UoE-DS-Coursework-Part3

Introduction

Your distributed lock manager has made significant progress and has become fault-tolerant. You have released the fault-tolerant version of your software to a few early adopter customers. Everyone has provided positive feedback so far. However, just as you start to feel confident in your design, your CEO calls you into a meeting with a worried look on his face. He says a few customers were complaining about the Mean Time To Repair(MTTR) when the lock server goes down. They can't afford to wait for service to resume, especially with critical workloads depending on us. You realize he's right. While your system is fault-tolerant, it's not highly available. When the server crashes, clients experience noticeable downtime as the system waits for the server to recover. This delay is unacceptable for your customers. Suddenly, memories of your Distributed Systems course flood back. You remember the lectures on replication and how crucial it is for achieving high availability. To solve this, you decide to add replicas of the lock server. With replication, even if one server fails, the others can seamlessly continue providing service, minimizing downtime. However, you also recall the complexity involved, like keeping replicas in sync and maintaining consistency. If you enforce strong consistency, every lock operation will be slower, as all replicas must synchronize before confirming the request. If you weaken the consistency, the system will be faster, though some replicas might temporarily hold outdated information. You sit at your desk again, start thinking about what to do, and pray that your CEO won't bother you anymore.

Administrivia

- **Project Due Date**: 18th November, 2024 (noon)
- **Questions**: We will be using Piazza for all questions.
- **Collaboration**: This is a group assignment. Given the amount of work to be done, we recommend you work as a group (3 members). Working individually on the project will be a daunting task.
- **Programming Language**: C, Python, or Other. We will only help debugging with C and Python.
- **Tests**: Some test case descriptions will be provided.
- **Presentation**: We will ask you to present a demo that effectively showcases your system's functionality and correctness. You will be graded based on your presentation. You will have to show the functionality that will be posted towards the deadline.
- **Design Document**: You will also submit a 2-page document that discusses the design details and the system model, assumptions, etc.
- **Office Hours**:
    - Office hours will take place in Appleton 4.07 from 1 to 5 P.M. every weekday, starting October 21, 2024. When there is a lecture on that day, office hours will begin 30 minutes after the lecture ends and will still last for 4 hours.
    - During office hours, we will prioritize design-related questions and are happy to help you understand the broader picture. Programming errors will be given lower priority. We can surely help with coding issues if time permits, but design questions can take a VIP pass and cut to the front of the line.
    - Please email Yuvraj to schedule office hours with him.

Background

To complete this part, you'll need a good understanding of replication, availability, and consistency.

- **Distributed Systems (MVS & AST), Chapter 5 & 7 & 8**
- **Replication**: https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/statemachines.pdf
- **Consistency**: Baseball Game
- **Raft**: https://opencourse.inf.ed.ac.uk/sites/default/files/https/opencourse.inf.ed.ac.uk/ds/2024/raft.pdf
- **Paxos**: https://opencourse.inf.ed.ac.uk/sites/default/files/https/opencourse.inf.ed.ac.uk/ds/2024/paxos-simple.pdf

Please also refer to Part 1 Specification for setting up the environment.

Specification

For this part of the coursework, you need to improve the availability of your lock server.

You have two options:

1. **Simple state machine replication**

or

2. **Consensus algorithms such as Paxos or Raft**

As you add replicas to your system, it's important to consider how to keep these replicas consistent.

You will also need to modify the client library depends on how you change the lock server. For example, the client should update its cached server address after the current server is down and it chooses a new server for sending requests.

Understanding consistency models

The consistency model plays a crucial role in how failovers and state transitions are handled.

- **Strong consistency model**, all replicas have the same state as the primary at all times. When the primary goes down, a replica can seamlessly take over as the new primary with up-to-date information. However, it can also slow your system down and hurt the availability. The primary waits for acknowledgments from all replicas to ensure they have the newest state before proceeding. If one replica goes down, the primary will not make any progress until that replica is back.
- **Weak consistency model**, replicas may have stale information; updates from the primary are not guaranteed to be immediately reflected. If the primary fails, a replica might think the lock is free when it is actually held by a client. There is no guarantee on how stale the replica's information might be, leading to potential conflicts and violation of atomicity. One possible solution is for the new primary to reject all ongoing requests and wait until the client releases the lock, but this can lead to delays and reduced availability. Or clients can inform the new primary about the status of the lock.
- **Bounded consistency model**, replicas can update after every lock release. In this case, replicas are guaranteed to be out of sync with the primary only within a known bound. Upon primary failure, the new primary knows that any inconsistency regarding lock ownership is bound. It can wait until the current lock ownership expires and move on.

By carefully selecting the replication strategy and consistency model, you can enhance the availability of your lock server while maintaining the necessary levels of performance and correctness.