UoE-DS-Coursework-Part2

Introduction

A week has passed since you began the project, and your Distributed Lock Manager (DLM) is shaping up well. The client library and server are functioning as expected. You are testing the code to figure out all the corner cases are taken care of. You are now considering the infrastructure within your startup where the distributed system would run. At the same time, the CEO knocks on your door and tells you that the network infrastructure is unreliable and the nodes may crash. You argue why they can't buy better equipment. However, the CEO explains that buying reliable equipment is costly, and they cannot afford it at this point. The CEO is wondering if he fires you, maybe he can buy the equipment. You are aware of that idea, too. You love your high-paying job. Additionally, you are planning to buy a house (a villa in the South Morningside area) in a few months. So, you cannot afford to lose the job.

With both hands on your head (and slightly worried about losing the job), you start thinking about what faults can happen that you need to worry about. You realize that while the perfect network assumption was convenient, real-world networks are far from perfect. Messages may be delayed, dropped, or duplicated, and nodes may crash mid-operation. Your system needs to gracefully handle these failures without introducing bugs like deadlocks or inconsistency in the append-only log.

Suddenly, the Distributed Systems course you took flashes in your mind again. One particular lecture on fault tolerance stands out, and you realize you are about to encounter the same challenges. Now, it is time to tackle fault tolerance.

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 network failure and fault tolerance in distributed systems.

- **Fault Tolerance**: Lecture 3 Slides, Why Do Computers Stop and What Can Be Done About It?
- **Idempotency**: RPC Paper
- **Distributed Systems (MVS & AST), Chapter 4 & 8**
- **Recovery**: https://dl.acm.org/doi/10.1145/128765.128770, The design and implementation of a log-structured file system

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

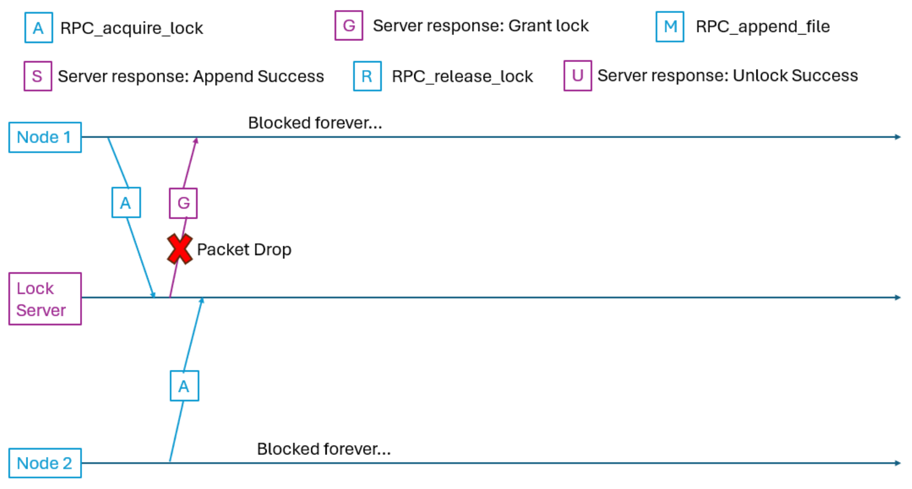
Part 2 Specification

After completing part 1, your DLM is able to handle client requests under the assumption of a perfect network and no fault happens. You are now facing challenges such as crash failures and network partitions. You want to make sure the system functions well while the network packets may be delayed, duplicated, or lost, and nodes may crash. You will have to think about the two-general's and byzantine general's problems as they model the network and nodes behavior, respectively. Let us start with network failures.

Network Failures

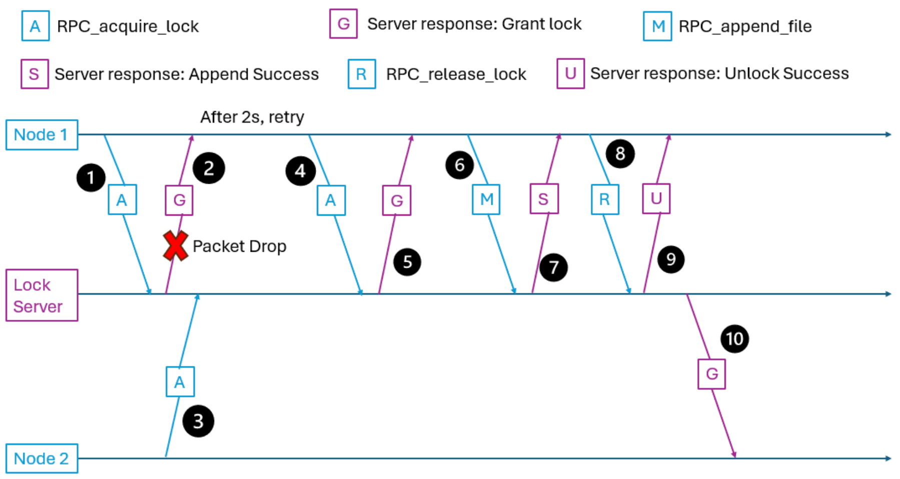
Case 1: Packet Loss

Recalling the thought experiment we had during the lecture, either the client's request or the server's response could be lost. In the example below, node 1 sends the `lock_acquire` RPC to the lock server but waits indefinitely for a response that never arrives, causing it to be blocked forever. The worst thing is that as node 1 is not aware of its lock ownership and will not release the lock, node 2 will be blocked forever too. The liveness property of the system is broken. Similarly, if a client request never reaches the server because it is lost in the network, that client will get blocked forever.



