

CS 43: Computer Networks

15: Transmission Control Protocol

October 31, 2019



Reading Quiz

Last class

- Automatic Repeat Request (ARQ) Protocols
 - Stop-and-Wait Protocol: send data, wait for response
 - ACKs/NACKs, Timeouts, Sequence Numbering
- Automatic Repeat Requests
 - Go-Back-N protocols
 - Selective Repeat

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

System parameters:

Link rate: 8 Mbps (one megabyte per second)

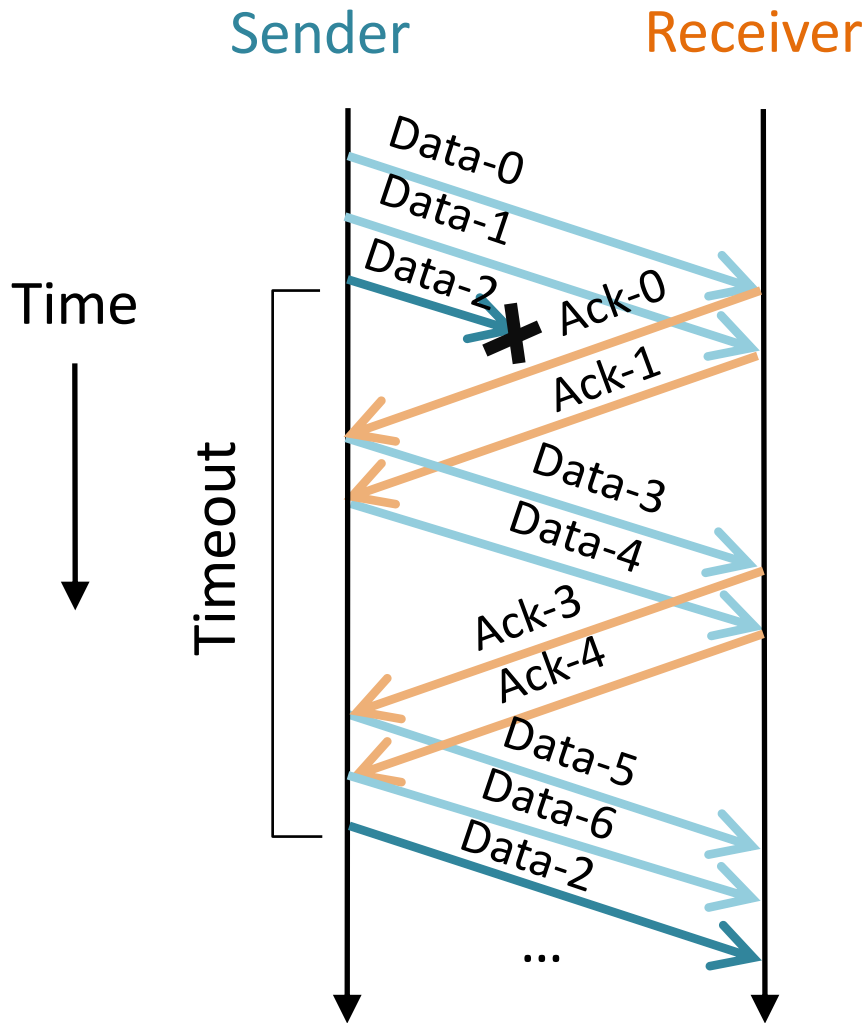
RTT: 100 milliseconds

Segment size: 1024 bytes = 1kB

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

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

Selective Repeat



- Receiver ACKs each segment individually (not cumulative)
- Sender only resends those not ACKed
- Requires extra buffering and state on the receiver

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.

Transmission Control Protocol

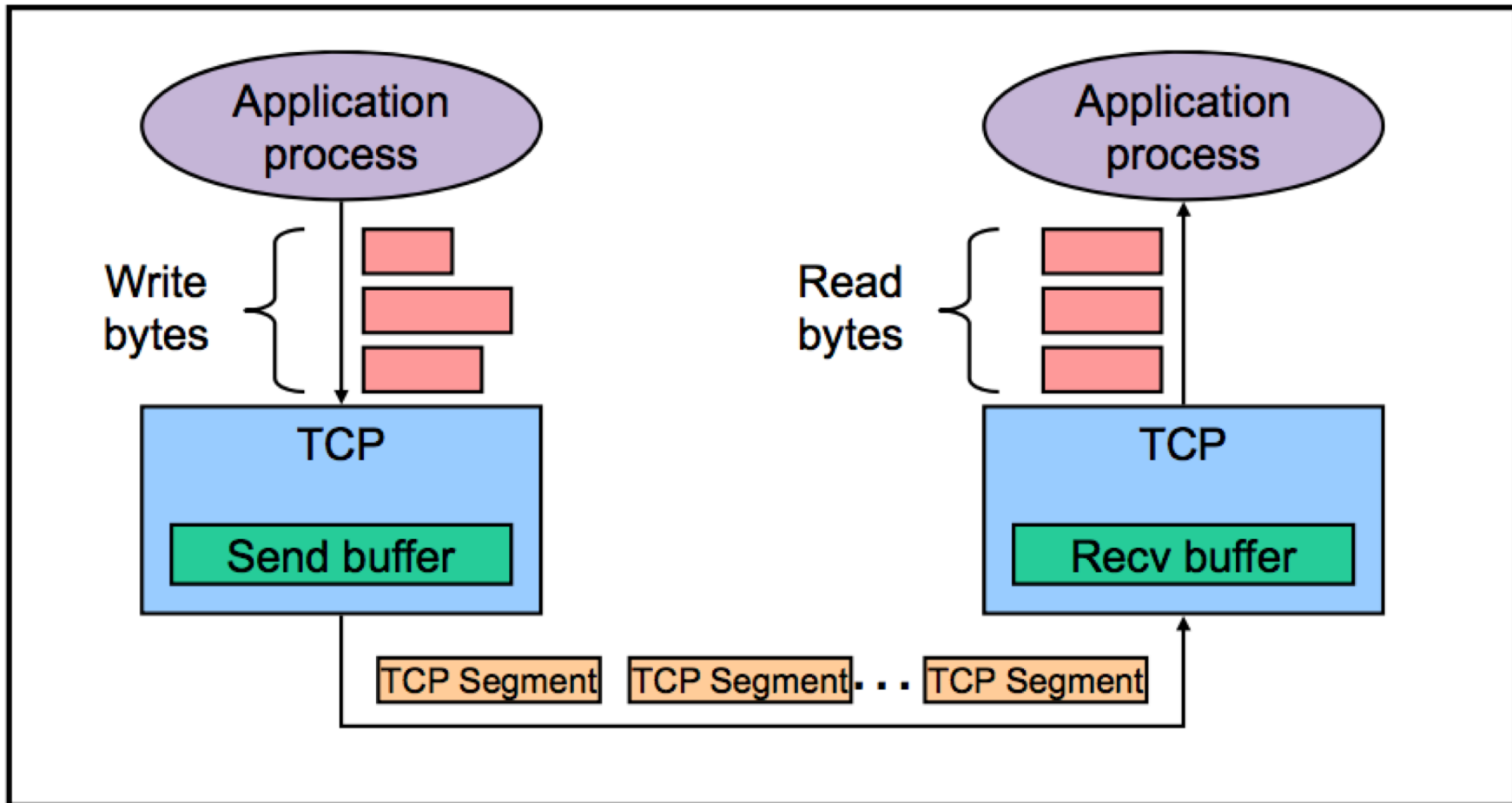
- Break message into packets (TCP segments)
- Should be delivered reliably & in-order

```
GET
http://www.google.com
HTTP/1.1
Host: www.google.com
```

...



