CSCI 555 Intermediate Report

Team members: Chieh Nien, Junkai Liang

Goals

Overview

Our goal is to implement the Chubby, a distributed lock service.
When Chubby's paper was published, Paxos was the pervasive consensus algorithm, so Chubby's original consensus protocol was Paxos-like. However, Raft was designed in 2014 and was intended to be more understandable while providing similar levels of fault tolerance. Meanwhile, Raft has an internal leader, which we think may be more suitable for Chubby.
So, we decided to implement a simple version of the Chubby system based on the two consensus protocols - Paxos and Raft and compare their performance differences. Then, we would try to figure out the reasons for the differences.

Changes from Proposal

In our proposal, we planned to use some existing Paxos and Raft protocol libraries. However, after we had completed the survey, we found some difficulties.

1. Most libraries are for more than just consensus protocols. They mainly focus on applications based on Paxos/Raft, and they hide the details of their basic architecture. The descriptions are Paxos-like or Raft-like, so it is hard for us to find and use the pure API of their consensus protocols.

2. In our evaluation, we want to test the availability when there are different kinds of failures. Due to our limitation, we do not have multiple servers, so we had to run multiple instances on a single machine. It is hard to shut down one instance and restart it back online using the existing libraries. Also, when using these libraries on localhost, they get nearly no delay and no package drops, which differs significantly from reality.

3. We want to compare the performance difference when using these two different consensus protocols, but optimizations of existing libraries may import ineligible bias to our results.

Therefore, we decided to change our plan for the reasons mentioned above.
To solve problem 1, we want to implement our own Paxos and Raft protocol and try to abstract and unify the API for Chubby. Also, we will implement only the essential functions from the original papers so that we can observe the differences purely caused by these two different protocols, which solves problem 3.
We use Golang for this whole project, which has built-in RPC calls. However, to solve problem 2, we will not use the built-in RPC but an RPC library that can simulate the delays and package drops based on settings.

Progress so far

RPC Library

After clarifying our requirements, we conducted extensive and in-depth research. We ultimately found that the RPC library developed for course 6.5840 perfectly aligns with our needs.
It was developed in Golang; it has easy-to-use APIs to make a component of the network disconnect and reconnect and to make network partitions; it can simulate random delays within a specified time period and simulate packet loss based on a specified probability.
This will significantly help our evaluation and make our results closer to the situation in reality, which means several servers communicate with each other through the Internet.

Paxos Protocol

Functions

We have implemented a simple Paxos protocol based on the paper titled "Paxos Made Simple." The protocol strictly follows the proposer-acceptor scheme described in the paper. It can choose a value and reach consistency under different situations, including multiple proposers, less than half of the servers offline, etc.

Persister

As we need to evaluate the crash and recovery of the servers, we implemented a persister for the Paxos server, which can persist the needed information for the server and let it catch up when the server restarts.

Correctness

To convince ourselves that the protocol functions correctly, we also built a whole test suite to verify its functionality. We tested it in the unreliable network simulated by the RPC library mentioned ahead, and also, under the situations where there were concurrent requests, the protocol behaved just as expected. We believe it is now correct and can be used as the basic consensus protocol of our Chubby system.

Roadblocks

It was hard for us to debug the distributed programs, as they produced totally different results in each trial of tests. We had to implement highly granular logging to record changes in every part of the system to find the problem. As we could foresee, we spent much time debugging and getting the protocol correct. Inspiringly, we have finally made it.

Raft Protocol

Functions

We have also implemented a simple Raft protocol based on the paper titled "In Search of an Understandable Consensus Algorithm (Extended Version)." The protocol has essential functions, including electing a leader, maintaining a connection with peers, requesting a vote when the leader is down, appending an entry to the log in a consistent order, log compaction, and notifying the up-level application when reaching consistency. For simplicity, we did not implement some of the optimizations mentioned in the paper, such as reading without contacting other peers and so on. We believe that at least the correctness of the Chubby system is non-related to these features.

Persister

Similar to Paxos protocol, we need a persister to record the logs and other information to restart a Raft server after it crashes. In Paxos, there is a forgetting mechanism so that the log will not ever grow. So, we also implement the log compaction function in Raft to reduce the log size. We will persist the snapshot and only the log entries after the checkpoint.

Correctness

We built a whole test suite for the Raft protocol as well. It tests the protocol for election functionality, basic agree functionality, reaching consistency under different kinds of failures, correctly handling concurrency and unreliable networks, recovery after crashing, and so on. We believe it can fulfill our needs to build the Chubby system.

Roadblocks

As we mentioned before, debugging distributed programs is always a challenging story. We encountered different bugs during the implementation. We need to consider all the corner cases, build the test according to them, and make the protocol work as expected for these tests. It was a time-consuming job. It is also worth mentioning that we conducted stress tests on our programs, and they could function correctly under a workload lasting at least 24 hours.

Chubby

Components we plan to implement

Although Chubby's paper mainly describes it as a lock service, it can be used for many other purposes and has many more functionalities than a simple lock service. The first job we have done is to figure out which parts of the paper we are going to implement.

- Files, Directories, and Handles
  Our goal is to use Chubby as a coarse-grained lock service and only focus on the lock acquisition and release performance.
  In paper, Chubby exports file system-like interfaces that support both files and directories. In a simple locking service, we see locks as files, and we only need to provide a map between path names and actual files, so we choose only to support files, which means the path name is seen just as part of the file name. One of the critical uses of the directory is that a node inherits access control lists (ACLs) from its parent directory by default. Since we assume all clients are under control and do not need access control, we will not implement the ACL part.
  We keep the design of the handles. However, it is more like an abstraction rather than a specific implementation. We will record the handle's information on each client.

