50.041 Distributed Systems and Computing
Final Project Report

Chubbz

Group 8

Prepared by:
Foo Chuan Shao (1005549)
Jon-Taylor Lim (1005053)
Sim Wang Lin (1005030)
Shaun Hin (1005446)
Jyotit Kaushal (1005245)

Table of Contents

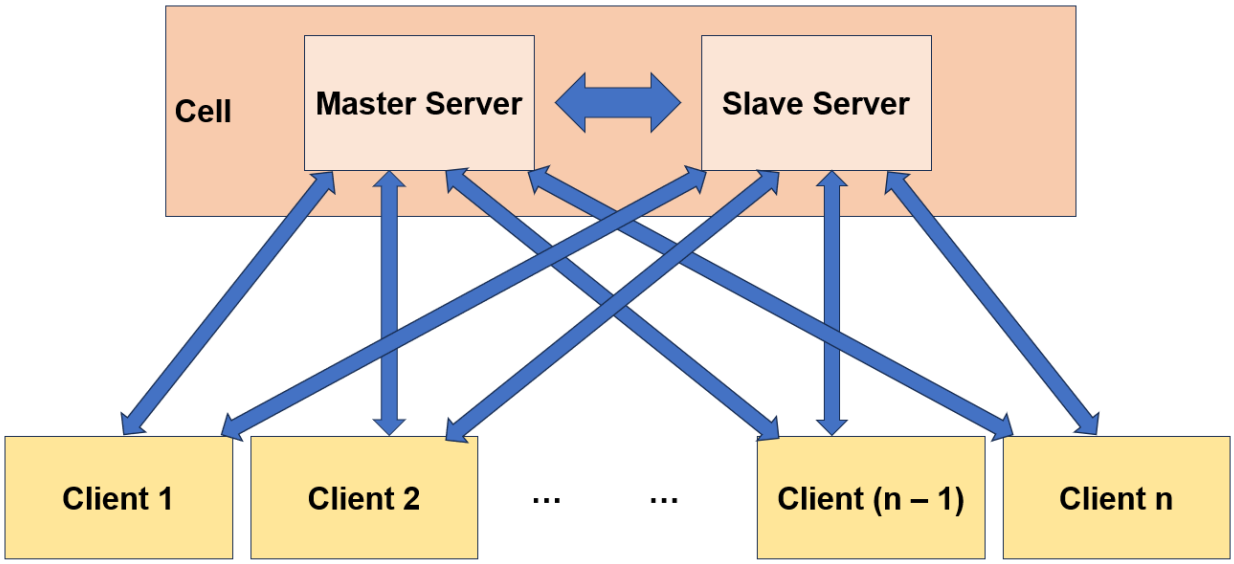
Introduction

The distributed computing environment presents unique challenges in terms of maintaining data consistency and coherency. A lock service, such as Google's Chubby, is essential to ensure that multiple machines can safely access shared resources. This report delves into the design considerations for our own implementation, named “Chubbz”, focusing on reliability, consistency, fault tolerance, and use-case scenarios.

Background

Chubby is a lock service designed by Google to provide coarse-grained locking and reliable storage for loosely-coupled distributed systems. While its interface resembles a distributed file system with advisory locks, Chubby emphasizes availability and reliability over high performance.

System Architecture



"Chubbz" loosely follows the implementation of Chubby by having a cell of servers to handle lock requests and other messages from the clients. In our implementation, there are only 2 servers, namely the master server and slave server to determine the behaviour of the lock service in terms of correctness, scalability, and fault tolerance which will be discussed further below. The master server is the leader that all clients connect to and communicate with, whereas the slave server is the backup server to take over the role of the master server in the case that the master server goes down. In line with Chubby, each cell has only 1 master server and at most 4 other replica slave servers. The detailed features and analysis of our lock service is further explored in the following sections.

System Design

Features

The table provided offers a succinct, at-a-glance overview of the key features of our distributed lock service system. It encapsulates the essential aspects that define the system's functionality and performance.

