Report for Assignment 3 – Distributed KV, P434-SP22 Distributed Systems

Design Details:

Eventual Consistency:

I utilized my previous KV store system built in Assignment 1 to implement both eventual and sequential consistency.

First, I created 3 different servers, server eventual (replica 1), server\_eventual\_2 (replica 2), and server\_eventual\_primary (primary). I edited my handle\_client function for replicas 1 and 2 to forward a request to write to the primary server, and then, the write occurs at the primary replica and the respective replicas. I also implemented an update database() function in all replicas, which reads the database\_master.txt and overwrites all data in the server replicas database to implement eventual causality.

Client API:

There are 3 main functions for the client:

1. set <key> <value> This sets a key-value pair in the database for the respective client and replica. It returns set 1000 <value> STORED, and sends a message to the client to notify of a successful set function
2. get <key> This returns the key-value pair stored in the database for the respective client and replica, NOT from the master database or other replicas’ databases.
3. quit This basically just ejects the client from the server, but I have not found a way to shutdown the server together with the client’s ejection.

Logs/Screenshots, Test Suite:

Graphical user interface, application

Description automatically generated

Figure : blank databases for replica 1 and 2, master has some input

I first initialized the databases of the replicas to be empty, to start off clean. Then, I started up all servers, and started 2 clients, client.py, and client2.py. client.py connects to replica 1 (server\_eventual), while client2.py connects to replica 2 (server\_eventual\_2).

Text

Description automatically generated

Figure : 1-client connection

From Figure 2, we can see that there are 2 servers running, and each has 1 client connected. Then, I initiated gets and sets for clients on replica 1 and replica 2, as shown in Figure 3.

A computer screen capture

Description automatically generated with medium confidence

Figure : gets and sets to show stale reads

From Figure 3, we can also see that there are stale reads when we write from different clients at different replicas. In the example, I did a set Orange Juice <k-v> entry in client 1 for replica 1, and a set Apple Rings <k-v> entry in client 2 for replica 2. We can see in figure 4 that the stale read databases show the relevant writes.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : stale-read database proof

We can also see that I have written other entries to the master database just to separate the entries. After a delay, the eventual consistency implementation kicks in, and our databases update.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : eventual consistency update

To show that I did not manually change the database text files to fit the requirement, I implore you to zoom in to the command prompt windows. Each time the replicas are updated through eventual consistency, the timer threads I implemented will print the text “Eventual consistency regular update complete… Time interval: 180 seconds”. Refer to the handle\_client function in both replica servers to look at the implementation of the timer threads to utilize the eventual consistency model.

A computer screen capture

Description automatically generated with medium confidence

Figure : n-client get

Looking at Figure 6, we can see that I have implemented a n-client connection to one replica database, and it is successful in doing a get function. If you let it wait for long enough (180 seconds), the eventual consistency message will be shown on the windows. These all prove the eventual consistency model has been implemented in my code for all cases.

Limitations of server and additional details:

I noticed that you could use OOP to instantiate these servers as objects, but I decided to create multiple replicas instead since I was short on time and wasn’t very familiar with OOP on python.

I am unsure of how to shut the server down when clients quit, but I don’t think that’s required. For now, my implementation doesn’t really shutdown the server properly, so you have to just close the command prompt window to shut the terminal down.

Also, I didn’t implement a script as I am unsure of how to do it on the linux terminal for the SICE terminals. So, most inputs are manually hardcoded without automation, and this includes: IP addresses, ports, set/get functions, etc.

The main challenges for me were the implementation of the network communication between servers and clients since I have no prior knowledge of networking code protocol. It was also hard to find a suitable implementation to run the eventual consistency model updates, but I found out that a timer thread works well in this situation.

In general, there is minimal latency for sets/gets, since I am running on a local server. I didn’t implement any sleeps in the writes/gets since I didn’t want to made my program not return the proper output.

I set the header size for the TCP ports to be 1000 bytes. Assuming my program uses the MSS method (<https://www.sciencedirect.com/topics/computer-science/maximum-segment-size>) to measure the maximum segment size that the TCP ports can offer to transfer data, I can assume that my program has a throughput of AT LEAST 1000 bytes per second.