Goal 1: In this scenario, the challenge for you is to ensure the liveness property by implementing retry mechanisms in your RPC library that allow clients to take further actions if they do not receive responses from the server.

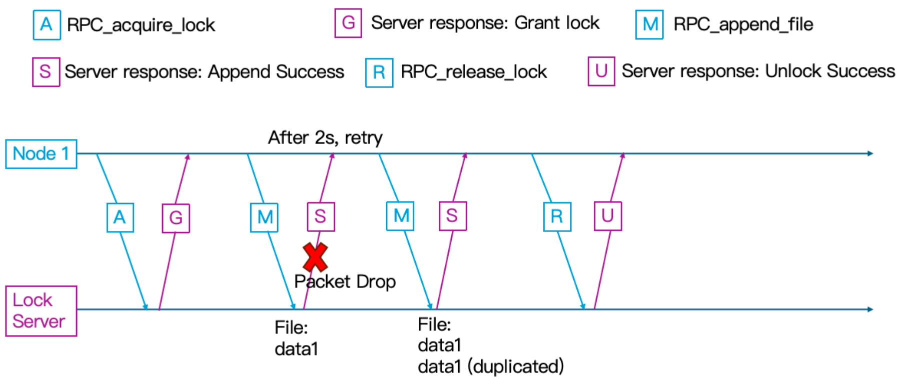
The workflow below shows an example of the retry mechanism: node 1 sends a `lock_acquire` RPC to the remote lock server(1). The lock server grants the lock to node 1 but the response message is lost(2). Node 2 sends a `lock_acquire` RPC to the server and waits for the lock(3). After two seconds of no response, node 1 sends the request again(4), and the server will send a response again(5). Finally, node 1 acquires the lock and can start modifying the file...



Case 2: Duplicated Requests

Wow, you seem to address the above problem, both the client and the server can make progress when they do not hear from each other, protecting your system against packet loss. Unfortunately, this introduces a new problem: what happens if the server successfully receives a request, but the client does not know and sends the same message again?

(After thinking it through, let's break it down together...) First, consider the case where the server's response packet is lost. In the following Figure, node 1 sends a `file_append` request, but the server's response packet does not arrive, node 1 wants to resend the same `file_append` request in order to urge the server and make progress. If the server processes every request without careful checks, it would append the same data twice, violating idempotency. Similarly, the server should not process `lock_acquire` and `lock_release` requests multiple times, as this could cause safety issues (e.g., node 1 might release a lock meant for another node).

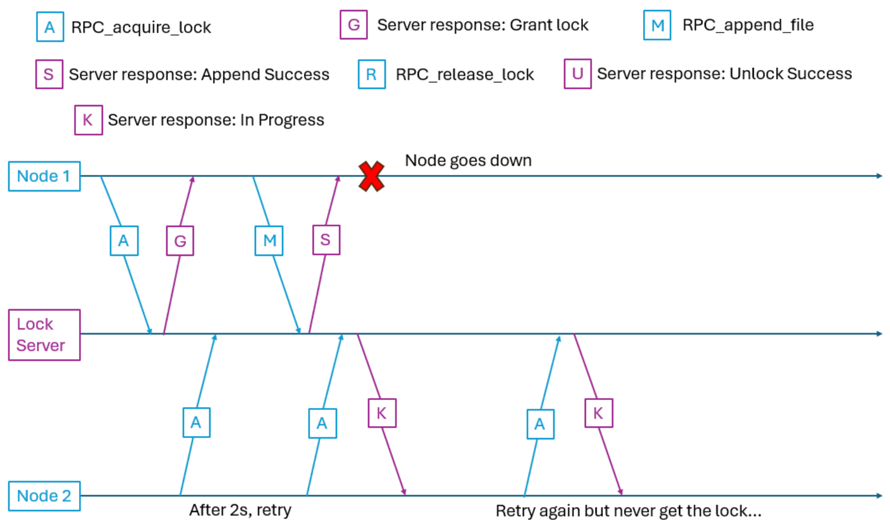

A RPC_acquire_lock
G Server response: Grant lock
M RPC_append_file
S Server response: Append Success
R RPC_release_lock
U Server response: Unlock Success
Node 1
After 2s, retry
Packet Drop
Lock Server
File: data1
File: data1 data1 (duplicated)

Goal 2: In this scenario, the challenge for you is to ensure the safety property, it is quite important that each operation should not be processed more than once. Also, upon receiving a duplicated request, the server should return the appropriate status code (corresponding to the one processed previously) to the client.

