

Design and Implementation of a Strongly Consistent Distributed Key–Value Store (PyazDB)

Subinoy Biswas¹, Subhadip Saha², Spandan Chakrabarty³, Souvik Sen Laskar⁴, Sanjukta Chatterjee⁵, Pijush Kanti Ghosh⁶

¹Computer Science and Engineering & JIS College of Engineering

²Computer Science and Engineering & JIS College of Engineering

³Computer Science and Engineering & JIS College of Engineering

⁴Computer Science and Engineering & JIS College of Engineering

⁵Computer Science and Engineering & JIS College of Engineering

⁶Computer Science and Engineering & JIS College of Engineering

Abstract - Modern distributed applications require systems that scale across multiple nodes while maintaining correctness during failures. Many distributed databases prioritize performance and availability by adopting weak or eventual consistency models, often leading to stale reads and inconsistent states. This paper presents the design and implementation of a strongly consistent distributed key-value store based on the Raft consensus algorithm. The system follows a leader-based replication model in which write operations are committed only after majority agreement. Implemented in Go with modular consensus, storage, and communication layers, the system guarantees linearizable behavior and prioritizes correctness over availability during failures.

Key Words: Distributed systems, strong consistency, Raft consensus algorithm, replicated state machine, distributed key-value store

1. INTRODUCTION

The explosive expansion of internet-scale applications has contributed to a fundamental change in the computing models characterized by centrally allocated computing models, to distributed computing systems. Modern services are expected to handle large data volumes, be able to support high concurrency and to remain functional even when network partitions / hardware failures occur. To satisfy these needs information is distributed across a number of machines which collaborate to present a single system to clients which is larger in capacity to scale and fault tolerant but has coordination and reliability problems due to distribution across many computers [5],[7].

Maintaining a coherent perspective of the information on a number of replicas is a significant issue of distributed systems. Even though increasing resilience by replication can help, network latencies, single-host failure, and simultaneous client connections can lead to multiple replicas seeing or implementing updates in new orders. Unless it is adhered to; such kind of behavior may cause conflicting writes, stale reads or lost updates [1],[11].

The consistency models give the guarantees of a distributed system concerning the ordering and visibility of operations. Some applications can withstand the temporal inconsistency between replicas that eventual consistency, such as weak

consistency, allow to achieve higher availability and performance. Correctness-critical systems such as configuration management, coordination services, and metadata storage must have strong consistency guarantees to prevent making poor decisions or failures in the system, nevertheless, which is not always guaranteed according to the current study [1][6].

The strong consistency in a distributed environment must be accomplished through coordination mechanisms, which are most often executed by distributed consensus algorithms. To ensure consistency between replicas, consensus ensures that all nodes that are not faulty agree on one sequence of actions that is well defined. This project is based on the Raft protocol to implement a strongly consistent distributed key-value store that applies a consensus-based approach. The system has a leader-based design giving them data integrity and predictable and fault-tolerant behavior which promotes correctness more than availability in the event of failures [1],[11],[12].

Based on these, the development of such systems as etcd points to an emphasis on fine-tuning of particular bottlenecks. To illustrate, K.~Murugan (2024) proves that decoupling of the logic of acknowledgment via WriteAheadLogging (WAL) can significantly decrease the write latency between 30ms to less than 4ms, rendering strongly consistent system an option in high-concurrent systems [14].

At the same time, the difficulty of implementing such systems has resulted in the development of architectural modularity; Alves et al. (2025) proposed Composing State Machine Replication (CSMR) that implies the use of YAML-based configurations to enable developers to build complex and reliable microservices in a modular manner [15].

Lastly, consensus has been applied in security areas, including the Decentralized Heterogeneous Redundancy (DHR) suggested by Chen and Shi (2026). Through the Raft clusters to control dynamically changing logs, this solution eradicates single points of failure within the IoT control planes, and decreases the attack surface by 70 percent of it, on average, in comparison to the traditional approach without the cluster development [13].

