P1 : Distributed Bitcoin Miner

15-440/15-640
09/19/2018

Overview

- Debugging Tips
- P0 Solution
- P1 Part A

Debugging Tips

Logging

- Add log statements around points of inter-thread communications
  - Channel
  - Connection
  - …
- Attach server/client IDs to each log line
- Create multiple log files, one for each thread
- (Demo)

Debugging Tips

Logging

- Limit log output size for easy navigation
  - Convenient to use flags to enable/disable logging
  - No multi-level logging support in Go's standard library
  - Beware of the complaints from '-race' when a single logger is used

Debugging Tips

Why read the tests programs?

- Understand the expected behavior of the program.
  - The system specification is written in natural language and thus inherently **ambiguous** and prone to **misinterpretation**.
  - The test program is more **precise**.
- Prepare for Part B.
  - The public tests are simple. The hidden tests are brutal.
  - You will need to write good test programs to catch bugs.

Debugging Tips

Tools

- GoLand
- GDB for Go
- Delve

P0 Reference

- Reference solution to p0 from a previous semester will be posted
- Should be structurally identical

Timeline

- Part A Checkpoint (Due 9/25)
- Part A (Due 10/6)
- Part B (Due 10/16)

Part A: LSP Protocol

- You will implement the *Live Sequence Protocol*
- LSP has some features of both UDP and TCP
- LSP also has its own features

LSP Features

- LSP supports its own client-server communication model
- Server communicates with multiple clients
- **Received messages must be processed in order**
- LSP includes **Sliding Window Protocol**
- **Payload size** and **Checksum** are used to verify data integrity.
- LSP includes **Epoch Events for Retransmission and Timeout Mechanism**

Messages

- Each message is consists of:
  - **Message Type:** Connect, Data, Ack
  - **Connection ID:** uniquely identifies each client-server connection
  - **Sequence Number:** sequence number increments with each message sent
  - **Payload Size:** used to verify data integrity
  - **Checksum:** used to verify data integrity
  - **Payload**

Messages

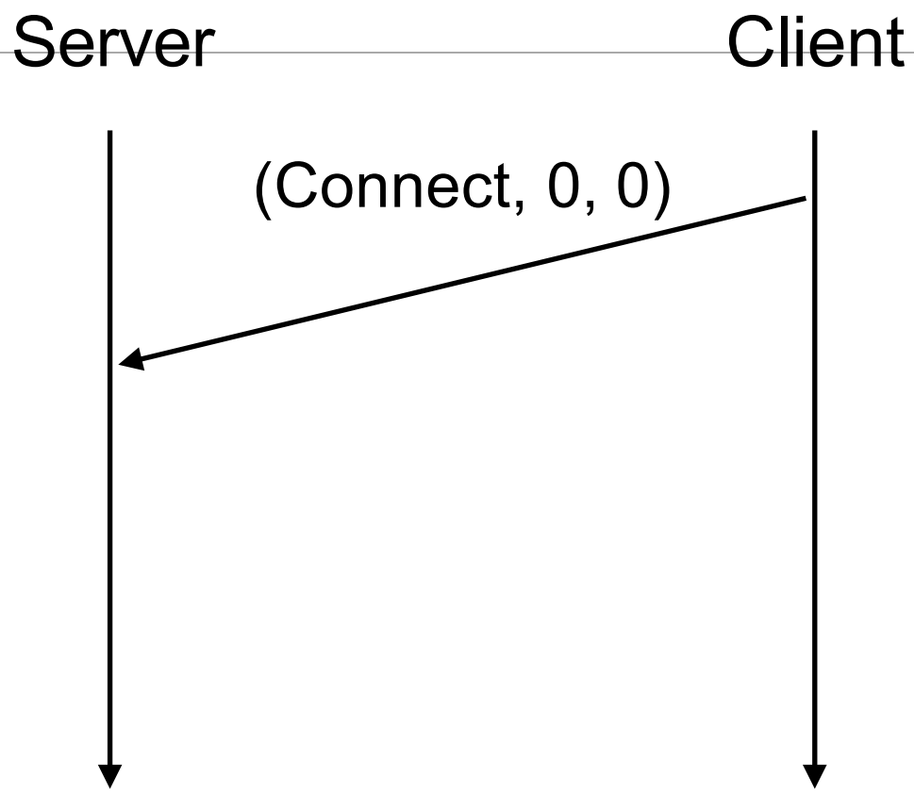
```
type MsgType int
const (
 MsgConnect MsgType = iota // Conn request from client.
 MsgData // Data message from client or server.
 MsgAck // Acknowledgment from client or server.
)

type Message struct {
 Type MsgType // One of the message types listed above.
 ConnID int // Unique client-server connection ID.
 SeqNum int // Message sequence number.
 Size   int // Size of the payload.
 Checksum uint16 // Checksum of the message.
 Payload []byte // Data message payload.
}
```

Messages

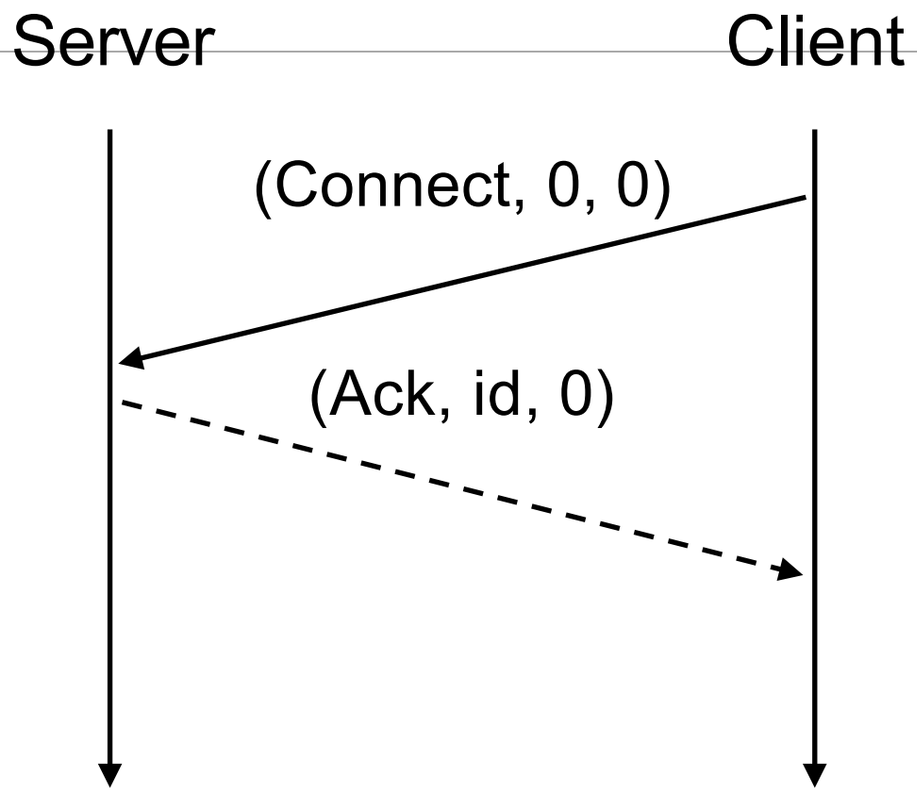
- Message size is limited to single UDP-packet size
- Each Message is received exactly once
- Messages are marshaled using Go's `Marshal` function in the `json` package and sent as a UDP packet

Client-Server Communication:
Establishing a Connection



Client begins by sending a connection request
(must have ID 0 and sequence number 0)

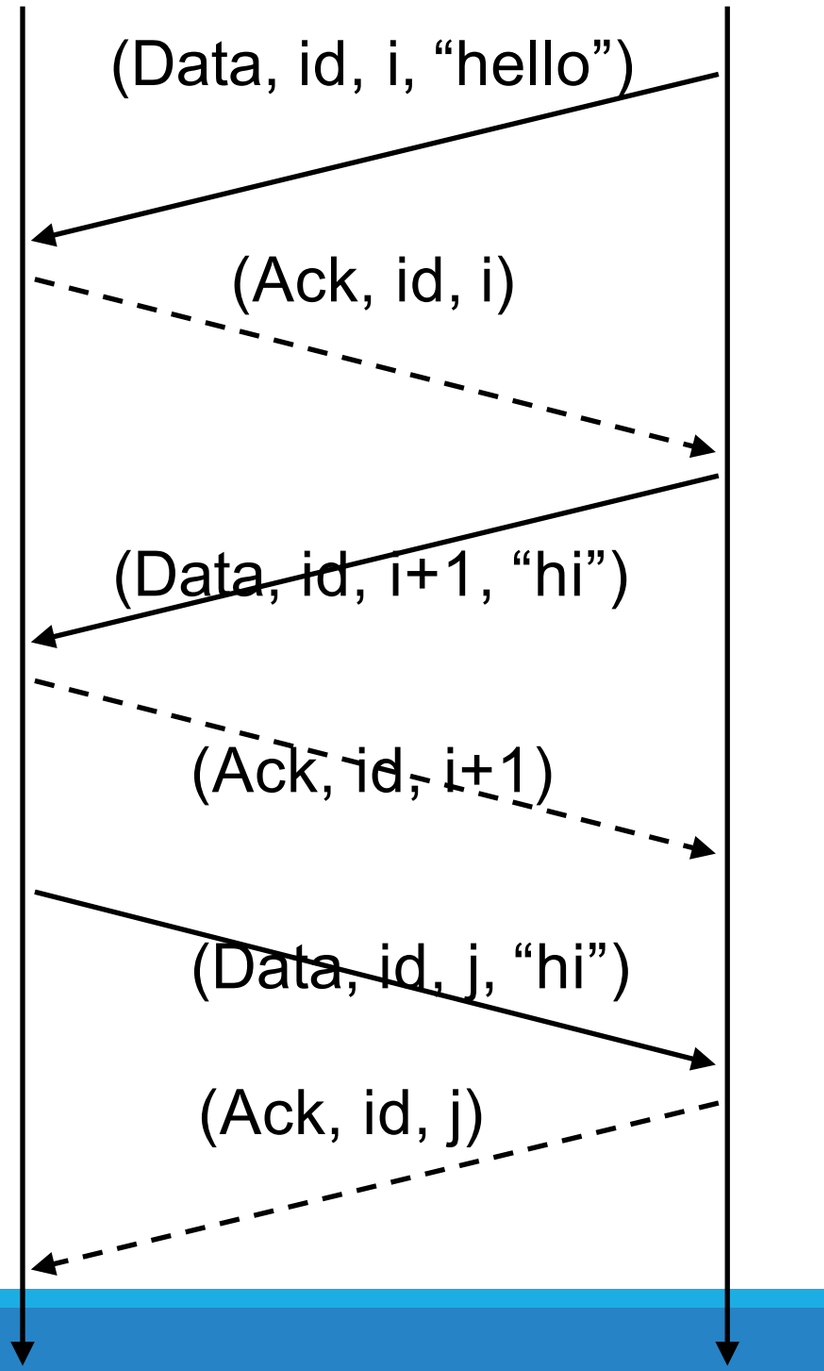
Client-Server Communication:
Establishing a Connection



Server generates a unique connection id
for this Client-Server connection
(you can just generate ID’s sequentially)

