Engineering Notebook - Replication

Hanying Feng and Vera Yang

In this design exercise, we decided to use the JSON protocol code from the first assignment. To make the system persistent and 2-fault tolerant in the face of crash/fail-stop failures, we modified the code on both the server and client sides.

**Server-side Implementation**

**Server Initialization**

The server initialization process establishes the server’s identity, loads persistent data, and prepares the system for communication with clients and other servers. In addition to the host and port number of the current server, the *Server* class requires two additional arguments to identify the status of the server:

* *is\_primary*: Whether this server starts as the primary (leader). Defaults to False.
* *is\_rejoin*: Whether this server is rejoining the network. Defaults to False.

To satisfy the requirement of 2-fault tolerance, we need to start three servers, and each server must store information about its peer servers by calling the function *self.get\_backup\_servers()*, which returns a list of dictionaries containing host and port information for the other two backup servers. For each server, we use a JSON file (*accounts\_{port}.json*) to store user account information.

The server also needs to identify the leader in order to monitor its status. When the system first starts, we set the server with port number 5555 as the leader. If a server is rejoining the network, it will call the function *self.get\_leader(self.host, self.port)* to retrieve the current leader’s information. If the rejoining server is the only alive server, then it becomes the leader. Otherwise, it cannot be set as the leader.

Additionally, two critical background threads are launched:

* *send\_heartbeat* thread: The leader server continuously sends heartbeats to the other backup servers to check if they are alive.
* *monitor\_leader* thread: The backup servers continuously monitor the leader’s status and initiate an election if the leader fails.

The use of threading ensures that these background tasks do not block the main server operations, allowing the system to remain responsive to client requests while maintaining fault tolerance.

**Heartbeat Mechanism (Leader server)**

The heartbeat system is crucial for detecting failures and maintaining cluster health. The leader server sends periodic heartbeat messages (every 0.5 seconds) to all backup servers (*send\_heartbeat* thread). This short interval ensures rapid failure detection while balancing network overhead.

Each heartbeat is a message *{‘action’: ‘heartbeat’}*. Backup servers respond with an acknowledgment (*{‘status’: ‘ack’}*), confirming they are alive. If a backup server fails to respond, the leader continues operating. Since the system is 2-fault tolerant, it can handle two server failures before becoming unavailable.

**Monitoring Leader (Backup servers)**

Similarly, backup servers also continuously monitor the leader’s health to detect failures. They send periodic heartbeat messages (every 0.5 seconds) to the leader server (monitor\_leader thread). If the leader does not respond within the expected interval, the backup server assumes the leader has failed and immediately initiates an election, which ensures a new leader is quickly selected to maintain system availability.

**Leader Election**

When a backup server detects a leader failure, it initiates an election. In this system, the server with the smallest port number is set as the leader. Since there are at most two servers in the system when the leader crashes, the election process works as follows:

* If there are two servers alive in the system, the detecting server compares its port number with the port number of another alive peer server. If the detecting server has the smaller port number, it declares itself as the leader and send a message to the other server (*{‘action’: ‘new\_leader’, ‘host’: self.host, ‘port’: self.port}*). When the another server with the larger port number receives the message, it updates the leader information.
* If the detecting server is the only alive server in the system, it will set itself to be the new leader.

This approach guarantees that the most eligible server (the one with the smallest port number still online) becomes the new leader. Once elected, the new leader broadcasts its status to all backups, ensuring cluster-wide consistency.

**Data Replication and Consistency**

To ensure fault tolerance, the leader replicates all account and message updates to backup servers before responding to clients. This synchronous replication guarantees that data is not lost if the leader fails immediately after a write. The replication mechanism works as the following:

1. A client sends a request that needs to modify account information (e.g., sending a message, creating an account, deleting a message/account).
2. The leader updates its local state and saves it to disk (*accounts\_{port}.json*).
3. The leader broadcasts the update to all backup servers by calling the function *self.replicate\_data()*.
4. When the backup servers receive the message, they update the account information and save it to disk (*accounts\_{port}.json*).

This design ensures durability and data consistency. If a backup server is temporarily unreachable, the leader continues operating and will synchronize missing data when the backup server rejoins.

**Rejoin Mechanism for Failed Servers**

When a failed server rejoins the network, it queries other servers to determine the current leader and saves the leader information by calling the function *self.get\_leader(self.host, self.port)*. It sends an *{‘action’: ‘ask\_leader’}* request to all peer servers. The first server to respond with status equals to ‘leader’ is the current leader. The leader server will immediately sends the latest account data to the rejoining server by calling the function *self.replicate\_data()*, ensuring the consistency of data. This prevents data divergence and guarantees that the recovered server has the latest dataset before accepting client requests.

If no leader is found, which means the rejoining server is the only alive server in the system, it will set itself as the leader. The rejoining server cannot be manually set as the leader upon restart. This is to avoid data inconsistency, as the rejoining server may not have the latest dataset if other servers are alive and still operating in the system.

