Mar 17, 2025

Today was our first meeting. So, we have a very large project from the first two design projects that includes three different communication protocols: custom wire protocol, JSON, and RPC. We decided to clean up the code and refactor it such that it will sheerly leverage JSON and remove the two other remaining protocols. The reason is that we believe it is the easiest protocol in terms of coding for implementing 2-fault tolerance.

Also, up until now we had had a persistent storage using MangoDB since the project 1; however, we assume that it will be more harder to replicate servers with separate MangoDB instances than if we had each server writing to the local file on the disc. Thus, our plan is to remove MangoDB dependencies and implement a local file storage for our server.

In general, today we spent all our time refactoring the code and preparing it for the requirements of the replication. To be honest, we expected that it would take a couple of hours, but instead, we had to dedicate more than that on debugging due to the errors in the business logic when converting the storage from MangoDB to local files.

Mar 19, 2025

Today we discussed and started implementing the mechanism that would allows our server to achieve 2-fault tolerance.

We will use the **primary-backup model** where one server acts as the primary and others as backups. This model provides a clear, simple approach to maintaining consistency by designating a single server as the authority for all write operations. Moreover, it simplifies consistency management by circumventing concurrent writes and reduces complexity compared to peer-to-peer replication models. In particular, we will probably just use 3 replicas to keep the architecture simple as much as possible but we suspect that the number of servers should not matter substantially as long as the architecture is correct.

Data Replication
In order to avoid single point of failure, each server will maintain its own complete copy of the data (local file document that stores data in JSON format). It also enables independent recovery of each replica’s storage files.

Leader Election
Inspired by the Raft consensus protocol, we will implement term-based elections that provide a clear mechanism to prevent split-brain scenarios. In addition, we will add heartbeats that provide continuous validation of the primary's health and network connectivity.

Then, followingly, there will be a voting meachanism where leader is selected democratically with majority support. To be specific, all servers start up in CANDIDATE state. When a backup server doesn't receive heartbeats from the primary within a timeout period or when the current primary detects a higher term from another server, a new election is initiated.

So far, we have programmed data storage replication and the leader election logic in the replication manager.

Mar 22, 2025

Today, we continued our work of developing the replication manager from the last couple of days. Apart from that, we have to adjust our client side to handle the replication mechanism, Our main focus lies on these features:

Write Operations
The write operations into the storage files will be centralized, i.e, only primary server writes. This logic prevents conflicts and ensures a single source of truth along with sequential consistency of operations. When a client sends a write operation (like sending a message or deleting a user account), here's what happens according to our plan:

- If the client connects to the primary server: The primary processes the write directly and also replicates the same operation across all backup servers.
- If the client connects to a backup server: The backup forwards the request to the primary server. The primary sends the operation to all backup servers. Each backup applies the same operation to its own state. It will provide consistency across independent server storages.

Client connection
In the beginning, the client tries to connect to one of the servers (ports 8091, 8092, 8093) in sequence until it finds an available one. If the server the client is connected to fails, then the client will automatically reconnect to available servers. If a backup server fails, then the primary detects the failure when it can't replicate operations to the backup, and the system continues to function as long as a majority of servers (2 out of 3) are still available. When the failed backup comes back online, it can rejoin the system and catch up. In case, the primary server fails, one of the remaining servers becomes a new leader through election. Importantly, all of these re-election and re-connections procedures between servers remain hidden from the client. This approach truly achieves 2-fault tolerance, enhances user experience, and reduces client reconnection overhead during failures.

Single server remaining
Also, we had problems when only server remains alive because in that case it is impossible to get majority of votes. Hence, we decided to automatically make remaining single server a leader.

Server re-connections issues Also, we tried ot implement the re-connection of servers that went offline back to the cluster but it is a very complicated process; thus, we decided to drop this idea. Now, the client can automatically reconnect to other available servers but a failed server itself remains offline until manually restarted. There is no self-healing mechanism where servers automatically attempt to rejoin the cluster after failure.

Mar 23, 2025

Today we tackled configuration management. Our initial implementation had many hardcoded values scattered throughout the codebase:

- Server IP addresses
- Port numbers
- File paths

This made deployment and testing in different environments challenging. We decided to implement a comprehensive configuration system that would allow these values to be set through environment variables or configuration files.

The most significant changes were made to:

1. `setup_distributed.sh` - Now loads environment variables for all server configurations (used to run multiple servers on multiple machines)
2. `client_socket_handler.py` - Server addresses now come from environment variables
3. client.py - Connection parameters are configurable

This change will make our system much more adaptable to different deployment scenarios and easier to maintain in the long run.

Mar 26, 2025

Testing

Today we conducted extensive testing of our fault-tolerant chat system. We simulated various failure scenarios:

1. Graceful shutdown of the primary server
2. Abrupt termination of a backup server
3. Shut down of 2 servers out of 3

Our testing revealed a few issues:

First, we discovered a race condition in the leader election process. If multiple servers initiated an election simultaneously, it was possible for the election to fail to elect a leader. We fixed this by adding randomized election timeouts inspired by the Raft consesnsus.

Second, we found that our file-based storage wasn't properly handling concurrent access during replication. We implemented file locking to ensure data integrity when multiple threads access the storage files.

Apart from that we performed unit testing and integration testing, and our system passed all of them.

Finally, we documented the entire system architecture, including the replication protocol, failover procedures, and configuration options.

Overall, we're satisfied with the robustness of our implementation. The system can now tolerate up to 2 server failures while maintaining data consistency and availability, which meets our original design goals.