Simple Consensus System for Distributed Locks

ICT3 Project n°2 — Shanghai Jiao Tong University

Alexandra Baron Charles Pelong Maria Stivala Arnaud Brisset
Baptiste Halçaren Mathis Liens

October 2025

Abstract

This report presents the design and implementation of a simple consensus system for managing distributed locks using a leader-follower architecture. The implementation, realized in `Java` using sockets, guarantees strong consistency: all mutation operations (*lock*/*unlock*) are validated by the leader and then replicated to all followers, while read requests (*own*) are served locally by any server. We describe the project's objective, the methodology used, the system architecture, and the code structure. Finally, we present experimental results and discuss the system's limitations.

1 Context and Objective

As part of the ICT3 (Information and Communication Technologies) course, the objective was to **design a simple consensus system** meeting the following requirements: (i) one leader and multiple followers; (ii) a replicated *map* (key → owner) for the locks; (iii) multiple clients able to `tryLock`, `tryUnLock`, and `ownTheLock`; (iv) **all** mutation operations (`LOCK`/`UNLOCK`) must be routed to the leader; (v) read operations (`OWN`) can be served locally by followers; (vi) the leader updates its map and **propagates an update proposal** to all followers.

The core challenge is ensuring data consistency, guaranteeing that all servers have the same view of who owns which lock, especially during concurrent client requests.

2 Method and Implementation Choices

We opted for a **leader-follower architecture** built on Java TCP sockets. This approach was chosen for its explicit control over the network communication and its suitability for the project's requirements.

- **Reasoning (Technology Choice)**: Java was selected as suggested in the project subject. Its built-in socket libraries (`java.net`) and robust concurrency utilities (`java.util.concurrent`)

are well-suited for this task. The `ExecutorService` provides an efficient way to manage client connections without the overhead of creating a new thread for every request.

- **Minimalist Text Protocol**: We designed a simple, human-readable text protocol (e.g., `LOCK,<lockName>,<clientId>`). This choice simplifies debugging and implementation compared to a binary protocol, which would require more complex serialization/deserialization logic.

- **Request Routing**: Followers act as simple routers for write operations, forwarding `LOCK/UNLOCK` requests to the leader. This design decision is central to ensuring consistency, as it establishes the leader as the single point of truth (or "single-writer") for all state changes, preventing split-brain scenarios.

- **Pending Request Mechanism**: Following the project specifications, when a follower receives a `LOCK` or `UNLOCK` request, it marks the request as *pending* and keeps the client connection open. The follower forwards the request to the leader. If the leader returns `FAIL`, the follower responds immediately. If the leader returns `SUCCESS`, the follower waits for the `SYNC` message. When the `SYNC` arrives, the follower updates its local map, checks if the request is pending, and if so, sends the `SUCCESS` response to the client. This mechanism ensures that clients receive confirmation only after the state change has been replicated via the `SYNC` message.

- **Synchronous Replication**: The leader uses a synchronous `SYNC/ACK` model for replication. When the leader approves a `LOCK` or `UNLOCK` operation, it updates its local map and sends a `SYNC` message to all followers in parallel (using a thread pool). The leader then **waits for ACK from all followers** using a `CountDownLatch` before considering the replication complete. This ensures that when the leader returns `SUCCESS` (either directly to a client or to a follower that forwarded the request), all followers have received and acknowledged the state change. The follower that forwarded the request also waits for the `SYNC` before responding to its client, providing strong consistency guarantees.

- **Reasoning (Consistency)**: The synchronous replication combined with the pending request mechanism provides strong consistency: when a client receives a `SUCCESS` response from a follower, it has a guarantee that the state change has been (1) accepted by the leader, (2) replicated to **all followers** (via synchronous SYNC/ACK), and (3) applied to that specific follower's map. This ensures that all servers in the cluster have a consistent view of the lock state before any client receives confirmation of a successful operation. The synchronous approach trades some latency for strong consistency guarantees, which is appropriate for a distributed lock system.