Note: If the server successfully receives a request and starts processing it but is slow, the client may still not receive the response in time and retry. You can make your server more responsive by communicating the status of ongoing requests back to the client when it receives a repeated request. This allows clients to discover the request status and take corresponding actions (e.g., wait longer) before retrying again. Alternatively, for simplicity, you could also ignore duplicate requests and return the final status only after completing the operation.

Client Crashes

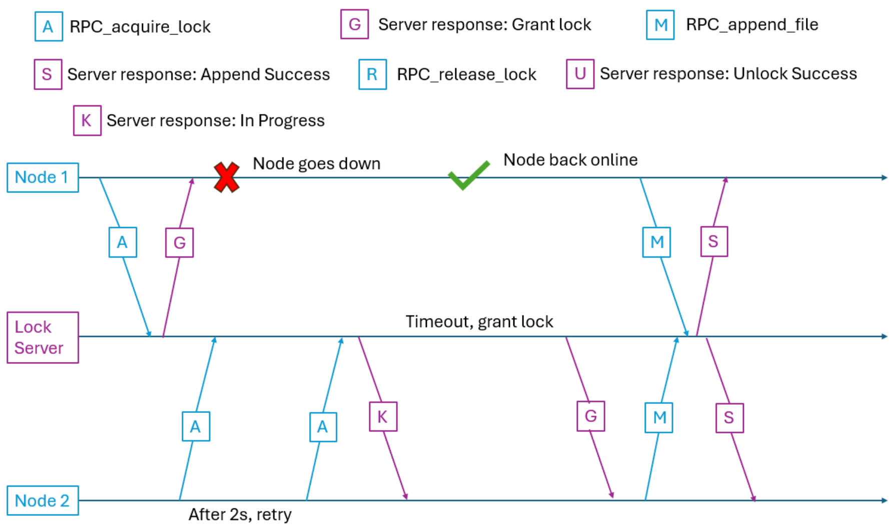
Upon this point, you have handled network failures well. However, it's still not enough. Consider the problem shown in the following Figure: if node 1 crashes as the lock holder, it will not release the lock, and node 2 will wait for the lock forever.


A RPC_acquire_lock
G Server response: Grant lock
M RPC_append_file
S Server response: Append Success
R RPC_release_lock
U Server response: Unlock Success
K Server response: In Progress
Node 1
Node goes down
A
G
M
S
Lock Server
A
A
K
A
K
Node 2
After 2s, retry
Retry again but never get the lock...

Goal 3: Your next challenge is to prevent a lock holder from holding the lock indefinitely. The server should set a deadline for each lock holder; after this deadline, the lock will be passed to the next waiting node.

Unfortunately, another correctness problem is introduced. **You may want to complain that there are countless faults :(It feels like just as you solve one issue, another one pops up, this is a common theme in distributed systems. The thought of getting fired or quitting the job flashes in front of you. However, you are smart and know that you are the best. You will overcome these issues and manage to impress your CEO and make him regret not taking the Distributed Systems class.**

Take a breath and look at the following example: Node 1 acquires the lock first but then crashes immediately. Because we have implemented a deadline for Node 1, its lock ownership expires, allowing Node 2 to acquire the lock and begin appending data to the file. Meanwhile, Node 1 comes back online, still believing it holds the lock, and also appends data to the file. As a result, the shared file ends up with corrupted data.



Goal 4: This time, your challenge is to address this issue and protect your file. Ensure that only the current lock holder can modify the file and that only the current lock holder is able to release the lock.

Server Failure

In a crash-recovery model, the server can restart after a crash and recover its previous state from logs or other persistent storage. Ultimately, it will resume functioning and handle client requests. If not managed properly, this can lead to data loss, inconsistent states, or stale information if the server recovers without the necessary context. For instance, if Client 1 previously held the lock but the lock is granted to Client 2 after recovery, it can create issues.

Goal 5: Your final challenge is to address server crash failures. You need to carefully consider this, as it will be relevant for Part 3.

Clients' Perspective

When a client retries a request multiple times (ensure you define a maximum retry count), it may conclude that the server is down if it still receives no response. Think about what actions clients can take instead of helplessly waiting for this non-responsive server. Don't worry if you don't have an immediate solution; this is your challenge for Part 3, and you will handle it properly. Just relax now!

Implementation & Hints

For part 2, you will be adding fault tolerance features to your lock server and client library to achieve **Goal 1-5**, which addresses various types of failures. Remember, the RPC paper is your sword and provides all the information you need to manage network failures.

Additionally, consider using a monotonically increasing number to identify the lock holder, similar to the concept of a logical clock. This approach can help solve client crashes.