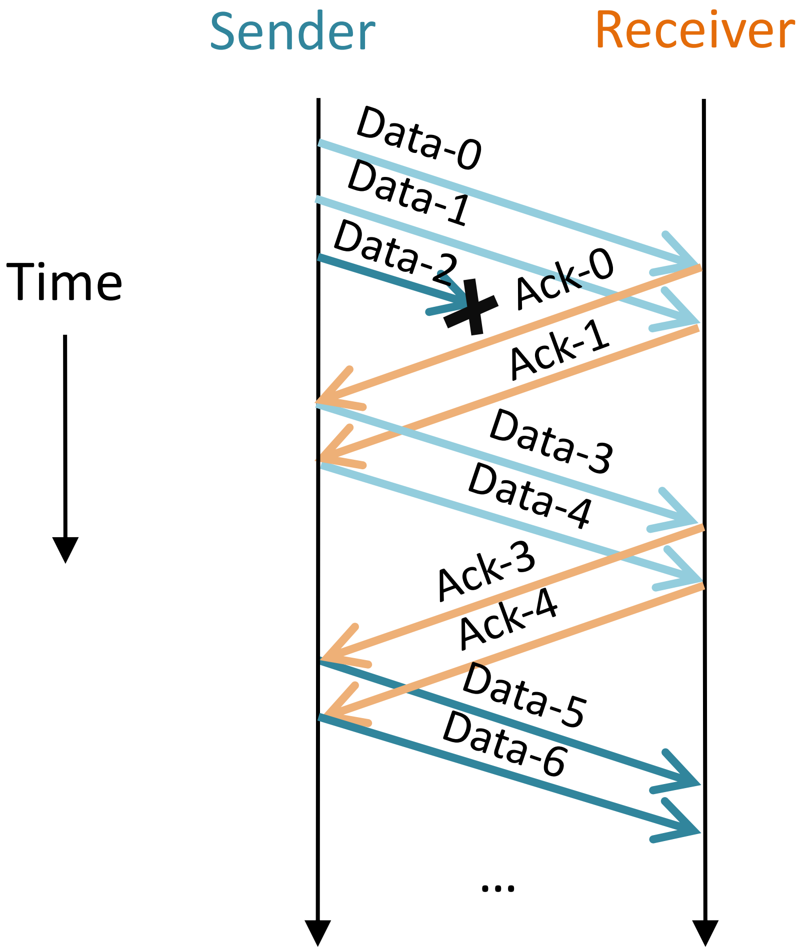
Selective Repeat



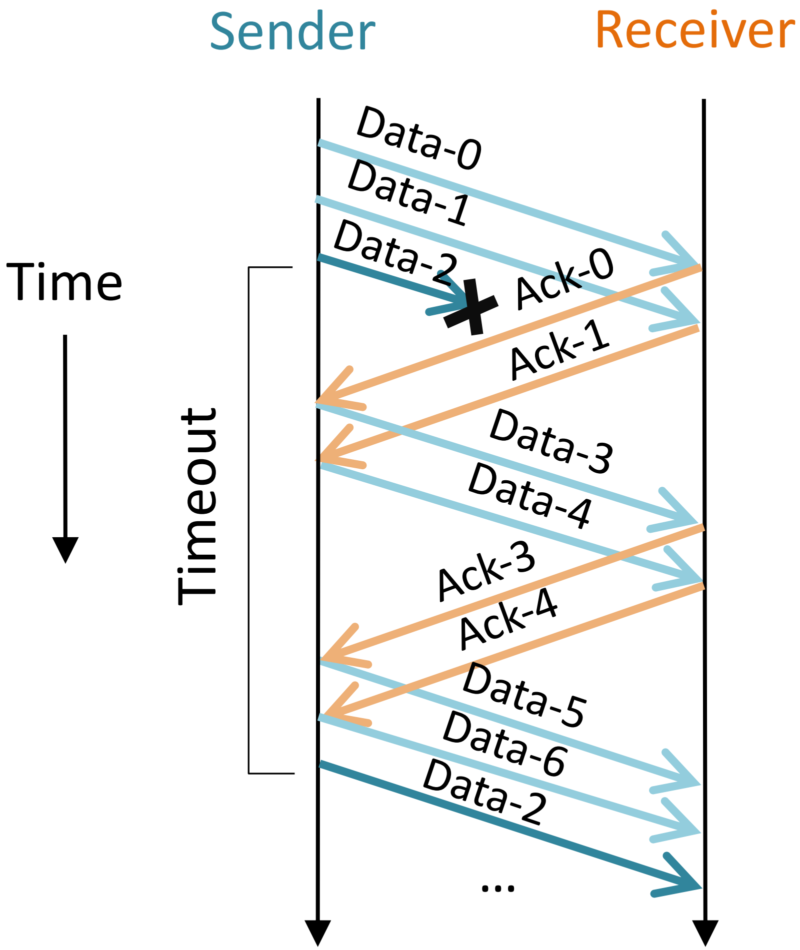
Selective Repeat



Selective Repeat



Selective Repeat



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

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?

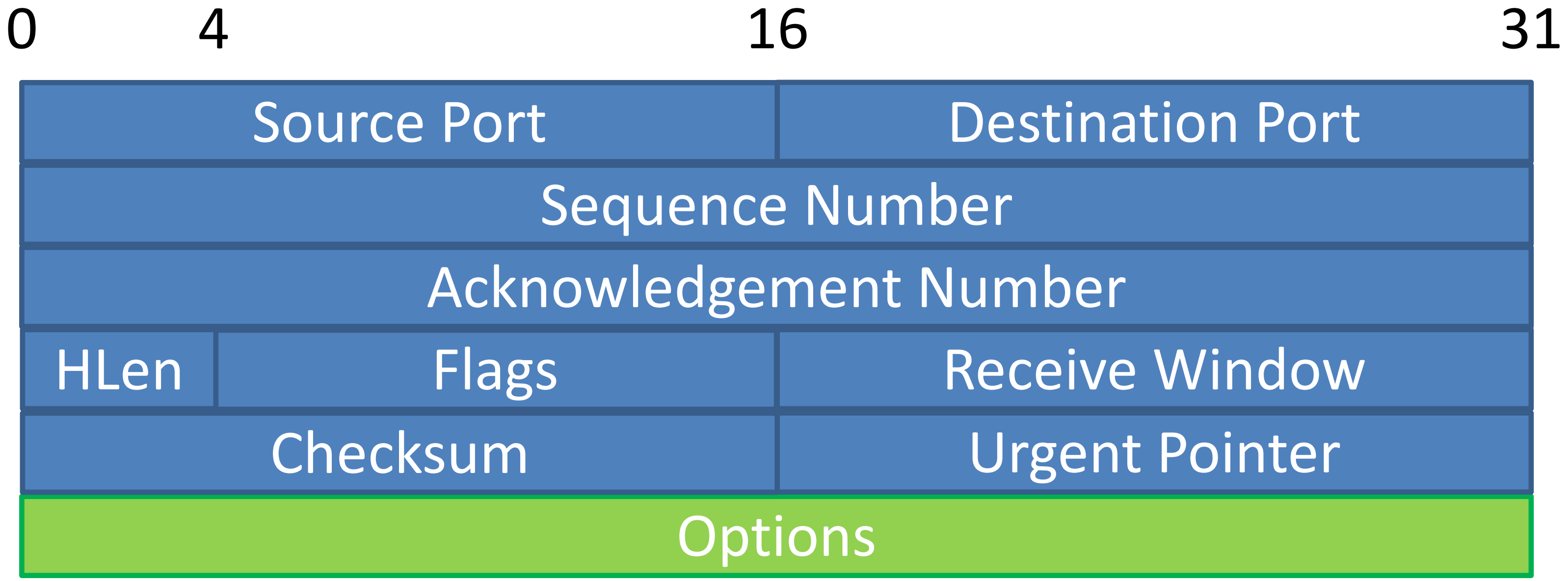
TCP Overview

- Point-to-point, full duplex
  - One pair of hosts
  - Messages in both directions
- Reliable, in-order byte stream
  - No discrete message
- Connection-oriented
  - Handshaking (exchange of control messages) before data transmitted
- Pipelined
  - Many segments in flight
- Flow control
  - Don’t send too fast for the receiver
- Congestion control
  - Don’t send too fast for the network