- **Simple Error Handling**: The system relies on socket timeouts (5-10s) and explicit response codes (`SUCCESS/FAIL/NONE/ERROR/TIMEOUT`). This provides basic robustness against network delays or unresponsive servers without implementing a complex failure-detection and leader-election protocol, which was deemed outside the project's scope.

3 Data Flow and Architecture

The system's functional architecture is centralized around the leader, which serializes all state changes. The interactions are as follows:

1. **Write Operation (LOCK/UNLOCK)**:

    - **From Follower**: Client → Follower (marks request as *pending*, keeps connection open) → Follower forwards request to Leader asynchronously → Leader processes, updates map, sends `SYNC` to all followers (including the one that forwarded), and waits for all `ACK` (synchronous replication) → Leader responds `SUCCESS` to follower's forwarding thread,

but the follower ignores this response and waits for `SYNC` → Follower receives `SYNC`, updates local map, checks if request is *pending*, sends `ACK` to leader, and if request is pending, sends `SUCCESS` to client and closes connection. If the leader returns `FAIL` (before sending SYNC), the follower responds immediately to client and closes connection.

- **From Leader**: Client → Leader → Leader processes, updates map, sends `SYNC` to all followers, waits for all `ACK` (synchronous replication), then responds `SUCCESS` to client.

2. **Read Operation (OWN)**: Client → (Any) Server → Client. This operation is handled locally by whichever server receives it to ensure low latency. The response is sent immediately and the connection is closed.

3. **Follower Registration**: When a follower server starts, it sends a `REGISTER,$ip:$port` message to the leader. The leader adds the follower to its internal list and responds with `REGISTERED`. This allows the leader to know which followers to notify during replication.

This flow is illustrated in Figure 1, which shows the sequence of network calls for both a write (`LOCK`) and a read (`OWN`) operation.

