Raft Monkey: Differential Testing for Raft Implementations

Rowan Chakoumakos (rowanc@cs)
Stephen Trusheim (tru@cs)

December 12, 2014

1 Abstract

We performed differential testing on three implementations of the Raft consensus protocol and found that two popular implementations have implementation issues that can result in inconsistent behavior. We are continuing to investigate over 30 identified differential test results to find whether they indicate deeper protocol issues.

To perform differential testing without great expense, we implemented a full environment manager ("River Raft") and a full test suite ("Raft Monkey"), all of which can run on one machine. River Raft uses Docker containers to easily create isolated black-box Raft servers without the expense of multiple machines; further, it allows the programmatic creation of network partitions between these instances. Raft Monkey generates and feeds clusters the sets of client commands, individual server failures, and arbitrary network partitions to allow differential analysis; a "diabolical" mode runs the worst possible legal set of clients to force potential issues. Our work builds on tools like Jepsen to allow plug-and-play, low-cost network partition and node failure testing for implementations of the Raft protocol.

2 Introduction

Like all protocols, the excellent Raft consensus protocol is limited in practice by the strength of its implementations. In theory, all implementations of Raft should result in the same externally-observable state after an identical sequence of events (such as external I/O requests, individual node failures, and network partitions). In practice, implementation errors or undefined behavior in edge cases can result in different behavior under the same conditions. This study was designed to find such anomalous behavior by automatically testing Raft implementations against one another and finding inconsistencies in externally observable state.

To our knowledge, no existing tools provide either scalable automated analysis or differential analysis of implementations of any consensus protocol. Jepsen is a standard tool used to model network partitions and node failures for distributed software, and has been used to analyze the correctness of numerous software packages versus theoretically guaranteed properties (e.g., linearizability). However, the tool does not allow easy "plug and play" of different implementations of the same protocol, nor does it allow parallelized testing on a single machine. Moreover, it is not designed for comparative test-

ing; it can only test against an extrinsic "gold standard," which may be hard to generate (Engler et al., 2001)

We built our entire test infrastructure from scratch because no existing tools provided a sufficient foundation. We built River Raft (section 3) to provide virtualization of an entire Raft cluster on a single machine, using Docker as a foundation and adding the ability to programmatically create network partitions. We built Raft Monkey (section 4) to generate Raft test cases, run them on a cluster, and perform differential analysis on the externally observable state of the cluster. Using our test infrastructure, we identified two implementation errors in common Raft libraries (section 5). We have published River Raft on Docker Hub.

3 Selection of Raft implementations

At least 57 implementations of Raft exist today, but in our experience, the quality of each implementation varies dramatically. Many do not actually implement all of Raft; some even warn against their own use. Others seem to work, but crash immediately after startup.[1] Since we were spending so much time just trying to get various implementations to run, we focused on implementations that had an example key-value store built on top of the Raft algorithm. We hoped that these projects would be at least runnable. Out of the 57 implementations we could find, we narrowed down to 6 implementations with key-value stores, and out of these, we were able to get 4 to run. We selected the most active 3 to focus our efforts, described briefly here:

1. `etcd` is the de-facto standard implementation of Raft, written in Go and supported by an active development team.

2. `raftd` is the reference implementation of the Goraft project. We use its example KVStore implementation to provide a key-value store.

3. `CKite` is a Scala implementation of Raft that provides a key-value store.

4 River Raft

The goal of River Raft is to abstract starting, killing, and partitioning Raft nodes, as well as their running details, to allow "plug and play," black-box use of multiple implementations with an abstracted, standard architecture. A secondary goal of River Raft is to allow performant virtualization of a full cluster on one machine, instead of requiring expensive and difficult-to-setup multi-machine clusters.

River Raft includes multiple components. At its core, it is a layer around the open-source tool Docker, which in turn is a developer-friendly wrapper around LXC. With River Raft, a new implementation can be created simply by copying the base River Raft container, adding the installation instructions for their tool, and specifying the commands necessary for the River Raft interface listed in Table 1.

[1] Our personal favorite is 'copycat'. From the documentation, it looked like it had implemented most of Raft and had a nice API. After spending hours getting this project to compile, however, we `grep`'d for a key class and found the only reference to its name was in the documentation.

Command	Description
`BOOTSTRAP-LEADER`	Launches the leader in bootstrap mode
`BOOTSTRAP-FOLLOWER`	Launches a follower and passes the ip and port of the leader
`KILL`	kills raft daemon (usually kill -9)
`START`	restarts a raft daemon after being killed (non-bootstrapping)

Table 1: River Raft interface for managing RAFT docker containers.