2. LITERATURE SURVEY

The necessity to manage growing data sets with requirements of fault-tolerance, scalability and reliability has stimulated the study of distributed data storage systems over a number of

decades. The principal themes of early research on distributed databases were data partitioning, replication techniques, and distributed query processing. Replication was mostly done to enhance fault tolerance and availability, but it also entailed the challenge of maintaining consistency between multiple data copies. Ensuring that replicas reliably and consistently reflect the system state constitutes one of the primary problems in distributed systems research, which is an important concern in this field of study [5],[7].

To deal with this challenge, numerous consistency models have been proposed. Strong consistency models such as linearizability and sequential consistency provide a intuitive set of guarantees by ensuring that the operations appear to run out of order in a consistent manner across the world. The models though provide a lot of simplification in correctness reasoning and rationale in the design of applications typically require coordination mechanisms that increase latency, and reduce availability in case of failure. Relative consistency, conversely, enable replicas to temporarily separate but ultimately converge to a consistent state by relaxing ordering requirements to improve availability and performance by which weaker consistency models allow replicas to temporarily separate before ultimately converging to a consistent state by weakening ordering guarantees to improve both availability and performance [8].

The eventual consistency gained increased popularity with the increase of the large scale web services where the emphasis was on the availability and the low latency. Influenced systems use systems with replicated data utilizing methods such as the background synchronization, the use of the vectors clock and quorum-based reads and writes. These systems will offload the conflict resolving burden to the application layer, although they perform well when heavily loaded or when they fail partially. They are thus not suitable when there is need to be accurate in what is happening in the applications on which outdated or conflicting information can cause serious system failure [6].

CAP theorem was formulated as a formalization of the inherent trade-offs of distributed system design by asserting that a system may possess at most three of the following properties in case of network separation: consistency, availability and partition tolerance. Network partitions are bound to occur in real-world scenarios, and the system designers must make an appropriate trade-off between availability and consistency. Whereas availability-oriented systems can sustain requests at the cost of sending erratic information, consistency-oriented systems can refuse operations whenever they cannot ensure they are correct at a failure [3].

Many systems utilize distributed consensus algorithms in order to ensure strong consistency even in the face of failures. Consensus ensures that a group of nodes will concur on a single value or a sequence of actions even in the case of failure of some of the nodes. Even though the initial algorithms such as Paxos demonstrated that consensus could be made, even in asynchronous environments they were still generally believed to be difficult to understand and well implement [4],[11]. Later research introduced more structured and intelligible methodologies, which made the practical implementation of

consensus-based replication in real systems possible offer the results of research on the topic [1],[2].

2.1 Review of Selected Research Papers

Consensus algorithms have been investigated in a number of recent studies as a means of achieving high consistency in distributed storage systems. The Raft consensus algorithm was developed by Ongaro and Ousterhout specifically to increase understandability while maintaining the security guarantees of previous protocols. Their research showed that leader-based log replication and distinct concern separation greatly ease implementation and correctness reasoning [1]. Many contemporary strongly consistent systems are conceptually based on this work [11].

The application of Raft to real-world storage and coordination services has been the main focus of later research. Majority-based commitment guarantees linearizable behavior even in the event of node failures, according to studies examining Raft-based replicated state machines in distributed key-value stores [2]. The authors emphasized that although leader-based designs can result in write bottlenecks, they significantly streamline failure recovery and consistency management.

Strong consistency models and eventual consistency approaches in large-scale distributed systems were compared in another line of work. According to these studies, availability-focused systems frequently expose applications to conflicting updates and stale reads, which pushes complexity to higher layers [3]. Strongly consistent systems, on the other hand, enforce a global operation order, reducing application-level complexity.

Furthermore, studies on fault-tolerant storage systems have shown that quorum-based decision-making and replicated logs are effective ways to achieve crash-fault tolerance [6]. Such systems guarantee that committed data is maintained even in the event of failures by requiring consensus from a majority of nodes prior to committing updates [10].

Optimizations to Raft-based systems, such as log compaction, snapshotting, and enhanced read handling strategies, have been investigated in more recent research [12]. Without compromising consistency guarantees, these improvements seek to increase performance and lower storage overhead [14].

