Comparison between Distributed Locking Service Implemented on Raft and Simple Paxos

Junkai Liang[1] and Chieh Nien[1]

[1]Computer Science Department, Viterbi School of Engineering, University of Southern California

Abstract

We implemented a distributed locking service based on a distributed key-value store, using different consensus algorithms (Raft and Simple Paxos) at the core. We conducted a comparison of performance, scalability, and fault tolerance between our distributed locking service running on Raft and Paxos to understand the impact of consensus protocols and the differences between Raft and Paxos. Using the same upper-level key-value store implementation for the locking service, the Raft-based version consistently outperformed the Paxos-based version in all aspects. We believe this is due to three main reasons: 1) Raft's strong leader structure is more compatible with the locking service scenario; 2) Raft requires only one bidirectional communication between the leader and followers, whereas Paxos requires three for each decided operation; 3) Raft can batch several operations at once, while Simple Paxos needs to send each operation individually.

Keywords: Locking Service, Distributed System, Raft, Paxos, Network.

1. Overview

The evolution of distributed systems has been characterized by the growing complexity of modern applications and the demand for scalable, reliable, and fault-tolerant infrastructure. Distributed systems have transitioned from simple client-server architectures to highly distributed and decentralized environments, driven by the needs of web-scale applications and cloud computing.In this context, robust coordination and synchronization mechanisms have become a pivotal component in the distributed systems landscape.

However, locks or mutexes that work well on single machines often face significant challenges when applied to distributed systems. Intrigued by this problem, we explored it further. Our research led us to Chubby [3], a locking service developed by Google in 2006 to ensure that multiple processes running on different machines could coordinate effectively . At the time, the predominant consensus algorithm was Paxos [1, 2], introduced by Leslie Lamport in 1998 . Chubby was built on an improved version of Paxos.

However, in 2014, Diego Ongaro and John Ousterhout introduced a new consensus algorithm called Raft [5]. Known for its simplicity and ease of implementation, Raft quickly gained popularity and is now widely used. This led us to ask: would a locking service inspired by Chubby, but built on Raft instead of Paxos, perform better due to the 16-year gap in development? To answer this question, we compared performance, scalability, and fault tolerance in systems using both consensus algorithms and found interesting results.

Raft outperformed Simple Paxos in all key areas. Regarding performance, the Raft version is approximately twice as fast as the Paxos version when network latency is low, and about 1.5 times faster when network latency becomes the dominant factor. This is because Raft requires only one bidirectional communication between the leader and followers—excluding communication

between client and leader—whereas Paxos requires three communications among servers for each operation, two of which must be completed before the proposer can reply to the client. Consequently, the client's waiting time in Raft is approximately two round trips, compared to three round trips for Paxos.

When it comes to scalability, both algorithms scale well as the number of servers increases, provided that network bandwidth does not become a bottleneck. However, Raft performs slightly better than Paxos as the number of clients grows, because Raft can batch operations from different clients into a single communication.

In terms of fault tolerance, both algorithms can continue operating as long as fewer than half the servers are down, and they can quickly resume operations even if the leader[1] fails . According to the quorum rule, neither algorithm can make progress if at least half of the servers are down. However, once the number of functioning servers returns to at least half, operations sent during the down period can be processed correctly.

We also observed that Raft handles both leader and non-leader failures more efficiently. For leader failures, Raft uses heartbeats to monitor the leader's status and allows followers to re-elect a new leader quickly if the current one fails. In contrast, Paxos acceptors must wait for a timeout to start a new proposal even if the proposer fails early, and the timeout is relatively long to avoid unnecessary contentions. For non-leader failures, Raft can send multiple operations in one communication to help restarted followers catch up, while Paxos must handle each operation individually. Additionally, later Paxos instances may need to wait for earlier ones to complete, even if the former were decided more quickly.

The rest of this report is organized as follows: In Section 2, we discuss the background and implementation details, covering the RPC library, the Paxos protocol, the Raft protocol, and the key-value store that supports the locking service. Section 3 provides an overview of the methodology. In Section 4, we present the evaluation settings and results. Finally, Section 5 outlines the challenges we encountered and possible directions for future work.

