European Journal of Computer Science and Information Technology,13(14),184-197, 2025
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

Raft Consensus Algorithm: Simplicity and Robustness in Distributed Systems

Kuldeep Deshwal
Proofpoint Inc, USA

doi: https://doi.org/10.37745/ejcsit.2013/vol13n14184197 Published May 05, 2025

Citation: Deshwal K. (2025) Raft Consensus Algorithm: Simplicity and Robustness in Distributed Systems, *European Journal of Computer Science and Information Technology*,13(14),184-197

Abstract: *The Raft consensus algorithm provides a more understandable alternative to previous protocols like Paxos while maintaining strong consistency guarantees in distributed systems. By breaking consensus into three distinct components—leader election, log replication, and safety—Raft creates a clear mental model for developers. Its widespread adoption spans distributed databases, configuration management, container orchestration, microservices infrastructure, and blockchain systems. Despite inherent challenges, including leader bottlenecks and brief unavailability during leader changes, Raft offers significant benefits through its straightforward design. Current innovations address these limitations through performance optimizations, multi-Raft architectures, formal verification, edge computing adaptations, and educational tools, ensuring the algorithm's continued relevance as distributed computing evolves.*

Keywords: Algorithm, Consensus, Distributed, Fault-Tolerance, Replication

INTRODUCTION

Imagine a team of computers working together to provide a service, like showing you your bank balance or letting you order something online. These computers need to stay in perfect sync about what's happening and what to do next. This becomes challenging when some computers crash, internet connections fail, or messages between them get delayed. This challenge is at the heart of distributed systems - the technology that powers everything from cloud services to banking apps to social media platforms. Consensus algorithms provide the essential rules that help these computer teams reach an agreement despite all these potential problems. They act like referees, ensuring everyone follows the same playbook. Without these algorithms, chaos would ensue. Some computers might think your bank account has $500, while others show $400. An online store might process your order twice or not at all. Your social media posts might appear to some friends but not others. Think about when you and your friends try to decide where to eat dinner. If everyone can't communicate clearly, you might end up at different restaurants! Computer systems face the same coordination problem but with much higher stakes. Without consensus algorithms like Raft providing coordination in distributed systems, the same challenges that led Ongaro and Ousterhout to state that 'understandability is a key requirement' would continue to plague modern computing infrastructure[1]. Consensus algorithms provide the digital equivalent of a formal meeting procedure - they establish who gets to speak when, how decisions are recorded, and what happens if someone steps away. This structured approach ensures all computers in the system maintain the same understanding of what's true, even when things go wrong. This foundation of agreement enables the reliable digital services we depend on every day, from streaming movies to making payments to storing photos in the cloud. Without consensus algorithms working behind the scenes, our connected digital world simply couldn't function reliably.

The Incumbents and Their Limitations

Before Raft arrived on the scene, the world of distributed systems was dominated by an algorithm called Paxos. While Paxos had earned respect in academic circles for its mathematical elegance and theoretical soundness, it created enormous headaches for engineers trying to use it in real-world systems.

Paxos suffered from being extraordinarily difficult to understand. Even veteran software engineers with years of experience would struggle to grasp how it worked. Reading the Paxos papers felt like deciphering a foreign language for many developers. The concepts were abstract, the explanations were dense with mathematical notation, and the overall approach seemed disconnected from practical programming concerns.

Implementing Paxos correctly proved even more challenging. Engineers attempting to convert the theoretical description into working code encountered countless edge cases and ambiguities not addressed in the academic papers. Many organizations spent months trying to build reliable Paxos implementations, only to discover subtle bugs when their systems faced unexpected conditions in production environments.

The complexity of Paxos created teaching challenges as well. Computer science professors frequently avoided covering Paxos in depth during distributed systems courses because students found it so bewildering. When they did teach it, instructors often needed multiple lectures just to explain the basic protocol, with students still left confused about how to apply it practically.

