Raft With RAPID - In Search of a Simpler Consensus Algorithm

Alex Chandler
The University of Texas at Austin
alex.chandler@utexas.edu

David Klingler
The University of Texas at Austin
davidklingler@utexas.edu

1 Abstract

In this paper, we present Raft with RAPID, a system that integrates RAPID's Stable and Consistent Membership Views into the Raft consensus algorithm. We show how this integration simplifies cluster management, leader and log implementation. Our evaluation shows that Raft with RAPID offers comparable performance, striking a practical balance between efficiency and simplicity.

2 Introduction

Consensus algorithms are pivotal in distributed computing, providing a reliable mechanism for multiple independent servers to agree on essential decisions. This ensures system integrity and consistency, especially in scenarios of server failures or network disruptions.Consensus is crucial in a variety of applications, ranging from database replication to blockchain technology, where the integrity and reliability of data are paramount. The primary goal of consensus algorithms is to coordinate among participating servers in a distributed system to reach a unified agreement on system states or decisions, ensuring consistency and reliability despite challenging conditions such as network delays, server failures, and the inherent asynchronicity in distributed communication.

The evolution of consensus algorithms in distributed systems has been marked by significant milestones. One of the earliest and most influential consensus algorithms is Paxos (Lamport, 1998). Paxos is known for its rigorous theoretical foundation, but was often criticized for its complexity and challenging implementation. Recognizing the complexity of the original Paxos algorithm, Lamport authored Paxos Made Simple (Lamport, 2001) to clarify its concepts and present the algorithm in a more accessible and comprehensible manner. Paxos Made Moderately Complex (Van Renesse and Altinbuken, 2015) was written to address the lingering ambiguities and implementation challenges associated with the Paxos algorithm. The paper provided a more detailed operational guide, explicitly addressing the Multi-Paxos process, and the role of various components like proposers, acceptors, and learners.

Over time, numerous variants of Paxos were developed, each aiming to refine the algorithm by addressing specific issues while upholding its strong consistency guarantees. For instance, Fast Paxos, introduced in (Lamport, 2006), was designed to reduce the latency in reaching consensus by allowing learners to accept proposals from a quorum of fast proposers, thus speeding up the decision-making process. Meanwhile, Flexible Paxos, detailed in (Howard et al., 2016), introduced flexibility in the quorum size requirements for different phases of the consensus process, enabling a more adaptable and efficient approach to achieving consensus in varied network conditions and workloads.

Despite the advancements offered by these Paxos variants, the underlying complexity and intricate nuances of the algorithm remained a challenge, motivating the development of Raft (Ongaro and Ousterhout, 2014). Raft was specifically designed to be more understandable and straightforward to implement, with its architecture emphasizing simplicity and clarity, factors that have contributed to its widespread adoption in modern distributed systems. However, Paxos is still quite commonly used in large-scale, high-availability systems, including RAPID (Suresh et al., 2018), a cluster configuration management system which we later use in an attempt to simplify Raft. RAPID was designed specifically for managing large-scale, dynamic distributed systems by ensuring rapid, stable, and consistent membership changes even in the presence of frequent network fluctuations and server failures.

2.1 Background

Building upon the our introduction of the history and evolution of consensus algorithms, we shift our focus towards explaining the higher level ideas of consensus algorithms used in our project.

2.2 Raft

Raft is a consensus algorithm renowned for its simplicity and understandability, particularly in comparison to its predecessors like Paxos. Central to Raft's design is a leader-centric approach, where the distributed system is organized into servers that can assume one of three roles: Leaders, Followers, or Candidates. The leader is responsible for managing the log entries, which consist of client commands and configuration changes, and ensuring their replication across the follower servers. This replication is crucial for maintaining a consistent state across the distributed system. The heartbeat mechanism is a key feature in Raft, used by the leader to maintain authority and for followers to detect leader failures. Followers regularly receive heartbeats from the leader; if a heartbeat is not received within a predetermined interval—often a sign of leader failure or network issues—the follower transitions to a candidate state.

In the candidate state, servers initiate a leader election process. This process involves a randomized timeout to reduce the likelihood of split votes, a condition where no single candidate receives a majority of the votes. Achieving a majority vote allows the candidate to become the new leader for a specific term. This election process ensures that at any given time, there is only one leader within the system, a critical aspect of maintaining order and consistency in decision-making. The leadership transition is seamless and maintains the system's operability even in the event of server failures or network partitions. Raft's approach to log management and leader election illustrates its commitment to a more straightforward and operational consensus algorithm, aiming to provide a reliable yet simpler alternative to the complexity often associated with traditional consensus mechanisms like Paxos.