TCP: Stream abstraction



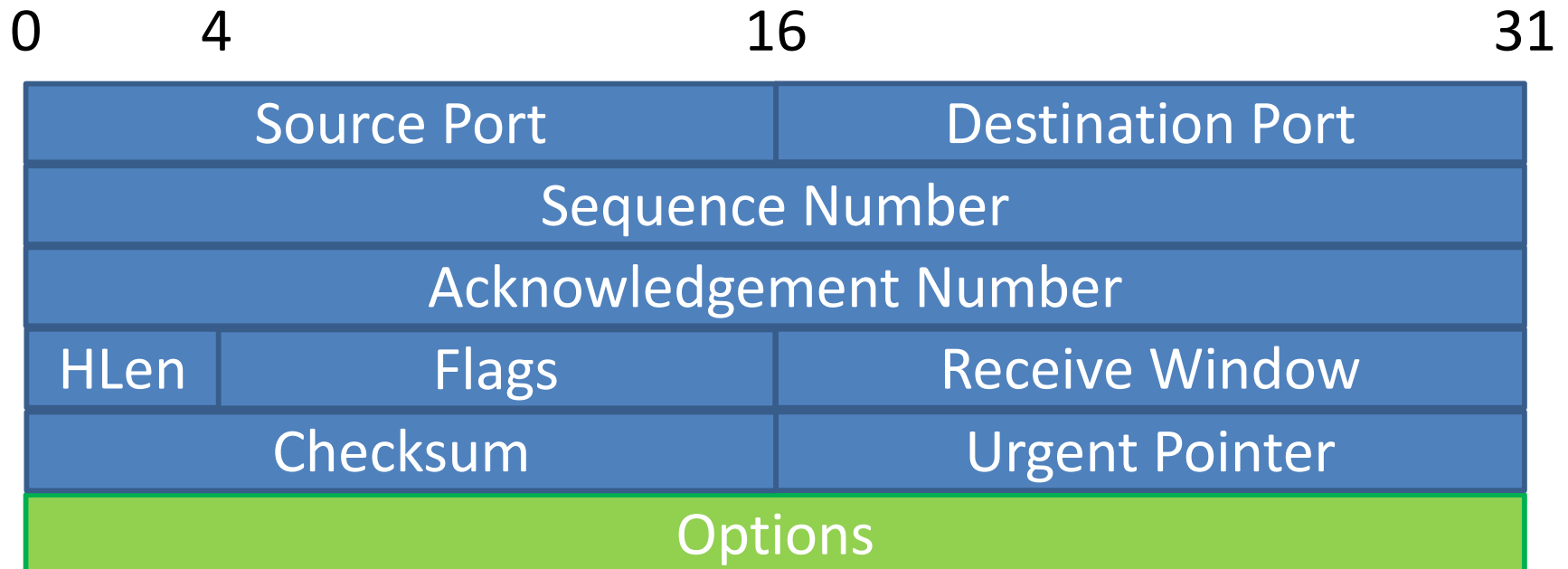
TCP Overview

- 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

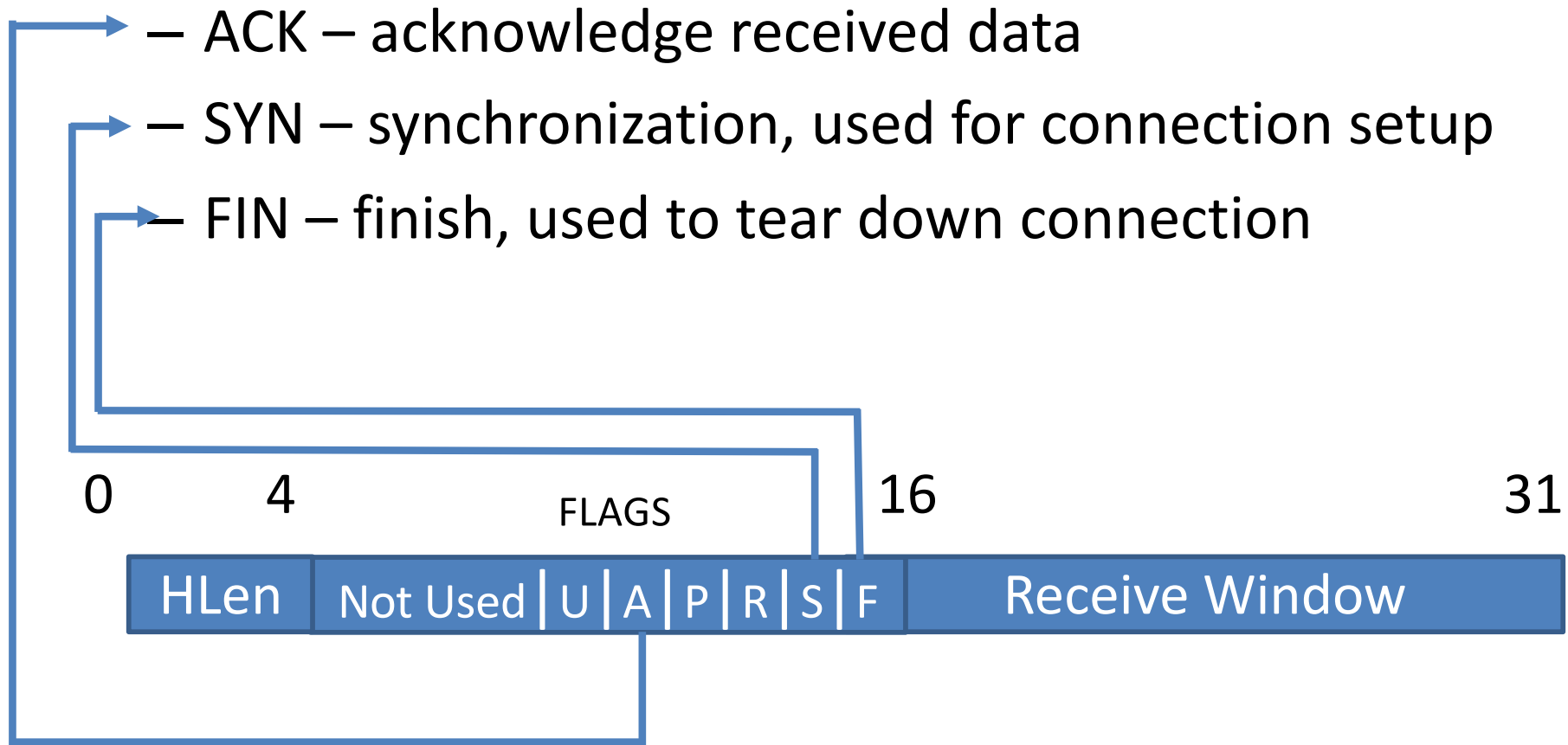
Reliable, in-order, bi-directional byte streams

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



Transmission Control Protocol

- Important TCP flags (1 bit each)
 - ACK – acknowledge received data
 - SYN – synchronization, used for connection setup
 - FIN – finish, used to tear down connection



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?

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...

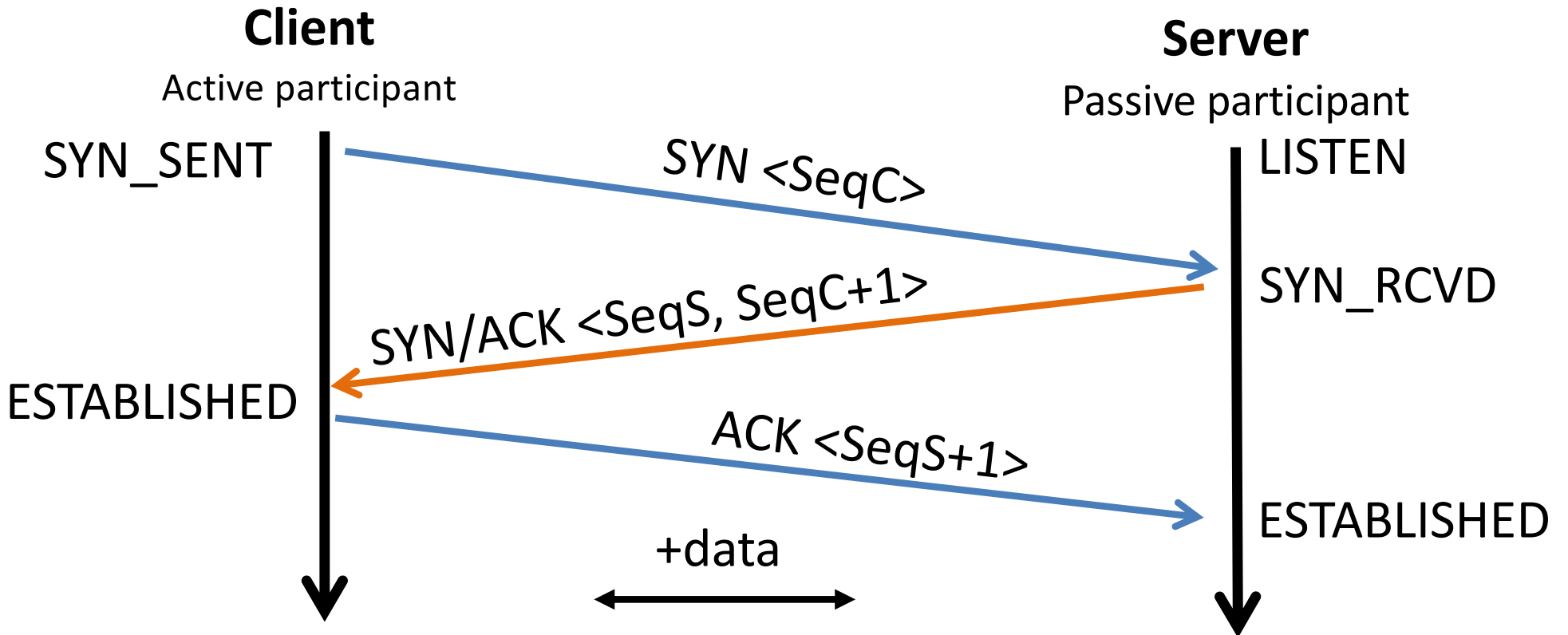
1. Requires stored state at two hosts.
2. Requires stored state within the network.
3. Establishes a path between two hosts.

