IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 June 2024, accepted 25 July 2024, date of publication 30 July 2024, date of current version 9 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3435978

RESEARCH ARTICLE

RaftOptima: An Optimised Raft With Enhanced Fault Tolerance, and Increased Scalability With Low Latency

KIRAN KUMAR KONDRU, (Member, IEEE), AND SARANYA RAJIAKODI
Department of Computer Science, Central University of Tamil Nadu, Thiruvarur 610005, India
Corresponding author: Saranya Rajiakodi (saranya@acad.cutn.ac.in)

ABSTRACT In cloud computing and distributed databases, ensuring strong consistency and system reliability of the leader depends on distributed consensus algorithms such as Raft and Paxos. Raft is particularly appreciated for its simplicity and ease of understanding, making it a go-to choice for contemporary distributed systems. However, it faces challenges in efficiency and reliability, especially during network interruptions and when scaling up. The growing dependence on these consensus protocols for modern applications calls for improvements to address these issues. This paper presents RaftOptima, an advanced version of the Raft algorithm, which incorporates proxy leaders into the system to enhance scalability and introduces robustness against network failures. RaftOptima distributes the tasks of command distribution and gathering responses to proxy leaders, aiming to mitigate Raft's scalability problems and efficiency bottlenecks. Through simulations and TLA+ verification, RaftOptima showed 60% reduction in latency in configurations with up to 25 servers, highlighting its ability to significantly improve performance. Our improvements also increased the resilience of the Raft algorithm against partial network issues. This adaptation not only enhances Raft's operational effectiveness, but also its applicability and reliability in distributed systems for modern computing demands, meeting the critical demand for better scalability and liveness.

INDEX TERMS Raft consensus, aliveness, partial network partitions, scalability, proxy leaders, decreased latency, simulation.

I. INTRODUCTION

The introduction of the Raft [1] consensus algorithm has been a significant milestone in the area of distributed consensus, offering a more understandable alternative to the complex and somewhat opaque Paxos [2] algorithm. Despite its success and widespread adoption for its ease of understanding and implementation, some recent events have underscored the need for further refinement, particularly in the areas of resilience to network partitions and scalability within large distributed systems. As Raft is being adopted to modern edge-based use cases, some of it's non-trivial issues are coming to the front as serious problems.

The motivation for this research is twofold: to solve real-world problems arising out of Raft's wider adoption spanning the globe and to make Raft more ready by making it more resilient. A notable example occurred on 2 November 2020 [3], when Cloudflare, a leading content delivery network, experienced a service disruption lasting over six hours. The outage was initially attributed to a Byzantine fault within 'etcd' [4], a prominent open source distributed key value store that relies on Raft for consensus. However, this diagnosis was later contested by researchers in the distributed systems community, who pinpointed the root cause as a specific edge case that was not adequately addressed by the Raft algorithm [5]. Such an oversight highlights a significant gap in Raft's design concerning its handling of nontrivial network partitions, leading to cascading failures that can severely impact service availability.

The associate editor coordinating the review of this manuscript and approving it for publication was Martin Reisslein.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Concurrently, the increasing reliance on consensus algorithms like Raft and Paxos in the backbone of modern distributed databases and various modern blockchain technologies has illuminated another pressing issue - scalability. As the Raft clusters grow in size, the traditional architecture of consensus protocols, characterized by a singular leader node responsible for managing multiple tasks, viz command sequencing and replication, becomes a serious bottleneck. This centralized design, while effective in smaller settings, significantly hinders performance in larger clusters by increasing latency and reducing throughput. Therefore, the challenge is to devise a mechanism within the Raft algorithm that maintains its core principles of consistency and availability while addressing the scalability bottleneck.

Our article proposes a comprehensive approach that addresses both the resilience and scalability challenges facing Raft. By delving into the specifics of partial network partitions of Raft [6] and their effects on consensus algorithms, we aim to enhance Raft's ability to handle edge cases more robustly, thereby mitigating the risk of cascading failures akin to the Cloudflare incident. This involves a detailed exploration of the underlying causes of such failures and the integration of solutions that bolster Raft's fault tolerance in the face of failures in modern network hardware such as switches and routers.

Also, we address the scalability issue by re-engineering the log replication process of the Raft consensus protocol. Our innovative proposal introduces the concept of proxy leaders, a significant modification designed to distribute the workload of the overburdened leader more evenly across the cluster. By reallocating certain responsibilities from the Raft leader to a select group of proxy leaders, we aim to significantly reduce the latency associated with the process of replicating the leader's log in large clusters. This not only alleviates the burden on the leader, but also enhances the overall performance of the Raft cluster, ensuring significantly lower latency when compared with a simple raft replication. This we achieve without compromising the integrity of the consensus process by not altering the leader election process.

Through repeated simulations and careful comparative analyses, our research evaluates the effectiveness of incorporating proxy leaders into the Raft consensus mechanism. We demonstrate how this adjustment not only addresses immediate scalability concerns, but also makes Raft a more resilient and efficient algorithm capable of supporting the next generation of distributed systems. This dual-focused approach solves multiple problems in the Raft algorithm and makes it more ready for modern applications. Our new improved Raft algorithm would have second and third order effects of the software that uses it - from cloud services and distributed databases to blockchain networks.

This paper seeks to solve significant problems in the Raft consensus algorithm, aiming to solve current and relevant problems and paving the way for more resilient and scalable distributed systems. Through thorough testing of our new Raft through simulations and verification with formal verification, in this paper, we have shown that Raft can be improved and can change to the demands of current trends in Internet-scale computing.

A. ALIVENESS IN RAFT

The Cloudflare incident in November 2020 highlighted a critical vulnerability within 'etcd' (a distributed key-value store) that relies on the Raft consensus algorithm to maintain consistency across its nodes. The initial diagnosis of the outage pointed to a Byzantine fault [7]—an assumption that later scrutiny revealed to be a misdiagnosis. Instead, the true culprit was identified as a nontrivial edge case in the Raft algorithm that it was not originally designed to handle, leading to a cascading failure affecting the availability of services for an extended period.

The incident has ignited a conversation among distributed systems researchers, bringing to light the need for a deeper examination of Raft's approach to handling network partitions and ensuring availability. It was uncovered that Raft is susceptible to specific types of availability problems, notably those categorized as Symmetric and Transitive Reachability issues. These problems, although identified in early discussions surrounding Raft, had not been the focus of significant research or solution development efforts up until recent events compelled a re-evaluation.

Our paper delves into these availability challenges inherent in the Raft consensus algorithm. We show the nature of these problems and the circumstances under which they manifest. We propose a single potential solution that could solve both of the reachability problems mentioned above. We highlight algorithmic changes and modifications to the Raft algorithm that would mitigate these availability concerns and prove the correctness of our changes through formal verification by TLA+.

B. PERFORMANCE IN RAFT

The distributed consensus algorithms (e.g. Paxos and Raft) are fundamental for ensuring robust consistency across distributed networks, and consistency is guaranteed through formal proof. Replicating a deterministic state machine across various nodes guarantees a unified and consistent state view, essential for the integrity and reliability of distributed databases, cloud configuration, and blockchain ledgers. The integration of consensus mechanisms into blockchain technology, especially those capable of operating in Byzantine environments, such as Proof-of-Work and Proof-of-Stake, marks a significant evolution in the design of decentralized systems. Raft, with its simplicity and efficiency in non-Byzantine [7] settings, has been adopted for consortium blockchains, highlighting its flexibility and the wide scope of its applicability.

However, the scalability of such consensus protocols, particularly Raft, in the face of expanding distributed systems presents notable challenges. The centralization of command sequencing and replication in a single leader node introduces bottlenecks that impede scalability and

performance, especially as the size of the system increases. Here, to mitigate the side effects of larger Raft clusters, we propose a novel architectural adjustment within the Raft protocol to address performance bottlenecks. By delegating specific broadcasting and replication duties to a selected group of proxy leaders, we aim to alleviate the central leader's workload, thereby enhancing the overall system's throughput and reducing latency. This approach does not alter the fundamental leader election process of Raft, but redistributes the operational load, potentially transforming the scalability dynamics of the Raft consensus.

Here, we focus on reducing latency as the Raft cluster scales up, while also making the algorithm configurable enough for administrators to take maximum advantage of the underlying network.

We thoroughly evaluated our changes through simulations with varying numbers of nodes, extending up to 25 servers, a scale previously unexplored in typical Raft simulations to the best of our knowledge. Our experimental investigation is particularly timely as the trend towards edge computing necessitates a reevaluation of the conventional limitations on the number of backup replicas. By addressing this problem, this paper seeks not only to elevate the performance of Raft within large clusters, but also to expand its applicability in increasingly distributed computing environments.

II. RELATED WORKS

There is growing research in blockchain technology, and this is reflected in the number of academic papers. The consensus mechanisms like Proof-Of-Work, Proof-Of-Stake, PBFT, etc. are at the core of any blockchain technology. With the rising popularity of permissioned blockchain platforms like Hyperledger Fabric, traditional distributed consensus algorithms like Paxos, Raft, Viewstamped Replication, etc., are seeing wider adoption. Given this, we provide a bird's-eye view of the latest literature on blockchain. The article [8] presents a comprehensive survey of the evolution of blockchain technology. Provides a comparative analysis of frameworks, classification of consensus algorithms, and more. The study [9] provides an overview of the use of Index SGX to improve the security and privacy of blockchain systems. SGX with blockchain is a promising technology. The article [10] provides an in-depth perspective on the deployment of blockchain-based solutions for telecommunications networks, estimating costs, comparing infrastructure options, and choosing appropriate blockchain platforms. This study shows that the interest of blockchain technology in various domains is still growing. The authors in the paper [11] evaluate the frameworks by exploring the risks of double spending and Sybil attacks in Ethereum-based healthcare applications and discuss various other security and implementation challenges of blockchain systems. The article [12] analyses data management in blockchain-based systems, covering aspects from governance to architecture. Examines storage, query processing, provenance management, and data models in blockchain systems. The survey [13] focuses on techniques for scaling blockchain systems to improve transaction throughput and reduce latency. It proposes a taxonomy based on layer 1 and 2 scaling solutions and compares their merits and demerits.