**Handle Client**

Unlike the simple chat application where the server only has to handle requests from clients, now the server needs to handle requests from other servers as well. For processing client requests, we largely maintain the code from before, with the only difference being that after each write operation we need to replicate the data to all servers. Thus, after each write operation (operations that will change the stored accounts data), we call the replicate\_data() function which will broadcast all changes to the backup servers. Since our system ensures that the client is always connected to the leader, the changes are always broadcasted from the leader to any backup servers still running.

Then, for processing requests from other servers, we add the following actions.

* heartbeat: if the server receives a heartbeat action, it will send ‘ack’ to indicate it is alive to the requester.
* new\_leader: if the server receives a message from another server claiming itself as the new leader, the server modifies its knowledge of who the current leader is, and prints out a message.
* ask\_leader: if the server receives a message asking if the server is the leader, it responds with status ‘leader’ if it is the leader and broadcasts its latest data, otherwise it responds with status ‘not\_leader’.
* update\_accounts: if the server receives a request to update its data (accounts) as well as the latest data, it will update its own stored data according to the received data.

Note that to distinguish between requests from other servers and those from clients, after handling each server request we need to close the connection. At first we did not implement this detail, and as a result when the server receives a request from another server, it thinks the request is from the client and continues to receive empty messages, which causes errors on the client side. Thus, we make sure that we close the connection immediately after handling each server request, as there is no need to maintain a session as is needed for the client.

In conclusion, this server architecture ensures 2-fault tolerance by starting with three servers (a leader and two backups). The system remains operational as long as at least one server is alive. The persistence is achieved through:

* On-disk storage (JSON files that save accounts information).
* Replication (all updates are propagated to backup servers).
* Recovery protocols (rejoining servers synchronize the dataset with the leader).

The combination of heartbeats, leader election, and replication ensures that the chat application remains available and consistent even under server failures. The design choices prioritize fast failure detection, minimal downtime, and strong consistency.

**Client-side Implementation**

**Connect to Server**

Before we only had one server, but now we have three servers and the client should be always connected to one of the running servers to maintain its operations. We construct a config.ini file that records the host and port of the three servers. When we run the client, we first read from the config file and organize the server information into a list of dictionaries of hosts and ports, which is passed as an argument to the client instance. To initialize the client connection, we loop through all servers and try to connect to the first running server. If the system is started properly with three servers running, the client should be connected to the primary server. If no server is running, the client prints out an error message reflecting the connection failure.

**Reconnecting on Failure**

To make the system fault tolerant, we need to ensure that the client can reconnect to any running server if the original connection fails. This is especially important whenever the client is communicating with the server. Thus, we implement a reconnect() function that loops through all the servers and tries to connect to the first running server. This function will be used in every client operation, such that whenever a connection error occurs during the operation, the client will try reconnecting to one of the remaining servers and the operation can be completed with the new connection. During reconnection the function prints out which connection trial fails (indicating that the server is down), and at the end which server it successfully connects to.

**Client Operations and Reconnection**

For each of the operations in the Client class, including send\_message(), read\_unread\_messages(), read\_messages(), get\_unread(), login(), list\_accounts(), delete\_message(), and delete\_account(), we need to try and catch the connection error while sending and receiving messages from the connected server. Whenever a connection fails, we call the reconnect() function to try to connect to one of the remaining servers. After a successful reconnection, we call the client operation again (inside the except block), such that the operation can be completed with the new connection. If the client fails to reconnect, an error message is printed out to reflect the connection failure.

**Client-Side Behavior & Multiple Machines**

After we implement both the server and client fault-tolerance and persistence mechanisms, we tried testing our system to see if from the client side the operation is smooth as the back-end servers goes down and rejoins the system. We observe that when the client is not communicating with the server and only does GUI-level operations, the operations are very smooth. When the client-side sends or receives messages from the server, as the current connection goes down, the client will experience a one-second freeze but the operation will be continued and successful. This freeze is because at the back end the client detects the connection failure and is trying to reconnect to another server. We also tried killing any 2 of the 3 servers and the client experience is still successful and continuous, and the data remains the same before and after the reconnection. This proves that our system is 2-fault tolerant and persistent. For the extra credit, we also implemented a rejoin mechanism. We test killing up to two servers and let the down servers rejoin the system. The rejoining server can successfully find who the current leader is and have its own data updated with the leader. We also test the system on multiple machines by setting the server hosts to the machine IPs and make sure that the servers and clients can be run from separate machines.

**Testing and Documentation**

We implement comprehensive unit tests to test the behavior of our system. For testing the chat app functionalities, such as account creation, logging in, sending and receiving messages, deleting messages and accounts, we maintain the test code from the simple chat app. In order to test our system with three servers, we add tests focusing on replication and fault-tolerance. We test if the servers are all successfully initialized and listening, and the client is connected to the primary server. We test 1-fault tolerance by killing one of the servers and testing if the client reconnects and the client-server communication is still successful. Similarly, we test 2-fault tolerance by killing 2 of the 3 servers and checking if the client operations are still successful. We also test leader election by testing if the leader election is initiated when the leader fails and if the correct leader is elected as expected. Lastly, we test data consistency between servers by checking if a change made to the data of the connected server is reflected in the data of all other running servers as well. For documentation, we add full docstring documentation to our files with detailed descriptions of each class and function. We also created a README.md file to facilitate understanding of the file structure and provide instructions of how to run our system.