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

Connections

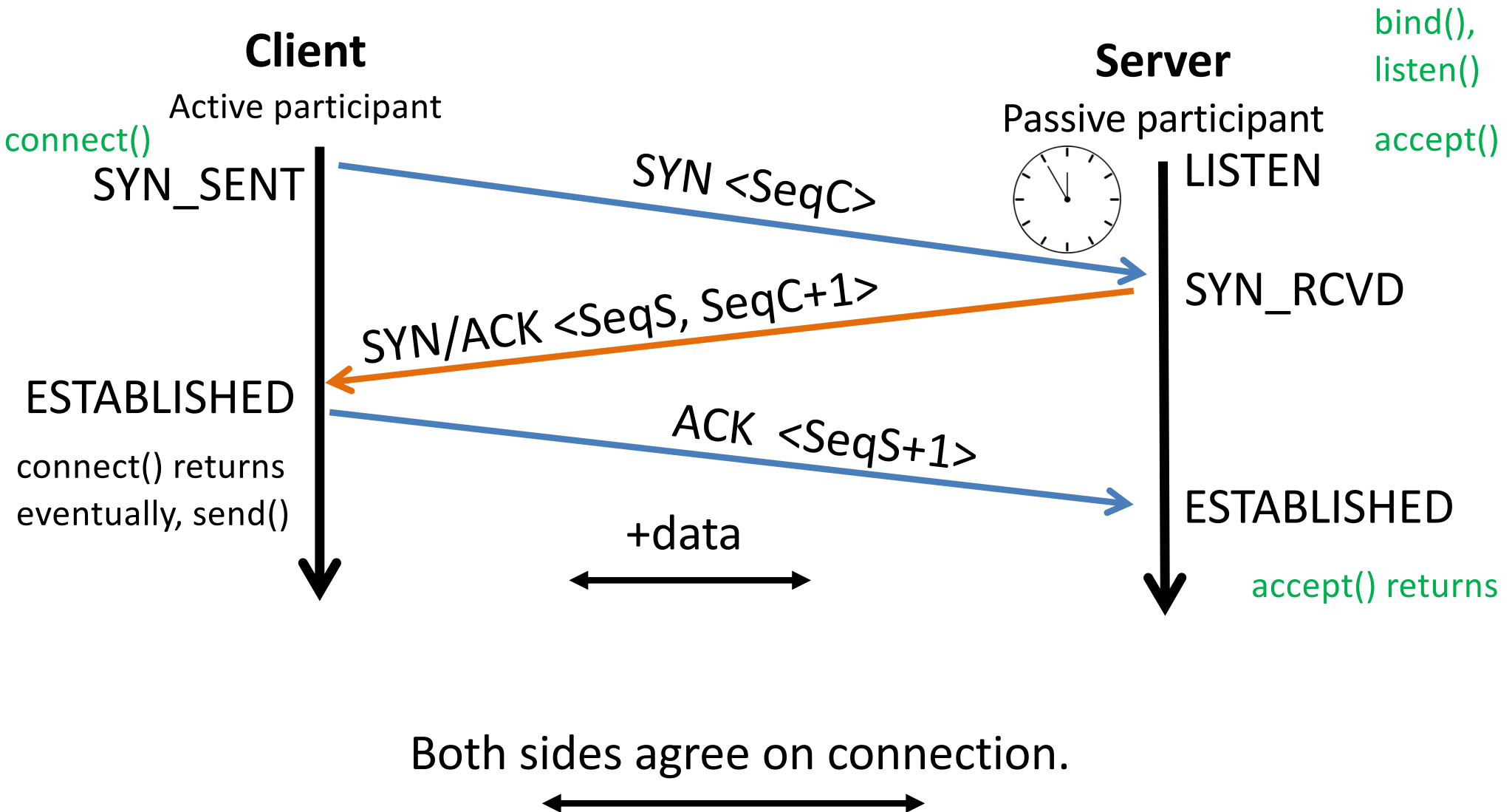
- In TCP, hosts must establish a connection prior to communicating.
- Exchange initial protocol state.
 - sequence #s to use.
 - maximum segment size
 - Initial window sizes, etc. (several parameters)

Three Way Handshake

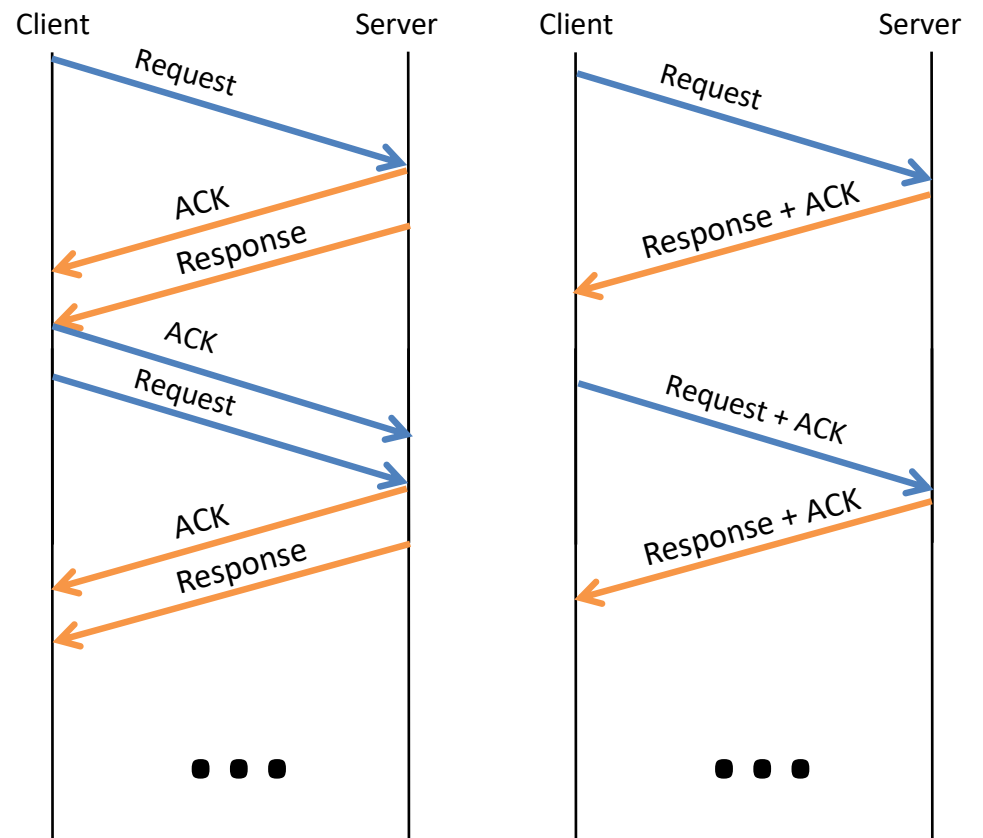


- Each side:
 - Notifies the other of starting sequence number
 - ACKs the other side's starting sequence number

