P1: Distributed Bitcoin Miner

15-440/15-640
Fall 2021

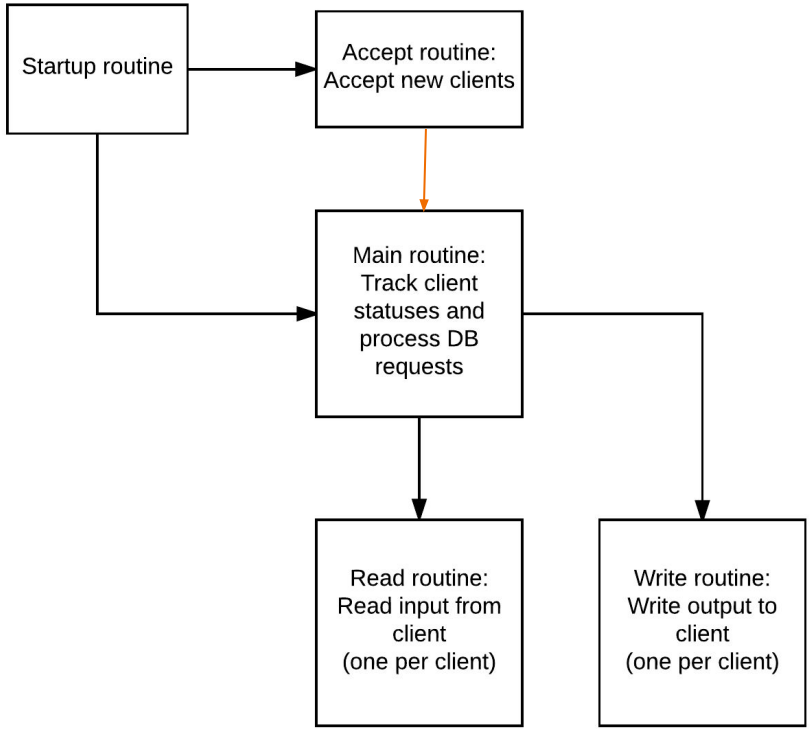
Overview

1. P0 Wrap Up
2. P1 Part A Introduction

P0 Overview

```
type keyValueServer struct {
    store              kvstore.KVStore
    listener           net.Listener
    currentClients     []*client
    newConnection      chan net.Conn
    dbQuery            chan *db
    ...
}

type client struct {
    connection         net.Conn
    messageQueue       chan []byte
    quitSignal_Read    chan int
    quitSignal_Write   chan int
}
```



Main Routine - all changes/updates happen here

1. Add a new client to the client list.
   a. go readRoutine(kvs, c)
   b. go writeRoutine(c)
2. Remove the dead client.
3. Run a query on the DB, all queries are directed to this channel.
   a. Process each case here
      i. **Update() -> slice**
4. Process CountActive() calls
5. Process CountDropped() calls
6. Process Close() calls

A Tour of Go

Slices are like references to arrays

A slice does not store any data, it just describes a section of an underlying array.

Changing the elements of a slice modifies the corresponding elements of its underlying array.

Other slices that share the same underlying array will see those changes.

Close function

- -1 Close not implemented. [Use this for no attempt at implementation.]
- -0.5 Close should signal goroutines to terminate. [Use this for if they call Close() on the socket but do nothing else.]
- -0.01 for minor close issues:
  - Did not close individual client connections -> `c.connection.Close()`
  - Go routines that may deadlock when trying to handle a close signal
  - Sending a quit message on a channel with multiple listeners, only one of whom will receive the message and actually quit.
    - **1 -> 1, good**
    - **Many -> 1, okay**
    - **1 -> many, DON'T DO THIS**

Using Channels As Mutexes

Case: count <- channel1:

    Count += 1

    channel1 <- count

Case: <- channel1:

    // Do your updates or read a field

    channel1 <- true

Using Channels For store things that can grow arbitrarily in size

Case: newConn <- newConnChan:

    activeCountChan <- 1

Return len(activeCountChan) for CountActive function.

P1 Logistics

- P1 is **HARD**, read the handout and start early!
- Deadlines:
  - Part A Checkpoint: Due Tuesday, 9/28 -> 20% (the easy 20%!)
  - Part A Final: Due Friday, 10/8 -> 60%
  - Part B: Due Thursday, 10/14 -> 20%
- Working with a Partner
- OH will be busy, so start early!
  - 10 min time limit
  - Tell us what you have tried

P1 Requirement

- **No locks and mutexes.**
  - **If you have lock-like behavior using channel, we will not help you debug your code.**
- **No buffered channels with size > 1.**
- **You cannot use `sync`, `sync.atomic`, or `net` packages.**

Part A: Live Sequence Protocol

- LSP is similar to TCP, it adds functionality to UDP
- LSP has some of its own 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 Re-transmission and Timeout Mechanism**

lspnet

- Contains every UDP operation needed
- **net** package is not allowed for this project. Use **lspnet**

```
import “github.com/cmu440/lspnet”

addr, err := lspnet.ResolveUDPAddr("udp", hostport)
udpConn, err := lspnet.ListenUDP("udp", addr)

n, cliAddr, err := udpConn.ReadFromUDP(buffer)

udpConn.WriteToUDP(msg, cliAddr)
```