Comparisons of consensus algorithms have repeatedly highlighted that Raft's main contribution is not superior performance but rather its simplicity and ease of correct implementation [6]. Because of this feature, Raft has been widely used in both industrial systems and academic prototypes.

Several studies have emphasized that routing all write operations through a single leader enforces a total order on updates, which is crucial for preserving linearizable semantics [7]. Although this design may limit write scalability, it is justified for systems where correctness and predictability are more important than raw throughput.

The replicated state machine abstraction has also been widely studied as a foundation for building fault-tolerant distributed

services. By replaying the replicated log deterministically, all replicas can reach the same state even after failures [8].

Recovery mechanisms such as log compaction and snapshotting have been shown to reduce storage overhead and speed up node recovery without compromising safety guarantees [9]. These techniques allow long-running systems to remain efficient while preserving correctness. Comparative analyses between CP and AP systems further demonstrate that availability-focused designs risk exposing applications to stale or conflicting data. Strongly consistent systems avoid these issues by rejecting operations when consistency cannot be guaranteed [13].

Recent deployments of Raft-based systems in coordination and metadata management platforms confirm that Raft's explicit safety rules and leader-based structure reduce

implementation errors and improve system reliability [11]. One study from the reviewed literature is particularly relevant to the present project. It presents a Raft-based distributed key-value store emphasizing correctness, fault tolerance, and architectural clarity [15]. Its design philosophy closely aligns with the objectives of this work, making it an appropriate base reference for comparison.

2.2 Comparative Summary of Related Work

A comparative analysis of the selected studies is provided in Table 1. The table highlights each work's application domain, methodology, key findings, and suggested future directions. This structured comparison reveals common design patterns as well as differences in approaches to scalability, consistency, and fault tolerance.

Table -1: Comparative Analysis of Related Work

Author(s) & Year	Domain	Methodology	Key Findings	Future Scope
Sharov et al. (2015)	Cloud Storage	Workload-driven optimization for replica placement.	Reduced average operation latency by up to 94%.	Multi-objective optimization (cost vs. speed).
Rahul Batra (2017)	Distributed Systems	Simulation-based comparison of Paxos and Raft.	Raft performs better under high load; Paxos is harder to implement.	Real-world network jitter testing.
D. Vishwanath (2018)	Software Testing	Failure injection on open-source Raft libraries.	Identified data inconsistencies in 2 out of 3 major libraries.	Automated test suites for all consensus types.
Howard & Mortier (2020)	Consensus Theory	Qualitative and structural analysis of Paxos vs. Raft.	Raft's simplicity is primarily a result of pedagogical design.	Re-engineering Paxos for better clarity.
Lewchenko & Kaki (2023)	Formal Verification	Replicated state model (non-message passing).	Simplified inductive invariants; Raft verified without history.	Verification of multi-leader protocols.
P. S. Almeida (2024)	Theoretical CS	Axiomatic framework defining the CAL trilemma.	Visibility cycles are the root of most consistency issues.	Application to specialized CRDTs.
N. S. Pasupuleti (2024)	Distributed Databases	Benchmarking of Linearizable vs. Serializable reads.	Serializable reads cut latency significantly for non-critical data.	Dynamic consistency switching based on load.
K. Murugan (2024)	Performance Eng.	Decoupling ACK logic using Write-Ahead Logging (WAL).	Achieved sub-4ms commit latency in high-concurrency etc.	Hybrid replication for edge computing.
Jialuo Hu (2024)	Gaming / Networking	Go-based framework combining Raft core with Pebble DB.	Scaled to 6,000+ commands/sec with node recovery.	Snapshotting and encrypted transport.
Mastromauro et al. (2025)	Cybersecurity	Systematic survey and taxonomy of consensus attacks.	Detailed 12+ attack vectors including leader poisoning.	Standardization of Byzantine defenses.
A. Scrivano (2025)	Database Systems	Review of Paxos, Raft, and BFT mechanisms.	Elucidated trade-offs between performance and consistency.	Energy-efficient green consensus.
Shah & Hazarika (2025)	Performance Opt.	Empirical analysis of ML-optimized caching patterns.	ML-based caching outshines traditional LRU/LFU metrics.	Predictive caching for serverless.
Afifi et al. (2025)	Security Research	Simulated replay and forgery attacks on Raft.	Standard Raft is highly vulnerable to message re-introduction.	Cryptographic hardening of message-passing.

