Michael Hla
Aayush Karan

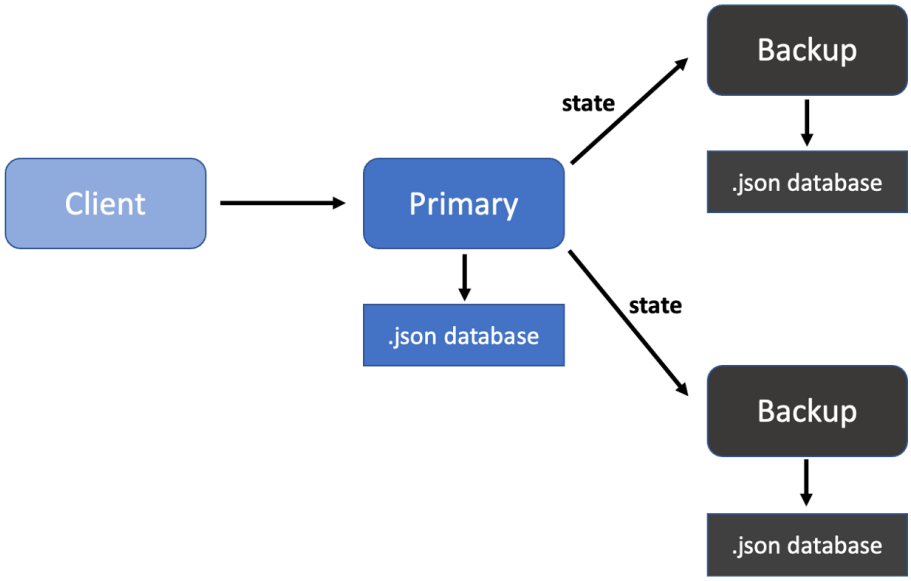
Replication.proto

System Design:

We employ the use of python sockets for this project, utilizing our wire protocol from Design Project 1. For further details on the server-client wire-protocol, see our first repository for the project. We begin by describing how we set up the replicated servers.

Primary-Backup Replication

Our methodology follows the primary-backup replication procedure, where a single server acts as a primary, sending messages of server state updates to the backup servers and updating its own state in response to the client. The backup servers simply update their state according to the primary's directives. When the primary fails, the connection with the backups are broken, and the backups engage in a leader election process to determine the subsequent primary. Once a primary is chosen, clients may now connect to this new server, and the process continues. Moreover, we allow for a dead server to rejoin the network, assuming a primary exists, as a backup server, and catches up by interaction with the primary server.



As a note for scalability, observe that the primary-backup method is well suited for a small number of servers, but when scaling, the load of tasks on the primary, especially with all clients directed towards the primary server, a consensus algorithm like Paxos coupled with clients being able to interact with all other servers works much better, in terms of load balancing.

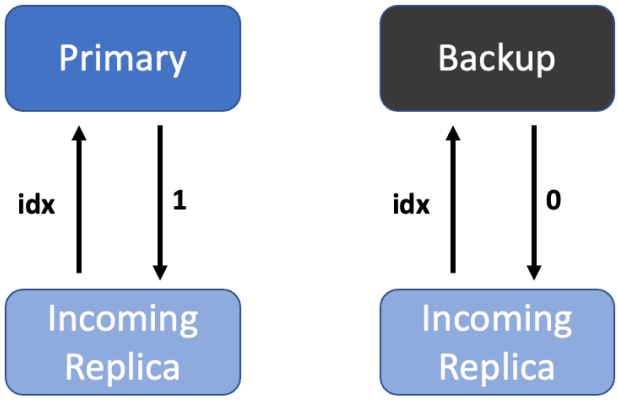
We ensure that our replicated servers are 2-fault tolerant, meaning that out of the 3 servers, the failure of any 2 ensures preservation of server state on the remaining server and continuable action with the client. Moreover, we ensure persistence, where even with the failure of all 3

servers, we can bring them back all online by loading a server state that is saved to local memory. We detail how exactly we implement these with sockets in what follows.

Initialization
We have our servers bind to 2 ports, one being server facing, and one client facing. The server facing server is **backupserver**, and the client facing server is **clientserver**. We fix the IP addresses and ports at which all of these bind initially, and so inputting the machine index (1, 2, 3) as an argument fully determines where these ports bind.

In addition, we maintain a global state variable **is_Primary**, which sets the state of the server to be either primary or backup. Another global variable is **prim_conn**, which defines the primary connection that backup servers listen to.

Upon initializing our **replica.py** code, a server will reach out to the other server facing addresses that are predefined. In this manner, this server detects whether a primary server exists or not. If other servers are running while this initialization occurs, the incoming connection of a recognizable replica server is taken by the other servers, and they will send 0, if they are backup, and 1, if they are primary. The initializing server takes this tag and determines whether a primary server exists.