Messages

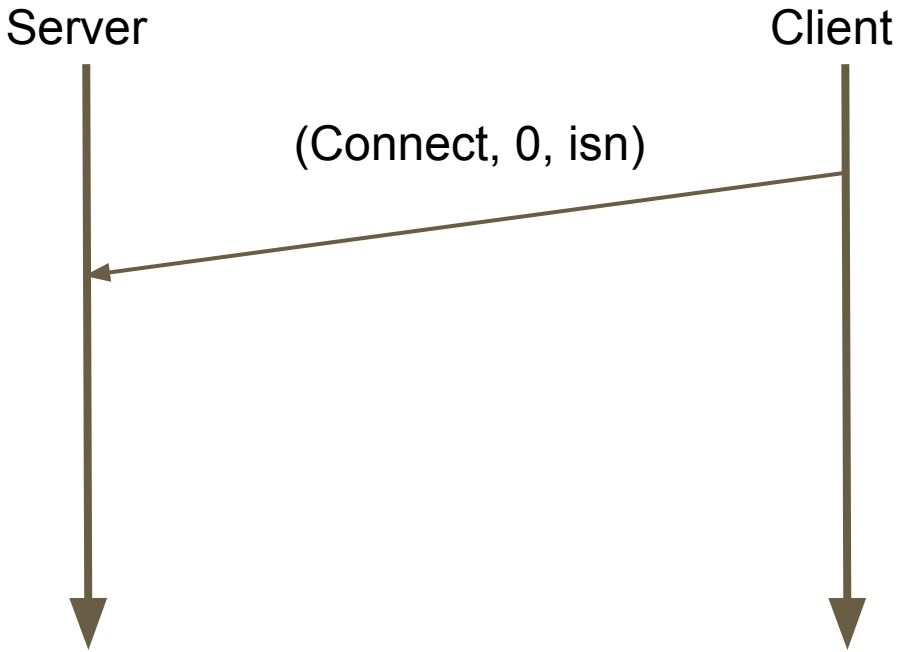
Each message is consists of:

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

Message

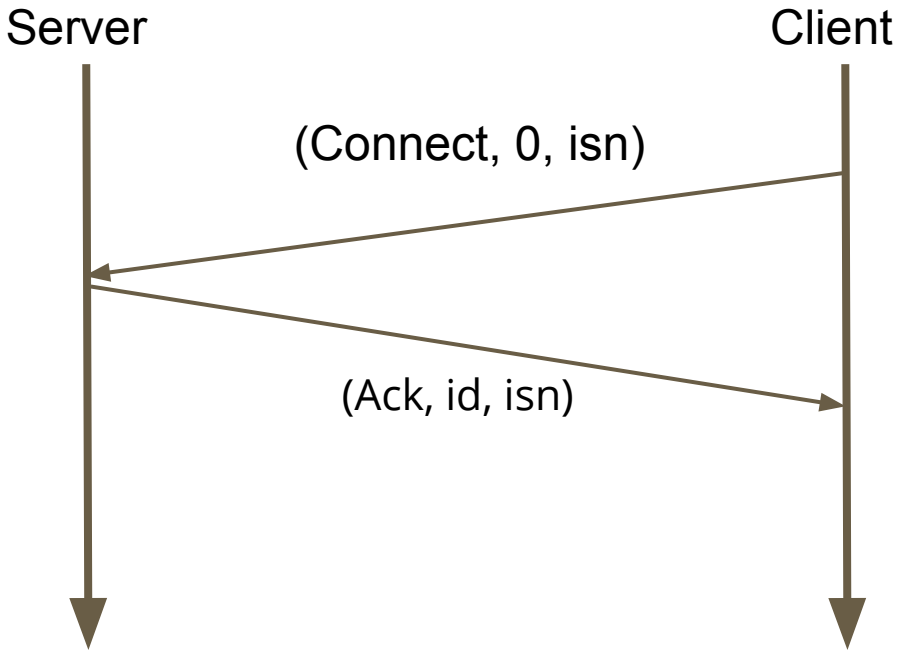
- Message size is limited to single UDP-packet size (~ 1000 bytes)
- Each Message is received exactly once (ignore duplicates)
- Messages are marshaled using Go’s Marshal function in the json package and sent as a UDP packet

Client Server Communication: Establish a Connection



Client begins by sending a connection message (must have ID = 0, and a given initial sequence number)

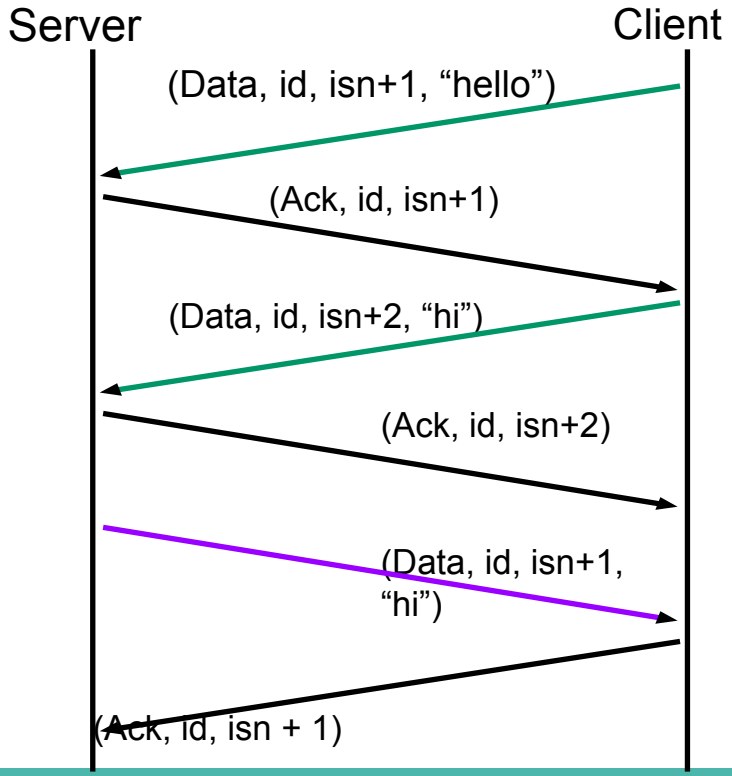
Client Server Communication: Establish a Connection