Client-Server Communication:
Sending & Ack-ing Data

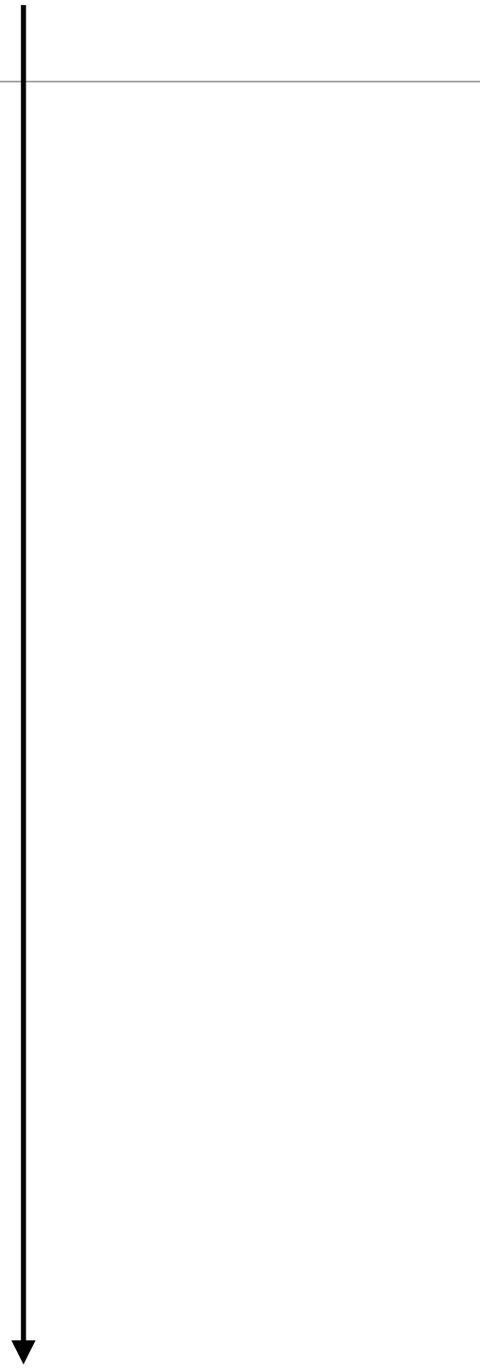
Server and Client maintain independent sequence numbers.



Messages must be received in order.

UDP Packets aren’t guaranteed
to arrive in order.

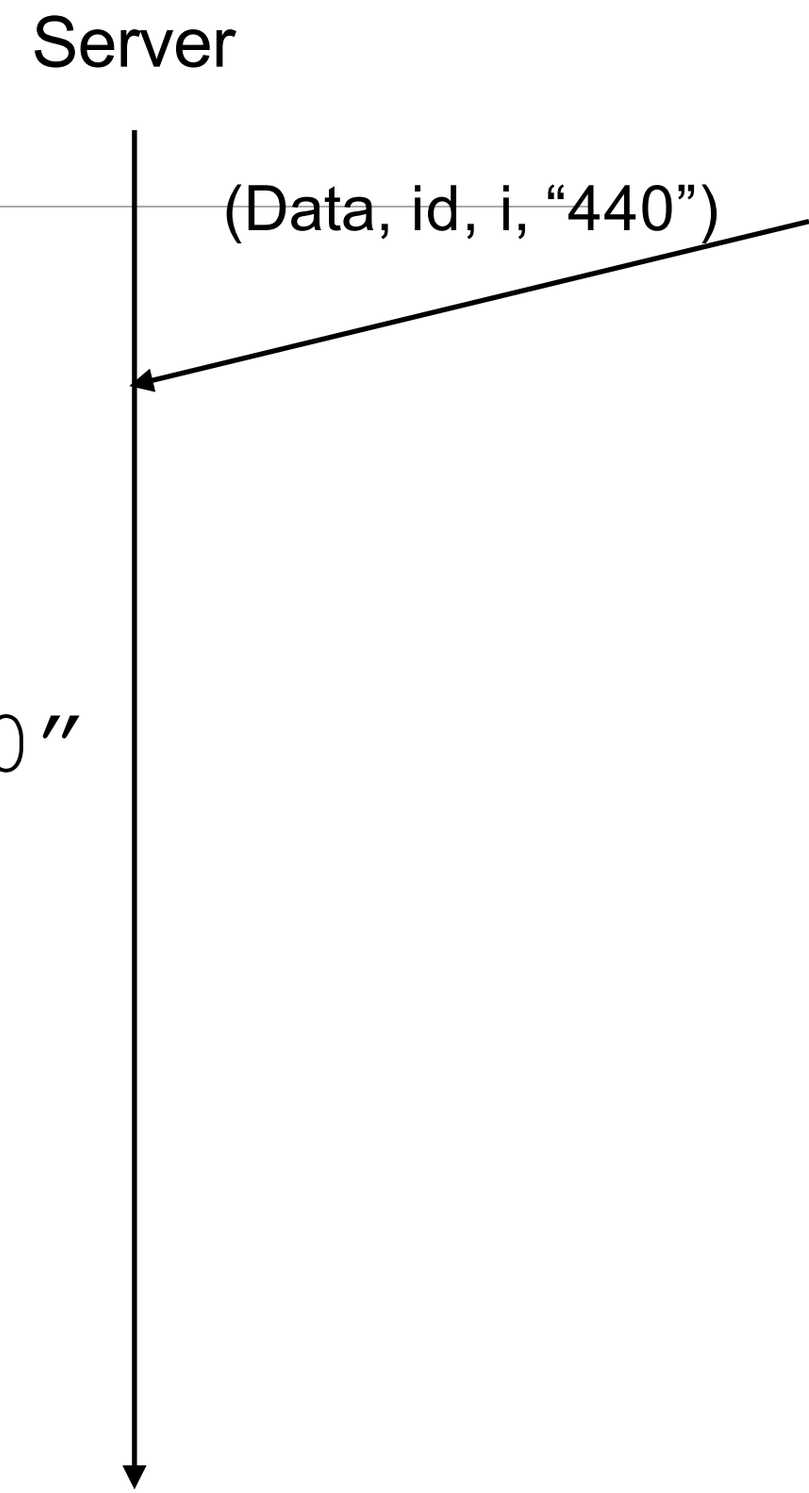
```
LSPServer.Read()//blocks
LSPServer.Read()
LSPServer.Read()
```



Messages must be received in order.

UDP Packets aren't guaranteed to arrive in order.

```
LSPServer.Read()//returns “440”
LSPServer.Read()
LSPServer.Read()
```

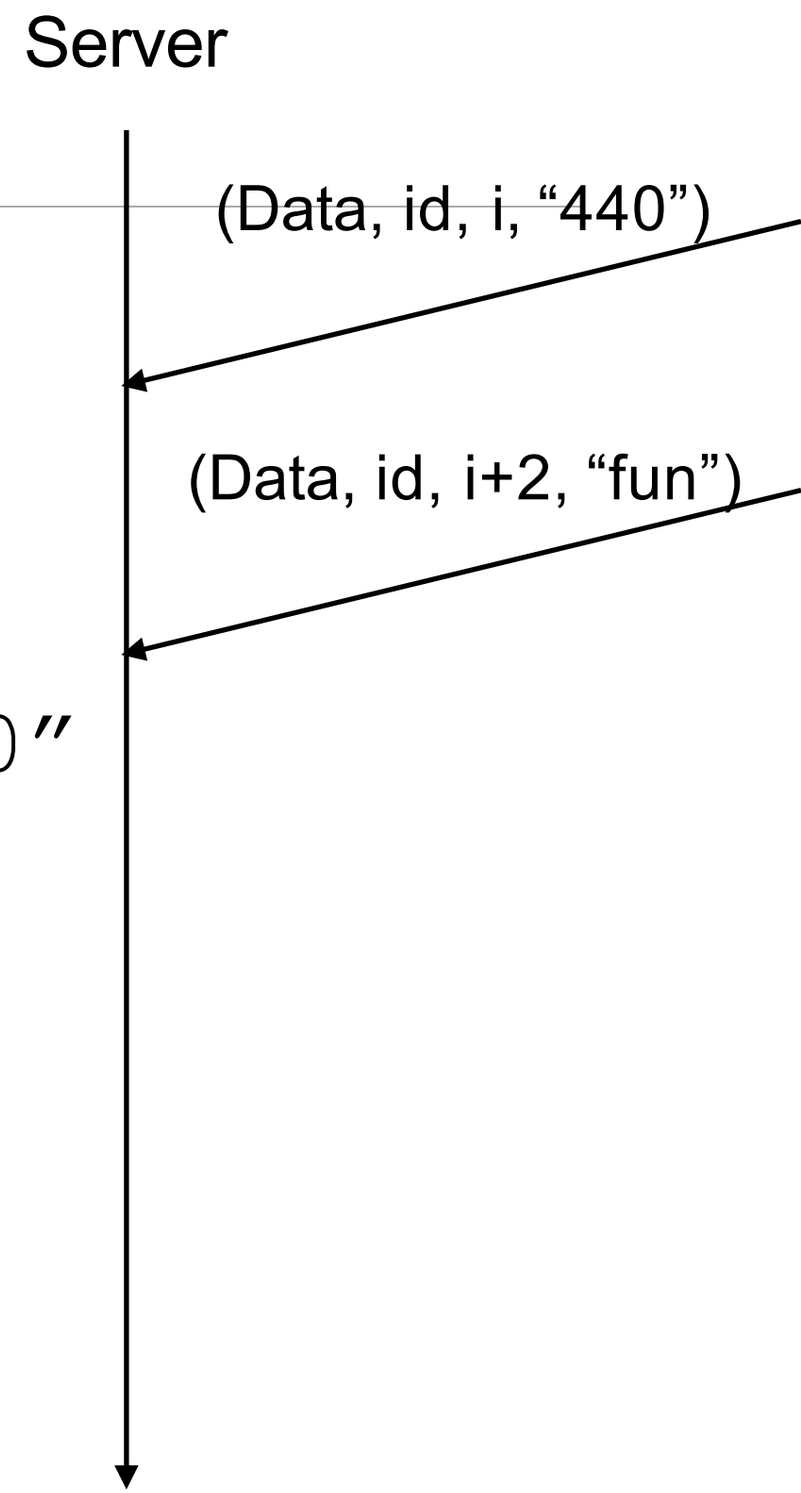


Messages must be received in order.

