How to Run

Running each server: *python3 server.py*

Running each client: *python3 client.py*

Run three instances of the server script, and two instances of the client script. We will refer to these server instances as (1, 2, 3) and the client instances as (A, B) respectively.

When all five scripts have launched, begin entering in the following inputs to get the program running. **All inputs must be correctly entered within a ten-second window following the first input for the program to run correctly.**

server.py

- Server 1 →Enter “1”
- Server 2 → Enter “2”
- Server 3 → Enter “3”

client.py

- Client A → Enter “A”
- Client B → Enter “B”

The program is only designed to handle the Alice/Bob instructions, which according to the TA should be sufficient. As a result, inputs must be sent in the following order.

1. Wait for all connection messages to display, there should be seven for each server.
2. Send “lost” on the Alice client. Wait for all the servers to display the message.
3. Send “wait” on Alice, input 3 seconds. This will stall the appropriate Lamport time for the next message.
4. Send “found” on the Alice client. Wait for all the servers to display the message.
5. Send “wait” on Bob, input 5 seconds. This will stall the appropriate Lamport time for the next message.
6. Send “glad” on the Bob client. Wait for all the servers to display the message.

Program Structure

The program is divided up into two scripts, one to handle the functions, input, and output of the servers and one for the clients.

The server script is divided as follows:

- The server begins by initializing global variables to use across threads, then asking the user to specify the identity of the server. These global variables will be periodically imported into threads as they need to update them.
  - The *clock* global variable is a Lamport clock which adjusts based either on delay or on the time specified by the client. It’s simply an integer which is incremented on events.

    This allows us to see the order in which messages are being sent and simulate the timing present in the diagram.
    - The *loc* global variable tells us which server we currently are, which we will use later when storing and sending that information.
    - The *dependency* global variable is a dictionary containing all the clients, what messages have been sent, to what servers, and when they were sent. We will use this when checking dependency.
- Depending on user input, more global variables will be set to specify the ports of the server and its peers.
    - Ports 8000, 8001, and 8002 (referred to as *ports*) handle the server-client connections for Servers 1, 2, and 3 respectively.
    - Ports 8100, 8101, and 8102 (referred to as *peer_ports)* handle the server-to-server connections for Servers 1, 2, and 3 respectively.
- When the user-input is given, the server waits for ten seconds before resuming. This wait is crucial as it gives time for the user to get all the other servers and clients set up before the program attempts to begin connecting. Without it, or if the user fails to input in time, the program will not run.
- The process of execution is divided up into three distinct threads, plus the main execution loop:
    - Within the main execution loop, the server is constantly listening for new client connections. On a connection, the server will spawn a new *client_target* thread to handle the server-client logic.
    - A single thread is spawned to begin setting up the “server-type” logic of the server-to-server connections. This thread is called *primary_listen* and has its own loop, listening for new connections. On a connection, it will spawn a *primary_target* thread to handle the connection’s logic. Print statements tell the user when the server is online and when the server has connected to other servers.
    - The *primary_target* thread is what actually sends messages over to the other servers on the server-to-server connection. Whenever *client_target* indicates that it is ready to propagate the information it received from the client, it will set the *status* global variable to a non-empty value. *primary_target* will detect when the list is non-empty and take that as a signal that it should propagate the information contained within the list. A simulated time-delay is applied which scales based on the distance between the two servers. When it is confident that it has sent out all the messages necessary, it will set *status* back to an empty list to indicate it is finished.
    - One *replica_target* thread is started per each other server on the network. The *replica_target* thread handles incoming messages from the other servers. Upon receiving a message, the program checks to ensure that the dependencies have been received. If not, the message is stalled. Once the checks are complete, the message is processed into the *dependency* dictionary and prints out the relevant information to the user.
    - The *client_target* thread is constantly listening for messages from the client in the form of various signals:
        - OP_NULL is the default signal. Whenever this is running, the server continues to wait for a signal indicating other operation logic. Any other operation must set the operation mode back to OP_NULL to ensure that the server can continue to receive input.
        - The remaining signals are write signals (with implicit read ability embedded). The general template is the same. A signal is received, processed, and then added to *dependency* under the client with both the server *loc* and current *clock* value. In the case of both *FOUND* and *GLAD* signals, which have dependencies, *dependency* is first scanned to check if the requisite signals are in place before anything is committed.

The client script is structured as follows:

- The script begins by asking the client to identify themselves. Depending on the answer, the port of the client's assigned homeserver will change. Alice has Server 1 as her homeserver while Bob has Server 2 as his.
- A socket and some global variables are set up, the socket then connects to the client's homeserver. The client then informs the server of its identity before going into one of two command modes depending on the identity of the client.
- Both Alice and Bob have commands which allow them to send messages and a command which allows them to wait.
    - The *wait* command prompts Bob to specify a time, then the script stalls for that time and *clock* updates. The user is informed of the new time.
    - The messages available to Alice include *lost* and *found*, whereas Bob is able to send *glad*. In all cases, the clock is incremented, and then a list containing the signal, the server_id, and the current timestamp is forwarded to the homeserver for processing.

Program Functionality

Parts of the program which work:

- The entire Bob/Alice scenario as laid out in the assignment.
- Sending messages from server to client
- Sending messages from server to server
- Ensuring causal consistency among messages in one server
- Ensuring causal consistency among messages across servers, and stalling when inconsistent

Sample Output

Server Output:

```
Which data center is this (1, 2, or 3)?

        2

Server 2 is online

Waiting 10 seconds...

Server on port 60358 ready to accept requests

Accepting requests from Server 1

Accepting requests from Server 3

Bob has connected at 127.0.0.1:41202

Server on port 60370 ready to accept requests

Alice: I have lost my wedding ring! (recieved from Server 1 at time 3)

Alice: I have found my wedding ring! (recieved from Server 1 at time 7)

GLAD recieved before dependency, stalling for 3seconds.

Bob: I am glad to hear that! (recieved directly from client at time 8).
```

Client Output:

```
Are you Alice (A) or Bob (B)?

        A

What would you like to send?

        lost

What would you like to send?

        wait
```

```
How many seconds would you like to wait?

            3

The time is now 4 seconds.

What would you like to send?

            found
```