CS2510 Final Project Proposal

1st Alejandro Ciuba
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
alejandrociuba@pitt.edu

1st Winston Osei-Bonsu
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
wio6@pitt.edu

***Abstract*—Consensus algorithms are a central tenant in creating reliable, stable distributed systems. While many consensus algorithms have been proposed and implemented, the Raft algorithm stands out due to its main design principle: intuitiveness. Thanks to this trait, dozens of officially documented Raft implementations across a wide variety of programming languages have been developed. However, very little work has been done comparing these implementations against each other. Our project seeks to compare implementations of Raft in a series of standardized experiments. Our ultimate goal will be to create a table which compares the tested implementations across metrics like CPU utilization and messages passed. Lastly, given the fact Raft was originally designed with simplicity in mind, we wish to have a short quantitative comparison on the ease-of-use for the varying implementations based on GitHub issue resolutions and developer activity.**

I. Introduction

To create distributed environments whose node information is consistent and agreed upon throughout is the research behind consensus algorithms. While there have been many specific algoitihms for consensus, such as Paxos [1], recent years have shown the rise of a new leader-based consensus algorithm, Raft [2]. What makes Raft interesting is its design philosophy, which focused on algorithm intuitiveness. It is an algorithm specifically designed for students to understand, which has no doubt contributed to its widespread adoption. Due to Raft's popularity for consensus in distributed computing, the original researchers behind Raft have made a website[1] to host the plethora of activities, lectures and, most importantly for our study, implementations. There are over 100 documented implementations of Raft there, spanning programming languages of all levels. The table also highlights important guarantees each system can provide: leader election and log replication, persistence, membership changes and log compaction.

A. Problem Statement & Research Questions

Given the importance of Raft and consensus algorithms in general for consistent, reliable, fault tolerant distributed systems, we believe it is important to understand the nuances between different implementations of Raft. We know the theory and implementation of algorithms can differ greatly, and that implementations can vary in speed, effectiveness and—as mentioned on the website—guarantees. **Our project seeks to compare different implementations of Raft on a benchmark test and compare their performance and examine their ease-of-use through a quantitative analysis.** In our project, we aim to address the following research questions:

RQ1: Which implementation will require the least amount of memory?
RQ2: Which implementation will be the fastest to fully commit a transaction?
RQ3: Which implementation will take the most/least time to complete our simulations.
RQ4: Which implementation produces (on average) the lowest amount of messages in our simulations?
RQ5: What is the average issue closed time for each implementation?

II. Related Work

A. Consensus Algorithms

Consensus algorithms allow systems to achieve reliability and agreement across nodes as they coordinate to accomplish an overarching task. These algorithms maintain consistency via system component states, allowing for each node to correctly progress toward the intended objective. We define consensus to be achieved when all non-faulty components concur on a specific matter at a given moment [3]. Given this, the design of consensus algorithms must focus on how nodes communicate and synchronize their actions, the types of component failures they can withstand, and the protocols needed maintain any achieved consensus.

Systems that attempt to achieve consensus can vary in the degree of coordination between their components. While some systems may employ a fully synchronous model—coordinating operations in batches and confirming among components, more practical systems often adopt a partially synchronous approach. In such systems, operations are not strictly coordinated, but message delivery between nodes is eventually guaranteed (assuming a reliable means of communication between nodes). Thus, addressing component failures becomes the next hurdle in achieving consensus. Failures are typically classified into crash failures[2] and Byzantine failures[3]. Not all consensus algorithms are designed to tackle both types

[1] https://raft.github.io

[2] A component abruptly halts and does not resume.

[3] A component acts maliciously or shares incorrect information with the other components.

of failures, but an effective algorithm will have strategies to handle the anticipated failure modes within its operational context. The consensus protocols themselves establish the rules for network communication and how to deal with component failures, ensuring a consistent state across nodes. Protocols for establishing consensus can so far be categorized into proof-based consensus algorithms, and voting-based consensus algorithms [4].

1) Proof-Based: These algorithms are commonly seen in contexts where security and decentralization is of high priority—a modern example of this being blockchain technology. In such approaches, components need to convey proof that they've expended some defined resource to be able to partake in the process of consensus. Examples include Proof of Work (PoW) and Proof of Stake (PoS), among others.

2) Voting-Based: These algorithms employ a more democratic process, where all nodes have the opportunity to cast votes to arrive at collective decisions. This approach is particularly effective in networks with known and trusted participants. Examples of these types of consensus algorithms would be those like Practical Byzantine Fault Tolerance (PBFT), which enables consensus in the face of Byzantine faults. Consensus protocols like Paxos and Raft, are commonly used in practice.