2. Background and Implementation Details

In this section, we will discuss the background of our project and outline the design choices we made during implementation.

2.1 RPC Library

Our project was developed in Golang, which comes with a built-in RPC library. However, we chose not to use this built-in library for the following reasons. Due to setup constraints, we need to run multiple client instances and server instances on a single machine. Using the built-in RPC library would utilize the local loop network, which makes controlling connections and latency difficult. Instead, we decided to use a simulated network that provides an interface similar to RPC. This approach allows us to easily add or remove instances from the network, as well as adjust parameters to simulate network instability by altering latency and packet loss rates.

The downside to this solution is that the routing logic, typically handled by the Network Interface Card (NIC), is now managed by the CPU. Additionally, a number of goroutines[2] used to simulate network requests in transfer are left idling in the background. However, during our tests, we observed that CPU and memory usage remained within acceptable limits, suggesting that these drawbacks would not significantly impact our results.

[1] In Paxos, there is no designated leader; however, in our implementation, all clients tend to connect to the same server to reduce competition among proposals. We consider this server the leader of the Paxos cluster.

[2] Lightweight threads in Golang

2.2 Locking Service Application

The locking service application that we built upon consensus algorithms is relatively simple. Although our project was inspired by Chubby [3], which supports complex file system-like operations, we decided to implement only exclusive locks. Clients can send five types of RPC calls to the current leader (the server that processed its last request): `Create`, `Remove`, `Acquire`, `Release`, and `Extend`. These calls respectively allow for creating a lock, deleting a lock, attempting to acquire an exclusive lock, releasing an acquired lock, and extending the lease for a holding lock (we'll discuss lease details later). Each server maintains a hash table mapping filenames to the metadata of their respective lock files. Initially, we used the built-in Golang hash table and found it was not a performance bottleneck during tests, so we continued with that implementation.

Servers maintain an operation log in a consistent order, which ensures a consistent state through Paxos or Raft, as discussed in the following subsections. This mechanism allows the application to avoid issues related to replicated or out-of-order requests. Clients communicate with servers through `Session`; each `Session` maintains connections to servers, keeps track of holding locks, and is responsible for extending leases.

In the context of distributed systems, various types of failures must be considered, including client failures. If a client with an associated `Session` holding a lock fails, it will no longer be able to release the lock. This necessitates a mechanism for servers to revoke these locks. Chubby employs a `KeepAlive` technique, where the server blocks RPC calls and responds only when the session is about to time out. We chose not to implement this design due to the potential for numerous blocked requests on the leader, which could create a significant burden.

Instead, we opted for a "lease" design. Locks are always granted with a lease, with each `Session` tracking its holding locks and their corresponding leases. When a lease is about to expire, it sends an `Extend` request to the server. If the server does not receive the `Extend` request within a grace period after the lease expiration, it revokes the lock. The `Extend` operation is similar to `Acquire` but with an important distinction: it only succeeds if the session currently holds the lock, while `Acquire` always succeeds if the lock is idle. This lease-based approach and the extending mechanism are handled by `Session`, making it completely transparent to clients, which only interact with `Session` interface.

2.3 Paxos - Consensus Algorithm

One of our versions uses Paxos to maintain a consistent operation log. Each entry in the log is treated as a Paxos instance. Whenever a server receives a request, it sends the operation to the Paxos protocol as the latest log entry. The server then periodically checks the status of the instance until it is decided. If the decided result does not match the original entry, the server resends it.

We use the simplest implementation of Paxos [1, 2], which includes three stages: propose, accept, and decide (also known as prepare, promise, and accept). This setup requires at least three bidirectional communications among servers for each operation. Although the proposer can reply to the client after the second phase, this is still a key reason why Paxos tends to be less efficient than Raft in our application.

Furthermore, if an acceptor makes a promise but the instance remains undecided for a certain period, it will repropose. To minimize contention among live proposals, we set the timeout relatively high. While this configuration enhances overall performance when there are no failures, it can lead to increased delays in the event of a leader failure.

It's worth acknowledging that there are numerous improved versions of Paxos. When Google developed Chubby, they made significant enhancements to Paxos. In the same year that Chubby was published, Lamport also introduced "Fast Paxos" [4] , a variant that allows values to be learned in two message delays, which addresses a critical bottleneck in our application. However,

due to limitations in time and resources, we chose to compare only the basic version of Paxos in our project.