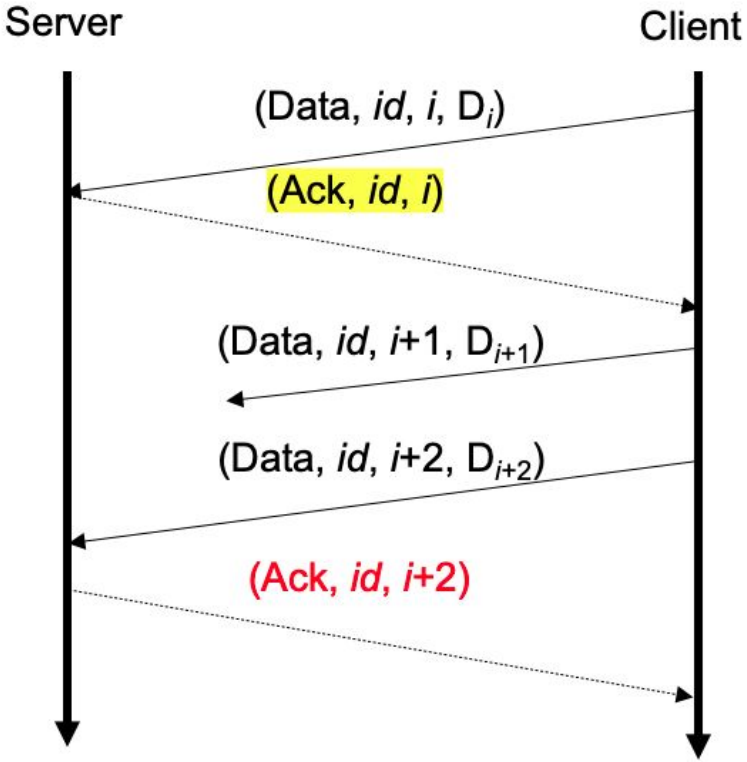
Server generates a unique connection ID for this client-server connection
(you can generate ID's sequentially)

Client Server Communication: Sending & Ack-ing Data

- In this example, assume server and clients just established the connection and started to send data messages, starting from isn:
  - Server and Client maintain independent sequence numbers.
  - Since all messages are received in order, we can ack with either Ack or CAck.



Client Server Communication: Ack & CAck



- Since all previous messages have been received and processed, the corresponding acknowledgement (highlighted with yellow) can be either Ack or CAck.
- Client then sends data messages i + 1 and i + 2, but only i + 2 is received. The server still needs to acknowledge data message i + 2, but this time only Ack should be used.
- Your implementation does not necessarily need to send CAck, but **both client and server should be able to handle CAck.**

Received Messages Must Be Processed In Order.

UDP Packets are not guaranteed to arrive in order.

```
LSPServer.Read() //Blocks

LSPServer.Read()

LSPServer.Read()
```

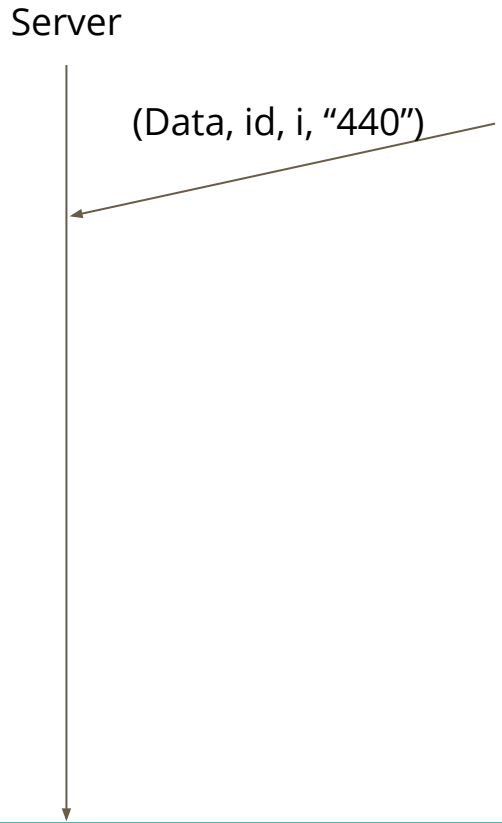
Received Messages Must Be Processed In Order.

UDP Packets are not guaranteed to arrive in order.

```
LSPServer.Read() //Returns “440”

LSPServer.Read() //Blocks

LSPServer.Read()
```



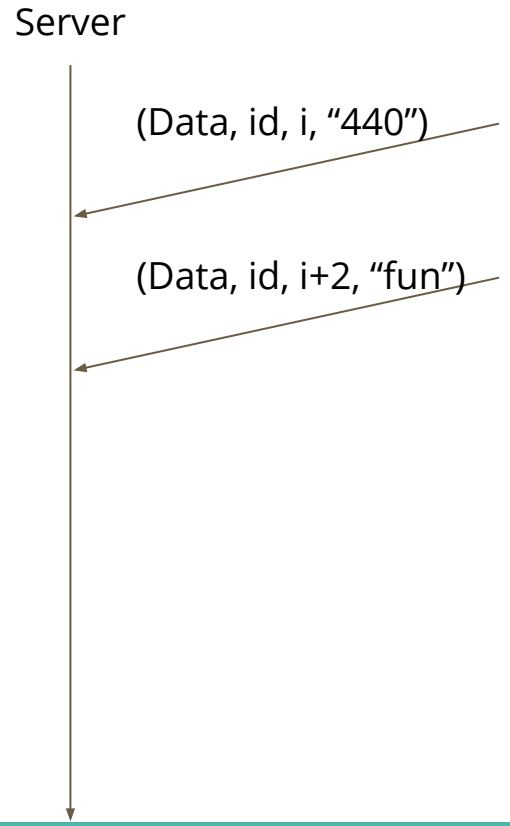
Received Messages Must Be Processed In Order.

