CSC458 Final Project Report

Ziliang Lin

May 6, 2020

Overview

In this term project, I am trying to implement the multi-paxos consensus algorithms for a simple key-value system, which is describe on [1]. The Paxos algorithm is the core of all the further consensus algorithms like Raft, and as an algorithm it doesn't take many practical issues in consideration so an implementation for better understanding is very important.

Goal

As stated in the project proposal, the goals are the following:

- Implement a single machine version that can pass all the tests for [1], which covers the cases of network partition, unreliable message passing, deaf nodes;
- Implement a cluster version and customize tests for it to pass;
- Realize the tradeoff when implementing consensus algorithms.

Support & Infrastructure

- Programming Language: Golang (for the single machine version), Elixir/Erlang (for the distributed version)
- Hardware: My two normal commercial laptops;
- Software: test codes for [1] (gained via [2]) test code from [3].
- Reading & Video: [4], [5], [6], [7]

Progress

- Finish the first goal (Golang version), passing all the tests from [1]; it contains deterministic testings for basic correctness, forgetting (old, decided entries), deaf nodes, partitions and nondeterministic testings for unreliable rpcs. Due to this reason, I have run the test for 10 times for each version to make sure the correctness (each run of test consumes 1-2 minutes).
- Finish the distributed version (Elixir/Erlang), translate basic Go tests in [1] into Elixir and pass them (not all due to the lack of time).

Implementation Issues

So far, multi-paxos has been a general framework for applying multiple instances of basic paxos for replicated logs and many issues are not realized until I really go to implement it:

1. Mutual Exclusion. In multi-paxos, multiple log entries (i.e. with different indexes) can be proposed simultaneously, and for the same index there can be different proposals, with the log as a shared source on each node. Furthermore, during this whole asynchronous process, any potential new proposer may need to ask for the state of an entry before starting to propose – if the entry has already been decided, there is no need to really propose it. Therefore, we need to provide mutual exclusion for accesses to the log. If we introduce mutex locks to deal with this, deadlocks need to be avoided and there is a tradeoff for the granularity. Adding a lock for each log entry would cost much memory while a coarser lock will undermine concurrency for proposals on different but adjacent indexes. The key-value map also has this issue.

2. Repeating requests handling. When the client waits for a moment and finds that its request is still not yet fulfilled, it will resend the same request. Then it could be the case that the last proposal for this request just becomes chosen. There must be a unique id for each client request and the nodes need to keep this in log to find that whether a request is already satisfied and avoid repeating another proposals for it. Worse still, the nodes which don't realize the majority's decision but receive the resended request will need to eventually learn this fact from other nodes.

3. Execution semantics. After deciding a command requested, a node may fail before responding to the client. Then if the client resend the request to it, the node need to find that whether it has already executed the command. To ensure at-most-once semantics the node would need to mark the entry as executed first before actually executing it.

4. Forgetting old entries to save storage. It's impossible to append the log entries forever and the nodes need to negotiate with each other to find which entries can be forgotten; if they forget entries not seen in slow nodes, then full replication fails. Thus they should exchange the information about the last index of the entry executed during the paxos instances.

Learned from Languages

For the same algorithm, the programming experience between Go and Elixir/Erlang is quite different:

1. Go is pretty easy to learn with; there are not many concepts to grasp to write some meaningful code (anyway, most programmers are sure more familiar with C and Java than Lisp and Haskell). For example, though it's object-oriented, there are no "classes" but only "interfaces" in Go. Error handling is also straightforward, most functions will return an extra value to indicate whether the execution is successful. In contrast, Elixir could be harder to pick up; it's process-oriented and you will have to first learn functions, modules and processes linking and so on before finally having an "object" that can hold states, which shall be very basic and at the beginning for many other languages.

2. Go is imperative while Elixir is functional. This differs greatly in that there are no "variables" or "objects" to hold states in Elixir. Instead, Elixir uses agent processes (Erlang VM's processes, different from those running in a OS and are much lighter) to hold states; whenever a program need to read/write the current state, it would need to send a message to the agent synchronously. An advantage of this is that you don't need to consider about mutual exclusion; the agent process will receive all the messages in a sequential way and will handle one by one. However, you would need to either write a agent process or use the common "reduce" operation paradigm in functional programming language to implement a simple counter. On the other hand, I need to handle mutex locks in Go version well to avoid deadlocks. By the way, due to the more advanced semantics in Elixir, finally the Elixir version has around 600 lines and the Go version has around 850 lines (excluding the tests).

3. In Elixir/Erlang, details about communication among servers could be hided by Erlang VM; there are not much differences coding servers for a single machine or a cluster. You even don't need to consider the concrete protocol and ports for remote communications (but still you can explicitly select tcp/udp). This is typically demonstrated by the `GenServer` module in Elixir. Communications between local `GenServer` server processes and remote `GenServer` server processes are almost the same (except the name registration). Actually my first runnable Elixir version is already ready for distributed settings.

4. So far, Elixir provides better interactive experience. It has a built-in REPL, where you can loop a server and interact with it on your will, or even interact with remote Elixir REPL. And since Elixir is based on Erlang VM (i.e., Elixir codes are first compiled into Erlang VM bytecodes and then executed), all the powerful tools in Erlang can be used for Elixir. In addition to a debugger setting breakpoints which is also provided by Go's delve, there is an GUI observer where users can check the states, messages, stack trace of all Erlang VM processes. Finally, though Elixir is a dynamic language, Erlang has a static analysis tool dialyzer for deducing all the type specification for type checking and unreachable code.

Further Improvements

Due to my lack of time, the following will be finished after the course:

- More comprehensive tests for the Elixir version;
- More decoupled version; like decoupling the proposer and acceptor's log;
- Use databases for keeping states of log entries (utilizes db's functionality to provide better concurrency);
- Add leader election;
- Allow configuration change.

Summary

For this final project, I have implemented a simple key-value system based on multi-paxos in both Go/Elixir. Through this project not only have I learned Paxos, Raft's core idea and the basic implementation trade-offs and possible optimization for multi-paxos, but also I have gained hands-on experience in test-driven development using Go and Elixir.

Acknowledgement

I would like to sincerely thank Professor Sandhya for her help through the final project.

References

[1] "Mit 6.824 lab 3: Fault-tolerant key/value service." `https://pdos.csail.mit.edu/6.824/labs/lab-kvraft.html`.

[2] "Mit 6.824 2015 lab code." `http://nil.csail.mit.edu/6.824/2015/labs/lab-1.html`, 2015.

[3] "Paxos kattis, university of chicago." `https://uchicago.kattis.com/problems/uchicago.paxos`.

[4] L. Lamport, "Paxos made simple," 2001.

[5] J. Ousterhout and D. Ongaro, "Implementing replicated logs with paxos." `https://ongardie.net/static/raft/userstudy/paxos.pdf`.

[6] "Raft consensus algorithm." `https://raft.github.io/`.

[7] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. 2017.