2.4 Raft - Consensus Algorithm

The other version uses Raft to maintain the log. Our implementation strictly adheres to the RPC definitions outlined in the original Raft paper [5]. As previously mentioned, once a leader is elected, each operation requires only one bidirectional communication among servers. Additionally, once the leader receives replies from more than half of the followers, it sends the operation back through Golang Channels. This approach, compared to Paxos, reduces overhead and decreases waiting time before each log entry is decided.

Raft also offers the ability for the leader to send a batch of log entries in a single RPC, helping a restarted follower catch up more quickly after a failure and reducing the impact of an unreliable network, which might drop requests. The use of heartbeats in Raft also allows followers to detect a leader's failure sooner compared to Paxos, contributing to Raft's better performance in tests involving leader failures.

One downside of Raft is that all client requests, especially write requests, must pass through the leader, which may pose challenges for load balancing compared to Paxos. However, in Chubby, even though utilized a Paxos-like protocol, forced all requests through a single "leader". This centralized handling of requests was essential for ensuring correctness in the locking service, prioritizing correctness over performance. Therefore, in the context where correctness is paramount, Raft emerges as a better choice inherently.

3. Methodology

Our evaluation centers on three key aspects: performance, scalability, and fault tolerance, as well as the differences between the two versions.

We divide our tests into two segments: speed tests, focusing on performance and scalability; and failure tests, focusing on fault tolerance.

3.1 Speed Tests

In this part, we focus on performance under various network settings without any failures, as well as scalability as the number of servers and clients increases.

3.1.1 Test Suites

We have three test suites:

`SingleClient`: In this test, a single client acquires a lock and then releases it, repeating this process sequentially 1000 times.

`MultipleClientsParallel`: This test involves multiple clients, each acquiring a unique lock and then releasing it. There is no contention because each client operates on a different lock. The test concludes after all clients collectively acquire and release locks 1000 times.

`MultipleClientsContention`: In this suite, multiple clients all attempt to acquire the same lock. Since only one client can hold the lock at a time, each client acquires and then releases the lock in turn. The test completes once the lock has been acquired and released 1000 times.

3.1.2 Network Settings

We have different network settings:

Delay: We have two delay settings: short delay and long delay. By default, the short delay is used. In this setting, each RPC call may experience a uniformly random waiting time between

0 and 20 ms in both directions, simulating real-world transfer time. The long delay setting, on the other hand, involves a uniformly random waiting time between 0 and 100 ms.

Reliability: We have two reliability settings: reliable and unreliable. The default setting is reliable, where all requests are guaranteed to reach their destination, and all replies are guaranteed to return to the sender. In the unreliable network setting, there is a 10 percent chance that a request will be dropped before reaching its destination, and a 10 percent chance that a reply will be dropped. Both situations result in a timeout from the sender's perspective, leading the sender to typically resend the request. Please note that in both network settings, we do not guarantee the order in which requests arrive.

In `SingleClient` test suite, we apply different delay settings within a reliable network to compare the outcomes when latency becomes the dominant factor.

In `MultipleClientsParallel` test suite, we apply different reliability settings within a short delay network to test the functionality when requests may be dropped and evaluate the impact of such drops.

In `MultipleClientsContention` test suite, we only use the default setting, which is a short-delay reliable network. This choice allows us to simulate the highest level of contention.

3.1.3 Scalability

We also conduct tests to observe trends as the number of clients or servers increases.

In `SingleClient` test suite, we vary the number of servers, running tests with 3, 5, 7, 9, and 11 servers respectively. For the other two test suites involving multiple clients, we fix the number of servers at 5, which is a common configuration in practice.

In both `MultipleClientsParallel` and `MultipleClientsContention` test suites, we test with 5, 10, 20, and 40 clients respectively.

3.1.4 Measurements

We use total completion time as the metric for performance, and we compare completion time with varying numbers of clients or servers to assess scalability. For each setting in each test suite, we conduct 5 identical runs and use the average to minimize errors and ensure consistent results.