Documentation problems compounded these issues. The original Paxos papers, while mathematically rigorous, left many implementation details unspecified. This forced development teams to make critical design decisions without clear guidance, leading to inconsistent implementations across the industry and difficulty in maintaining systems over time. Imagine Paxos as an intricate recipe written by a brilliant chef who assumes readers already understand advanced cooking techniques. The recipe might produce an amazing dish, but following it requires interpreting vague instructions, making educated guesses, and possibly failing several times before getting it right. Even professional chefs would struggle to execute it properly.

As distributed systems became increasingly important in the technology landscape—powering everything from cloud services to financial systems—the industry desperately needed a more accessible approach to consensus. Engineers needed an algorithm that was not only theoretically sound but also practical to implement and maintain in real-world conditions. This growing need would eventually lead to the development of Raft, an algorithm designed specifically to address these limitations. The complexity of Paxos implementation led to numerous inconsistent variants and interpretations across the industry, making it difficult for engineers to be confident in their distributed systems [2].

What is Raft?

Raft emerged as a consensus algorithm specifically designed to be understandable first without sacrificing reliability. Unlike previous approaches that prioritized mathematical elegance, Raft's creators focused on making something that real engineers could implement without confusion. Raft's core design philosophy breaks consensus into distinct subproblems of leader election, log replication, and safety, creating a more approachable mental model for developers [1].
The key insight behind Raft is breaking down the complex consensus problem into three clear, separate pieces that can be understood independently:

Leader Election

The system democratically chooses one computer to be in charge (like electing a team captain). When a Raft system starts up, all nodes begin as "followers." After a random timeout period (typically between 150-300 milliseconds), a follower who hasn't heard from a leader will become a "candidate" and request votes from other nodes. If it receives votes from the majority, it becomes the leader. This randomized approach prevents endless ties while ensuring a leader emerges quickly.

Replication

The leader keeps track of all changes and makes sure every node receives them. Each change (like updating a user's profile or adding an item to a shopping cart) becomes a "log entry" with a unique sequential number. The leader sends these entries to all followers, who add them to their own logs. Once the leader confirms that a majority of nodes have saved an entry, it's considered "committed" and can be applied to the system's state.

Safety

Rules that ensure the system never makes contradictory decisions, even during failures. Raft enforces strict ordering of log entries through "term numbers" that increase with each new leader election. If network problems cause temporary disagreement, Raft includes mechanisms to detect and resolve these conflicts. For example, if a follower's log differs from the leader's, the leader will identify exactly where they diverged and send the correct entries to bring the follower up to date.

This modular approach makes Raft much easier to understand. You can learn one piece at a time rather than having to grasp everything at once. Engineers can start by implementing leader election, then add log replication, and finally ensure safety properties—gradually building a complete consensus system. User studies demonstrated that participants could understand Raft consensus much more quickly than Paxos, with teams implementing correct Raft systems in significantly less time than comparable Paxos implementations[3]. The structured approach also helps during debugging. When something goes wrong, developers can focus on which of the three components is failing rather than getting lost in a complex web of interconnected processes. This clarity directly translates to more reliable distributed systems in the real world.

How Raft Improves on Previous Approaches

Raft introduces several key innovations that make it significantly more practical for real-world distributed systems. These improvements address the fundamental challenges that made previous consensus algorithms difficult to implement and maintain. Comparative performance analysis between Paxos and Raft demonstrated that Raft's simplicity comes with minimal performance overhead while providing substantial gains in implementation clarity [4].

Clear leadership

It forms the cornerstone of Raft's design philosophy. Unlike Paxos, where any node can propose changes at any time (creating complex coordination scenarios), Raft designates a single leader who coordinates all operations for a given term. This leader-follower model creates a clear flow of information: clients send all requests to the leader, the leader orders these requests into log entries, and followers simply replicate the leader's log. This straightforward hierarchy eliminates the need to resolve competing proposals from different nodes, dramatically simplifying the protocol's behavior. In technical terms, each Raft term has exactly one leader, and that leader is the only node that can append new entries to the log, creating a single source of truth.