Transmission Control Protocol

Reliable, in-order, bi-directional byte streams

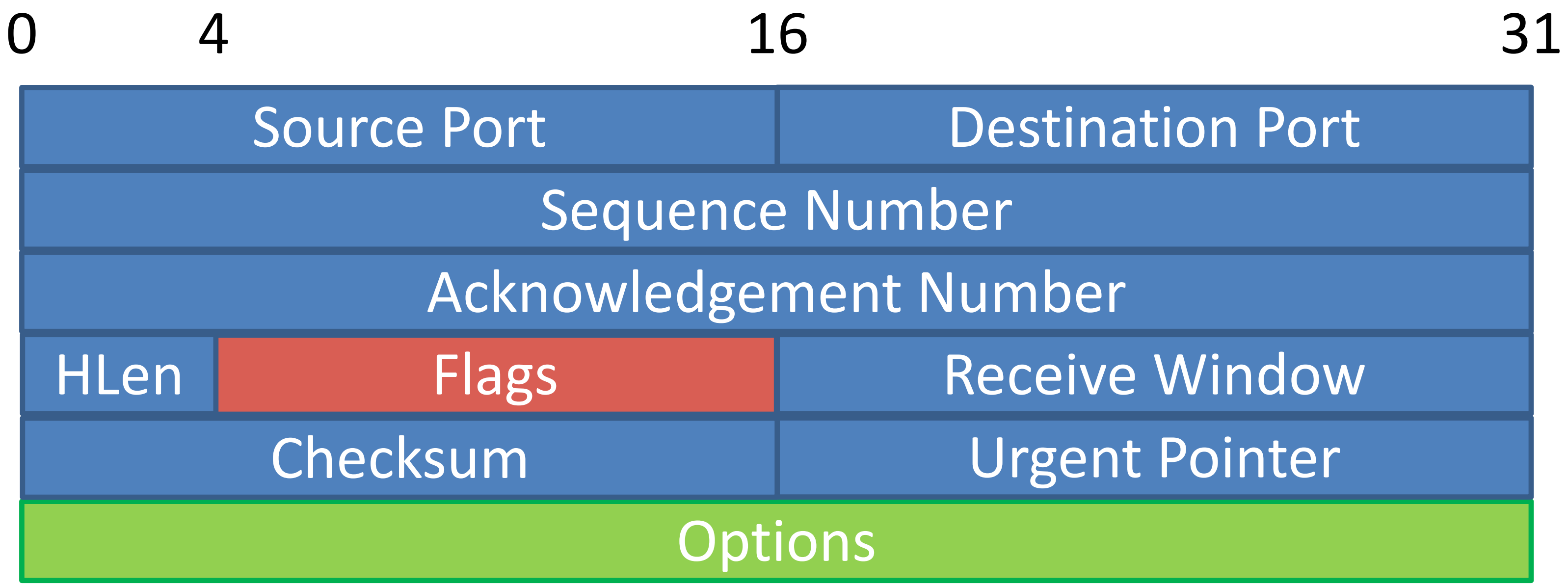
- Port numbers for demultiplexing
- Flow control
- Congestion control, approximate fairness



Transmission Control Protocol

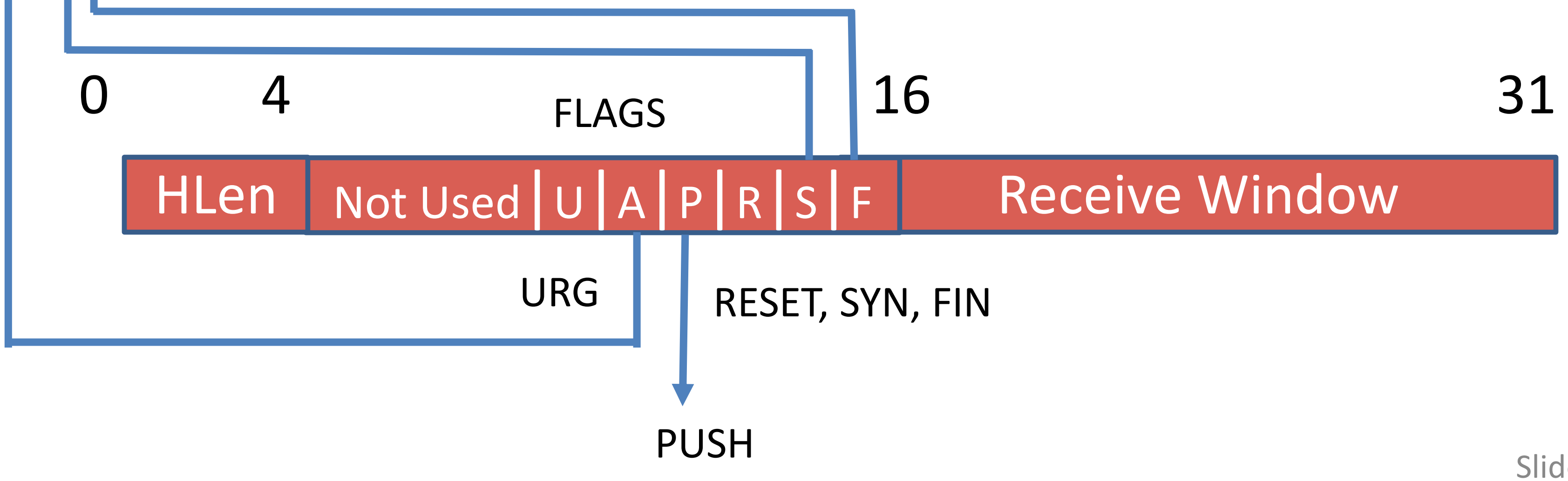
Reliable, in-order, bi-directional byte streams

- Port numbers for demultiplexing
- Flow control
- Congestion control, approximate fairness