Alves et al. (2025)	Cloud Architecture	Modular Composing SMR (CSMR) via YAML/RPC.	Enabled reusable building of reliable microservices.	Global multi-cloud service compositions.
Chen & Shi (2026)	IoT / Security	Raft-based Dynamic Heterogeneous Redundancy.	Eliminated single points of failure; 70% less attack surface.	Lowering consensus latency for edge devices.

2.3 Base Paper Identification

Among the research studies reviewed one of the books is reference [12] that is the foundation of this research, and that is correctness, fault tolerance, and simplicity of architecture. These two aspects of its design were selected with the specific purpose of its leader-centric approach of log replication and its majority-based approach of commits. Since the bases paper is premised on the feasibility, this paper is an expansion of those thoughts based on systematic implementation with highlight on understandability and practical implementations. To guarantee performance constraints among leaders, the system uses the most up to date optimizations constructed by K. Murugan (2024), dependencies on the streamlined write-ahead log (WAL) handling to lower latency on commitments in the system [14]. Mathematical types This raft-like rush of toffeeping endorses the didactic clearness of Raft as much as the have-through necessities of the present itinerary distributed databases.

The architecture design of PyazDB is mathematically successful and follows the modern trends of modularity, in which the consensus engine and storage back-end is decoupled to modules. According to the Composing State Machine Replication (CSMR) model put forward by Alves et al. (2025), this isolation of concerns [15] enhances long-term maintainability of a system as well as ensuring the core Raft safety guarantees will be maintained by the storage-specific optimizations. Furthermore, the system addresses the threat environment that keeps evolving considering the strict safety invariants and log matching property established in research of DHR architecture by Chen Shi (2026). By these, the system provides a high level of resistance to state divergence and unauthorised/illegallog injections into the logstore thus making key-value store emerge as a powerful tool which can be used in pursuit of the complex environment of a network.

3. RESEARCH METHODOLOGY

3.1 System Overview

The proposed system would be a highly consistent distributed key-value store which uses a consensus based replication protocol. The main aim of the system is to make sure that all the involved nodes monitor and implement changes in a similar sequence even when a node fails or there is a network interference. In order to accomplish this, the system implements a consuler-based architecture that runs using the Raft consensus algorithm.

The system is made up of a fixed amount of nodes that are arranged in a cluster. Only one node is the leader as compared to the other nodes, which perform the role of followers at any point in time. Any written request by the clients is channeled to the leader who manages the process of replication and consistency. The leader can also provide read operations to ensure that clients can always view the latest state of the system that had been committed. This design alternative

offers the linearizable semantics and ease of maintaining the consistency.

Information is formatted as key-value pairs each update is represented as a command instead of a direct change of the data store. These instructions are added to a copied log which is used by the Raft protocol. individual nodes use committed log entries to update their own local state machine in such a way that exactly the same states are traversed by all the replicas sequentially. The system acts like one logical database to the client.

The system focuses on rightfulness rather than accessibility in case of failure. In case the cluster can not build a majority quorum the system deliberately fails read and write requests rather than endangering inconsistent behavior. This design is consistent with the needs of the applications that are correctness-critical, with a failure to deliver correct data deemed more damaging than unavailable human. Such a design is in line with the needs of the applications considered correctness-critical, where failures to provide correct data are deemed to be more detrimental compared to unavailability.

3.2 System Architecture

The high-level design of the proposed system has four main components which are the client interface, the Raft consensus module, the replicated log and the state machine. Clients get access to the system by way of a minimal API that resonates with the basic key-value functions. These requests are sent to the then leader that organizes all further actions.

