RAFT System Analysis and Testing Framework

David McLaren DMCLAREN@STANFORD.EDU
Elizabeth Walkup EWALKUP@STANFORD.EDU
Patrick Harvey P1HARVEY@STANFORD.EDU
Computer Science Department, Stanford University

Abstract

A system is presented for analyzing the operation of implementations of the RAFT consensus algorithm under both normal operation and in the presence of various failures. Its implementation is described with an API allowing the RAFT checking system to interface with additional RAFT implementations with a minimum of additional implementation-specific code.

1. Introduction

Raft is a consensus algorithm that keeps replicated logs of transactions to a database or key-value store (Ongaro & Ousterhout, 2013). Despite its relatively recent release, it has gained a large following. There are currently over 50 different implementations of Raft listed on their website.

As with any system, each implementation is slightly different, and this introduces possibilities for protocol errors and inconsistent corner case performance. When evaluating (or even just observing) these systems, it would be beneficial to be able to quickly compare how different implementations handle the same error conditions. Since some of the implementations are incomplete, this can highlight the ways in which they behave differently because of that. Such comparisons would also be beneficial for anyone writing a new Raft implementation and wishing to check its behavior against that of some of the more established implementations (like LogCabin).

With this goal in mind, we have built a testing framework that provides common interfaces and runs some of the cases that people might be interested in testing. To do so, this system simulates a cluster on which RAFT is deployed, but logs internal communication between the RAFT nodes while also executing client applications that make requests of the RAFT cluster. This setup allows both external results-centric testing of the RAFT implementation from the client point of view, and recording the internal communications for measuring different message frequencies or other manual inspection.

This document is part of a programming assignment for CS 244B: Distributed Systems, Stanford University (Autumn 2014) and was created on December 12, 2014.

2. System Overview

One of the biggest challenges of our system is that it needs to accommodate the wide variety seen in the different Raft implementations. Although their functionality is very similar, these implementations vary widely in structure, language, and networking choices.

The checker is composed of various modules which are held together by a test driver program written in C++. There are three main pieces of the framework: the RAFT implementation interface, the packet monitor, and the testing framework. For the first part, base classes representing Raft clusters and clients are adapted for different Raft implementations to support reuse of functionality and enable the same tests to be executed across different platforms. In the second part, the monitor watches all network traffic and reports statistics. Then, using the third part, the end user can run a variety of tests on the RAFT cluster.

3. RAFT Interface

3.1. Implementation Setup

3.1.1. Network Setup

Simulating a cluster makes use of generated, virtual network interfaces to allow RAFT cluster nodes to appear to one another as though on independent devices. Much of this setup is independent of the specific RAFT implementation; it is executed via shell script. Since there is no common script for this between implementations, we wrote separate setup and teardown scripts to save time. These set up the network interfaces and configure the necessary Raft parameters.

Since most of our work was done simply on a single

machine, the RAFT nodes communicate between specified ports rather from different IPs (as would be expected in a production RAFT system).

3.1.2. Cluster Setup

Once environment setup is complete, the test driver begins by starting a new Raft cluster with a specified number of nodes, and clients to request operations on the cluster. The driver calls the client interface to request operations on the cluster.

3.2. Client API

Different implementations of RAFT may have a different interface for receiving and responding to client requests, so the implementation of tests that include client requests must include functionality specific to individual RAFT implementations. To avoid large amounts of redundant functionality and allow the checker to be applied to new RAFT implementations more easily, a Client API is provided. The client class provides a common interface to connect to a Raft cluster and read or write files on it. Each client subclass supports these operations for a different Raft implementation.

The API is defined as RaftClient and RaftClusterConfig base classes in C++ with several pure virtual methods, which must be extended to provide a client for a particular RAFT implementation. This client can then be passed to the RAFT checker's tests and manipulated via the API in a manner externally identical to any other client. The RaftClusterConfig implementation implements implementation-specific logic to set up a RAFT cluster, and the RaftClient implements logic to handle a client capable of communicating with that cluster. RAFT implementations that provide a much cleaner interface for clients in some language other than C++ can be supported by creating a C++ client implementation that calls out to execute functionality in scripts or programs of the language in question.

The RaftClusterConfig API provides helper functions to create or kill subprocesses (to take down nodes in the Raft cluster). It requires subclasses to implement the following methods for different Raft implementations:

- `launchCluster(int numNodes, int port)`: Start a new Raft cluster with the requested number of nodes, listening on the specified port.
- `stopCluster(int numNodes)`: Shut down Raft cluster with the provided number of nodes.

The Client API requires that RaftClients implement the following methods:

- `createClient(RaftClusterConfig* config)`: Create and initialize a client.
- `destroyClient()`: Stop a client and free any resources it uses.
- `connectToCluster(string hosts)`: Connect client to a Raft cluster consisting of the specified hosts.
- `writeFile(string path, string contents)`: Write a file with content value on cluster at specified path.
- `readFile(string path)`: Read file contents at path from the cluster and return as a string.

4. RaftMonitor

The RaftMonitor module monitors and responds to messages passed internally among the RAFT cluster's nodes by listening on a common interface. In the case of a local cluster, it listens on the loopback interface (lo). This should be completely transparent to the RAFT cluster, since the monitor is simply a sniffer that logs packets without interfering with their delivery.

