Project 1: Distributed Bitcoin Miner

1 Overview

This project will consist of the following two parts:

- **Part A:** Implement the Live Sequence Protocol, a homegrown protocol for providing reliable communication with simple client and server APIs on top of the Internet UDP protocol.
- **Part B:** Implement a Distributed Bitcoin Miner.

Schedule:

Part A Checkpoint:	Due: **Tuesday 9/29**,	20% of project grade
Full Project Part A:	Due: **Saturday 10/10**,	60% of project grade
Full Project Part B:	Due: **Tuesday 10/20**,	20% of project grade

You will have 15 submissions for each due date. The same policy of P0 will apply here as well - at most 2 late days are automatically granted for every assignment for **valid reason** with no late penalty. No submissions will be accepted 2 days after each deadline. Your Gradescope submission will output a message showing how many submissions you have made and how many you have left. Gradescope will allow you to submit beyond this limit, but we will be checking manually. **Only your last submission counts.**

Gradescope will count how many submissions are under your name, and help you calculate number of remaining submissions. To better keep track of number of submissions of your group, please add your group mate as a member for EACH submission. We won't accept request to update scores if you miscount the number of submissions of your group.

For Part A Checkpoint, there will be no manual style grading. However, both Part A final and Part B final will have 4 points for manual style grading. Specifically, we are looking for good function headers, comments for variables, and no debugging print statements or chunks of dead code. Also note that you will get points deducted if you use mutexes in your implementation.

For this project you will have one partner of your choosing. In the case you do not have one, a partner will be provided to you. Only one member of each group will need to submit

to Gradescope (check the README for submission guidelines), and after submitting, make sure to add your partner under Group Members.

We will begin by discussing part A of this project, in which you will implement the *Live Sequence Protocol*. In your implementation, you will incorporate features required to create a robust system, handling lost, duplicated, or corrupted Internet packets, as well as failing clients and servers. You will also learn the value of creating a set of layered abstractions in bridging between low-level network protocols and high-level applications.

WARNING: P1 is much more complex than P0, so it is imperative that you start early with a solid design in place. We give you an outline of what this can look like in this handout.

Also, although you have approximately 50% of the time of Part A for the checkpoint, the checkpoint is much less than 50% of the features, therefore it would benefit you to go beyond what we require you to do.

2 Part A: LSP Protocol

The low-level Internet Protocol (IP) provides what is referred to as an "unreliable datagram" service, allowing one machine to send a message as a packet to another, but with the possibility that the packet will be delayed, dropped, or duplicated. In addition, as an IP packet hops from one network node to another, its size is limited to a specified maximum number of bytes. Typically, packets of up to $1,500$ bytes can safely be transmitted along any routing path, but going beyond this can become problematic.

Very few applications use IP directly. Instead, they are typically written to use UDP or TCP:

UDP: The "User Datagram Protocol." Also an unreliable datagram service, but it allows packets to be directed to different logical destinations on a single machine, known as *ports*. This makes it possible to run multiple clients or servers on a single machine. This function is often called multiplexing.

TCP: The "Transmission Control Protocol" offers a reliable, in-order stream abstraction. Using TCP, a stream of arbitrary-length messages is transmitted by breaking each message into (possibly multiple) packets at the source and then reassembling them at the destination. TCP handles such issues as dropped packets, duplicated packets, and preventing the sender from overwhelming both Internet bandwidth and the buffering capabilities at the destination.

Your task for Part A of this project is to implement the *Live Sequence Protocol* (LSP). LSP provides features that lie somewhere between UDP and TCP, but it also has features not found in either protocol:

- Unlike UDP or TCP, it supports a *client-server communication model*.
- The server maintains *connections* between a number of clients, each of which is identified by a numeric connection identifier.
- Communication between the server and a client consists of a sequence of discrete messages in each direction.
- Message sizes are limited to fit within single UDP packets (around 1,000 bytes).
- Messages are sent *reliably*: a message sent over the network must be received exactly once, and messages must be received in the same order they were sent.
- Message *integrity* is ensured: a message sent over the network will be rejected if modified in transit.
- The server and clients monitor the status of their connections and detect when the other side has become disconnected.

The following sections will describe LSP in-depth. You will only implement part of LSP for the Part A Checkpoint. We begin by describing the protocol in terms of the messages that flow between the server and its clients.

2.1 LSP Overview

In LSP, communication between a server and client consists of a sequence of discrete messages sent in each direction. Each message sent over an LSP connection is made up of the following six values:

Message Type: An integer constant identifying one of three possible message types:

Connect: Sent by a client to establish a connection with the server.

Data: Sent by either a client or the server to transmit information.

Ack: Sent by either a client or the server to acknowledge a Connect or Data message.

Connection ID: A positive, non-zero number that uniquely identifies the client-server connection.

Sequence Number: A positive, non-zero number that is incremented with each data message sent, starting with the number 0 for the initial connection request.

Payload Size: The number of bytes of the payload.

Checksum: A 16-bit number derived from the message to verify data integrity.