Intuitive election process

It ensures the cluster can quickly recover from leader failures. Raft implements this through a randomized timeout mechanism. Each follower waits for a randomized period (typically 150-300ms) before initiating an election if it hasn't heard from the current leader. This randomization naturally prevents split votes where multiple candidates might tie. When becoming a candidate, a node increments the term number, votes for itself, and requests votes from other nodes. A candidate wins the election upon receiving votes from a majority of servers, becoming the leader for that term. This approach typically resolves leadership within 1-2 election timeouts, even after failures, minimizing system disruption.

Understandable log management

It provides clear rules for handling the replicated log across the cluster. The log in Raft is an ordered sequence of commands, with each entry containing a term number and command to be executed. The leader maintains consistency by forcing followers' logs to match their own through a simple mechanism: when sending new entries, the leader includes the index and term of the preceding entry. If followers can't find this entry in their logs, they reject the new entries, prompting the leader to send earlier entries until consistency is established. This approach guarantees logs will eventually converge without requiring complex reconciliation algorithms.

Simple membership changes

It allows Raft clusters to evolve over time without disrupting service. Adding or removing servers from the cluster is handled through a two-phase approach called "joint consensus." During this transition, the cluster operates under both the old and new configurations simultaneously, requiring agreement from majorities in both configurations. This approach guarantees safety during reconfigurations while maintaining availability. The protocol includes specific rules for how log entries are committed during these transitions and how the leader manages the membership change process.

Complete specification

It sets Raft apart from its predecessors by leaving little to interpretation. The Raft papers include detailed pseudocode for all major algorithms, timing parameters, exact message formats, and explicit state transition rules. For example, the papers specify exactly how leaders should retry failed communications, what information must be persisted to stable storage before responding to messages, and how to handle every possible server state transition. This comprehensive approach eliminates the implementation guesswork that plagued earlier consensus protocols [1].

To use an analogy: if Paxos is like a complex mathematical proof with critical steps left as "exercises for the reader," Raft is like a detailed instruction manual with clear diagrams and troubleshooting guides. Both can build reliable distributed systems, but Raft dramatically reduces the expertise required to implement them correctly.

Implementation in Modern Systems

Raft's simplicity has led to widespread adoption across many types of systems in the software industry. High-throughput peer-to-peer systems have successfully adapted Raft consensus to provide strong consistency guarantees without compromising on performance expectations [5]. The algorithm's clear design principles have made it accessible to development teams without requiring specialized expertise in distributed consensus.

Distributed databases

Distributed databases have embraced Raft as a foundational component for maintaining consistency. CockroachDB, a distributed SQL database, implements Raft at its core to synchronize data across multiple servers[7]. Each table's data is divided into ranges, and each range has its own Raft group consisting of multiple replicas. When a client writes data to CockroachDB, the request goes to the Raft leader for that particular range, which then replicates the change to other nodes in the Raft group. This architecture ensures that even if individual nodes fail, the database remains available and consistent. TiKV, the storage engine behind TiDB, similarly uses multiple Raft groups to manage different data partitions, with each group independently handling leadership and replication for its assigned key range [11]. Kafka's move to KRaft (Kafka Raft) represents a significant evolution in how it handles consensus and leader election. Traditionally, Kafka relied on ZooKeeper, an external system, to manage cluster metadata, leader election, and configuration. This created additional complexity as teams needed to operate and maintain two separate distributed systems. With KRaft, Kafka implemented its own consensus protocol based on the Raft algorithm, allowing it to handle leader election internally without ZooKeeper. This change simplified the architecture, reduced operational overhead, and improved performance during leader transitions. By consolidating these critical functions within Kafka itself, the system became more self-contained and easier to deploy, while still maintaining strong consistency guarantees when electing new leaders after failures [15].

Configuration management systems

Configuration management systems rely on Raft to store critical system settings and service information. HashiCorp's Consul uses Raft to maintain its service registry and key-value store across a cluster of servers. When configurations change, Consul's Raft implementation ensures all nodes eventually receive the same updates in the same order.