Feature	Details
Core Locking Service	Singular Lock to be managed by distributed servers. Only one lock to be granted and collected back after clients use
Data Consistency	Consensus Protocols: Replicated state machine Atomic Operations: Operations like acquiring or releasing locks are atomic to prevent inconsistencies using Mutex
Failover Procedures	Failure Detection: Mechanisms to detect node failures, such as heartbeat check polling mechanisms Backup Leaders: Backup server for quick failover Recovery Mechanisms: Master state recovery after restart. Upon recovery, recovered node is to take reference to leaders metadata
Communication Protocols	Message Passing: Asynchronous communication through RPC function calls and TCP connection messages with timeout Message Types: Different message types for lock requests, releases, heartbeats, acknowledgement

Intermittent and Permanent Fault Tolerance	Retry Mechanisms: Retry logic on client side Node Replacement: Backup server takes over from where the breakdown occured
Data Replication	Replication Strategy: Replication of full metadata each time with acknowledgement Consistency Model: Strong & Sequential consistency; replicate on server change state

Correctness

In the implementation, the lock for a particular resource is granted only by the leader server. At any point in time, only one client can acquire the lock for that resource. The lock is released by the client to the leader. Only then may the next client acquire the lock.

Although the leader or replica may experience faults at any time, the implementation necessitates that the leader handles all request and release messages, and responds only after data has been successfully replicated on the backup leaders. This ensures that any leader taking over is at least as updated as the previous leader who went down such that the state of replicas is the latest observed state of the leader before it experiences fault, hence **strong consistency** of the lock granting service is observed such that there can only be one client or the leader node itself holding onto the lock at any point in time **guaranteeing the correctness** of the implementation.

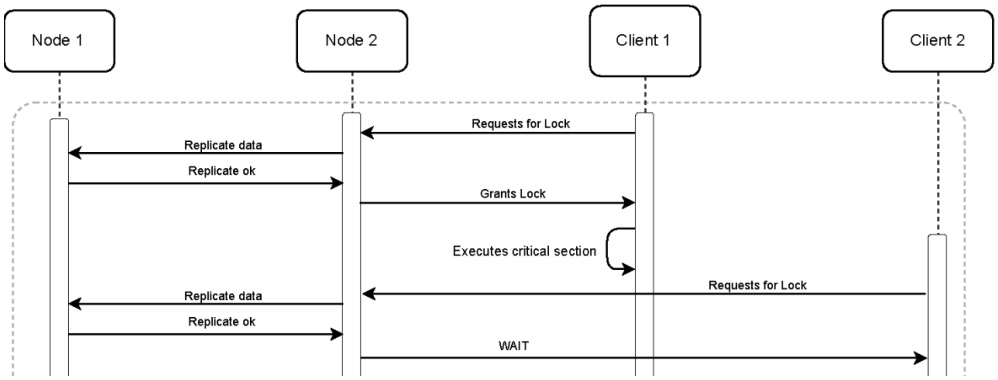


Figure 1: Strong Consistency Protocol

Our system implements **sequential consistency** through the help of a queue, ensuring that all nodes in the distributed environment process requests in a sequential manner. When a request, such as a lock acquisition or release, is made, it is placed into this queue. The leader server then processes these requests in the order they were received. In the event of a leader node failure, from our strong consistency and replication protocol, the backup node that takes over has the latest state, preserving the sequential order of operations. This methodical processing and replication ensure that the operations on all nodes reflect a single, consistent order of execution. By strictly adhering to this mechanism, our system effectively maintains an order of operations across its distributed architecture, thereby upholding the principles of **sequential consistency**.

Scalability

The simulation results for our lock server implementation reveal a linear relationship between the processing time and the number of clients simultaneously requesting locks. This suggests that in practical scenarios, the server can efficiently manage a moderate increase in client numbers with minimal impact on response time.