UDP Packets aren't guaranteed to arrive in order.

```
LSPServer.Read()//returns “440”
LSPServer.Read()//blocks
LSPServer.Read()
```

i + 2 : “fun”

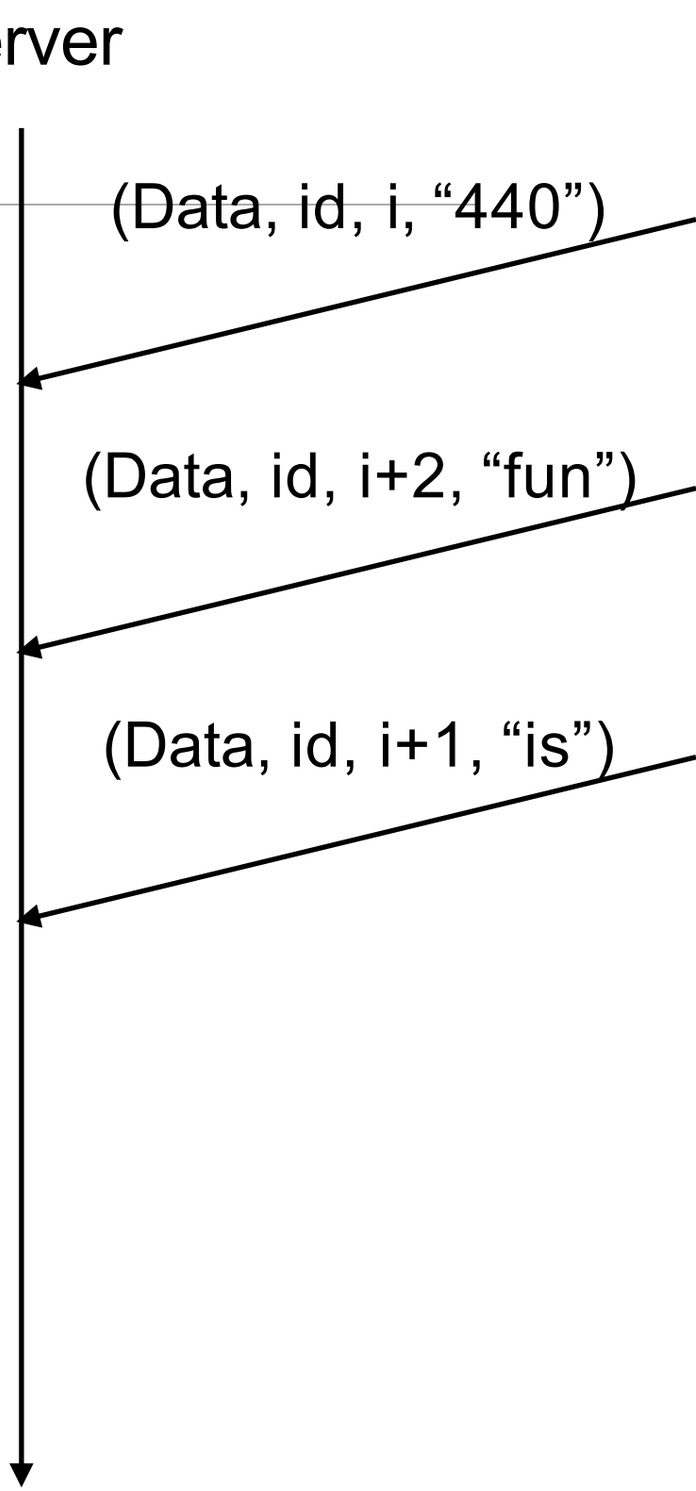


Messages must be received in order.

How to maintain order?

```
LSPServer.Read()//returns “440”
LSPServer.Read()//returns “is”
LSPServer.Read()//returns “fun”
```

i + 2 : “fun”

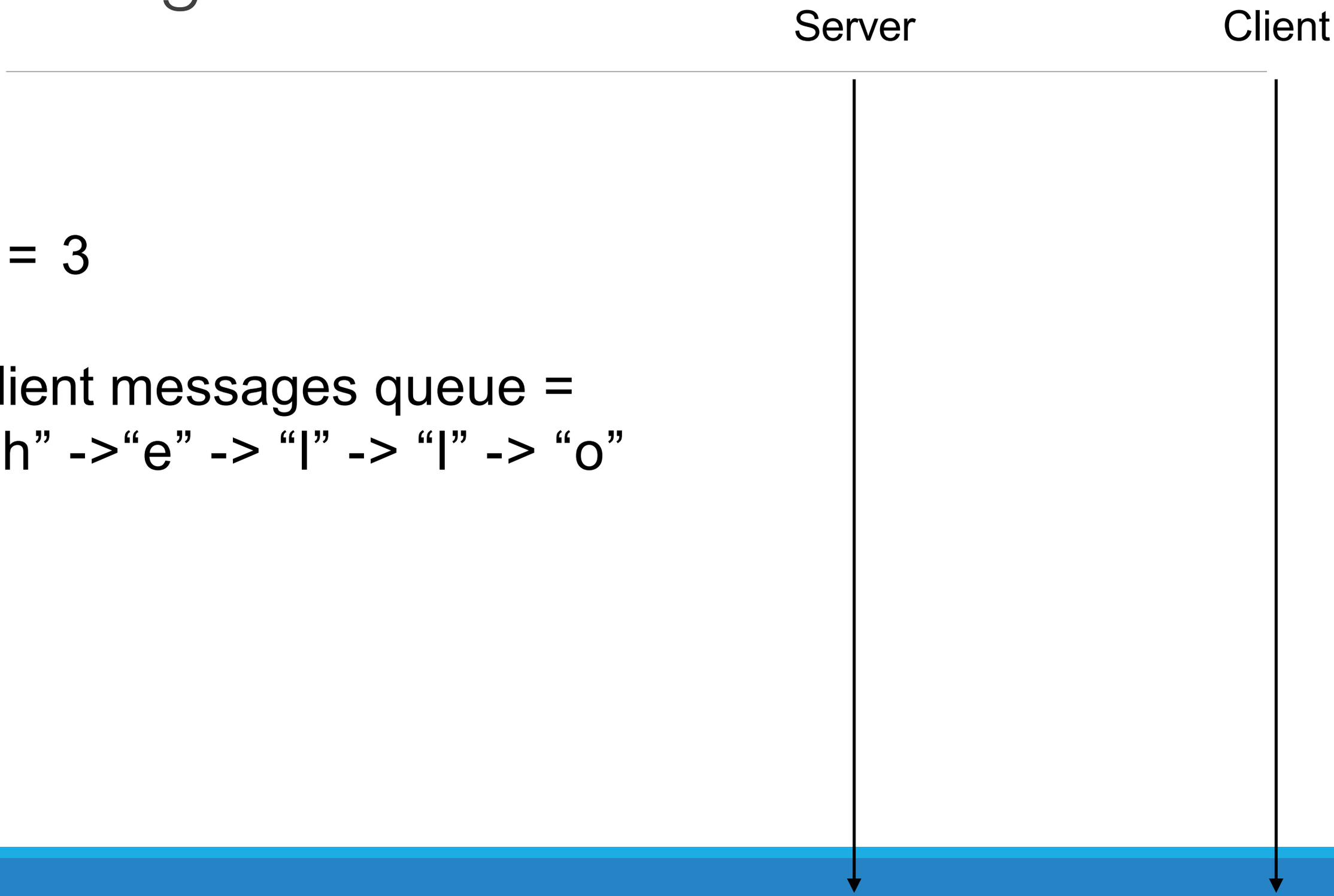


Sliding Window Protocol

- Like TCP, LSP uses a *sliding window protocol*
- Given a window size ω, we can send up to ω messages without acknowledgement.
- If the oldest unacknowledged message has sequence number *n*, then only messages with sequence numbers *n* + ω - 1 (inclusive) may be sent

  i.e. [n, n+ ω -1]

Sliding Window Protocol



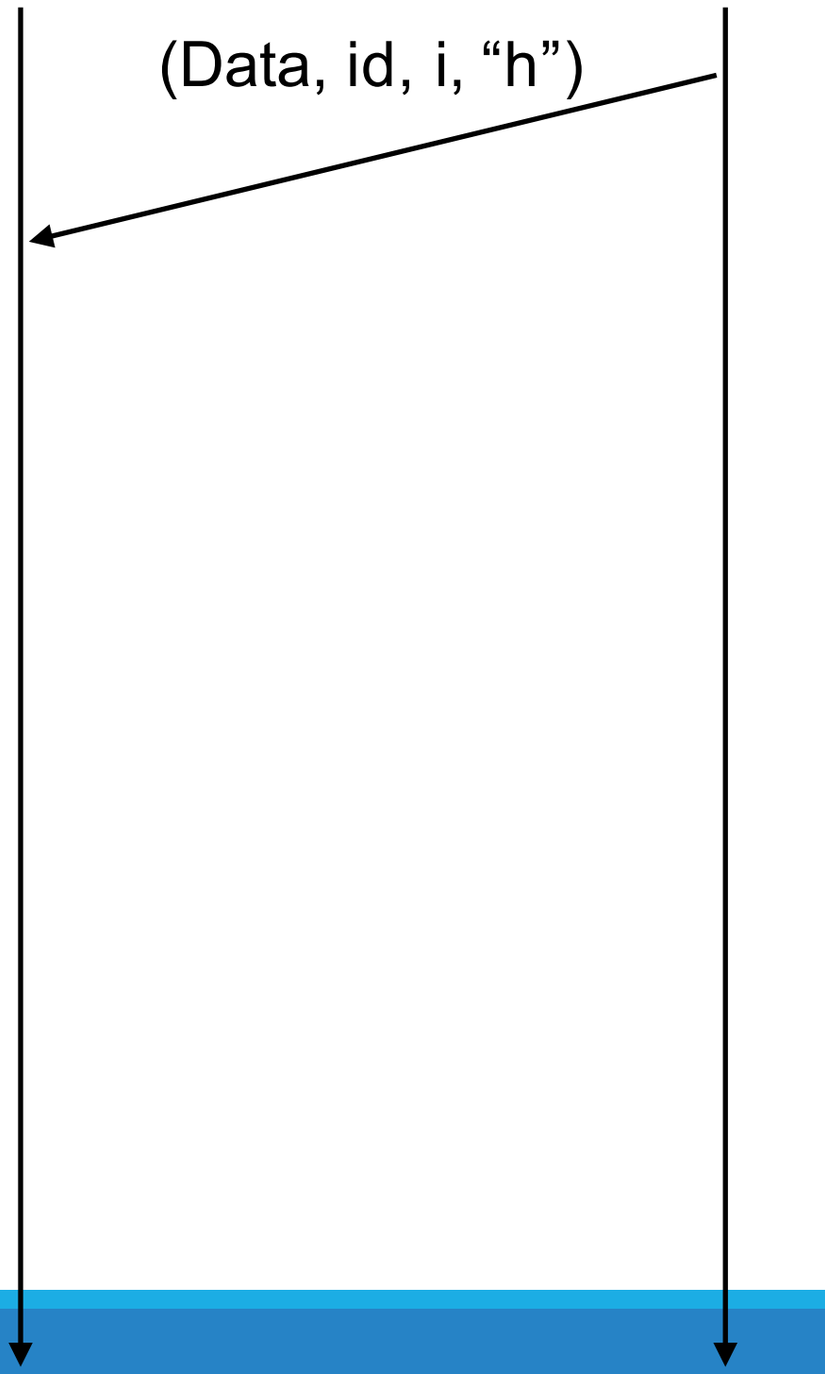
Sliding Window Protocol

$\omega = 3$

Client messages queue =
“e” -> “l” -> “l” -> “o”

Oldest Seq # without Ack = i

Window = [i, i+2]