Container orchestration

Container orchestration platforms like Kubernetes depend on Raft indirectly through their reliance on etcd. Kubernetes stores all cluster states—including which containers should be running, which nodes are available, and current configurations—in etcd. The Kubernetes control plane continuously compares the actual state to this desired state stored via Raft consensus. This architecture means that even if master nodes fail, the cluster's intended state remains safely preserved in the etcd Raft cluster. Kubernetes typically deploys a 3-5 node etcd cluster, configured with specific storage and network requirements to ensure Raft can maintain consensus efficiently.

Microservices infrastructure

The HashiCorp stack includes Nomad for scheduling and deploying applications across a cluster, which uses Raft internally to maintain job specifications and allocations. These service coordination tools require the strong consistency that Raft provides to prevent "split-brain" scenarios where independent parts of the system make conflicting decisions about service locations or task assignments [13].

Blockchain systems

Blockchain systems have also adopted Raft variants for consensus in permissioned environments. Hyperledger Fabric, an enterprise blockchain platform, offers a Raft ordering service option that outperforms its previous consensus mechanisms in benchmarks. Unlike public blockchains that require complex proof-of-work or proof-of-stake protocols, permissioned blockchain networks with trusted participants can leverage Raft's performance advantages. These implementations typically modify Raft slightly to accommodate blockchain-specific requirements, such as handling cryptographic verification of transactions while still using Raft's core leader election and log replication mechanisms [14].

The diversity of these implementations demonstrates Raft's versatility across different domains. Each of these systems has implemented the core Raft protocol—leader election, log replication, and safety guarantees—while adapting peripheral details to their specific use cases. The clarity of Raft's specification has enabled development teams to make these adaptations confidently without requiring consensus algorithm specialists to validate every design decision. This accessibility has accelerated Raft's adoption throughout the industry and contributed to more reliable distributed systems overall.

Benefits of Using Raft

Raft offers several practical advantages that have contributed to its rapid adoption across the software industry. Formal verification of Raft implementations provides mathematical certainty about correctness properties, giving developers extraordinary confidence in their distributed systems [6]. These benefits directly address the pain points that development teams experienced with earlier consensus algorithms.

Fewer bugs

Fewer bugs emerge in production systems built on Raft due to its clarity and comprehensiveness. The structured nature of the algorithm, with its clear separation of concerns between leader election, log replication, and safety guarantees, allows developers to reason about each component individually. When implementation issues arise, engineers can identify which specific part of the protocol is misbehaving rather than trying to debug a monolithic system. This modularity translates directly to more reliable software in production. Engineering teams report spending significantly less time troubleshooting consensus-related issues in Raft-based systems compared to their previous experiences with more complex algorithms. The risk of subtle edge cases—like those that might occur during network partitions or when multiple nodes believe they are leaders—is reduced through Raft's explicit handling of these scenarios in its specification.

Strong consistency

Strong consistency guarantees provide a solid foundation for applications that cannot tolerate divergent states. Raft ensures that once a command is committed (meaning it has been replicated to a majority of nodes), it will never be overwritten or reordered in the log. All nodes will eventually execute the same commands in exactly the same order, leading to an identical state across the cluster. This linearizable consistency model means that once a write operation completes, all subsequent reads will reflect that write, regardless of which node handles the read request. For applications like financial systems, inventory management, or any service where an accurate state is critical, this strong consistency eliminates entire categories of potential bugs related to stale or conflicting data that eventually plague consistent systems.

Fault tolerance

Fault tolerance allows Raft-based systems to continue operating normally despite server failures. Raft clusters can tolerate the failure of up to (N-1)/2 nodes while maintaining both availability and consistency, where N is the total number of nodes. In practical terms, this means a three-node cluster can tolerate one failure, a five-node cluster can tolerate two failures, and a seven-node cluster can tolerate three failures. This property ensures that scheduled maintenance, hardware failures, or network issues affecting a minority of nodes won't disrupt service. When node failures occur, Raft's leader election mechanism activates promptly to establish a new leader if needed, typically restoring full functionality within a few hundred milliseconds. This resilience is achieved without complex recovery procedures or manual intervention, reducing the operational burden on engineering teams.