Additionally, Raft's application extends beyond traditional distributed systems, finding utility within consortium blockchains. The paper [14] proposes a simple but accurate analytical model to analyze the probability of split of distributed networks in the Raft consensus algorithm. This model allows for predicting the network split time and optimizing the Raft algorithm parameters. The article [15] proposes KRaft, which is based on K -buckets for permissioned blockchains. Their results show that throughput increased by 41% and election speed by 67% while maintaining the requirements of safety and liveness. The article [16] optimizes the Raft consensus algorithm for the Hyperledger Fabric platform in terms of log replication and leader election. AdRaft algorithm improves throughput by 5.8% and reduces latency by 1.3%. The paper [17] shows that the performance of blockchain consensus can be improved by lowering the probability of network failure. It suggests exploiting federated learning to evaluate them for network stability for private blockchains. The paper [18] proposed weighted RAFT, an improved blockchain consensus mechanism for Internet of Things applications. The exploration of enhancements to the performance and liveness of the Raft consensus algorithm encompasses a wide range of academic and industry research, with several notable contributions mentioned below.

A. ALIVENESS

Many academic papers are proposed to make consensus protocols, like Paxos and Raft, more robust and able to survive failures. Below are some of the articles that try to improve consensus protocols, especially Raft. Their work tries to improve many aspects of the consensus mechanism and not just liveness.

- The study [19] developed a byzantine fault-tolerant version of Raft algorithm called B-Raft and it used Schnorr signature mechanism. Their experiment demonstrated that B-Raft significantly improved safety when compared with the original Raft consensus algorithm.
- The article [20] introduced an improved Raft Consensus algorithm called hhRaft. It adds a monitor role to supervise the participant nodes and identify malicious behaviour during both the leader election and log replication process. The results show that hhRaft outperforms Raft in transaction throughput, consensus latency and anti-byzantine fault capabilities. However, the study does not thoroughly explore the potential vulnerabilities or attack vectors that could still affect the hhRaft algorithm in highly adversarial environments
- The paper [21] proposed MLRaft which divides the single Raft log file intro multiple logs, each with a

different leader of it's own. These multiple leaders work together to improve throughput, latency and load balancing functions. Experiments shows it out performed the original Raft. However the paper does not address the potential complexities and overhead introduced by managing multiple leaders and log files and the experimental setup is limited to a 3 node Key-Value storage system which may not fully represent the scalability and robustness of the protocol.

- The article [22] developed Pirogue, a lighter and dynamic version of Raft that replaces static quorums with dynamic ones. It bases this on the number of available nodes in the cluster. Analysis shows that a 4-node Pirogue cluster have the nearly the same availability as that of a 5-node ordinary Raft cluster while also tolerating failures. However, Pirogue introduces additional dynamic metadata on top of the Raft update protocol. This adds some complexity. The analysis is also limited to three specific configurations. The analysis assumes server failures are independent events following an exponential distribution. Real-world failure patterns might deviate.
- The study [23] argues for Raft as a better replacement for P2P systems like Cassandra databases. Their experiments proved that Raft offers better load balancing and performance than Paxos. The testing in the context of Cassandra Distributed Database. The experiment were conducted using 4 virtual machine in a single physical server. Testing on more realistic multi-server deployments could reveal additional performance and scalability limitations
- The paper [24] an energy efficient Raft variant. The key idea is to disable some nodes that do not participate in leader election (as per the configurable ''suspended rate'' parameter). This reduces the number of messages required thereby improving energy efficiency. The algorithm resumes suspended nodes if the number of active nodes decreases a certain majority. Simulations showed the average number of messages is reduced by up-to 40 %. However, the paper does not quantify the actual energy savings achieved when compared to the original Raft algorithm. By disabling some nodes from participating in leader election, the protocol potentially sacrifices some degree of fault-tolerance.
- The paper [25] proposes Raft-PLUS, an improved Raft algorithm that addresses the asymmetric relationship between the leader and the follower nodes. Here the followers can send negative votes to the leader if performance deteriorates. The leader steps down voluntarily if it receives majority of negative votes. The voting mechanism also changes with multiple policies. Their experiments prove that when network quality fluctuates, Raft-PLUS provided a 38-40% higher average write throughput. However the study has some limitations. It would have been good if a formal proof of correctness is provided. The experimental simulation is done with a small 12-node cluster. For evaluating scalability and performance in a larger more realistic deployments would have been better.

A particularly compelling study [26] ventured into the realm of experimental replication through the development of a Discrete Event Simulator (DES) in O'Caml [27]. This work underscores the challenges inherent in systems research, especially within computer science, where the replication of experimental results is fraught with complexities due to the intricate design and configuration diversity of networked systems. Despite these hurdles, the researchers achieved results similar to those promised by Raft, although not identical, highlighting the nuanced differences that experimental settings can introduce. Their findings also illuminated potential optimizations that could further refine Raft's operational efficiency.

Moreover, the discourse around addressing Raft's reachability issues has seen the proposition of alternative mechanisms like Pre-vote [28], aimed at mitigating specific scenarios that could compromise the system's integrity. However, the Pre-vote mechanism, despite its merits, does not offer a panacea for all reachability challenges. It opens the dialogue for additional solutions to these complex problems, emphasizing the need for continued innovation and exploration to enhance the robustness and reliability of consensus algorithms. This article aims to contribute to this ongoing conversation by proposing alternative strategies to navigate the nuanced challenges of network reachability within Raft-based systems, seeking to improve their resilience and performance in distributed environments.

B. PERFORMANCE

The quest to improve the performance of consensus algorithms such as Raft and Paxos has spurred a multitude of research efforts aimed at mitigating latency and increasing throughput. These endeavors typically address specific architectural bottlenecks with the goal of refining the efficiency of these protocols without eroding the safety properties they guarantee. In the following, we outline several notable studies that have ventured to improve Raft throughput or latency, showcasing their objectives, methodologies, and contributions within the context of this paper's focus. Although not exhaustive, this summary emphasizes studies of particular relevance to our investigation.

- An Optimized Key-Value Raft Algorithm for Satisfying Linearizable Consistency [29]: This research introduces ALB-Raft, an optimized iteration of Raft designed to boost performance by resolving log entry conflicts and minimizing message traffic for linearizable consistency. The study highlights ABL-Raft's efficacy in enhancing key-value-based Raft algorithm performance through rapid conflict resolution and backward tracer updates.
- MLRaft: Improvement of Raft Based on Multi-log Synchronization Model [21]: Proposed an innovative

approach - MLRaft employs multiple logs with distinct leaders to process commands in parallel, thereby increasing throughput. The study introduces a dynamic leader transfer mechanism to maintain balance within the cluster, effectively distributing workload across multiple leaders.

- Research on Optimization of an Efficient and scalable Multi-Raft Consensus Algorithm for Supply Chain Finance [30]: Addressing scalability and performance in supply chain finance, this paper develops a multi-raft consensus algorithm that leverages contractor nodes, selected via a machine learning-based KNN algorithm, to expedite consensus, thus enhancing scalability and efficiency.
- An Improved Raft Consensus Algorithm Based on Asynchronous Batch Processing [31]: This study aims to augment Raft's throughput by introducing a pre-proposal phase for batch log processing. By facilitating multiple leaders within a Raft cluster, the proposed methodology seeks to alleviate bottlenecks associated with a singular leader, significantly increasing workload management and efficiency.
- Raft Consensus Algorithm Based on the Reputation Mechanism [32]: Tailored for consortium blockchains, this research adapts Raft to Byzantine environments by integrating a reputation mechanism. By evaluating node honesty and excluding potentially malicious nodes, the study aims to enhance Raft's security and reliability in diverse network conditions.
- Rethink the Linearizability Constraints of Raft for Distributed Key-Value Stores [33]: This paper presents KV-Raft, an improved algorithm that accelerates operations in distributed key-value storage by optimizing write and read processes. By challenging and relaxing certain Raft constraints, the study reports significant throughput improvements and latency reductions.
- vRaft: Accelerating the distributed consensus in virtualized environments [34]: vRaft, a variant of Raft optimized for virtualized settings, introduces fast followers to improve performance. The study focuses on optimizing log replication and management processes to improve performance in distributed consensus environments.

These research initiatives collectively underscore the ongoing efforts to refine Raft and related consensus mechanisms, addressing the critical balance between throughput enhancement and latency reduction. As distributed systems continue to evolve, the insights derived from these studies contribute significantly to the development of more efficient, scalable, and robust consensus algorithms.

III. BACKGROUND

The Raft consensus algorithm, similar to Paxos [2], operates on the assumption of an asynchronous network environment, where messages/packets might encounter drop, reordering, or delay. Furthermore, it assumes a non-Byzantine network, where the nodes in the cluster do not engage in malicious behavior. Like the Paxos consensus algorithm, Raft does not require clock synchronization. The configuration of a Raft cluster is fixed, and clients exclusively communicate through the leader, even for read requests. Furthermore, Raft assumes a one-to-one direct connection between all nodes in the cluster, regardless of the underlying network topography.

A. RAFT ALGORITHM

In this section, we discuss the components of the Raft Algorithm and how it operates, in general. The following description establishes the background necessary to understand how the algorithm works.

1) LEADER ELECTION

The Raft consensus algorithm [35] operates under the paradigm of State Machine Replication (SMR), using a leadership-based model in which a leader is elected for each term through an election mechanism. In instances where a leader is absent, a server will time out in a randomized fashion and initiate an election by soliciting votes from its peers. Votes are granted on a first-come-first-served basis, underscoring the algorithm's design focus on fault tolerance. This design principle allows the cluster to remain operational as long as a majority of nodes (more than half) are functioning and can communicate with the leader, ensuring resilience against node failures. Specifically, a Raft cluster can tolerate failures of 'f' if it includes nodes '2f + 1'.