3.2 Failure Tests

In this part, we focus on the functionality and performance when some of the servers fail. We maintain the network setting as a short-delay reliable network, and the number of servers is fixed at 5 for this part.

3.2.1 Test Suites

The test suites are similar to those in the speed tests. We inherit the test suites involving multiple clients.

The difference in `MultipleClientsParallel` suite is that each client must perform 1,000 lock acquisitions individually, rather than 1,000 in total, providing enough time to gather data and observe trends. `MultipleClientsContention` suite remains unchanged.

Additionally, we introduced `ClientFailure` test suite to evaluate scenarios involving client failures.

3.2.2 Failures

We simulate two kinds of server failures for our tests:

`LeaderFailure`: Every 12 seconds, we shut down the current leader. To maintain more than half of the servers operational, we restart the server we last shut down. This ensures that 4 out of 5 servers remain running.

`NonLeaderFailure`: In the first phase, we shut down a non-leader server every 3 seconds. After 12 seconds, all 4 non-leader servers will have been shut down. In the second phase, we restart a server every 3 seconds, until all servers are back online after 12 seconds. We then repeat these two phases.

Due to space constraints, we will only present the `LeaderFailure` results for `MultipleClientsContention` test suite. However, results for both `LeaderFailure` and `NonLeaderFailure` in the `MultipleClientsParallel` test suite will be provided.

In `ClientFailure` suite, 20 clients attempt to acquire the same lock. The client that successfully acquires the lock crashes before it releases it. The test concludes after the lock is successfully acquired 100 times. This suite is designed to test the functionality of lease expiration and lock revoking.

3.2.3 Measurements

For server failure test suites, we first run a baseline test without any server failures, recording the number of operations conducted by all clients per second (ops). Subsequently, we run the test suites with various types of failures described above. We record the fluctuations in ops to check whether our locking service can automatically recover from the failures and observe the impact of these failures.

For the client failure test, we record the total completion time. If the lease mechanism is functioning correctly, the expected completion time should be

$$(t_{lease\_expiration} + t_{grace\_period}) \times 100. \tag{1}$$

Our results indicate that completion time for the Raft version is 1,883 seconds, compared to 1,905 seconds for the Paxos version. With a lease expiration time of 12 seconds and a grace period of 6 seconds, this demonstrates that our system is robust enough to handle client failures.

4. Evaluation

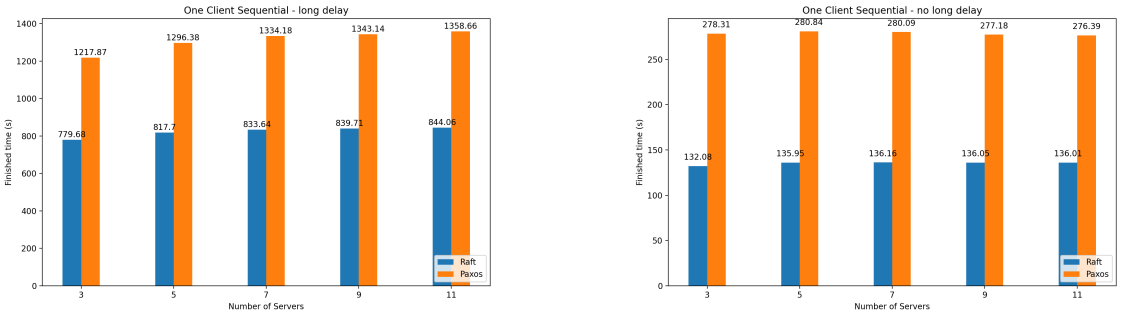
4.1 Test Environment

The tests were conducted on a virtual machine running Ubuntu 22.04.4 LTS within the Windows Subsystem for Linux (WSL). The system is powered by an AMD Ryzen 5700X CPU, with the virtual machine configured to use 16GB of memory.