The Raft consensus module is in charge of the election of leaders, replication of logs, and safety assurance. It also makes sure that there is only one leader in operation at any given time and that the log entries will be committed to after a subscription by the majority of nodes. The each replicated log is the official record of all those operations that have been implemented in the system whereas the state machine performs the commands that are committed to modify the key-value store.

This modular structure divides consensus logic and application logic thus enhancing clarity, maintainability and correctness. Because every component is associated with a specific responsibility, the system is more rational to argue about, to prove and to expand.

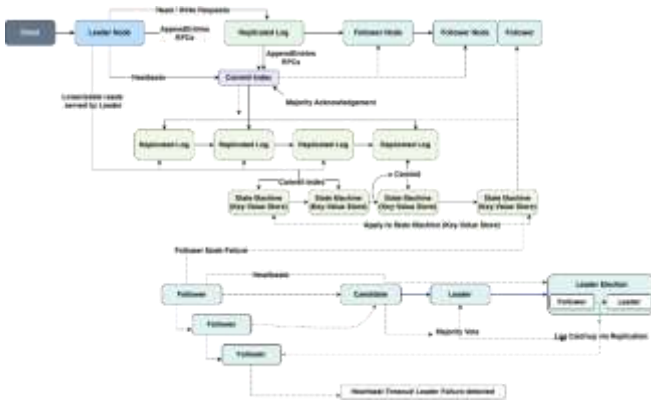


Fig-1: Workflow of the proposed Raft-based distributed key-value store

3.3 Workflow of the Proposed System

Raft-based distributed key-value store operational process (Fig-1) extensively covers the steps in a request processing, log replication, recovery, and failure handling. The leader coordinates all the interactions with clients. In the case of write operations, the leader writes about the request and pushes the request to the leader-followers using AppendEntries RPCs. This entry upon acknowledgements (ACKs) being received by a majority is committed and applied, in order on all the replicas, to achieve complete consistency.

The leader handles read operations, which ensures that its authority is maintained by maintaining linearizability using constant heartbeats. This ensures that the clients always get the latest dedicated data and avoids read staleness among the followers.

Majority-based is recovery, as well as elections. In case of failure of one of the followers, the cluster would keep operating provided a quorum exists and the recovered node restarts its log with the leader. On failure of the leader, his or her followers sense the absence of a heartbeat, alter their status to candidate state and choose a new leader. In the case of quorum loss, then the cluster fails temporarily so that safety can remain. The system ensures a high degree of consistency as well as failure resilience through the use of leader-based coordination and replicated state machines.

3.4 Implementation Details

The system is applied over the Go programming language which provides an effective concurrency management and lightweight communications primitives which can be used in the distributed systems. The go routines and channels of Go enable parallel tasks to be easily done e.g. client requests and replication processes and timeouts. Communication between nodes is done with the assistance of Remote Procedure Calls (RPCs), as a part of which enables to arrange the process of the exchanges of messages systematically and efficiently.

Its implementation is organized as a network of modules between them the Raft consensus, the networking, the replicated log, and the key-value state machine. Persistent storage is the storage that is employed in order to store the log entries and significant metadata so that the nodes can continue with their state after a crash. This modular design leads to an increase in maintainability and to the testability and separability of individual components and their extension.

The consensus algorithm that was chosen is Raft as a consensus algorithm since it is straight forward and very secure. To make the rationale process more basic in terms of how the system works, Raft divides the set of rules and consensus into leader election, log replication, and safety rules to reduce the risks of errors in the implementation. What characteristics of a distributed storage system render it to be suitable to an educational but yet practically applicable system?

4. RESULTS AND FINDINGS

To test against the proposed system, functional and failure-oriented tests were done in order to test the correctness, consistency, and fault-tolerant behavior of the system and took the maximum emphasis on foreseeable global running sequence rather than uncooked throughput. In the normal operation, the system was highly linearized in terms of writing by ensuring it only wrote after the majority of the nodes would acknowledge it. Reads based on leaders made sure that clients would see the last value that has been committed, and that the replication and commit mechanisms is reliable at ensuring that stale data cannot be used when doing a concurrent workload.