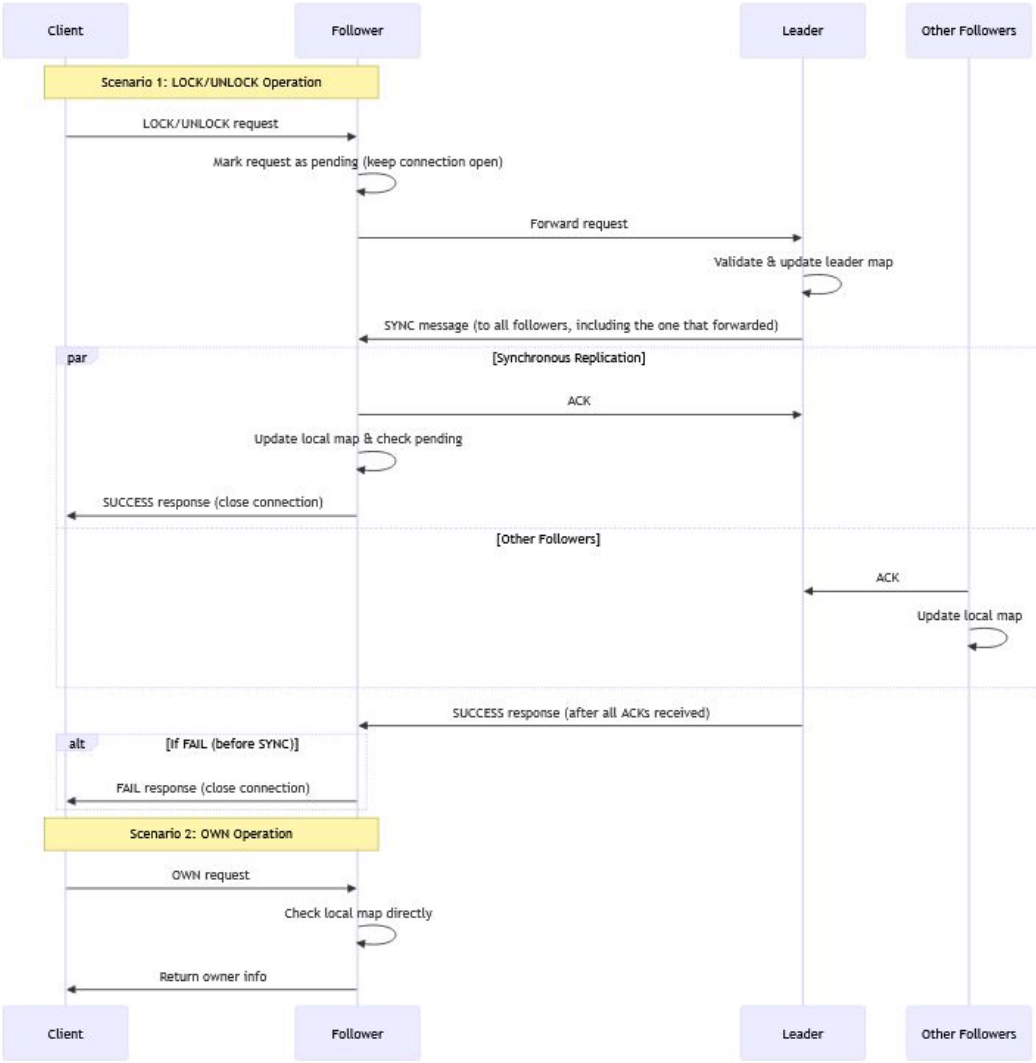

Figure 1: Sequence diagram of system operations (LOCK and OWN).

4 Structure of the Code

The project consists of three main Java classes:

- `Client.java`: This class provides the client-facing API: `tryLock`, `tryUnLock`, and `ownTheLock`. Each operation opens a new, short-lived socket, sends its request message via `sendMsg()`, and reads the single-line response from the server.
- `DistributedLockTest.java`: A test suite that uses a `FixedThreadPool` to simulate concurrent clients. These clients connect to different servers in the cluster (the leader at 10.0.2.3, and followers at 10.0.2.4 / 10.0.2.5) to verify concurrent lock acquisition, owner-reading, and lock release.
- `Server.java`: This is the core class, capable of running in either "leader" or "follower" mode. It uses a `CachedThreadPool` to handle each incoming connection in a separate thread (`handleClient()`).
  - `handleClientRequest()`: Parses client messages (`LOCK`, `UNLOCK`, `OWN`) and handles them according to the server role. For `OWN` requests, it responds immediately. For `LOCK`/`UNLOCK` on a leader, it processes and responds immediately. For `LOCK`/`UNLOCK` on a follower, it implements the *pending request* mechanism (see below).
  - `processRequest()`: This `synchronized` method is the main logic gate for business rule enforcement.
    - If **isLeader**, it calls `handleLeaderRequest()` to enforce business rules, update the local `lockMap`, and trigger replication via `notifyFollowers()`.
    - If **isFollower**, it calls `handleFollowerRequest()` (only used for `OWN` requests in the pending implementation).
  - **Pending Request Mechanism** (for LOCK/UNLOCK on followers): When a follower receives a `LOCK` or `UNLOCK` request:
    - The request is marked as *pending* (stored in `pendingRequests` map with key `lockName:clientId:`) and the client connection is kept open.
    - The request is forwarded to the leader in a separate thread via `forwardToLeaderForPending()` (non-blocking for the main connection handler).
    - If the leader returns `FAIL`, the follower responds immediately to the client and closes the connection.
    - If the leader returns `SUCCESS`, the follower waits for the `SYNC` message.
  - `handleSyncMessage()` and `processSync()`: When a follower receives a `SYNC` message, `handleSyncMessage()` is called, which then invokes `processSync()`. This method:
    - Updates the local `lockMap` according to the SYNC command.
    - Checks if the request is *pending* (by constructing the pending key from the SYNC parameters).
    - If the request is pending, sends `SUCCESS` response to the client and closes the connection.