4.2 Speed Test

4.2.1 SingleClient

The Raft and Paxos algorithms fundamentally incur different network overheads. Raft requires 2 RPCs (client to server, leader to follower) while Paxos requires 3 RPCs (client to server, prepare phase, and promise phase). As expected, Raft performs better with shorter finished time in both long delay and no long delay network setting (Figure 1). Besides RPC cost, Paxos requires additional communication cost - Raft will notify servers immediately after acheiving consensus, whereas Paxos relies on servers periodically querying Paxos to check for consensus. As shown in Figure 1 (b), Raft is almost 2x faster than Paxos considering both RPC cost and additional communicate cost. However, RPC cost dominates the finished time under long delay scenario, thus Raft becomes 1.5x faster than Paxos. (Figure 1 (a))

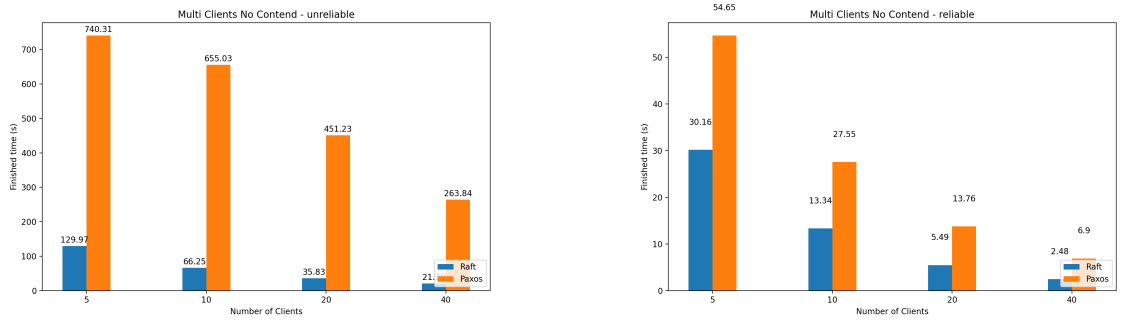


(a) (b)

Figure 1: Speed test of single client: (a) with long delay network setting (b) without long delay network setting

When increasing servers number, the increasing of finished time is not apparent since the communication between servers is parallel. In long delay setting (Figure 1 (a)), finished time only increase 1.08x - 1.1x when number of servers increase from 3 to 11, showing that RPC cost is the bottleneck in this experiment.

4.2.2 MultipleClientsParallel



(a) (b)

Figure 2: Speed test of multiple clients parallel: (a) with unreliable network setting (b) with reliable network setting

In MultipleClientsParallel experiment, as expected, the termination time decrease as the number of clients increase since both Raft and Paxos servers can handle multiple client requests simultaneously. As shown in Figure 2, when number of clients doubles, the completion time also half. Moreover, Raft performs better than Paxos as predicted due to their algorithm design.

In our implementation, since Raft can batch multiple requests in a single RPC call, it shows more noticeable improvement as the number of clients grows. Raft's performance increases by 12x, whereas Paxos only improves by 8x when the number of clients increases by 8x (Figure 2 (b)).

Interestingly, unreliable network setting has a greater impact on Paxos - when there are only five clients, the completion time of Paxos increases by 13.7 times, while Raft only increases by 4.3 times (Figure 2 (a)). We believe this is because Paxos requires individual resolutions for each request, and any dropped requests during the prepare, promise, or accept phases could potentially result in the failure of the consensus.

4.2.3 MultipleClientsContention

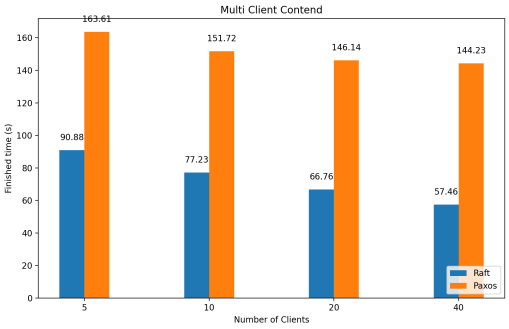
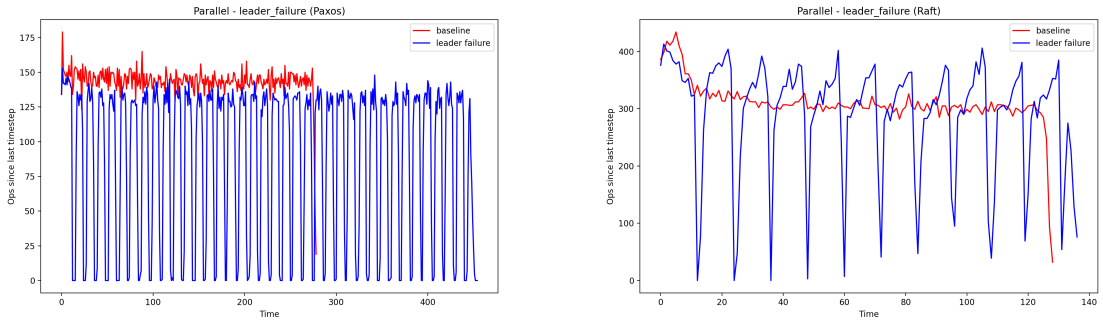


