Nimay Desai
Armaan Rai

Readme

Client:

To run the client, follow these steps:

1) Compile Client.java
   javac Client.java
2) Execute the file with the following arguments:
   java Client <receiving port> <replica id from config file> <config file>
3) Now you can start send messages by entering them into stdin.

Implementation:

The client will get commands from stdin and put them on a queue, sending a command only after it has received the ack from the previous command. If the client attempts to the send a command and the server cannot receive the command, it will assume the server has crashed and connect to the server with the next higher id and resend the command to it. If the client successfully sends the command to the replica, but does not receive an ACK within 2*max_delay time, it will assume the server crashed and connect to the server with the next higher id also. We did this by having the server socket attempt to time out every 2*max_delay time period, and then check to see if t is waiting for an ack and compare the current system time to the time the last message was sent to verify that the ack has not come in the maximum possible delay period. In this case, it will not resend the command. The client prints the acknowledgements it receives to standard out. For the delay command, the client will not block from stdin, so the user can keep entering commands, which will be placed on a send queue as usual.

Eventual Consistency:

To run the eventual consistency model, follow these steps:

1) Compile ServerEC.java
   javac ServerEC.java
2) Execute the file with the following arguments
   java ServerEC <server port from config file> <config file> <R> <W>
3) If this does not work, you may need to add the file to your classpath.

Implementation:

We used Lamport timestamps to compare messages. Each variable has its own Lamport Timestamp. When a replica receives a put request from the client, we initialize the variable with the provided value. When a replica receives a request from a client, it updates its value for the variable and then forwards the request to all other replicas. Once it receives W acks, it sends an ack back to the client. It knows it has received W acks because it will keep track of a count associated with the message id. With each ack it receives, it

updates the count. A read request from the client undergoes a similar process, except that we also store the first R ack values from the other replicas and their respective timestamps. We send the one with the latest timestamp back to client. When a replica receives a forwarded message from another replica, it checks to see whether it is a put or get. If it is a put, it will check the timestamp of the variable. If the variable is a greater timestamp, it will write that value to its copy of the variable and then send an ack. If not, it will just send an ack. If it is a get, it will send its value for the variable in its ack along with the timestamp. In general, we break timestamp ties with lower server id winning. We create unique client id's by appending the server's record of its client id to the server's id and using that in the log file. The server updates its timestamp to the new timestamp whenever it updates its value to the value of another server and increments its timestamp when it does a write on its own value due to a client request.

Linearizability:

To run the eventual consistency model, follow these steps:

1) Compile ServerLinear.java
   javac ServerLinear.java
2) Execute the file with the following arguments
   java ServerLinear <server port from config file> <config file>
3) If this does not work, you may need to add the file to your classpath.

Implementation:

We used total ordering multicast between server messages to ensure linearizability. To implement the total ordering, we took a sequencer based approach. When the client sends a command to a replica Ri, the replica then broadcasts that message to all replicas, including itself. Out of these replicas, the one with ID = 1 is assigned the role of the sequencer. This sequencer then increments its global sequence number (GS), which starts at 0, and then broadcasts the message to all replicas along with the GS. Each replica maintains its own local received sequence number, Li (initially 0). Now when a replica Ri receives a message M from another replica Rj, it buffers the message until it receives a <M, GS(M)> from the sequencer, where GS(M) = Li + 1. When this happens, it delivers the message and increments Li by 1. In the case of a write command (put), it updates the data hash map, which stores all the information, and in the case of a read command (get), it checks the value of the variable. When a message is delivered, the replica also checks the origin of the message. In the case that the received message is from the same server, it responds to the client using the client ID that is sent along with all messages. These client IDs are unique to each client that joins the network of replicas.