4.1 Implementing node kills

We experimented with multiple ways of killing a River Raft node. We first attempted to simply kill the Docker container, but quickly realized that Docker data was not persisted across runs. We then used Docker's volume feature to provide persistent storage for each node.[2] Even then, however, every Docker restart forces the assignment of a new IP address; this breaks Raft implementations that expect the IPs to remain static.[3]

To solve these issues, we expose a long-lived server (the River Raft node server) that takes commands and manages the state with the docker instance. In our final implementation, we send `start` and `kill` commands to the River Raft node server inside the Docker container, and do not kill the container until cleanup is necessary.

[2]This feature could also be used to implement killing a node as soon as a write comes through.

[3]In fact, avoiding this behavior is one of the most heavily requested features for Docker.

4.2 Implementing network partitions

The objective of River Raft's partitions feature is to be able to create any arbitrary topology, including one way partitions (i.e. A can send messages to B, but B cannot send messages to A) and overlapping partitions (i.e. B can send messages to both A and C, but A and C cannot send messages to each other). While complex network partitions of this sort are not common, they are invaluable for testing distributed systems. We were surprised that no simple tool existed for this purpose.

We built rapid prototypes involving a man-in-the-middle proxy, an `etables` virtualized network interface, and the open-source tool Blockade. These tools were all either too complex or did not fit our needs: the man-in-the-middle proxy broke black box simplicity, the `etables` implementation was far too involved, and Blockade did not provide the advanced partitioning we desired.

Inspired by Blockade, we designed a simple but powerful partition manager for Docker. The interface is shown in Table 2. This API enables our arbitrary partitions, including blocking all traffic to a node or splitting a network into a two groups overlapping via a single node. A partition manager exists outside the Docker instances, as part of the River Raft management package. A custom launch script executes upon the launch of a River Raft container and passes the node IP address to the partition manager. The partition manager can then adjust the iptables rules of the host machine to block or allow any specified traffic. The docker containers use a `tap` interface and are each assigned a unique IP address, and the host OS continues to route all traffic. Thus, we are able to install any partition we desire.

Partition API	Description
`PARTITION (a, b)`	severs the link between node a and node b
`HEAL (a, b)`	heals the link between node a and node b
`GET PARTITIONS`	returns all partitions as a list of tuples
`GET ACTIVE LINKS`	returns all active links as a list of tuples (non-partitioned links)

Table 2: River Raft interface for partitioning raft containers.

Test Event	Description
`GET (key)`	writes the value, or failure, to the results log
`SET (k, v)`	sets the value of key k to value v and logs the change.
`KILL (node)`	fail-stops the node.
`REV (node)`	revives the killed node.
`PART (n, p)`	partitions node n from peers p.
`HEAL (n, p)`	heals the partition between nodes n and p.

Table 3: Test events used by Raft Monkey

5 Raft Monkey

Named as an homage to Netflix's Chaos Monkey, Raft Monkey creates sequences of test events, runs them on multiple implementations, and compares the results to find anomalous behavior. Raft Monkey is implemented in Python 2.7.

Raft Monkey has four major components, each implemented as separate Python scripts, corresponding to the four steps of our study.

5.1 Test plan generation

Raft Monkey generates valid testable sequences of events. Each event has a user-specified probability of occurring, and the next valid event is selected from the weighted random distribution. If the event cannot occur (e.g. there are no nodes to revive), a different event is selected. Possible events are specified in Table 3.

In this study, we generated 300 test sequences, each with 100 events. All of the sequences randomly used get/set on one of ten values to affect the externally observable state. In addition:

- 100 sequences included kill/revive commands (no partition/heal commands)
- 100 included partition/heal commands
- 100 included all four possible changes to the cluster.

5.2 Test running

Raft Monkey accepts tuples of (`test plan`, `client library`) and manages the process of starting a n-node cluster (here, $n = 5$), waiting for the cluster to stabilize, then executing the test plan. The externally observable events in the log (i.e. the results of `GET` and `SET`) are saved in a machine-readable log. Each test is repeated at least three times. Tests run in parallel, but the same test case does not run in parallel.

In "standard" (non-"diabolical") mode, the cluster tracks the most recently successful leader node, and attempts to send all requests to that node. If the leader cannot be reached or responds with a failure message, the rest of the cluster is tried in ascending order (node 0 -¿ 4)

until the whole cluster cannot be reached, then the request is logged as a failure.

The details of killing and reviving individual nodes, as well as causing and healing network partitions, are handled by River Raft. Raft Monkey simply assumes that `Kill` results in a fail-stop that does not remove node's persistent log (e.g. `kill -9`), and that `Partition` takes effect immediately.