Three Way Handshake



Piggybacking



Without
Piggybacking

With
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

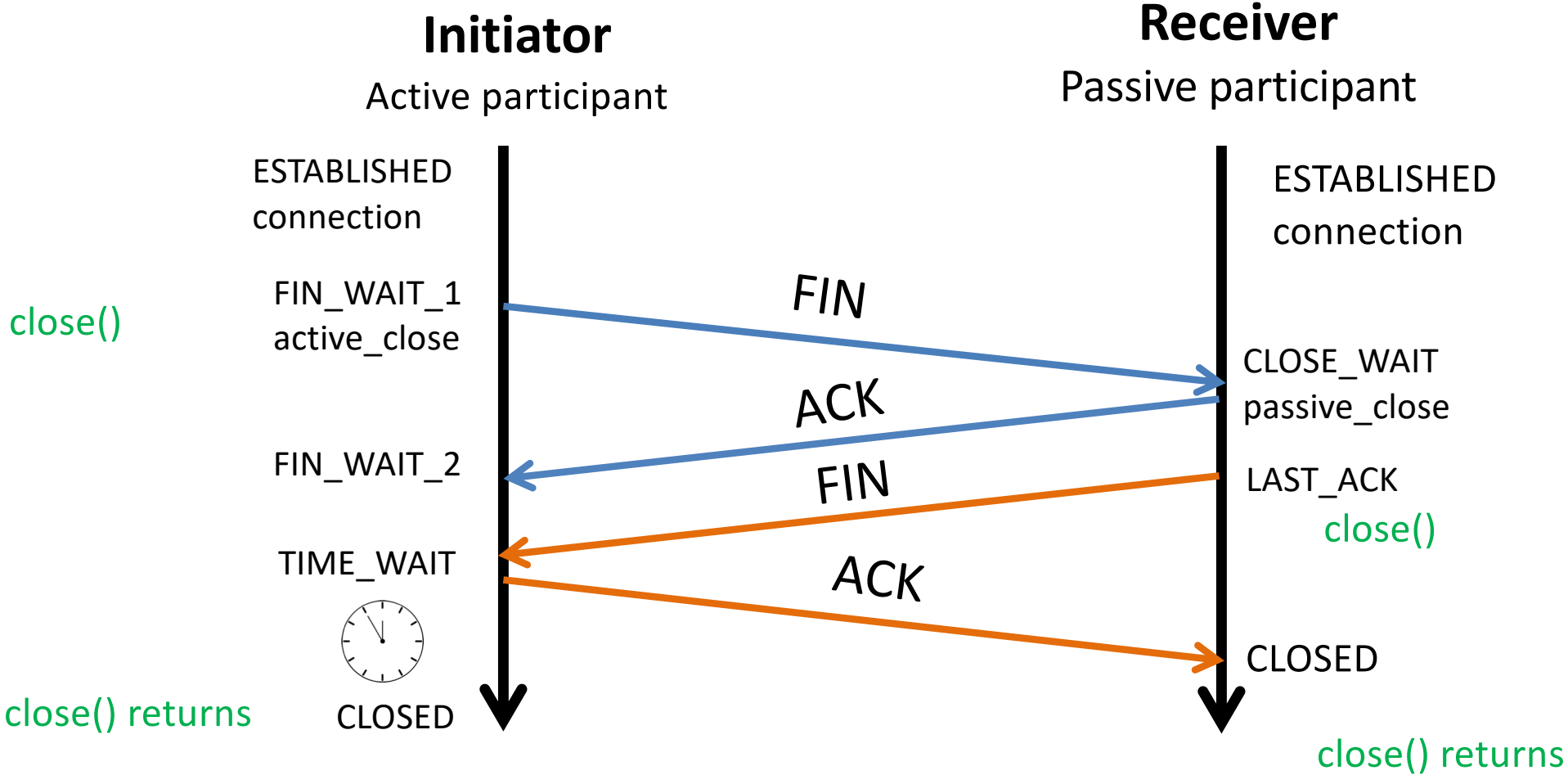
Receiver

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

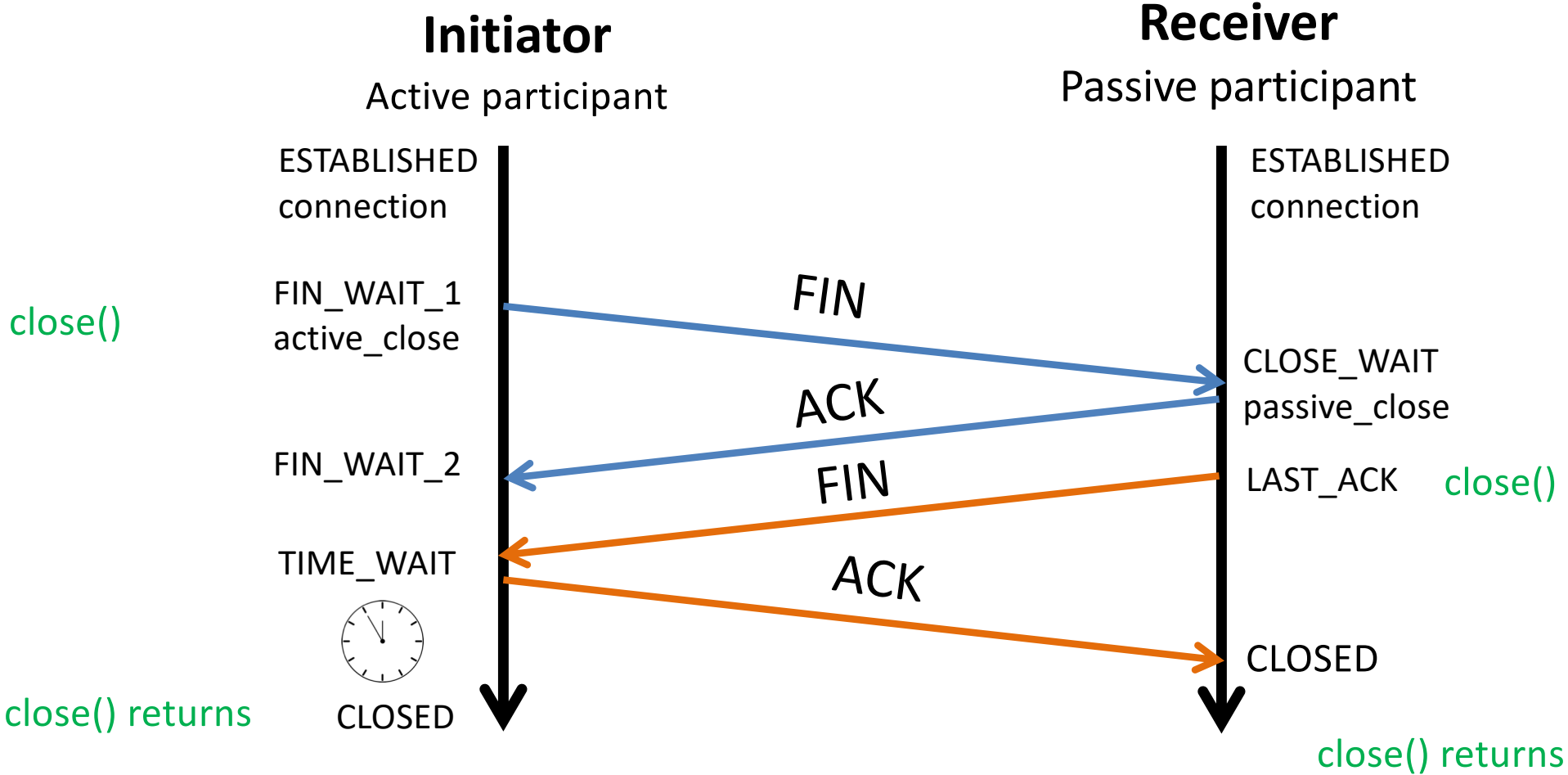
TCP Connection Teardown



Both sides agree on closing the connection.



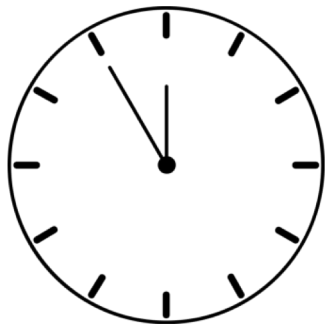
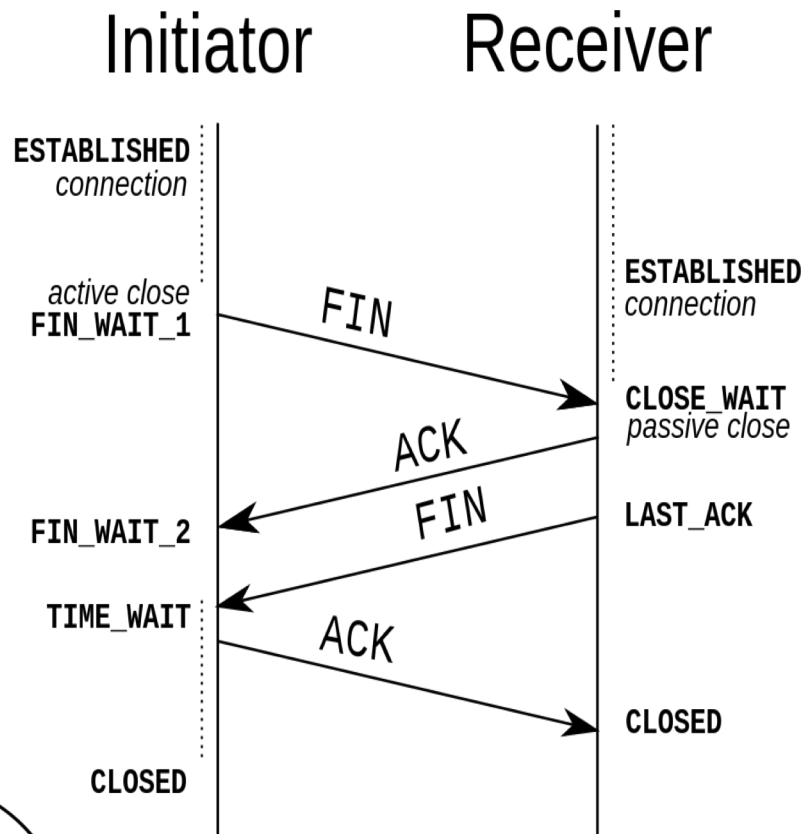
TCP Connection Teardown



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 $2 * \text{MSL}$ (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