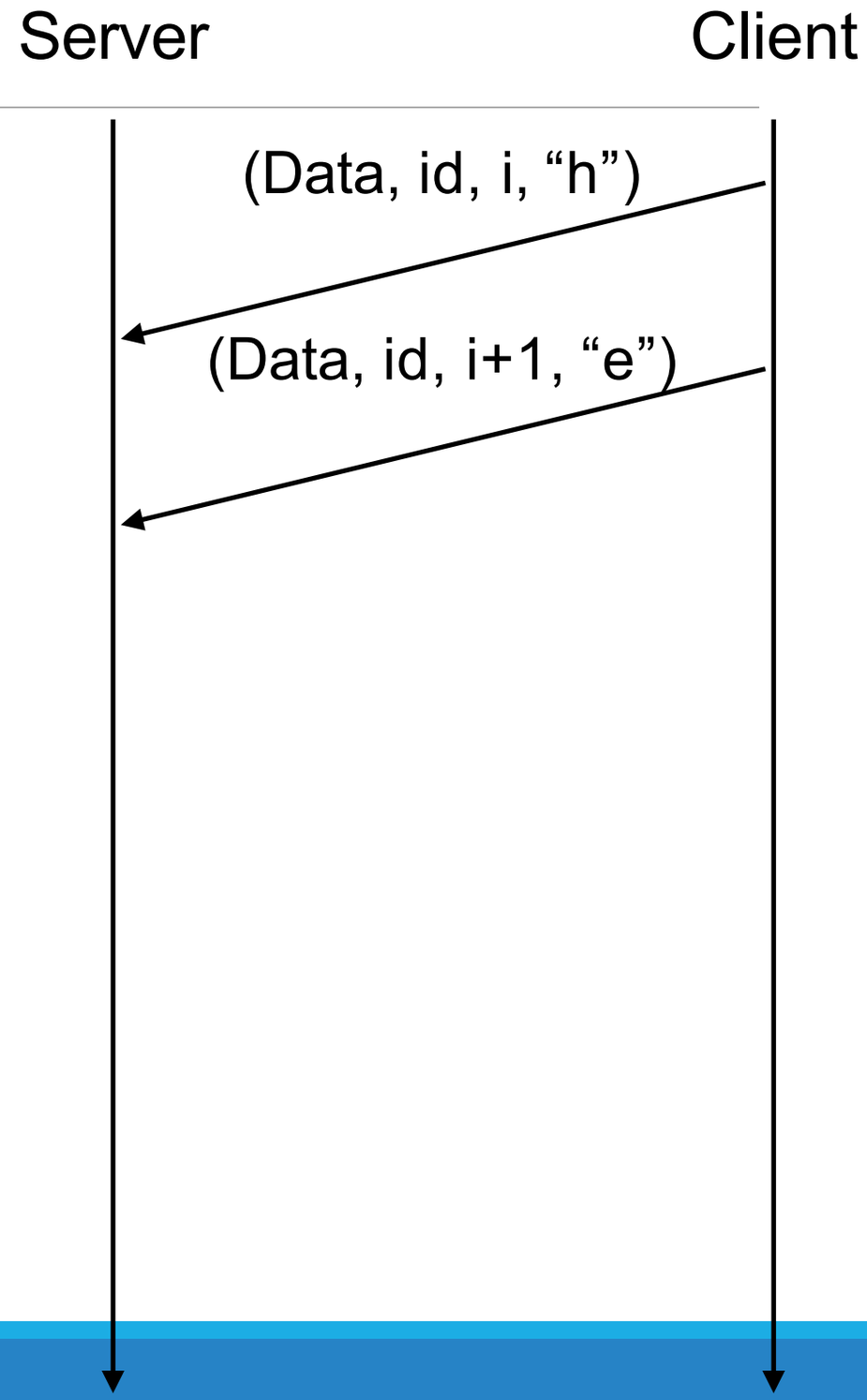
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "l" -> "o"

Oldest Seq # without Ack = i

Window = [i, i+2]



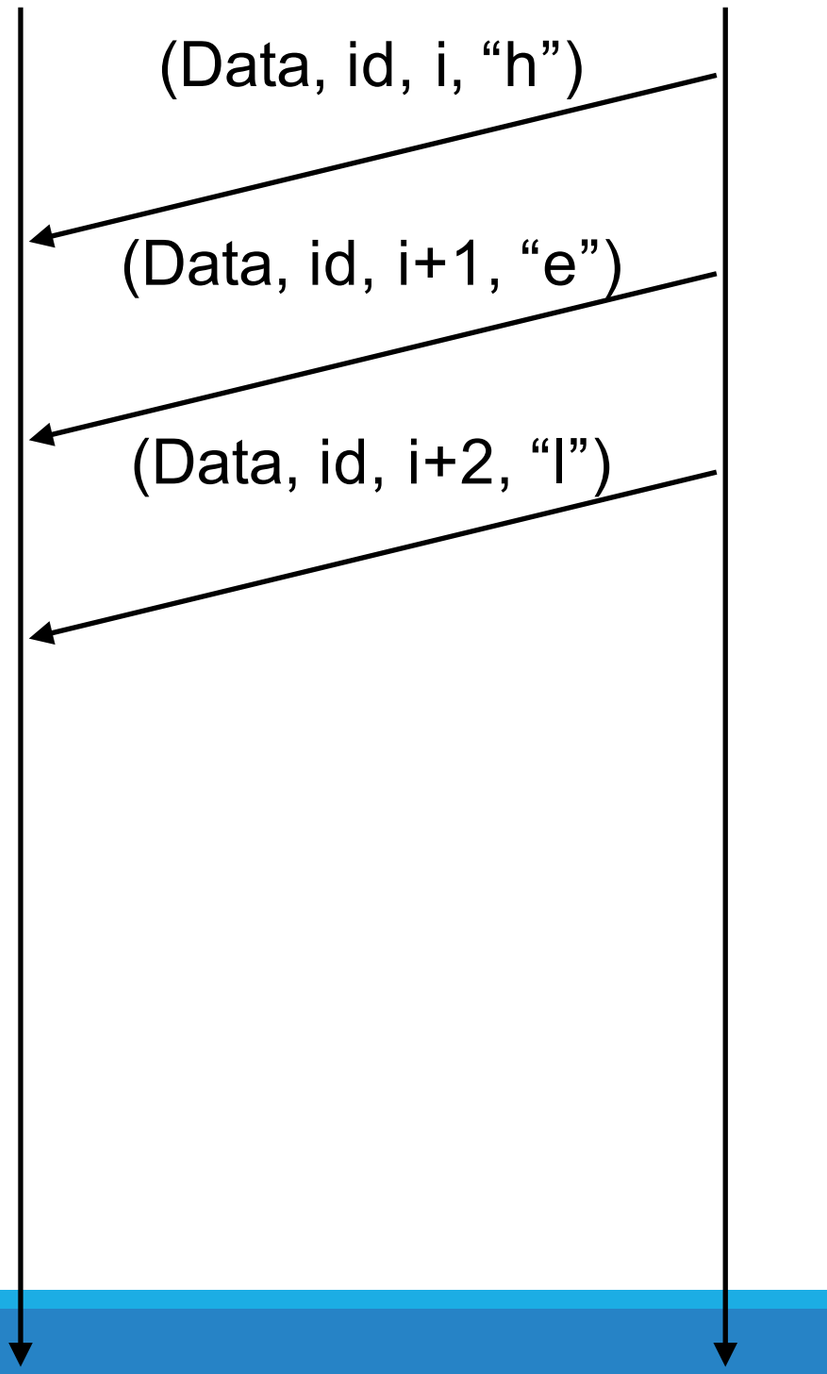
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "o"

Oldest Seq # without Ack = i

Window = [i, i+2]



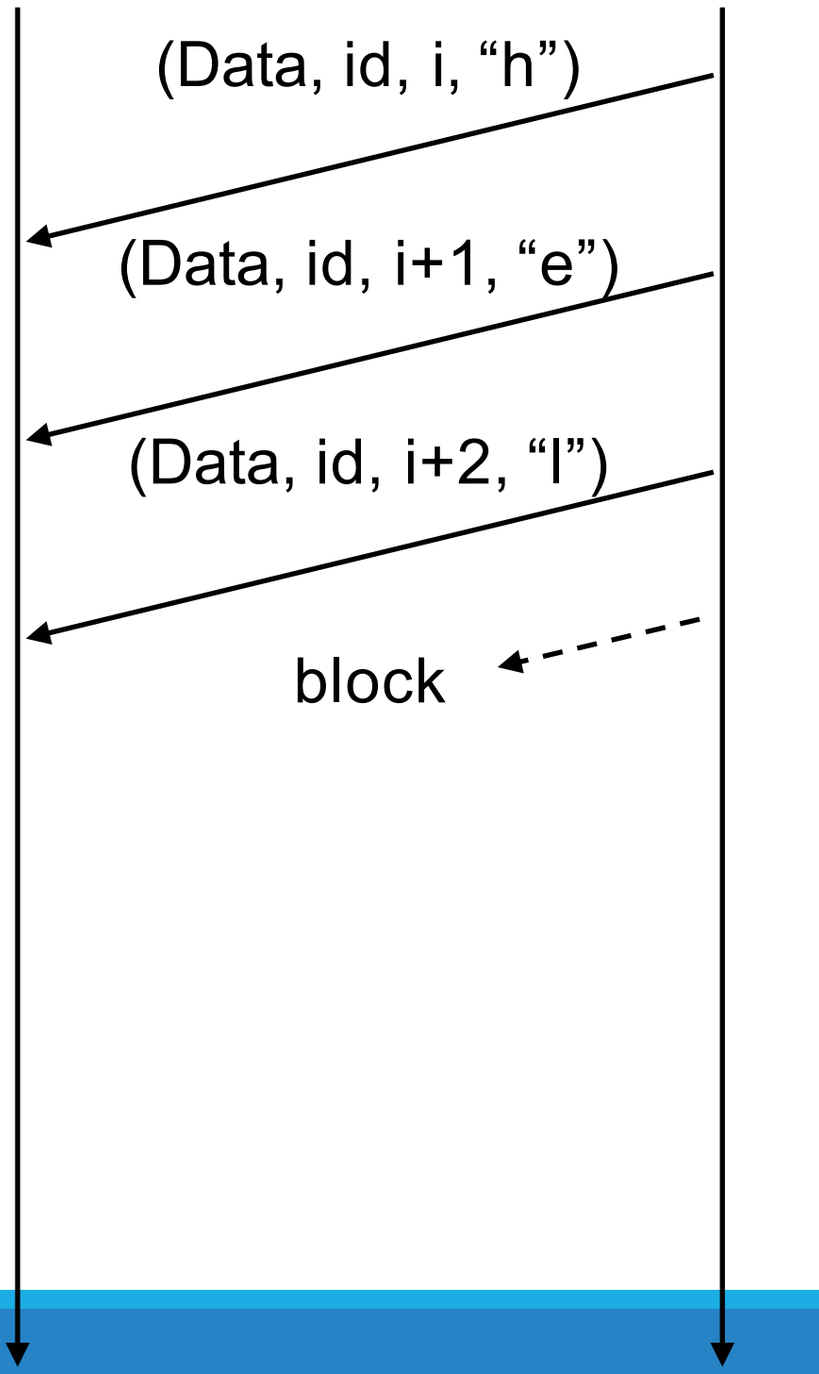
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "o"

Oldest Seq # without Ack = i

Window = [i, i+2]