UDP Packets are not guaranteed to arrive in order.

```
LSPServer.Read() //Returns “440”

LSPServer.Read() //Blocks

LSPServer.Read() //Blocks
```



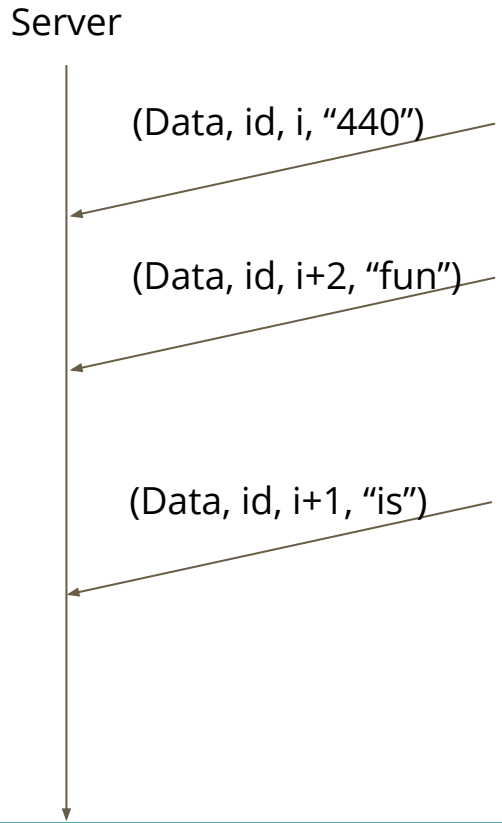
Received Messages Must Be Processed In Order.

UDP Packets are not guaranteed to arrive in order.

```
LSPServer.Read() //Returns “440”

LSPServer.Read() //Returns “is”

LSPServer.Read() //Returns “fun”
```



Checkpoint (Due 9/28)

- Assume no packet loss (no need to implement window, epoch, retry)
- You need to implement:
  - Interaction between Server & Client
  - Receiving In Order
  - Simple Read & Write
    - Only reads and writes data messages
    - Please read the handout and api files carefully!
- This is the easy 20% of the project.

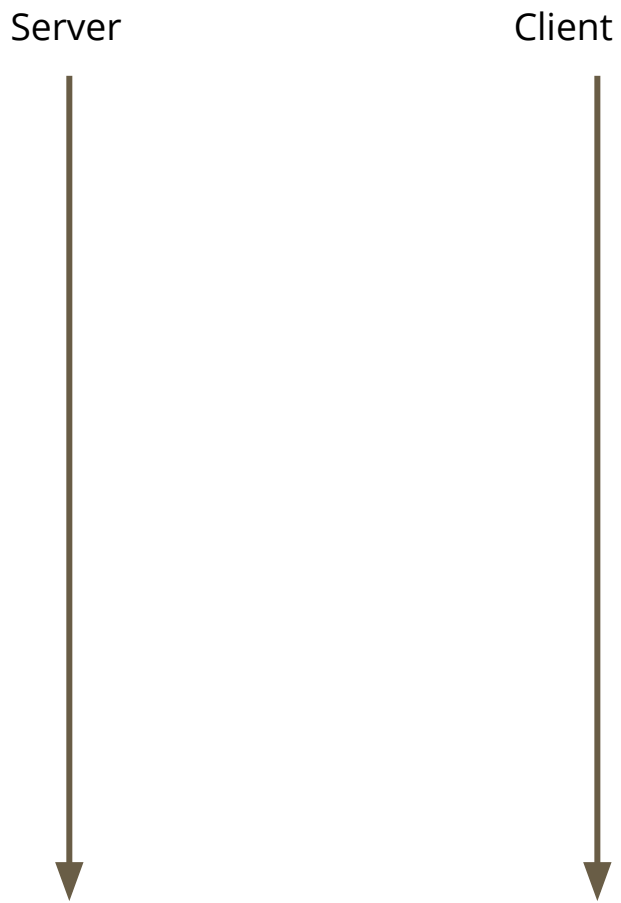
Sliding Window Protocol

- Given a window size `w`, we can send up to `w` messages without acknowledgement.
- If the oldest unacknowledged message has sequence number n, then only messages with sequence numbers `n + w - 1` (inclusive) may be sent i.e. `[n, n+ w -1]`
- In addition, number of unacknowledged messages cannot exceed **MaxUnackedMessages**

Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “H” -> “E” -> “L” -> “L” -> “O”
- Oldest SN without Ack = i

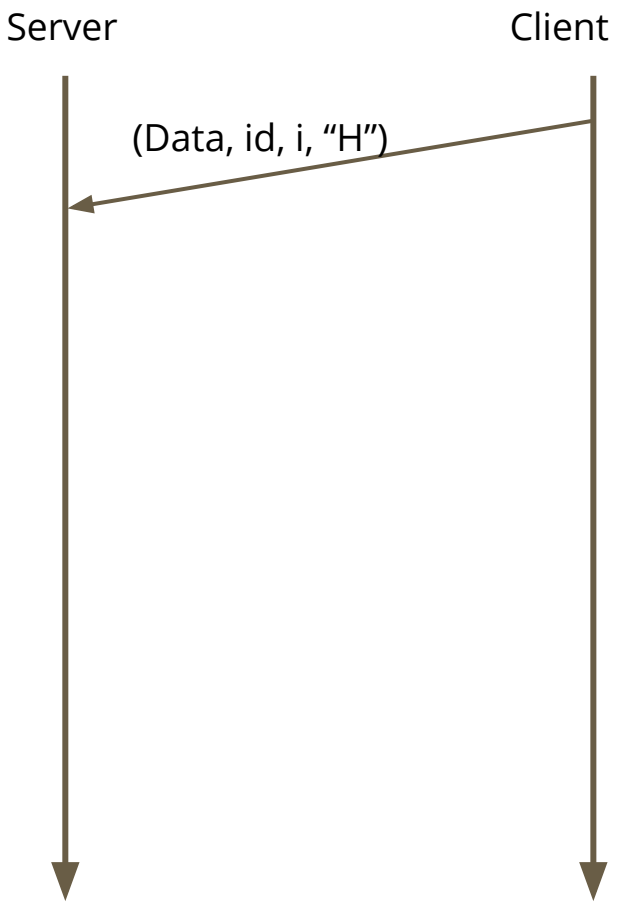
  Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “E” -> “L” -> “L” -> “O”
- Oldest SN without Ack = i

