Server
The Go implementation stores a variety of different fields for every server.

The main entry point for the logic of the server is in processRequest. Here, there are 4 types of messages that the server can process. MessageType 0 corresponds with a client operation request. MessageType 1 corresponds with receiving a gossip message from a different server, fulfilling client requests if dependencies are now satisfied, and also sending an acknowledgement to the server we have received the gossip message from. MessageType 2 corresponds with updating internal metadata from receiving an acknowledgement of a gossip message. MessageType3 corresponds to sending a gossip message to other servers.

For MessageType 0, when we process a client operation we first check whether or not the dependencies are satisfied. The client is responsible for making sure to send the correct version vector. If the dependencies are satisfied then depending on if it is a read or write we update our internal state accordingly and also append to the outGoingRequests array any type of message we need to send. If the dependencies are not satisfied then we append the message to the UnsatisfiedRequests array.

For MessageType 1, when we process a gossip request we attempt to maximally apply all the gossip operations if their dependencies are satisfied; if not then any operations that are not applied will be stored in PendingOperations. Afterwards, we go through any unsatisfiedClientRequests and see if we can apply them. If we can apply them then we do and then inform the client.

For MessageType 2, when we receive an acknowledgement, we advance the counter for what messages we need to send to each server since this will now be the new lower bound.

For MessageType 3, we send the operations that other servers haven't seen from our applied operations.

Client
The main entry point for the client implementation in Go is processRequest. There are 3 different types of requests. RequestType 0 corresponds with a read, RequestType 1 corresponds with a write and RequestType 2 corresponds with an acknowledgement message from the server. When we first start a client we spawn multiple threads each with their own client structs. These threads all run in parallel and send messages to the server waiting for the server's acknowledgement. This is done for an arbitrary amount of time corresponding to how long we want the experiment to run.

Main File

The main file facilitates setting up all the parameters for the client and the server.

Coq

For the Coq implementation we only verify processRequest and all functions that processRequest calls in the server file. The same is done for processRequest in the client file. The way this works is that we write a function in Coq that is functionally equivalent to each of the corresponding functions in the Go version. We then prove that given equivalent inputs we will get equivalent outputs between GooseLang function and the Coq function. Equivalence is defined by a proposition that relates terms in GooseLang and the pure Coq versions.

In order to translate go files to their GooseLang counterparts you will need to install Goose using https://github.com/goose-lang/goose and use the following command:

```
~/go/bin/goose -out [name of output folder your file will be contained in] [location of go file]
```

https://github.com/alanwang67/goose_converison/blob/master/src/program_proof/session/definitions.v defines how the structs in GooseLang are related to records in Coq. For more resources to understand how Goose represents slices, see https://tchajed.github.io/sys-verif-fa24/notes/ownership.html.