2.3 RAPID

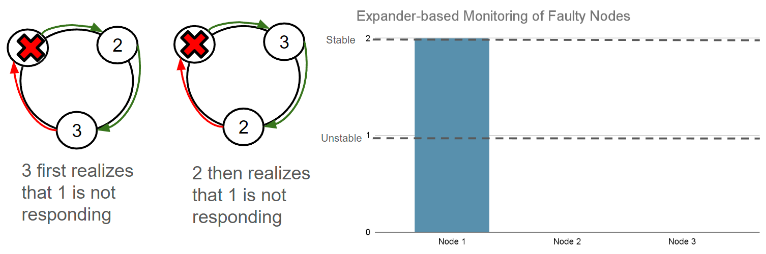
RAPID (Suresh et al., 2018), introduced in a 2018 paper, offers a leaderless consensus protocol and system that can provide stable, consistent, and reliable membership views at scale. It excels in efficiently detecting cluster-wide conditions and handling complex network failures. Multi-Process Cut Detection underpins RAPID's ability to accurately detect failures, playing a crucial role in its efficiency. This method, coupled with a broadcasting system, ensures that any operational server in the cluster promptly recognizes and agrees on configuration changes, thereby maintaining widespread consensus.

2.3.1 RAPID Fault Detection Mechanism

Figure 1a highlights the RAPID fault detection system's capabilities, particularly its Multi-Process Cut Detection mechanism used for identifying server faults in a cluster. The left hand side of 1a depicts two rings, each representing cluster servers as nodes. These nodes are connected by arrows, indicating their error monitoring responsibilities. In the left most ring, server 1 monitors server 2, server 2 monitors server 3, and server 3 monitors server 1, creating a cyclical pattern. The right most ring shows a different monitoring configuration: server 1 oversees server 3, server 3 watches over server 2, and server 2 keeps an eye on server 1. The right-hand side of the figure presents a chart titled "Expander-Based Monitoring of Faulty Nodes," where each server updates fault counts for each other. While the figure shows a single graph, each server tracks the status of every other server through its own fault detection responsibilities and receiving broadcasts from other non-faulty nodes.

In both depicted rings, a situation arises where server 1 becomes unreachable. This triggers a response mechanism within the system: any server detecting an unreachable peer, such as server 1 in this scenario, updates its failure count for that specific server. Concurrently, it sends out an alert to the entire cluster. Upon receiving this alert, other operational servers adjust their own failure counts for server 1, incrementing by one. Given that two servers (one in each ring) are tasked with monitoring server 1, each non-faulty server ultimately updates its failure count for server 1, reflecting the total number of servers that have detected the issue.

On the right side of 1a, the "Expander-based Monitoring of Faulty Nodes" chart displays the current status of server 1's failure. This chart includes two key threshold lines, marked 'stable' and 'unstable,' which guide server behavior. Servers wait to propose configuration changes until their failure counts surpass the 'stable' threshold or fall below the 'unstable' threshold. This approach ensures that decisions are based on the system's stabilized


Expander-based Monitoring of Faulty Nodes
2
3
3 first realizes that 1 is not responding
3
2
2 then realizes that 1 is not responding
Stable
Unstable
2
1
0
Node 1
Node 2
Node 3

(a) RAPID Fault Detection Mechanism

state rather than reacting to frequent, minor fluctuations. Counts below the 'unstable' threshold are considered mere background noise.

2.4 Motivation

Our goal was to develop a simpler consensus algorithm, with Raft as the foundational framework. While Raft emerged as a simpler alternative to the complex Paxos algorithm, it still harbors its own intricate aspects. The complexities involved in handling overlapping configurations during cluster changes, the shared log, and the diverse duties associated with append entries underscore the necessity for further simplification. Our goal was reinforced by the concept of attaining simplicity by decomposition highlighted in the Raft paper, a principle we sought to advance further. In our search for an ideal solution, we were specifically looking for a system that could alleviate the burdens associated with these complexities.

Ultimately, RAPID emerged as an ideal candidate to combine with Raft. With RAPID handling cluster management, we can limit Raft's focus need for consensus to client values. This integration of Raft with RAPID not only simplifies the management of cluster log entries but also autonomously handles cluster configurations and heartbeats, thereby reducing the operational complexity of our consensus algorithm. We additionally valued RAPID's contribution to enhancing stability and ensuring consistent membership views, alongside its provision of strong consistency guarantees.

