Detailed Design of Chubby: Part II

Let’s understand the locking service, how to handle the problems that occur in it, and how to perform operations in Chubby.

We'll cover the following

- Data consistency and concurrency problems
- Locking
  - Complexity in locking
- Sequencers
- Lock delay
- Events
- API design of Chubby
  - Chubby handle
  - Actions
  - Workflow

Data consistency and concurrency problems

Let’s assume that thousands of clients are trying to access the nodes in the primary replica and constantly reading data from them and also writing data into them. In such a scenario, Chubby will face the following data consistency and concurrency issues.

- How will it ensure that only one client can write a file at a time?
- What happens when clients are reading from a file, and a write request comes in and updates the data?

This lesson addresses the above-mentioned concerns in Chubby's design. The table below summarizes the goals of this lesson.

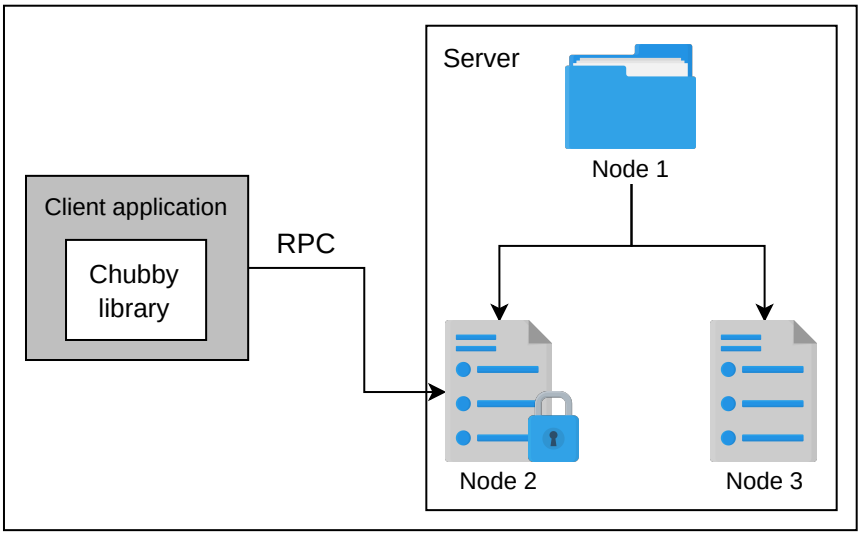
Lesson Summary

Section	Purpose
Locking	This allows files of a server to be locked by clients in exclusive and share and addresses the issues mentioned above.
Sequencer/lock delay	These methods provide support for the complexity of the primary replic delayed messages from clients that previously held a lock which is held other clients now.
Events	This allows primary replica to report any changes in the database to the
API design of Chubby	This gives the commands that are used to perform several different oper Chubby.

Locking

Chubby's files and directories (nodes) can act as a reader or writer lock. Clients can hold these locks in exclusive and shared modes.

- Numerous clients can hold a lock in reader mode (shared).
- A single client handle can hold a lock in writer mode (exclusive).



Client holding a lock on node 2

Locks are **advisory**, meaning that holding a lock is not necessary to read a file, and it does not prevent other clients from reading that file. The clients accessing the

files will have to cooperate to ensure no conflict occurs and locks are observed properly. Conflicts can occur only when other clients attempt to acquire the exclusive lock.

Question

Why is advisory locking chosen over mandatory locking?

Hide Answer

Chubby's advisory locks are more flexible and scalable, and if application clients follow the Chubby protocol fully, they can emulate strict locking (like exclusive locks) easily. Advisory locks are more robust in distributed settings where difficult-to-detect failures (for example, due to network partitions) can happen. Error checking is performed traditionally, i.e., by writing assertions like "lock X is held," and advisory locks return results that indicate that a lock is held or not. However, a mandatory lock suspends a process until the lock is free.

Acquiring the lock in Chubby, either in exclusive or shared mode, requires write permissions. The write permissions ensure that an unprivileged reader (a reader not having a lock) cannot prevent a writer from making progress.

Question

The write lock does make sense (only a single client can hold it), however, if numerous clients can hold a read lock and holding a lock is not even necessary for reading why do we have a read lock in Chubby?

Hide Answer

The read lock can be held by multiple clients, but while the clients that are holding the lock are reading, they block all the write lock requests until they are done.

Complexity in locking

Locking, when implemented in distributed systems, can be complex. The reason is that any client can request reading/writing to any node, so the processes can fail independently, and requests may get delayed and reach a primary replica server without any specific order.

