The Chubby Lock Service

Abstract

This project implements and evaluates a subset of the Chubby lock service in Rust. In contrast to the original Chubby, we use Raft to implement a replicated state machine.

1 Introduction

This project aims to implement Chubby [4], a coarse-grained, fault-tolerant lock-service for loosely coupled distributed systems, with the goal of understanding the challenges of building locking mechanisms for distributed systems and contrast them with locking mechanisms found in conventional operating systems such as mutexes and semaphores.

Locking mechanisms in distributed systems allow nodes to synchronize their activities such that they have a consistent view of the environment in which they operate. Locks have traditionally been used to solve one of the following problems [1]:

- Efficiency - Acquiring a lock prevents two nodes from performing the same computation twice. This is useful in situations where duplicating a computation does not lead to incorrect results and performance can be gained by simply avoiding the duplication.

- Correctness - Threads executing on multiple machines use global locks for synchronizing access to shared data, to prevent conditions such as data races.

However, correctly implementing locks for distributed applications is a challenging task because network communication could lead to delays and nodes could fail arbitrarily. First, a node holding a lock may fail, or the lock release message could be arbitrarily delayed by the network, in which case other clients may be prevented from acquiring the lock. Second, pauses in application execution and network delays may lead to a situation in which multiple clients believe they have acquired a lock. Consequently, updates to shared data will not be protected by mutual exclusion, resulting in data races and inconsistent state. Sections 2.3 and 2.4 explore these issues in more detail and looks at solutions to overcome them.

Chubby's original implementation uses Paxos [7] with view-stamped replication [8] to implement a fault tolerant replicated state machine that is capable of leader election and surviving membership changes. Instead, we use Raft [9] to solve the distributed consensus problem. Our choice of the Raft protocol was motivated by the fact that it is easier to understand than Paxos, and has built in mechanisms to solve the problems of leader election and membership changes. Moreover, the availability of a Raft library written in Rust made integration with our implementation easier.

The rest of this report is organized as follows. Section 2 explores the design space of distributed locking mechanisms and looks more closely at the design choices/tradeoffs made by Chubby. Section 3 lists some useful primitives that could be built with a distributed locking service and explains how these could be implemented with Chubby. Section 4 discusses details of our implementation. Section 5 performs an evaluation of our Chubby implementation. Section 6 looks at future work, which are some ideas that we wanted to play around with, but weren't able to do so due to a lack of time.

2 Design

Chubby is designed with the goal of providing a highly available lock service to distributed applications. The authors justify the abstraction of a lock service as opposed to a library that provides access to a replicated service by highlighting some of the benefits of a lock service.

First, exposing a lock service makes it easier for developers to focus on functionality without having to prematurely incorporate availability into their design. However, planning ahead for availability is non-trivial, as a result of which early software prototypes do not account for it. Consequently, availability is a feature that is added explicitly to software once it has reached maturity, a process that could potentially involve restructuring code. Instead, using a lock service from the get-go make it easier to maintain existing program structure and communication patterns.

Second, lock services provide a natural mechanism that allows clients to read and write small quantities of data. This could be useful in situations where the result of an operation involving multiple nodes, such as leader election or configuration updates, must be broadcast to all nodes in the system.

Third, most software developers are familiar with using locks for mutual exclusion and co-ordination on shared memory multiprocessors. As a result, exposing a lock-based interface allows them to write distributed applications just as they would write multi-threaded applications on a single machine.

Finally, replication using distributed consensus typically involves communication with a majority of servers in the system. For example, in a system with a large number of replicas, each client would have to wait for operations to be replicated on a majority of these replicas before being able to proceed with execution. On the other hand, the advantage of a lock service is that a client only needs to communicate with the lock service in order to make progress.

2.1 Locking Granularity

A key design decision to be made when implementing a lock service for distributed applications is whether to use coarse grained (where locks are held for long periods of time) or fine grained locks (where locks are held for short periods of time).

Coarse grained locks are typically used by applications where the lock acquisition rate is much smaller than the rate of transactions generated. Since locks are acquired infrequently, coarse-grained locks impose a lesser load on the lock server. Moreover, temporary unavailability of the lock server does not adversely impact clients.

Fine-grained locks on the other hand, impose a significant load on the lock server. Moreover, due to the high rate of lock operations, failure of the lock service could impede client progress.Therefore, using a centralized service for applications that frequently acquire and release locks

Chubby chooses to use coarse grained locks because in the common case, it expects applications to only acquire and release locks infrequently. Quite naturally, a the Chubby leader could become a performance bottleneck if the rate of lock operations is high. In our experimental evaluation, we try to quantify the lock acquisition rate that Chubby provides and also compare it with performance in the common case, where lock acquisition rates are low. But, the authors argue that implementing fine-grained locks atop Chubby's coarse-grained locks is straightforward.

2.2 Locks

Chubby exports a file system interface to clients. Our design uses a simple path-based approach where every lock file is identified by a unique path name. Clients must use the lock's path name for every operation they request from the lock service.