Developer-friendly

Developer-friendly design makes Raft accessible to mainstream software engineers without specialized expertise in distributed systems. The algorithm can be implemented by following the clearly defined rules and procedures in the Raft papers without requiring a deep understanding of the theoretical complexities underlying consensus problems. This accessibility has enabled more organizations to build strongly consistent distributed systems without assembling teams of distributed systems experts. Development teams report that new engineers can understand Raft implementations more quickly than other consensus mechanisms, reducing onboarding time and enabling broader participation in system development and maintenance. The existence of educational tools like interactive visualizations further lowers the barrier to understanding how Raft operates under different conditions.

Comparable performance

Comparable performance to more complex alternatives makes Raft practical for production use cases. Despite its focus on understandability, Raft achieves throughput and latency metrics similar to other consensus protocols in most common scenarios. Write operations typically complete within a few milliseconds in local deployments, with performance primarily bounded by the latency of persisting log entries to stable storage. Read operations can be optimized in various ways, including serving reads from follower nodes for improved scalability or implementing read leases to avoid consensus overhead for reads altogether. These performance characteristics enable Raft to support demanding applications without requiring developers to sacrifice clarity for efficiency, removing a common objection to adopting strongly consistent systems [12].

These benefits combine to make Raft particularly valuable for teams that need reliability without excessive complexity. By providing a consensus solution that balances theoretical correctness with practical implementation concerns, Raft has democratized access to strong consistency and enabled more organizations to build robust distributed systems.

Challenges and Limitations

Despite its many advantages, Raft faces several challenges and limitations that affect its application in certain contexts. The standard Raft model faces read scalability challenges by routing all operations through the leader, but this can be mitigated through thoughtful extensions to the protocol [7]. These constraints represent inherent trade-offs in Raft's design rather than implementation flaws, and understanding them is crucial for architects considering Raft for their distributed systems.

Leader bottleneck

Leader bottleneck issues arise from Raft's centralized leadership model. Since all write operations must flow through the leader node before being replicated to followers, the leader can become a performance bottleneck in write-intensive workloads. This single-leader architecture means that the overall throughput of the system is fundamentally limited by what a single node can process. When a client requests exceed the leader's capacity to append entries to its log, process responses, and manage replication, latency increases, and overall system throughput plateaus. This bottleneck becomes particularly apparent in systems that handle thousands of writes per second or that process large entries that consume significant network bandwidth during replication. Some implementations address this limitation through batching techniques that group multiple client requests into single log entries, but the fundamental constraint remains inherent to Raft's design.

Brief unavailability during leader changes

Brief unavailability during leader changes affects system responsiveness following failures. When a Raft leader becomes unavailable due to crashes, network issues, or scheduled maintenance, the cluster must detect the failure through timeout mechanisms and then elect a new leader before it can resume processing write requests. This election process typically takes at least one election timeout period (often configured between 150-500ms) and sometimes longer if initial election attempts don't establish a clear winner. During this interval, the system cannot process new write operations, creating a brief but noticeable pause in service. While this unavailability period is typically short in stable networks, it can extend longer in environments with network instability or when multiple nodes fail simultaneously. This characteristic makes standard Raft implementations potentially problematic for applications with strict real-time requirements that cannot tolerate any interruption in write availability.

Scaling limitations

Scaling limitations become evident as Raft clusters grow beyond a handful of nodes. The leader in a Raft cluster must communicate with every follower, sending heartbeats and replicating log entries. As the number of nodes increases, this communication overhead grows linearly, placing an increasing burden on the leader and the network. Additionally, since Raft requires acknowledgment from a majority of nodes before committing entries, larger clusters increase the likelihood that slower nodes will delay the commit process. These factors combine to make standard Raft implementations less efficient in very large clusters. While five to seven nodes represent a common and efficient configuration for most applications, scaling beyond this size often requires architectural modifications like hierarchical Raft or partitioned consensus groups rather than simply adding more nodes to a single Raft cluster.