5.3 Diabolical mode

This mode intentionally submits requests to the Raft cluster in the legal way most likely to cause an edge case. Specifically, enabling this mode intentionally avoids sending requests to the leader node (as a good Raft implementation should fail or forward the request), and intentionally submits as many requests as possible to recently-revived nodes (even as they may be starting up).

5.4 Differential analysis

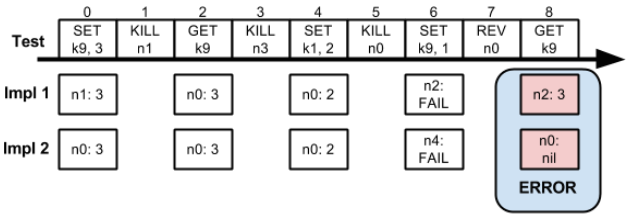

Figure 1: An example test plan and results from two different Raft implementations.

The results of logs from the same test plan are compared in three ways.

- The repeated trials from one implementation are compared to ensure internal consistency.
- The logs from one implementation are compared against the logs from the same implementation under "diabolical mode."
- The logs from one implementation are compared against the equivalent logs from different implementations.

This three-step comparison was designed to find "easy" issues before progressively more powerful tests take place. At each step, we excluded results that had suffered issues found at an earlier step. Figure 1 shows a graphical example of a test plan and two sets of logs, which were found to be inconsistent in one location.

The system flags results that show an externally inconsistent view of the data. Logs that show inconsistent `GET` results, or an inconsistent ability to `SET`, are flagged for human analysis. Human analysis focuses on a second set of logs with richer data available (e.g., debug data from the cluster and client libraries).

5.5 Challenge: Internally inconsistent trials

One major difficulty we encountered was in comparing the results of internally-inconsistent trials (that is, test cases where repetitions of the same plan on the same implementation resulted in different externally observable states.

While our initial reaction was that we had hit a jackpot of errors, continued analysis showed that all three tested implementations of Raft could report a failure state to the client (e.g., by closing the connection), but actually successfully complete the request. Though this behavior violates our expectations, it is not an implementation error; successfully completing an operation without replying to the client is explicitly allowed by the Raft protocol. Our software

would log the error in all cases, and the analysis step would later find an inconsistent state between trials when one trial actually succeeded and another failed.

Unfortunately, we found that 35-47% of our test cases resulted in internally inconsistent behavior for each implementation. Informal analysis has shown all of these errors to be of the form described above; but unfortunately, we were forced to exclude internally inconsistent results from further analysis in steps 2 and 3. We expect that creating a better model of internal state consistency would allow a richer set of differential results.

6 Identified Raft implementation errors

We identified two likely errors in two different Raft implementations as a result of our testing.

6.1 Invalid return errors during startup in `raftd` and `CKite`

When run in "diabolical mode," both `raftd` and `CKite` exhibit unallowed behavior during the recovery of a killed node. Our client submits a GET request to the recovering node as it is completing its startup phase, and instead of forwarding the request to the leader node or rejecting the request as "not leader," the node responds with a null value (indicating that key has not been set.)

This issue is observed in at least five tests for `raftd` and three tests for `CKite`, but is unfortunately difficult to replicate on-demand because it relies on exact timing. We have not observed this behavior in `etcd`.

6.2 `raftd` provides inconsistent reads

We encountered and investigated numerous discrepancies related to the availability of raftd and CKite. Many differential results showed that `raftd` would reply to `GET` requests when `CKite` would not, but did not show similar results for `SET` requests. In addition, consistency checks in `raftd` trials showed inconsistent values.

The root cause of these errors is that `raftd` does not provide strongly consistent reads; if the cluster has failed (e.g. a network partition) but the leader lease remains, the leader replies to `GET` commands without first checking with the cluster; in practice, it routinely replies with inconsistent values. This issue has previously been shown in `etcd`; we note the same issue in a different library, automatically flagged by our tool.

7 Conclusion

Raft Monkey and River Raft represent a step forward in automated testing of the Raft consensus protocol. Our differential testing model and easy-to-use Docker containers allows us to comprehensively test multiple implementations of the Raft protocol with simple "plugins."

Future work in this direction should focus on creating a better model of state consistency between diverse Raft implementations to better identify errors caused by timing in each Raft implementation. We have released River Raft as an open-source project on Docker Hub.

8 Citations and Thanks

All cited papers and code are hyperlinked within the paper. Thanks to Prof. Dawson Engler for his guidance and advice on this project.