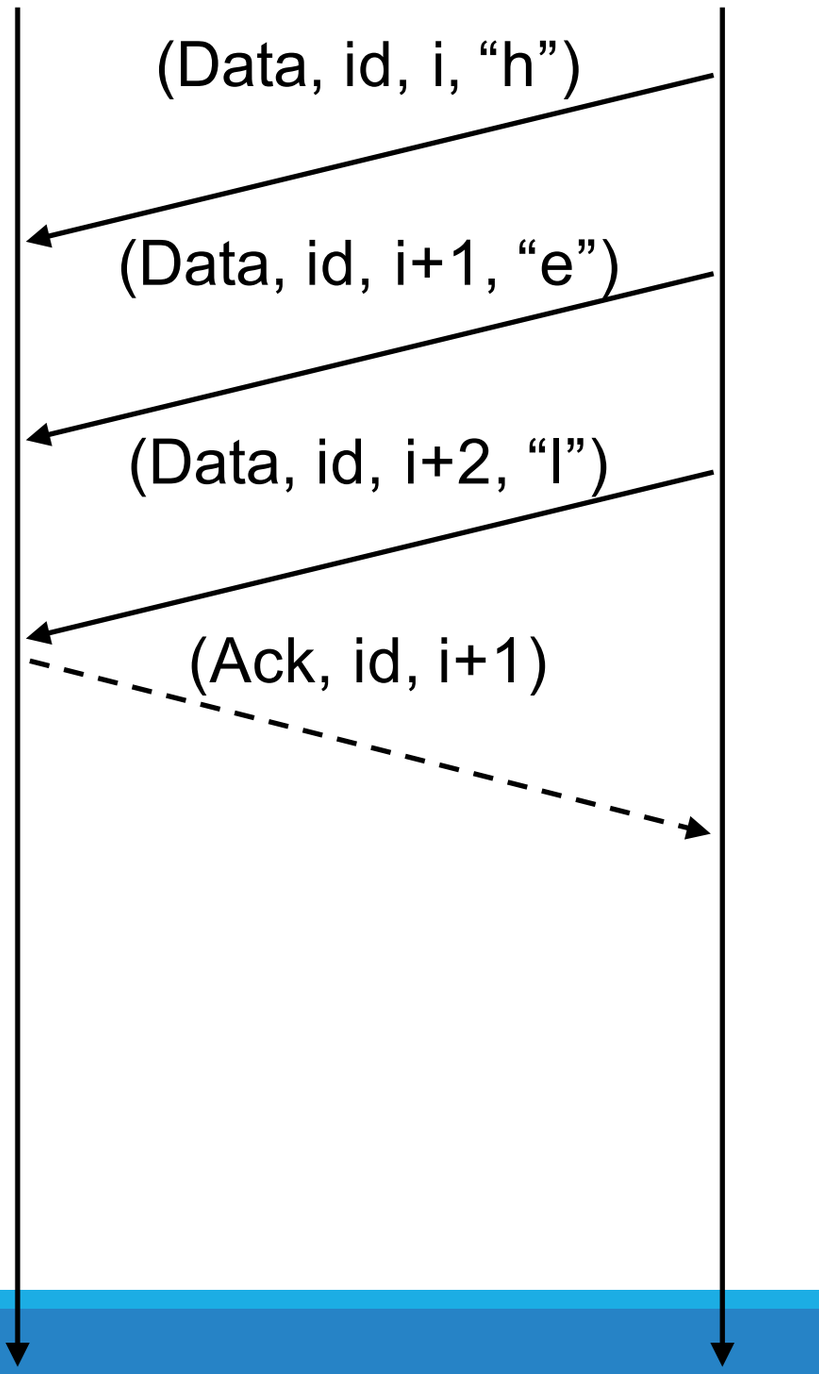
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "o"

Oldest Seq # without Ack = i

Window = [i, i+2]



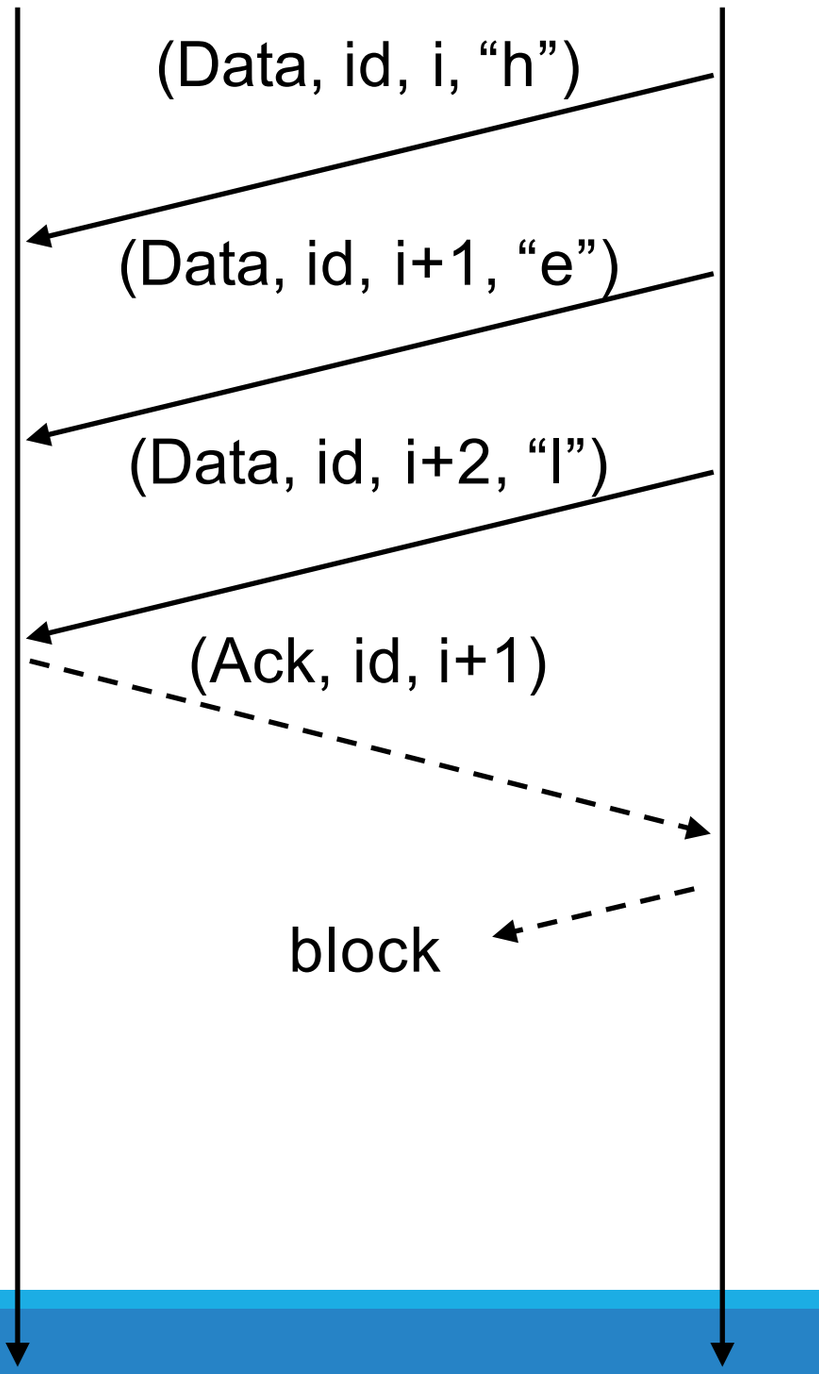
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "o"

Oldest Seq # without Ack = i

Window = [i, i+2]



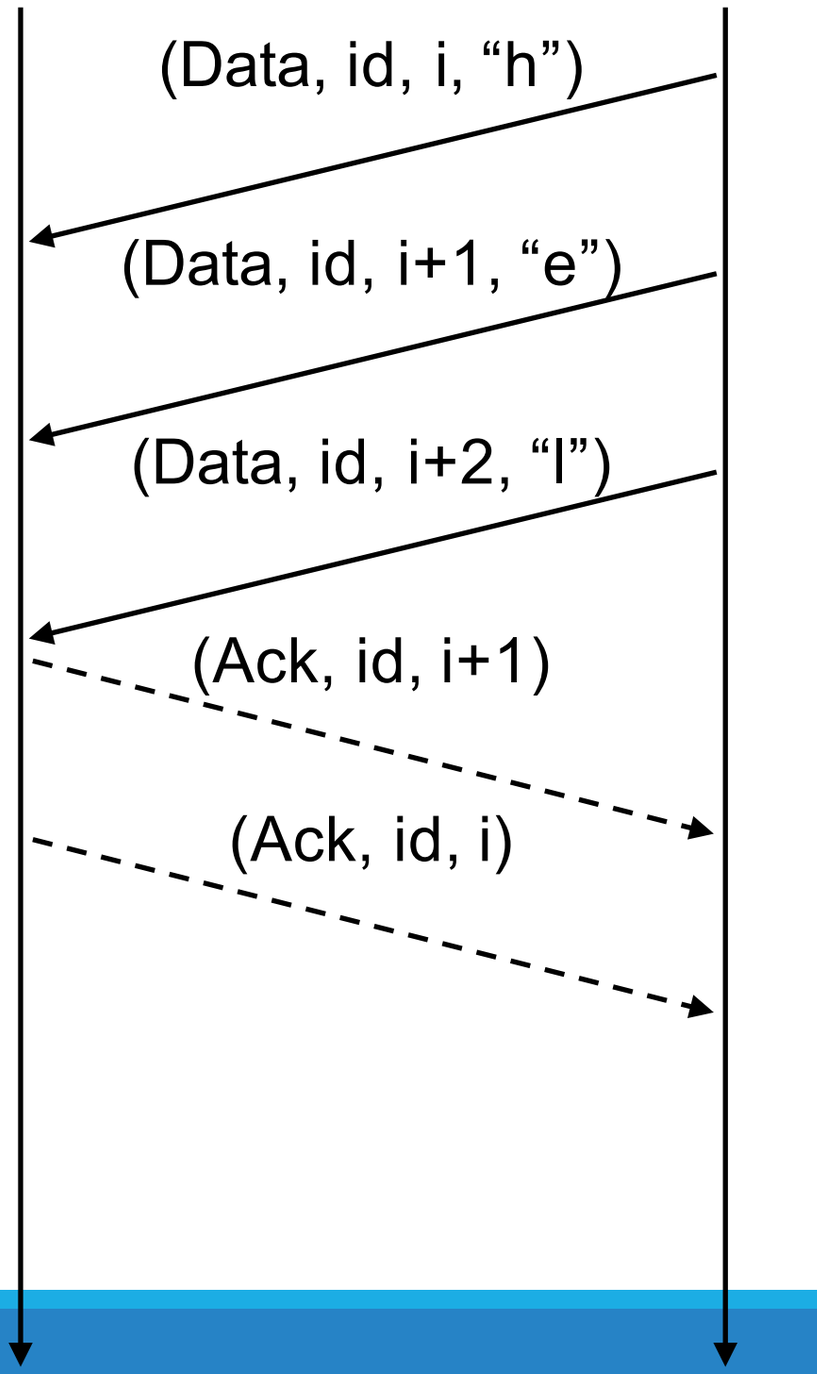
Sliding Window Protocol

$\omega = 3$

Client messages queue =
"l" -> "o"

Oldest Seq # without Ack = i+2

Window = [i+2, i+4]