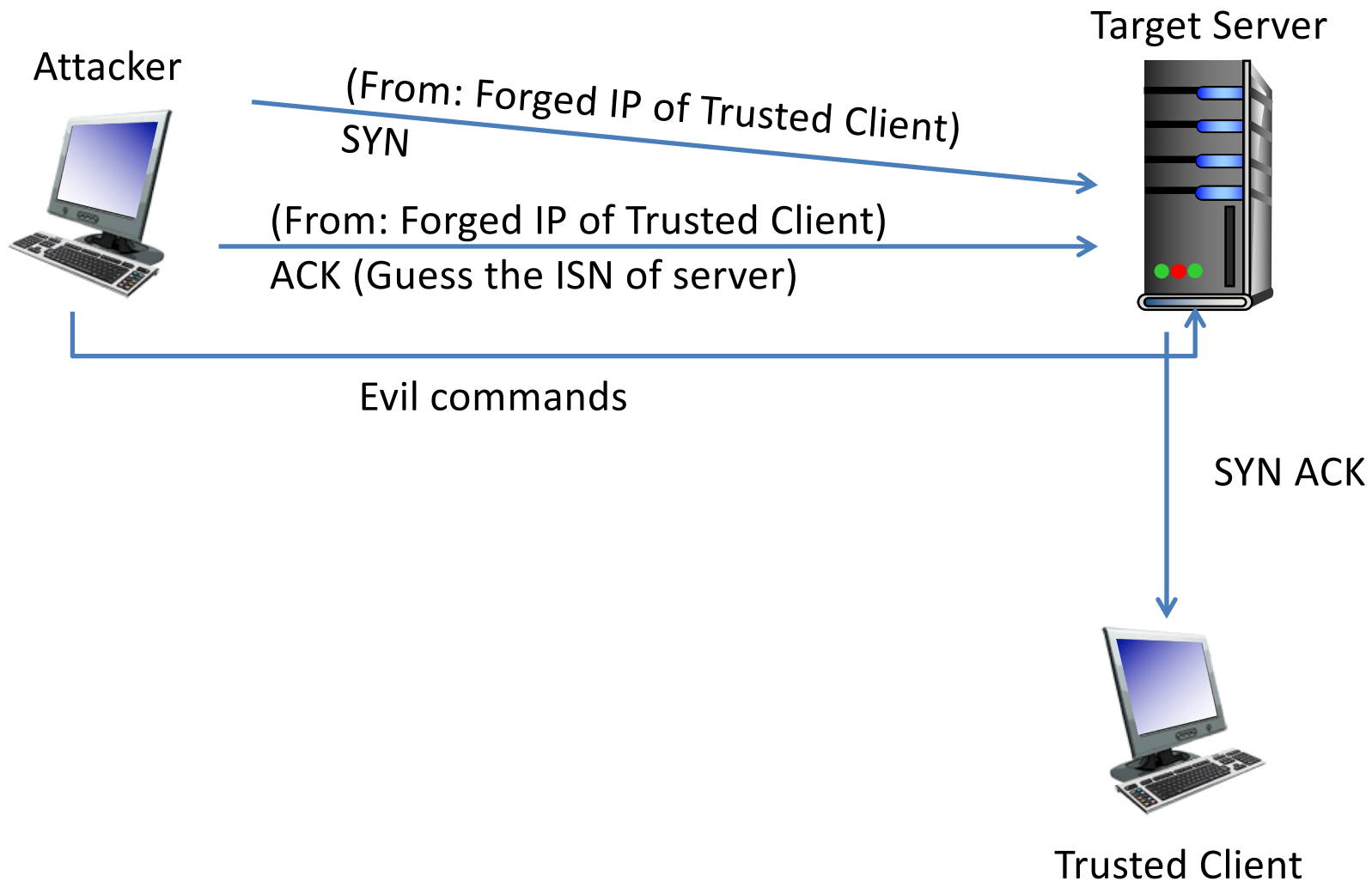
D. Start from some other value (such as...?)

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)

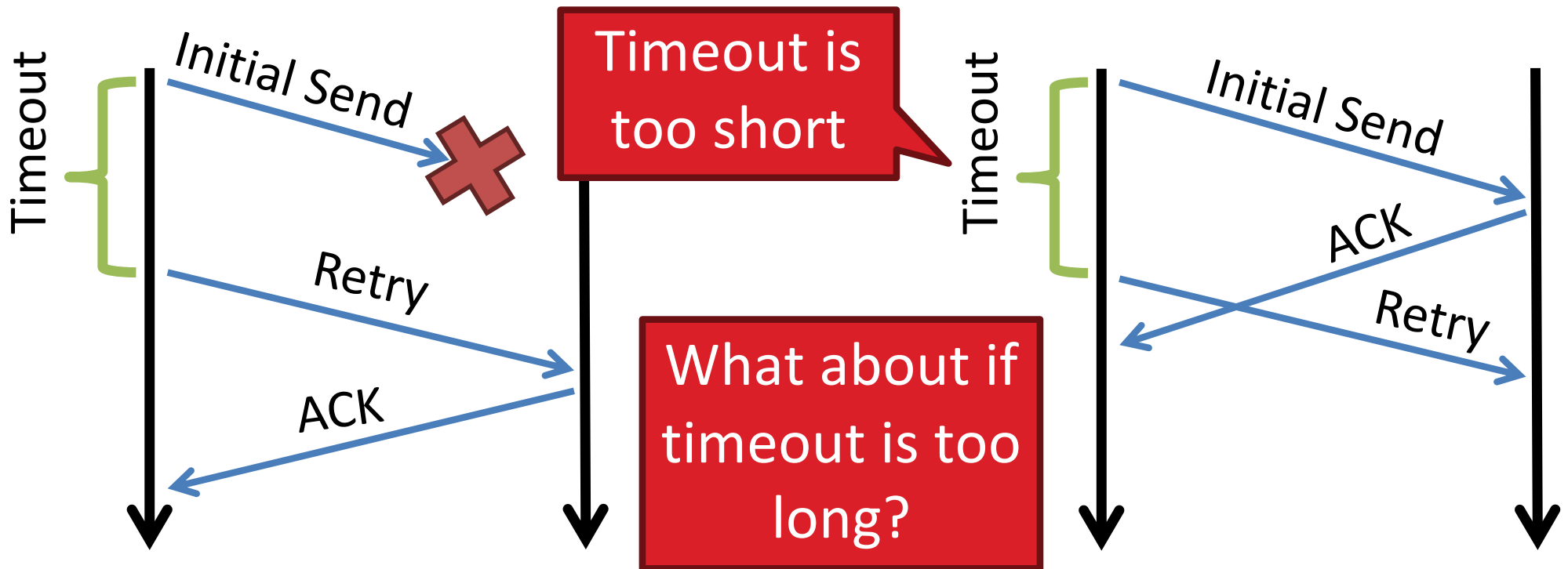


Practical Reliability Questions

- 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?

Setting Timeout Values

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



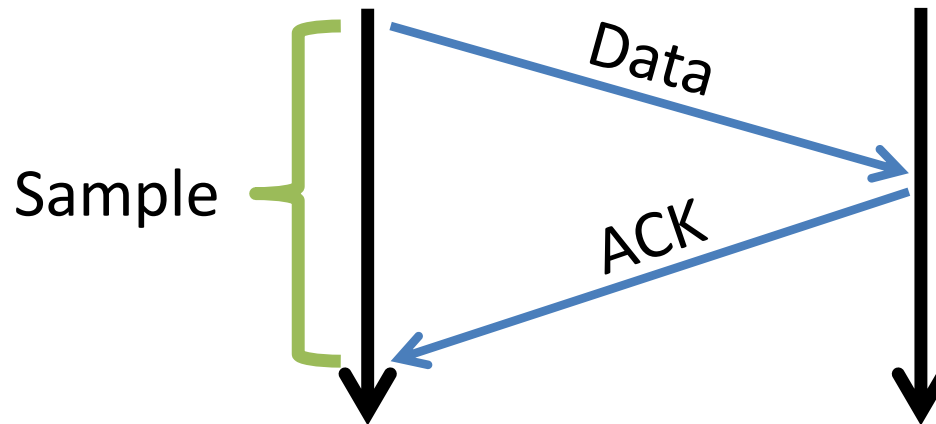
Timeouts

- 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

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)



$$\text{EstimatedRTT} = (1 - a) * \text{EstimatedRTT} + a * \text{SampleRTT}$$

– a is usually 1/8.