3 Main Idea

In this section, we highlight how the following main modifications lead to a simpler consensus algorithm. This integration brings several key improvements:

Faulty Server Identification and Cluster Management: Raft encountered complexities in managing cluster configurations and identifying faulty servers, particularly during periods of dynamic change. The integration with RAPID brings a significant simplification to this aspect. By having RAPID autonomously overseeing server failures and cluster reconfigurations, Raft no longer has to reach to reach consensus on cluster configuration changes and only has to manage the consensus of client proposed values.

Log State Reduction: We have streamlined Raft's log management by reducing the log state to only encompass client-proposed values. Previously, managing the log included a complex state of various configurations and system states. This change significantly simplifies the log's structure and state.

Simplified Cluster Configuration Changes: Raft's complex process of transitioning through $'old'$, $'old\_new'$, and $'new'$ configurations was a significant challenge. Our approach removes these complexities, as RAPID runs its own consensus algorithm for cluster changes, making the cluster transition from Raft's point of view much simpler and only from $'old'$ to $'new'$.

Reduced AppendEntriesRPC Role: Previously, Raft's AppendEntries function was multifaceted, serving as a heartbeat, handling cluster configuration changes, and appending client values to the log. Our integrated system refines this by assigning AppendEntries the singular role of appending client values. Additionally, the integration with RAPID's leaderless protocol eliminates the need for the heartbeat mechanism, traditionally crucial in Raft for leader election and failure detection.

4 Design and Implementation

As the design Raft with RAPID is based heavily off of Raft, we elect to only discuss the major changes we made to the Raft algorithm. Many aspects of the Raft algorithm including the mechanism by which log entries are committed to the log remain unchanged. We used JRaft (DataTechnology, 2022), an open source implementation of the Raft consensus algorithm, as a basis for our implementation of Raft with RAPID.

4.1 Leader Election and Configuration Changes

The motivation for using RAPID with Raft was to have RAPID handle the cluster configurations. This includes detecting failures and thus alleviates the leader from having to send heartbeats in the form of AppendEntries requests to all the followers. However, now that followers are not being sent heartbeats, there must be a new mechanism for starting an election. A new election should start when the current leader fails and so whenever RAPID notifies servers that a configuration change has occurred, the servers check to see if the current leader is in the new configuration. If the leader is not in the new configuration, that means it has failed. However, instead of immediately changing their role to a candidate and attempting to request votes, servers wait a randomized amount of time before converting to a candidate. Just like in Raft, this randomized election timeout is used to prevent split votes and ensure that a leader is eventually elected.

For configuration changes, we decided to trade off flexibility for simplicity. The configuration change procedure described in the Raft paper is very flexible, allowing a cluster to change to an arbitrary new configuration with any number of servers. This flexibility comes at the cost of simplicity and easy implementation. Raft with RAPID chooses the other end of the spectrum favoring simplicity over flexibility. In Raft with RAPID, the replication factor of a cluster does not change. This also imposes a requirement that the cluster administrator ensure that only n instances of Raft with RAPID servers are alive at any time for a cluster with replication factor $\lfloor \frac{n}{2} + 1 \rfloor$. Finally, for any cluster configuration change, a majority of the servers in the old configuration must also be in the new configuration.

A valid configuration change is any change of servers in the cluster that meets the limitations as described above. Servers may be added and removed in the same configuration change. RAPID notifies servers of a configuration change whenever it detects one or more servers requesting to join the cluster and/or one or more server failures. RAPID only notifies servers of a configuration change after reaching consensus among a majority of servers in the cluster. Once a server receives the notice of a new configuration, it switches to the new configuration by updating its list of servers in the cluster. Servers only respond to messages (AppendEntries/RequestVote requests) from other servers in the same configuration. This is enforced by sending the configuration id of the current cluster configuration generated by RAPID in each message between servers. The old configuration is able to append log entries and elect a new leader until it no longer has a majority of servers left in the old configuration. Because there can only be at most n alive servers at any time for a replication factor of $\lfloor \frac{n}{2} + 1 \rfloor$, and a majority of servers in the old configuration are also in the new configuration, it is impossible for there to be $\lfloor \frac{n}{2} + 1 \rfloor$ alive servers in the old and new configuration at the same time. Thus at most one configuration is able to append new values to the log at any time.

