UoE-DS-Coursework

Introduction

You recently graduated from the University of Edinburgh with flying colors and have started working in a startup. The startup initially was building a stand-alone server. Their product is gaining traction and now they are wondering about extending the stand-alone server into a distributed system. Your CEO recalls a conversation you had about the Distributed Systems course you took at the University and asks you to lead the project. He skipped taking the Distributed Systems class while he was at the University. You are given 4 weeks to complete the project. Your task is to design a fault-tolerant and highly available append-only log that is accessible by a distributed set of clients. You decide to break the entire problem into three parts -- something that you learned while working on your distributed systems coursework. You take a deep breath sit down at your desk and start thinking about the project. You want to make the CEO regret not taking a Distributed System class in the past. **Of course, you may understand the regret of taking the Distributed System class.**

In distributed systems, coordinating access to shared resources among multiple nodes or processes is a fundamental challenge. Distributed locking is a mechanism that ensures mutual exclusion, preventing concurrent access that could lead to inconsistencies or race conditions. It extends the concept of locks from single-machine, multi-threaded environments to distributed systems composed of multiple machines connected over a network. It allows processes running on different nodes to synchronize their actions when accessing shared resources, ensuring that only one process can hold a lock on a resource at any given time. A Distributed Lock Manager (DLM) is responsible for providing this coordination, allowing processes in different nodes to safely acquire and release locks on shared resources. It is a critical building block of many cloud applications such as databases, file systems, and in-memory storage.

After a while, you realize that you will be implementing a distributed lock manager. For the first part, you need to implement a client library that can be integrated into a client application, and a server that can handle client library requests. Next week, you will start worrying about the other two parts -- fault tolerance and availability. You know that your clients and servers may crash (and restart) and you need to support the crash-failure scenario. Clients and servers may operate independently and at different speeds. They communicate with each other via messages. Your network is generally reliable. However, your message receipt is asynchronous and can take a longer time to be delivered, can be duplicated, and lost in the network. Messages are never corrupted. You can assume that your setup is trustworthy and no malicious behavior is expected.

Administrivia

- **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 hour will take place in Appleton 4.07 from 1 to 5 P.M. on every weekday, starting October 22, 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 threading, mutual exclusion using locks, and Remote Procedure Call (RPC) protocols. Basic networking communication knowledge will also be helpful, though the necessary networking code will be provided.

- **Introduction to Threads**: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf, https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf
- **Python Threading**: https://docs.python.org/3/library/threading.html
- **Locking**: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf
- **Remote Procedure Call**: https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf
- **Distributed Systems (MVS & AST), Chapter 4 & 5**
- **Distributed Systems: Concepts and Design, Chapters 16 & 17**

Objectives

As mentioned on the course website, on completion of this course, the student will be able to:

- Outcome 1: Develop an understanding of the principles of distributed systems and be able to demonstrate this by explaining them.
- Outcome 2: Being able to give an account of the trade-offs which must be made when designing a distributed system, and make such trade-offs in their own designs.
- Outcome 3: Develop practical skills in the implementation of distributed algorithms in software so that they will be able to take an algorithm description and realize it in software.
- Outcome 4: Being able to give an account of the models used to design distributed systems and to manipulate those models to reason about such systems.
- Outcome 5: Being able to design efficient algorithms for distributed computing tasks.

The coursework will help you with Outcome 1, Outcome 2, Outcome 3, and Outcome 4. Additionally, you will

- Get experience with multi-threading, RPC, client-server architecture
- Understand the challenges of synchronization in distributed systems

