Project 4: Report

Introduction

In this project, I mainly implemented the Paxos algorithm using the PaxosServer which was supplemented by the client, KVStore, and AMOApplication from the previous projects. The goal of this project was to improve Project 3 by adding more liveliness and more fault tolerance even if a server fails. In Project 3, if the primary dies before a backup is created, the service could not recover, but Paxos overcomes this by having a leader election mechanism. Initially, I followed the PMMC paper for the Paxos algorithm code, however, later I found the Paxos implementation in Project 5 to be easier to understand so I have built my implementation on top of that code. The following are the new notable code and features added to this project:

- PaxosServer is the server that implements the Paxos algorithm and communicates with the clients and other PaxosServers. It behaves differently based on whether the server is a leader or a non-leader. A leader election is started at the initialization of these servers and also when some server does not receive a heartbeat from the leader within a certain timeframe.
- New timers have been added: HeartbeatTimer, HeartbeatCheckTimer, P1aTimer, and P2aTimer. The first two timers are for the leader and non-leader servers to keep track of whether the leader is alive or not. P1aTimer and P2aTimer are timers to resend phase 1 and phase 2 messages of the Paxos algorithm due to network loss.

Flow of Control & Code Design

The basic idea of my Paxos implementation is that there is the acceptor role and leader role, but they aren't really roles, they are just specific message handlers in the servers. Every server act as an acceptor, but only one server acts as the leader. The first step is for the servers to choose a leader and once that leader is chosen (phase 1), all client requests will be acted upon by the leader (phase 2).

At initialization, the servers set their leader and acceptor ballots and issue a phase 1 request for leader election. Each server will send out a P1aMessage to all the servers including itself to see if they become the leader. The acceptor role will accept the highest ballot number as the leader on a rolling basis and notifies that to every server. The servers wait for their ballot to be accepted and if they get a majority vote from the acceptors that their ballot got accepted, they become the leader.

When the server becomes the leader, it applies all the Pvalues of each slot from each acceptor's PvalueRecords to a map of pvalues for each slot. It takes the highest ballot for each slot. This is then used to update the log of the server by marking that log slot as ACCEPTED and storing the request. Then the phase 2 request is issued even though the server just became the leader.

Phase 2 is only done by the leader and it is a way to propose a request to be chosen by the acceptors so that it can be executed in the KVStore. In phase 2, the server takes all the client requests that were sent to the server and checks to see if they are already executed or already proposed. If they are, those requests are ignored. If not, the logs are updated to say they are ACCEPTED and store the request in the corresponding slot. This ensures that all the incoming client requests are either already executed or are in the ACCEPTED state in the log. Then, the server will iterate from the slotOut value to slotIn – 1 value of the log to propose those values to the acceptors by sending a P2a message. The slotOut and slotIn values are updated in the updatePaxosLog function and so it keeps track of the newest values of slotOut and slotIn.

The acceptor handles the P2aMessage by checking if the acceptor's ballot number is the same as the one sent by the leader. If it is, the acceptor creates a new Pvalue for the message and adds that to the corresponding slot of its log. The handler also either starts a heartbeat timer or heartbeat check timer depending on whether the server is a leader or not. If it already has one of the timers for the current leader, it keeps track of the current heartbeats. At last, the acceptor sends back a P2b message to the leader only if the acceptor's ballot is the same as the leader's ballot.

The leader waits for the majority of the acceptors to send back the message. Once it has the majority, it updates the log by making the corresponding slot to be CHOSEN and sends back a reply to the client.

In terms of the timers, the P1a timer is set after broadcasting the P1a message to all servers in Phase 1. This is to make sure that if a leader is not elected before the timer is up, the server will try to send another p1a message with a higher ballot number so that it can get elected. The P2a timer is set after broadcasting the p2a message to all servers during phase 2. This is to make sure that if the leader does not get a majority before the timer is up, it sends the p2a message again in hopes to get the majority vote. The heartbeat timer and heartbeat check timer are set in the acceptor when it handles the p2a message. This is because the first message all servers receive after a server becomes the leader is the p2a message. The leader will periodically send the heartbeat timer and the non-leaders will keep track of the heartbeat check timer. The heartbeat message is sent every 25ms and the heartbeat check timer checks if the last seen heartbeat is within two timeframes of the heartbeat check timer (150ms). If the heartbeat is not received

on time by some non-leader, it assumes the leader is dead and the non-leader will issue a phase 1 request to start a new leader election.

