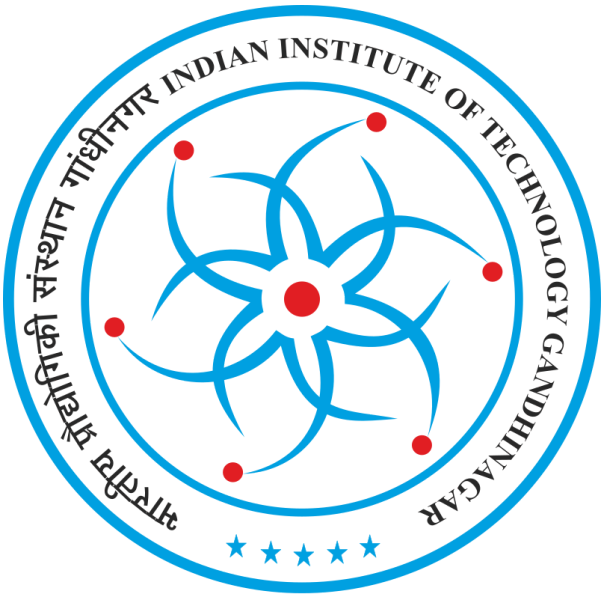
CS 616 Distributed Systems
Project Report

Distributed Lock

April 20, 2025

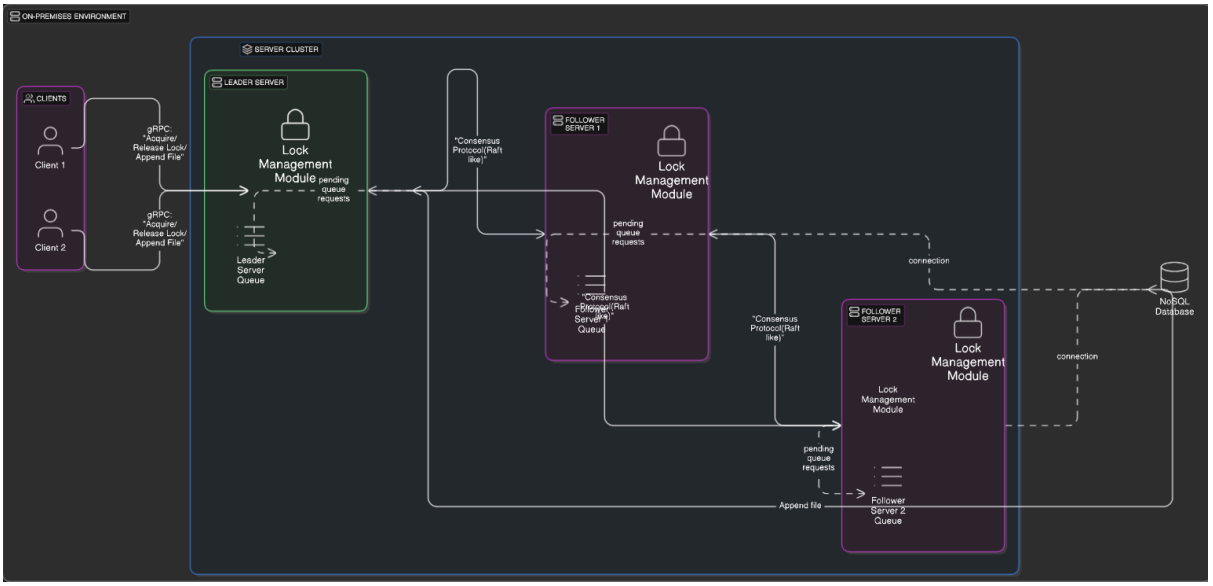


Kaushal Kothiya, 21110107 Anish Karnik, 21110098

1. Introduction

In this project, we implement a distributed lock service using a Raft-like consensus algorithm to ensure coordination among multiple nodes. The lock service is designed to serialize access to a shared file hosted on a file server. The system ensures bounded consistency and fault tolerance through leader election and replication across follower nodes.

2. System Architecture



The system comprises the following components:

- **Lock Server Cluster:** A set of nodes responsible for managing locks. One node acts as the *leader*, while the rest operate as *followers*.

- **Client:** Requests access to the lock and interacts with the leader to acquire, release, or use the lock.

- **File Server:** A centralized storage point that clients interact with only after acquiring the lock via the leader lock server.

Each client request is routed to the current leader. The leader handles lock state, replicates decisions to the followers, and maintains consistency within a bounded delay.

3. Consensus and Leader Election

A Raft-like algorithm is used to maintain consistency among the nodes:

- **Leader Election:** We use a heartbeat mechanism to detect leader failure. On failure, a new election is initialized, and a new leader is selected on the basis of who has the most recent data.

- **Replication:** Every lock acquisition or release request is appended to a replicated followers queue by the leader and propagated to all followers.

- **Bounded Consistency:** We ensure eventual agreement among nodes, allowing for temporary divergence, but all committed operations are reliably applied in the same order. The data in the follower would not be more than 20 seconds old.

4. Lock Management and Semantics

- **Acquisition:** Clients send a lock request to the leader. The leader grants the lock if it is free and appends this action to the queue.

- **Release:** Clients may voluntarily release the lock via another request. The release is similarly logged, andthe entry is popped from the queue and replicated to all followers.

- **Auto-Release:** If a client does not release the lock within 10 seconds, the system automatically frees it to prevent deadlocks.

- **File Access:** Clients may only issue a file append request if they hold the lock. The leader accepts these requests and are not permitted otherwise. All write requests go through the leader to the DB server.

5. Fault Tolerance and Recovery

- If the leader fails, a new leader is elected automatically.
- Using the recovery function, on startup, the followers catch up to the leaders' current entries.

This design enables our system to tolerate node failures while ensuring that the lock state remains consistent and available under typical operating conditions.

7. RPCs used

InitConnection	Initializes a client connection and assigns a unique client ID.
LockAcquire	Requests to acquire the lock; returns a stream of lock status updates.
LockRelease	Requests to release the currently held lock.
AppendFile	Appends data to a file if the client holds the lock.
GetQueueIndex	Retrieves the queue index for the requesting client.
UpdateLeader	Updates the current leader's IP address across nodes.
Heartbeat	Sends a heartbeat signal for leader liveness detection.
AddQueue	Adds a client to the lock request queue.
RemoveQueue	Removes a client from the queue upon lock release or timeout and sync with leader
GetQueueState	Returns the current state of the lock queue and holder information.

8. Conclusion

Our distributed lock service implements core concepts of distributed systems, including consensus, leader election, and replication. Using a Go-based Raft-like approach, we offer a practical solution for mutual exclusion with built-in fault tolerance and bounded consistency guarantees. The system is designed to be resilient, efficient, and applicable in real-world distributed coordination scenarios.