Transmission Control Protocol

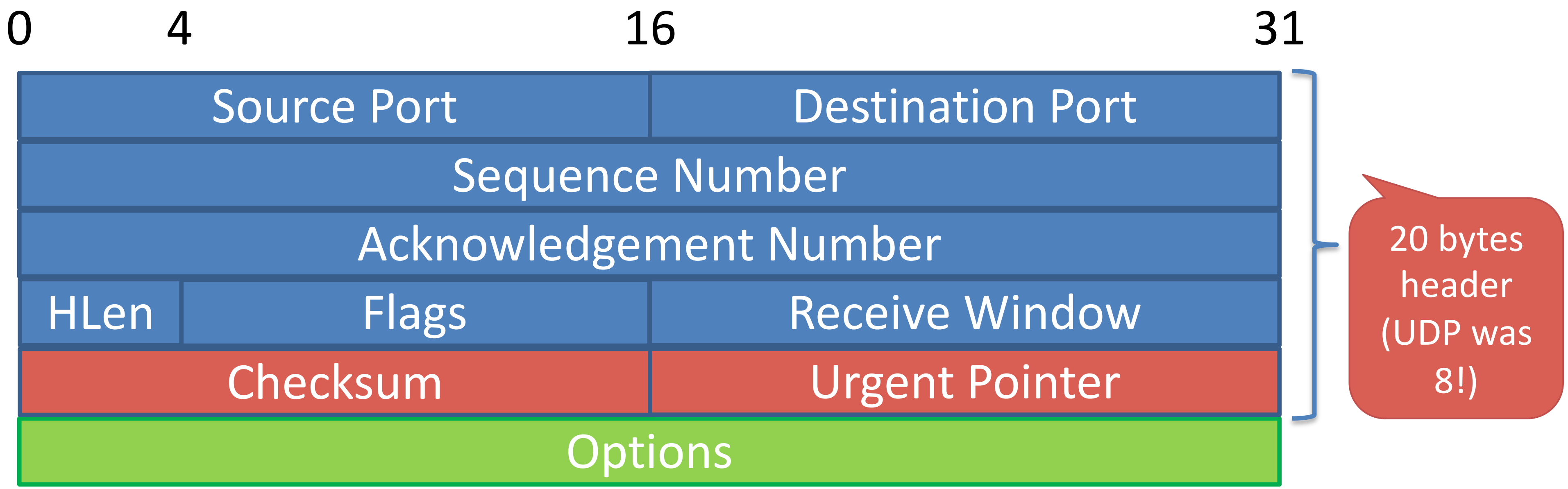
- Important TCP flags (1 bit each)
  - ACK – acknowledge received data (ACK valid or not)
  - SYN – synchronization, used for connection setup
  - FIN – finish, used to tear down connection



Transmission Control Protocol

Reliable, in-order, bi-directional byte streams

- Checksum: similar to TCP
- Urgent Pointer: Goes along with URG (U) flag in flags field
- Options: extensibility to TCP/not required



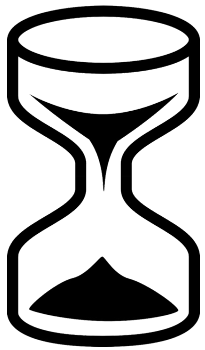
Practical Reliability Questions

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

A connection…

1. Requires stored state at two hosts.
2. Requires stored state within the network.
3. Establishes a path between two hosts.

A. 1
B. 1 & 3
C. 1, 2 & 3
D. 2
E. 2 & 3



A connection…

1. Requires stored state at two hosts.
2. Requires stored state within the network.
3. Establishes a path between two hosts.

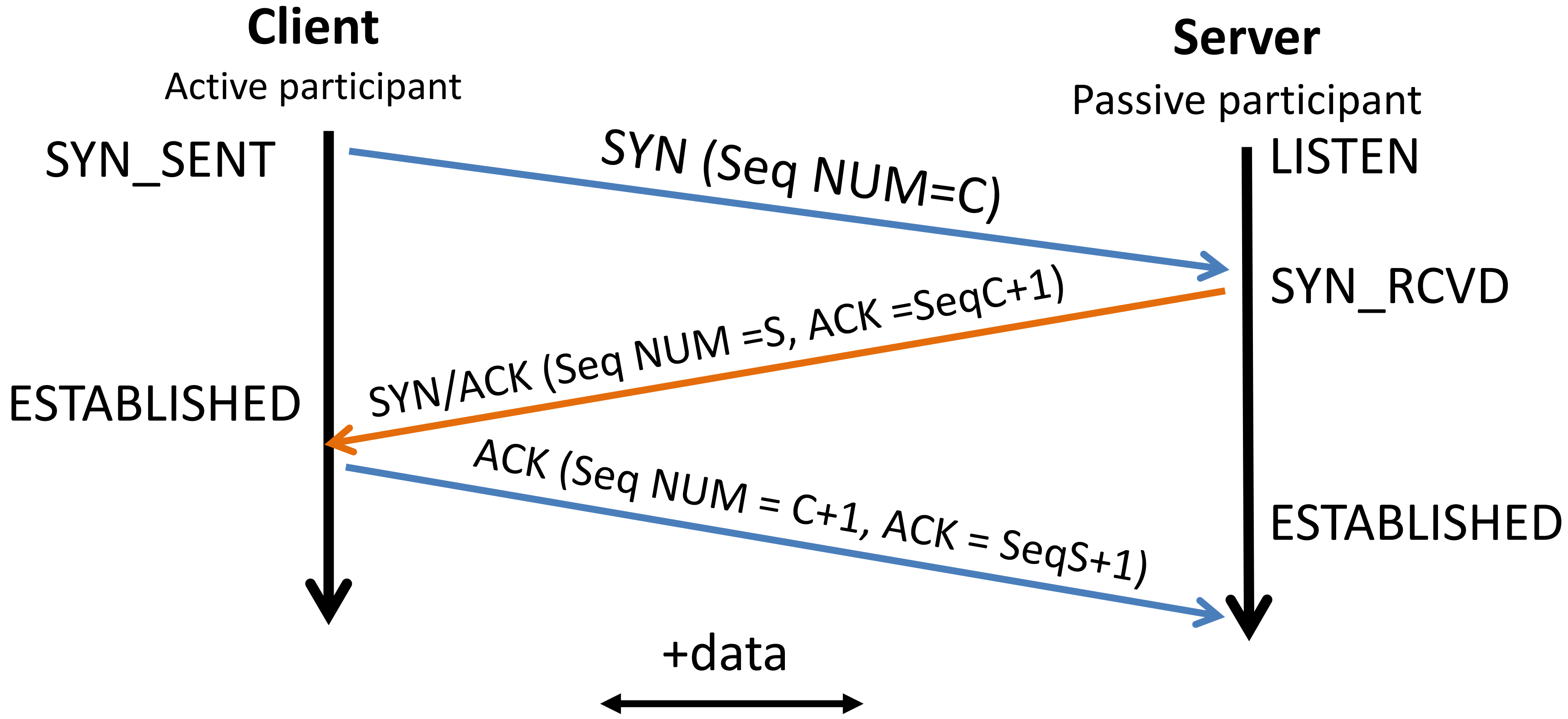
A. 1
B. 1 & 3
C. 1, 2 & 3
D. 2
E. 2 & 3

Connections

- In TCP, hosts must establish a connection prior to communicating.