Network partition

Network partition handling in Raft prioritizes consistency over availability, which creates operational challenges during network splits. When the network divides the cluster such that no partition contains a majority of nodes, Raft's safety properties prevent any partition from making progress on write operations. While this behavior preserves consistency by preventing divergent states between partitions, it means that the entire system becomes unavailable for writes during certain network failure scenarios. Even when a majority partition exists, nodes in the minority partition become unavailable until connectivity is restored. This strict consistency approach differs from eventually consistent systems that might allow continued operation with potential reconciliation later. Organizations deploying Raft must carefully consider their network reliability and design their cluster topologies to minimize the impact of network partitions.

Resource efficiency

Resource efficiency concerns arise from the asymmetric workload distribution in Raft clusters. Follower nodes in standard Raft implementations primarily serve as passive replicas that maintain copies of the log and respond to heartbeats and appendEntries messages from the leader. This design means that computational resources on follower nodes are often underutilized, particularly their CPU and memory. In a standard three or five-node deployment, this represents a significant portion of the cluster's total resources being used sub-optimally. Some implementations address this limitation by allowing read operations to be served directly from follower nodes (with various consistency guarantees) or by colocating multiple independent Raft groups on the same physical servers, but these approaches add complexity to the otherwise simple Raft model.

These challenges reflect the fundamental trade-offs inherent in Raft's design philosophy, which prioritizes clarity and safety over-optimization for every possible scenario. Much like choosing a reliable family sedan versus a high-performance sports car, Raft offers exceptional ease of use and maintenance at the cost of some specialized performance characteristics. For most distributed applications, these limitations are acceptable compromises given the significant benefits in implementation simplicity and operational reliability. However, architects should carefully evaluate these constraints against their specific requirements when deciding whether Raft is the appropriate consensus mechanism for their systems.

Future Directions

Researchers and engineers continue to advance Raft in several exciting directions that address its limitations while preserving its core benefits. RaftOptima enhances the original Raft algorithm with optimizations that substantially improve fault tolerance and scalability while maintaining the core understandability that made Raft popular [8]. These innovations extend Raft's applicability to new domains and challenges in distributed computing.

Performance enhancements

Performance enhancements are addressing one of Raft's primary limitations: the leader bottleneck. Advanced batching techniques allow leaders to group multiple client requests into single log entries, amortizing the overhead of consensus across many operations. Some implementations now support pipelined replication, where the leader can send new append entry requests before receiving responses to previous ones, keeping the network saturated and improving throughput. Read optimization strategies have evolved beyond basic leader-only reads to include follower reads with various consistency guarantees, lease-based approaches that reduce consensus overhead, and snapshot-based reads that don't require log traversal. These techniques can dramatically improve throughput in read-heavy workloads without compromising Raft's safety guarantees.

Log compression and efficient state transfer mechanisms reduce the overhead of bringing new nodes up to date. Rather than sending thousands of individual log entries, optimized implementations transfer compressed snapshots of state followed only by recent entries. Some systems have implemented parallel execution of independent commands, allowing the state machine to apply non-conflicting operations simultaneously after they're committed through the consensus protocol. These performance enhancements collectively enable Raft-based systems to support higher throughput and lower latency while maintaining the algorithm's understandability.

Multi-Raft approaches

Multi-Raft approaches have emerged as the dominant strategy for scaling Raft to manage large datasets and high request volumes. Instead of using a single Raft group for an entire system, this architecture partitions data into multiple shards, each managed by an independent Raft consensus group. TiKV, the storage engine behind TiDB, exemplifies this approach by dividing its keyspace into ranges called "regions," each with its own Raft group. CockroachDB similarly uses "ranges" of around 64MB each, with separate Raft groups handling replication for each range. This horizontal scaling allows systems to distribute load across many leaders instead of funneling all requests through a single consensus group.