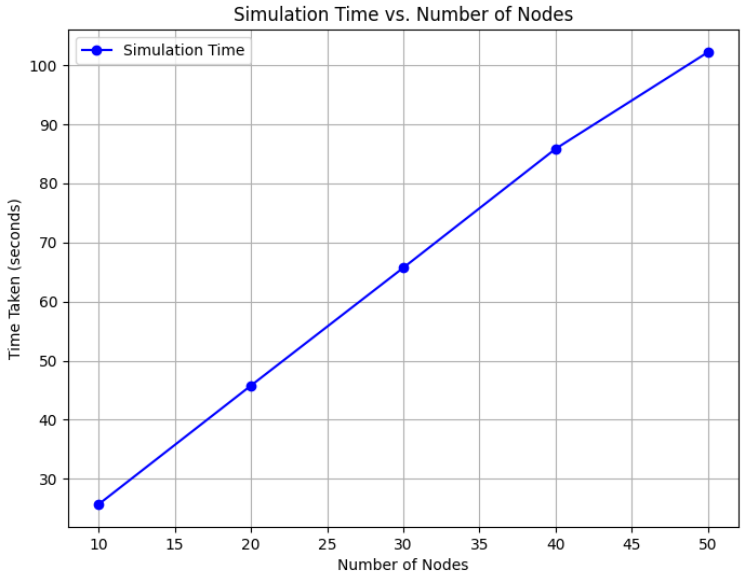


Figure 2: Simulation Time vs Number of Nodes graph

In a simulated worst-case scenario, where only one server is operational and it encounters a fault, our current system does not automatically reload the most recent saved state from the disk. This is because our design assumes that there's always at least one functioning server, as the likelihood of all replicas and the leader failing simultaneously is extremely low. However, in a hypothetical scenario where the sole operational server restarts, becomes the leader, and the only active server, it would load the latest state from the disk. Even in this situation, the system's scalability remains robust. The primary additional overhead is the server's restart and disk loading time, which is relatively constant. Therefore, as the number of clients increases and even with multiple failures of this single server, the average time to process a request is expected to increase only polynomially. Furthermore, the time complexity is likely still within a linear factor for a small number of clients, $n < 100$ based on extrapolation within the same magnitude from our testing of 10 to 50 clients.

Fault Tolerance

The fault tolerance of the implementation accounts for both intermittent and permanent faults such that the replica server waits for a health check to time out before declaring itself as the new leader and announcing this to all clients. Further for intermittent faults, as the previous master server restarts, the original replica server would continue to be the leader to minimize disruption to the service, and the restarted server would serve as the new backup server.

Health Check and Leader Election: The system employs a proactive health check mechanism where the replica server continuously monitors the health of the master server. If the master server fails (a permanent fault), this is detected through a timeout of the health check. Upon such detection, the replica server initiates a leader election process and, in the absence of the master server, declares itself as the new leader. This prompt transition minimizes downtime and ensures continuous operation, which is crucial for maintaining the system's availability.

Handling Intermittent Faults: In scenarios where the master server experiences intermittent faults, such as temporary network issues or brief downtimes, the system is designed to maintain operational continuity. During the master server's downtime, the replica server assumes the role of the leader, handling all requests and maintaining the system's state. Once the master server is back online, it reverts to a backup role. This role reversal is seamless, ensuring that there is no interruption in service and that data consistency is preserved across all nodes.

Replica Server’s Role in Fault Tolerance: The replica server is not just a passive standby but an active participant in the system's health monitoring. The replica server's 'slave poller' function, which checks the health of the leader every 5 seconds, is a critical component of this mechanism. This frequent polling allows for quick detection of any issues with the master server and a swift response to maintain system stability.

Client-Side Retry Logic: On the client side, the system incorporates a retry mechanism to handle intermittent network issues. If a client fails to establish a connection with the leader server, it automatically attempts to reconnect, with a specified maximum number of tries and a timeout for each attempt. If these attempts fail, the client then redirects its request to the backup server, which would have assumed the leader role by this time. This feature ensures that client requests are not lost and are handled efficiently, even in the face of network instability.

Data Replication and State Recovery: Data replication and state recovery mechanisms further bolster the system's fault tolerance. The system replicates full metadata with each change, ensuring that the backup server always has the most up-to-date information. In the event of a master server failure and subsequent recovery, the system ensures that the recovered node syncs with the current leader's metadata, thereby preserving data consistency and state coherency.

Limitations

One of the limitations in our lock server management system, akin to Chubby, revolves around the absence of periodic saving of the replicated state to a remote shared disk. This limitation is particularly noteworthy in scenarios where all servers are simultaneously down, rendering the latest known state inaccessible. Unlike some distributed systems that incorporate periodic checkpoints to remote shared storage for fault tolerance, our system lacks this specific mechanism essentially due to the small-scale nature of this implementation owing to the limited time available for development.

One other way however, to explain this limitation can be rooted in a scenario where the replicated backup or slave server is hosted in-house. In this scenario, we make the assumption that at least one server will remain operational at all times. This assumption aligns with the model of Chubby's cell system, serving as the basis for our implementation (referred to as "Chubbz"). In essence, the design assumes that even in the event of failures, there will be at least one surviving server capable of preserving the system state.

A potential avenue for future improvement is the implementation of a mechanism to periodically save the system state outside the realm of strong consistency. By introducing periodic checkpoints or snapshots of the system state to an external storage medium, we could enhance fault tolerance in the worst-case scenario where all servers are down. This improvement would ensure that, even in the absence of operational servers, the system can recover from the latest known state saved externally. This approach aligns with the broader goal of fortifying the system against total failures and addressing the potential data loss associated with simultaneous server outages.