An important design choice that Chubby makes is the use of *advisory* locks over *mandatory* locks. In a mandatory locking scheme, once a client acquires an exclusive lock to a resource, the system prevents access to this resource from other clients until the exclusive lock is released i.e. the system guarantees the failure of operations that attempt to concurrently operate on a resource which has been locked by another client. On the other hand, advisory locks assume that all clients follow a locking protocol to obtain access to a shared resource i.e. the system does not take any steps to prevent an uncooperative client from concurrently accessing a resource for which some other client has already acquired a lock. Consequently, advisory locks only conflict with other attempts to acquire the same lock.

One of the main reasons to prefer an advisory locking scheme is that Chubby's locks were designed to protect resources outside its purview. As a result, implementing a mandatory scheme would require Chubby to understand the semantics of these services and possibly interact with them, a situation that doesn't scale due to the wide range of available services. Clearly, doing this would require extensive modifications to all such services.

Another problem with mandatory locks is that in the face of software errors, getting the system to release these locks could prove difficult. On personal computers and workstations, the solution could be as simple as forcing the machine to shutdown and reboot. However, the authors argue that forcing users to shut down their machines in a distributed environment is not a feasible solution.

To create a lock, a client issues an `open()` call. The lock service then proceeds to mark this path as being a valid path name, which can be subsequently used for acquisitions and releases. Acquiring a lock is performed with the `acquire()` call, which takes a parameter indicating which mode the lock must be acquired

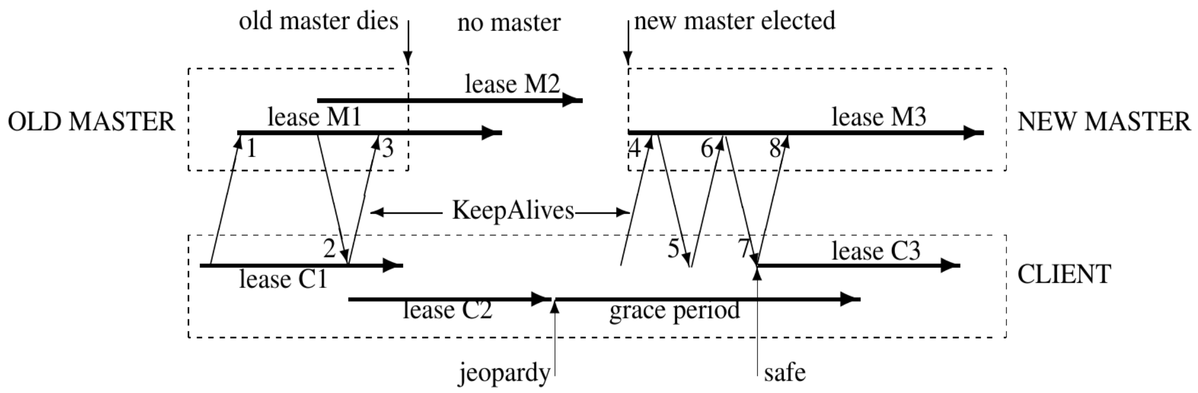

Figure 1: **Chubby Client Jeopardy**

in. Chubby supports two modes - exclusive (writer) and shared (reader) mode. Every lock service operation performs validation checks to ensure that a lock has first been opened and acquired by a particular client before allowing the operation to proceed. The API is discussed in section 2.6.

2.3 Sessions

Implementing locking mechanisms for distributed applications is fundamentally more challenging than locking in multi-threaded applications because compute nodes and the inter-connecting network may fail independently. As a result, we may find ourselves in an unfortunate situation in which a node that's currently holding a lock is unable to relinquish ownership of the lock. To better understand the problem, let's consider an application that needs update a file in shared storage. To do this, the application would first acquire a lock, read the file, modify the file, write back the modified file to shared storage and finally release the lock. In this case, the lock is used to prevent data races by providing mutual exclusion. However, in an asynchronous model, either the application's server could fail after acquiring the lock or the release-lock message maybe unduly delayed before reaching the lock server. As a result, no other application will be allowed to acquire the lock, thereby impeding progress.

Chubby uses the notion of leases [5] to solve this problem. Every client that uses the Chubby lock service must first establish a session with it. Upon creation of a session, the lock service assigns the session a lease, which is an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The client is responsible for renewing this lease (using KeepAlive messages) before its expiration, failing which the lock service assumes this client has failed and therefore reclaims all the locks this client had acquired. A design choice that must be made while using leases is whether to use short-term leases or long-term leases. Short term leases have the advantages of minimizing the delay resulting from failures and also minimizing the effects of false-sharing (for example, a client that holds a lock may not be doing anything useful with it). On the other hand, long-term leases are significantly more efficient for data that is accessed repeatedly by the same client and is not frequently shared with other clients. Since Chubby is designed as a coarse-grained lock service where operations on locks occur infrequently, it uses a relatively long lease of 12 seconds, which can be tuned depending on lock acquisition rates and server load.

The lock service is free to advance a client's lease, but may not move it backwards in time. A client's lease is renewed in any of the following circumstances:

Session creation When a new session is created, the server assigns it a lease with the default duration.