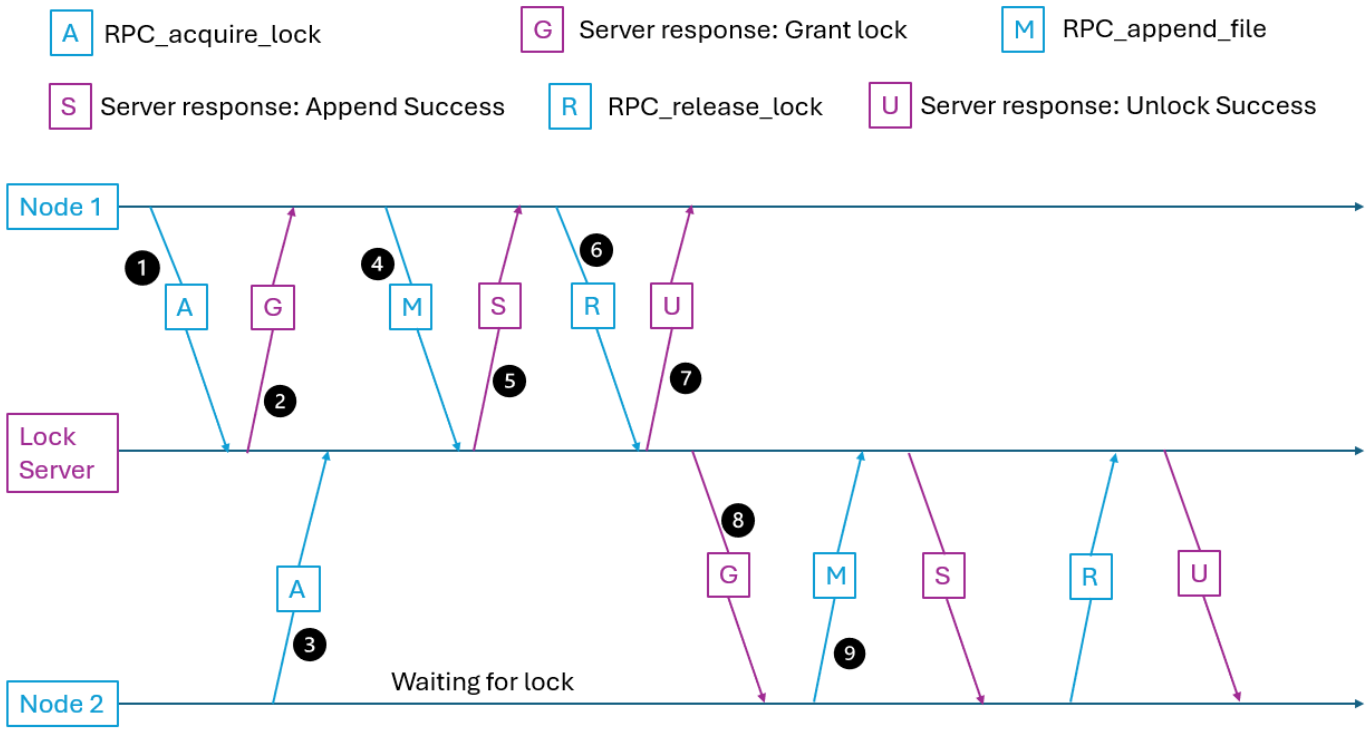
Environment

This project should be done on a UNIX-based platform(Linux). You are welcome to use any of the DICE machines or Virtual DICE. For Apple users, you can also refer to this guide and set up a Linux VM within your local machine. For Windows users, WSL is an excellent choice. Clients and servers can be run in any fashion: process, container, or VM.

Project Specification

In a distributed system, services run on multiple nodes, and local locking mechanisms alone cannot guarantee consistency across these nodes. To ensure that resources are correctly synchronized between nodes, a distributed locking mechanism is required. To make lock distributed, your first task is to create a simple lock server and a client library that allows a client to request locks from the server via Remote Procedure Calls (RPC) assuming a perfect underlying network. RPC is a popular method for simplifying distributed system programming, as it gives the illusion that a function is executed locally, even though it runs on a remote system. Whenever a client wants to access a resource, it must first acquire the corresponding lock from the lock server. For simplicity, in this project, the resources(files) will reside within the lock server.

Example Workflow: Basic Case



Whenever a client wants to modify a shared file (a log), it must acquire the lock first. Doing so sends a lock_acquire RPC to the remote lock server(1). The lock server will create a new thread to handle this request, if the lock is free, it sends a message back to the client and that client acquires the lock(2). If the lock is currently held by someone else, this client is blocked and waiting for the lock(3). Once a client acquires the lock, it can modify the shared file by sending the append_file() RPC to the lock sever(4), and the server will append the data to the target file and reply to a success message(5). After a client finishes modifying a file, it will release the lock by sending a lock_release RPC to the server(6 and 7). The server then can grant the lock to the next waiting client(8), and that client can modify the file(9).

Example Pseudocode

```
Client Code                                              Server Code
                                                         ...waiting for clients' requests
RPC_acquire_lock();
...sent a lock request to the server
...waiting from the server
                                                         ...receive request from client
                                                         ...creates a thread to process it
                                                         ...finish request, inform the client
...lock acquired
RPC_append_file();
...append new data to a file
...waiting from the server
                                                         ...receive request from client
                                                         ...creates a thread to prcoess it
                                                         ...finish request, inform the client
...data appended
RPC_release_lock();
...release the lock
...waiting from the server
                                                         ...receive request from client
                                                         ...creates a thread to process it
                                                         ...finish request, inform the client
...lock released
```

Implementation

To implement the RPC, you can use `rpcgen`, `grpc`, socket programming, or any possible solutions. We describe the implementation details from the socket programming perspective as it gives a fundamental view of the project.