By testing the system with node failures, fault tolerance was found to be high and the system does not fail unless there is a majority of nodes on-line and safe rejoining of followers after a synchronization of the logs. A new cluster leader was also automatically chosen in the event of a leader failure with neither data loss nor any prejudice to the normal operations. In the event that a majority quorum could not be achieved, the system simply ignored all requests and this showed a willingness to focus on data integrity and correctness over availability in the uncertain circumstances.



Fig-2: Cluster initialization and container startup



Fig-3: PyazDB benchmark dashboard showing overall execution summary, including total PUT, GET, and DELETE operations (100% success rate), average latency, operation success distribution, and latency trends over time.



Fig-4: Detailed benchmark results of PyazDB illustrating throughput over time, latency distribution across operation types, and statistical latency metrics including minimum, maximum, P95, and P99 values.

5. DISCUSSION

Through experimentation, it has been shown that consensus-based replication is a useful method of generating high consistency in distributed key-value store. The proposed system assures accuracy and fault-tolerance of majority-agreement and leader-based log replication, which is consistent with the existing Raft-based frameworks. The system ensures there is only one authoritative state on all

functioning nodes by ensuring that only a quorum is required before a given update is finalized, which is a solid base in distributing data.

In contrast to most modern applications, which focus on raw performance and scalability, architectural simplicity and clarity are of primary importance in this system. It has a deliberately minimalist design in order to incorporate only the necessary Raft mechanisms, which is easier to analyze, understand, and prove. Such emphasis on simplicity makes the system specific (especially in academic and instructional and correctness-sensitive application where transparency is more desirable than volume of throughput).

It is very much modular in its architecture with complete segregation of consensus layer and key-value application layer. This explicit separation of replicating of logs, election of leaders and state machine execution enables replicating of logs, election of leaders, and execution of a state machine to be tested and confirmed individually. This can be achieved with such a systematic approach not only making it easier to implement but also making the subsequent extensions or maintenance less risky in terms of unintended side effects in the long-term stability of the system.

Reliability is also boosted by deterministic replicated state machine that guarantees that committed operations are performed identically on all the replicated machines. Such predictability is essential in debugging and validation of concurrent workloads because even in node crashes or network partition the behavior of the system is predictable. This is because this model of deterministic execution facilitates formal reasoning about the rightness of the system of execution giving stronger assurance of its overall integrity.

Lastly, the system deliberately makes its decisions take correctness over availability as operation will be rejected when a majority quorum is not available. The design decision will help avoid making stale or inconsistent data, which is a usual threat in availability-oriented distributed databases. The project is not supposed to compete with the high-throughput production databases, but it provides a solid reference implementation of strong consistency and can provide a clear and accurate model of coordination services and metadata management.

Table-2: Comparison Chart: Proposed System vs Existing Raft-Based Systems

Aspect	Existing Raft-Based Systems	Proposed System
Primary Focus	Performance, scalability, feasibility	Clarity, correctness, conceptual understanding
Architecture Complexity	More complex, system-optimized	Simplified and modular
Learning & Analysis	Harder to reason about	Easier to understand and verify
Failure Handling	Optimized for availability	Correctness prioritized over availability
Target Use Case	Production-scale systems	Academic, educational, correctness-critical systems
Design Philosophy	System-level optimization	Clean reference design

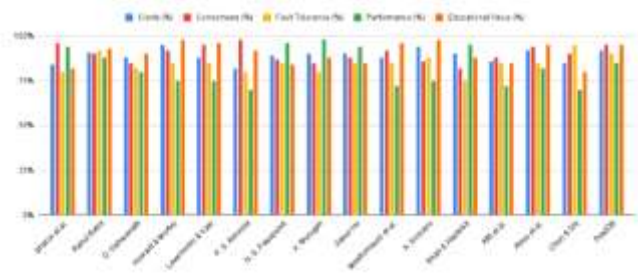


Fig-5: Comparative Study of Raft-Based Systems Using Design and Correctness Metrics



Fig-6: Line Graph Raft-Based Systems Using Design and Correctness Metrics Based on Recent Research Papers

6 CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

This paper has employed the Raft consensus algorithm to create and build a strongly consistent distributed key-value store. The primary goal of the work was to prove that consensus-based replication can be applied to correctness and fault tolerance of distributed storage systems. The proposed system provides the ability to ensure that all replicas take note of operations in a well-defined order by having the leader-based architecture and having majority agreement requirement before committing updates.