    After `processSync()` completes, `handleSyncMessage()` sends the `ACK` back to the leader.
  - `notifyFollowers()`: The leader iterates through its list of followers, submitting a new task to the thread pool for each one to send a `SYNC` message in parallel. The leader uses a `CountDownLatch` to synchronously wait for `ACK` responses from all followers. This ensures that the leader does not return `SUCCESS` until all followers have received and acknowledged the state change, providing synchronous replication and strong consistency guarantees. Each task has a 5-second timeout, and the leader has a 10-second overall timeout for all ACKs.

Business Rules (Enforced by Leader)

- **Preemption (LOCK)**: Success if the lock (key) does not exist in the `lockMap`. Failure otherwise.
- **Release (UNLOCK)**: Success if the lock (key) exists *and* the caller's `clientId` matches the one stored in the map. Failure otherwise.
- **Read (OWN)**: Any client can query the owner. The request is handled by any server and returns the owner's `clientId` or `NONE` if the lock does not exist.

5 Communication Protocol and Error Handling

The system uses a simple, text-based, comma-delimited protocol.

Client-Server Messages

Table 1: Client-Server and Inter-Server Protocol

Type	Format	Description
Client → Server	`LOCK,$name,$client`	Lock acquisition request
Client → Server	`UNLOCK,$name,$client`	Lock release request
Client → Server	`OWN,$name,$client`	Read owner request
Leader → Followers	`SYNC,$cmd,$name,$client`	State replication (e.g., SYNC,LOCK,...)
Followers → Leader	`ACK`	Acknowledgment of SYNC
Followers → Leader	`REGISTER,$ip:$port`	Follower registration on startup
Leader → Follower	`REGISTERED`	Confirmation of successful registration
Leader → Follower	`NOT_LEADER`	Response when non-leader receives REGISTER

Server Responses and Error Handling

Server responses to the client are simple, single-word strings, as shown in Table 2. This provides basic robustness. Timeouts are implemented on socket operations to prevent threads from hanging indefinitely on network issues: 30 seconds for pending client connections (to allow time for SYNC to arrive), 10 seconds for forwarding requests to the leader, 5 seconds per follower for SYNC operations, and a 10-second overall timeout for receiving all ACKs during replication.

Table 2: Server Response Codes

Response Code	Meaning
`SUCCESS`	Operation (LOCK/UNLOCK) succeeded.
`FAIL`	Operation (LOCK/UNLOCK) failed (e.g., lock taken, not owner).
`NONE`	`OWN` request returned no owner for the lock.
`<ClientID>`	`OWN` request returned the current lock owner.
`ERROR`	A generic connection or processing error occurred.
`TIMEOUT`	A network operation (e.g., forwarding) timed out.
`INVALID_FORMAT`	Message format is invalid (not enough parts).
`INVALID_COMMAND`	Command type is not recognized (not LOCK/UNLOCK/OWN).

Note: Additional inter-server response codes include `REGISTERED` and `NOT_LEADER`, used during follower registration with the leader.

6 Experimental Results

The following figures illustrate the system in operation, using three virtual machines as specified in the `README.md`.