Raft's architecture is segmented into three principal states or roles: (1) Follower, (2) Candidate, and (3) Leader. These roles facilitate a structured transition of responsibilities within the cluster, ensuring seamless operations even in the face of server breakdowns. The state transition between the states mentioned above is illustrated in Figure 1. Should a leader server fail or stop, a follower, after a certain amount of time, will escalate to a candidate status, launching a new election to elect a successor. This flexibility enables Raft to effectively manage network partitions, maintaining operational continuity and swiftly recovering after reconnection. This capability ensures that the Raft algorithm not only prioritizes consistency and availability but also remains alive and serves in diverse network conditions.

Every Replicated State Machine algorithm maintains a sequential log of commands, including Raft. The operational phases of a Raft cluster, shown in the sequence diagram 2, unfold in two main stages. At the outset, in scenarios where a leader is absent or a leader's failure occurs, the process kicks off with a follower timing out, leading to the initiation of an election. This step involves incrementing the term number and the follower transitioning into a candidate status. The candidate then seeks votes from its followers in clusters. Achieving a majority of votes enables the candidate to become the leader.

Once at power, the new leader sends an initial empty *AppendEntries* message across the cluster to announce the start of a new term and to establish itself as the leader. These

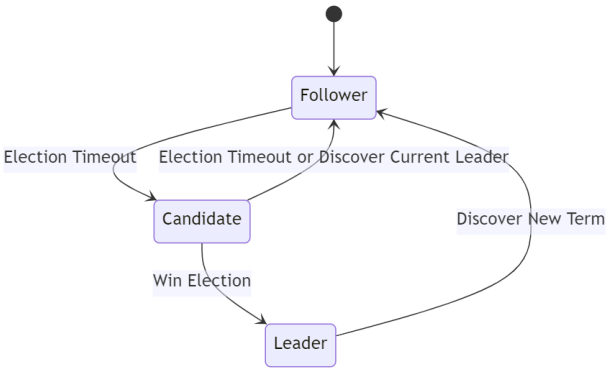

FIGURE 1. State transition diagram of raft consensus algorithm.

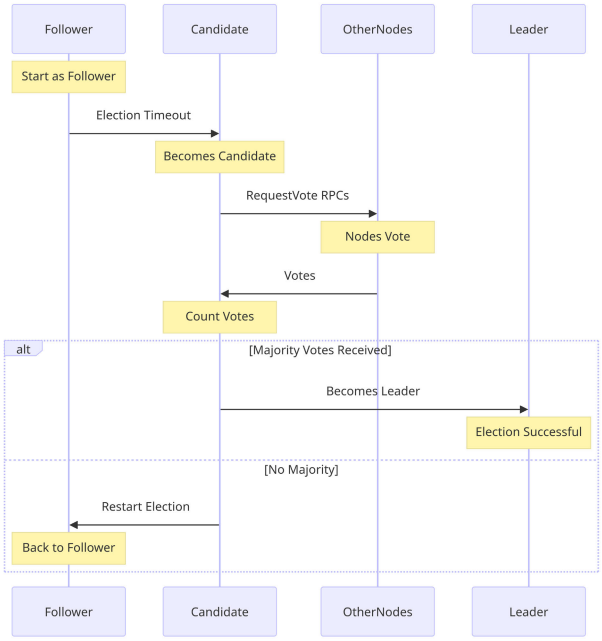

FIGURE 2. Sequence diagram for leader election in raft.

empty *AppendEntries* messages are also periodically sent to reinforce the leader's status to the followers, ensuring that they are reminded of who is currently leading. The reception of this empty or loaded *AppendEntries* prompts followers to reset their election timeouts, thus preventing the occurrence of unnecessary elections. This process is represented in the activity diagram in Figure 3.

2) LOG REPLICATION

Figure 4 shows the structure of a log within a Raft node, which consists of a series of entries. Each entry in this log includes a term number and a single client's command. This log operates on an append-only basis, meaning it continuously adds new entries without altering or removing existing ones. This leader's log is replicated among the follower nodes as is. It is common for followers to lag behind the leader; however, the leader persistently issues '*AppendEntries*' commands to ensure the followers eventually synchronize with the leader's log as can be seen

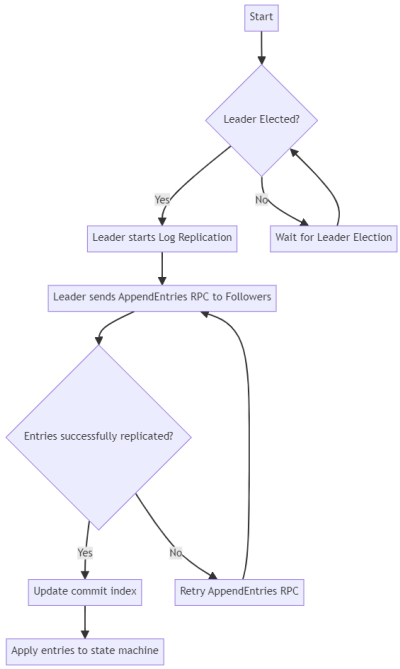

FIGURE 3. Activity diagram for log replication.

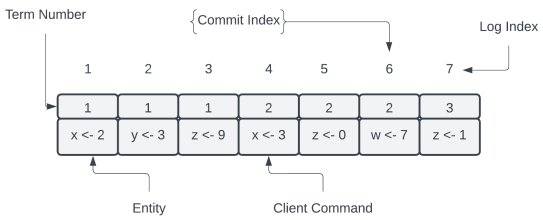

FIGURE 4. Entities and log of a raft node.

in Figure 3. An entry is considered 'committed' once it has been duplicated on the majority of nodes within the cluster. Following a majority commit, the leader then communicates the 'success' message back to the originating client.

Figure 5 elucidates the operational dynamics of a replicated state machine within a Raft cluster, highlighting the interaction process with clients. In this scenario, a client sends a command, such as set(x,2), directly to the Raft Leader, reflecting the system's design where only the Leader is authorized to process client requests. This mechanism ensures consistency and simplifies the command routing logic; should a Follower receive a request, it redirects the client to the Leader, providing the Leader's address if available. This setup presupposes that the clients are aware of the IP addresses for all nodes within the Raft cluster.

The architecture of a Raft server integrates three core components: (1) the consensus module, (2) the replicated log, and (3) the state machine, with their interactions shown in Figure 5. Upon receiving a command, the Leader stores it in its log as a new entry, which includes the command itself and the term number of the current leader. This entry is then propagated to the rest of the cluster through the consensus module's *AppendEntries* command, ensuring that all Followers replicate the entry in their local logs.

The *AppendEntries* message is comprehensive, containing information such as the term number, leader ID, previous log index, and term, among others. For efficient log replication, the Leader keeps track of two indices for each Follower: *nextIndex*, indicating the next entry to send, and *matchIndex*, reflecting the highest log entry confirmed to be replicated on the Follower. This structured approach facilitates precise and reliable log synchronization across the cluster.

The *AppendEntries* command is crucial to ensuring the integrity of each follower's log. When discrepancies arise, be it from server lags or packet losses on the network, the leader, tasked with monitoring the *nextIndex* and *matchIndex* for each follower, initiates the process of sending missing entries. This ensures that the follower's log is brought into alignment with the leader's log.

Followers, upon receiving these entries from the leader, integrate them into their local logs and acknowledge the leader. They then sequentially apply these new entries to their state machines from the log. Once the leader has received affirmative responses from the majority for the *AppendEntries* command, it sets the *commitIndex*, indicating which entries are reliably stored in the majority of nodes and can be considered committed. The leader then informs the client about the successful operation and applies the committed entry to its state machine.

This methodical replication and application of log entries ensure that all node state machines are synchronized. Consequently, even in the event of a leader's failure and the subsequent election of a new leader, the state machine's continuity and integrity are preserved, maintaining the cluster's overall stability and reliability.

IV. ALIVENESS IN RAFT CLUSTER

The Raft consensus algorithm operates under a set of predefined assumptions [36] about the environment in which it functions:

1) It is predicated on the assumption of a Byzantine fault-free context, offering support solely for tolerance to crash faults.
2) The system does not rely on synchronized global clocks, instead operating within an inherently asynchronous distributed environment.
3) Considerations for network-related issues such as delays, packet loss, and duplication, and network partitions are integral to the protocol's design.
4) Client communications are handled exclusively by the leader. Requests made to followers are redirected to the leader to ensure centralized processing.
5) Designed as a Replicated State Machine, the algorithm requires that all node state machines begin from an identical initial state, with client interactions processed in a deterministic manner.
6) The architecture assumes that nodes have unlimited access to secondary storage, which is presumed to be reliable and secure against corruption.
7) Initially, the configuration of a Raft node cluster is static, without support for dynamic changes. However, subsequent enhancements, as discussed in extended literature, have introduced a degree of flexibility to it's configuration for the Raft algorithm.

A. ISSUES WITH RAFT ALIVENESS

Numerous cloud services leverage the Raft consensus algorithm for distributed systems such as key-value stores, with 'etcd' being a prominent example of an open source distributed store that uses Raft to synchronize all cluster nodes with the leader. An incident in Cloudflare [37], attributed to a malfunction in a network device, illustrates the unique challenge facing Raft in maintaining cluster coherence. This malfunction allowed unidirectional communication, effectively isolating a node by preventing it from receiving any packets, including the leader's heartbeat messages.

Examination of the Symmetric Reachability problem in Figure 6 and the Transitive Reachability Problem Figure 7 reveals significant insight into the challenges faced by the Raft cluster. In Figure 6, a node located on the top right does not receive incoming messages, leading to a timeout, an increase in its term to 4, and the initiation of an election by seeking votes from other nodes. However, due to a malfunctioning network device, this candidate does not receive the votes cast. Subsequently, another node, having not received heartbeat messages from the assumed new leader, times out, raises its term to 5, and initiates its election. Secures the majority of votes, assuming leadership, yet the cycle of elections continues due to the continuing communication failure of the node on the other side of the faulty network device, rendering the cluster perpetually stuck in election mode and incapable of serving client requests.

