Flotsam: Evaluating Implementations of the Raft Consensus Algorithm

CS244B Project Report

Connor Gilbert

***Abstract*—The Raft consensus algorithm [1] is designed to be easily understood and implemented. Many open-source projects now implement Raft as a library or standalone service or incorporate it into more complicated distributed systems. However, the quality of these implementations has not been studied extensively. This work presents a system, *Flotsam*, for empirically testing Raft open-source implementations for errors. Randomized test cases are generated and tested in a virtualized environment based on Docker. Test operations are specified using a general interface to allow new implementations to be simply "plugged in", and outputs are checked against each other using configurable criteria after injecting. This work identifies several implementations of Raft and tests them against each other using Flotsam. The promise of this approach, both for assessing Raft implementations and for testing distributed systems in general, is demonstrated through initial test results.**

I. Introduction

The Raft consensus algorithm [1] is designed to be easily understood and implemented. Its understandability has led to a large number of implementations—56 separate open-source implementations are listed in one collection alone [2]. These implementations are of various levels of quality, as quickly becomes clear through cursory inspection of repository "README" files and issue trackers, some of which include prominent warnings about problems with their correctness or reliability.

The proliferation of Raft implementations, and their relative uniformity (especially compared with other distributed consensus algorithms), gives a unique opportunity to test them against each other rather than against a separately-generated specification. This approach has some clear benefits: it does not require—though it can include—labor-intensive manual test case generation, and it does not rely on a human to accurately specify the correct outputs given a specification like an academic paper or an RFC. While generating test cases without *a priori* expected outputs does have some disadvantages, including the possibility that all of the tested implementations might behave incorrectly in the same way or that the same observed behavior might hide lower-level errors, the advantages make it at least worth exploring.

This work presents *Flotsam*, a prototype testing framework for Raft implementations that uses this approach. It pits many Rafts against each other and attempts to identify any wreckage that would indicate inconsistencies between them. The remainder of this paper briefly discusses related work (section II), outlines Flotsam's design (section III), and evaluates its performance (section IV). Finally, it identifies further work (section V) and concludes (section VI).

The author is with the Computer Science Department at Stanford University, Stanford, CA 94305.

Email: connorg@cs.stanford.edu

II. Related Work

Naturally, almost as long as there have been distributed systems, there have been efforts to check that they behave as they should. However, existing strategies tend to rely on manual test case generation and concentrate on testing the correctness of a single system at a time.

Jepsen [3] has been used to discover errors in a variety of distributed systems, including the Raft-based systems etcd and Consul [4]. Jepsen has produced a number of important results and inspired many other efforts, including this one. However, it concentrates on testing a single system at a time and checks for specific desired traits (*e.g.,* linearizability or durability) rather than pitting implementations against each other.

Blockade [5] creates virtualized test environments for distributed systems using Docker [6]. It simu-

lates network problems, providing configurable latency, packet loss, and partitions. Blockade does not provide any testing facilities, and is instead meant to be run interactively or scripted into a larger test framework.[1]

More generally, there are a variety of approaches for checking or proving correctness of systems—whether distributed or not—ranging from "bug-checking"-style empirical tests to formal proofs. It is out of this work's scope to provide a summary or assessment of the full breadth of these techniques.

III. System Design

Flotsam's testing system pits different Raft implementations (called *candidates*) against each other and checks that the observable output matches across the candidates.

A Flotsam test is specified by a JSON file that details the candidates to test, the test generators to use, the output checking mechanisms to employ, and other necessary parameters, including the number of nodes that will participate in the protocol.

Flotsam can currently only test Raft implementations that provide a key-value store implementation. It would be possible to extend Flotsam to other abstractions, or to implement key-value stores on top of other libraries, if desired. For this prototype, the simple *get/set* interface provided by a key-value store allows a simpler testing framework.