Client Library

The client library needs to consist of the following functions:

- RPC_init()
  - This function will initialize the rpc_connection struct such as an open socket and bind it to a port, fill up the address of the server
  - send a very simple request to the server for testing purposes
- RPC_lock_acquire()
  - This function will send a lock acquire request to the server
- RPC_lock_release()
  - This function will send a lock release request to the server
- RPC_append_file()
  - This function will send the append file request to the server including the name of the file append to and data to append
- RPC_close()
  - This function will do the cleanup work such as closing the socket To note that, a client should be blocked until it receives a response from the server.

Socket programming

You need to implement a client library so that clients can use it to acquire the lock, write data to file, and release the lock from the remote server. For socket programming, a basic `rpc.h` file and codes for communication `udp.h`, `udp.c` are provided. Client and server use functions in `udp.h` to communicate with each other. If you decide to use Python with socket programming, please check this resource: Python Socket Programming.

rpcgen

We provide a `lock.x` file for `rpcgen` (and a `lock.proto` file for Python) that defines all function prototypes, helping you understand the overall client-server communication structure. While detailed arguments for client requests and server responses are not included, you are encouraged to add any fields that you find useful for implementing the lock and file functionalities described below.

To generate client and server stubs, client and server functions, and a makefile using `rpcgen`, execute the following command:

```
rpcgen -a lock.x
```

This will create all the necessary files for your RPC program, including the makefile.

Note: If you add additional `.c` or `.h` files to your implementation, make sure to update the generated makefile accordingly and add your new source files to the appropriate sections. E.g., for client-side sources, modify the `SOURCES_CLNT.c=` and `SOURCES_CLNT.h=` sections. This ensures your new code will be compiled and linked properly.

For Python with gRPC, please check the `lock.proto` file and the following command:

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. lock.proto
```

Lock Server

The lock server will handle all the RPC requests from clients and reply correct return codes them. It should contain the following implementations:

- init
  - reply a message like "Connected!" to the client with a return code of 0
- lock_acquire
  - grant the lock to the client if it is free or wait until the lock is acquired
  - returns 0 to client if the lock is acquired
- lock_release
  - release the lock and pass the lock ownership to the next waiting client
  - returns 0 to client
- append_file
  - find the file that the client want to modify and append new data sent from the client to it
  - returns 0 to client if success
  - returns 1 to client if file not found
- close
  - reply with a message like "Disconnected!" to the client with a return code of 0

The server is required to have the following functionality:

1. Create 100 files that clients can write. The file name should strictly follow this format "file_0", "file_1", ..., "file_99".
2. Maintain one lock object, we use one lock to protect all the files for the sake of simplicity. In real world, there can be one lock per file.
3. Receiving packets from a client and spawning a new thread to handle it. This will allow for multiple clients to connect at the same time. To note that, servers generated by `rpcgen` do not have multi-threading, you have to implement it.

Server-side lock implementation

We provide codes of spinlock that you can use to implement `lock_acquire` and `lock_release`

The spinlock has two interfaces: `spinlock_acquire(spinlock_t *lock)` and `spinlock_release(spinlock_t *lock)`.

Thread calls `spinlock_acquire(spinlock_t *lock)` will try to acquire the lock which is the argument passed to the function. It will return immediately if the lock is free, marking the lock as acquired. If the lock is currently acquired by another thread, it will wait until its turn. To implement `lock_acquire`, you can create a `spinlock_t` object and simply let the thread handling the request call `spinlock_acquire(spinlock_t *lock)`, once it is return from `spinlock_acquire(spinlock_t *lock)`, the lock is automatically acquired and it can inform the client.

In contrast, `spinlock_release(spinlock_t *lock)` will pass the lock ownership to the next waiting thread or mark the lock as free if there are no other waiters. Similarly, you can use this function to implement `lock_release`.

Handling multi-thread concurrency

The lock server needs to handle multiple client requests concurrently, requiring a multi-threaded design where different server threads can execute client requests simultaneously. It's always helpful to use local locking to protect shared data.
Please refer to https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf and https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf for programming using threads.

Grading and Submission

Each of the three parts is worth 25% of the total points. The presentation accounts for 15%, and the design report is worth 10%. If you only complete some parts of the project, you can still earn points by finishing the presentation and design document.

Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance on the School page [here]. This also has links to the relevant University pages.

The deadline and late policy for this assignment are specified on Learn in the "Coursework Planner". Guidance on late submissions is at [this link].

Hints

You should start the project by implementing RPC_init(). RPC_init() serves as a hello world test. The purpose of this function is to let you understand how RPC works by printing a "hello world" message on a remote server.

Given the daunting task, you realize you will need to start working on the project sooner than later. Your second and third parts may need some thinking.