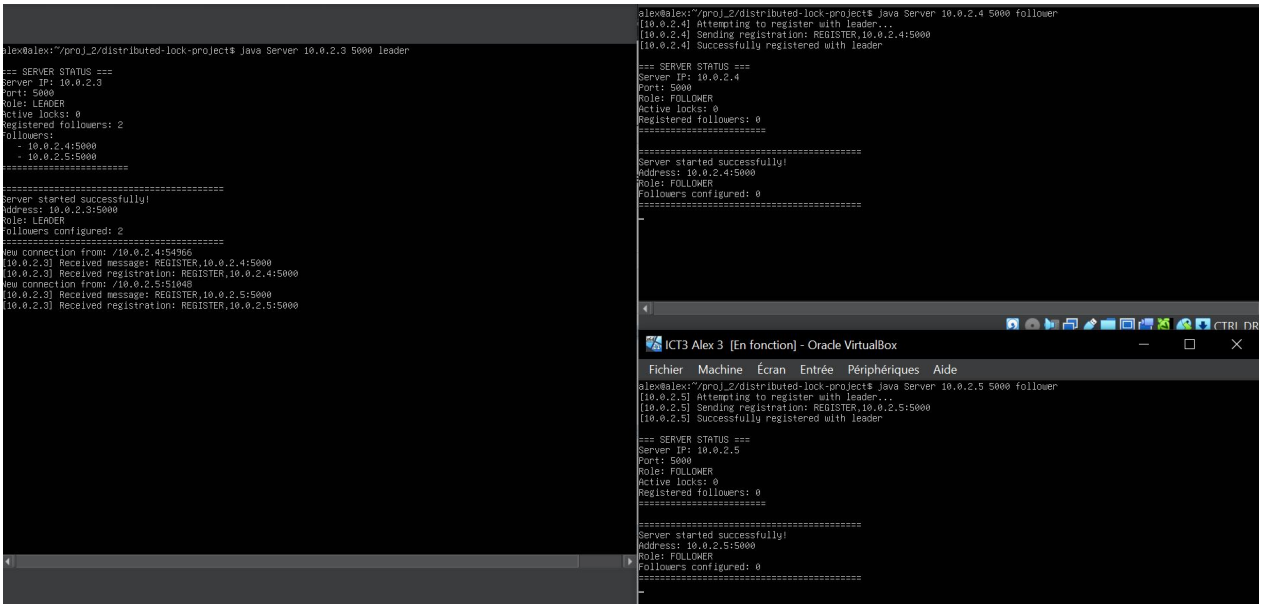

Figure 2: Initialization of the leader (10.0.2.3) and the two followers (10.0.2.4, 10.0.2.5).

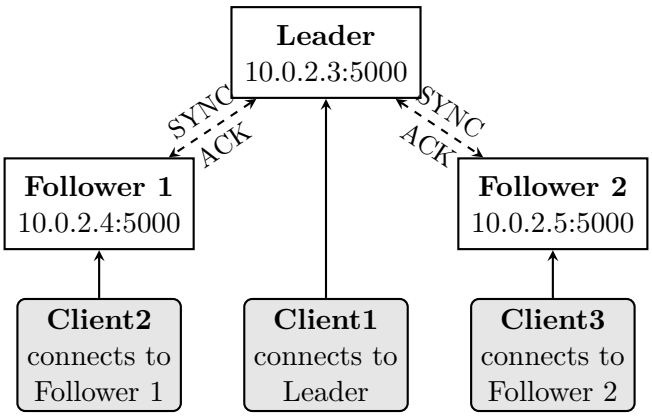

Figure 3: Test architecture for `DistributedLockTest`: three clients connect to different servers (Client1 to the leader, Client2 and Client3 to the followers) to test concurrent lock operations and replication.