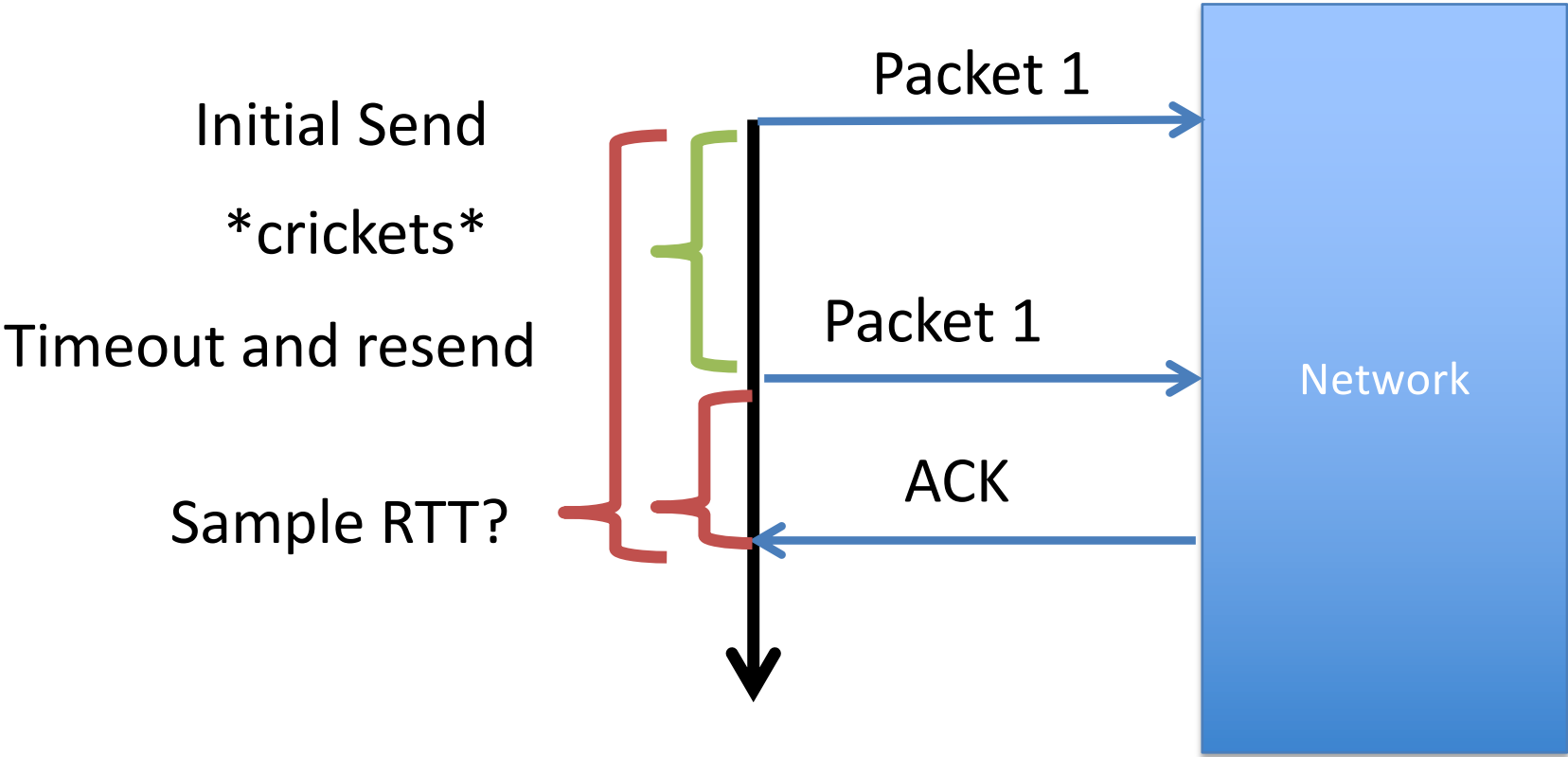
In words current estimate is a blend of:

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

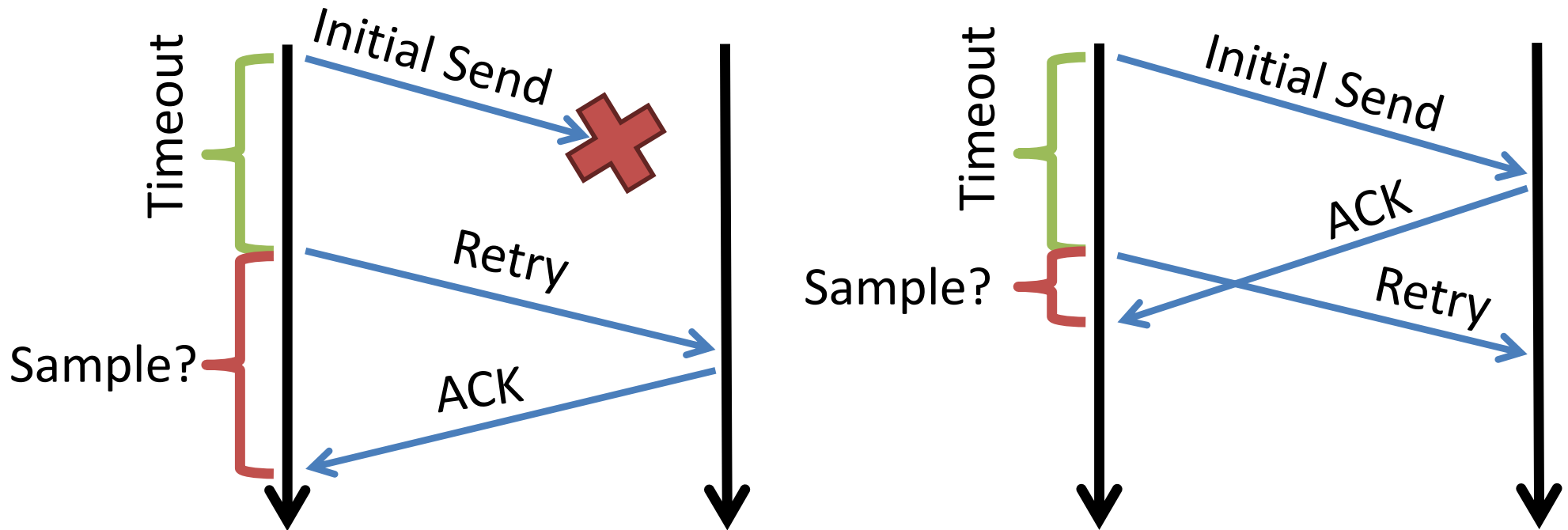
$$\text{DevRTT} = (1 - B) * \text{DevRTT} + B * | \text{SampleRTT} - \text{EstimatedRTT} |$$

- B is usually 1/4

RTT Sample Ambiguity: Sender's Perspective



RTT Sample Ambiguity



Ignore samples for retransmitted segments

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)

Example RTT Estimation

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

$$\text{New estimate} = 7/8 * 64 + 1/8 * 120 = 56 + 15 = 71$$

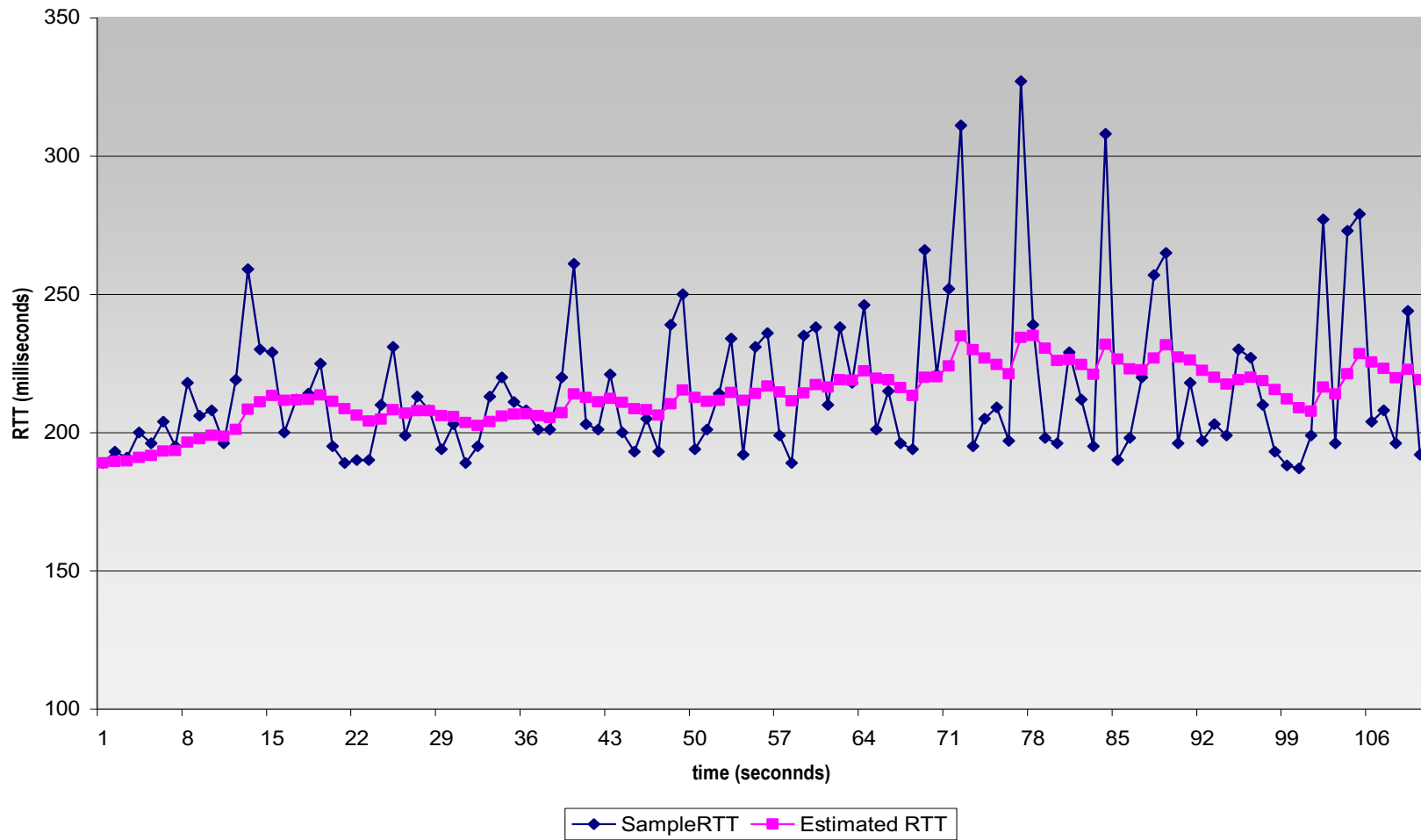
$$\text{New dev} = 3/4 * 8 + 1/4 * | 120 - 71 | = 6 + 12 = 18$$