If a primary server does exist, then the initializing server stays with default **is_Primary = False**, and it receives .json files of the server state. For a currently running primary server, it correspondingly sends these over anytime a known replica connects to it. Then, the thread of **server_interactions()** that maintains live primary/backup behavior stays in the backup state, barring any changes. Note **client_interactions()** does not accept client connections until **is_Primary = True**.

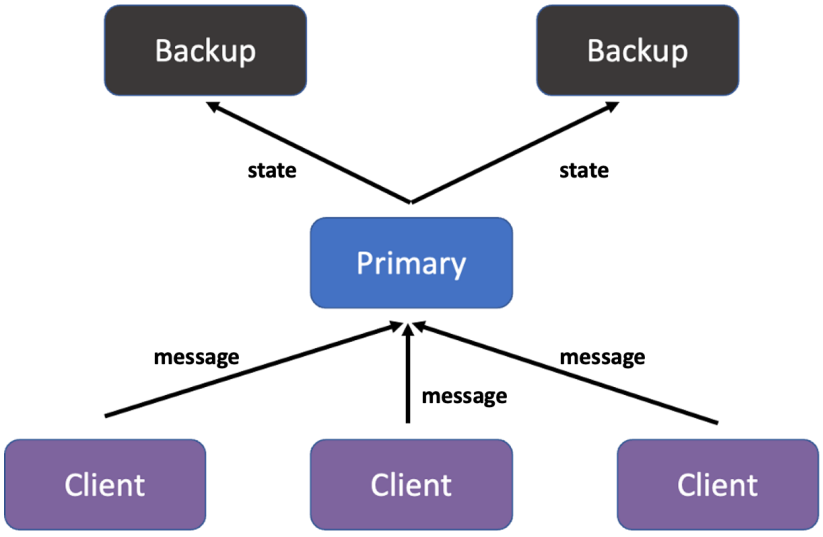
If a primary server does not exist, the initializing server assumes itself to be the primary. We have **server_interactions()** enter primary behavior, and **client_interactions()** enter primary behavior.

Server State and Primary and Backup Communications

The server state consists of a **client_dictionary** and a **message_queue**, alongside with a history of sent messages that were received by the recipient (i.e., not in message_queue for that recipient). All of these are persistently stored. The **client_state_dictionary** records what new accounts are created and what new accounts are deleted, while **message_queue** marks what messages have been sent by one user but not opened by the other. Notice that for backups, the login state of the user does not matter. Once a primary server is killed, the user is automatically logged out, and will need to log back into the new primary server. Locally, the logins/logouts are stored, and this does not matter when re-electing a primary server.

Now, let us focus on primary communications with backup servers. Again, any time a replica joins the network, the primary server sends it a 1 tag, followed by the .json files indicating server state and history of sent messages. Backup servers send it just a 0 tag. This is performed in **server_interactions()**. Also, the way a replica is recognized is that a joining replica will send its machine index (1, 2, 3) as a byte to the current replicas in play. The current replicas store the new connection in **replica_connections**.

Given a client interacting with the primary, and backups still online, we have a live method of the primary sending the backup state updates as the clients interact with the primary. This is conducted in **client_interactions()**, where upon reception of a message from a client, the primary server updates its own local state, and then sends a message (**with a new wire protocol**) to the backup servers, and the backups update their state as prescribed by **handle_message()**. The backup servers listen in the **backup_message_handling()** thread.