Suppose that client 1 holding Lock A issues a request, R, and then fails (the primar releases a lock if a client holding a lock fails, which we discuss in further detail later). However, before R arrives at the destination, client 2 takes Lock A and performs a certain action. After that, R arrives out of order at the destination and could face two problems. It could be processed without the protection of Lock A, and the data could be malicious too.

Question

How does Chubby cope with this problem?

Hide Answer

It deploys sequencers and lock delays.

Sequencers

Chubby only introduces sequence numbers for the interactions that use locks. A **sequencer** is a byte string that describes the state of a lock after it is acquired. It contains the following components:

- A lock name
- A lock mode (exclusive/shared)
- A lock generation number

Question

Why does Chubby *not* introduce sequence numbers in all the interactions?

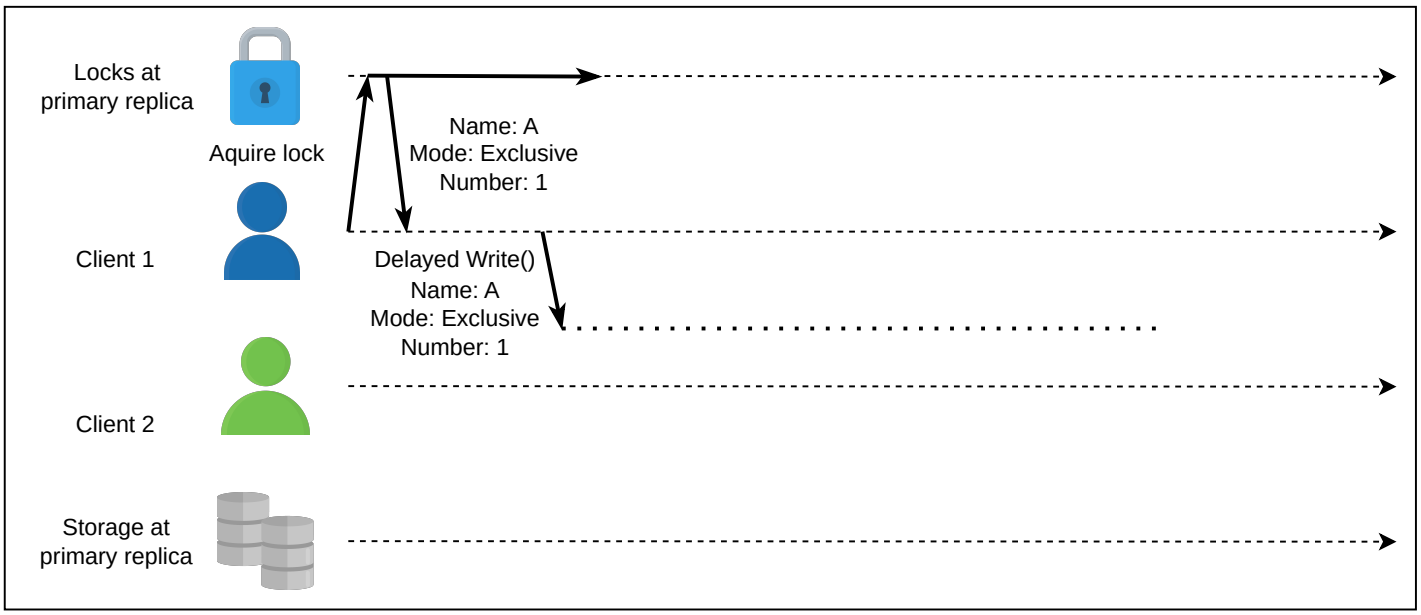
Hide Answer

Chubby does not add sequence numbers for all interactions of a complex system because it is costly to do so in Chubby's locking, where we are only concerned about the operations performed by the clients that have the locks. If all the interactions are given a sequence number, then we will also have to check which one is the lock and which one isn't. We will not compare the sequence number of normal interactions with the sequence numbers of the locks to make them invalid.