- Locks and Sequencers
  We are going to take most of the designs described in Section 2.4 of the paper, including sequencer and lock-delay unless we only support exclusive locks.

- Events
  In the paper's design, Chubby clients may subscribe to a range of events, which greatly helps programming. However, again, we want to focus only on the locking service, so we will not support any of the events.

- Caching
  In Chubby, clients can cache file data and meta-data in memory. However, in our evaluation, we want to figure out the differences between Paxos and Raft protocols. Caching data on the client's side will reduce the communication between clients and servers and among servers, mitigating the performance impact of underlying protocols. So, we decide not to implement caching.

- Sessions, KeepAlives and Fail-overs
  This is a significant part of the distributed system's correctness so that we will follow the paper's design of sessions and keepalive RPCs. The performance of failovers is also a significant part of our evaluation.

- Backup and Mirroring
  Backup and mirroring are important in a productive system, but they are separate from our test. So, we will not implement this part.

System Structure

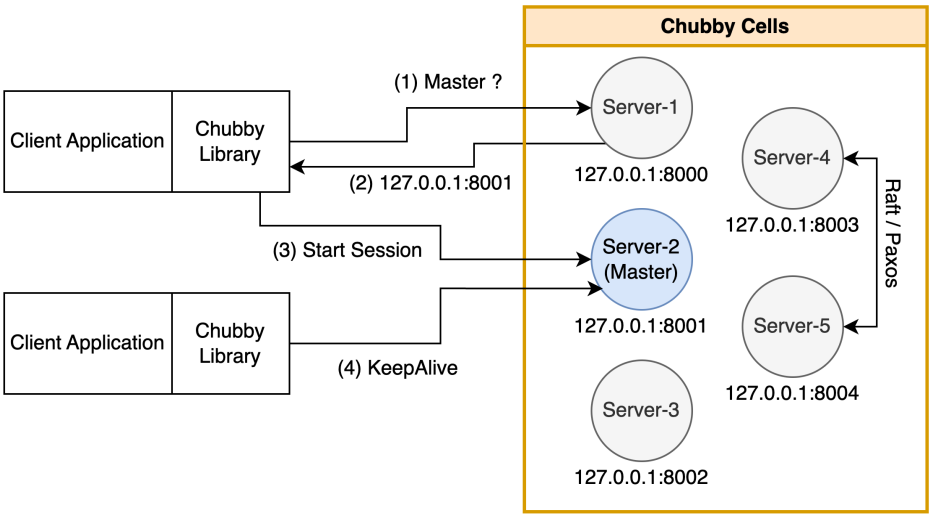
Chubby has two main components that communicate via RPC: a server and a client library. A Chubby cell consists of five servers known as replicas. In our current implementation, servers will run on our local machine and listen on different ports. In the original system structure, the client contacts the DNS to get the listed Chubby replicas. However, this function is not necessary for our evaluation; we assume clients already know the addresses of all possible servers.

Upon initialization, a Chubby client performs the following steps:

- Client read the server list to know all the Chubby replicas
- Client calls any server via RPC
- If that replica is not the master, it will return the address of the master
- Once the connection succeeds, both the server and client create a new session.
- Maintain session via KeepAlive call

While a Chubby server performs these steps:

- Open a TCP server at the given port
- A master is chosen among all replicas using Raft or Paxos
- All replicas become aware of the master and keep consist via Raft or Paxos



Locks and Files

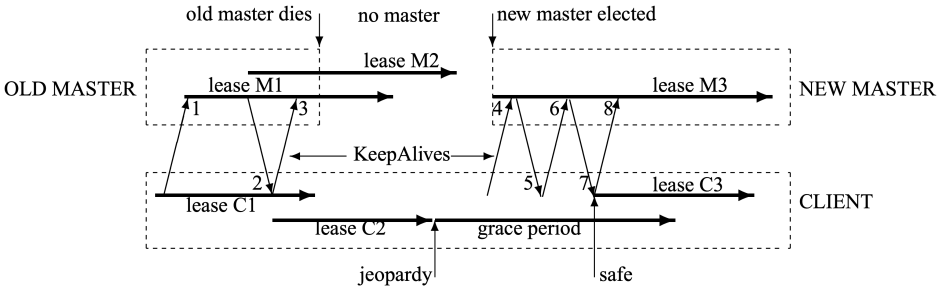
The state machine of our system is simply a map between the path name and file meta-data. We use the built-in map in Golang, with the path and file names as the key and the pointer to the file struct as the value. This simplifies the path looking up, and since there will be only a few pairs in the map, the performance of the built-in map should be sufficient for our evaluation.

Sessions

The functionalities of the "2.8 Sessions and Keepalives" section in the paper have all been implemented. A client requests a new session on first contacting the master. Sessions have associated leases guaranteed by the master. For each client session structure, it maintains lease length, master address, start time, and locks, while for each server session, it maintains lease length, start time, client ID, and locks.

KeepAlive

KeepAlive requests are used to maintain the session (extend the corresponding lease time). If the client's session leases timeout, it enters the jeopardy state. During the grace period, the client keeps sending RPC calls to another server in order to connect to a new master.



Plan of remaining time

Since we changed our minds about implementing consensus protocols on our own, we have spent more time than planned here. However, we also get some benefits. We can abstract similar interfaces of the Paxos and Raft layer, so we only need to rewrite a tiny part of the Chubby layer to make it work for the other protocol.
For the remaining time, we plan to finish implementing the system in one week, which leaves about three weeks to conduct the evaluation and write the final report.