Payload: A sequence of bytes, with a format determined by the application.

In the sections that follow, we will use the following notation to indicate the different possible messages that can be sent between a client and a server (note that both Connect and Ack messages have payload values of `nil`):

- (Connect, $0, 0$): Connection request. It must have ID 0 and sequence number 0.
- (Data, id, sn, len, cs, D): Data message with ID id, sequence number sn, payload size len, checksum len and payload D.
- (Ack, id, sn): Ack message with ID id, and sequence number sn.

Note, when reading sections **2.1.1** and **2.1.2**, assume no messages are dropped.

2.1.1 Establishing a connection

Before data can be sent between a client and server, a connection must first be made. The client initiates a connection by sending a *connection request* to the server. In response, the server generates and assigns an ID that uniquely identifies the new client-server connection, and sends the client an acknowledgment message containing this new ID, the sequence number 0, and a `nil` payload. Figure 1 illustrates how a connection is established.

Your server may choose any scheme for generating new connection IDs. Our implementation simply assigns IDs sequentially starting with 1.

2.1.2 Sending & acknowledging data

Once a connection is established, data may be sent in both directions (i.e. from client to server or from server to client) as sequences of discrete *data messages*. Figure 2 gives an example of a normal communication sequence between the server and a client. The figure illustrates the transmission of three data messages from the client to the server, and one data message from the server to the client. Note that all messages are acknowledged, and

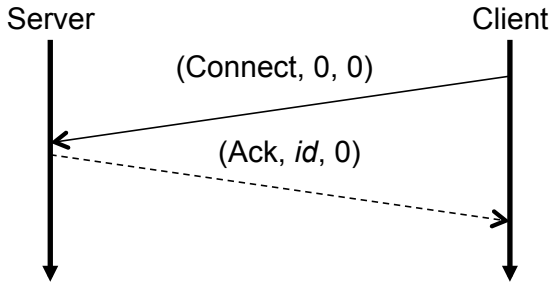


Figure 1: Establishing a connection. A client sends a connection request to the server, which responds to the client with an acknowledgment message containing the connection's unique ID. The vertical lines with downward arrows denote the passage of time on both the client and the server, while the lines crossing horizontally denote messages being sent between the two.

that the client and server maintain their own series of sequence numbers, independent of the other.

Also, note that it is entirely possible for one side to receive data messages while waiting for acknowledgments for data messages it sent—the client and server operate asynchronously, and there is no guarantee that packets arrive in the same order they are sent. However, it is still the responsibility of the client and server to *process* the received messages in order. Messages must be acknowledged when received, but processing cannot occur out of order. For example, if the server has not yet received a message with sequence number 5, but received a message with sequence number 6, it must store message 6 until it receives message 5.

Like TCP, LSP includes a *sliding window protocol*. The sliding window represents an upper bound on the range of messages that can be sent without acknowledgment. This upper bound is referred to as the "window size," which we denote with the symbol ω. For example, if $\omega = 1$, every message must be acknowledged before the next message can be sent. If $\omega = 2$, up to two messages can be sent without acknowledgment. That is, a third message cannot be sent until the first message is acknowledged, a fourth message cannot be sent until the first and second messages are acknowledged, etc.

Note that the range of messages that are allowed to be sent without acknowledgment is fixed. If the oldest message that has not yet been acknowledged has sequence number n, then only messages with sequence numbers n through $n + \omega - 1$ (inclusive) may be sent. Only when the oldest (leftmost) messages are acknowledged can the window slide over to the right, possibly allowing more recent data messages to be sent.

To simulate real world scenarios where the receiving end has storage limitations, we place a further restriction called `MaxUnackedMessages`, which we denote with the symbol M. At any point in time, the number of unacknowledged messages should not be larger than

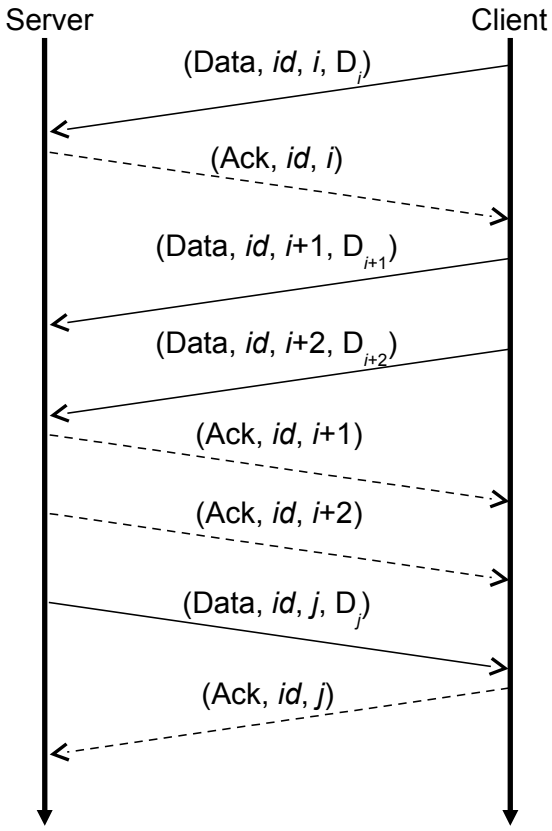


Figure 2: Sending & acknowledging data. Data may be sent from both the client or the server, and all data messages must be acknowledged. Size field is ignored for illustration purpose.