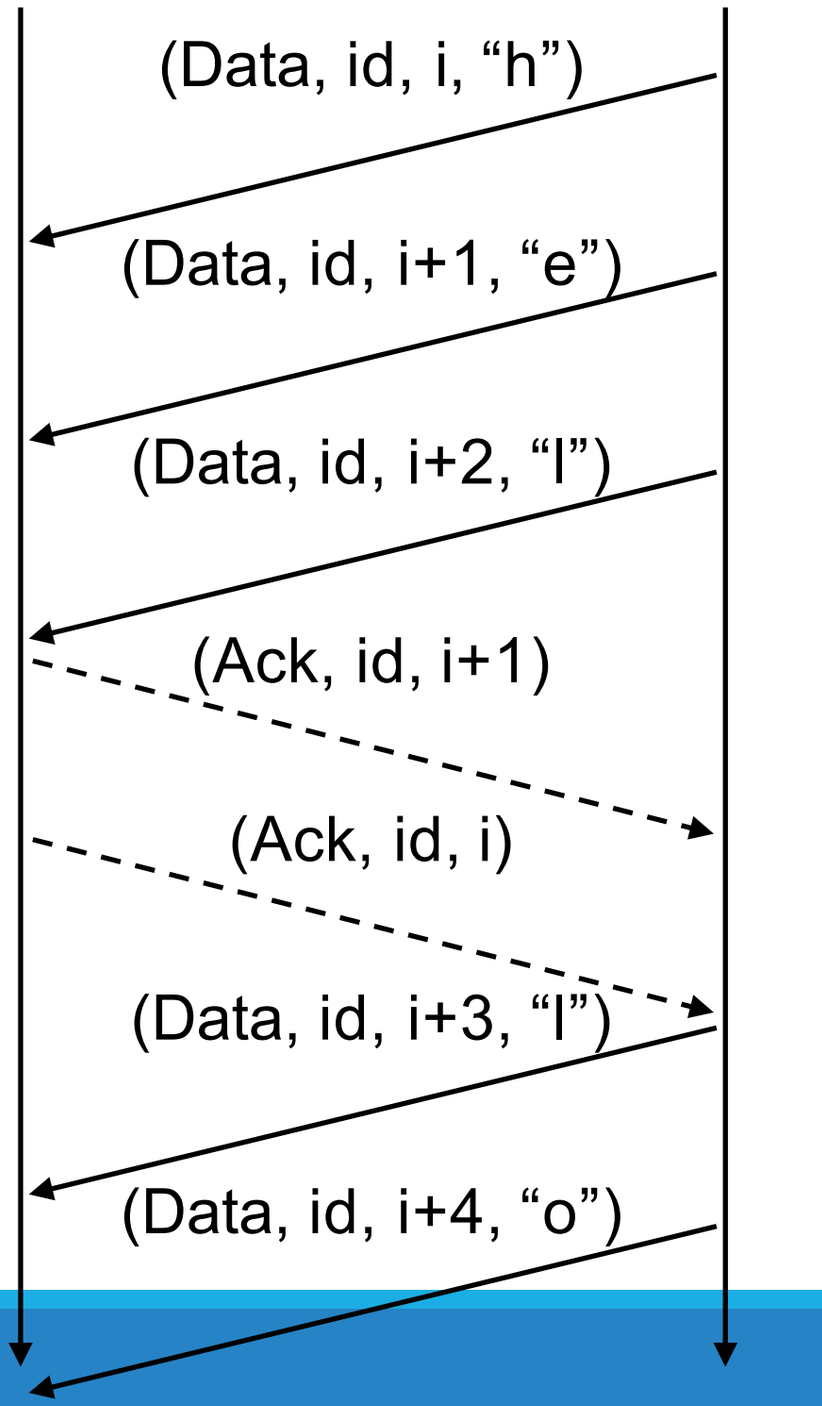
Sliding Window Protocol

$\omega = 3$

Client messages queue =

Oldest Seq # without Ack = i+2

Window = [i+2, i+4]



Payload size and Checksum

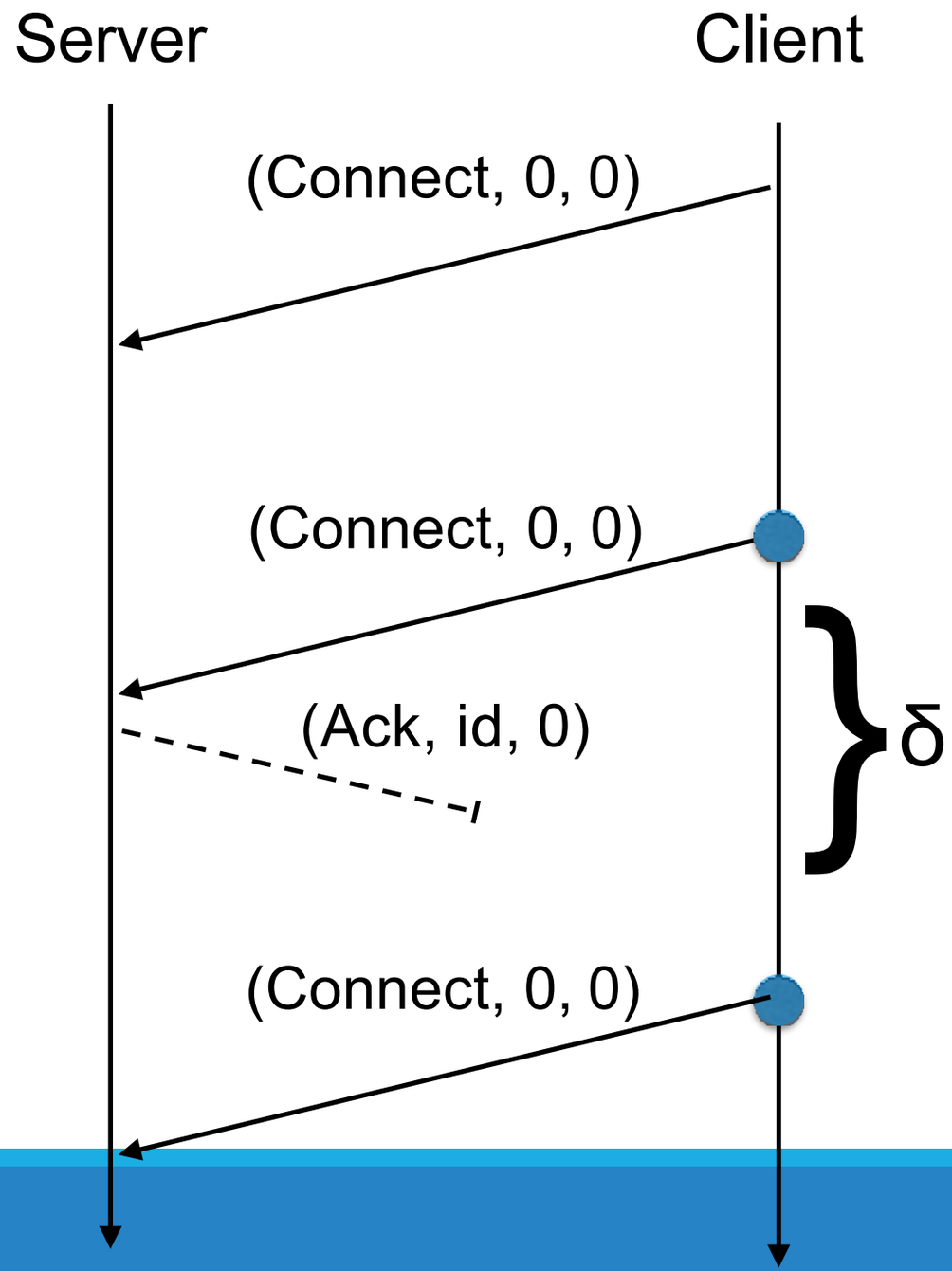
- Both payload size and checksum are used to verify data integrity.
- Payload Size (What if received data is shorter? longer?)
- Checksum
    - Carries more information than payload size
    - Can detect flipped bits introduced in the process of data transmission and storage
    - See writeup section 2.1.5 for detailed description of the 16-bit one’s complement sum algorithm

Epoch Events

- We still need to deal with dropped packets.
- On both the clients and servers, we have a simple time trigger to fire periodically.
- Time interval between two epochs (δ) is fixed.
- Clients and server take **epoch actions** in case of dropped packets or lost connection
- Epoch actions happen when timer trigger fires
- Data Message retransmission happen only when CurrentBackOff epochs have elapsed
- CurrentBackOff increases according to **exponential backoff rules**

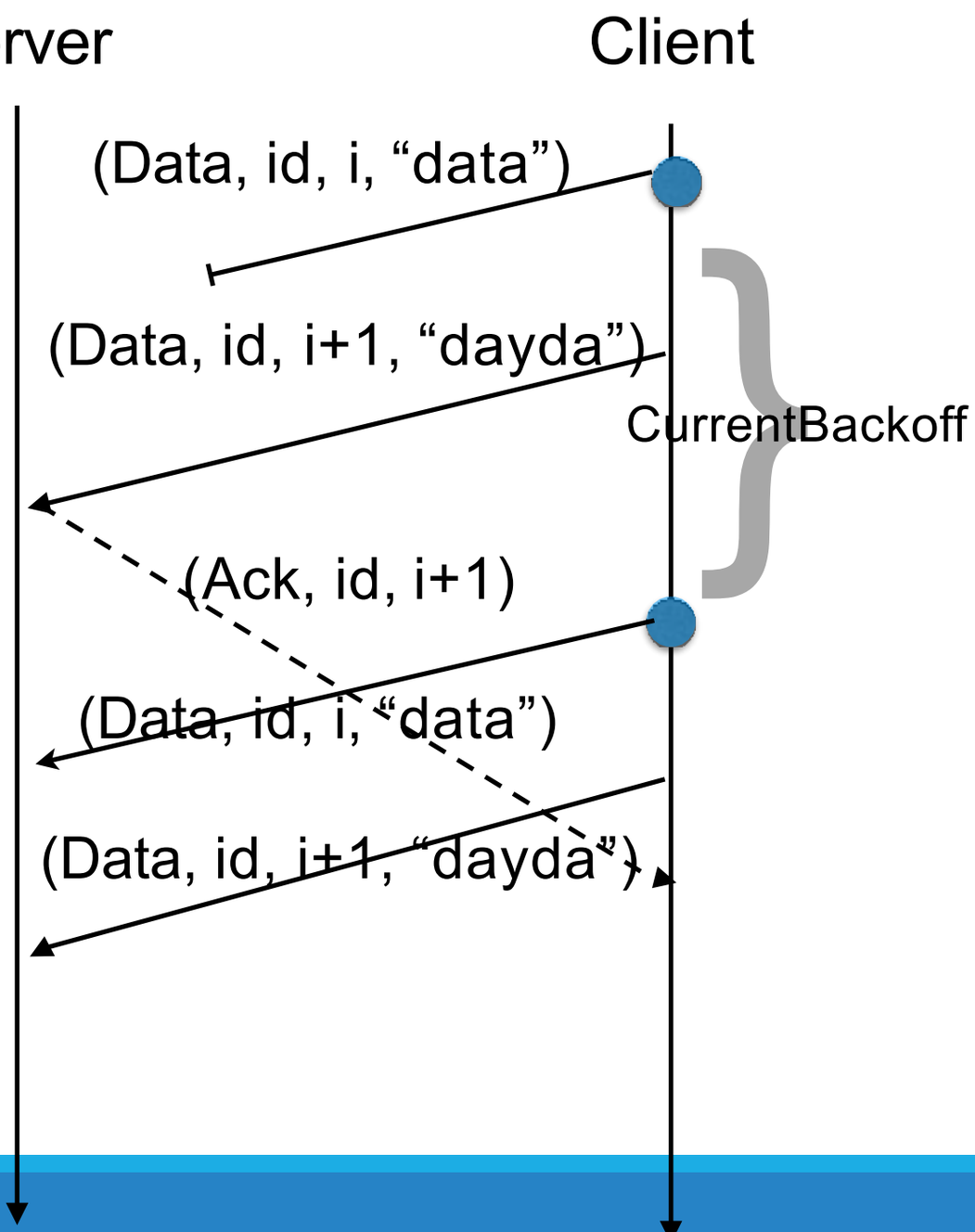
Client Epoch Actions: Connect Message Retransmission