- Exchange initial protocol state.
  - sequence #s to use.
  - maximum segment size (MSS)
  - Initial window sizes, etc. (several parameters)

Three Way Handshake

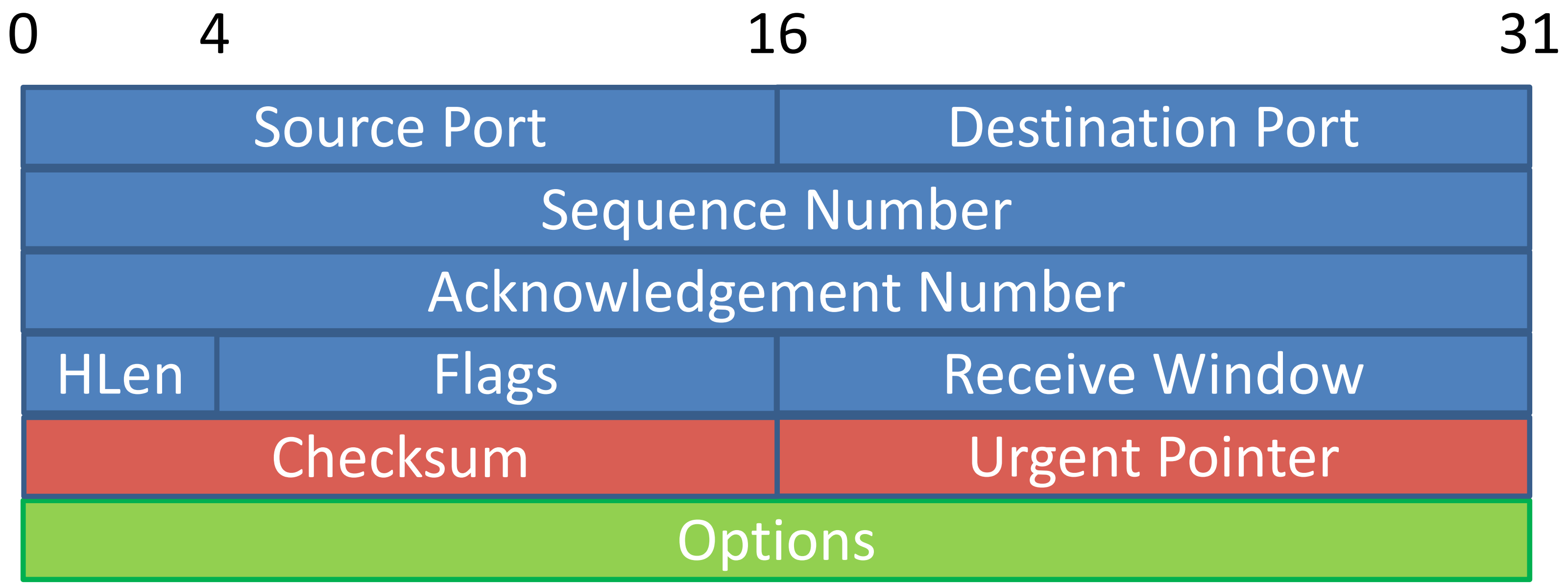


- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side's starting sequence number

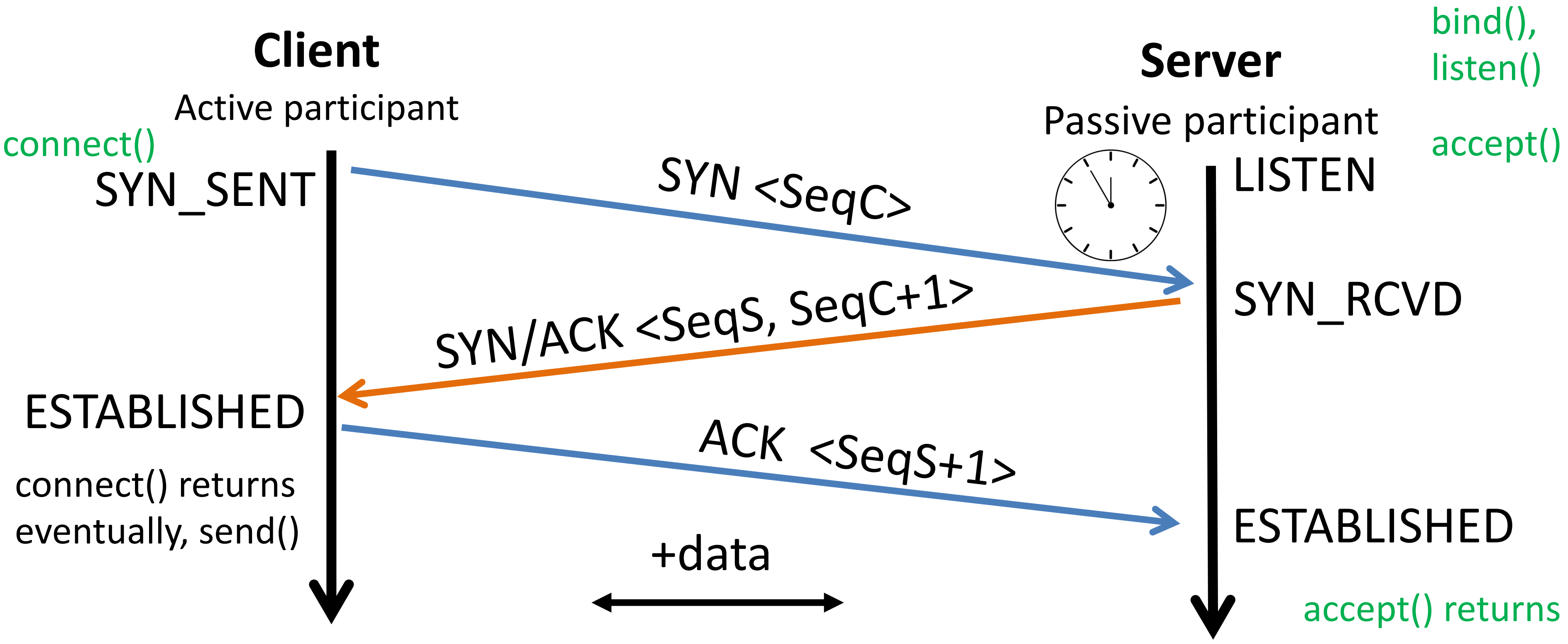
Transmission Control Protocol

Reliable, in-order, bi-directional byte streams

- Checksum: similar to TCP
- Urgent Pointer: Goes along with URG (U) flag in flags field
- Options: extensibility to TCP/not required

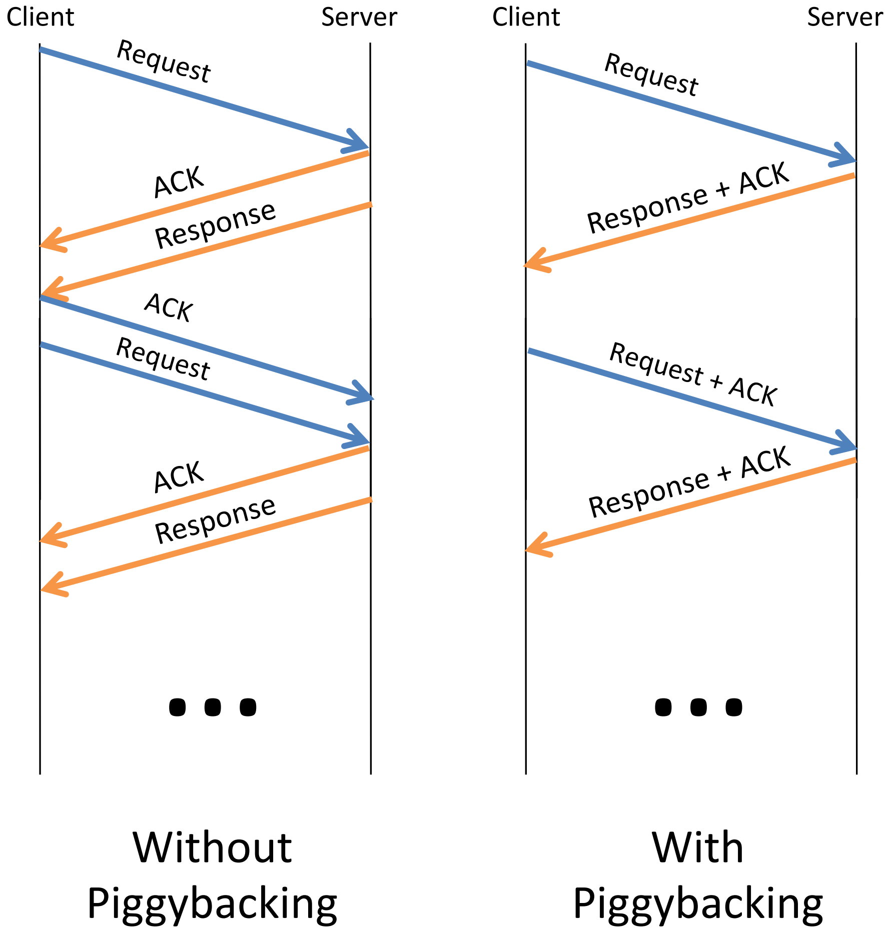


Three Way Handshake



Both sides agree on connection.