`MaxUnackedMessages`. For example, assume $M = 2, \omega = 4$, and we have sent four messages. If neither of the first and second messages are acknowledged, we cannot send the third message even though the third message is within the sliding window. Note that $M \leq \omega$.

`MaxUnackedMessages` and `WindowSize` ensure that in the cases of many failures (dropped messages) we don't flood the network with too many messages being resent.

2.1.3 Epoch events

Unfortunately, the basic protocol described so far is not robust. On one hand, its sequence numbers makes it possible to detect when a message has been dropped or duplicated. However, if any message—connection request, data message, or acknowledgment—gets dropped, the linkage in either one or both directions will stop working, with both sides

waiting for messages from the other.

To make LSP robust, we incorporate a simple time trigger into the servers and clients. Timers fire periodically on both the clients and servers, dividing the flow of time for each into a sequence of *epochs*. We refer to the time interval between epochs as the *epoch duration*, which we denote with the symbol δ. Our default value for δ is $2,000$ milliseconds, although this can be varied.

Epoch events ensure two separate things:

- Server and clients retransmit messages that may have been dropped
- Server and clients detect when the connections they're communicating with timeout

Try to decouple these two events in your mental model of what's going on, as well as in your implementation, because they are actually very different protocols!

One function of epochs is that the server and clients retransmit data that may have been dropped. Instead of trying to resubmit this data every single epoch however, you will be implementing an *exponential back-off* approach.

Let's define three other variables in addition to δ.

- `EpochLimit` - the maximum amount of epochs that can elapse without a response before we declare the connection dead. We need an epoch limit in order to ensure that we eventually drop dead connections.
- `CurrentBackoff` - the amount of epochs we wait before re-transmitting data that did not receive an ACK. For instance, if `CurrentBackoff` is 8, and we *just* tried to send some data but did not receive an ACK, then we will only attempt to resend the data again after waiting 8 epochs. We provide some more examples below.
- `MaxBackOffInterval` - the maximum amount of epochs we wait without retrying to transmit data. As in the name, `CurrentBackoff` is always less than or equal to `MaxBackOffInterval`.

`EpochLimit` and `MaxBackOffInterval` are given as runtime parameters, but you are responsible for keeping track of the current back-off.

Furthermore, `EpochLimit` is in relationship to detecting when connections are dead, whereas `Current` and `Max Backoff` are in relationship to the retry strategy for un-ACK'd data messages.

The key to exponential back-off is that after `CurrentBackoff` epochs have elapsed and we still have unacknowledged sent data, we *double* the value of `CurrentBackoff` (unless it is currently 0, in which case we add one).

By default, `CurrentBackoff` is initially 0. This means that if a server did not receive the ACK for a data message from a client at a given epoch, we would resend the data to the client on the 1st, 3rd, 6th, 11th, etc. epoch events afterwards.

This is because at first we wait 0 epochs, trying to resend at the very next epoch - epoch 1.
Then we wait one epoch (epoch 2) before retrying.
Next, we wait two epochs (epochs 4, 5) before retrying.
Next, we wait four epochs (epochs 7, 8, 9, 10) before retrying...etc.

Put another way, consider the same case as above where the client stops receiving any messages from the server. Let X be an epoch event where the client retried, and O to be an epoch event where the client did nothing. Assume that before the very first epoch shown below, the client sent some un-ACK'd message to the server. Then we would have this pattern:

$$XXOXOOXOOOOXOOOOOOOOX...$$

Again, the pattern is that we wait 0 epochs, then 1 epoch, then 2 epochs, then 4, etc., until either we reach `EpochLimit` and terminate the connection, or we receive the ACK for the data message in question back, in which case we're good!

When the epoch timer fires, a client takes the following actions :

- If the client's connection request has not yet been acknowledged by the server, then resend the connection request.
- For each data message that has been sent but not yet acknowledged, resend the data message, *following the exponential backoff rules above.*
- If the connection request has been sent and acknowledged, but the client has not sent or resent any data message to the server in the last epoch, then send an acknowledgment with sequence number 0 (This is called a Heartbeat and is explained below).

The server performs a similar set of actions for each of its connections:

- 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 an acknowledgment with sequence number 0 (This is called a Heartbeat and is explained below).

IMPORTANT: Note that we keep track of a `CurrentBackoff` *for each data message*, not for a connection. Also note that for resending connect messages, we should retry every epoch.

Figure 3 illustrates how the epoch events make up for failures by the normal communication. We show the occurrence of an epoch event as a large black oval on each time line. In this example, the client attempts to send data D_i, but the acknowledgment message gets dropped. In addition, the server attempts to send data D_j, but the data message gets dropped. When the epoch timer triggers on the client, it will resend data D_i as `CurrentBackOff` is 0.

