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 A is due on **Friday, February 21 at 11:59pm**.
- **Part B:** Implement a Distributed Bitcoin Miner. Details will be announced shortly before the part A deadline. Part B will be due on **Friday, February 28 at 11:59pm**.

Details on how to hand in your code via Autolab will be provided closer to the due dates.

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 following Git command:

```
git clone https://github.com/cmu440/p1.git
```

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 or duplicated 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.

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.
- 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. 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 four 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: 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, D): Data message with ID id, sequence number sn, and payload D.
- (Ack, id, sn): Ack message with ID id, and sequence number sn.

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

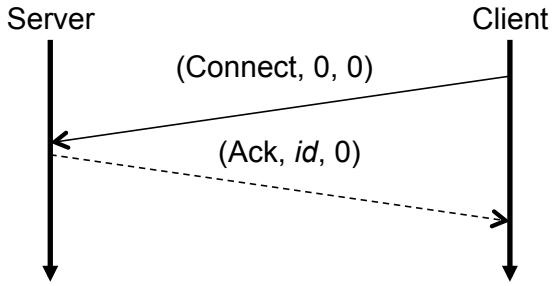

Figure 1: Establishing a connection. A client sends a connection request to the server, which responds to the client with an acknowledgment message containing the connection's unique ID. The vertical lines with downward arrows denote the passage of time on both the client and the server, while the lines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

data message from the server to the client. Note that all messages are acknowledged, and that the client and server maintain their own series of sequence numbers, independent of the other.

Like TCP, LSP includes a *sliding window protocol*. The sliding window represents an upper bound on the number 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 message are acknowledged, etc. Note that it is entirely possible for one side to receive data messages from the other as it is waiting for these acknowledgments—the client and server operate asynchronously, and there is no guarantee that packets arrive in the same order they are sent.

The sliding window protocol provides the benefit of decreasing overall transmission time when acknowledgment packets have high latency. However, it comes at the costs of more implementation complexity and more re-transmission traffic: clients and servers need to maintain a buffer of not-yet-acknowledged messages (i.e., the ones inside the window) that might need to be re-sent later.

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

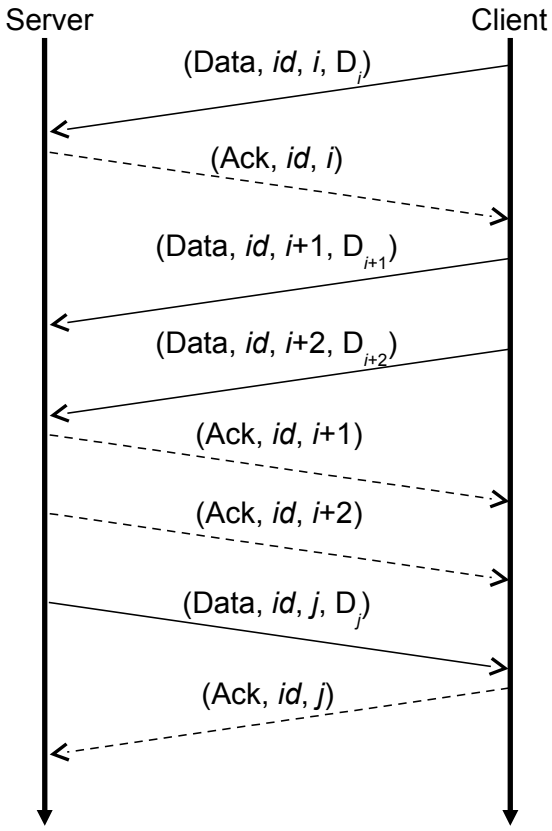

Figure 2: Sending & acknowledging data. Data may be sent from both the client or the server, and all data messages must be acknowledged.

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.

Each time the epoch timer fires a client takes the following actions (assuming a window size of ω):

- Resend connection request, if the client's initial connection request has not yet been acknowledged by the server.
- If the connection request has been sent and acknowledged, but no data messages have

been received, then send an acknowledgment with sequence number 0.

- For each data message that has been sent but not yet acknowledged, resend the data message.
- If any data messages have been received, resend an acknowledgment message for each of the ω (or fewer) most recently received data messages.

The server performs a similar set of actions for each of its connections (also assuming a window size of ω):

- If no data messages have been received from the client, then resend an acknowledgment message for the client's connection request.
- For each data message that has been sent, but not yet acknowledged, resend the data message.
- If any data messages have been received, resend an acknowledgment message for each of the ω (or fewer) most recently received data messages.

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 send an acknowledgment of data message $j-1$, the last data message received, and it will resend data D_i.

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 send an acknowledgment for data D_i, and it will resend data D_j.

We can see in this example that duplicate messages can occur: for example, the server might 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

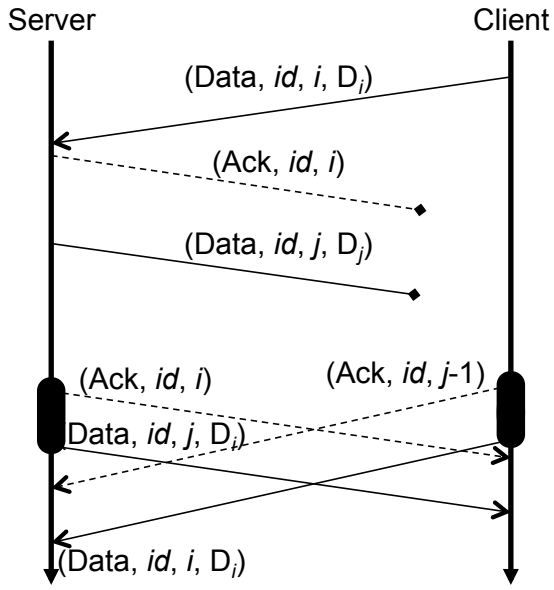

Figure 3: Epoch events. Both sides resend acknowledgments for the most recently received data, and (possibly) resend any unacknowledged data.

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 *epoch limit*, which we denote with the symbol K, we will declare that the connection has been lost and notify the application that it has lost this connection. 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 terminated.

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 four 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.
        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.
}
```

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 either (1) data has successfully been received from the server, or (2) the connection to the server has been lost. In the latter case, a non-`nil` error should be returned.
- `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 Autolab 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 right 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 closed.

A few other details are worth noting:

- `Read` should block until either (1) data has successfully been received from some client, or (2) the connection to some client has been lost. In the latter case, a non-`nil` error should be returned. `Read` must not return data from a connection that has been closed.
- 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 Autolab tests to hang).
- You may assume that neither `Write` nor `CloseConn` will be called on a connection ID that has already that has already been signaled to be closed. You may also assume that no other `Server` methods will be made after `Close` is 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 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 Autolab to test and evaluate your final submission.
- The `lspnet` package contains all of the UDP operations you will need to complete this assignment. 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 and/or packages 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.

2.4 Requirements

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

- The project must be done individually. You **must not** use any code that you have not written yourself. 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.
- You **must not** use Go's `net` package for this assignment. Instead, 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.

3 Part B: Distributed Bitcoin Miner

Part B will be released (and announced on Piazza) shortly before the part A deadline.

To update this document, simply execute `git pull` from the command line. You can also download an updated copy on the course website once it is released.