Another limitation is that we have configured the cells to be of a maximum of 2 servers. The reason this was implemented in this manner was that the original Chubby that we relied on was intended to hold only a small subset of servers. Therefore, aligning with the goals, to simplify the system for the servers, we implemented a simple failover procedure where the backup server just takes over upon detecting a failure. If the server comes back up, it will behave as the backup and continue as accordingly. In this case, as there is no election implemented, if >2 backup servers detect faults, it may cause problems with consistency due to confusion of who should be the leader and network partition.

A potential avenue to solve this would be to implement a leader elections scenario. Simple algorithms such as Bully, Ring, and Voting could be used for the election of the leader. However, do note that implementing would cause more overhead and time for message passing and failure procedure during faults.

Experiments & Tests

5 different simulations were conducted to determine the fault tolerance and performance of the implementation. These scenarios were run using docker containers to simulate a networked environment with multiple virtual client and server services, and can be found in the Github repo linked in the Appendix. The 5 simulations are listed below.

Note: Node represents servers. Node 2 is the master server and node 1 is the slave server.

1. 10 clients requesting and releasing from 1 master server and 1 slave server **without fault.** This represents the normal working of the system where the clients are concurrently requesting for a lock from the master server and throughout the entire simulation both servers stay up.

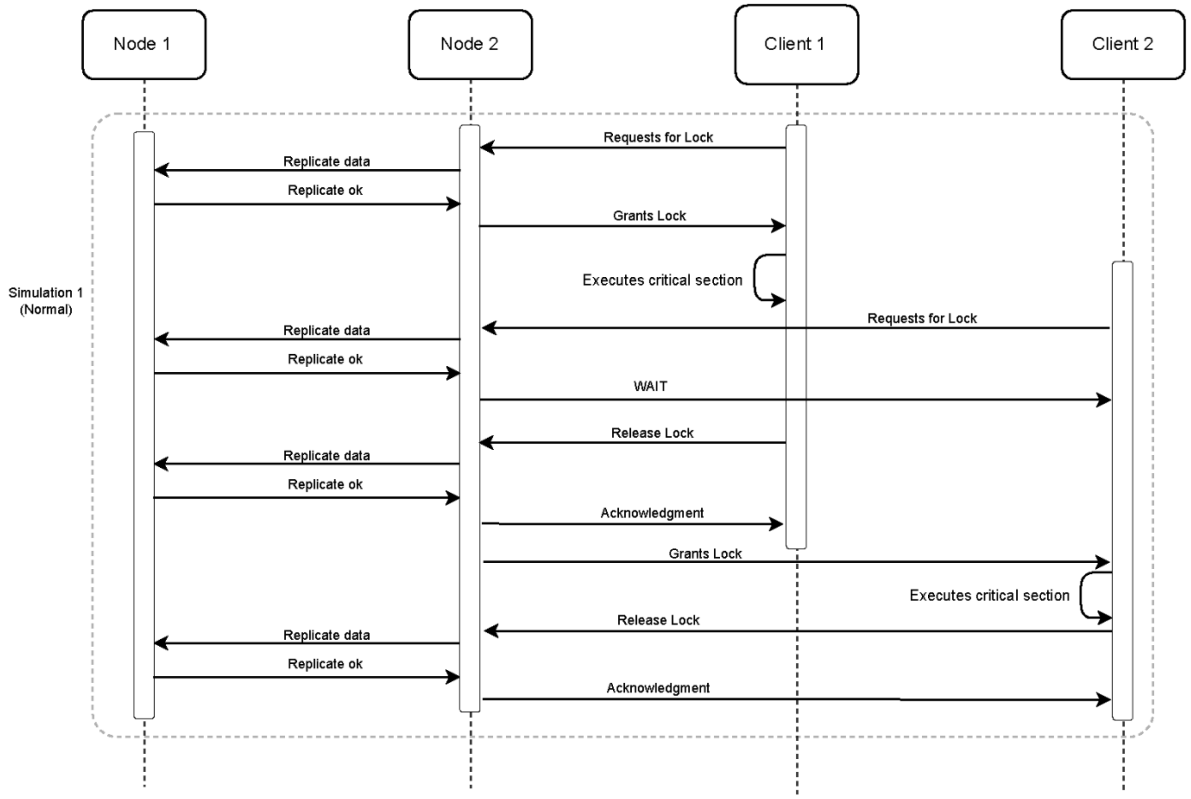


Figure 3: Simulation 1 sequence diagram

2. 1 client requesting and releasing from 1 master server and 1 slave server with 1 **intermittent fault** on the master server. In our simulation, we only simulate with 1 client to simplify the terminal output for easy debugging.