One of the nice properties of this way of changing configurations is that if the leader of the old configuration is in the new configuration it remains the leader in the new configuration. This is because leader elections to not start unless servers notice that the current leader is not in the new configuration. In addition, newly added servers in the new configuration also do not start elections since they have never seen a leader.

One of the requirements of Raft is that any leader must have the most up to date log. This is enforced by requiring that any server that votes for a candidate only does so if that candidates log is as up to date as the server's own log. This remains unchanged in Raft with RAPID. However, how configuration changes are handled breaks this requirement in extreme cases. In the scenario where $\lceil \frac{n}{2} - 1 \rceil$ servers fail (one less than the majority of servers in the cluster) including the leader and then those servers are replaced in the same configuration change, the cluster may elect a leader that is missing committed entries. This can occur because the leader failed and the servers replacing the failed servers are voting members of the cluster. A server from the old configuration that did not fail but was behind on committed log entries could become the leader using itself and the set of newly joined servers to form a quorum. This compromises safety.

In order to ensure this does not occur, we require that servers in the old configuration only switch to the new configuration if the current leader is also in the new configuration. If the leader of the old configuration fails and thus is not in the new configuration, just the servers of the old configuration elect a new leader before switching over to the new configuration. This is ensures that the newly elected leader has the most up to date log. As discussed previously, this newly elected leader also becomes the leader of the new configuration. This causes one minor issue in that a configuration change cannot occur unless there is an elected leader. However, when the cluster is first starting it cannot make the first configuration change because there is no leader. To resolve this, the server bootstrapping the cluster automatically becomes the leader in order for more servers to join the cluster.

4.2 AppendEntriesRPC

In Raft, the AppendEntriesRPC is used as a mechanism for detecting leader failure, appending client values to the log, and appending cluster configuration changes to the log. However, in Raft with RAPID, RAPID handles both heartbeats and the consensus involved with changing the configuration. Therefore, in Raft with RAPID, the leader only sends AppendEntriesRPCs whenever the leader wants to append a client specified value to the log. This means that every AppendEntriesRPC includes at least one log entry. In addition, the leader no longer has to keep track of when it next needs to send a heartbeat to each follower to ensure it does not timeout and start an election.

4.3 Log State

The other major change we implemented is to the log. Specifically, because RAPID handles the consensus involved with changing the configuration, the log no longer has to serve information about cluster configuration changes. This means that the log only stores client specified values. Log entries no longer have to have a type field because there is only one type of log entry.

5 Evaluation

For our evaluation, each test was completed on cluster sizes of 3, 5, 7, and 9. These sizes were picked since Raft cluster sizes do not get much larger than nine servers in practice and odd numbers are more commonly used since they provide more fault tolerance per server compared with even numbers. In addition, each test was run five times for each cluster size. We report the average across the five runs.

The machines used for the tests were AWS EC2 machines running Amazon Linux 2023. Specifically, t2.Medium instances were employed, featuring 2 vCPUs and 4 GiB of memory. All machines were located in the same availability zone within the North Virginia Region.

Our aim with this evaluation is to show that Raft with RAPID performs similarly to Raft in terms of resource utilization, performance and handling failure/configuration changes.

5.1 Resource Utilization

To understand how Raft with RAPID compares with Raft in terms of resource utilization we ran four tests to measure the amount of memory, bandwidth (sending and receiving) and latency of each system.

Figure 2a shows the memory usage of both Raft with RAPID (RaftRAPID) and Raft (Base). Raft displays a near constant usage of memory across cluster configuration sizes. This is due to the fact that a Raft server only needs to store a list of the servers participating in consensus. The leader in Raft needs to store slightly more information for each server, including information about the last acknowledged heartbeat, and most up to date log index. In comparison, RAPID introduces much more state for each server in addition to the state required to implement Raft. RAPID requires an alert count for each server in the system as well as k pseudo random rings in order for each server to determine which servers it is observing. Finally, RAPID has sever external dependencies that increase the base memory consumption by a considerable amount. This leads to Raft with RAPID requiring about 10 megabytes more memory that Raft for a configuration of three servers. In addition, as the cluster sizes increase, Raft with RAPID consumes roughly one more megabyte of memory per server added. While this is comparatively much more memory consumption compared to Raft, modern commodity hardware comes with gigabytes of memory and combined with the fact that Raft cluster sizes remain small in practice the difference between twenty megabytes and two megabytes is practically negligible.