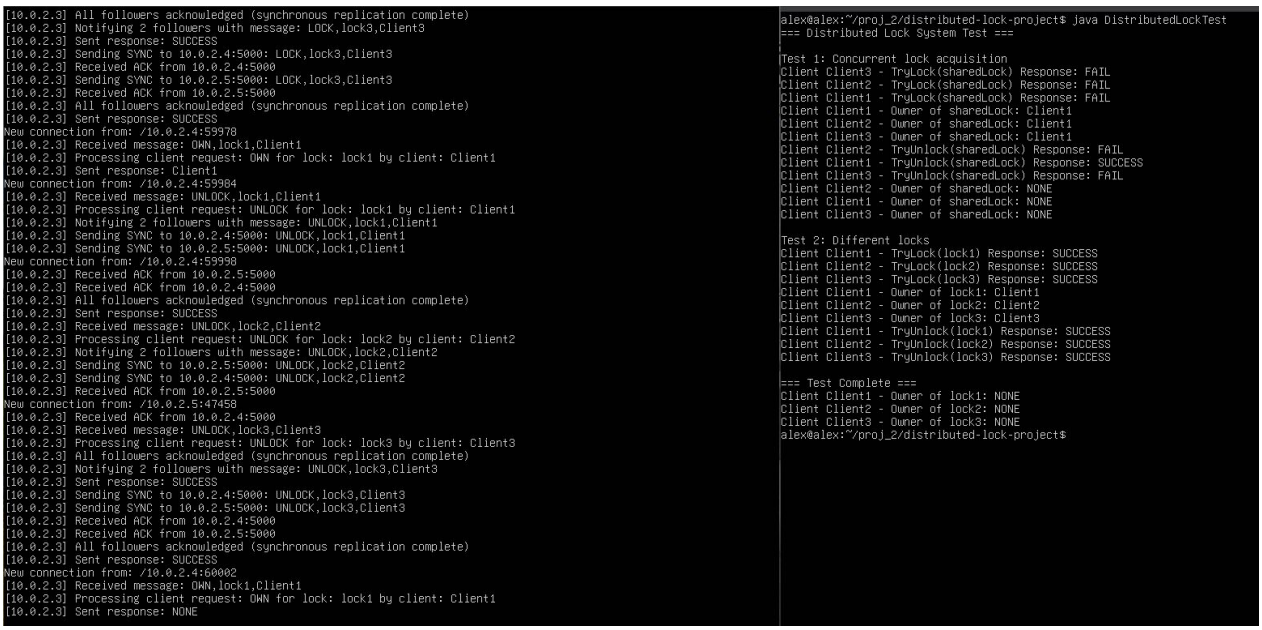

Figure 4: Execution of `DistributedLockTest`: concurrent acquisitions, reads, and releases. The test architecture is shown in Figure 3.

Observations

- The test results show that for a single lock, only one client's `LOCK` request succeeds, while concurrent attempts from other clients correctly fail (returning `FAIL`). The architecture shown in Figure 3 demonstrates how the three clients connect to different servers in the cluster.
- `OWN` requests, even when sent to followers, correctly return the ID of the lock owner or `NONE` after the lock is released. Due to the synchronous replication mechanism, followers should always have the most up-to-date data, ensuring strong consistency across all servers.
- The `SYNC` replication mechanism successfully maintains strong consistency across the leader and all followers. The synchronous replication (with ACK confirmation) ensures that all state changes are replicated to all followers before any client receives confirmation, guaranteeing a consistent view across the entire cluster.

7 Limitations and Future Work

While functional, this implementation has several limitations that could be addressed in future work.

- **Hard-coded Configuration**: The leader's address (10.0.2.3) and the list of followers are hard-coded in `Server.java`. This is inflexible. A better solution would use a configuration file or command-line arguments to define the cluster topology.
- **Fault Tolerance**: The system has a single point of failure (SPOF): the leader. If the leader server crashes, the system can no longer process any `LOCK` or `UNLOCK` requests. A robust system would require a fault-tolerance mechanism, such as leader election (e.g., using Paxos or Raft), which was outside the scope of this project. Additionally, if a follower fails during replication, the leader logs the error but continues, potentially leaving that follower out of sync until it recovers and re-registers.
- **Network**: Communication is unencrypted (plaintext), and there is no client authentication. This is insecure for a production environment.

- **Connection Management**: The pending request mechanism keeps client connections open while waiting for SYNC messages. While a 30-second timeout prevents indefinite hanging, this approach could be improved with connection pooling or more sophisticated connection lifecycle management for high-throughput scenarios.
- **Thread Safety**: While `ConcurrentHashMap` and synchronized methods provide thread safety, the combination of pending requests and synchronous replication could be further optimized to reduce contention under high load.

8 Conclusion

We have successfully implemented a distributed lock system that fulfills the core requirements of the project. It uses a leader-follower architecture with a simple Java socket protocol to manage `LOCK`, `UNLOCK`, and `OWN` operations. The system ensures strong consistency by routing all write operations through the leader and using synchronous replication (with pending request mechanism) to update followers. The leader waits for acknowledgments from all followers before confirming successful operations, ensuring that all servers maintain a consistent view of the lock state. The pending request mechanism further strengthens consistency by ensuring that clients receive confirmation only after their request has been replicated to the follower that processed it. Tests demonstrate the system's correctness in handling concurrent requests and maintaining a consistent state across all servers. This project serves as a solid foundation for understanding consensus and distributed system.