CS 244 Project Proposal: Raft

Donovan Jasper
djasper@stanford.edu
Stanford University

Yasmine Mitchell
yasminem@stanford.edu
Stanford University

Kelechi Uhegbu
kuhegbu@stanford.edu
Stanford University

Figure 1. Image of DAKY the rainbow wombat, the mascot of our project team

Abstract

We aim to replicate the implementation of Raft. Specifically, we aim to replicate Figure 16 of the Raft paper which measures the time to detect and replace a leader that has crashed.

Keywords: Distributed Systems, Domain Name System, Networking

1 Introduction

Citation of the Paper

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

URL: https://raft.github.io/raft.pdf

Blog Post Presence

The paper was the subject of a post on the reproducingnetworkingresearch blog in 2015 (https://reproducingnetworkresearch.wordpress.com/2015/05/31/cs244-15-raft-understandable-distributed-consensus/). We have requested an exception and are waiting for a response.

Intellectual Contribution

The paper introduces Raft, a consensus algorithm distinct from Paxos by being more understandable and easier to implement in practical systems. Raft has an easier learning curve for students and developers, simplifying the consensus process by separating leader election, log replication, and safety. This simpler and more straightforward approach, the authors argue, leads to higher quality and more reliable practical system implementations.

Status Quo

Before Raft, Paxos was the primary algorithm used for teaching and performing consensus in distributed systems. However, Paxos is notoriously complex and difficult to understand. Systems implementing Paxos often had to adapt or simplify the algorithm, which in some cases leads to mistakes or inefficiencies.

2 Replication

Result for Replication

The result we intend to replicate is depicted in Figure 16 of the Raft paper. This figure illustrates the time to detect and replace a crashed leader under various randomized election timeout configurations.

Figure for Replication

Figure 16 2 is a combination of two graphs. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials and corresponds to a particular range of election timeouts. For example, "150–155ms" indicates election timeouts chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of five servers with a broadcast time of roughly 15ms, and similar results are reported for a cluster of nine servers.

Explanation of the Result

Figure 16 demonstrates Raft's ability to handle leader failures efficiently. The graphs illustrate that with a well-chosen

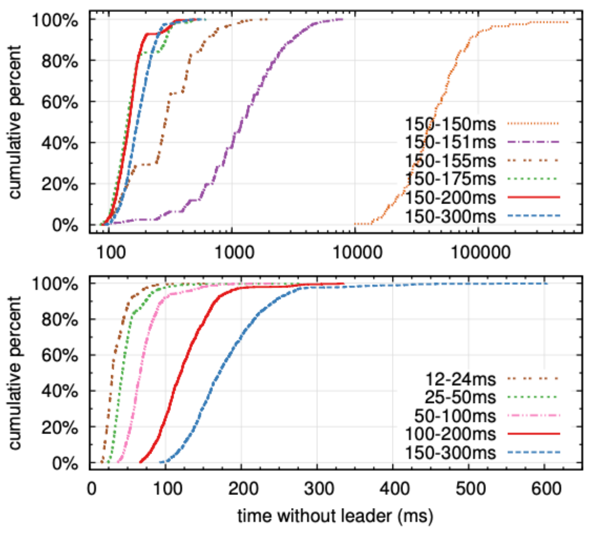

Figure 16: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150–150ms") and corresponds to a particular choice of election timeouts; for example, "150–155ms" means that election timeouts were chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of five servers with a broadcast time of roughly 15ms. Results for a cluster of nine servers are similar.

Figure 2. Figure 16 from the paper - time to detect and replace a crashed leader.

range of election timeouts, the system is capable of recovering and electing a new leader within a few hundred milliseconds. This rapid recovery is crucial for maintaining the availability and consistency of the cluster, showcasing the robustness of Raft's consensus mechanism in operational scenarios.

Replication Approach

To replicate this result, we will set up an environment reflecting the conditions described in the paper: a cluster of 5 servers running the Raft consensus algorithm. We will code our own implementation of Raft in Go. Each server will be an EC2 instance on AWS. The paper doesn't provide a lot of details about the environment this experiment was run in (i.e. server hardware, network speeds, etc.). We will reach out to the authors to ask for more information.

We will induce intentional leader failures while systematically varying the election timeout parameters as presented in the paper. Our objective will be to observe and record the time intervals for new leader election post-failure in order to confirm the performance claims of the Raft algorithm with respect to resilience in leader election timing.

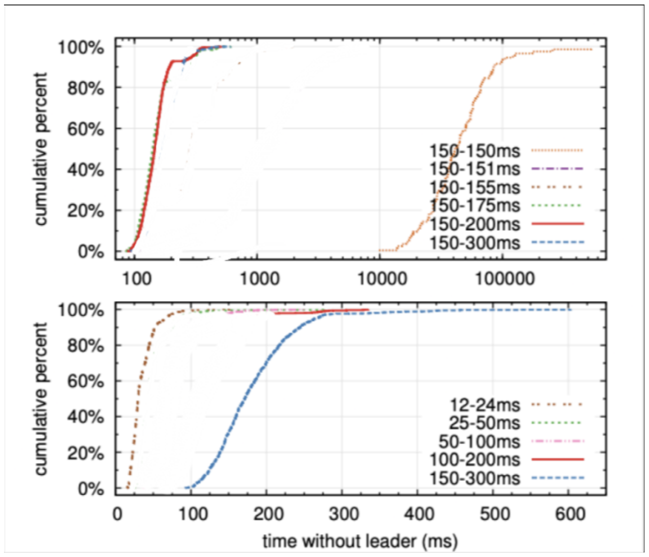

Figure 3. Intended Replicated Figure 16.

2.1 Roadblocks and Progress

Potential Risks to Project Completion

Several risks could potentially get in the way successful replication:

- **Implementation Discrepancies:** Even though Raft is designed to be simpler than Paxos, there are still a lot of complexities. Small mistakes in algorithm implementation may have large effects and be difficult to find.
- **Environmental Variability:** The times in the figure are heavily influenced by the network speed and availability.

Mitigation Strategies

To mitigate these risks we will do the following:

- **Thorough Planning, Review, and Testing:** We will make our implementation as modular as possible and peer review each module. We will also write thorough tests to check for expected behavior.
- **Environmental Controls:** When we hear back from the authors we will do our best to replicate their environment. If that is not possible organically, we will use a network simulation tool.

Midterm Goals

- **Working Raft Implementation:** Have a working Raft implementation in Go.
- **Test Suite:** Have a working test suite for said Raft implementation to ensure it works as intended.

After we complete these goals, we would then implement the leader test for replicating Figure 16.