   This simulation demonstrates how an intermittent fault is handled by the system, where the master server goes down for a split second and the backup server does not take over. This experiment is aimed to test the retry mechanisms that we have proposed above for the clients to make re-requests to the server. As can be seen in the sequence diagram, the system works as "without fault" after node 2 is back up.

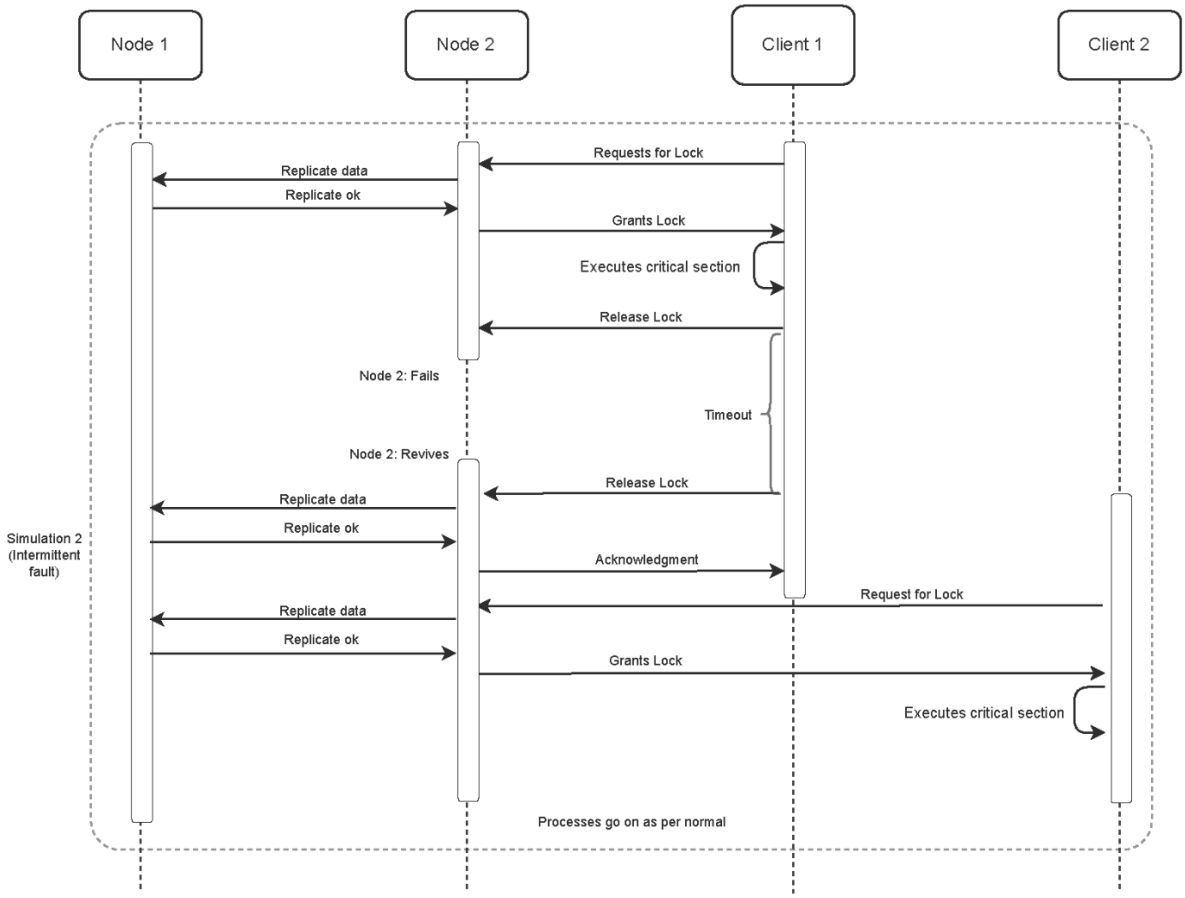


Figure 4: Simulation 2 sequence diagram

3. 10 clients requesting and releasing from 1 master server and 1 slave server with 1 **permanent fault** on the master server.

   This simulation demonstrates how a scenario of permanent fault tolerance is handled by the system, where the master server goes down permanently and the backup server elects itself as the leader and takes over. In the end, the backup server is the main and only server and all requests will go to that and work similar to the case of "without fault", but without data replication calls.