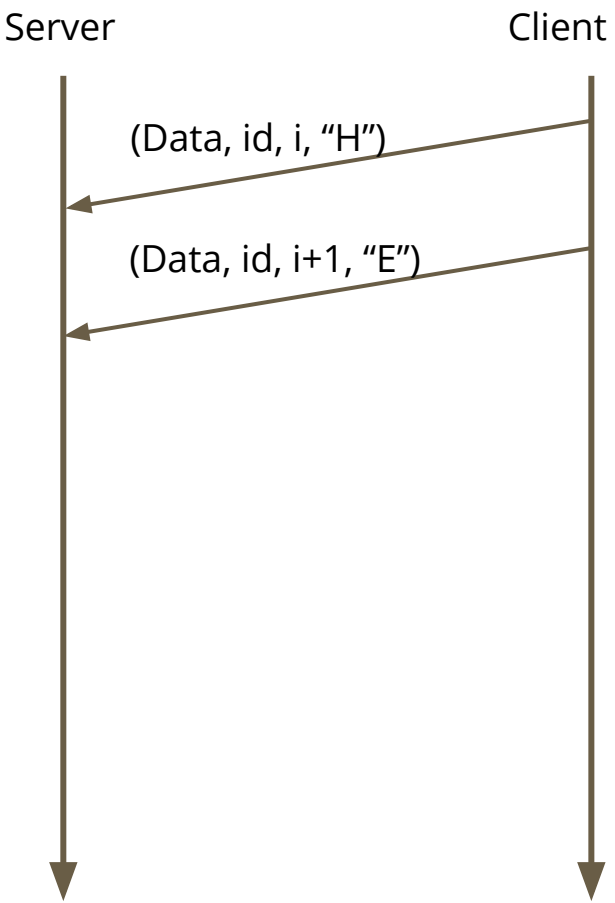
  Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “L” -> “L” -> “O”
- Oldest SN without Ack = i

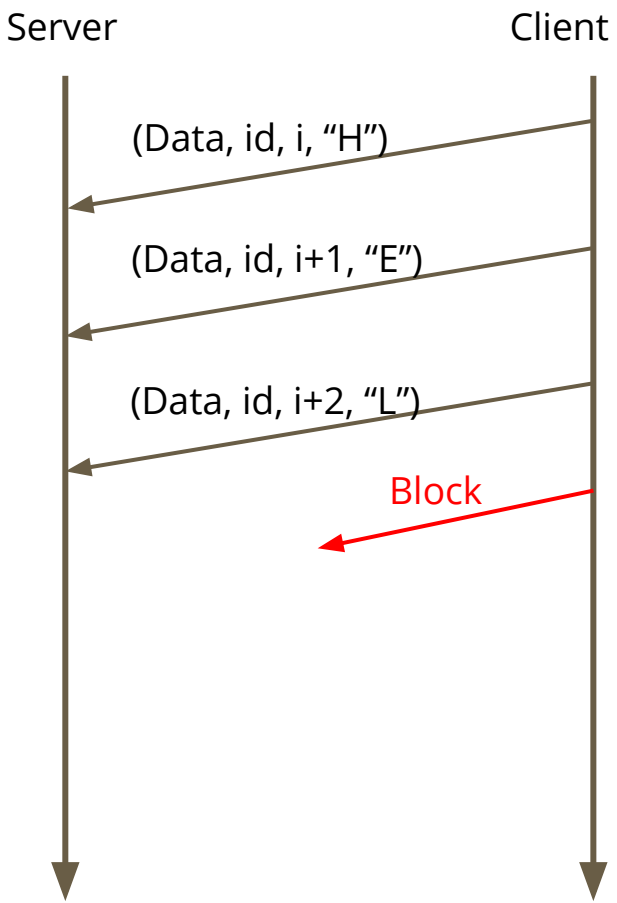
  Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “L” -> “O”
- Oldest SN without Ack = i

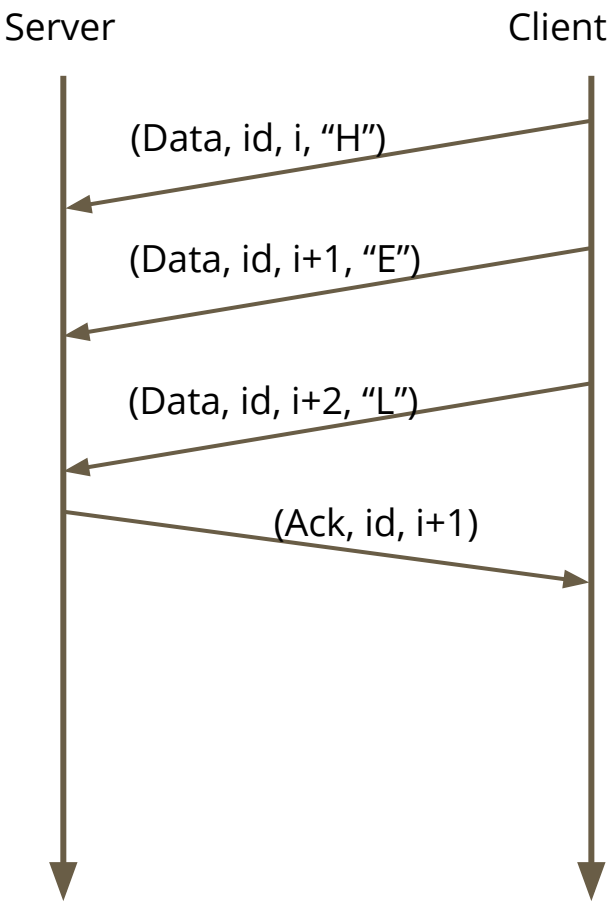
Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “L” -> “O”
- Oldest SN without Ack = i