The process of passing sequencers in Chubby is as follows:

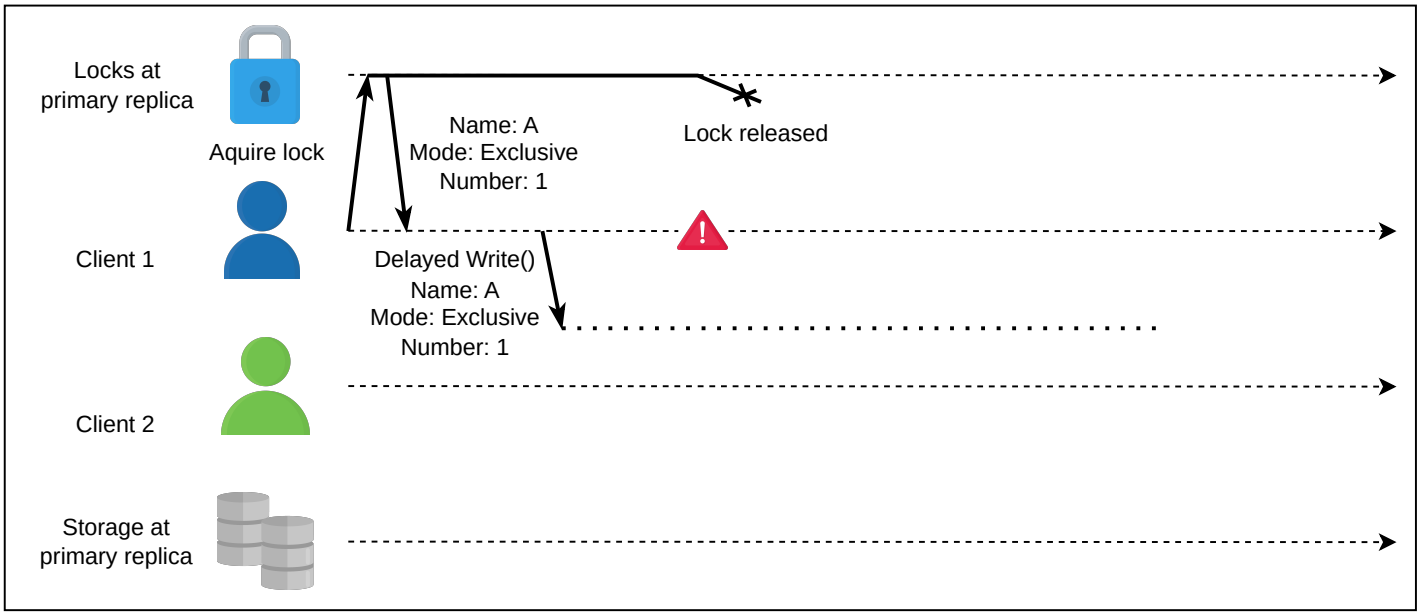
- If an operation is protected by a lock, the client passes the sequencer to the servers.
- The server that receives the sequencer tests the sequencer for:
    - Validity
    - Appropriate mode

- If the sequencer is invalid or does not have the appropriate mode, the server rejects it.
- Chubby only needs to add a string to each message for sequencers.



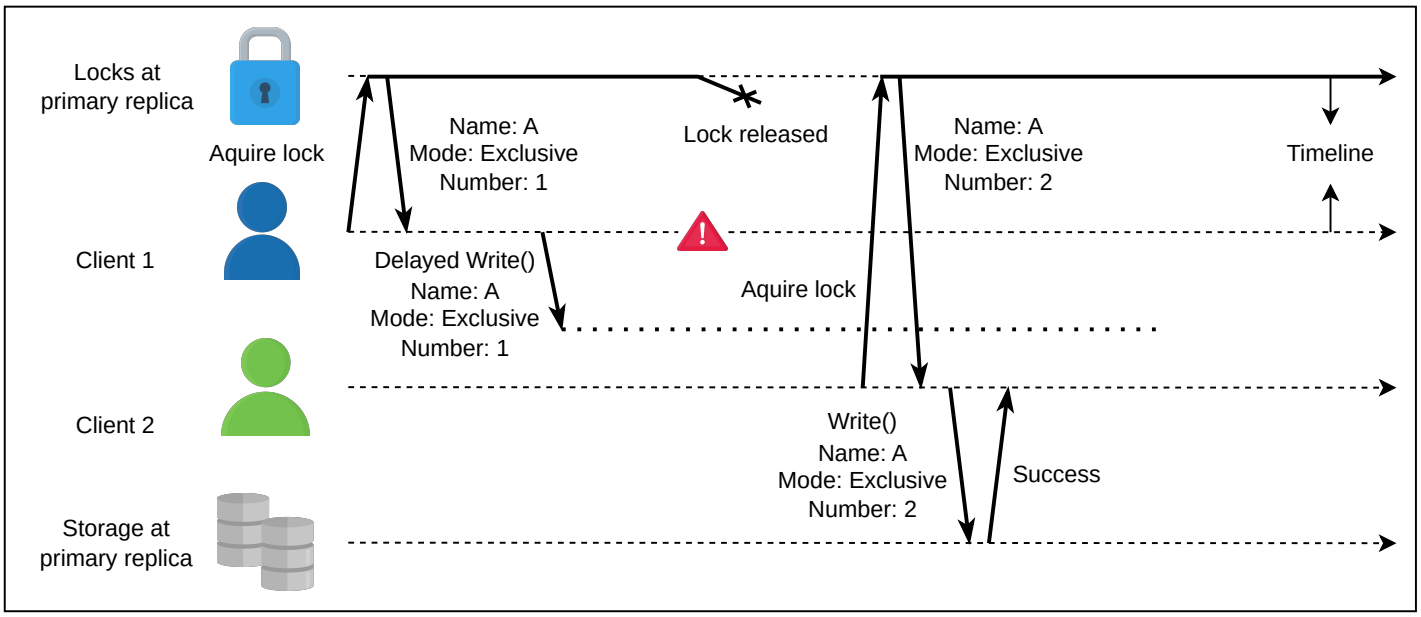
Client 1 acquires the lock, gets a sequence number, and sends a write request that gets delayed