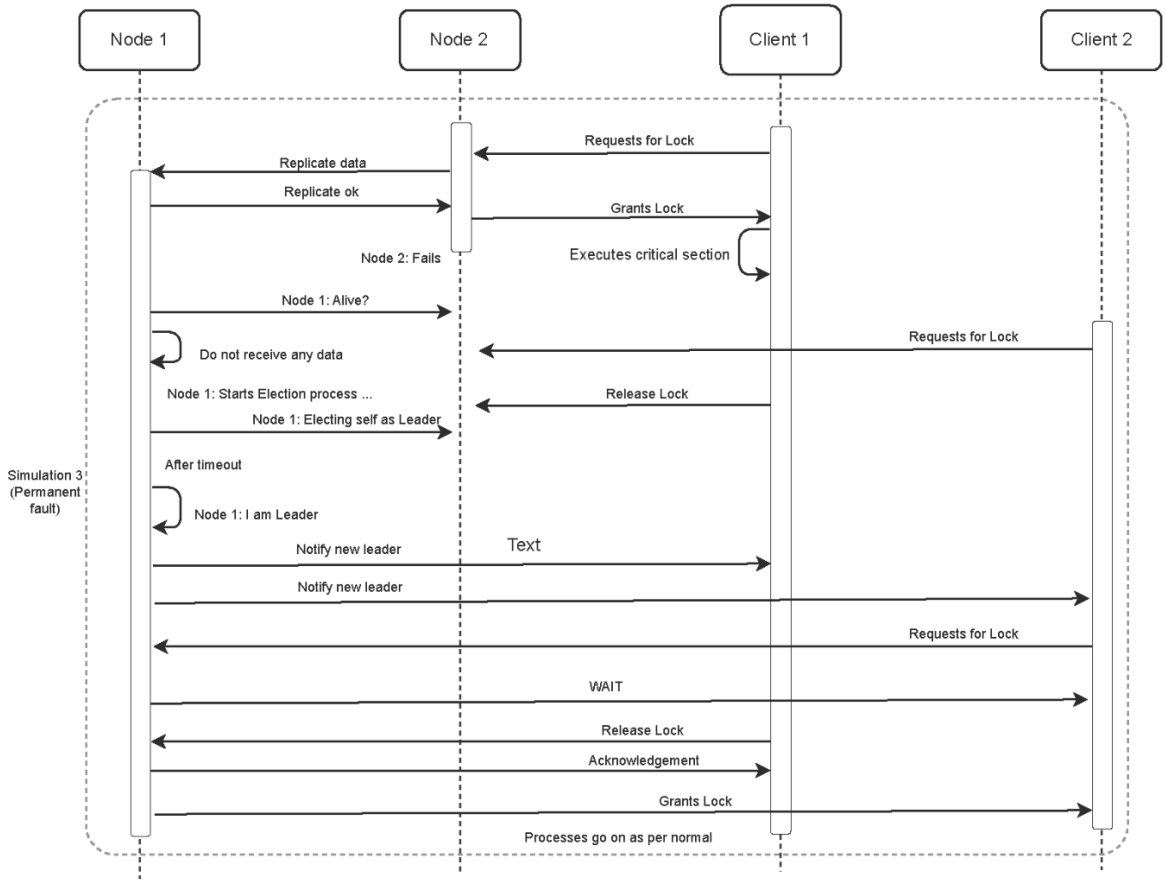


Figure 5: Simulation 3 sequence diagram

4. 10 clients requesting and releasing from 1 master server and 1 slave server with 1 **permanent fault** on the master server **AFTER** replication of data but before replying acknowledgement for release to the client.

   This simulation is a build up from the previous experiment and tackles the advanced case where the server goes down after the data has been replicated. As this case would cause Node 1 to be ahead in metadata compared to Node 2, we decided to test this and applied a patch to tackle this additional particular scenario.