These findings demonstrate that the system is linearizable in normal operation conditions and it can deal with crash failures without loss to data integrity. The ability to recover automatically whenever there is a failure of any node in the network is made possible by leader election and log replication mechanisms, whereas intentional rejection of operations whenever there is quorum loss maintained safety. Moreover, the separation of consensus logic and application logic followed by its modular design is more compelling to clarity and reliability. All in all, this paper shows that Raft is a useful and comprehensible way of constructing strongly-consistent distributed systems.

6.2 Future Scope

Though the given system fulfills the main objectives of the design, it is possible to conduct a number of extensions in the future research. The introduction of dynamic cluster reconfiguration is one of the possible improvements that can be suggested; this feature will enable adding or removing nodes without interrupting the system operation.

The other enhancement may be the performance optimizations, e.g., follower-side read support, and quorum

validation to enhance read scalability, and a strong consistency guarantee.

REFERENCES

- [1] A. Sharov, A. Shraer, A. Merchant, and M. Stokely, "Take me to your leader! Online Optimization of Distributed Storage Configurations," Google, Inc. Technical Report, 2015.
- [2] R. Batra, "Implementation and Evaluation of Paxos and Raft Distributed Consensus Protocols," Master's report, The University of Texas at Austin, 2017.
- [3] D. Vishwanath, "Validating Key-value Based Implementations of the Raft Consensus Algorithm for Distributed Systems," Master's project, San Jose State University, 2018.
- [4] 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 (PaPoC '20), 2020.
- [5] N. V. Lewchenko and G. Kaki, "Distributed Consensus Algorithms as Replicated State Applications," in Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23), University of Colorado Boulder, 2023.
- [6] P. S. Almeida, "A Framework for Consistency Models in Distributed Systems: Time, Visibility, Order, Convergence, Basic Axioms, and a New Trilemma," arXiv preprint arXiv:2411.16355, 2024.
- [7] J. Hu, "Distributed Game Server based on RAFT Consensus Core," Technical Report, University of California, San Diego, 2024.
- [8] K. Murugan, "Performance Optimization for etcd using Write-Ahead Logging," Technical Report, 2024.
- [9] N. S. Pasupuleti, "Optimizing Read Performance in Distributed Key-Value Stores Using Serializable Consistency," International Journal of Innovative Research in Computer and Communication Technology (IJIRCT), vol. 10, no. 3, 2024.
- [10] T. Afifi, A. Hegazy, and E. Abousaif, "From Consensus to Chaos: A Vulnerability Assessment of the RAFT Algorithm," International Journal of Advanced Computer Science and Applications (IJACSA), vol. 16, no. 12, 2025.
- [11] C. M. Alves et al., "Composing State Machine Replication," Technical Paper, 2025.
- [12] L. Mastromauro, D. S. Andrade, M. O. Ozmen, and M. A. Kinsy, "Survey of Attacks and Defenses on Consensus Algorithms for Secure Data Replication in Distributed Systems," IEEE Access, vol. 13, pp. 143644–143667, 2025.
- [13] A. Scrivano, "Distributed Database Systems and Consensus Protocols: Understanding the Challenges and Solutions in Achieving Consistency and Reliability," Review Article, 2025.
- [14] M. Shah and A. V. Hazarika, "An In-Depth Analysis of Modern Caching Strategies in Distributed Systems: Implementation Patterns and Performance Implications," International Journal of Science and Engineering Applications, vol. 14, no. 1, pp. 09–13, 2025.
- [15] K. Chen and L. Shi, "Decentralized DHR Architecture for Network Security," Research Paper, 2026.