Similarly, the transitive reachability issue shown in Figure 7 shows the ease with which a leader can be replaced by simply initiating an election with a higher term, suggesting that the mechanism to call elections could be a contributing factor to cluster instability. In this scenario, two nodes are disconnected, with one being the leader of term 3. The disconnected node, unable to receive heartbeats, times out, and seeks leadership through a new election. The rest of the cluster, receiving this new vote request with a higher term number, responds positively, leading to a leadership change. The original leader, upon recognizing the new term and the leader, relinquishes its role because of the higher term. After the old leader relinquishes control, it awaits the new leader's heartbeat, but since there is no direct link between the two nodes, the old leader, now a follower, starts its own election with an incremented term number. This oscillation in leadership is driven by network issues.

The PreVote mechanism, as described in the Foundational thesis [35], aims to stabilize cluster operations by requiring partitioned nodes to obtain preliminary consensus before escalating their term numbers for election reentry. This protocol adjustment is specifically designed to address disruptions

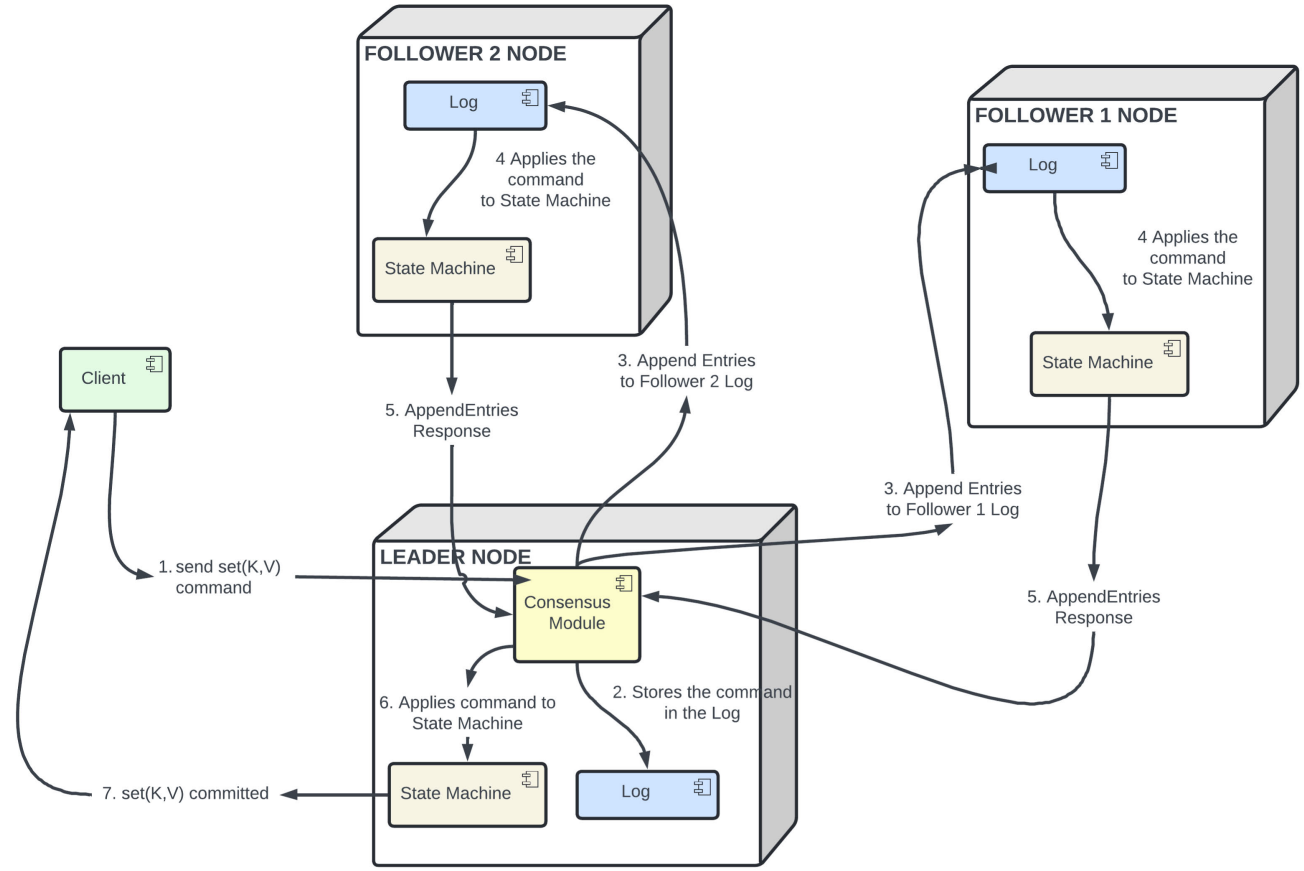

FIGURE 5. Component Diagram - Replicated State Machine Architecture of Raft Consensus.

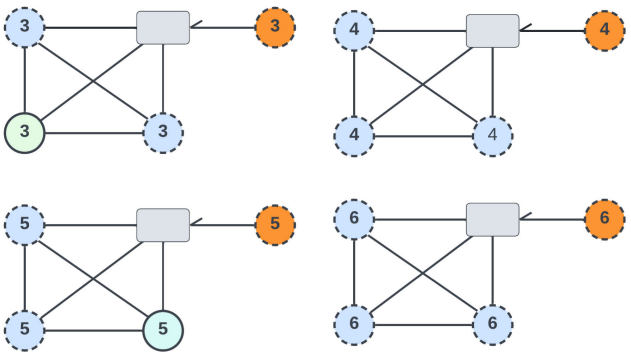

FIGURE 6. Issue of asymmetric rechability.

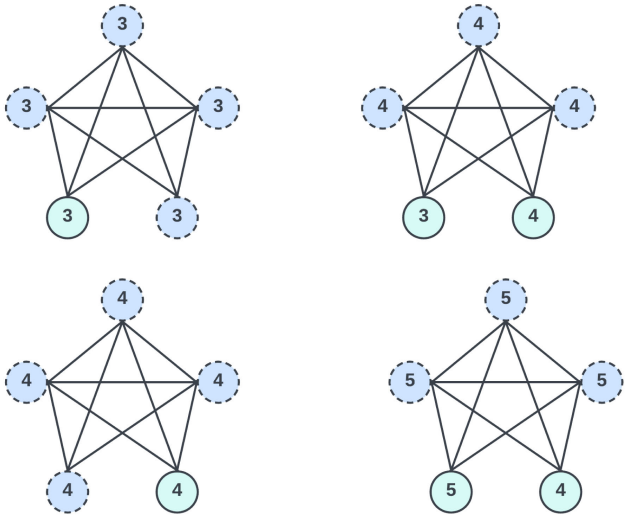

FIGURE 7. Issue of non-transitive rechability.

associated with a node's reintegration postpartition, ensuring that it does not unilaterally trigger an election or disrupt ongoing cluster activities with an outdated high term.

Upon reintegration, a node must issue a PreVote request to gauge cluster readiness for its potential leadership. The mechanism effectively prevents the node from advancing its term and participating in the election process without the approval of the majority. Nodes with PreVote enabled prioritize the continuity of the existing leadership, denying PreVote requests if they detect heartbeats from the current leader within the election timeout window.

Although PreVote improves cluster resilience against specific reachability issues (nontransitive and asymmetric), it is not a panacea. In complex scenarios, such as in a five-node configuration, PreVote may inadvertently exacerbate availability concerns, underscoring the need for a nuanced

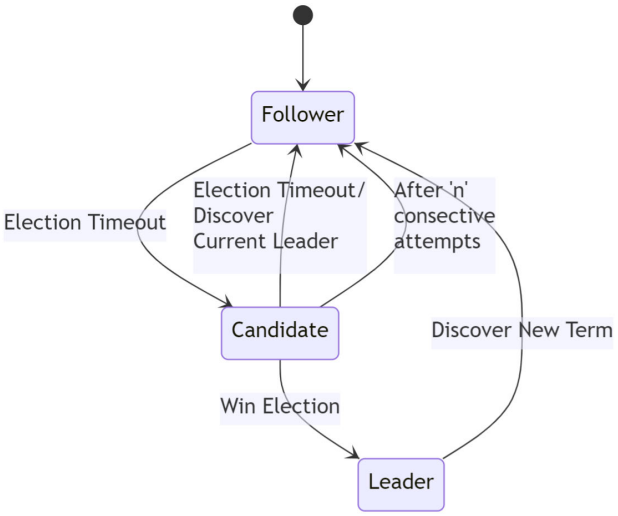

FIGURE 8. State transition diagram for the new improved algorithm.

application of this protocol extension in diverse operational contexts. This is explored in the blog post [28]

B. RAFTOPTIMA - CHANGES TO LEADER ELECTION

Adjustments to distributed algorithms can inadvertently lead to unforeseen problems that affect liveness or throughput. In search of a more straightforward and elegant solution that mitigates liveness challenges without introducing new complications, we propose an approach centered on simplicity.

Our strategy involves enabling a candidate node, persistently initiating elections across a faulty network device, to monitor its repeated term increments and vote solicitations. Recognizing this pattern as indicative of a communication issue, the node ceases further election participation, thus curtailing continuous re-election cycles at the cost of sidelining one backup node. This trade-off is preferable to the entire cluster being ensnared in endless elections, rendering it incapable of client service. Subsequent intervention by an administrator to rectify the communication flaw and reboot the node allows it to rejoin the cluster as a follower, restoring its operational stance.

Distributed consensus algorithms, typically framed as state machines, benefit from this paradigm because of the precise modeling it provides. The life cycle of distributed systems, from conception and design to implementation and testing, is fraught with complexity and susceptibility to errors. Employing state machines as a conceptual foundation enhances understanding and intuition, while formal modeling in languages such as TLA + [38] and Coq [39] facilitates early detection and correction of potential error states.

However, the introduction of additional states can lead to a combinatorial explosion in state space, making exhaustive outcome evaluation both impractical and undesirable. With this consideration, our solution eschews the addition of new states, opting instead for a refined approach that leverages existing states effectively.

We illustrate in Figure 8 our modified transition schema. Our focus is on rectifying the issue of perpetual leader elections triggered by partial network link failures, where a candidate node, failing to receive vote confirmations or leader notifications, persists indefinitely in its election efforts. To counteract this, we suggest limiting the election initiation to a predefined threshold 'n', set at server startup. Upon reaching this limit, the candidate ceases to make election attempts, reverting to a permanent follower state with its term adjusted to reflect its status prior to the cyclic election attempts.