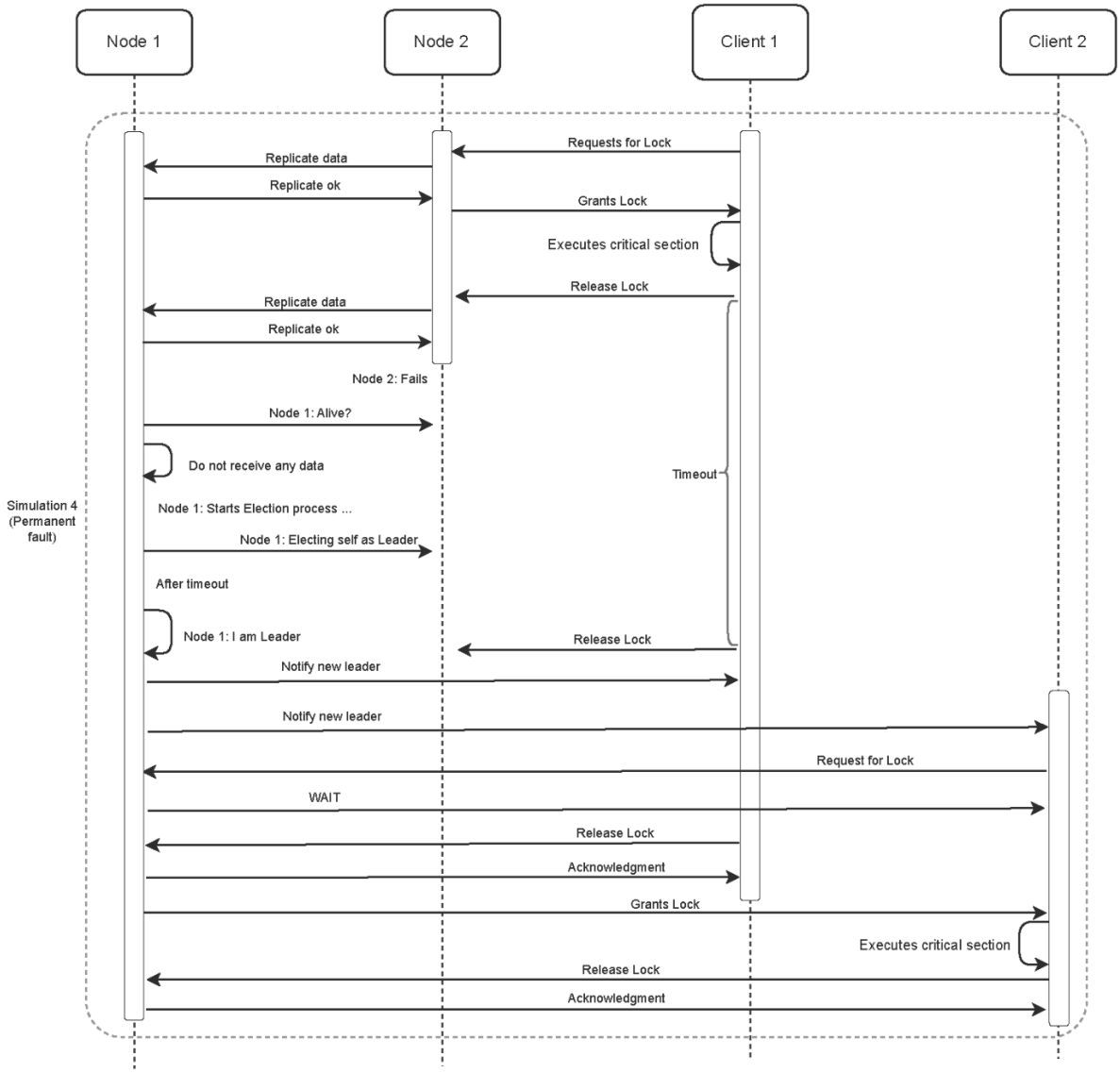


Figure 6: Simulation 4 sequence diagram

5. 10 clients requesting and releasing from 1 master server and 1 slave server with 1 intermittent fault on the master server after replicating data.

   Last but not least, this simulation demonstrates the extended second case of intermittent fault tolerance, specifically when Node 2 goes down for a longer duration, allowing Node 1 to take control, then subsequently coming back up. This tests for a fine balance between permanent and intermittent faults and how the system will manage it.

   When Node 2 comes back up, it establishes itself as the backup server rather than retaking leadership of the system. After this establishment of being alive, the operations will continue as “without faults”, with Node 1 being the master, and Node 2 being the backup.

   Please refer to the next page for the sequence diagram.