Piggybacking



Initiator/Receiver

- Assumed distinct “sender” and “receiver” roles
- In reality, usually both sides of a connection send some data
- request/response is a common pattern

Initiator
Active participant

Receiver
Passive participant

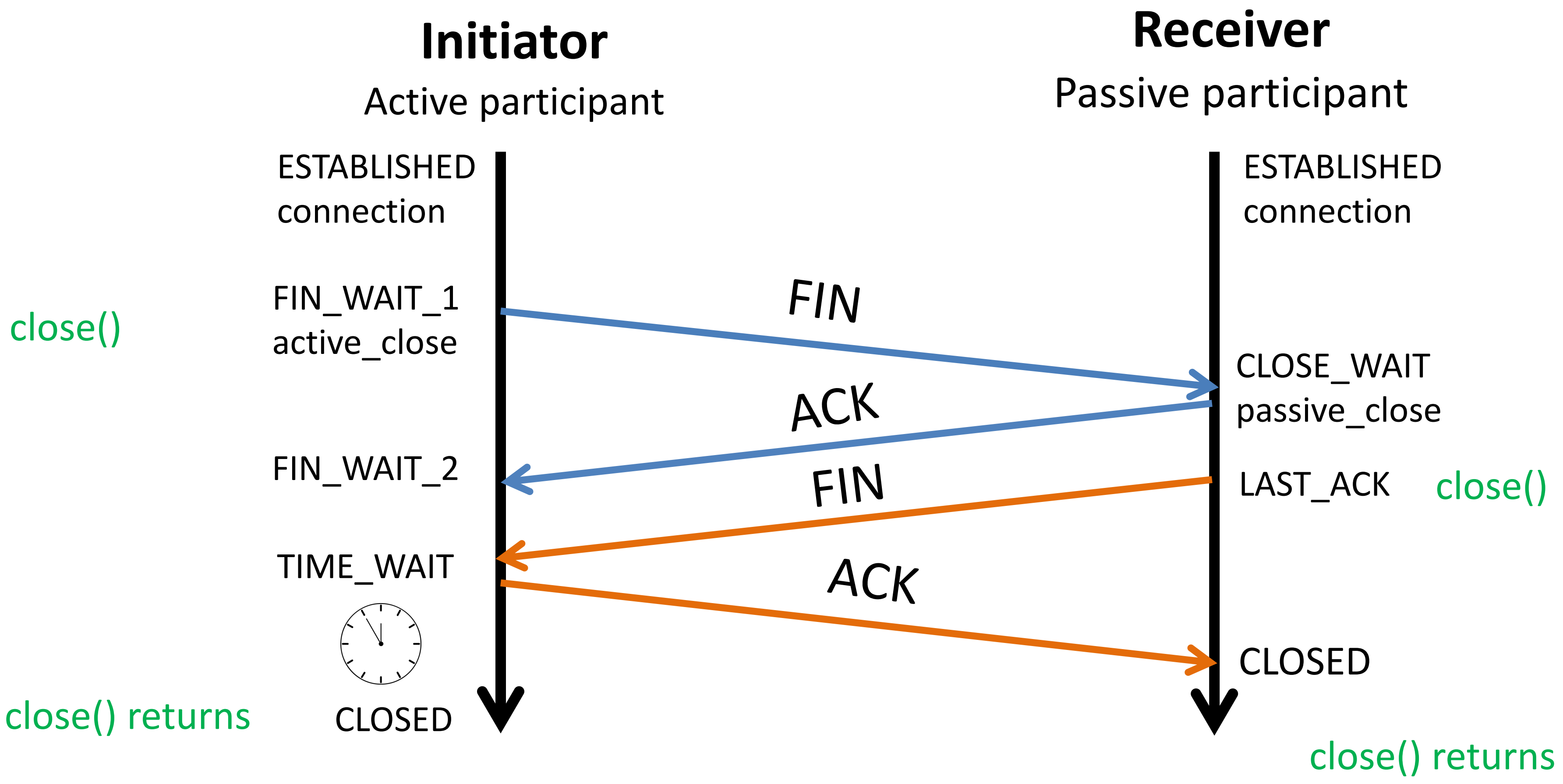
Connection Teardown

- Orderly release by sender and receiver when done
  - Delivers all pending data and “hangs up”

- Cleans up state in sender and receiver

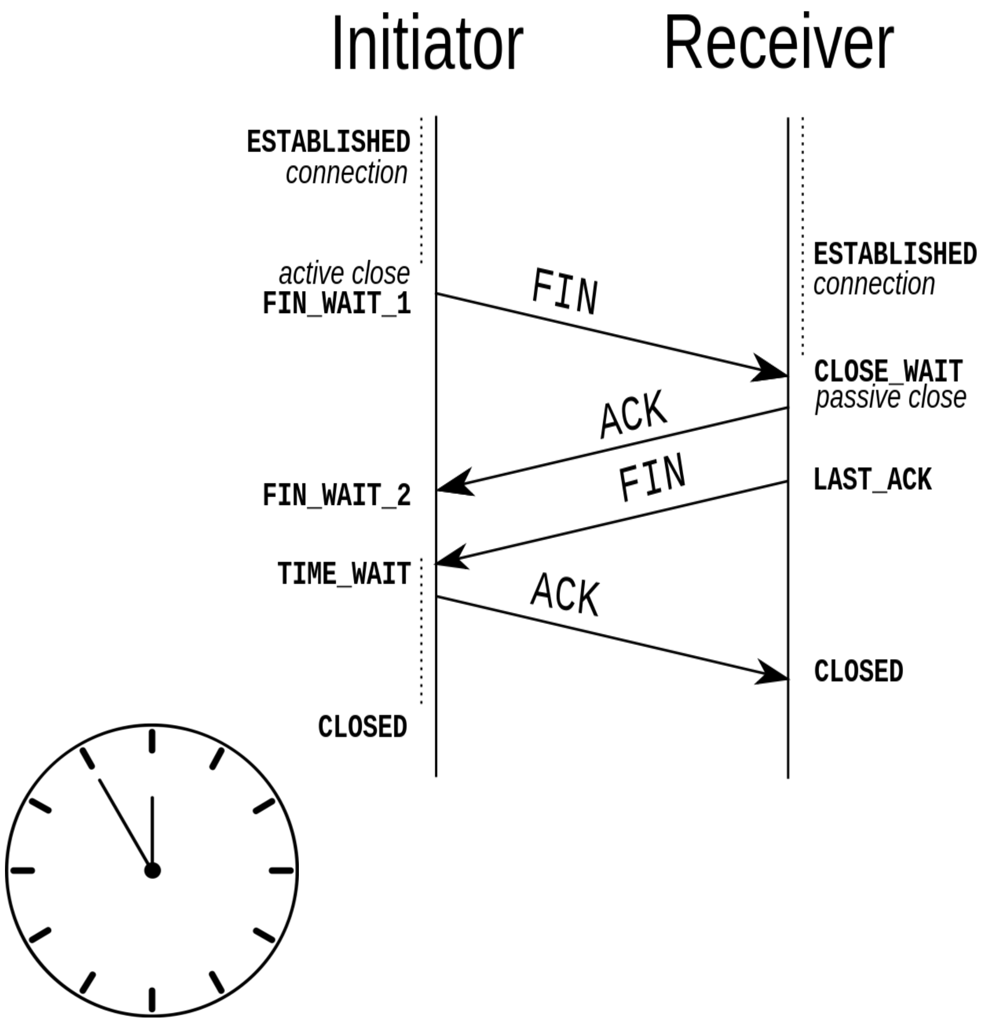
- Each side may terminate independently

TCP Connection Teardown



Both sides agree on closing the connection.

