Joseph Murphy
05/17/2018

6.824 Final Project Report

Foreword

I'm going to be super straightforward here and say that I did not accomplish nearly as much as I wanted, and while somewhat unfortunate, I still managed to learn a lot in the process. The Raft implementation is at the very least mostly there, but something between porting the test cases and porting the Raft implementation is not playing nicely. The key-value server is a different story. I didn't get to testing it due to the Raft issues, but what is there is almost certainly correct for reasons I'll explain later.

Introduction/Rationale

As the reader may know, Rust is a concurrency focused language that syntactically prevents data-races, is memory-safe, and has no garbage collection. These three features are rather desirable in the systems programming domain for a number of reasons. The memory safety and data race safety are extremely useful for preventing unplanned behavior in programs such as segfault crashes and race induced logical errors. The lack of a garbage collector is nice because it makes the resulting code more performant and allows for real time systems programming (especially common in embedded systems). In distributed systems, programmers can lean into these features allowing the programmer to focus on a smaller set of potential pitfalls than before. My goal was to port the 6.824 Raft labs into Rust to explore the language myself and also as an proof-of-concept for the future. Furthermore, the Rust ecosystem lacks a clean implementation of Raft, and thus another nice goal would be to publish my implementation on crates.io (Rust's public package server).

Phase 1 - Design, Exploration

The first stage of the process was understanding the problem and which tools I would need to solve it. The biggest challenge from the outset was the testing strategy and the libraries that I would use. The first library requirement that I started exploring was the RPC framework. Initially I thought I would use tarpc, which I had seen in passing before, but when I started attempting to use it, I quickly decided that I liked the somewhat more clunky, but decidedly more understandable, gRPC implementation for Rust (https://github.com/stepancheg/grpc-rust). This library made it very obvious what was happening due to the verbosity of its generated code (as well as its lack of use of macros). The next step was to get a good feel for actually implementing something with gRPC that was representative of what I would need. Therefore I made a HelloWorld of sorts that uses gRPC to ping a server for a greeting phrase to use. It

demonstrated that I could have my semi-mutable (immutable reference with enclosed mutexes) server and also still use the RPC server.
The other large exploration I did was for a framework for network manipulation and server crashing. I came across several projects that looked promising, but were ultimately inadequate. The most common shortfall was the lack of a programmatic API (most of them presented a CLI), however some others didn't take appropriate measures to simulate the "unexpectedness" of failures by using same host iptables rules. This is known to trigger a different failure chain than a true network dropped packet of a typical system. I eventually found a project called "Blockade" (https://github.com/worstcase/blockade). This was the most promising lead thus far, and only required that I send REST requests to a local server. This would require that I write an interfacing library, but it was the most straightforward option that I had yet, and thus I went with that. I wrote an initial version, but it would not be the last version of that library. After that, I started writing the test fixtures and the Raft itself.

Phase 2 - Raft Implementation

I implemented Raft in a relatively similar fashion to my Go implementation because I knew that worked well, and thus was my best bet to quickly get the project up and running. However that didn't stop me from making some design improvements along the way. Firstly, I designed the log data structure to be trimmed from the start. This way I wouldn't need to rewrite a substantial portion of my code when I wanted to add snapshotting functionality. The log's indexing function takes into account an offset that is of course persisted with the log and other persistent state. This was a source of major headache in my previous implementation, and thus I was intent on avoiding it. Secondly, due to Rust's syntax requirements, the locking scheme had to be somewhat reworked, which was unfortunate, but necessary in pursuit of no data races. Concurrently, I started porting the same test fixture API that the config.go used in the Lab 2 materials.

Things were going well until I tried to test anything. Getting anything to compile was actually more straightforward than expected because Rust's compiler gives really excellent feedback on how your code is wrong, and I would say that the compiler and I are on pretty good terms since I don't see pagefulls of red and yellow text as "yelling". Though I could see how that would be interpreted that way. It's kind of like a good English teacher grading an essay, the more red marks, the more you'll learn to fix for next time. The bigger issue was after I got it compiling, however. Debugging code that is running on no fewer than four hosts (virtualized or not) is hard. Coming up with a scheme for logging and getting those logs to somewhere I could see them was one of the most difficult parts of the project. Getting the whole bootstrap process working exactly correctly was essential for me to be able to get anything else done. Thus after a lot of debugging of my blockade package (https://crates.io/crates/blockade), I finally got a runtime environment that was workable for testing Raft. Realistically speaking, I probably ended up spending most of the time that I had on this portion of the project more so than anything else. I'm reasonably happy with the product of this effort as it offers one of the best

environments that I've seen before for testing networked and distributed systems. The typical workflow for testing an arbitrary system would be:

1. Create a test configuration using the types offered in my blockade function
    a. These can be procedurally generated obviously, which is the scheme I use in the Raft tester code
2. Set up a binary target for your project that exposes the API that you want to test on the network.
    a. For me that meant making a Raft Test Server that did nothing but run Raft and send its results over the network to the tester code.
3. Create integration tests that target the API over the network
    a. Thus instead of directly interfacing with my code, my integration tests connect to Raft via RPC
4. Decide on a system for communicating the test setup to both the tester code and the necessary startup configuration to Raft
    a. For me, the tester code had direct access to the port numbers that each Raft would be accessible at, and I used Docker's volumes feature to give the containers access to files that they could use to determine their startup parameters.
5. Decide on a system for logging
    a. The way that worked for me was to give the containers access to a log file that they could write to, but was in fact on the host machine.

This system is effective and almost completely generic to any system you wanted to test. There's no protocol or API preference, there's no modifications to the RPC framework, etc. The caveat is of course that one needs to be able either send REST API requests to Blockade or use my Rust interface to it. Docker is a great choice of platform for testing because it allows programmatic "host" failures, it allows for network manipulation, and importantly it is often the production target of many applications.

I improved my logging infrastructure several times to better diagnose and fix problems. Rust has extremely convenient support for this through the use of the "log" crate and its macros. I used the "fern" crate to actually do the logging (the "log" crate only exposes a facade and relies on other crates to handle the implementation). It allowed module level overrides on the log level, colorized output, and several other desirable features that made my life somewhat easier. This was definitely superior to the Go situation at the cost of a little extra complexity. Very importantly, however, was the integration with my dependencies. Since many of my dependencies also supported logging through the "log" facade, their output was also reported, so I got some crucial hints from logging that appeared through those dependencies on several occasions.

Eventually I got a decent number of tests passing, but due to time constraints and the difficulty of debugging both test cases and the code being tested, I did not get everything working. In the near future I hope to fix that because I set out to program a "fully correct" implementation of

Raft, and I won't be satisfied until that is achieved. Perhaps I can even make it type generic someday (at least among things that are serializable), but correctness and panic safety would be nice in the near term.

The key-value store on the other hand was implemented by and large, but not tested. However, I'm more optimistic about that code's correctness. This is because my Lab 3 was specifically structured in a Rustacean way because I had started work on structuring my final project. That meant object associated locks and heavier use of my event library which provided data safe abstractions for concepts that were easier to reason about. This translated well into Rust for a number of reasons, mainly due to the locking scheme I used (also the enormous amount of time I spent validating that code for 3B meant that it was rewritten to be much easier to follow). Given a working Raft implementation and a bit of testing, this would be workable in a minimal amount of time. It really reflected how my mental model was changed after thinking about it under the constraints imposed by Rust.

Outcomes

As far as code goes, the relevant stuff will be accessible at:
https://github.com/JMurph2015/blockade-rs
https://github.mit.edu/murphyj/rusty_raft
https://github.mit.edu/murphyj/raft_kv

The last two will be public for a while for ease of access, but if necessary, they can be taken back to private (like they've been up to now), since in theory they would essentially be an answer key to all of labs 2 and 3. The first one is relatively unrelated to the labs, so it seems fine to be out in the wild.

The greater outcomes were probably in my understanding of concurrency and of course more exposure to Raft and data stores built on it. I have some ideas for the future that would be interesting to try out to try to squeeze more the performance out of it, such as batching and dynamically tuned election timeouts (to improve response times to failures). Rust's principle of "lock data, not code" was properly drilled into my head, as well as lessons on mutability and lifetimes. Particularly the idioms of around ownership and locking have influenced my design decisions since then, and I think it could be interesting to see about pushing that concept up the abstraction layer a couple levels since it is essentially enforcing strong consistency at a local memory level. One could make a lease system that allowed quick reads so long as other servers hadn't given out a write lease. There is still the matter of replication, which one could leverage Raft for, but that would only have to be done for mutation of the data.

Overall, I think it was a great dream, but perhaps one that was a bit unachievable to do on my own. Had I been able to convince some people to get on board, I think it would've been very reasonable to achieve the full set of goals.