4.1. Simulating Partitions

In distributed systems, one of the most common problems is network partitions, where some subset of nodes are cut off from the rest. Since we are operating on a small scale (one machine as opposed to a production-size network), it is necessary to simulate partitions rather than actually causing them through switch or router manipulation.

There are two methods we examined for simulating partitions. The first, and arguably the simplest, is to set up a local iptables firewall. By adding and removing rules, we can drop or allow packets going between specific nodes. The downside of this method is that we cannot see the dropped packets, so that insight into the behavior of the partitioned nodes is lost.

The second method was to make RaftMonitor a sort of proxy firewall. Since all the system packets would go through this module, it could decide which packets to forward and which ones to drop, while still keeping track of what each node is trying to do.

While the concept of the proxy was theoretically attractive in terms of its potential functionality, in the end we selected the iptables-based method. The reason for this is that building a proxy compatible with all the RAFT implementations proved prohibitively complicated, as it would need to support many protocols (TCP, UDP, and HTTP at the very least).

4.2. Packet Interpretation

RaftMonitor keeps track of what commands pass between nodes. It does this by parsing out operation codes

directly from the packets themselves. This allows it to run independently of the implementation language or function names. Unfortunately, it does still require implementation-specific adapters. These adapters are based off how each implementation constructs its packets.

Some parsers require byte-level parsing of operation codes, while others are simply HTTP string interpretation. While these require some research, they are not lengthy (usually only around 50 lines of case-style code each) and seem to perform quite well.

4.3. Behavior Comparison

The RaftMonitor module outputs the behavior information to a log file based on the packets that it sees. This is not simply a packet logging mechanism - it is closer to an event logger. For instance, the monitor does not log every single heartbeat and reply. Instead, it watches for when a node does not respond to a heartbeat (indicating a possible failed node event), or if a write event comes from a different node than previously seen (indicating a possible leader change event). These events are logged both to the console and to a file.

When a test is run, the test driver tells the RaftMonitor when the test starts and ends. This allows the monitor to collect statistics just for the test interval. These specfic numbers are written to a test log at the end of the test. These log files must be manually inspected to see differences between implementation executions.

5. Test Driver

The test driver ties the three sections of the framework into a coherent program. It offers a simple command line interface:

```
sudo ./testdriver/testdriver <impl>
    ↪ <nNodes> <nClients> <testname>
    ↪ <iterations> <time> [test args]
```

The test driver follows a simple algorithm:

1. Spin up a RAFT cluster using the implementation API
2. Start up the RaftMonitor
3. Call the RaftMonitor to signal the beginning of the test. RaftMonitor will set up any partitions necessary.
4. Use the implementation API to make any necessary read or write calls
5. Call the RaftMonitor to signal the end of the test

6. Tests

Obviously there are many, many test cases that could be implemented. We have picked what we feel to be a representative sample of useful ones.

6.1. Log-Based Tests

1. A basic sanity check, where the client submits a set of operations to the master that are expected to succeed.
2. Multiple client submitting write operations at the same time.

6.2. Partition Tests

These are currently written for the assumption that the system only has three nodes, since this is the case with our setup scripts on a single machine. Once a system is set up using our implementation interfaces, the tests can be run without any further modification.

1. A non-master node becoming partitioned from the rest and, after a fairly long interval, rejoining the cluster.
2. The master node being partitioned from the other nodes and rejoining the cluster later.

6.3. Node Failure Tests

1. The current leader node fails by crashing.
2. A non-leader node fails by crashing, then restarts.
3. The current leader node fails by crashing, then restarts.

7. Future Work

Since this is a testing framework, tests are central. Adding more tests is always possible. We could expand into bigger networks, given more time and computing resources, to make it a better tool for people using large implementations. We could also make more complex tests, like mixing partitions with dead nodes.

Since there are so many implementations of RAFT, simply writing adapters is a task on its own. At the moment, the framework has fully written adapter for LogCabin and etcd. Since there are over 50 different implementations, more adapters will always been needed, so adding more in the future will improve our functionality and ease of use.

As mentioned previously, looking into automatic comparison of test results is another useful feature. Because some Raft changes (like who becomes leader) are somewhat non-deterministic, it may be difficult to have a

simple algorithm to pull out the differences. However, such automation would still be of value, since the files would still be there for manual comparison if necessary.

Finally, there is the potential extension of producing the proxy-style network partition simulation, in which the RaftMonitor acts as a proxy and can make detailed pass/-drop decisions in response to packets. This would enable more precise invasive testing, particularly in that this could potentially be used to enforce a packet-accurate time-of-drop when comparing implementations. For example, a test could consist of sending a particular request and then killing a node immediately after it receives the first packet pertaining to that request. If RaftMonitor is acting as a proxy, then this test can be accurately enforced: the packet is passed, then a thread is spun off to kill the node; meanwhile, the monitor blocks all packets going to and from the node until it is killed (to better represent a precise time-of-failure to the rest of the simulated network).

References

Ongaro, Diego and Ousterhout, John. In search of an understandable consensus algorithm (extended version). 2013.