Assuming the server has an epoch event around the same time (there is no requirement that they occur simultaneously), we can see that the server will also resend data D_j.

After that, both the client and server send acknowledgments to each other for data D_i and D_j respectively, and this time the messages are not dropped.

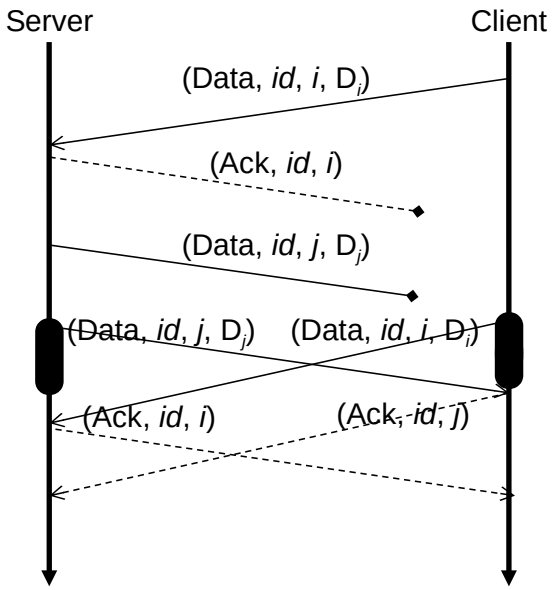


Figure 3: Epoch events. Both sides resend acknowledgments for the most recently received data, and (possibly) resend any unacknowledged data. Size field is ignored for illustration purpose.

We can see in this example that duplicate messages can occur: the server receive two copies of D_i. For most cases, we can use sequence numbers to detect any duplications: Each side maintains a counter indicating which sequence number it expects next and discards any message that does not match the expected window range of sequence numbers. One case of duplication requires special attention, however: it is possible for the client to send multiple connection requests, with one or more requests or acknowledgments being dropped. The server must track the host address and port number of each connection request and discard any for which that combination of host and port already has an established connection.

One feature of this epoch design is that there will be at least one message transmitted in each direction between client and server on every epoch. As a final feature, we will track at the endpoint of each connection the number of epochs that have passed since a message (of any type) was received from the other end. Once this count reaches a specified `EpochLimit`, which we denote with the symbol K, we will declare that the connection has been lost. Our implementation uses a default value of 5 for K; thus, if nothing is received from the other end of an established connection over a total period of $K \cdot \delta$ seconds, then the connection should be assumed lost. Note, if the epoch limit is reached when the client is attempting to establish connection with the server, then the client should close.

Heartbeats: There may be epochs in which the client or server has no data messages they need to send, but they are still alive and running. As described above, in the case that there are no data messages to send or resend in an epoch, we send a "Heartbeat" message, which is an acknowledgment with sequence number 0. These heartbeat messages ensure that the client doesn't think the server has timed out and vice versa. Without these heartbeat messages, if the server and client stay idle for $K \cdot \delta$ seconds, they will consider each other lost.

2.1.4 Data Size

In networking, the packet length is often unknown. Sometimes the data in the packet is of variable length. Sometimes multiple pieces of variable length data are contiguous, and a size is used to skip from one piece to the next. In this project, we only worry about the first case, where there is one piece of variable length data. If the size of the data is not included, and some bytes are lost in transmission, the receiver would not be able to know of the data loss. Although in your LSP implementation, Go's *marshalling* takes care of this for us, we want to send the size before data in case the protocol is ever extended. As a basic data integrity check, if the size of the received data is shorter than the given size included in the message, the message should be rejected, i.e. it should be as if the message was dropped and the server never received the message. If the size of the data is longer than the given size, there is no "correct" behavior, but one such solution, which LSP should employ, is to simply truncate the data to the correct length.

2.1.5 Checksum

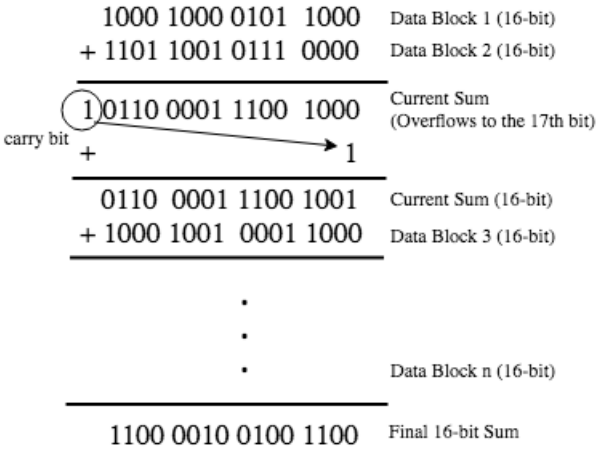
Checksum is used to detect potential errors introduced during the transmission or storage of the data. If the checksum of the received Data message is different from the checksum included in the message, the message is considered as corrupted and thus should be rejected.