We also measured the amount of bandwidth, both sending and receiving, that each server used with no clients appending to the log as well as under a full load. For the full load test, clients, which continuously appended to the log, were added until the throughput of log entries no longer increased.

This indicated that that the leader was fully saturated with client requests. Figures 2b and 2c show the amount of bandwidth used on average by a single server in Raft with RAPID (RaftRAPID) and Raft (Base) both with and without load. With no active client requests in the system Raft with RAPID sent and received around 2 kilobytes per second while Raft only sent and received about .1 kilobytes per second. This is due to the fact that in Raft, with no client requests, only the leader has to send heartbeats. However, in Raft with RAPID all servers participate in sending heartbeats to other servers. The amount of bandwidth used under no load is significantly less than the amount of bandwidth under max load and so this slight difference is not impactful on the performance of Raft with RAPID.

In fact, under max load, Raft with RAPID and Raft use comparable amounts of bandwidth. This is expected as the algorithm for appending log entries in Raft with RAPID is the same as Raft and so the amount of messages send should be the same. The amount of bandwidth that both Raft with RAPID and Raft use under load decreases as the cluster size increases. This is expected because the throughput of log entries decreases as the replication factor increases and so less messages are sent per second with larger cluster sizes. These tests demonstrate that using RAPID as a mechanism for cluster management has a negligible impact on the bandwidth utilization.

The final test we performed to compare the resource utilization of Raft and Raft with RAPID was measuring the latency of both systems. Figure 2d shows that the latency of both systems, with no active client requests, was essentially equivalent. Under load, Raft with RAPID had slightly higher latency than Raft. However, with the latency still remaining quite low at less than 1 millisecond, it is difficult to determine if using RAPID was the cause for this slight increase in latency or just factor of the AWS environment. Either way, this test demonstrates that using RAPID has negligible impact on the latency of the network.

5.2 Performance

The next major aspect of Raft and Raft with RAPID we wanted to compare was the relative performance of the systems, specifically the throughput of log entries. Similar to the network utilization test, for each system and cluster size, we added clients, which continuously appended log entries, until leader became fully saturated with client requests and the throughput stopped increasing.

Figure 2g shows the throughput of Raft with RAPID (RaftRAPID) and Raft (Base). Both systems show a linear decrease in throughput of log entries and the cluster size increases. This is expected as increasing the cluster size also increases the replication factor and the system has to wait for the slowest server in a quorum before committing an entry. For a cluster size of three servers, Raft outperforms Raft with RAPID by almost one thousand log entries per second. The reason for this discrepancy in throughput is not immediately apparent when comparing the implementation of Raft and Raft with RAPID as the procedure for appending log entries is essentially the same in both systems. However, for larger cluster sizes of five, seven and nine, the throughput of both systems is functionally equivalent. In fact, Raft with RAPID actually slightly outperformed Raft in our testing for cluster sizes of seven and nine.

5.3 Failure Handling and Configuration Changes

The final major aspect of Raft with RAPID we wanted to compare with Raft was how the two systems handled the failure of servers and the network and how each system recovered. Thus we tested how long each system took to: detect a failed leader and elect a replacement, elect a leader following a network partition, and completely replace all of the servers in a configuration.

Figure 2e, shows the time Raft with RAPID (RaftRAPID) and Raft (Base) took to detect a crashed leader and elect a replacement. To make the comparison as fair as possible, we set the heartbeat timeout in Raft and the edge detection timeout in RAPID to take the same amount of time. Raft is clearly faster than Raft with RAPID at detecting leader failure and then electing a new leader. This is because of how elections work in Raft with RAPID. In RAPID, multiple servers must detect the failure of the leader and broadcast that to the rest of the cluster. The cluster must then perform consensus to agree that the leader has failed. Only once consensus has been reach can servers start their randomized election timeout. Comparatively in Raft, as soon as a single server times out it can become a candidate and start requesting votes. The round of consensus required by RAPID and the delayed start to the randomized timeout both contribute to the increase in time it takes Raft with RAPID to elect a replacement. In addition, for a cluster size of three servers, Raft with RAPID takes considerably longer then for larger cluster sizes. This is because for larger cluster sizes, RAPID performs consensus using the Fast Paxos (Lamport, 2006) consensus protocol which requires only one round of communication. However, for a cluster size of three with one failed server, RAPID has to fall back to regular Paxos (Lamport, 1998) which takes two rounds of communication. Having to use regular Paxos is the reason for the much longer time to elect a replacement leader for a cluster size of three servers. While electing a replacement leader takes a couple of seconds longer in Raft with RAPID compared to Raft, failed leaders in a healthy system are a rare occurrence. We believe that Raft with RAPID presents a reasonable trade off between simplicity and performance of electing a new leader.

