
TITLE

David McLaren
Elizabeth Walkup
Patrick Harvey

Computer Science Department, Stanford University

DMCLAREN@STANFORD.EDU
EWALKUP@STANFORD.EDU
PHARVEY@STANFORD.EDU

Abstract

ABSTRACT

1. Introduction

basic description of system (maybe 1/2 page)

2. Simulating Partitions

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

3. System Overview

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 im-
plementation - 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 “Adap-
tation Layer”) - teardown, if it’s worth it

3.1. Environment 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 implemen-
tation; it is executed via shell script.

3.2. Client API

Different implementations of RAFT may have a dif-
ferent interface for receiving and responding to client re-
quests, so the implementation of tests that include client

requests must include functionality specific to individual
RAFT implementations. To avoid large amounts of re-
dundant functionality and allow the checker to be applied
to new RAFT implementations more easily, a Client API
is provided. The API is defined as base `RaftClient` and
`RaftClusterConfig` classes in C++ with several pure virtual
methods, which should 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 should implement
any necessary implementation-specific logic to set up a
RAFT cluster, and the `RaftClient` should implement logic
to handle a client capable of communicating with that clus-
ter. 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 Client API requires that `RaftClients` implement
the following methods:

- `createClient(RaftClusterConfig*)`:
Create and initialize a client

3.3. RaftMonitor

The `RaftMonitor` monitors and responds to messages
passed internally among the RAFT cluster’s nodes by act-
ing 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 as-
signed to the monitor. However, on setup this should be
completely transparent to the RAFT cluster, since the de-
fault behavior of the monitor is just to log packets, rewrite
their destination to the appropriate node based on the ad-
dress 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 pro-

cessing 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 (Ontaro & 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 would 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.

3.4. Future Work

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

4. Tests

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

4.1. External Tests

External tests examine the results of requests made of the RAFT cluster; fundamentally for correctness they are limited to examining a cluster only as a normal client of it could, although the monitor's message logs and statistics can still be considered.

4.2. Invasive Tests

Invasive tests rely on the RaftMonitor's position as a middleman in the internal communications between the RAFT nodes. The monitor allows the dropping of particular packets or closing of connections, as well as the killing or restarting of nodes at more specific points in execution than can be done externally, such as causing a node to die after having received a request but before it replies.

4.3. Future Work

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.