There are many checksum algorithms such as MD5 and SHA-1. In this section, we will introduce a simple UDP checksum algorithm. It computes the 16-bit one's complement sum of the Data message as follows:

Step 1: Separate Data message into blocks of 16-bit values. If the message has odd number of bytes, we can pad it with a zero byte, 0x00.

Data Message	Data Blocks
ConnID int (32-bit)	Data Block 1 (16-bit) Data Block 2 (16-bit)
Payload []byte	Data Block 3 (16-bit) Data Block 4 (16-bit) ...
...	...

Step 2: Sum up all the 16-bit values two at a time. After each addition, if a carry bit is produced, add it to the least significant bit.



Step 3: Take the one's complement of the final sum. In the above example, the final sum is

1100 0010 0100 1100, so the checksum of the Data message would be 0011 1101 10110011.

To make your life easier, we have provided the following functions in checksum.go:

```
/*
 * Calculate the 32-bit checksum of a given integer.
 */
func Int2Checksum(value int) uint32;

/*
 * Calculate the 32-bit checksum of a given byte array.
 */
func ByteArray2Checksum(value []byte) uint32;
```

You are also welcome to come up with your own implementation of other checksum algorithms. Note that a good checksum function will always output very different checksums even if only a single bit is flipped in the input.

2.2 LSP API

We will now describe LSP from the perspective of a Go programmer. You must implement the *exact* API discussed below to facilitate automated testing, and to ensure compatibility between different implementations of the protocol.

The LSP API can be found in the `lsp` package (included as part of the starter code). This file defines several exported structs, interfaces, and constants, and also provides detailed documentation describing the intent and expected behavior of every aspect of the API. Consult it regularly!

2.2.1 LSP Messages

The different LSP message types are defined as integer constants below:

```
type MsgType int

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

Each LSP message consists of six fields, and is declared as a Go `struct`:

```
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.
}
```

To send a `Message` over the network, you must first convert the structure to a UDP packet by *marshalling* it into a sequence of bytes. This can be done in Go using the `Marshal` function in the `json` package.

2.2.2 LSP Parameters

For both the client and the server, the API provides a mechanism to specify the epoch limit K, the epoch duration δ, and the sliding window size ω when a client or server is first created. These parameters are encapsulated in the following `struct`:

```
type Params struct {
        EpochLimit         int // Default value is 5.
        EpochMillis        int // Default value is 2000.
        WindowSize         int // Default value is 1.
        MaxUnackedMessages int // Default value is 1.
}
```

2.2.3 LSP Client API

An application calls the `NewClient` to set up and initiate the activities of a client. The function blocks until a connection with the server has been made, and returns a non-`nil` error if the connection could not be established. The function is declared as follows:

```
func NewClient(hostport string, params *Params) (Client, error)
```

The LSP client API is defined by the `Client` interface, which declares the methods below:

```
ConnID() int
Read() ([]byte, error)
Write(payload []byte) error
Close() error
```

The `Client` interface hides all of the details of establishing a connection, acknowledging messages, and handling epochs from the application programmer. Instead, applications simply read and write data messages to the network by calling the `Read` and `Write` methods respectively. The connection can be signaled for shutdown by calling `Close`.

A few other details are worth noting:

- `Read` should block until data has been received from the server and is ready to be returned. It should return a non-`nil` error if either (1) the connection has been explicitly closed, or (2) the connection has been lost due to an epoch timeout and no other messages are waiting to be returned by `Read`, or (3) the server is closed. Note that in the third case, it is also ok for Read to never return anything.
- `Write` should *not* block. It should return a non-`nil` error only if the connection to the server has been lost.
- `Close` should not forcefully terminate the connection, but instead should block until all pending messages to the server have been sent and acknowledged (of course, if the connection is suddenly lost during this time, the remaining pending messages should simply be discarded).
- No goroutine should be left running after `Close` has returned (this will cause our Gradescope tests to hang).
- You may assume that `Read`, `Write`, and `Close` will not be called after `Close` has been called.

For detailed documentation describing the intent and expected behavior of each function and method, consult the `client_api.go` and `client_impl.go` files.

2.2.4 LSP Server API

The API for the server is similar to that for an LSP client, with a few minor differences. An application calls the `NewServer` to set up and initiate a LSP server. However, unlike `NewClient`, this function should *not* block. Instead, it should simply begin listening on the specified port and spawn one or more goroutines to handle things like accepting incoming client connections, triggering epoch events at fixed intervals, etc. If there is a problem setting up the server, the function should return a non-`nil` error. The function is declared as follows:

```
func NewServer(port int, params *Params) (Server, error)
```

The LSP server API is defined by the `Server` interface, which declares the following methods:

```
Read() (int, []byte, error)
Write(connID int, payload []byte) error
CloseConn(connID int) error
Close() error
```

Similar to the `Client`, the `Server` interface allows applications to both read and write data to its clients. Note, however, that because the server can be connected to several LSP clients at once, the `Write` and `CloseConn` methods take a client's unique connection ID as an argument, indicating the connection that should be written to or that should be closed.

A few other details are worth noting:

- `Read` should block until data has been received from some client and is ready to be returned. It should return a non-`nil` error if either (1) the connection to some client has been explicitly closed, or (2) the connection to some client has been lost due to an epoch timeout and no other messages from that client are waiting to be returned by `Read`. This method should not return data from a connection that was explicitly closed by a call to `CloseConn`.
- The `Write` and `CloseConn` methods should *not* block, and should both return a non-`nil` error value only if the specified connection ID does not exist.
- `Close` should block until all pending messages to each client have been sent and acknowledged (of course, if a client that still has pending messages is suddenly lost during this time, the remaining pending messages should simply be discarded).
- No goroutine should be left running after `Close` has returned (this will cause our Gradescope tests to hang).
- You may assume that after `CloseConn` has been called, neither `Write` nor `CloseConn` will be called on that same connection ID again. You may also assume that no other `Server` methods calls will be made after `Close` has been called.

For detailed documentation describing the intent and expected behavior of each function and method, consult the `server_api.go` and `server_impl.go` files.

2.3 Starter Code

The starter code for this project is hosted as a read-only repository on GitHub. For instructions on how to build, run, test, and submit your server implementation, see the `README.md`

file in the project's root directory. To clone a copy, execute the followings Git command:

```
git clone https://github.com/15-440/p1.git
```

Part A starter code can be found in the `p1/src/github.com/cmu440/` directory, and is organized as follows:

- The `lsp` package contains the API, tests, and the starter code you will need to complete:
  - The `client_api.go`, `server_api.go`, `message.go`, and `params.go` files collectively define the LSP API. To facilitate automated testing, you **must not** modify these files.
  - The `client_impl.go` file contains a skeletal implementation of the `Client` that you will write.
  - The `server_impl.go` file contains a skeletal implementation of the `Server` that you will write.
  - The `*_test.go` files contain the tests that we will run on Gradescope to test and evaluate your final submission.
  - The `checksum.go` file defines the helper functions you will need to compute the 16-bit one's complement checksum.
- The `lspnet` package contains all of the UDP operations you will need to complete this assignment. To import, use `import "github.com/cmu440/lspnet"`. You will not need and are not permitted to use the `net` package. Under-the-hood, the `lspnet` package provides some additional functionalities that allow us to more easily grade the robustness of your implementation.
- The `srunner` (server-runner) and `crunner` (client-runner) packages each provide simple executable programs that you may use for your own testing purposes. Instructions on how these programs can be used are posted in the project's `README.md` file on GitHub.

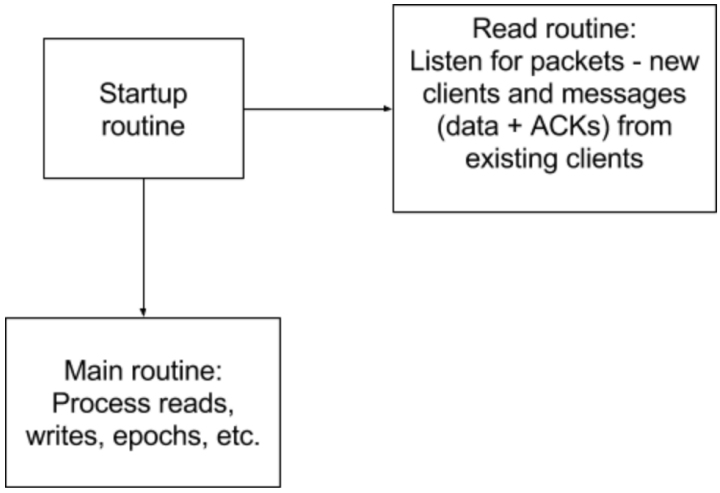
We have also provided pre-compiled executables of the `srunner` and `crunner` programs discussed above that you can use for testing. The binaries were compiled against our reference LSP implementation, so you might find them useful in the early stages of the development process (for example, if you wanted to test your `Client` implementation but haven't finished implementing the `Server` yet, etc.). Instructions on how these programs can be used are posted in the project's `README.md` file on GitHub.

In addition to the starter code we provide, you may create your own utility files if you wish. For example, it might be a good idea to create a `common.go` file and use it to store code that can be shared between both your `Client` and `Server` implementations.

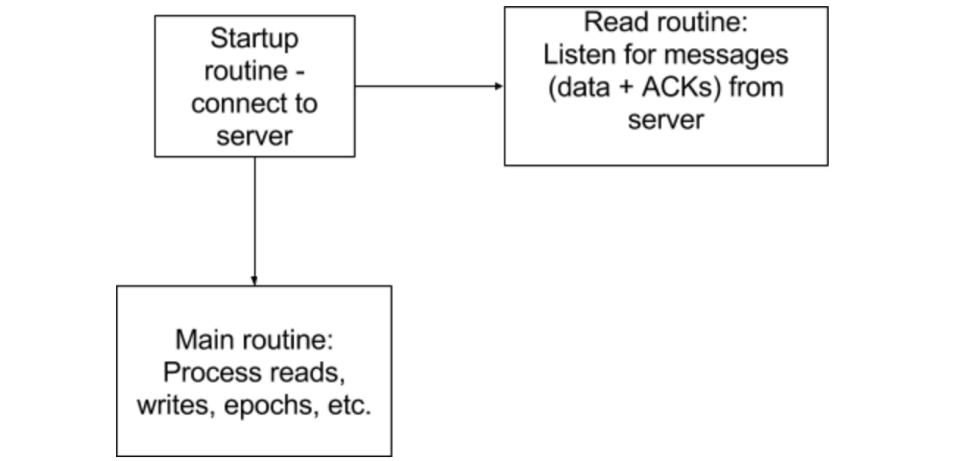
3 Diagrams

These diagrams are to try to help you get started and think about how to structure your goroutines. Keep it mind that it is important to design and plan out your structure carefully before writing code!

Server:



Client:



4 Checkpoint: Read/Write Server

WARNING: This checkpoint is *much* less work than the rest of Part A. Please plan accordingly.

Epochs are a necessary aspect of the LSP protocol due to imperfect networks: there is no way to prevent messages being dropped, we can only detect a dropped message and retransmit it. For the checkpoint, we will assume the network is perfect and no packet loss occurs. Therefore, epoch events are not necessary.

However, although packet loss will not occur, it is possible that messages will be sent out of order, and you will need to correct for that via the sliding window protocol discussed earlier.

We will only require you to pass the first 9 basic tests and 3 out of order tests, and we will not test for race conditions.

5 Part B: Distributed Bitcoin Miner

In part B of this project, you will implement a simple distributed system using your LSP implementation you wrote in part A. Your system will harness the power of multiple processors to speed up a compute-intensive task and will be capable of recovering from sudden machine failures.

5.1 Bitcoin Overview

Bitcoin is a decentralized digital currency. At the heart of the Bitcoin protocol is a replay prevention mechanism that prevents participants from double spending Bitcoins. The replay prevention mechanism uses a crypotographic proof-of-work function that is designed to be computationally hard to execute. Clients compete to be the first to find a solution to the proof-of-work in order to get their signature attached to a sequence of transactions. If a client wins, they are rewarded with Bitcoins. The process of finding a solution to the proof-of-work is called *mining*.

In this project, we will spare you the details of the real Bitcoin protocol. You will instead implement a mining infrastructure based upon a simplified variant: given a message M and an unsigned integer N, find the unsigned integer n which, when concatenated with M, generates the *smallest* hash value, for all $0 \leq n \leq N$. Throughout this document, we will refer to each of these unsigned integers as a *nonce*.

As an example, consider a request with string message `"msg"` and maximum nonce `2`. We can determine the desired result by calculating the hash value for each possible concatenation. In this case, nonce `1` generated the least hash value, and thus the final result consists of least hash value `4754799531757243342` and nonce `1`. Note that you need not worry about the details of how these hash values are computed—we have provided a `Hash` function in the starter code for you to use to calculate these hashes instead.

- `bitcoin.Hash("msg", 0) = 13781283048668101583`
- `bitcoin.Hash("msg", 1) = 4754799531757243342`
- `bitcoin.Hash("msg", 2) = 5611725180048225792`

One simple approach to completing this task is to perform a brute-force search, in which we enumerate all possible scenarios across multiple machines. For tasks that require searching a large range of nonces, a distributed approach is certainly preferable over executing the compute-intensive task on a single machine. As an example, our measurements show that a typical Andrew Linux machine can compute SHA-256 hashes at a rate of around 10,000 per second. Running sequentially, a brute force approach would require around 28 hours to try all possible hashes consisting of 9 decimal digits. But, if we could harness the power of 100 machines, then we could reduce this time to around 17 minutes.

Your task for this project is to implement a simple distributed system to perform this task. We discuss our proposed system design in the next section.

5.2 System Architecture

Your distributed system will consist of the following three components:

Client: An LSP client that sends a user-specified request to the server, receives and prints the result, and then exits.

Miner: An LSP client that continually accepts requests from the server, exhaustively computes all hashes over a specified range of nonces, and then sends the server the final result.

Server: An LSP server that manages the entire Bitcoin cracking enterprise. At any time, the server can have any number of workers available, and can receive any number of requests from any number of clients. For each client request, it splits the request into multiple smaller jobs and distributes these jobs to its available miners. The server waits for each worker to respond before generating and sending the final result back to the client. The server should be "fair" in the way it distributes tasks. We will discuss this below.

Type	Fields	From–To	Use