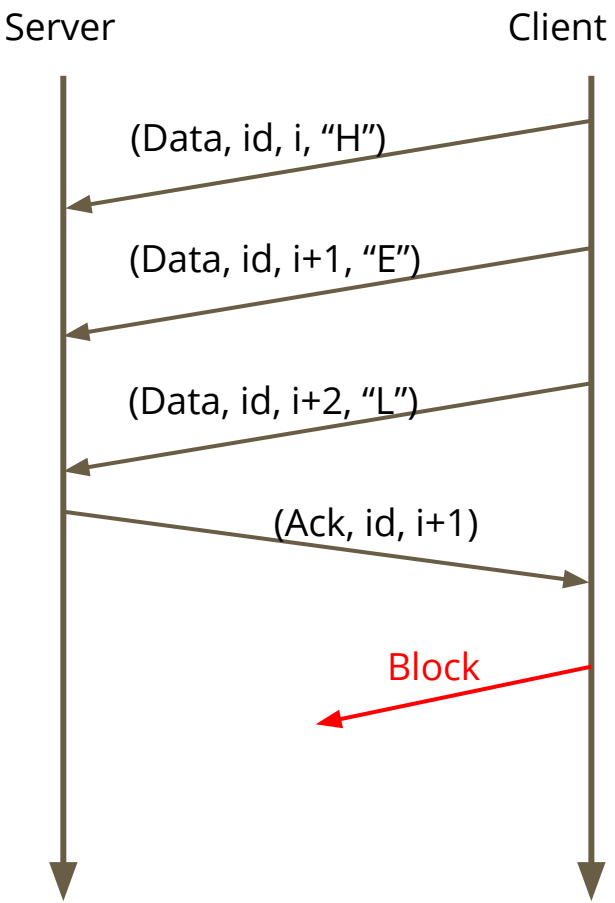
Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “L” -> “O”
- Oldest SN without Ack = i

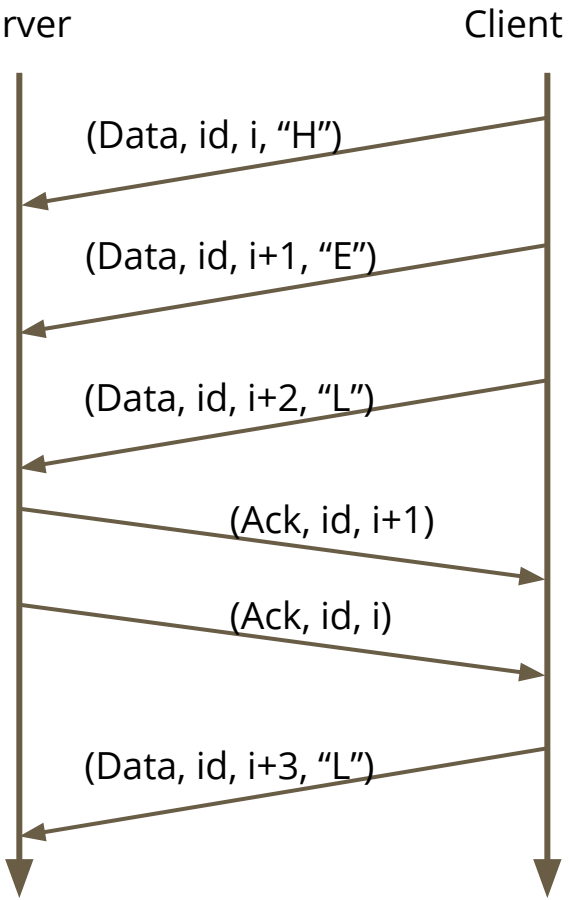
Window = [i, i+2]



Sliding Window Protocol Example1

- W = 3
- Client messages queue:
  “O”
- Oldest SN without Ack = i + 2

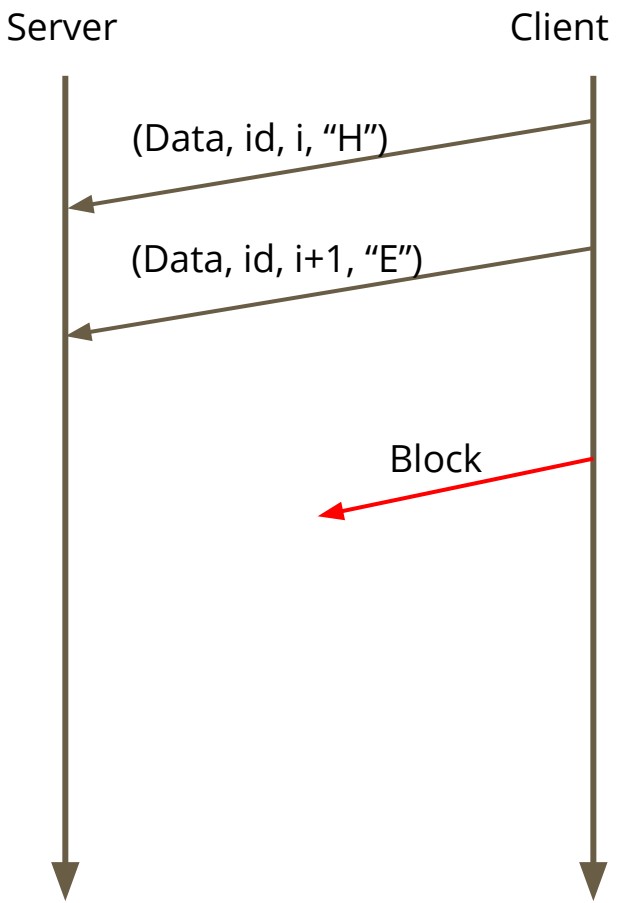
  Old Window = [i, i+2]
  New Window = [i+2, i+4]