We next tested how both Raft and Raft with RAPID handle network partitions. In a partition, where a majority of servers are still able to communicate both systems elect a new leader if necessary and are able to continue to processing client requests. When the partition heals both systems gracefully handle the re-joining of servers from other partitions. Their operation is functionally equivalent. However, in a network partition where there is no majority of servers that can still communicate, Raft and Raft with RAPID handle the situation differently. In Raft, every server that cannot communicate with the leader of the term in which the partition occurred times out and converts to a candidate. These servers will continuously attempt to become the leader until the partition is healed. However, in Raft with RAPID, RAPID cannot make a decision about the cluster configuration without a majority of servers. This means that servers that cannot communicate with the last known leader are not notified that the leader has failed and thus do not convert to candidates. This means that when the partition heals, all servers remain in the same state they were in when the partition occurred and thus no election is required. Figure 2f shows the time to elect a leader after the a network partition has been healed by Raft (Base) and Raft with RAPID (RaftRAPID). This effectively shows the time it takes to resume appending values to the log after the network partition heals as entries cannot be appended to the log without an elected leader. As explained above, Raft with RAPID has an elected leader as soon as the partition heals. In Raft, an election, which takes a few seconds, must occur so that all the servers can agree on a leader. We created network partitions by using the iptables program to drop messages sent from specific machines.

Finally, we tested the amount of time it takes to completely replace the servers in a cluster. In addition, we also ran this test on LogCabin (Ongaro et al., 2017) which was the first implementation of Raft. We performed this test on LogCabin because it implements cluster configuration changes as described in the original Raft paper. The implementation we based Raft with RAPID off of (Base) only allows one server to be added or removed at a time. We were unable to find a open source implementation of Raft in Java that used the configuration change approach as detailed in the Raft paper. Figure 2h shows that LogCabin was by far the fastest to completely change the configuration taking around a second for each cluster size. This is due to the fact that it can completely change the cluster all at once. Raft with RAPID required a roughly constant amount of time to completely change the cluster configuration. This is due to the fact that Raft can remove and replace $\frac{n}{2} - 1$ (one less than a majority) at a time. This means that it takes Raft with RAPID three separate cluster configuration changes to completely change the cluster configuration to all new servers. The base implementation of Raft we used showed a linear increase in time due to the fact that it can only add or remove one server at a time. Raft with RAPID was the slowest for cluster sizes of three and five but was faster than the Raft implementation for cluster sizes of seven and nine. While Raft with RAPID had much slower configuration changes than LogCabin, configuration changes are a relatively rare occurrence and it makes up for its lack of performance with a much simpler implementation.

5.4 Discussion of Simplicity

In our research, we successfully simplified the implementation of our system, reducing the Raft base implementation from 1,096 to 733 lines by integrating RAPID. The decrease in code was primarily due to simplifying configuration changes and AppendEntriesRPC. Furthermore, to assess user perception of our system, we surveyed students enrolled in a distributed computing course. Although all of four respondents indicated they found our version simpler, it's important to note that these results may not be statistically significant, and there is potential selection bias. Since the survey was not conducted blindly, there is a possibility that students expressed a preference for our system out of a desire to be supportive or polite, rather than based on objective evaluation. The survey included four questions asking to select which system they found simpler - cluster configuration management, leader implementation, log implementation, and overall simplicity. In each category, students unanimously found our Raft with RAPID implementation to be simpler compared to the base Raft algorithm.

6 Limitations

Our system presents several limitations. It does not allow for altering the replication size. While servers can be added or removed, the number of servers alive cannot exceed n for a replication size of $\lfloor \frac{n}{2} + 1 \rfloor$. This constraint may impact scalability and flexibility in certain scenarios, and add an extra requirement to the server admin to ensure the condition. Additionally, a majority of servers is required for the transfer from c_{old} to c_{new}. Finally, we must acknowledge that the perceived simplicity of our work is contingent upon viewing RAPID as a 'black box'. This perspective might limit a deeper understanding of the underlying complexities and could skew comparisons with other systems where such assumptions are not made.