Why does one side need to wait before transitioning to CLOSED state?



A. Random protocol artifact there is no reason for it to wait.

B. There is a reason for it to wait the reason is …

The TIME_WAIT State

- We wait 2*MSL (maximum segment lifetime) before completing the close. The MSL is arbitrary (usually 60 sec)

- ACK might have been lost and so FIN will be resent
  - Could interfere with a subsequent connection

- This is why we used SO_REUSEADDR socket option in lab 2
  - Says to skip this waiting step and immediately abort the connection

Practical Reliability Questions

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

How should we choose the initial sequence number?

A. Start from zero

B. Start from one

C. Start from a random number

D. Start from some other value (such as…?)

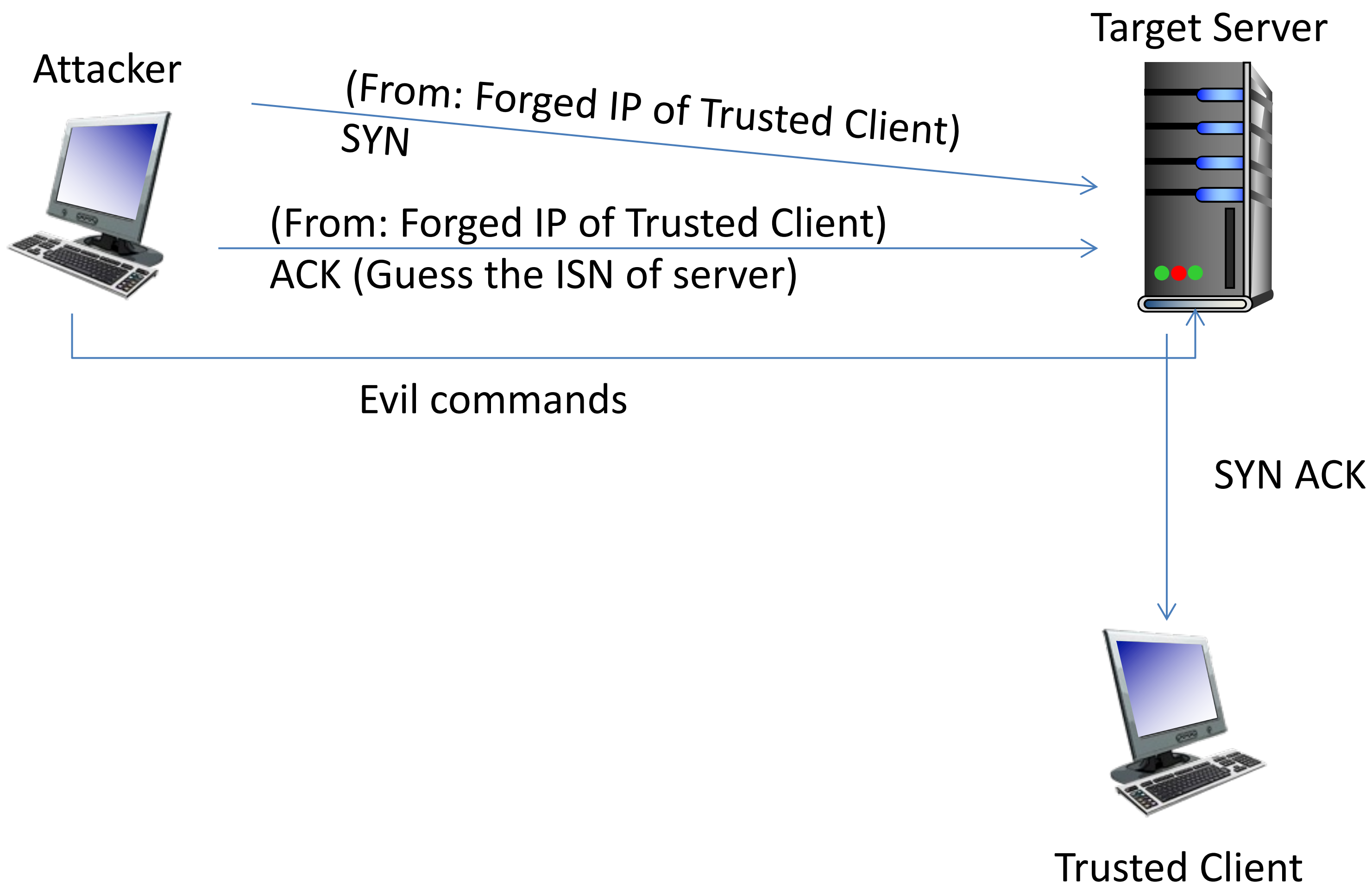
What can go wrong with sequence numbers?
-How they're chosen?
-In the course of using them?

Sequencing

- Initial sequence numbers (ISN) chosen at random
  - Does not start at 0 or 1 (anymore).
  - Helps to prevent against forgery attacks.

- TCP sequences bytes rather than segments
  - Example: if we're sending 1500-byte segments
    - Randomly choose ISN (suppose we picked 1150)
    - First segment (sized 1500) would use number 1150
    - Next would use 2650

Sequence Prediction Attack (1996)



Practical Reliability Questions

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

Timeouts

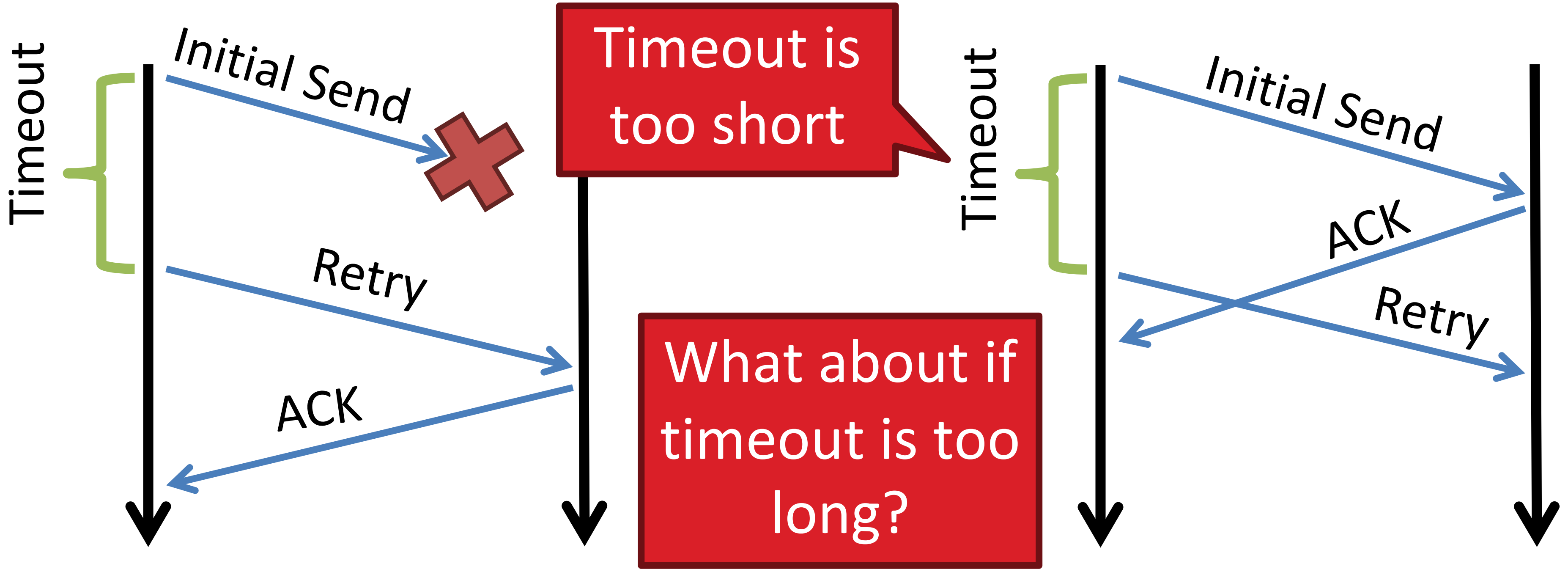
- How long should we wait before timing out and retransmitting a segment?

