Identity Server (Phase II: Replication for Reliability)

Abstract

In this project, we will take the Identity Server from p2 and replicate it in order to improve reliability. Most of the design choices will be determined by the Raft replication algorithm (https://raft.github.io/raft.pdf).

Client Server Model

We are opting to do a one-shot client with separate servers because we believe it would be easier to implement in relation to determining the coordinator. In the servent model, we would need to consistently keep updating the processes id list every time a new client/servent joins the system. We would also need to take into account how to modify the process list if a a client/servent decides to fully leave the system. This would make the coordinator algorithm a bit more complex, which leads us to choose one-shot client because in that model we would only need to keep track of the servers that we need to put on to the system.

Coordinator

The election algorithm we want to implement to determine the coordinator is the Raft algorithm. In the Raft algorithm, servers are in three states: leader, candidate, and follower. The leader handles clients, log replication, and sending heartbeat pings to other servers. Candidates are servers that start an election to be coordinator after they timeout requesting a heartbeat ping from the leader. These servers request for votes from other servers. Followers are passive servers that only respond to requests from the leader or candidates. They sync data with the leader and can initiate an election when they notice that the leader is missing, at which point, they become a candidate.

The election proceeds as such:

1. Every server in the system has a “term” number stored.
2. There is one leader server that handles clients, data replication, and sending heartbeat pings.
3. All the rest of the servers are followers that just response to requests from other servers and listen for heartbeat pings from the leader.
4. After not receiving a heartbeat ping for a predetermined time (defined as the timeout time), a follower will increment its “term”, change to a candidate, and broadcast to other servers that it started an election and is requesting votes.

5. When a follower receives a vote request, it will compare its term with the received term. If the received term is higher, the server will respond to the candidate with its vote and update its term to the received term, otherwise it will ignore.

6. A candidate wins an election when it receives a majority of the vote.

7. If a candidate does not win an election by the timeout period and it has not conceded to another candidate, it will re-increment its term and start another election.

Note- if multiple elections are running, a follower will only send its vote once per election on a first come, first served basis; if a candidate receives a vote request from another candidate, it will follow the same logic a follower and revert back to a follower if it concedes to another candidate that has a higher term.

Consistency Model

We chose to use the sequential consistency model because we want to prioritize keeping track of the order of commands sent from the client to the servers that affect the database. The main areas that we're concerned about in relation to read and write sequential order are the commands to create, modify, and delete user information. If this data is not properly stored sequentially, there could be a lot of cases where the data inconsistencies could create big problems (such as different users trying to create with the same login name at slightly different times or user information being deleted or modified near the same time).

Our inconsistency window would be the maximum election timeout. This comes from the fact that any data sent between the last ping from the old leader and the end of the election for the new leader would be lost.

Server Synchronization

Our system will keep track of time using log indices, which are similar to Lamport clocks in that the log index monotonically increments at each entry. We plan to keep server data synchronized with log messages because each log would be the individual database change, which eliminates the need for checkpoints.