Sliding Window Protocol Example2

- W = 3
- **MaxUnackedMessages = 2**
- Client messages queue:
  “L” -> “L” -> “O”
- Oldest SN without Ack = i

  Window = [i, i+2]



Payload Size & Checksum

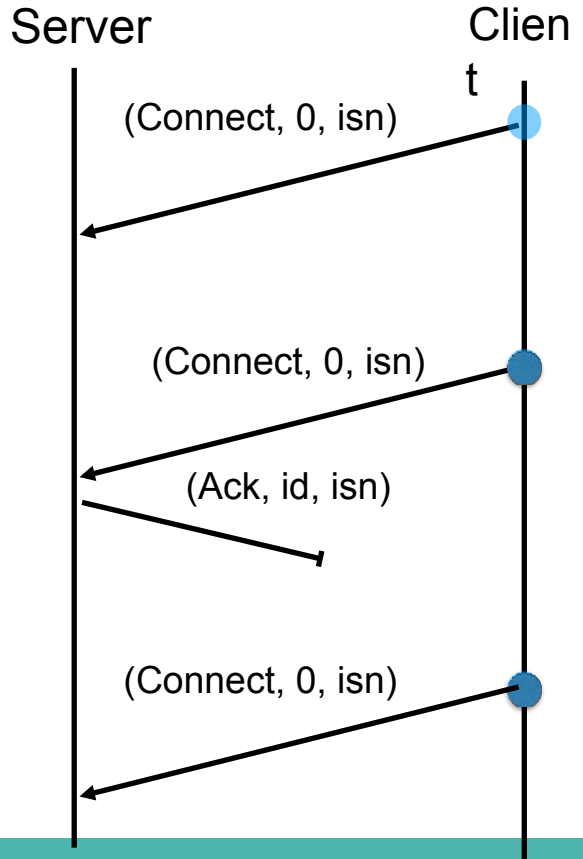
- Both payload size and checksum are used to verify data integrity.
- Payload Size (What if received data is shorter? longer?)
- Checksum
  - Carries more information than payload size
  - Can detect flipped bits introduced in the process of data transmission and storage
  - See writeup for detailed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

Epoch Events

- **We need to deal with dropped packets + check if connection is live**
- **Time interval between two epochs (t) is fixed.**
- **Clients and server take epoch actions when a periodic timer trigger fires.**
- *For every data message that isn't acknowledged yet, resend following* ***the exponential backoff rules*** *(0 -> 1 -> 2 -> 4)*
- ***CurrentBackoff*** *- the amount of epochs we wait before re-transmitting data that did not receive an ACK.*
- ***CurrentBackOff*** *increases according to* ***exponential backoff rules****, until it reaches* ***MaxBackOff***
- *If sent/resent nothing in the past epoch, send a* ***Heartbeat*** *message — keeps the connection alive*

Client Epoch Actions: Connect Message Retransmission

If connection request has not been acknowledges, **resend connection request every epoch.**

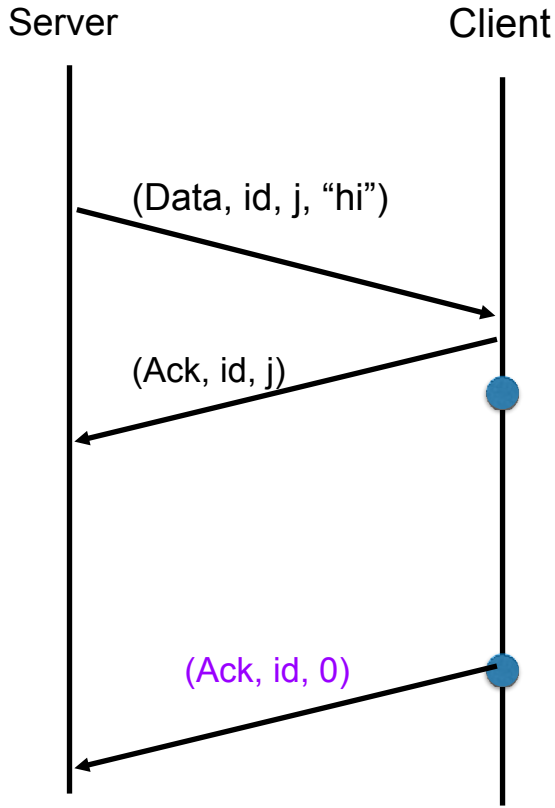


Client Epoch Actions: Is the connection dead?

If the client:

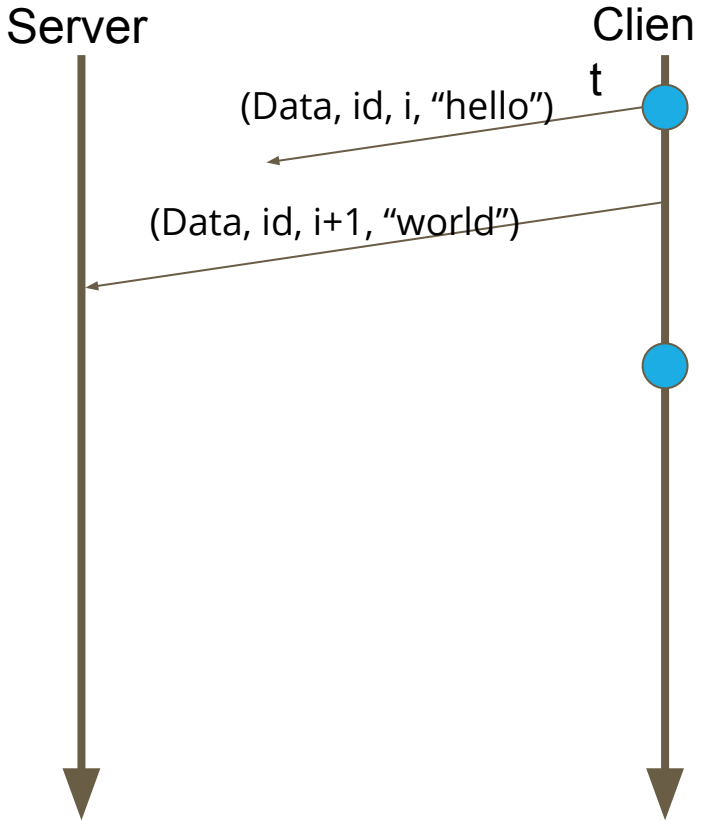
1. Has received Ack for the Connect request
2. Has **NOT** received any Data message