- Too short: needless retransmissions
- Too long: slow reaction to losses

- Should be (a little bit) longer than the RTT

Retransmission Timeouts

- Problem: time-out is linked to round trip time



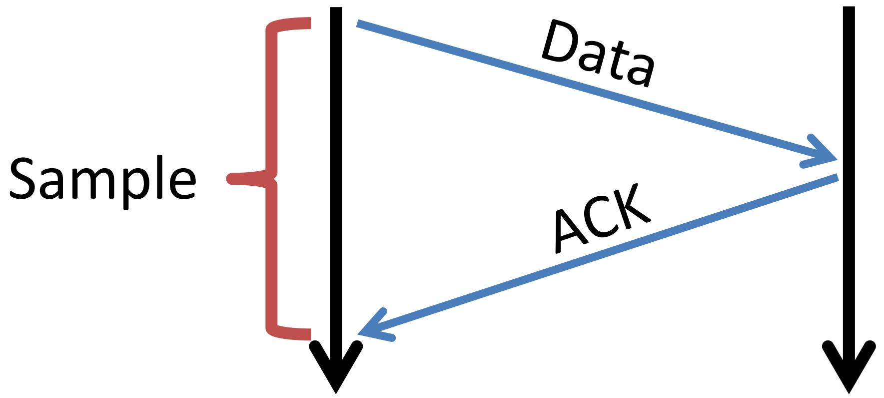
Estimating RTT

- Problem: RTT changes over time
  - Routers buffer packets in queues
  - Queue lengths vary
  - Receiver may have varying load

- Sender takes measurements
  - Use statistics to decide future timeouts for sends
  - Estimate RTT and variance

- Apply “smoothing” to account for changes

Round Trip Time Estimation:
Exponentially Weighted Moving Average (EWMA)



<u>EstimatedRTT = (1 – a) * EstimatedRTT + a * SampleRTT</u>

– a is usually 1/8.

In words current estimate is a blend of:

- 7/8 of the previous estimate
- 1/8 of the new sample.

<u>DevRTT = (1 – B) * DevRTT + B * | SampleRTT – EstimatedRTT |</u>

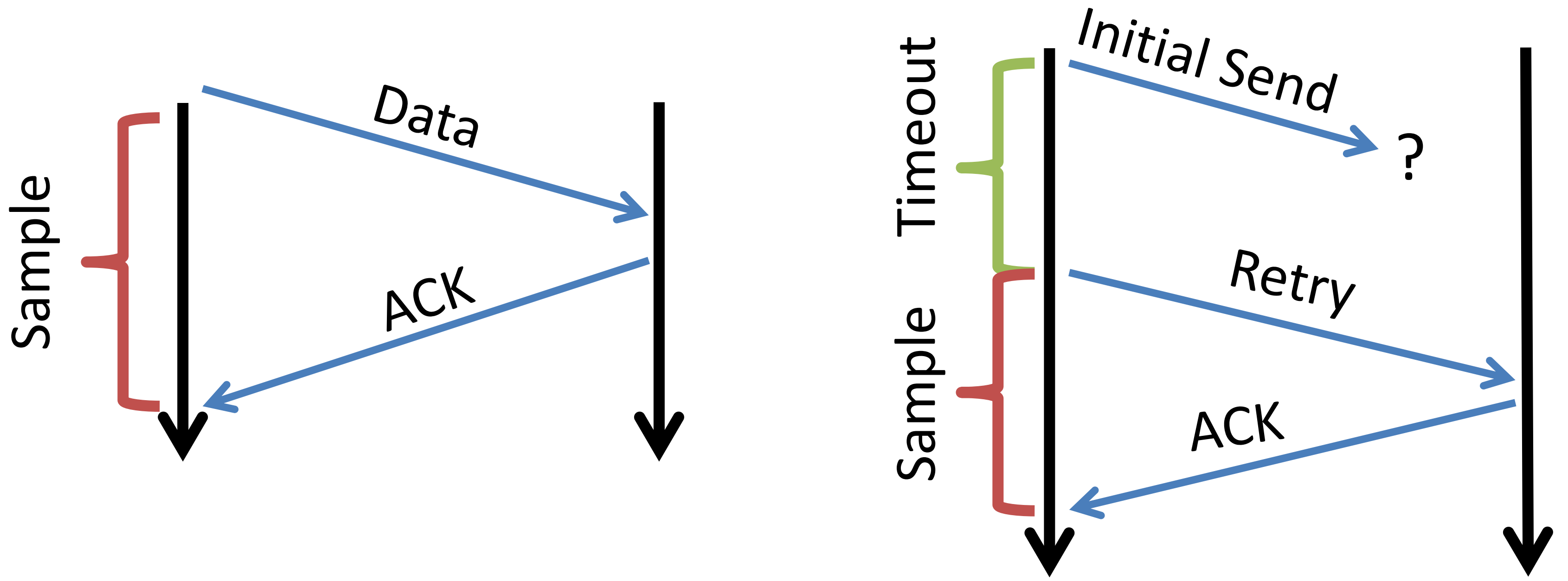
- B is usually 1/4

Estimating RTT

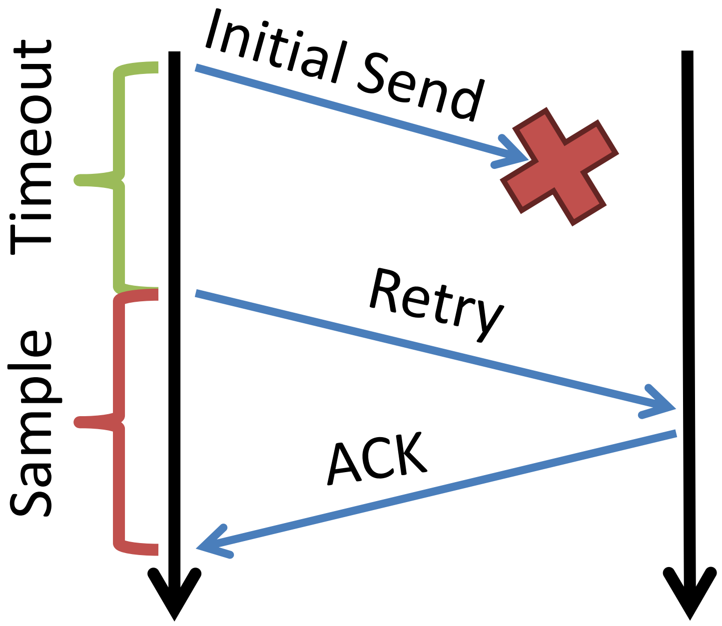
- For each segment that did not require a retransmit (ACK heard without a timeout)
  - Consider the time between segment sent and ACK received to be a sample of the current RTT
  - Use that, along with previous history, to update the current RTT estimate