- If connection request has not been acknowledged, resend connection request



Client Epoch Actions: Data Message Retransmission

- For every unacknowledged data message sent, resend the data message

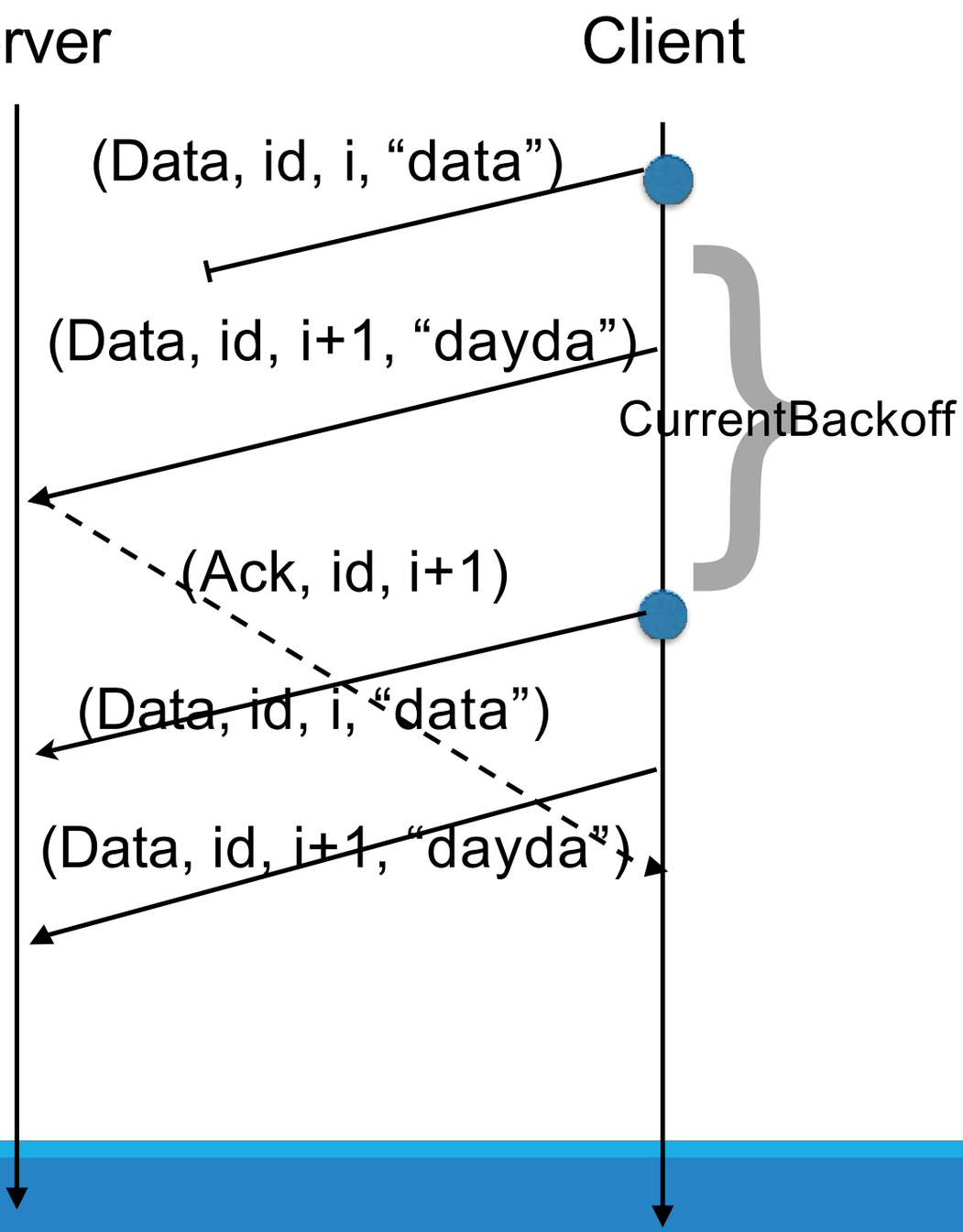


Note that message i+1 is duplicated on the server

Client Epoch Actions: Data Message Retransmission

- For every unacknowledged data message sent, resend the data message

Which message(s) to resend on each epoch?



Note that message i+1 is duplicated on the server

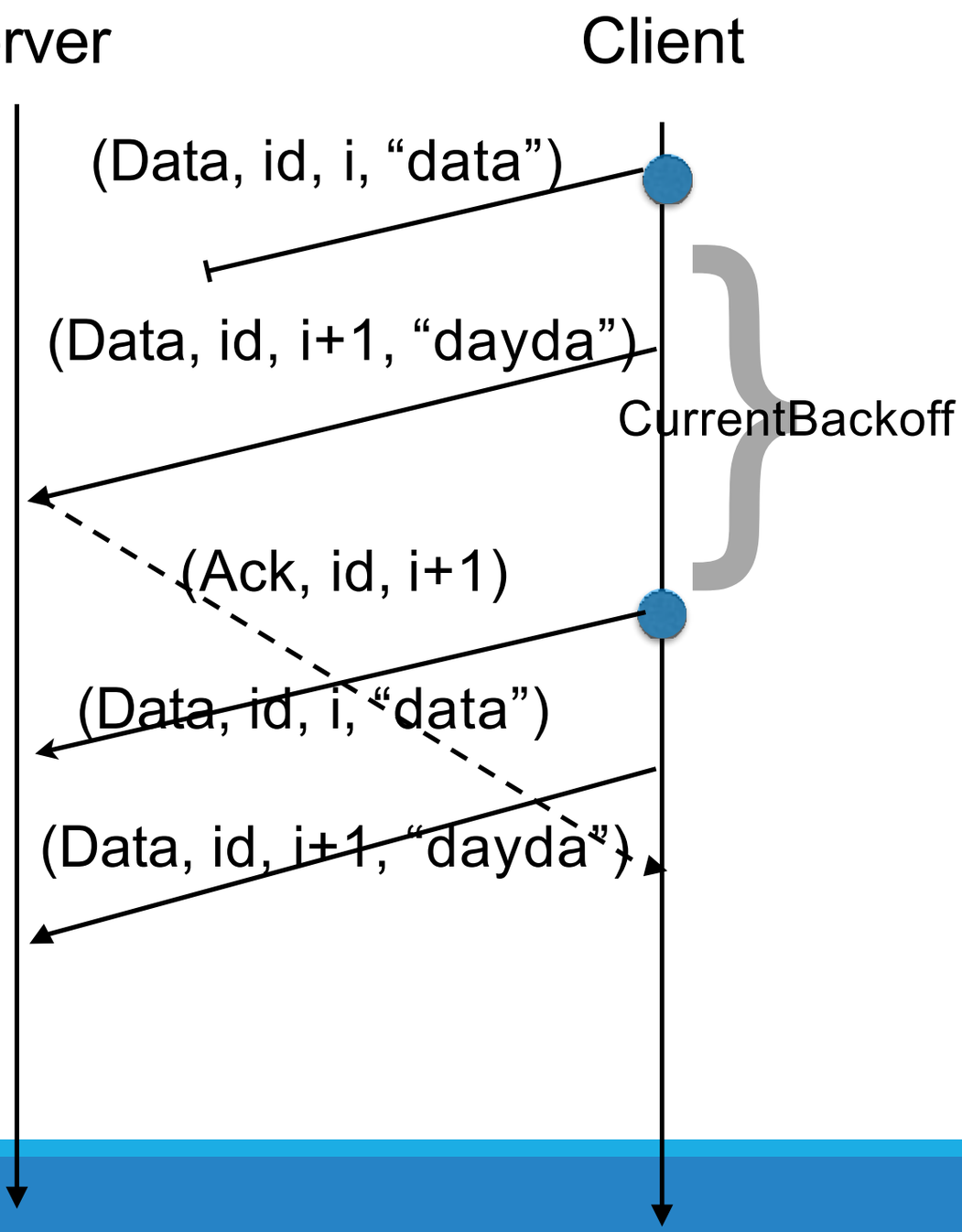
Client Epoch Actions: Data Message Retransmission

- For every unacknowledged data message sent, resend the data message

Which message(s) to resend on each epoch?

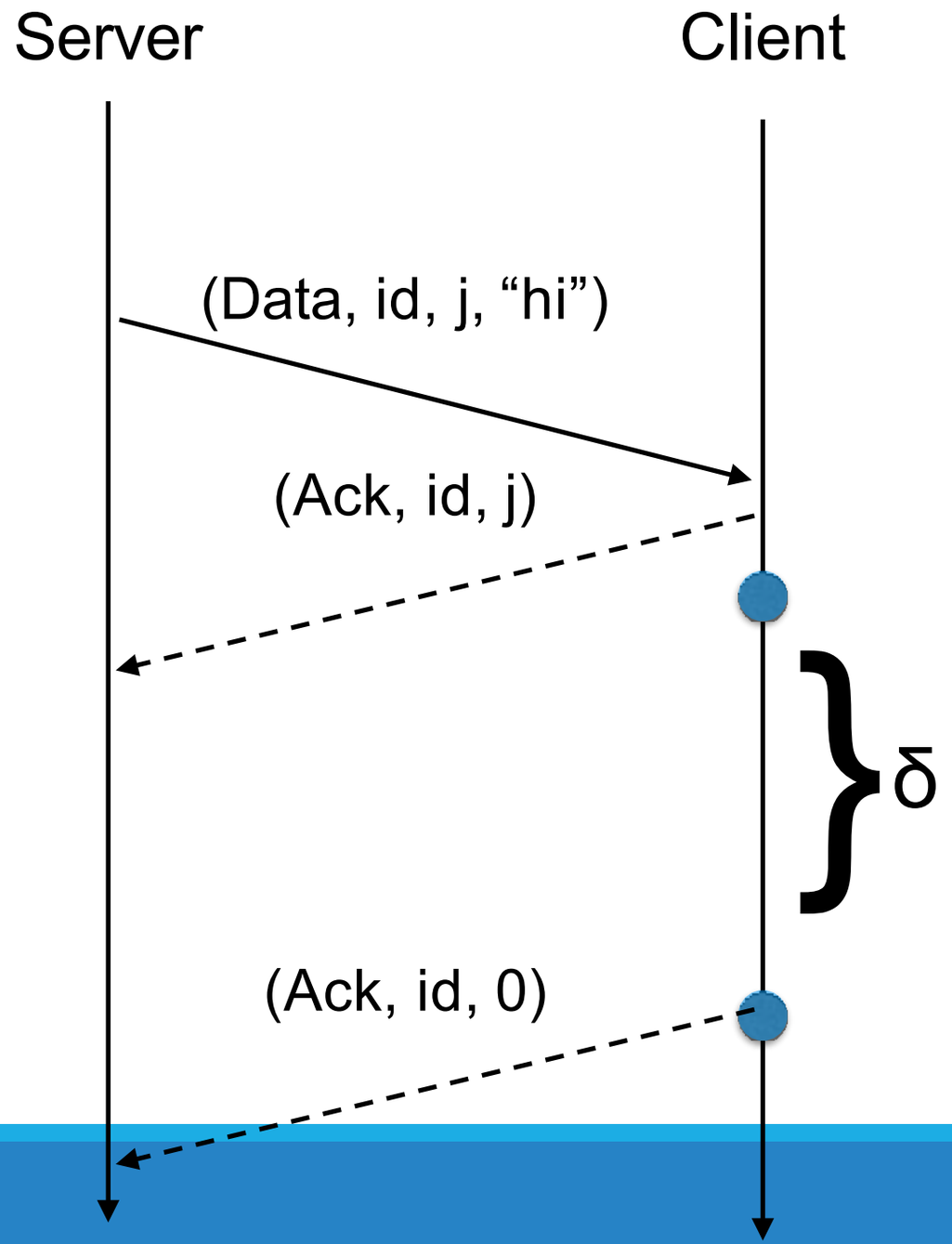
What should the server do if it receives a repeat?

Note that message i+1 is duplicated on the server



Client Epoch Actions: Is the connection dead?

- If the client:
  (1) has received Ack message for the Connect request;
  (2) has not received any Data message;
  Then it should send an ack with sequence number 0



Server epoch actions are very similar to client epoch actions.