Then it should send a **heartbeat** message (Ack with sequence number 0)



Client Epoch Actions: Data Message Retransmission 1

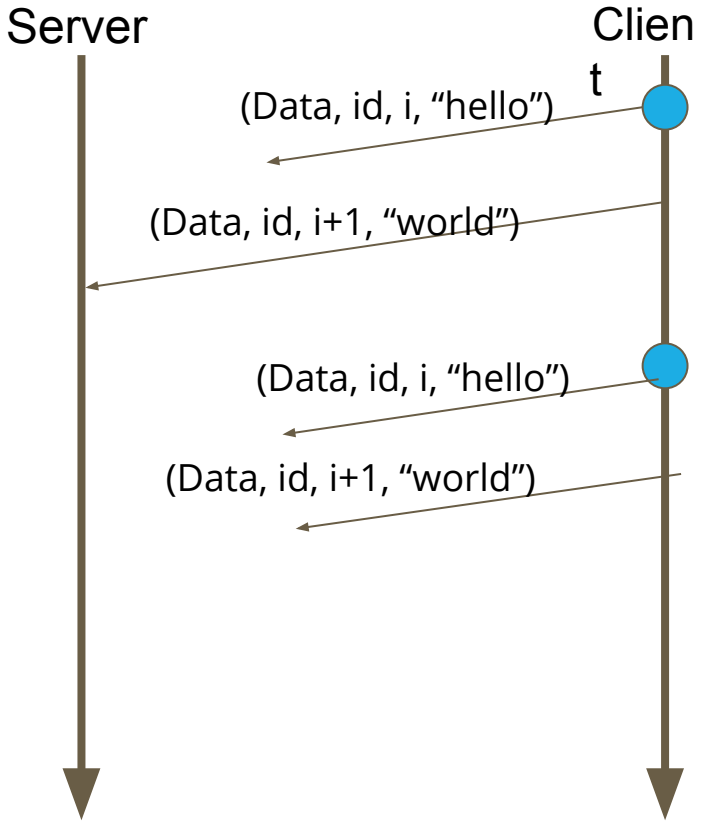
For every unacknowledged data message sent, resend the data message.



Client Epoch Actions: Data Message Retransmission 1

For every unacknowledged data message sent, resend the data message after **CurrentBackOff**.

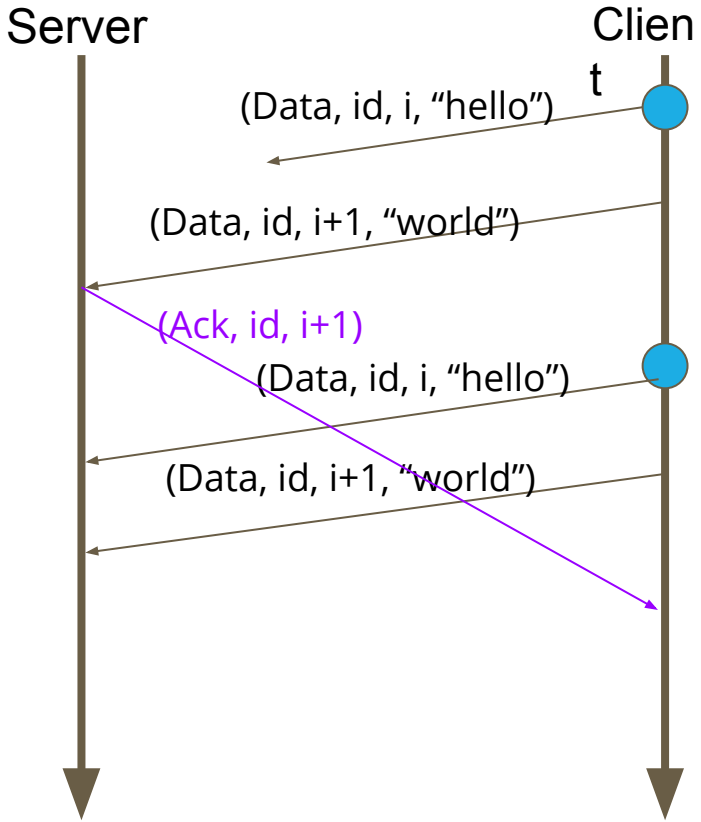
Note: CurrentBackOff = 0 in this example.



Client Epoch Actions: Data Message Retransmission 2

For every unacknowledged data message sent, resend the data message after **CurrentBackOff**.

Note: CurrentBackOff = 0 in this example.



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 following the exponential backoff rules above.
- If the server has not sent or resent any data message to the client in the last epoch, then send a **Heartbeat** message (i.e., 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.

Read(), Write(), Close(), CloseConn()

- We don’t have time to go thru these during recitation, please go over the handout, Server.api.go, and Client.api.go
- Note that Close() is blocking while CloseConn() is not
- Close() ensures that all pending messages to send are sent and acknowledged before Close() returns
- If Read(), Write, Close(), or CloseConn() is called after Close() is called, they must either return an error, or never return anything