Algorithm 1 RaftOptima

```
1  if RPC request or response contains term T > currentTerm then
2      set currentTerm = T, convert to follower
3  end
4  if (currentTerm - lastCandidateTerm) => 1 then
5      increment repeatedElectionAttemp
6  else
7      set repeatedElectionAttemp to Zero
8  end
9  if repeatedElectionAttempt >= 'n' repeated attempts then
10     Stop resetting election timer;
11     Set currentRole as Follower;
12     Set lastCandidateTerm to currentTerm;
13     Return;
14 end
15 if conversion to candidate then
16     start election;
17     Increment currentTerm;
18     Vote for self; Reset election timer ;
19     Send RequestVote RPCs to all other servers;
20     if votes received from majority of servers then
21         become leader
22     end
23     if AppendEntries RPC received from new leader then
24         convert to follower ;
25     end
26     if election timeout elapses then
27         start new election;
28     end
29 end
```

1) PROPOSED ALGORITHM

In the revised algorithm, RaftOptima, we add the code from lines 1 to 15 [1]. In our algorithm 1, we introduce two variables to track the last term encountered by a candidate and the count of its attempted elections. These variables are instrumental in identifying patterns of repetitive term escalations. The variable tracking repeated elections is incremented with each successive term increase, closely

monitored up to a predefined threshold 'n' (for example, 5). Reaching this threshold signals the realization of the node's disruptive impact on cluster operations, likely due to a network link failure.

Subsequently, the node transitions to a follower state, abstaining from further attempts at starting the election. This modification ensures that the node abstains from election participation, reducing the pool of potential leaders but mitigating the risk of continuous election cycles that could destabilize the cluster. Although the reduction in eligible leader nodes raises concerns, we argue that maintaining cluster stability and preventing service disruptions outweigh the drawbacks. This approach aims to circumvent the extensive delays and misdiagnoses associated with incidents such as the CloudFlare outage, where initial error attributions to Byzantine faults diverted attention from actual connectivity issues.

V. PERFORMANCE IN RAFT CLUSTER

The Raft consensus algorithm is acclaimed for its robust fault tolerance, particularly effective when the number of faulty nodes is less than half of the cluster, and its operational efficiency with a limited number of backup replicas. The leader election mechanism within Raft has undergone formal verification using TLA+ [40], demonstrating its reliability. In the proposed change to log-replication part of the algorithm, we introduce a refinement aimed at enhancing Raft's performance metrics, notably latency. We did not change the leader election process deliberately as it might introduce more bugs. Our preliminary work towards enhancing the performance of log replication is here [41]. Here, we attempted to create a multilevel log propagation with one main leader and one proxy leader. The leader delegates the process of log propagation to the proxy leader, and the leader keeps on accepting the client's requests. However, here in this paper, we have improved upon it and tested it thoroughly with simulators. The following paragraphs describe how we achieved performance enhancements.

After election, a leader appoints several proxy leaders tasked with disseminating the "*AppendEntries*" command and gathering responses. This modification facilitates a scenario where, upon receiving a write request from a client, the leader forwards this request to the proxy leaders.

These proxy leaders then relay the command to their designated followers and collect their responses. This strategy allows the proxy leader to act as an intermediary, ensuring the rapid dissemination of log updates from the leader to the followers. The allocation of followers to specific proxy leaders by the leader is a strategic move designed to optimize response times, particularly beneficial for followers that are geographically dispersed.

Proxy leaders consolidate the responses from their followers and relay this information back to the leader, who then aggregates these responses to determine the success of the write operation before notifying the client. This process is illustrated in the diagram in Figure 9. While this modification redistributes the communication load, the ultimate authority to confirm the success of the write r to the client remains with the leader, preserving the integrity and centralized decision-making characteristic of the Raft protocol.

Unlike the hierarchical model of [42], which employs nested Raft clusters to manage a larger ecosystem, our changes maintain the original quorum structure while introducing proxy leaders to share the load of message broadcasting and response collection. Similar work is done by [43] with multiple optimizations of two of the Paxos derivatives, but this work makes changes directly to the Raft algorithm. The paper [44] proposes a method to collect blockchain transactions by dividing those transactions into cells.

This choice minimizes the need for significant alterations or complete rewrites of the existing algorithm, addressing the common issue of bugs that arise from the complex nature and often insufficient testing of distributed systems. The inherent unpredictability in distributed system behaviors underscores the limitations of unit testing and highlights the necessity for deterministic simulation to accurately validate system performance and reliability.

Our adaptation of the Raft algorithm aims to improve its operational efficiency without diluting its fundamental principles. After the leader chooses the proxy leaders, the leader itself chooses the followers for each of these proxies. In our implementation, we distributed the followers equally to each of the proxy leaders. But administrators can implement in such a way that proxy leaders and their associated followers can be assigned as suited. This allows for a targeted approach to optimize network performance based on geographic considerations and latency metrics. This flexible configuration not only distributes the communication workload more evenly, but also aims to reduce overall network latencies by optimizing interactions within strategically defined subgroups of the cluster. This modification leverages the spatial distribution of the nodes to improve the responsiveness and throughput of the Raft-based system. Here, we have to mention that the proxy leaders too have the right to vote on the log replication RPC from the leader. It's response - *AppendEntriesReponse* - is also included when the response is sent back to the Leader. The proxy leader is also a follower of the leader. As such, the core function of the leader election mechanism is also applicable to proxy leaders. If a proxy leader did not receive a heartbeat from the leader, it can be the first to timeout and start a new election seeking votes from the followers and become a leader.

This architecture introduces additional message types - *ProxyAppendEntries* and *ProxyAppendEntriesResponse* - and appears to add an additional communication layer. However, this structure significantly offloads the leader's broadcasting and consensus duties by incorporating parallelism in these operations. It is important to note that Raft mandates persistence of both server logs and state machines on disk, a process that inherently delays operations. Our model preserves the original server roles without altering

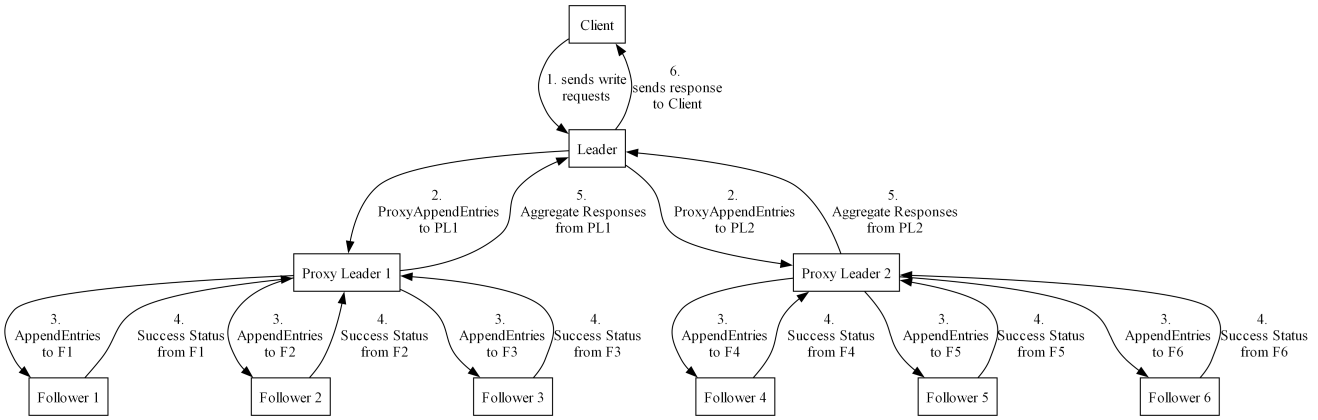

FIGURE 9. Interaction diagram for raftoptima with 2 proxy leaders.

Algorithm 2 Followers

```
Respond to RPCs from candidates to leaders if the election timeout elapses without AppendEntries then
    RPC from current leader or granting vote to candidate ;
    convert to candidate
end
if selected by the leader then
    convert to proxy leader ;
    get the list of followers from the leader for broadcast
end
```

Algorithm 3 Proxy Leader

```
Upon Leader selecting Follower, convert to Proxy Leader if AppendEntries received from Leader then
    append entry to local log first ;
    respond after entry applied to state machine ;
end
if last log index >= nextIndex for a Follower in the group then
    send AppendEntries with log entries starting at nextIndex;
    if successful then
        update nextIndex and matchIndex for that Follower
    end
    if AppendEntries fails because of log inconsistency then
        decrement nextIndex and retry;
    end
end
if there exists an N such that N > commitIndex, a majority of matchIndex[i] >= N and log[N]. term == currentTerm then
    Set commitIndex = N
end
if empty AppendEntries from a new leader emerge with higher term number then
    convert to Follower
end
```

the core rules, but adds a "Proxy Leader" state to the existing Follower, Candidate, and Leader states, simplifying algorithm development.

After the election of the leader and the initial heartbeat signal establishing the leadership of the cluster, the proxy leaders are chosen, typically based on the latency of the follower, a task managed by the cluster administrator to enhance the efficiency of the configuration. Proxy leaders are tasked with disseminating the leader's *AppendEntries* commands and gathering follower responses. This mechanism, designed exclusively for log replication, does not modify the leader-election process. In the event of leader failure, any follower can assume candidacy and initiate an election, and the new leader replicating the delegation process outlined above.

We have divided the raft algorithm into follower part in algorithm 2, proxy leader part in algorithm 3, and leader part in algorithm 4. We have followed the style and pseudocode of the original raft algorithm from [1]. In this modified Raft framework in parts of algorithms, RaftOptima, the leader replicates logs exclusively to its designated proxy leaders using the *AppendEntries* command. The proxy leaders are responsible for further broadcasting these commands to their followers and compiling the responses. These responses, including the count of "*success*" messages and the "*term number*," are then relayed back to the leader. Should the leader encounter a returned term number exceeding its own, it concedes its position, acknowledging the presence of a new leader. This mechanism is aligned with the original Raft design, ensuring that the fundamental safety properties of the protocol remain intact. This preservation of the integrity of the original protocol is crucial to maintaining the reliability and fault tolerance of the algorithm without compromising its core safety features.