Coordination between these independent Raft groups introduces new challenges, particularly for operations that span multiple partitions. Distributed transaction protocols layered on top of multi-Raft architectures enable consistent cross-partition operations, often using two-phase commit or other coordination mechanisms while relying on Raft for consistent replication within each partition. These multi-Raft systems can scale to hundreds or thousands of nodes by limiting the size of individual consensus groups while using many such groups in parallel, overcoming the inherent scaling limitations of single-group Raft.

Mathematical verification

Mathematical verification efforts have produced formal, machine-checked proofs of Raft's correctness properties. Unlike informal reasoning or testing, formal verification uses mathematical techniques to prove that an algorithm satisfies certain properties under all possible executions. Adaptations of Raft for federated learning demonstrate its flexibility beyond traditional distributed systems, providing fault tolerance and self-recovery capabilities in machine learning contexts[9]. Other projects have verified TLA+ specifications of Raft, mathematically proving properties like leader uniqueness (no two nodes can be leaders for the same term) and log matching (if two logs contain entries with the same index and term, those entries are identical). These verification efforts provide extraordinary confidence in Raft's correctness beyond what testing alone can achieve. As formal methods tools become more accessible, we can expect more Raft implementations to undergo rigorous mathematical verification, further increasing confidence in systems built on this foundation.

Edge computing adaptations

Edge computing adaptations are extending Raft to environments with intermittent connectivity and limited resources. Standard Raft assumes relatively stable network conditions and continuous majority availability,

assumptions that don't hold in edge computing scenarios where devices may frequently disconnect or have limited power and bandwidth. Modified Raft variants for these environments introduce concepts like "blessed" majorities that can make progress during partitions, delayed consistency mechanisms that maintain safety while improving availability, and hierarchical approaches where edge devices form local consensus groups that periodically synchronize with cloud-based Raft clusters.
Some edge-oriented modifications incorporate concepts from Conflict-free Replicated Data Types (CRDTs) to allow for offline operation with eventual reconciliation. These adaptations enable Raft-like consensus in challenging environments like Internet of Things (IoT) deployments, mobile device clusters, and remote locations with unreliable connectivity while maintaining as many of Raft's safety guarantees as possible, given the constraints. As computing continues to move toward the edge, these adaptations will become increasingly important for maintaining consistency in distributed applications.

Visual teaching tools

Visual teaching tools have significantly contributed to Raft's accessibility and adoption. Interactive visualizations allow students and engineers to observe Raft's behavior under different conditions, experimenting with scenarios like network partitions, node failures, and message delays to develop intuition about the algorithm's properties. The Raft website (raft.github.io) offers a visualization that has become a standard teaching tool in distributed systems courses. More advanced simulators allow users to modify parameters like election timeouts and heartbeat intervals to observe their effects on system behavior, reinforcing understanding through experimentation.

These educational resources complement Raft's understandability-first design, further reducing the barrier to entry for engineers working with consensus systems. As these tools continue to evolve with more sophisticated scenarios and clearer explanations, they will help train the next generation of distributed systems engineers with a solid understanding of consensus fundamentals. The combination of a clearly designed algorithm and excellent educational resources represents a significant advancement in making distributed consistency accessible to mainstream software developers. Out-of-order execution extensions to Raft enable higher throughput by allowing independent operations to proceed in parallel while maintaining consistency guarantees for dependent operations [10].

These ongoing developments ensure that Raft will continue to evolve alongside the changing landscape of distributed computing. By addressing Raft's limitations while preserving its core clarity and safety, these innovations extend its relevance to new domains and requirements. As distributed systems become increasingly prevalent across all areas of computing—from cloud infrastructure to edge devices to blockchain networks—Raft's influence on consensus protocol design will likely persist for years to come.

CONCLUSION

Raft has transformed distributed systems by making reliable consensus accessible to everyday developers without sacrificing correctness. Its focus on clarity fundamentally changed how engineers approach distributed consistency, enabling wider adoption of strongly consistent systems across critical applications. The algorithm's decomposition into distinct components creates both mental clarity and practical advantages for implementation and debugging. While certain inherent trade-offs exist in its design, particularly around centralized leadership and scaling, these limitations have sparked creative extensions that preserve Raft's core simplicity while expanding its capabilities. Through batched operations, partitioned consensus groups, formal verification, and adaptations for challenging network environments, Raft continues to evolve alongside distributed computing needs. This combination of foundational clarity and ongoing innovation ensures Raft will remain influential in distributed systems design far into the future.

