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.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 and controls all 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 invasive testing including reading and analyzing the internal messages sent among the RAFT nodes, node failures, network partitions, and lost messages.

This document is part of a programming assignment for CS 244B: Distributed Systems, Stanford University (Autumn 2014) and was created on December 4, 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++. 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. The test driver proceeds through several stages.

2.1. Implementation Setup

In order to actually test a Raft implementation, a running copy is needed. 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. In practice, people who use this program will most likely have their own Raft implementation already running, so this is offered more as a convenience than as an integral part of the program.

2.2. 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.

Once environment setup is complete, the test driver begins by starting a new Raft cluster with a specified num-

ber of nodes, and clients to request operations on the cluster. The driver calls the client interface to request operations on the cluster.

2.3. 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.

2.4. RaftMonitor

The RaftMonitor module monitors and responds to messages passed internally among the RAFT cluster's nodes by acting as a proxy or middleman to all of the cluster's internal communications. Currently, iptables redirects are used to point all traffic to the monitor's interface. For stability, we are in the process of transitioning into a more proxy-like behavior, where the nodes are set up to communicate directly through RaftMonitor. This should be completely transparent to the RAFT cluster, since the default behavior of the monitor is just to log packets. RaftMonitor rewrites and forwards packets, changing their destinations to the appropriate node based on the address and/or port the monitor received the packet on. Thus when non-invasive testing is being performed, the cluster should operate in an entirely normal manner unaffected by the monitor's presence.

2.5. 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, which is the one we have chosen to use, is to make RaftMonitor a sort of proxy firewall. Since all the system packets go through this module, it can decide which packets to forward and which ones to drop, while still keeping track of what each node is trying to do.

2.6. Issues Considered

In order for RaftMonitor to be transparent, is important that any processing the monitor performs on the packet not lengthen too much the time required for the packet to reach the RAFT node it is intended for, lest the monitor interfere with RAFT's heartbeat messages and thus affect the cluster's operation. However, this does seem a manageable constraint: since the checking program simulates a cluster

within a single machine, round-trip times for packet transmission are low (on the order of 0.05-0.2 milliseconds). Recommended settings for RAFT election timeout are in the low hundreds of milliseconds (Ongaro & Ousterhout, 2013). Simple logging, modifying, and resending of packets, then, is unlikely to cause an unexpected timeout under this sort of mode of operation. However, more expensive actions such as killing one of the RAFT node processes could be concerning given this time constraint; actions that are too expensive are placed in a separate thread. For example, the action to kill a node will consist of creating a separate thread tasked with killing the node's RAFT process, and the monitor will begin dropping all packets to and from that node in the meantime.

In order for RaftMonitor to be transparent, is important that any processing the monitor performs on the packet not lengthen too much the time required for the packet to reach the RAFT node it is intended for, lest the monitor interfere with RAFT's heartbeat messages and thus affect the cluster's operation. However, this does seem a manageable constraint: since the checking program simulates a cluster within a single machine, round-trip times for packet transmission are low (on the order of 0.05-0.2 milliseconds). Recommended settings for RAFT election timeout are in the low hundreds of milliseconds (Ongaro & Ousterhout, 2013). Simple logging, mangling and resending of packets, then, is unlikely to cause an unexpected timeout under this sort of mode of operation. However, more expensive actions such as killing one of the RAFT node processes could be concerning given this time constraint; actions that are too expensive are placed in a separate thread. For example, the action to kill a node will consist of creating a separate thread tasked with killing the node's RAFT process, and the monitor will begin dropping all packets to and from that node in the meantime to present to the remainder of the cluster the impression of the node having been killed at the precise moment decided by the monitor.

2.7. 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. While they require some research, they are not lengthy (usually only around 50 lines of code each) and seem to perform quite well.

2.8. Behavior Comparison

The RaftMonitor module outputs the behavior information 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 the heartbeat comes from a different node than previously seen (indicating a possible leader change event).

These events can be logged to the console or to a file. Right now, they must be manually inspected to see differences between implementation executions, but we are working on automating that to some extent in the future.

3. 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.

3.1. 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.

3.2. 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.

3.3. 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.

4. 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.

At the moment, the framework's adapters are only completely written for the LogCabin implementation. They are partially complete for two other implementations (copycat and CKite). Finishing these adapters would be a priority. 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.

References

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