B. Raft

As mentioned previously, the Raft consensus algorithm is intended to be the focus of this project. Generally, Raft is considered to one of the more understandable and easier to implement consensus algorithms compared to those like Paxos, and it is able to do this without sacrificing correctness and performance. Part of this is has to do with how it has been presented and communicated in academic and practical contexts [5]. As a general overview, Raft clusters are composed of several nodes, one of which is elected as the leader. The leader handles all client interactions that would modify the state of the system, as well as log entry replication and consistency. If the leader fails, a new leader is elected. This election process ensures that there is always a single leader at any given time, simplifying the management of the replicated log. Once a leader is elected, it begins to process client requests, each of which contains a command to be executed by the replicas. The leader appends the command to its log as a new entry and then replicates this entry to the follower nodes. Because raft logs keep note the current election term in which they were created, the system is able to detect inconsistencies and ensure log entries are replicated in the correct order. An entry is considered committed when it's safely replicated on a majority of the nodes. The leader keeps track of this and communicates the commit index to followers, indicating which entries are safe to apply. [2]

C. Algorithm Implementation Comparison

The comparison of differing algorithms is a fundamental research area of computer science. Our research is not the comparison of different consensus algorithms, but different implementations of the same algorithm. We are inspired by the recent work of [6] who compared a traditional Raft implementation to their "energy efficient" version. Their main metric was the number of messages exchanged within the system. Our study plans to go beyond this, also considering other algorithm metrics like CPU time and average memory usage. Additionally, given that a core foundation of Raft is its ease-of-learning, we want to see if its implementations follow that tradition by examining some aspect of their developer experience [7]. We will try to examine the average length of an issue being opened and maintainer activity.

III. Methodology

A. Proposed Experimental Setup

We plan to pick between 2-5 Raft implementations having the following criteria:

- Stand-alone functionality.
- Raft website endorsement (linked from the website).
- Semi-active GitHub community and repository.
- All four criteria listed in the Raft website table (leader election and log replication, persistence, membership changes and log compaction).

From there, we will design a basic simulation which commits and requests key-value pairs from our servers. Additional experiments may also focus on certain edge-cases (sudden leader disconnection, high client traffic, etc.). We plan to use the standard of five servers for each implementation. Ideally, servers and experiments will all be launched on one computer using Docker Compose[4].

B. Proposed Evaluation Metrics

Our evaluation metrics are on a per-implementation level:

- **Memory Usage:** The average memory used at any given point in the simulation.
- **Commit Time:** The average time it takes to complete a single commit.
- **CPU Utilization:** The total user time spent completing the simulation.
- **Total Messages Passed:** The number of messages passed between all nodes within the simulation.

If possible, we would also like to examine the **leader election time** and **ending log size**, but these might require greater under-the-hood access than implementations might allow.

C. Developer Experience Comparison

Given the short turnaround time for this project, we will be unable to get a sample of programmers who have used this program, nor would we be able to set up experiments where participants would implement similar scenarios with our chosen APIs. Instead, we will examine some basic quantitative metrics based on the issues opened on each implementation's GitHub page. We will examine the average issue closed turnaround time to see if it correlates to the number of active maintainers.

[4]https://docs.docker.com/reference/cli/docker/compose/

IV. Expected Outcomes

From our study, we hope to contribute the following:

- A comparison of different Raft implementations according to our metrics defined in III-B.
- A quantitative study that captures the activity and responsiveness of each implementation's development team.

A. Challenges & Limitations

We expect to encounter several limitations during our progress. Due to time, GitHub issues will serve as our user sample for our usability metrics. In addition, we are both new to the world of low level algorithm and implementation comparison, so we expect to have challenges creating and measuring the simulations. We also expect issues related to networking, managing Docker Compose and potentially hardware access. Lastly, the limited time and resources available for the project means we might have to reduce our scope regarding the depth and complexity of the simulations. Nevertheless, we believe this work to be worthwhile.

V. Proposed Timeline

We have outlined the main goal to be reached by each date. If we are unable to meet a given day's goal, we would work on the next goal in-parallel. We plan to divide the work up as we go along.

Proposed Project Timeline	
Date	Goal
April 10	Begin Raft Implementation Selection
April 12	Begin Experiment Development
April 15	Cont. Experiment Development
April 17	Finish Experiment Development
April 19	Begin GitHub Issues Parsing
April 22	Run Experiments & Gather Measurements
April 24	Finish Analyzing GitHub Issues
April 26	Finalized Report & Code Submission

References

[1] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001, edition: ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001). [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[2] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[3] S. S. Shetty, C. A. Kamhoua, and L. L. Njilla, Eds., *Blockchain for Distributed Systems Security*. Hoboken, New Jersey: IEEE Computer Society; John Wiley Sons, 2019. [Online]. Available: https://books.google.com/books?id=dhaMDwAAQBAJ

[4] M. A. Khan, M. T. Quasim, F. Algarni, and A. Alharthi, Eds., *Decentralised Internet of Things: A Blockchain Perspective*, ser. Studies in Big Data. Cham, Switzerland: Springer Nature Switzerland AG, 2020, vol. 71, corrected publication 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-38677-1

[5] H. Howard and R. Mortier, "Paxos vs Raft: have we reached consensus on distributed consensus?" in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3380787.3393681

[6] T. Nakagawa and N. Hayashibara, "Energy Efficient Raft Consensus Algorithm," in *Advances in Network-Based Information Systems*, L. Barolli, T. Enokido, and M. Takizawa, Eds. Cham: Springer International Publishing, 2018, pp. 719–727.

[7] L. Murphy, T. Alliyu, A. Macvean, M. B. Kery, and B. A. Myers, "Preliminary analysis of REST API style guidelines," *Ann Arbor*, vol. 1001, p. 48109, 2017.