- Another sample: 400

$$\text{New estimate} = 7/8 * 71 + 1/8 * 400 = 62 + 50 = 112$$


$$\text{New dev} = 3/4 * 18 + 1/4 * | 400 - 112 | = 13 + 72 = 85$$

Example RTT Estimation (Smoothing)



TCP Timeout Value

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

 ↑ ↑

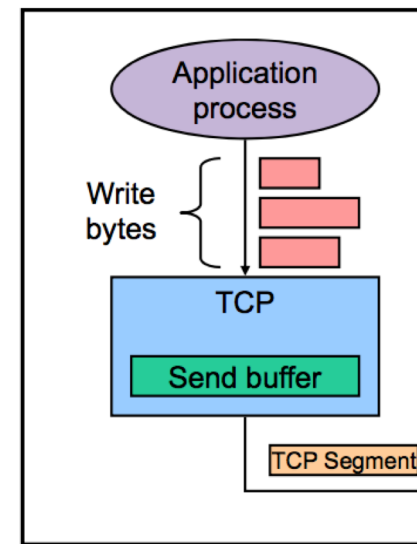
estimated RTT “safety margin”

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?

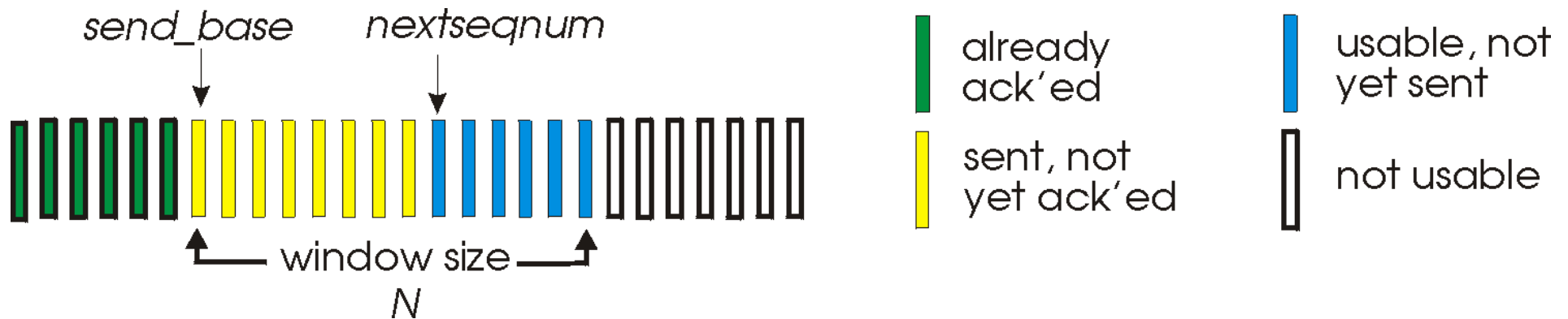
Windowing (Sliding Window)

- At the sender:
 - What's been ACKed
 - What's still outstanding
 - What to send next
- At the receiver:
 - Go-back-N
 - Highest sequence number received so far.
 - (Selective repeat)
 - Which sequence numbers received so far.
 - Buffered data.



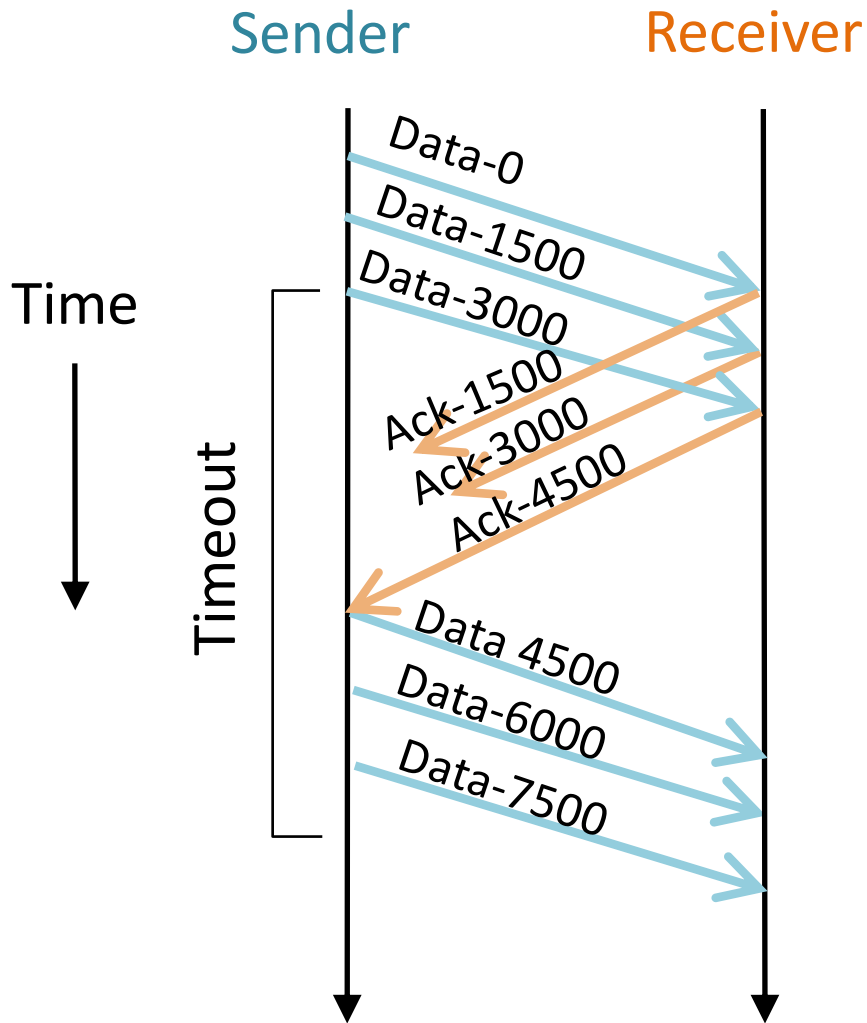
Go-back-N

- At the sender:



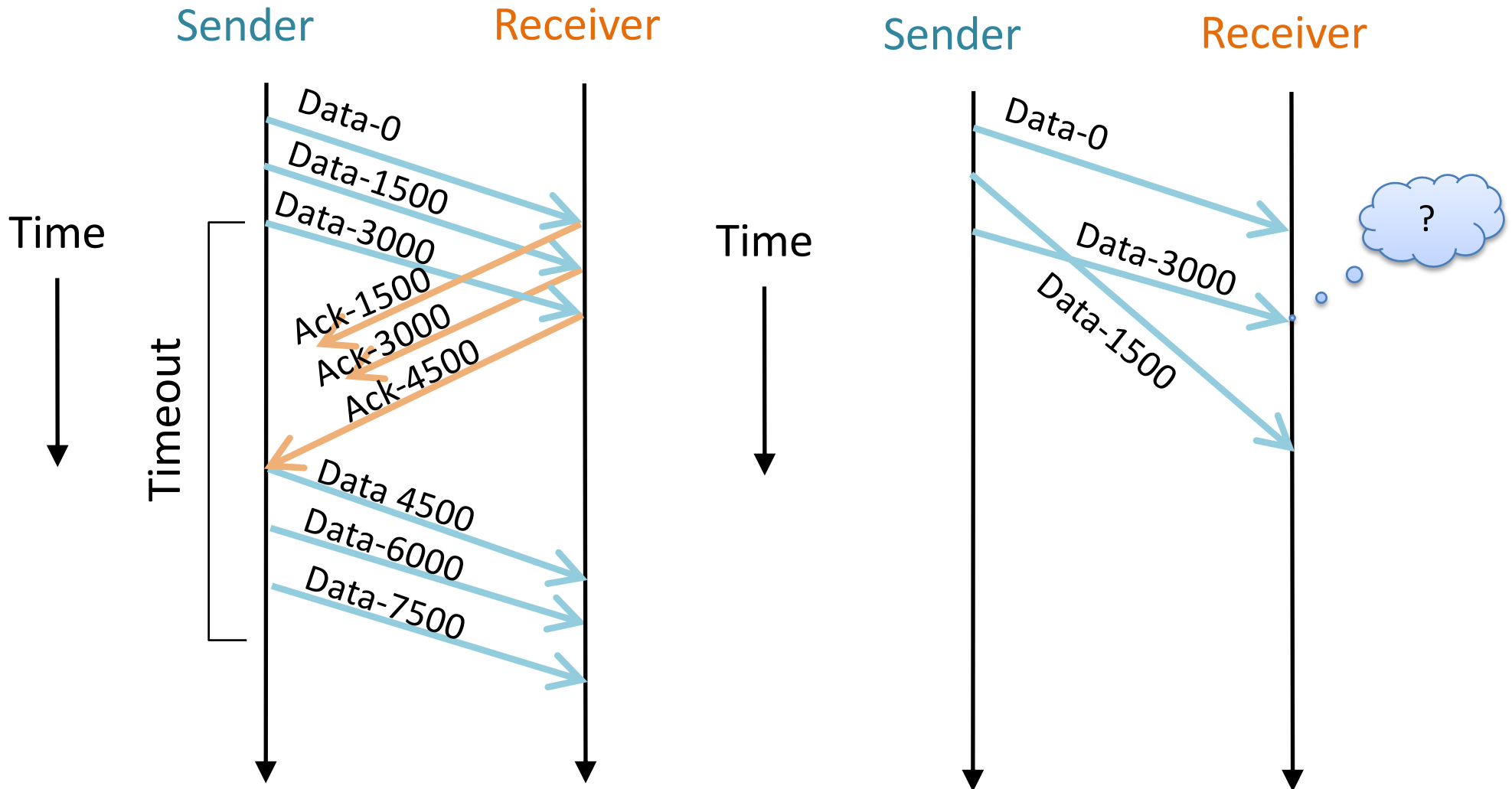
- At the receiver:
 - Keep track of largest sequence number seen.
 - If it receives ANYTHING, sends back ACK for largest sequence number seen so far. (Cumulative ACK)

