Our implementation uses Zookeeper for coordination between servers to recover from failures and to elect a new leader whenever the current one fails.

Leader Election:

The function “checkLeader” checks for the leader whenever any server boots up. The way it works is by having each server create a sequential ephemeral znode at the “/election/znode_{seqID}” path and the server which has the lowest sequence number gets picked as the leader by default. If the leader crashes, the next server owning the znode with the minimum sequence number gets elected as the leader. Every server watches the “/election” path and whenever a server crashes(its respective znode gets deleted), it gets the update that some node went down and every server checks whether it was the leader that crashed and if so, gets the new leader (which is the server with the next smallest sequence number”) and update it’s leader variable accordingly.

Logging and Coordination:

The coordination step involves each server finding out if the leader died and taking the necessary steps. If a follower server dies, nobody except the leader does anything(the leader maintains the dead server’s log as explained in the Leader Logging section). If the leader dies, everybody elects a new leader and updates their leader so that they can send the next client requests to that leader.

Leader Logging:

The leader logs each request message from another server before inserting the message in its queue. This log is maintained at the “/service” znode where only the current leader logs its queue of requests and where the new leader picks up from in case the old leader dies. The leader logs each request before inserting it in its message queue and further removes it after broadcasting it as a proposal(even if the leader dies here, everybody would receive the proposal and execute the request). After broadcasting the request to the alive nodes, the leader also logs the request at each dead server’s persistent znode at the path “/service/myID” so that the follower server knows what it missed when it comes back up after a failure.

Follower Logging:

Each follower server, including the leader itself (the leader is also its own follower in our implementation), has an associated znode at the path “/service/myID” where it maintains a proposal log of up to 400 requests after which it checkpoints the application state and clears the log. See Crash Recovery for more info.

Crash Recovery:

Checkpointing:

The functions exportDataToCSV and restoreDataFromCSV serve as the drivers for checkpoint/backup creation and consumption. ExportDataToCSV does a select * from all the tables the keyspace has and parses the output into a CSV, located in a local directory. RestoreDataFromCSV parses this CSV and executes a bunch of insert statements after truncating the table.

Checkpoint writing occurs for each server after logging MAX_LOG_SIZE messages of type: PROPOSAL. When checkpointing happens, the proposal log is cleared at path "/service/myID". Checkpoint restoring happens on startup during crashRecovery.

We acknowledge that we have a problem with our checkpointing system. If a server goes down, we have it so that the leader logs the requests for the downed server by writing to its znode. If the log reaches MAX_LOG_SIZE while the server is still down and the leader is still writing to it, it is possible for the proposal log to reach a size greater than MAX_LOG_SIZE.

Follower Recovery

After a server starts up again after crashing, it re-creates its ephemeral znode in the leader election path "/election/znode_{seqID}" in order to make itself available for leader election again. The node also checks if the leader has changed.

Then it enters crash recovery, which involves first restoring from the last checkpoint if it has one, truncating the table, and inserting the data back in. After checkpoint restoration, the server then reads its proposal logs from the "/service/myID" persistent znode. The server parses the logs and re-executes the commands. The node is then ready for normal operation.

Leader Recovery

Every time the current leader dies, our system unanimously decides to choose the next leader as the next sequence number ephemeral znode at the “/election” path. Whenever a server finds out that the current leader has died and it is the new leader, it calls the function “recoverLeaderQueue” which basically gets the queue of requests logged at the “/service” path and runs them one by one as the normal leader would i.e broadcast the request to all alive nodes, log the request in the dead server’s respective znodes at the “service/serverID” path, wait for acknowledgments from all alive servers and then execute the next request in the queue until the queue becomes empty. After this, the leader resumes its normal operations as a leader.

Results & Anomalous Behavior

Sometimes our test cases fail because of a Keeper Exception Problem. Since we are doing the leader election part ourselves instead of just inquiring who the leader is from the zookeeper. The way it works is that the first server that boots up creates a “/election” and “/service” path followed by a child path to conduct leader election and store logs respectively. We are handling the case when the parent path was already created by some other server and in that case, the child server just creates its child path under that parent. However, if two servers boot up at the same time and try to create this parent path at the same time, our code generates a Keeper Exception and some or all of our test cases fail in those scenarios.

As said in the Crash Recovery section, we are also aware that in some rare cases, logs do go past MAX_LOG_SIZE.