1 of 4



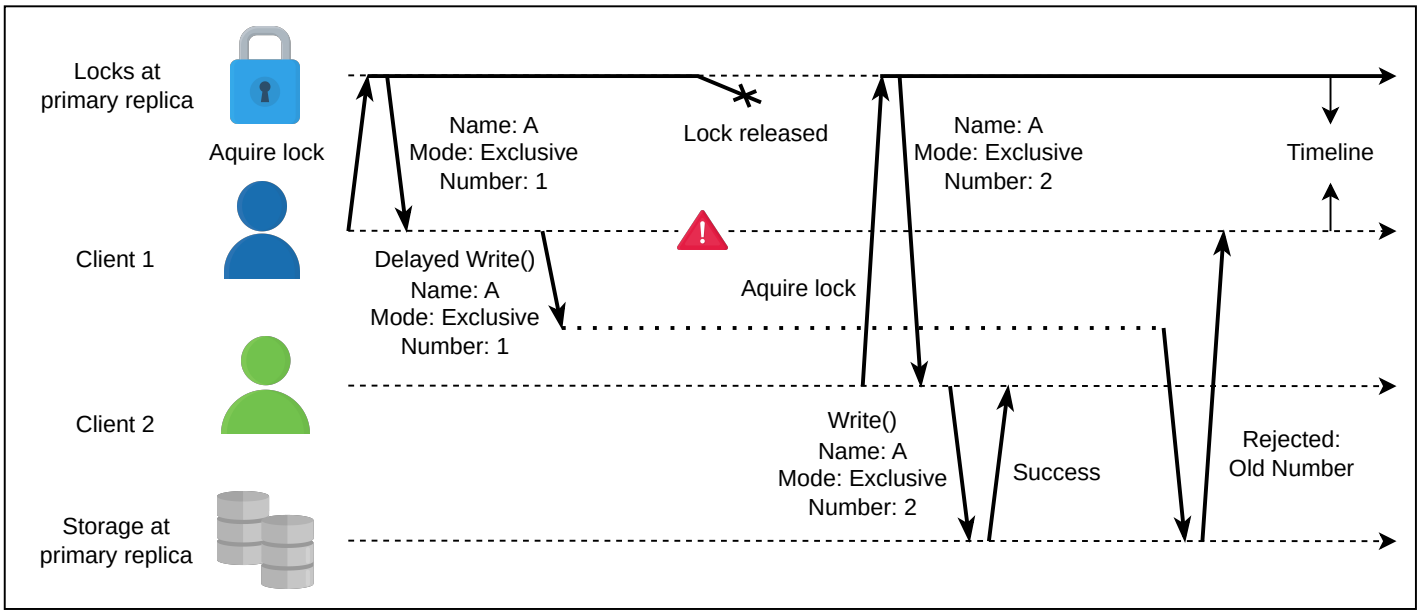
Client 1 fails and the lock is released

2 of 4



Client 2 acquires the lock, gets a sequence number, and sends a write request that is successful

3 of 4



The delayed write request reaches the primary replica with an invalid sequence number and gets rejected

4 of 4

The sequencers are easy to use but are not perfect because some servers do not support them. Therefore, Chubby provides another mechanism called a lock delay, which allows them to avoid the problem of unordered messages.

