Executive Summary

Assignment Overview

This purpose of this assignment is to understand and implement the Paxos algorithm. The scope of the project is:

1. Understand the basic algorithmic steps of the Paxos algorithm. Being able to understand the roles that are involved in the algorithm — the proposer, acceptor and the learner. Understand how consensus is achieved by using sequence number comparison, the concept of a quorum, how acceptors makes decision based on the sequence numbers, how proposers decide which value to propose etc.
2. Understand how the failure cases are dealt with. In the Paxos algorithm, any server/role could potentially fail. The use of persistent logs are essential to deal with the failure cases. When an acceptors fails (which is a hard requirement for this project), it can read from the persisted logs and retrieve the highest sequence number it has seen, as long as the accepted proposal/value. If a learner fails, we could persist all previous committed operations in a separate log. When the learner restarts, it can redo all previous operations and back to its previous states.

Technical Impression

I put together a code result walk-through pdf file in the “src/snapshot” directory and it has a detailed walk-through of some testing I’ve done and how I believed happened during the Paxos algorithm process. During implementation of the project, the most difficult part is to understand the algorithm itself. I tried to gain more understanding of the algorithm through reading different references, discussing repeatedly with classmates, professor and TAs. Some part of the algorithm is counter-intuitive. Professor brought up a good point that not all client requests got executed and that’s OK, because the goal is to achieve consensus, to remain consistent across all server replicas. Once the understanding is there, the coding part is pretty straight-forward. The main communication channel between client/server, proposer/acceptor and proposer/learner is still through Java RMI, which makes it straight-forward. I composed a few utilities class such as Promise and Response class which makes the code cleaner and more maintainable. I was able to get familiar with some Java I/O methods, such as creating files, writing to files, resetting file contents and doing them synchronously and asynchronously as needed.

The basic Paxos algorithm is pretty straight-forward, however some corner cases (especially failure cases) are tricky. There’re some corner cases that I have yet to address. For example: 1. What happens if an acceptor crashed and jumped back in again with stale log? Consider this scenario: An acceptor crashed and it took some time to get restarted. when it joined back, it’s already a completely new round of Paxos process. When the cracked acceptor restarted, it read some stale data such as accepted proposal value from the previous Paxos round (that has already been completed). However, the acceptor itself doesn’t know that and continued to proposed the stale data. 2. What if there’re more than half of the acceptors that failed and took a while to restart? Before they’re completely restarted, I guess no proposals can move forward because a quorum can’t be achieved at all. These corner cases probably require a more thorough and complicated implementation. I’m very curious that how real life enterprise/ products implement a much more robust version of Paxos to address all corner cases.

Further Enhancement

There’re many aspects where the project can be further improved.

1. Right now, only the learner is multi-threaded. The proposer can be made multi-threaded to increase efficiency. Each client request is handled by a separate thread (a round of Paxos algorithm on its own). Similarly, the acceptor could also be multi-threaded. Wasn't able to implement this part due to limited time and experience with concurrency programming.
2. As pointed out in many references, there's a chance for live lock when having multiple proposers. The way to avoid that is electing a "leader" proposer that initiates all the proposals.
3. Can also make the learners fault-tolerant by persisting all previous committed operations. Whenever a learner restarts, it reads all the previous committed operations and returns to its previous state.