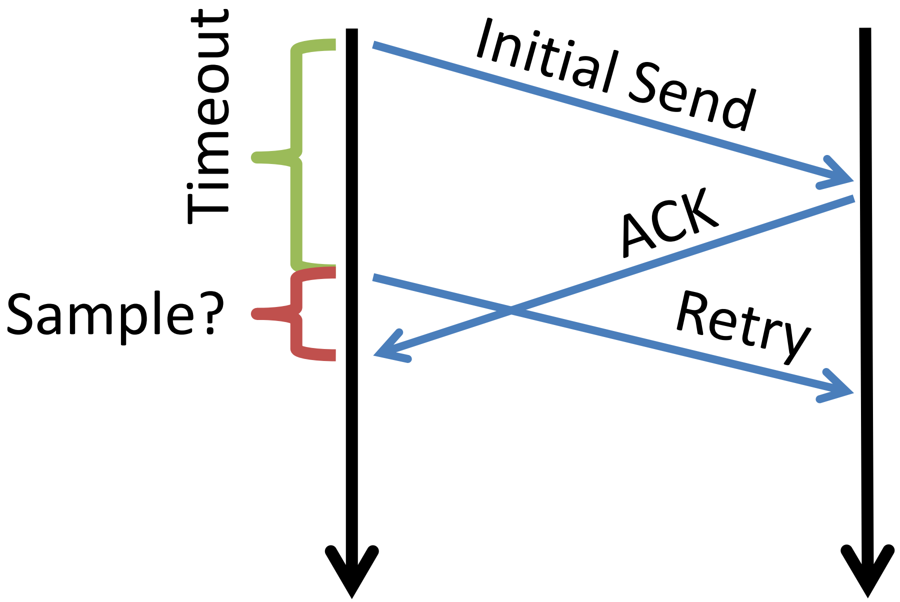
- Exponentially Weighted Moving Average (EWMA)

Round Trip Time Estimation:
Exponentially Weighted Moving Average (EWMA)



RTT Sample Ambiguity





<u>Ignore samples for retransmitted segments</u>

EWMA

EstimatedRTT = (1 – a) * EstimatedRTT + a * SampleRTT

a is usually 1/8.

In other words, our current estimate is a blend of 7/8 of the previous estimate plus 1/8 of the new sample.

DevRTT = (1 – B) * DevRTT + B * | SampleRTT – EstimatedRTT |

B is usually 1/4

Example RTT Estimation

- Suppose EstimateRTT = 64, Dev = 8
- Latest sample: 120

New estimate = 7/8 * 64 + 1/8 * 120 = 56 + 15= 71

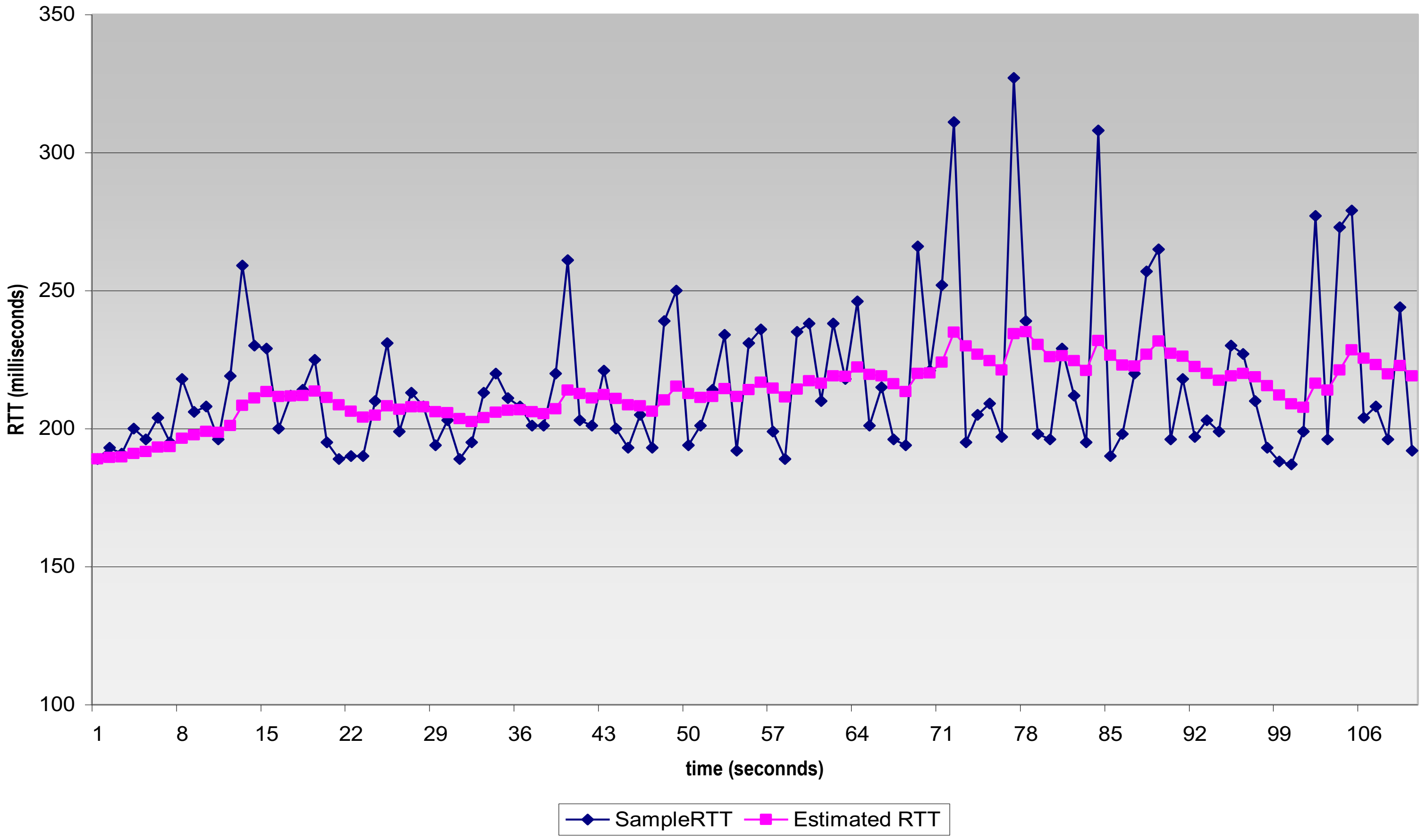
New dev = 3/4 * 8 + 1/4 * | 120 - 71 | = 6 + 12 = 18

- Another sample: 400

New estimate = 7/8 * 71 + 1/8 * 400 = 62 + 50 = 112

New dev = 3/4 * 18 + 1/4 * | 400 - 112 | = 13 + 72 = 85

Example RTT Estimation (Smoothing)



TCP Timeout Value

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"