However, there are some corner cases that should be mentioned for the purpose of completeness, especially when

there is a failure of the proxy leader. We describe below how failure of a proxy leader in our modified RaftOptima is handled in a point-by-point manner.

- The leader sends log replication commands (AppendEntries RPC) or heartbeats to all proxy leaders
- The leader receives acknowledgements from the proxy leaders
- If a proxy leader doesn't respond after several retries, the leader assumes it has failed
- The leader chooses a follower from the failed proxy leader's sub-group
- The leader elevates this follower to be the new proxy leader
- The new proxy leader sends heartbeats to its sub-group
- This informs the followers of the change in proxy leadership
- If the old proxy leader comes back online, it starts as a follower
- After receiving a heartbeat or ProxyAppendEntries, it continues as a follower

Algorithm 4 Leader

```
Upon election: send initial empty AppendEntries RPCs
(heartbeat) to each server; repeat during idle periods to
prevent election timeouts ;
if command received from client then
    append entry to local log, respond after the entry
    applied to state machine ;
end
if last log index >= nextIndex for a Proxy Leader then
    send AppendEntries RPC with log entries starting at
    nextIndex ;
    if successful then
        update nextindex and matchIndex for Follower
    end
    if AppendEnries fails because of log inconsistency
    then
        decrement nextIndex and retry
    end
end
if there exists an N such that N > commitIndex,
a majority of matchIndex[i] >= N and log[N] then
    term == currentTerm ;
    set commitIndex = N
end
```

VI. EVALUATION

The choice of Java over higher-performance languages like *C* or *Rust* was motivated by its widespread familiarity among students and developers, along with robust multithreading support, despite Java's relative performance disadvantages. Our multithreaded Raft model was meticulously synchronized to make it configurable, enabling simulations with varying counts of Raft servers and clients. This design flexibility allows for extensive scalability in testing, supporting simulations with up to 100 Raft servers and thousands of clients. More details are provided in Table 1

TABLE 1. Simulation environment.

Criteria	Detail
Device Type	Desktop
CPU	Intel Core i3-4030
Memory	8GB
OS	Windows 10
Language	Java (JDK18)
Encoding	UTF-18
Main libraries	java.io, java.net, java.util, https://raft.github.io/

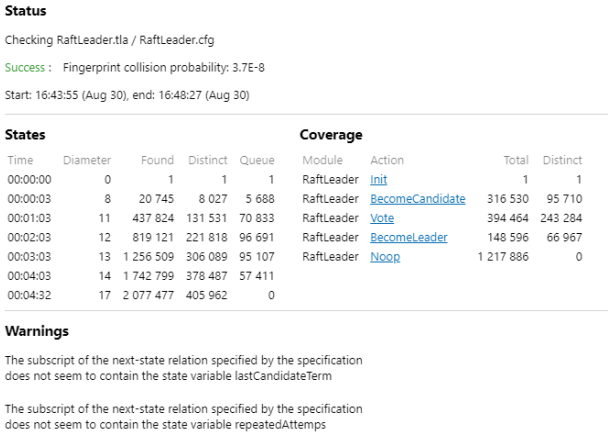

FIGURE 10. Result of tla+ model checking.

For communication, we employed sockets, conducting numerous trials with hundreds of clients and multiple Raft servers. Although our implementation was executed on a single system for simplicity, it is adaptable for distributed setups across multiple machines. Our tests ranged between different Raft server configurations, from 5 to 25, with outcomes recorded in a *CSV* file and visualized through graphs. The primary focus was on comparing the original Raft algorithm's throughput and latency against modified versions incorporating two proxy leaders and assessing performance impacts across various server scales.

A. ALIVENESS

1) FORMAL VERIFICATION

To validate the integrity and correctness of our modified leader election algorithm, we adapted the TLA + code for the leader election and used the TLA + model checker to scrutinize it for potential undefined states. TLA+ provides a mathematical framework for algorithmic modeling, enabling a precise specification of state transitions within predefined boundaries. This approach aligns with the State Machine paradigm and has been widely adopted by major corporations such as Amazon and Microsoft for verifying the reliability of complex distributed systems' algorithms, including Paxos and Raft. Using TLA + to formally verify our algorithm, we can mathematically assert its functionality in all scenarios, ensuring that no unexpected states arise during execution.

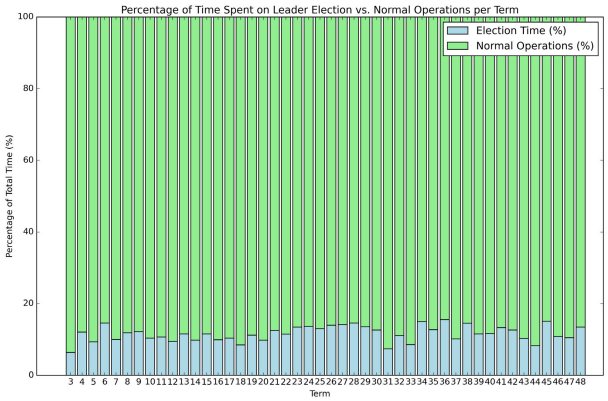

FIGURE 11. Repeated leader election.

The fundamental principles of the TLA+ model facilitate this rigorous validation process, offering a robust tool to ensure algorithmic correctness and system reliability. We have modified the code from [45] and ran a model check to ensure that our changes to the leader election do not break the whole process. We have added two new variables *lastCandidateTerm* and *repeatedAttempts* to the TLA+ code and altered it by incorporating the above two variables. The result of the model check in TLA + is shown in Figure10

2) RESULTS

The simulation results, depicted in Figure 12, illustrate the effects of modifying the algorithm with a set value n, which limits the allowed leader transitions (flip-flops) to enhance the stability of the cluster. Figure11 illustrates the time taken for the leader election and the operational time for each term as a percentage. As can be seen in the graph, significant time is being consumed by the leader election process, halting serving clients. This is the reason for the severe performance degradation reported by the Cloudflare incident [3]. Figure 12 shows the leader election times for each term when the threshold is set to 3, 9, and 15 respectively. The green line shows that after 3 flip-flops, that is, 6 leadership changes (6 terms), it stops. So is the case with the n value set to 9 and 15. This illustrates that the whole cluster has stabilized. Further tests with increased 'n' values, such as 9 and 15, demonstrate that the cluster stabilizes after approximately the same number of flip-flops as the 'n' value. Figure 13 shows how the cluster came to operational mode after flip-flops 3, 9, and 15. The sudden and steep spike of the election time shows that no more elections are started until the simulation run is over.

Our RaftOptima modification effectively prevents a node with a compromised communication link from participating in the election process until the link is repaired and the node is re-started. Given that a follower's primary role within the cluster is to serve as a standby during leader failures, this adjustment ensures the cluster's liveness property is maintained, safeguarding overall system reliability and operational continuity.

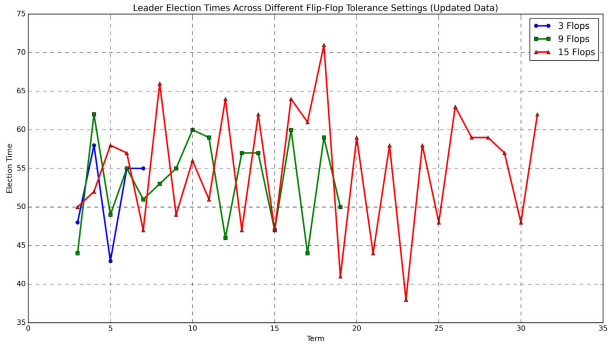

FIGURE 12. Stopping of flip-flopping of Leader election after 'n' attempts.

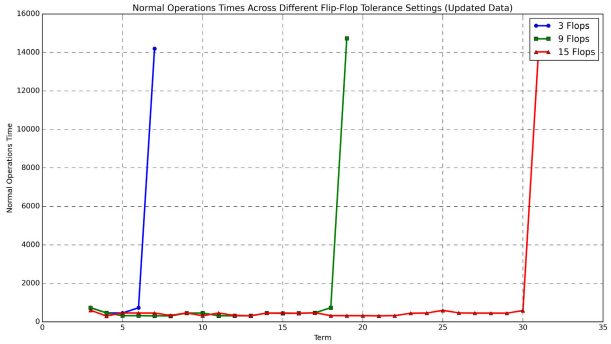

FIGURE 13. Election Time stopped and normal operations returned after '3', '9' and '15' flip-flops.

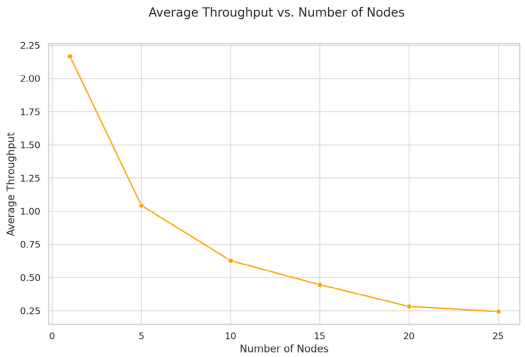

FIGURE 14. Average throughput vs number of nodes.

B. PERFORMANCE

To demonstrate the performance gains of our modified Raft algorithm, we performed an initial test using the original Raft setup as a benchmark to clearly underline the enhancements. This involved deploying 100 Java client threads, each dispatching 100 write requests to the Leader, accumulating to a comprehensive 10,000 communication exchanges aimed at evaluating throughput and latency. The results of these tests are visually documented in Figures 14 and 15.