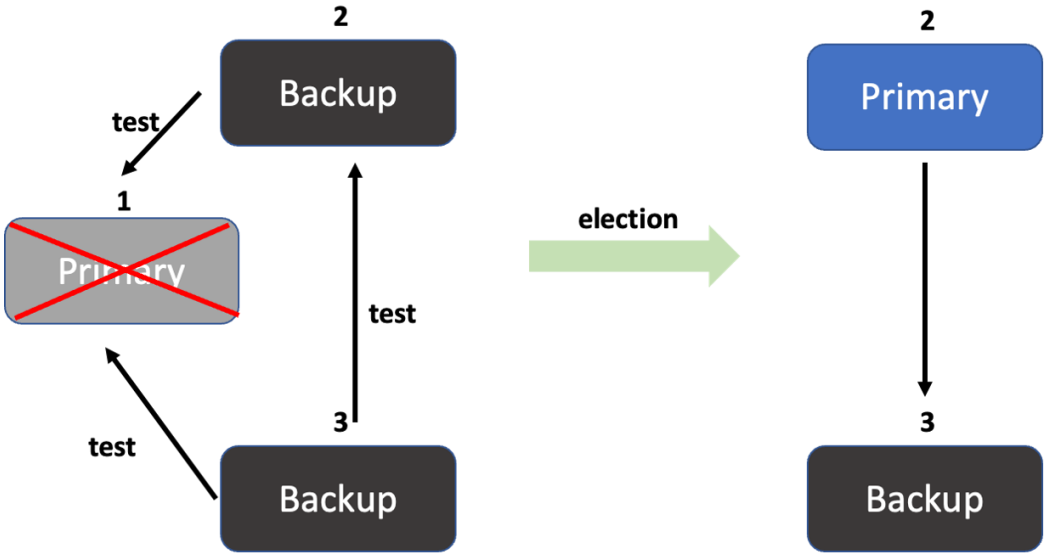
Let us now describe the primary-to-backup wire protocol. The primary attaches a tag of message type (see our design project 1 wire protocol for more information), followed by the length of the username and the username of the client servicing the state update. For Create/Delete Account/Dump states, this is all the information needed. The backups locally change **client_dictionary** to reflect new/deleted accounts, and upon receiving a Dump, sets the

message_queue to empty. These changes are propagated to disk by writing the new dictionaries to the pertinent .json files.

For sent messages, an additional tag, **queue_tag**, is needed to mark whether the recipient was logged in or out at the time. If logged in, the message is stored in the history of sent messages database, by the primary, and this sent message is likewise stored by backups. If logged out, then the backups append the text of the message in the list of unread messages stored in message_queue under the recipient's username. These changes are once again written to disk. In general, any time server state changes, it is written to disk for persistence.

Primary Failure and Leader Election

The previous describes what happens when a primary and backup are online. Upon failure of the leader, the continual message listening thread the backups have, called **backup_message_handling()**, will declare a returned message with empty bytes, signifying the connection with the primary is broken.



Our leader election process is that the lowest numbered currently live backup replica will become the primary. The way our code senses that for a given live backup, for machines 1 to the index of that backup (minus 1), that backup tests the connection with that server machine. If all these connections fail, the backup knows it is currently the lowest index replica running, and switches its state to the primary. If one of these connections works, the backup knows another primary will be assumed, so the backup remains a backup, and updates **prim_conn**, the primary connection global variable, accordingly, and listens to it in the **backup_message_handling()** listening thread.

Rejoining

Rejoining necessarily means that the code is reinitialized. Our reinitialization process automatically accounts for rejoining a replica, as if there is a primary, the replica will enter as a backup and receive the update server state from the primary.

Client Interaction

See Design Project 1 Notebook for details. When the primary goes down, the client must reconnect manually, although this could be easily refactored such that the client would redirect to the primary on the client side or with some load balancer.

Persistent Storage

Server state is persisted across json files local to each machine, with a database of the users and their expected auth state when rerouting to another server, a log of all messages sent in the system, and a database of all message queues. Anytime a change is made to state, the primary writes the change to appropriate databases, and then sends a message to the backups telling them exactly what change was made. The backups then parse this message, update their local state, and then persist the changes to the database. The system can be started and stopped while maintaining the state when running when asked to restart. When a server rejoins, the primary sends their current database over to the backup server to ensure synchronization. JSON files can be cleared by running python3 clear.py in the main folder.

Testing:

Preliminary tests we run to test replication are not unit tests, since the machines are distributed. To demonstrate 2-fault tolerance, we start up 3 servers, create two accounts and send an undelivered message on the first server. We kill the first server, and then connect a client to a second server and dump messages. Correct behavior is seeing the dumped message on the new primary. We repeat the same thing by killing the second server, and then dumping unreceived messages on the third server, showing how server state is not affected by the death of a primary replica.

To demonstrate persistence, on the third server as above, we send yet another unreceived message by the counterparty. We then kill the third server. Now, we **restart the most recently down server**, as this server has the most recent information. Restarting will load server state from persistent memory, and the receiver can dump this unreceived message stored persistently.

More tests: Modular functionality and edge cases for replication are included as unit tests in the sockets_test folder (handling message, leader election, DB writes and loads).

Installation/Usage:

To run an experiment, open three terminals (one for each machine) and input

python3 replica.py {machine_index}

to start the machine with index (1, 2, 3). To connect a client, the terminal input is

python3 socket_client.py {IP Address} {Port Number}

to connect a client to a predefined machine index. The client will have a list of IP addresses and corresponding Ports of servers, and will need to manually input this into the terminal. Upon disconnection, the user will manually input the IP Addresses and Port Numbers until reaching a server that is running.

Modules used for this are **socket, threading, _thread,** and **json**.