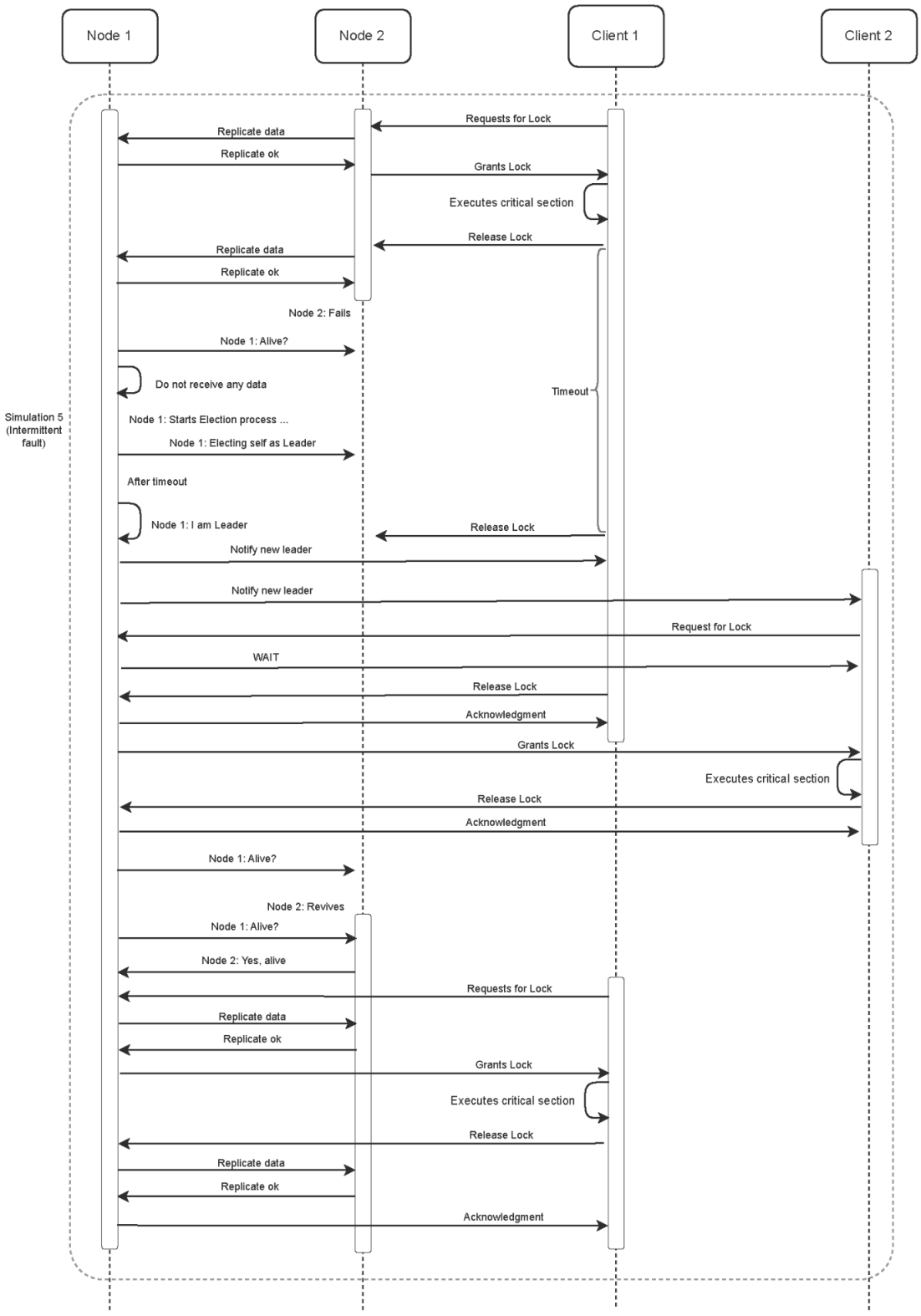


Figure 7: Simulation 5 sequence diagram

Conclusion

Overall, "Chubbz," our distributed lock service, has demonstrated its robustness and reliability in a variety of scenarios. Following the guidelines of Google's Chubby, "Chubbz" effectively maintains data consistency and coherency in a distributed computing environment, proving to be a valuable asset for managing shared resources across multiple machines. Our implementation, with a focus on simplicity and functionality, has shown its performance in terms of fault tolerance, scalability, and correctness. “Chubbz” prioritises safety through a careful balance between Consistency, Availability, and Partition Tolerance. In practice, the system places a higher emphasis on Consistency and Availability. This design choice ensures that the lock state remains uniform across the system at all times and that the service remains operational, even amidst server failures. For future work, incorporating external state saving mechanisms could further enhance the system's robustness, ensuring its effectiveness even in the most challenging scenarios.

Repository

https://github.com/s-hin-kansen/chubbyGo/tree/main