A notable decline in throughput was observed as the server count increased from 5 to 10 to 15, highlighting a significant variance in performance. Furthermore, latency exhibited a nearly linear relationship with the increase in the number of servers in Figure 15. For instance, operations with a single Raft server interfacing with a key value data store yielded

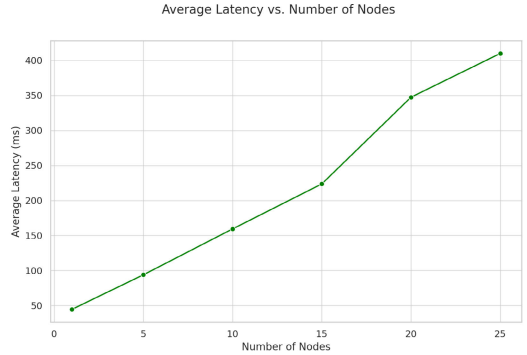

FIGURE 15. Average throughput vs number of nodes.

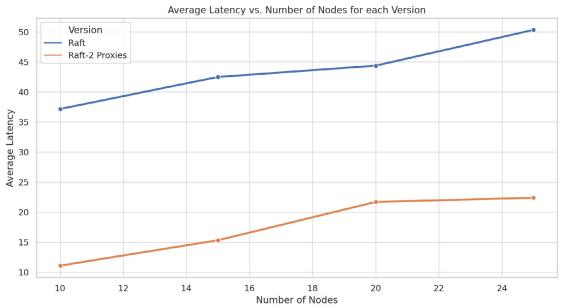

FIGURE 16. Comparison of original raft with modified raft with 2 proxy leaders.

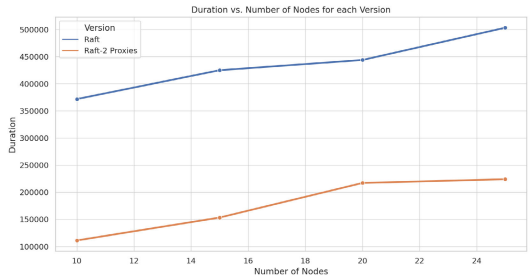

FIGURE 17. Comparison of original raft vs modified raft with duration (ms) with the number of servers.

TABLE 2. Comparison of average latencies between original raft and raft with 2 proxy leaders up to 25 nodes.

No. of Nodes	Raft (avg. latency) – ms	RaftOptima - 2 Proxy Leaders (avg. latency) - ms
10	37.1849	11.1183
15	42.4888	15.3293
20	44.3753	21.7084
25	50.3276	22.3985

TABLE 3. Comparison table of avg. throughput and avg. latencies for Raft with varying number of proxy leaders.

Version	Avg. Throughput	Avg. latency (ms)
Raft	19.86	50.327
RaftOptima – 2 Proxy Leaders	44.74	22.347
RaftOptima – 3 Proxy Leaders	36.58	27.333
RaftOptima – 4 Proxy Leaders	27.02	36.998

an average latency below 50ms, which markedly rose to over 400 ms when the server tally reached 25.

For a more focused analysis and to attain precise comparative insights between the original and improved Raft algorithms, we scaled down to a single client issuing 10,000 commands, varying the server count from 5 to 25 across both setups. This approach revealed a substantial difference in latency, as depicted in Figure 10, underscoring the enhanced efficiency of our modified RaftOptima design.

Integrating two proxy leaders into the Raft cluster and performing multiple simulation runs revealed significant performance enhancements, particularly in latency reduction, as demonstrated in Figures 16 and 17. These graphs compare the performance of the original Raft algorithm across a range of server configurations (5 to 25) and illustrate the performance degradation, notably in latency, with an increasing number of servers. Inclusion of just two proxy leaders results in a noticeable improvement in latency and overall duration, a benefit that could be amplified by adding more proxy leaders as the cluster size expands.

This improvement becomes significantly more evident in clusters comprising more than 10 servers. As the follower count rises, the leader's responsibility to broadcast to all followers and wait for their responses until a majority commit is achieved becomes increasingly burdensome. This process, coupled with the leader's obligation to continually accept and queue incoming client requests in serial order, contributes to escalating latency within the Raft algorithm. Proxy leaders alleviate the leader's broadcasting and response collection duties, allowing the leader to focus on log maintenance for the proxies and other critical tasks such as client request handling.

Our analysis suggests that incorporating proxy leaders into smaller clusters, with fewer than 10 servers, may not yield substantial performance gains, given the added complexity and negligible latency improvement. For instance, in an 8-node Raft cluster with one leader and seven followers, introducing two proxy leaders redistributes the followers, necessitating additional communication rounds that could inadvertently increase latency. Therefore, we conclude that clusters with fewer than 10 nodes are less suitable for a proxy leader-enhanced Raft (RaftOptima) configuration due to the disproportionate trade-off between added complexity and performance benefits.

n clusters comprising 10, 15, 20, and 25 servers, our enhanced Raft design with proxy leaders demonstrated average latency improvements of 70%, 63%, 55%, and 51%, respectively. These data reveal that while the latency of the original Raft algorithm significantly escalates with each addition of 5 servers, the modified Raft with proxy leaders exhibits a more subdued, nonlinear growth rate, as can be inferred from the table 2. The marked latency reduction in our modified algorithm mainly results from the disproportionately higher latency increases observed in the original Raft setup.

Further experimentation with an increased range of proxy leaders (2 to 4) in a 25-server setup, along with a latency benchmark against the unmodified Raft, is illustrated in Figure 18. It emerges that a configuration with two proxy leaders optimizes the latency reduction, halving the average

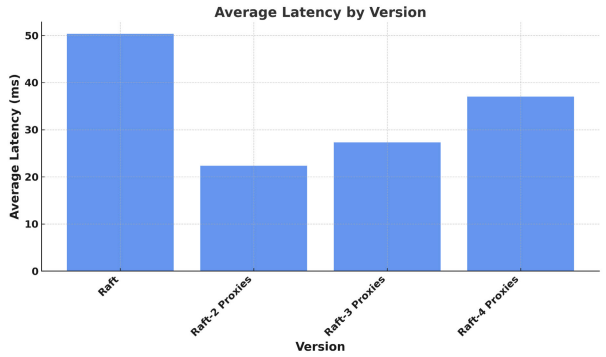

FIGURE 18. Avg. latency vs raft with varying number of proxy servers.

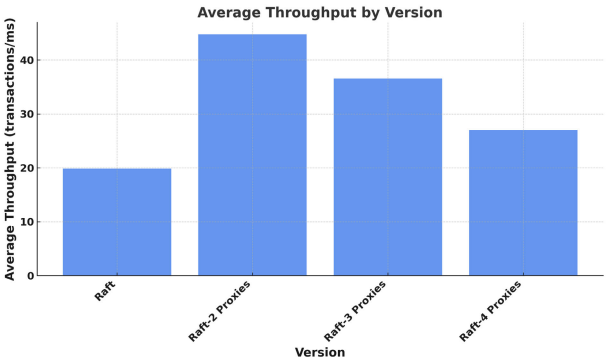

FIGURE 19. Avg. throughput vs raft protocol with varying number of proxy leaders.

latency compared to the standard Raft model. However, the introduction of more than two proxy leaders leads to a marginal increase in latency, likely due to the compounded communication overhead of the added proxy leaders. The average latencies are shown in table 3.Despite this, latency figures with up to 4 proxy leaders remain superior to those of the original Raft algorithm.

Furthermore, the deployment of two proxy leaders appears to nearly double the average throughput, underscoring the efficiency of this configuration. However, as the number of proxy leaders increases beyond two, a decrease in throughput is observed, as can be inferred from the figure 19 and table 3. This trend suggests that while the introduction of proxy leaders significantly improves both latency and throughput, there exists an optimal number of proxy leaders beyond which the benefits begin to diminish, highlighting the importance of balancing proxy leader count to maximize cluster performance.

VII. CONCLUSION

This study tackles the performance and scalability challenges inherent to the Raft consensus algorithm, especially in scenarios of (1) partial network partitions and (2) increasing cluster sizes. Raft's design, while straightforward and robust under stable network conditions, encounters liveness issues when network links among cluster nodes are unreliable or partially partitioned, leading to unreachable nodes.

Our approach involves excluding nodes affected by partial network partitions from election processes until full connectivity is restored and the node is rebooted. Although not without its weaknesses, this method avoids the complexities and potential errors associated with significant modifications to the Raft protocol. The effectiveness of our modifications was validated through comprehensive simulations using a custom-built discrete event simulator (DES), and the correctness of our modifications was verified with TLA + model verification.

By incorporating proxy leaders to manage log replication broadcasting commands, we observed a notable enhancement in the algorithm's performance across clusters ranging from 5 to 25 servers. Our results show a significant reduction in average latency, up to 60% - highlighting the efficacy of proxy leaders in reducing communication overhead and improving throughput. This strategy is particularly effective in larger clusters, where it prevents the linear increase in latency that typically accompany cluster expansion.

The introduction of proxy leaders addresses the scalability challenge head-on, ensuring that the Raft algorithm remains scalable and fault-tolerant even as the cluster size grows. This modification is crucial for distributed databases that are increasingly deployed closer to client locations to optimize performance. Our findings demonstrate that the strategic addition of proxy leaders can maintain low latency levels, thus enhancing the overall scalability of database systems without a linear increase in latency as the number of replicas grows. Thus, our study presents a scalable and fault-tolerant solution to the challenges faced by the Raft consensus algorithm, offering significant improvements for distributed systems' efficiency and reliability. The study can be further extended by implementing a full distributed KV-data-store and test for performance and fault-tolerance. This can also be tested with a varying number of follower nodes.

