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

ABSTRACT

1. Introduction

Raft is consensus algorithm that keeps replicated logs of transactions to a database or key-value store. 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.

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.

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.

This document is part of a programming assignment for CS 244B: Distributed Systems, Stanford University (Autumn 2014) and was created on December 4, 2014.

2.2. Implementation Interfaces

The checker is composed of various modules which are held together by a single 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.

2.3. Cluster and Client Setup

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 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 test driver calls the client interface to request these operations.

- the monitor/sniffer module (next subsection, but TODO: not clear yet where this will live?)

2.4. 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 internal 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.

For this to be the case it 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

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. concepts for simulating partitions, sniffing packets - iptables and packet redirection - sniffing with libtins

discussion of tests supported by the system: - basic sanity check (client does op expected to succeed) - multiple clients writing to file system - failure and restart of raft nodes - partitions

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.