Question

How does the server receiving a sequencer check for its validity?

Hide Answer

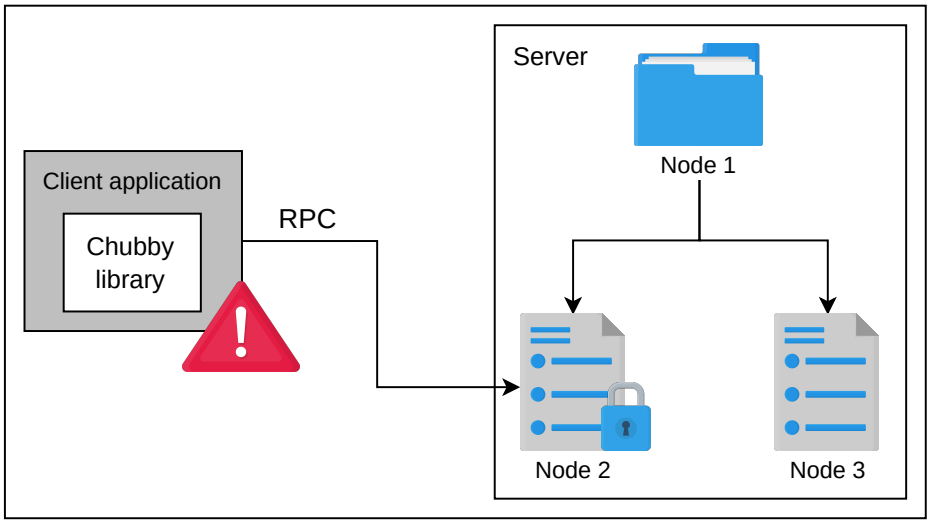
A recipient server can check for validity in two ways:

- From the last noticed sequencer by the server (this option is only opted if the session of the server has expired)
- From the server's Chubby cache (discussed later)

Lock delay

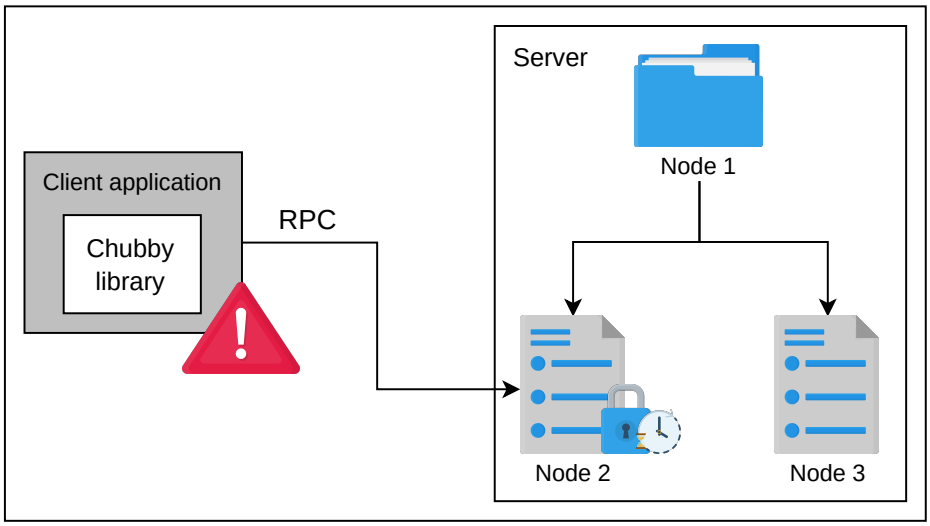
A client usually releases a lock in a normal way. However, a lock can become free if the holder fails. In the case of a normal release, a lock is free for other clients to hold. However, in case of the client holding it fails and the lock becomes free, the lock server does not let any other client claim it for a specific time period. This is called a **lock delay**.

A lock delay of up to 1 minute can be set. This limit is introduced to prevent faulty clients from claiming a lock and making it unclaimable for long periods. A lock delay might not be the perfect method to prevent faulty clients from holding a lock, but it protects clients from message delays and restarts.