Figure 3: Speed test of multiple clients contention

Under conditions of client contention, augmenting the client count yields marginal enhancements in performance. Comparing Figure 2 (b) and 3, it is evident that under non-contention conditions, the throughput of Raft and Paxos increases by factors of 13.7 and 4.3, respectively, when the number of clients is multiplied by eight. However, under contention circumstances, Raft and Paxos only experience modest improvements of 1.58 and 1.13, respectively. This is because the only area where performance can be enhanced as the number of clients increases is when a release request coincides with an incoming acquire request, allowing the acquire request to be promptly fulfilled.

4.3 Failure Test

4.3.1 MultipleClientsParallel with LeaderFailure



(a) (b)

Figure 4: Failure test of leader failure with multiple clients parallel: (a) Paxos (b) Raft

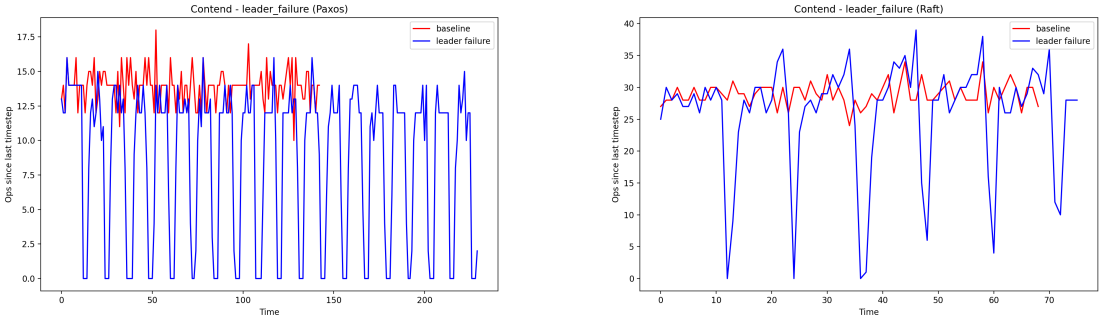
Experimental findings demonstrate that even in the event of leader failure, our system continues to operate smoothly. As depicted in the Figure 4, for every 12 seconds, operations are required to await the election of a new leader before proceeding.

As discussed earlier, Raft's performance tends to surpass that of Paxos. Analysis of the experimental results reveals that in the event of leader failure, Paxos experiences an approximately 1.5-fold delay before completion, whereas Raft encounters only around a 1.01-fold delay.

Furthermore, we have observed an intriguing phenomenon: upon the generation of a new leader, the operational throughput in Raft momentarily surpasses the baseline throughput. This can be attributed to the process of leader failure, where although some operations remain in-

complete, they have already been partially synchronized with some servers. Consequently, upon the emergence of a new leader, completing these half-way operations becomes faster.

4.3.2 MultipleClientsContention with LeaderFailure

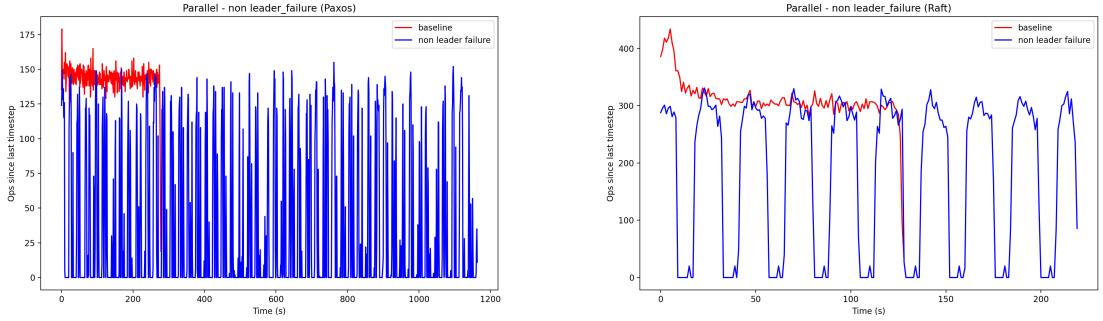


(a) (b)