For garbage collection, it is done within the updatePaxosLog function which is called in 4 different places: when a server becomes leader, issuing a phase 2 request, handling a p2a message, and sending the decision after getting majority for a proposal. When a server becomes the leader, it gets all the Pvalues it has received from the acceptors to update its log. During this time, the log is cleared for the slots that have CHOSEN has the log status. When issuing a phase 2 request, the server sets the slot for a request to be ACCEPTED and during that time, it clears the logs. When the acceptor accepts a request, it sets the corresponding slot to be ACCEPTED and also does garbage collection there. Finally, when the proposal gets the majority vote, the log is updated to CHOSEN for the corresponding log slot and the CHOSEN slots are to be garbage collected after sending the reply to the client.

The code is heavily influenced by the Paxos algorithm implementation of Project 5 as I have used it to build the missing parts on top of it. Originally, I referenced the PMMC code (see my repo's git commit history) for implementing Paxos, however, the Project 5 implementation seemed to be more intuitive, hence I re-structured my code to fit the current flow. The most notable difference in code design compared to the pure PMMC implementation is that there are substantially fewer roles and actors involved in this implementation. In the PMMC code, there were replicas, leaders, acceptors, commanders, and scouts. However, in this design, there is one server that handles all the roles at once by using the message handlers. This implementation has the benefit of not needing to create inner classes and not needing to send additional messages such as the messages from replica to leader or making commanders and scouts.

Design Decisions

In terms of the design decisions, I initially design my Paxos algorithm based on the PMMC code. However, I believe that the implementation was overly complex since it had many roles and two maps (decisions and proposals) to keep track of the log. So, I ended up referencing the Paxos implementation existing in Project 5 and based my design on that. The Project 5 implementation had a PaxosLog hashmap that was intuitive and easy to visualize as it was a union of the decisions and proposals maps in the PMMC implementation.

Due to the change from PMMC to Project 5 Paxos implementation, several changes needed to be made to the implementation design. Instead of having roles such as replica, leader, and acceptor, I merged the replica and leader roles into the server and also designed it so that the handlers take care of the messages

instead of rerouting them into specific roles. This made things much simpler as roles were just an extra layer to add unnecessary complexity to the servers.

In the initial PMMC implementation, the commander and scout were made in phases 1 and 2. Spawning these were not ideal in our project so I had to design a workaround for that. Instead of using these roles, for each p1a and p2a message, I would have a hashmap called proposeRequestWaitForMajority and a pair called waitForP1b to keep track of whether the messages got the majority vote. This design allowed for a more compact implementation compared to the PMMC implementation.

In terms of how my implementation caused me to revise my design, in my initial PMMC implementation, I had specific roles such as the replica, leader, and acceptor and many inter-server communications happening. This was not ideal since we can reduce the number of messages sent if we merge the replica and leader. Hence, I chose to revise my design by referencing the structure of Project 5's implementation and keeping message delivery to a minimum.

Upon reflection, if I were to develop a project like this next time, I would spend more time digging down into the implementation itself as it was more delicate than I expected it to be. I spent too much time worrying about the design and did not have enough time to get into the details of specific cases and scenarios where Paxos might fail.

Missing Components

I am only passing approximately half of the test cases (1-10, 12-14, 21, 24) for this project.
I believe I did implement every feature that needs to be in this project to pass the tests, but it seems like there are some (many) bugs and edge cases that I have not considered to pass the tests. Since the tests that I am failing are unreliable (network loss) test cases, there may be some issues with how I am using my phase 2 timers to reissue the proposals. Unfortunately, I did not have enough time to complete a fully functional Paxos implementation.

References

I referenced the course repository for the instructions:

https://github.students.cs.ubc.ca/CPSC416-2022W-T2/project4-paxos

Project 5 Paxos implementation as a baseline for my current implementation:

https://github.students.cs.ubc.ca/CPSC416-2022W-T2/project5-shardkvstore/blob/main/lab/src/dslabs.paxos/PaxosServer.java

PMMC Paper:

https://paxos.systems/