Flotsam interacts with each implementation through a consistent interface which must be written to "wrap" each one. This takes the form of two functions: `read(key)` and `write(key,val)`; each candidate simply specifies the proper actions required to get or set a value. Generally, this takes the form of using a library function to make an HTTP request, but implementations may specify arbitrary actions: the *lite-raft* POSIX shell implementation, for instance, uses `docker exec` calls to execute shell scripts. If, for instance, the leader might change (as in Raft), the candidate must specify how to recover. (This may be as simple as implementing a round-robin loop, depending on the behavior of the implementation and how it communicates leader changes and errors.)

The parties to the protocol are run in virtualized containers that are specified by the user and then managed by the system. The container is specified using a Dockerfile in the Docker system [6], which details the dependencies, environmental configuration, and preparation steps necessary to run the candidate program. Since implementations may need to configure themselves, for example to provide each other with runtime information like IP addresses or ports, Flotsam's setup interface consists of three functions: `create`, which allocates the containers with necessary static files and programs pre-loaded; `launch`, which launches the Docker container (but not the candidate program) so that configurations can be generated; and `start`, which actually starts the candidate program.

A test generator produces a test plan that is run against each candidate implementation. A test plan consists of get/set operations interspersed with failures. This version of Flotsam supports host failures and network partitions. Host failures are implemented by killing the entire Docker container. Failed hosts do not recover in this version of Flotsam. Network partitions are implemented with host-based `iptables` rules that drop traffic between the partitioned hosts and non-partitioned hosts. We initially sought to allow traffic from the test client to the partitioned containers while effectively partitioning the targeted hosts from the rest of the Raft parties, but we disabled this access for this prototype because of the difficulty of distinguishing partitioned hosts from properly functioning ones in the client.

Flotsam simulates a client that issues the planned requests to the virtualized containers. During the test execution, the system performs writes to and reads from the parties. After all of the candidates have completed the test plan, the verification system checks for inconsistencies. The verification system can be configured to check various conditions; in this version of Flotsam, we simply check that the result of each read request is the same across each implementation.

IV. Evaluation

A. Implementations to Test

We present the results of two sets of tests.

Toy Project: We tested the relatively popular *goraft* implementation [7] using its *raftd* key-value store reference implementation [8], against a "toy"

[1] We became aware of Blockade incidentally through Internet searches after this work was substantially complete.

version of *raftd* that we modified to remove persistence by disabling the write path [9]. The goal of this test is to demonstrate that Flotsam can detect this obvious class of errors.

Real Implementations: We selected three implementations of the Raft algorithm to demonstrate "real-world" applicability: *raftd* [8], the *libraft* project's example *KayVee* store [10], and *lite-raft* [11]. We selected these from the subset of implementations on the Raft algorithm website that implemented a key-value store [2]. The goal of this test is to demonstrate Flotsam's ability to test real implementations even across a variety of different architectures—*raftd* is written in Go, *KayVee* in Java, and *lite-raft* in POSIX shell script.

B. Experimental Setup and Procedure

Our experimental setup consists of a Raft cluster of five nodes. (The number of nodes can be reconfigured per experiment.)

The cluster is configured and, if needed, is given time to initialize (*e.g.,* to elect an initial leader).

Then, we begin to issue requests according to a testing plan. The testing plan is the same across all implementations and is specified as a sequence of *get/set* operations and network or host state changes. State changes could include loss of network connectivity with a node or a fail-stop host failure.

C. Our Testing Plans

The default testing plan issues requests over a finite key-space and distributes the volume of requests per key according to a uniform distribution. For simplicity, reads immediately follow writes; however, reads could be allowed even before writes, since empty responses can be compared just like non-empty ones. Other testing plans can be designed to stress different aspects of a Raft implementation; for instance, its ability to handle many keys and its ability to handle multiple updates to the same key might be better exercised with a power-law distribution across the key-space.