7 Future Work

Future enhancements to our system could focus on making the integration of RAPID with Raft more production-ready, enhancing its robustness, efficiency, and scalability for real-world applications. Future work could consider developing a configuration change mechanism that allows for changes in the replication factor, thereby increasing the system's adaptability and efficiency. While our current approach treats RAPID as a 'black box', future work could involve modifying RAPID itself to boost overall efficiency. Exploring the integration of RAPID with other consensus algorithms is another potential area of interest that could broadening RAPID's applicability to other distributed systems.

8 Conclusion

We have successfully integrated RAPID with the Raft consensus algorithm, creating 'Raft with RAPID', and further simplifying a consensus algorithm designed for simplicity. We argue that this integration simplifies Raft's complex components, including leader election, cluster configurations, and log management. Our work demonstrates that integrating RAPID with Raft not only simplifies these operation but does so without significant losses in performance.

References

DataTechnology. 2022. jraft: Raft consensus implementation in java. `https://github.com/datatechnology/jraft`.

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible paxos: Quorum intersection revisited.

Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58.

Leslie Lamport. 2006. Fast paxos. *Distrib. Comput.*, 19(2):79–103.

Diego Ongaro and John K. Ousterhout. 2014. In search of an understandable consensus algorithm (extended version). *arXiv*, abs/1407.5490.

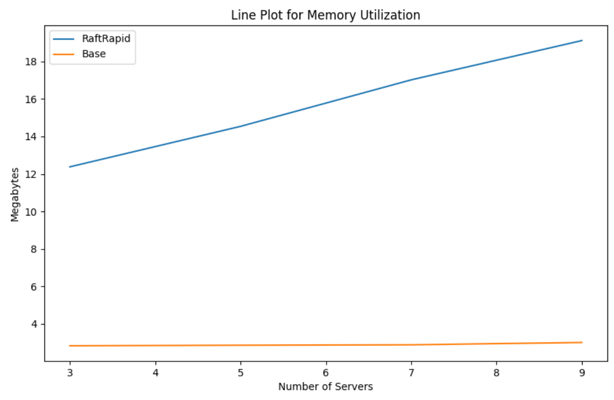
Diego Ongaro et al. 2017. Logcabin: A distributed system providing a small amount of highly replicated, consistent storage. `https://github.com/logcabin/logcabin`.

Lalith Suresh, Dahlia Malkhi, Parikshit Gopalan, Ivan Porto Carreiro, and Zeeshan Lokhandwala. 2018. Stable and consistent membership at scale with rapid. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 387–400, Boston, MA. USENIX Association.

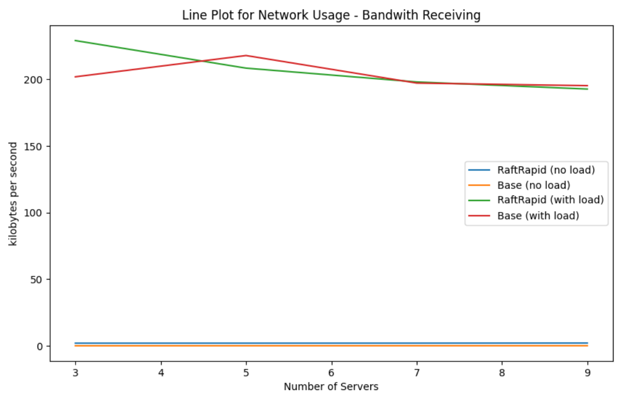
Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3).

Appendix

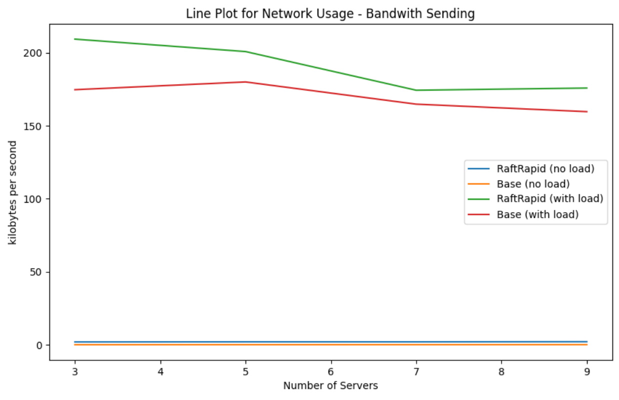
Our implementation and evaluation code for is located at: Raft with RAPID