`Join`		M–S	miner's join request
`Request`	`Data Lower Upper`	C–S, S–M	send job request
`Result`	`Hash Nonce`	M–S, S–C	report job's final result

Figure 4: Predefined message types in the Bitcoin distributed system. In the "From–To" column, 'M' denotes a miner, 'S' denotes the server, and 'C' denotes a request client.

Each of the three components communicate with one another using a set of predefined messages. Figure 4 shows the types of messages that can be sent between the different system components. Each type of message is declared in the `message.go` file as a Go `struct`. As in part A, each message must first be marshalled into a sequence of bytes using Go's `json` package.

5.2.1 Sequence of events

The overall operation of the system should proceed as follows:

- The server is started using the following command, specifying the port to listen on:

  `./server` *`port`*

- One or more miners are started using the following command, specifying the server's address and port number separated by a colon:

  `./miner` *`host:port`*

- When a miner starts, it sends a join request message to the server, letting the server know that it is ready to receive and execute new job requests. New miners may start up and send join requests to the server at any time.

- The user makes a request to the server by executing the following command, specifying the server's address and port number, the message to hash, and the maximum nonce to check:

  `./client` *`host:port message maxNonce`*

  The client program should generate and send a request message to the server, specifying lower nonce "0" and maximum nonce "maxNonce":

  `[Request` *`message`* `0` *`maxNonce`*`]`

- The server breaks this request into more manageable-sized jobs and farms them out to its available miners (it is up to you to choose a suitable maximum job size). Once the miner exhausts all possible nonces, it determines the least hash value and its corresponding nonce and sends back the final result:

  `[Result` *`minHash nonce`*`]`

- The server collects all final results from the workers, determines the least hash value and its corresponding nonce, and sends back the final result to the request client.

The request client should print the result of each request to standard output as follows (note that you must match this format precisely in order for our automated tests to work):

- If the server responds with a final result, it should print,

  `Result` *`minHash nonce`*

  where `minHash` and `nonce` are the values of the lowest hash and its corresponding nonce, respectively.

- If the request client loses its connection with the server, it should simply print

  `Disconnected`

5.2.2 Handling failures

We will assume that the server operates all the time, but it is quite possible that a request client or some of the miners can drop out. You should take the following actions when different system components fail:

- When a miner loses contact with the server it should shut itself down.
- When the server loses contact with a miner, it should reassign any job that the worker was handling to a different worker. If there are no available miners left, the server should wait for a new miner to join before reassigning the old miner's job.
- When the server loses contact with a request client, it should cease working on any requests being done on behalf of the client (you need not forcibly terminate a job on a miner—just wait for it to complete and ignore its results).
- When a request client loses contact with the server, it should print `Disconnected` to standard output and exit.

5.2.3 Load balancing

Your server will need to implement a *scheduler* to distribute work among workers that satisfies:

- **Efficiency:** the *scheduler* aims to minimize mean response time for all client requests.
- **Fairness:** requests that arrive earlier have some sort of priority.

You are free to choose any load balancing/task allocation algorithm of your liking, but we will expect behaviour like this:

- If the server gets an extremely large request followed by a small request, we expect that the small request will finish before the large request.[1]
- If the server gets multiple requests of similar length, the requests should finish approximately in the order they arrived. For example, if 100 identical requests come in and the 100th request finishes first, that would not be very fair.

[1] In fact, Shortest Remaining Time First is the optimal policy to minimize mean response time, as seen in this paper, A New Proof of the Optimality of the Shortest Remaining Processing Time Discipline

- Work should be divided roughly evenly across workers. Try to minimize the idle time of each worker.

Your code should contain documentation on how your scheduler achieves these requirements. Feel free to look up popular load balancing algorithms. Be creative and have fun!

5.3 Starter code

The starter code for part B of this project can be found in the `bitcoin/` directory, and is organized as follows:

- The `message.go` file defines the message types you will need to implement your system.
- The `hash.go` file defines a `Hash` function that your miners should use to compute `uint64` hash values.
- The `client/client.go` file is where you will implement your request client program.
- The `miner/miner.go` file is where you will implement your miner program.
- The `server/server.go` file is where you will implement your server program.

We have provided some starter code in the three files that you need to implement as mentioned above, and feel free to make any changes to them. However, do not modify `message.go` and `hash.go`. For instructions on how to compile and test your client, miner, and server programs, please consult the `README.md` file on GitHub.

6 Project Requirements

As you write your code for this project, also keep in mind the following requirements:

- As with all projects in this course, we will be using Moss to detect software plagiarism (including comparisons with student submissions from past semesters).
- Your code **must not** use locks and mutexes. All synchronization must be done using goroutines, channels, and Go's channel-based `select` statement. We strongly discourage implementing lock-like behavior using channels. **If you use lock-like behavior using channels, we will not help you debug your code.**

- You **must not** use Go's `net` package for this assignment. Instead, you should use the functions and methods in the `lspnet` package we have provided as part of the starter code for this project instead.
- Avoid using fixed-size buffers and arrays to store things that can grow arbitrarily in size. For example, **do not** use a buffered channel to store pending messages for a particular connection. Instead, use a linked list—such as the one provided by the `container/list` package—or some other data structure that can expand to an arbitrary size.
- You may assume that the UDP packets will not be corrupted, and that you do not need to check your messages for proper formatting (unless, of course, you want to defend against your own programming errors).
- You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.