REFERENCES

[1] Diego Ongaro and John Ousterhout, "In Search of an Understandable Consensus Algorithm," in the Proceedings of USENIX ATC ’14: 2014 USENIX Annual Technical Conference, 2014. Available: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

[2] Heidi Howard, Richard Mortier "Paxos vs Raft: Have we reached consensus on distributed consensus?" PaPoC ’20, April 27, 2020, 2020. Available: https://dl.acm.org/doi/pdf/10.1145/3380787.3393681

[3] Heidi Howard et al., "Raft Refloated: Do We Have Consensus?," ACM SIGOPS Operating Systems Review, vol. 49, no. 1, pp. 12-21, 2015. Available: https://www.cl.cam.ac.uk/research/srg/netos/papers/2015-raftrefloated-osr.pdf

[4] Harald Ng, "Distributed Consensus: Performance Comparison of Paxos and Raft," Kth Royal Institute Of Technology, 2020. Available: https://www.diva-portal.org/smash/get/diva2:1471222/FULLTEXT01.pdf

[5] Mahmood Fazlali et al., "Raft Consensus Algorithm: an Effective Substitute for Paxos in High Throughput P2P-based Systems," arXiv preprint arXiv:1911.01231, 2019. Available: https://arxiv.org/pdf/1911.01231

[6] Doug Woos et al., "Planning for Change in a Formal Verification of the Raft Consensus Protocol," CPP’16, January 18–19, 2016. Available: https://dl.acm.org/doi/pdf/10.1145/2854065.2854081

[7] Vaibhav Arora et al., "Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols," in Proceedings of the 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), 2017. Available: https://www.usenix.org/system/files/conference/hotcloud17/hotcloud17-paper-arora.pdf

[8] Kiran Kumar Kondru and Saranya Rajiakodi, "RaftOptima: An Optimised Raft With Enhanced Fault Tolerance, and Increased Scalability With Low Latency," IEEE Access, vol. 10, pp. 123456-123467, 2024. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10614451

[9] Rustem Dautov, Erik Johannes Husom, "Raft Protocol for Fault Tolerance and Self-Recovery in Federated Learning," IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2024. Available: https://dl.acm.org/doi/pdf/10.1145/3643915.3644093

[10] Xiaosong Gu et al., "Raft with Out-of-Order Executions," International Journal of Software and Informatics, 2021. Available: https://hengxin.github.io/papers/2021-JOS-PRaft-en.pdf

[11] “Consensus algorithm”, Tikv Org Available:https://tikv.org/deep-dive/consensus-algorithm/introduction/

[12] Martin Kenyeres,Jozef Kenyeres, “Comparative Study of Distributed Consensus Gossip Algorithms for Network Size Estimation in Multi-Agent Systems”, Future Internet, 2021. Available : https://www.researchgate.net/publication/351669184_Comparative_Study_of_Distributed_Consensus_Gossip_Algorithms_for_Network_Size_Estimation_in_Multi-Agent_Systems

[13] Joseph DeChicchis, “Cloudlet Caches: Managing State for a Microservices Based Edge Computing Platform”, Duke University, December 13, 2018. Available: https://www.dechicchis.com/assets/Cloudlet_Caches.pdf

[14] Hao Xu, Lei Zhang and Yinuo Liu, and Bin Cao ,“RAFT Based Wireless Blockchain Networks in the Presence of Malicious Jamming”, IEEE Wireless Communications Letters, Vol. 9, No. 6, June 2020. Available : https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8982036

[15] Anna Povzner, et al., “Kora: A Cloud-Native Event Streaming Platform For Kafka”,Proceedings of the VLDB Endowment, Vol. 16, No. 12, Available : https://www.vldb.org/pvldb/vol16/p3822-povzner.pdf