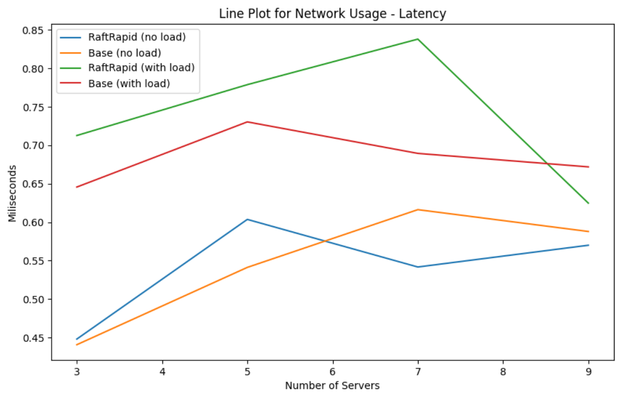
(a) Memory Utilization



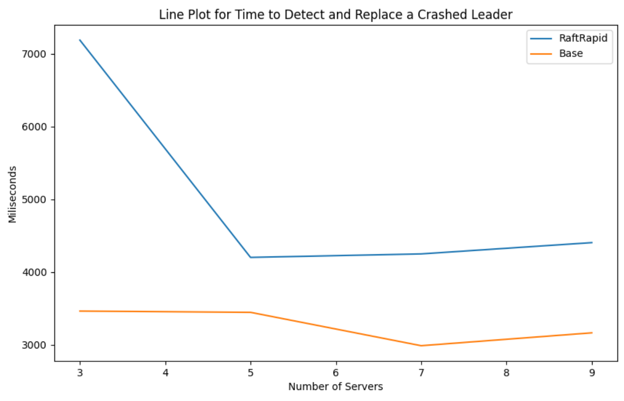
(b) Bandwidth Receiving



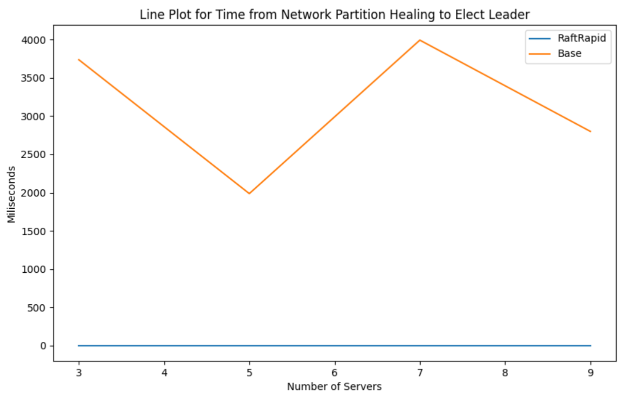
(c) Bandwidth Sending



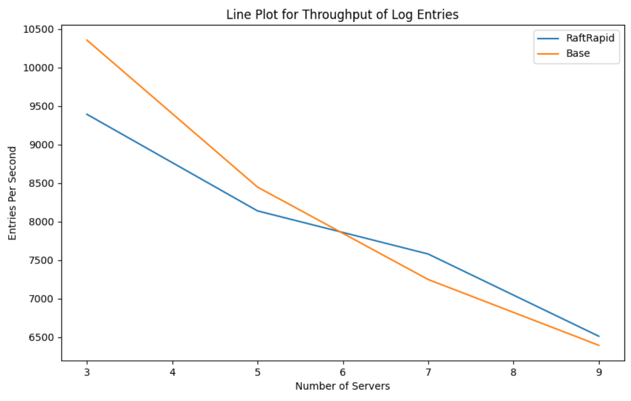
(d) Latency



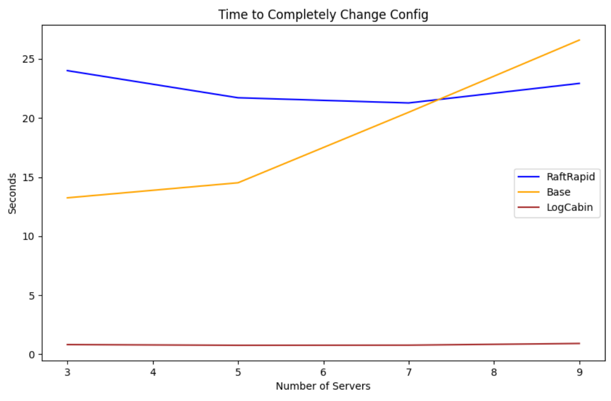
(e) Time to detect and replace a crashed leader



(f) Time for network partition healing to elected leader



(g) Log entry throughput



(h) Time to switch to a new config of all new servers