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.

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. Implementation Interfaces

discussion of different modules in system, and how we support adaptation for different raft implementations

- test driver (glue that holds everything together) - setup scripting - the monitor/sniffer module - the Client API – API functions to read/write using a specific Raft implementation – system-wide API also getting tucked in here – (The Client API will probably be renamed to something else to better reflect its meaning– something like an “Adaptation Layer”) - teardown, if it’s worth it

2.3. RaftMonitor

The checker 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 inter-

nal communications. The IP address and port presented to the rest of the cluster for a RAFT node is instead one assigned to the monitor. However, on setup this should be completely transparent to the RAFT cluster, since the default behavior of the monitor is just to log packets, rewrite their destination to the appropriate node based on the address and/or port the monitor received the packet on, and resend the packet. Thus when non-invasive testing is being performed, the cluster should operate in an entirely normal manner unaffected by the monitor's presence.

2.3.1. 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, 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.

2.3.2. 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, 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.

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 master node fails by crashing.
2. A non-master node fails by crashing, then restarts.

4. Future Work

- plans for additional checkers for more raft implementations - more functionality in monitors? delaying or blocking packets

additional tests we plan to do or didn't get to

References

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