Figure 5: Failure test of leader failure with multiple clients contention: (a) Paxos (b) Raft

Our system ensures that even in the scenario of multiple clients competing for the same lock and a leader failure, system functionality remains operational. In this scenario, as expected, Raft's performance still surpasses Paxos (Figure 5). Additionally, it can be observed that Paxos requires more time to recover after failure. The reason is that Paxos, in order to ensure the sequence of operations, must ensure that all preceding operations are decided before proceeding with new operations. Each preceding operation is independent and has its own waiting time. However, after electing a new leader, Raft will have the new leader batch all operations that occurred during the failure period and synchronously distribute them to all followers at once.

4.3.3 MultipleClientsParallel with NonLeaderFailure



(a) (b)

Figure 6: Failure test of non leader failure with multiple clients parallel: (a) Paxos (b) Raft

This experiment sheds light on the role of 'majority vote' in consensus algorithms. From Figure 6 (b), it can be observed that when 3 or 4 servers are down, due to the inability to achieve a majority, operations cannot proceed. It's interesting to note that when only two servers are operational, a small peak occurs. This is possibly because some operations were synchronized with the some servers that have now failed, but a majority was not reached at that time. When the second server restarts and achieves the majority, the operations can proceed. For example, consider a scenario with five servers, where only the leader and server 4 are alive. After server

4 goes down and server 1 restarts, operations that have already synchronized with server 4 will synchronize with server 1 and achieve a majority.

Another interesting phenomenon is that when only three servers are operational, the operation rate is slightly lower than the baseline. We hypothesize that this is because the baseline only needs to wait for responses from the fastest two servers to decide. However, when there are only three servers, the two servers other than the leader may be the slowest among all servers.

5. Challenges

We encountered several obstacles during our project. Initially, our plan was to leverage existing Raft and Paxos libraries to implement the locking service. However, we discovered that most libraries encompassed more than just the consensus protocols, introducing various overheads and errors into our tests. Additionally, it was challenging to find and use the pure API for consensus protocols.

Therefore, we decided to implement Paxos and Raft from scratch. Debugging consensus algorithms is never easy, and we spent a significant amount of time on this. However, this endeavor allowed us to thoroughly grasp every detail of our implementation and explain every phenomenon we observed in our results.

Our project also had some shortcomings. We were ambitious to implement Chubby at first, but due to its complexity and lack of implementation details, we had to simplify many functionalities.

Regarding the implementation of Paxos and Raft, the persistence of logs on durable devices is crucial to withstand sudden failures. However, as the log grew, the persistence process became a bottleneck, overshadowing all other factors. To obtain current results, we had to omit persistence. This was possible because the crashes in our tests were controlled shutdowns, allowing us to persist logs only before shutting down the server. This problem should be solved through log compaction, which we implemented in Paxos and Raft, but we didn't have sufficient time to incorporate this functionality into the upper-level application. This is something we could address in future work.

Despite these challenges, we gained a deep understanding of the Paxos and Raft algorithms, as well as distributed systems in general. We also acquired significant experience in debugging complex programs. Most importantly, we had the opportunity to explore an intriguing direction and conduct research in that area.

References

[1] Leslie Lamport. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems 16, 2 (May 1998)* (May 1998). ACM SIGOPS Hall of Fame Award in 2012, pp. 133–169. URL: https://www.microsoft.com/en-us/research/publication/part-time-parliament/.

[2] Leslie Lamport. "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[3] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 335–350.

[4] Leslie Lamport. "Fast paxos". In: *Distributed Computing* 19 (2006), pp. 79–103.

[5] Diego Ongaro & John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319.