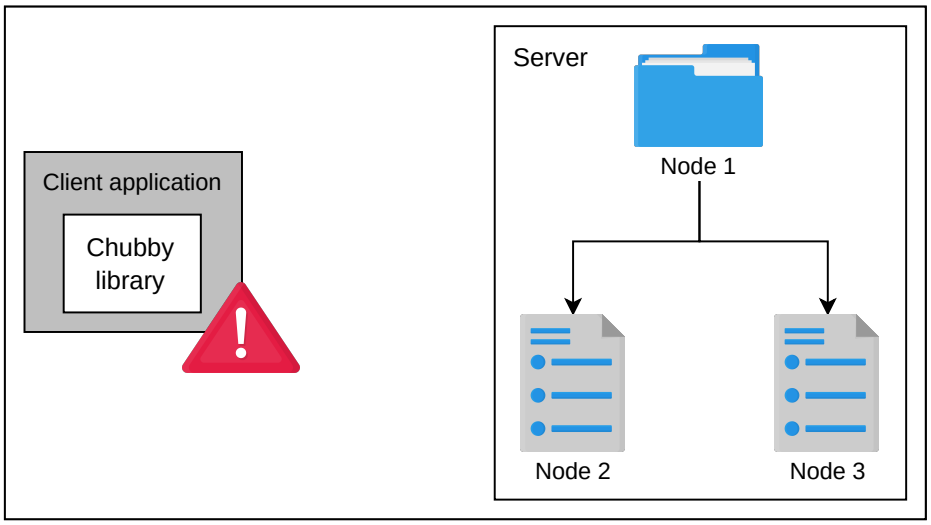
The client holding the lock fails

1 of 3



The lock delay started on the lock freed from a faulty client

2 of 3



The lock on node 2 is free to be held by any client

Question

How does Chubby let the clients monitor certain processes in Chubby?

Hide Answer

To allow Chubby clients to monitor specific processes, Chubby provides events.

Events

A wide range of activities are known as **events** in Chubby. Let’s go through some of the events and understand why they are used.

- **Modifying file contents:** It is used to keep track of a service's location that is being advertised through a file.

- **Adding, removing, or modifying a child node**: It is used to implement mirroring (discussed later). Moreover, it is also used to allow new files to be discovered and monitor ephemeral files in child nodes.
- **Primary replica failover:** It is used to alert other clients that some events might be lost so that they can rescan the data.
- **A handle becoming invalid:** It is used to suggest a problem in communication.
- **Acquiring lock:** It is used to monitor the election of a primary.
- **Conflicts in lock acquiring:** It is used to determine the caching of data from other servers if locks should cache data.

Chubby clients can subscribe to any event after creating a handle. Events get delivered to clients whenever a corresponding action happens via a KeepAlive call (discussed later in detail) from `Chubby` library. For example, if a client is notified that a file has changed and reads the file subsequently, it will see the new data.

API design of Chubby

The API design of Chubby is described below.

Chubby handle

The Chubby **handle** appears as a pointer that supports multiple operations to Chubby clients. They can be created and destroyed by using the following command:

- `Open()`: It opens a file or a directory and gives a handle to it. It takes a node name, and all other commands operate on handles. The node name is created relative to the existing directory handle.
- The client can open a file or directory with the following options:
  - How to use the handle, i.e., read, write, lock or change an ACL. The handle can only be created by the client with appropriate permissions.
  - Which events to deliver

    - Lock delay
    - Whether to create a new file or directory
- `Close()`: This command is used to close an open handle. After this, the handle can no longer be used.
- `Poison()`: This is a call similar to the `Close()` call. It causes all the subsequent operations applied on the handle to fail. It can nullify the calls initiated by other threads. However, it does not deallocate the memory accessed by these calls.

Actions

A client can perform the following actions on the handle:

- `GetContentsAndStat()`: This call reads the content and metadata of a file.
    - `GetStat()`: It reads only the metadata of a file.
    - `ReadDir()`: It reads the names and metadata of each child of a directory.
- `SetContents()`: This call writes the contents of a file.
- `SetACL()`: This call writes the content of ACL names associated with a node.
- `Delete()`: It deletes a node with no children.
- `Acquire()`, `TryAcquire()`, `Release()`: These calls are used for acquiring and releasing locks.
- `SetSequencer()`, `GetSequencer()`, `CheckSequencer()`: These calls are used on the sequencers. The `SetSequencer()` call associates a sequencer with a handle (if the sequencer is not valid, all the subsequent operations on the handle will fail), the `GetSequencer()` call is used to get any sequencer that describes the locks with this handle, and the `CheckSequencer()` call checks for the validity of a sequencer.

Workflow

This API can be used by clients for the primary election.

- All the servers open the lock file and attempt to acquire the lock.
- The first server to do so becomes the primary server, and all the others