REFERENCES

[1] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.
[2] L. Lamport, "Paxos made simple," in *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*. New York, NY, USA: ACM, Dec. 2001, pp. 51–58. [Online]. Available: https://dl.acm.org/doi/10.1145/568425.568433
[3] (2024). *A Byzantine Failure in the Real World*. Accessed: Jan. 31, 2024. [Online]. Available: https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/
[4] (2024). *ETCD*. Accessed: Jan. 31, 2024. [Online]. Available: https://etcd.io/
[5] D. Thinkers. (2024). *Raft Does Not Guarantee Liveness in the Face of Network Faults*. Accessed: Jan. 31, 2024. [Online]. Available: https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/
[6] C. Jensen, H. Howard, and R. Mortier, "Examining raft's behaviour during partial network failures," in *Proc. 1st Workshop High Availability Observability Cloud Syst.*, Apr. 2021, pp. 11–17, doi: 10.1145/3447851.3458739.
[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: ACM, 2019, pp. 203–226. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3335772.3335936?casa_token=bMLrIotBC6EAAAAA%3A8tajD-eWY7vR_CJfGqrb0KKBLZro4pf5KQNhecVkqNpK1XhnHB97GSYjJPuW4lwQHPHY6Srbp-mUbck
[8] M. N. M. Bhutta, A. A. Khwaja, A. Nadeem, H. F. Ahmad, M. K. Khan, M. A. Hanif, H. Song, M. Alshamari, and Y. Cao, "A survey on blockchain technology: Evolution, architecture and security," *IEEE Access*, vol. 9, pp. 61048–61073, 2021.

[9] Z. Bao, Q. Wang, W. Shi, L. Wang, H. Lei, and B. Chen, "When blockchain meets SGX: An overview, challenges, and open issues," *IEEE Access*, vol. 8, pp. 170404–170420, 2020.
[10] N. Afraz, F. Wilhelmi, H. Ahmadi, and M. Ruffini, "Blockchain and smart contracts for telecommunications: Requirements vs. Cost analysis," *IEEE Access*, vol. 11, pp. 95653–95666, 2023.
[11] M. Iqbal and R. Matulevicius, "Exploring Sybil and double-spending risks in blockchain systems," *IEEE Access*, vol. 9, pp. 76153–76177, 2021.
[12] H.-Y. Paik, X. Xu, H. M. N. D. Bandara, S. U. Lee, and S. K. Lo, "Analysis of data management in blockchain-based systems: From architecture to governance," *IEEE Access*, vol. 7, pp. 186091–186107, 2019.
[13] A. Hafid, A. S. Hafid, and M. Samih, "Scaling blockchains: A comprehensive survey," *IEEE Access*, vol. 8, pp. 125244–125262, 2020.
[14] D. Huang, X. Ma, and S. Zhang, "Performance analysis of the raft consensus algorithm for private blockchains," *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. 50, no. 1, pp. 172–181, Jan. 2020.
[15] R. Wang, L. Zhang, Q. Xu, and H. Zhou, "K-bucket based raft-like consensus algorithm for permissioned blockchain," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2019, pp. 996–999.
[16] W. Fu, X. Wei, and S. Tong, "An improved blockchain consensus algorithm based on raft," *Arabian J. Sci. Eng.*, vol. 46, no. 9, pp. 8137–8149, Sep. 2021.
[17] D. Kim, I. Doh, and K. Chae, "Improved raft algorithm exploiting federated learning for private blockchain performance enhancement," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2021, pp. 828–832.
[18] X. Xu, L. Hou, Y. Li, and Y. Geng, "Weighted RAFT: An improved blockchain consensus mechanism for Internet of Things application," in *Proc. 7th Int. Conf. Comput. Commun. (ICCC)*, Dec. 2021, pp. 1520–1525.
[19] S. Tian, Y. Liu, Y. Zhang, and Y. Zhao, "A Byzantine fault-tolerant raft algorithm combined with Schnorr signature," in *Proc. 15th Int. Conf. Ubiquitous Inf. Manage. Commun. (IMCOM)*, Jan. 2021, pp. 1–5.
[20] Y. Wang, S. Li, L. Xu, and L. Xu, "Improved raft consensus algorithm in high real-time and highly adversarial environment," in *Proc. Int. Conf. Web Inf. Syst. Appl.*, Kaifeng, China. Cham, Switzerland: Springer, Sep. 2021, pp. 718–726.
[21] M. Zhang and L. Xu, "MLRaft: Improvement of raft based on multi-log synchronization model," in *Proc. 6th Int. Conf. Electron. Inf. Technol. Comput. Eng.* New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 1050–1055, doi: 10.1145/3573428.3573617.
[22] J.-F. Pâris and D. D. E. Long, "Pirogue, a lighter dynamic version of the raft distributed consensus algorithm," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2015, pp. 1–8.
[23] M. Fazlali, S. M. Eftekhar, M. M. Dehshibi, H. T. Malazi, and M. Nosrati, "Raft consensus algorithm: An effective substitute for Paxos in high throughput P2P-based systems," 2019, *arXiv:1911.01231*.
[24] T. Nakagawa and N. Hayashibara, "Energy efficient raft consensus algorithm," in *Proc. Int. Conf. Netw.-Based Inf. Syst.* Cham, Switzerland: Springer, 2017, pp. 719–727.
[25] J. Xu, W. Wang, Y. Zeng, Z. Yan, and H. Li, "Raft-PLUS: Improving raft by multi-policy based leader election with unprejudiced sorting," *Symmetry*, vol. 14, no. 6, p. 1122, May 2022.
[26] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft refloated: Do we have consensus?" *ACM SIGOPS Operating Syst. Rev.*, vol. 49, no. 1, pp. 12–21, Jan. 2015.
[27] H. Howard. *GitHub Heidihoward/OCAML-Raft: Implementation of Raft Consensus Algorithm—Github.Com*. Accessed: Feb. 1, 2024. [Online]. Available: https://github.com/heidihoward/ocaml-raft
[28] D. Thinkers. *Raft Does not Guarantee Liveness in the face of Network Faults—Decentralizedthoughts.Github.Io*. Accessed: Feb. 1, 2024. [Online]. Available: https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/
[29] X. Liu, Z. Huang, Q. Wang, and N. Luo, "An optimized key-value raft algorithm for satisfying linearizable consistency," in *Proc. Int. Conf. Netw. Netw. Appl. (NaNA)*, Dec. 2022, pp. 522–527.
[30] N. Du, D. Yan, Z. Liang, Y. Huang, T. Liu, and C. Zhong, "Research on optimization of efficient and scalable mutli-raft consensus algorithm for supply chain finance," in *Proc. IEEE Int. Conf. Sens., Diag., Prognostics, Control (SDPC)*, Aug. 2022, pp. 342–347.
[31] H. Li, Z. Liu, and Y. Li, "An improved raft consensus algorithm based on asynchronous batch processing," in *Proc. Int. Conf. Wireless Commun., Netw. Appl.* Cham, Switzerland: Springer, 2021, pp. 426–436.
[32] X. Wu, C. Wang, and Z. Liu, "Raft consensus algorithm based on reputation mechanism," in *Proc. Int. Conf. Comput. Netw. Secur. Softw. Eng. (CNSSE)*, Oct. 2022, pp. 272–281.
[33] Y. Wang, Z. Wang, Y. Chai, and X. Wang, "Rethink the linearizability constraints of raft for distributed key-value stores," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 1877–1882.
[34] Y. Wang and Y. Chai, "vRaft: Accelerating the distributed consensus under virtualized environments," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, Taipei, Taiwan. Cham, Switzerland: Springer, Apr. 2021, pp. 53–70.
[35] D. Ongaro, *Consensus: Bridging Theory and Practice*. Stanford, CA, USA: Stanford Univ., 2014.
[36] H. Howard, "ARC: Analysis of raft consensus," Dept. Comput. Sci. Technol., Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-857, Jul. 2014, doi: 10.48456/tr-857. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf
[37] (Nov. 2020). *A Byzantine Failure in the Real World*. [Online]. Available: https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world
[38] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Correct Hardware Design and Verification Methods* (Lecture Notes in Computer Science), L. Pierre and T. Kropf, Eds., Berlin, Germany: Springer, 1999, pp. 54–66.
[39] *Welcome! | The Coq Proof Assistant—Coq.Inria.fr*. Accessed: Feb. 1, 2024. [Online]. Available: https://coq.inria.fr/
[40] *The TLA+ Home Page—Lamport.Azurewebsites.Net*. Accessed: Feb. 1, 2024. [Online]. Available: https://lamport.azurewebsites.net/tla/tla.html
[41] K. K. Kondru and S. Rajiakodi, "Improving scalability of permissioned blockchains by making raft orderer to understand the underlying network topology," in *Proc. 1st Int. Conf. Artif. Intell., Commun. (EAI)*, Mar. 2024, doi: 10.4108/eai.23-11-2023.2343202.
[42] R. Yadav and A. Rahut. (2023). *FlexiRaft: Flexible Quorums With Raft*. [Online]. Available: https://www.cidrdb.org/cidr2023/papers/p83-yadav.pdf
[43] M. Whittaker, "Compartmentalizing state machine replication," Univ. California, Berkeley, CA, USA, 2021. [Online]. Available: https://escholarship.org/uc/item/7h89264s
[44] D. Yang, I. Doh, and K. Chae, "Cell based raft algorithm for optimized consensus process on blockchain in smart data market," *IEEE Access*, vol. 10, pp. 85199–85212, 2022.
[45] *GitHub Heidihoward/Leaderelection-Tlaplus: An Example TLA+ Specification for a Simple 'Raft Style' Github.Com*. Accessed: Feb. 1, 2024. [Online]. Available: https://github.com/heidihoward/leaderelection-tlaplus

KIRAN KUMAR KONDRU (Member, IEEE) is currently pursuing the Ph.D. degree in computer science with the Central University of Tamil Nadu, Thiruvarur, India. His research interest includes improving distributed consensus protocols, especially raft consensus. His research interests also include modern byzantine consensus algorithm and blockchain technologies. He also focuses on the latest research developments in distributed consensus.

SARANYA RAJIAKODI received the B.E. and M.E. degrees (Hons.) in CSE program from Anna University, Chennai, and the Ph.D. degree in the domain of requirement engineering from Madurai Kamaraj University, in 2015. She has been an Assistant Professor with the Department of Computer Science, Central University of Tamil Nadu, Thiruvarur, since 2016. She has a diverse range of research interests in the field of computer science. She has accumulated 12 years of academic experience. Her scholars are pursuing their research in the area of RAFT consensus algorithm, multimodal abusive/offensive data classification, and blockchain in healthcare under her guidance. She has published papers in highly reputed journal papers and multiple international conferences. Her expertise spans several domains, including social media data analytics, content moderation, data science, the IoT, and distributed networks and systems.