D. Results

Flotsam accurately detects the different behavior of the two candidate implementations in the "Toy Project" case. The communicated result of a write is the same across the implementations, so problems are only visible when the test plan includes a read for a key for which a write has previously "succeeded". These errors are, as expected, reported by Flotsam.

The efficacy of host failure and partition failures has been verified through inspection of logs on candidate containers and by the test client's logged detection of nonfunctional leaders.

The behavior of the candidates in the "Real Implementations" test has not yet diverged across multiple tests of over 1000 requests each. More complex testing and failure injection scenarios might cause the candidates' behavior to diverge.

V. Further Work

This work is limited in scope due to the limited time available for its implementation. However, the prototype system we present could be extended in a number of ways. It is specifically designed for extensibility: test generation, setup and interaction with candidate implementations, and output checking are implemented as separate modules that can be selected arbitrarily using a test specification file.

New candidate Raft implementations can be tested in the system by simply defining the runtime environment using a Docker image and implementing the API to access it. New test types (*e.g.,* new types of network problems) could similarly be applied to all implementations by improving the core tester code; no modifications to the implementation-specific code would be necessary.

A more exhaustive error checker might simulate multiple clients to increase the likelihood of finding synchronization bugs and to increase load on the system.

This prototype implements only a simple notion of output checking; many classes of errors—especially those that affect consistency only for certain time intervals or from certain parties to the protocol—may not be detected. More sophisticated strategies would probably yield more comprehensive results.

We restricted our work to the space of Raft implementations that also provided a key-value store out-of-the-box. This restriction was one way to avoid encoding any specific knowledge of Raft parameters into Flotsam, but other Further work could test with different state-machine abstractions, with the goal of being easily applicable to more

Raft implementations or of exposing more complex issues.

Even more broadly, this type of framework—or even Flotsam itself—could be applied to algorithms other than Raft.

VI. Conclusions

Flotsam demonstrates the feasibility and utility of a "blind", output-based testing method that pits multiple implementations of the Raft protocol against each other to check correctness without an *a priori* specification of expected behavior. In addition, it provides an example of the utility of containerized, portable testing environments for distributed systems—all that is needed to run Flotsam is an installation of Docker and access to the Internet to download container images. Such testing techniques may be a useful addition to a more conventional test suite, and, if applied on a wider scale, could both help ensure the quality of Raft implementations and provide additional data to judge the central claim of the Raft paper [1] that Raft is actually easier to understand and implement than older algorithms like Paxos.

VII. Acknowledgements

Thanks are due to the CS244B staff for including this idea on the list of suggested projects and to Kyle Kingsbury (Aphyr) for inspiring this line of inquiry with his excellent work on Jepsen [3].

This work would not have been possible without the large number of programmers who have created their own Raft implementations and published them online, or, ultimately, without the excellent work by the authors of Raft [1].

This project incorporates or uses a number of open-source projects, including Docker [6], dnsmasq, and netifaces.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[2] D. Ongaro. Raft consensus algorithm. [Online]. Available: https://raftconsensus.github.io/

[3] Jepsen. [Online]. Available: https://github.com/aphyr/jepsen

[4] K. Kingsbury. (2014, June) Call me maybe: etcd and consul. [Online]. Available: http://aphyr.com/posts/316-call-me-maybe-etcd-and-consul

[5] Blockade. [Online]. Available: https://github.com/dcm-oss/blockade

[6] Docker. [Online]. Available: http://www.docker.com

[7] goraft. [Online]. Available: https://github.com/goraft/raft

[8] goraft/raftd. [Online]. Available: https://github.com/goraft/raftd

[9] raftd-forgetful. [Online]. Available: https://github.com/connorg/raftd-forgetful

[10] A. George. Kayvee. [Online]. Available: https://github.com/allengeorge/libraft/tree/master/libraft-samples/kayvee

[11] L. Tarenga. lite-raft: raft consensus algorithm written in posix shell script. [Online]. Available: https://code.google.com/p/lite-raft/