- For each client connection:
  - For each data message that has been sent, but not yet acknowledged, resend the data message
  - If no data message has been received from the client, then send an ack with sequence number 0

Epoch Events: EpochLimit

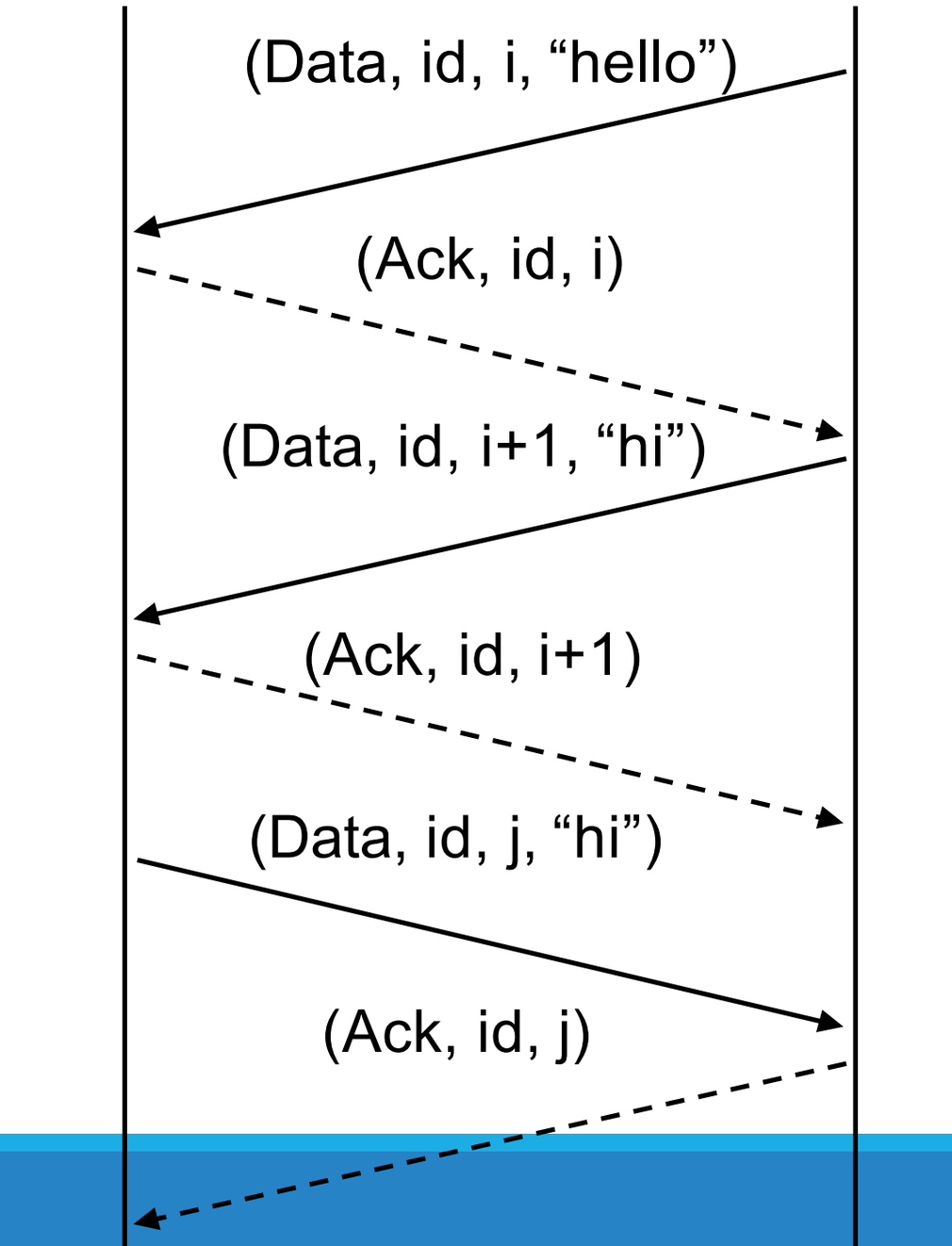
We can keep track of epochs passed since the last message was received. If this goes over a limit, we can assume the connection is lost.

Checkpoint (due 9/25)

- Assume no packet loss (no need to implement epoch)
- Race conditions will not be checked
- Messages might be sent out of order:
- **Need to receive messages in order**
- **Need to implement Sliding Window Protocol**

Checkpoint (due 9/25)

Simple Read/ Write Server

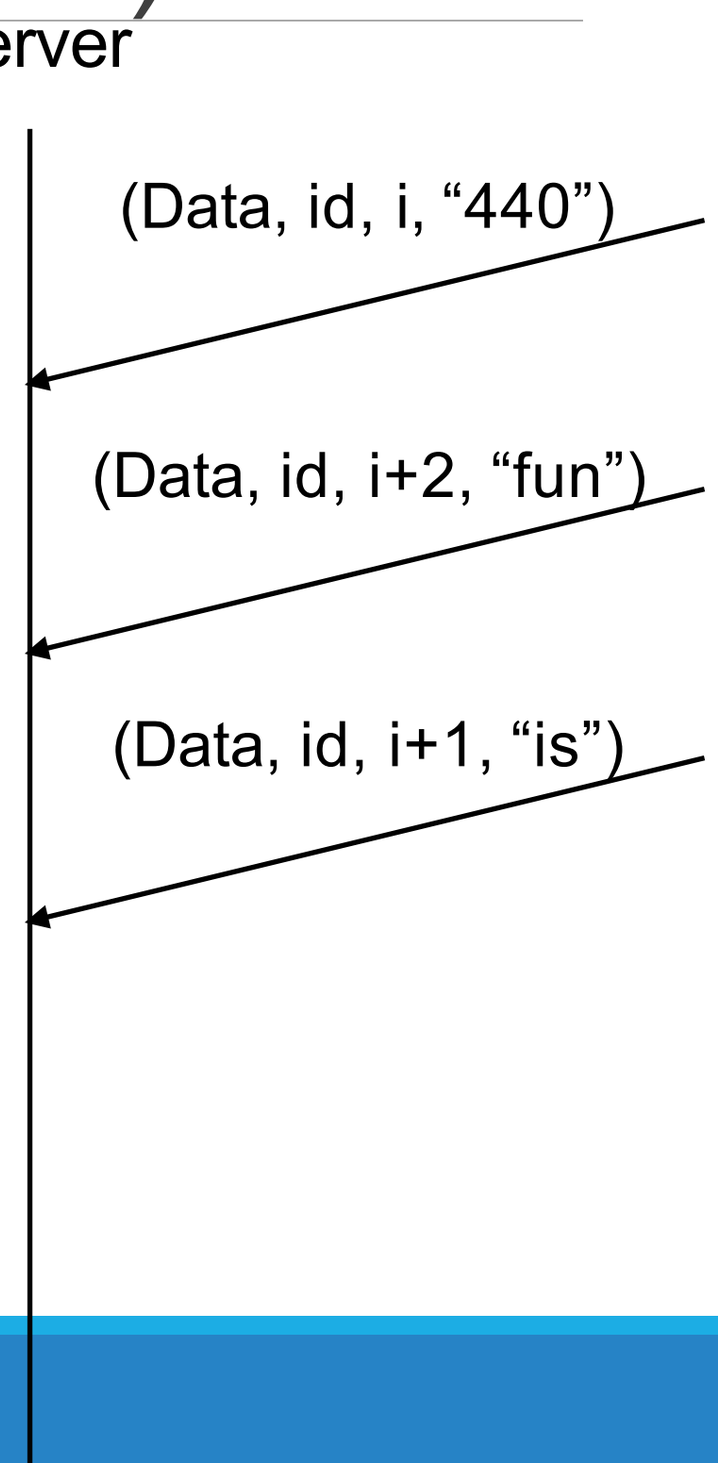


Checkpoint (due 9/25)

Simple Read/ Write Server

+

Receiving In Order



Checkpoint (due 9/25)

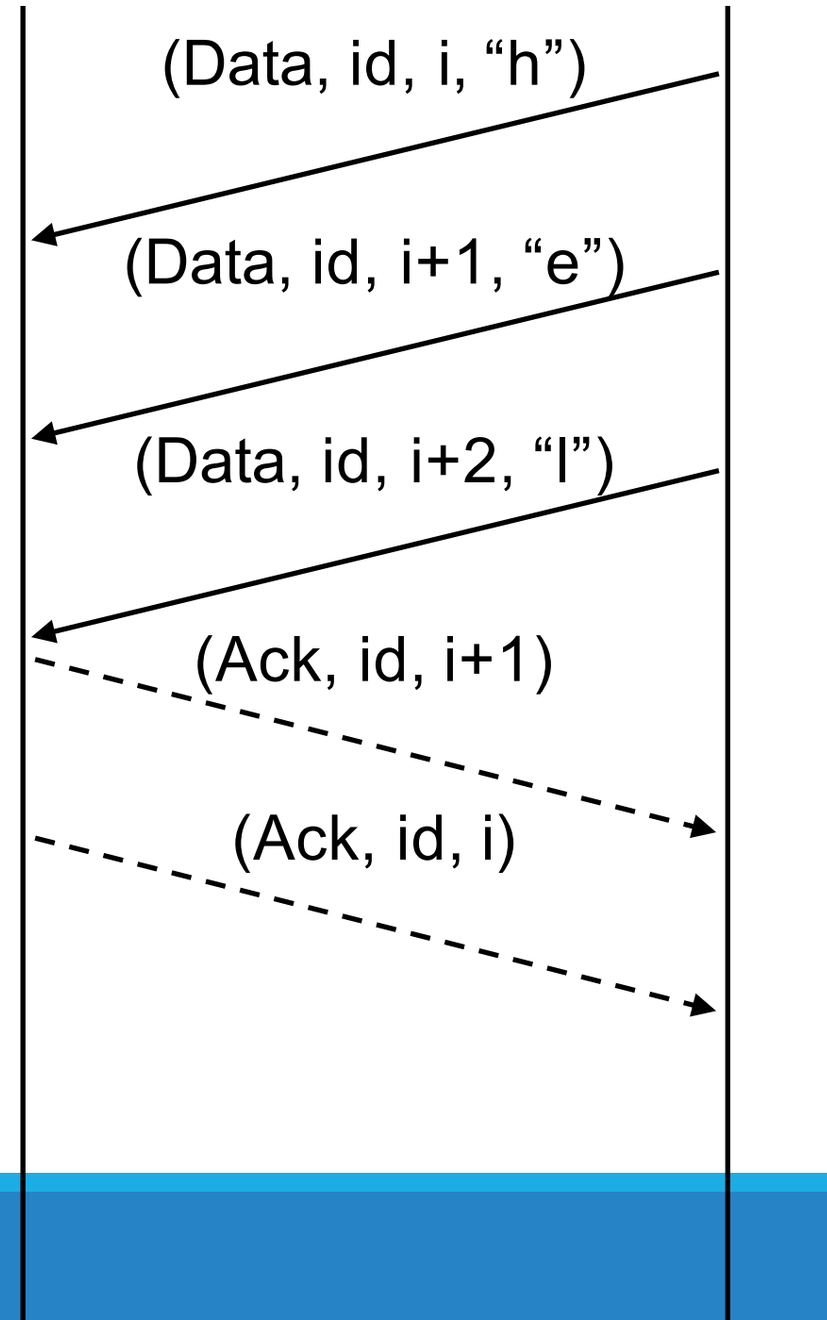
Simple Read/ Write Server

+

Receiving In Order

+

Sliding Window Protocol



lspnet

- Contains every UDP operation needed.
- **net** package is not allowed!

```
import “github.com/cmu440/lspnet”

addr, err := lspnet.ResolveUDPAddr("udp", hostport)
udpConn, err := lspnet.ListenUDP("udp", addr)
n, cliAddr, err := udpConn.ReadFromUDP(buffer[0]:)
udpConn.WriteToUDP(msg, cliAddr)
```

Implementation notes

- No locks and mutexes
- There’s no limit on message queue size, so don’t use buffered channel to store pending messages as in p0. Instead use something like linked list.