Cumulative ACKs



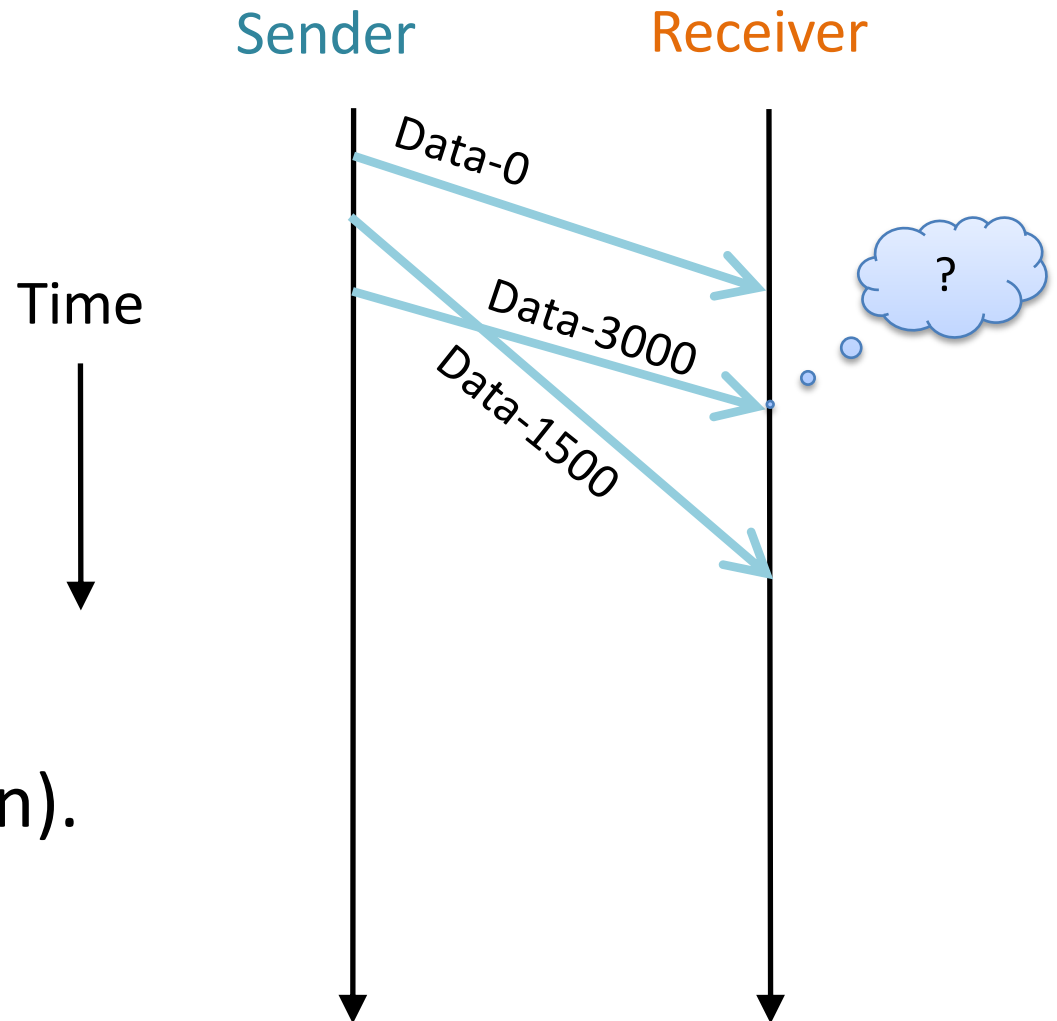
- An ACK for sequence number N implies that all data prior to N has been received.

Cumulative ACKs

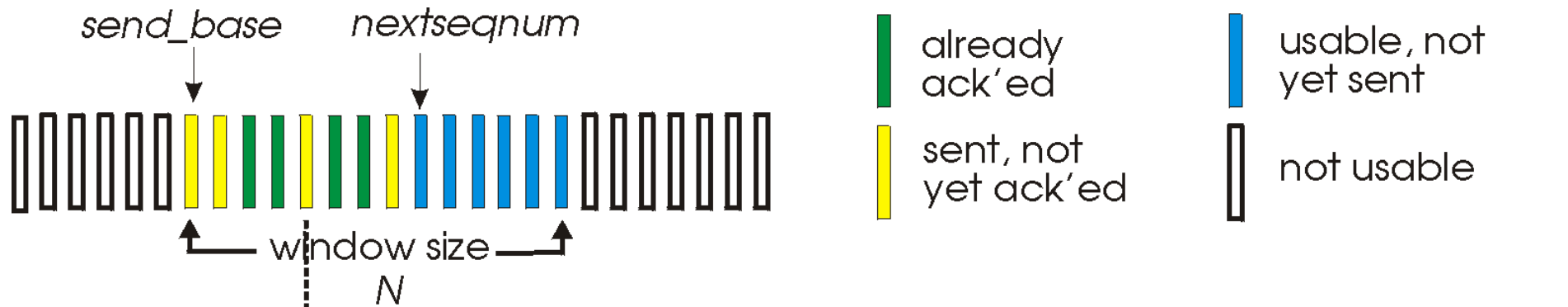


What should we do with an out-of-order segment at the receiver?

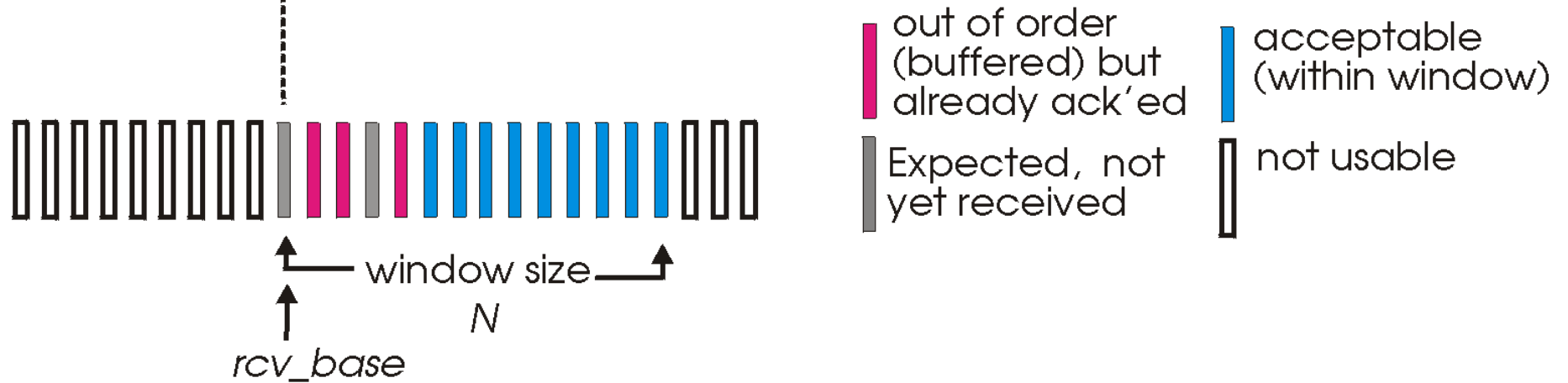
- A. Drop it.
- B. Save it and ACK it.
- C. Save it, don't ACK it.
- D. Something else (explain).



Selective Repeat



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

If you were building a transport protocol,
which would you use?

- A. Go-back-N
- B. Selective repeat
- C. Something else (explain)

TCP Summary

- 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