KeepAlive Response When the lock service receives a KeepAlive message, it blocks the RPC till the lease is close to expiring. The reason for this is the avoid excessive network messages because the master assumes that the client will be alive till its lease expires. When the lease is close to expiring, the master renews the client's lease and responds to the RPC with the new lease value. Once a client receives a response to a KeepAlive message, it immediately sends another KeepAlive. As a result, the client always has a KeepAlive call blocked at the lock server, thereby ensuring that the its lease will eventually be renewed. Another point worth noting in Chubby's design is that it piggy-backs event notifications and cache invalidation messages on responses to KeepAlives.

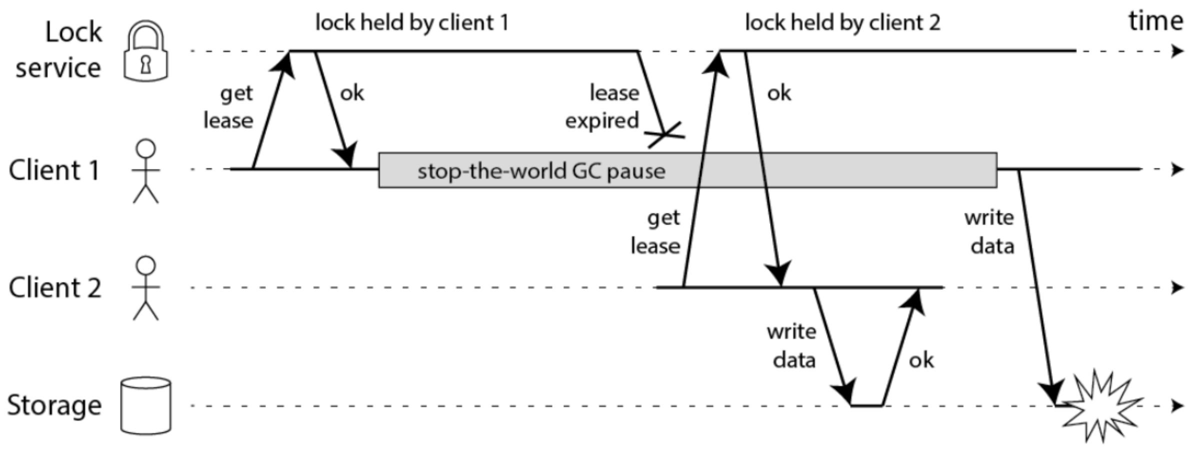

Figure 2: **Unsafe Locking**

Leader Failover Chubby's lock service typically consists of a cell of 5 replicas, with one of them being designated as a leader. Clients interact with the leader, which then replicates all lock-service operations on the other nodes in the cell using Raft consensus. In addition to the leader maintaining a lease for each client, every client maintains its own lease timeout, which is used to detect a potential failure of the leader. If the leader's lease expires at a client, the client isn't sure whether the leader has actually failed or whether it has terminated its session. In case the leader has failed, the client must wait a certain period of time before a leader fail-over occurs and a new leader is elected. As a result, it enters a *jeopardy* state for a time duration referred to as the 'jeopardy period' (45s being the default). During this period, the client periodically tries to re-establish a session with the leader (by sending more KeepAlive messages) and if it is successful in doing so before the jeopardy period ends, the client's session has been saved. If the jeopardy period ends before being able to save the session, the client discards all session states and terminates the session. Figure 1 illustrates the function of a jeopardy period during fail-over operations.

2.4 Fence Tokens

Another problem that arises in the context of distributed locking is that failures and communication delays could lead to incorrect behaviour. Let's take the example discussed previously where an client that needs update a file in shared storage establishes a Chubby session with a lease and acquires a lock. In an unfortunate turn of events, it may happen that the client, for whatever reason, pauses for an extended period of time that is longer than the duration of its lease. The client, if when unpaused, does not realize that its lease has expired, would proceed to perform an update operation at the shared storage, assuming that it is in possession of the lock. However, in the interim period between the client's session expiry and the unpause operation, it may have so happened that a different client acquired the same lock from the lock service (which would have been reclaimed by the lock service when the original client's session expired) and we therefore now have two clients with the same lock, which could potentially perform concurrent writes to the shared storage, violating mutual exclusion and leaving the data in an inconsistent state. This scenario is depicted in Figure 2.

There could be a multitude of reasons due to which an application's write request reaches the server only after its lease expires:

- If the application is written in a language with a garbage collected runtime, stop-the-world garbage collection could lead to unpredictable and potentially long pauses
- If the application tries to read the contents of a page that have not yet been brought to memory from disk, the corresponding page fault could lead to a pause
- If secondary storage is accessed over the network (as in the case of a distributed file system), a disk read could be delayed by arbitrary network congestion
- It could be the case that the application is starved of CPU time because the operating system never schedules it (presumably due to many processes contending for CPU usage)
- Perhaps the process was accidentally sent a SIGSTOP signal, which could cause it to be suspended till a SIGCONT signal was sent

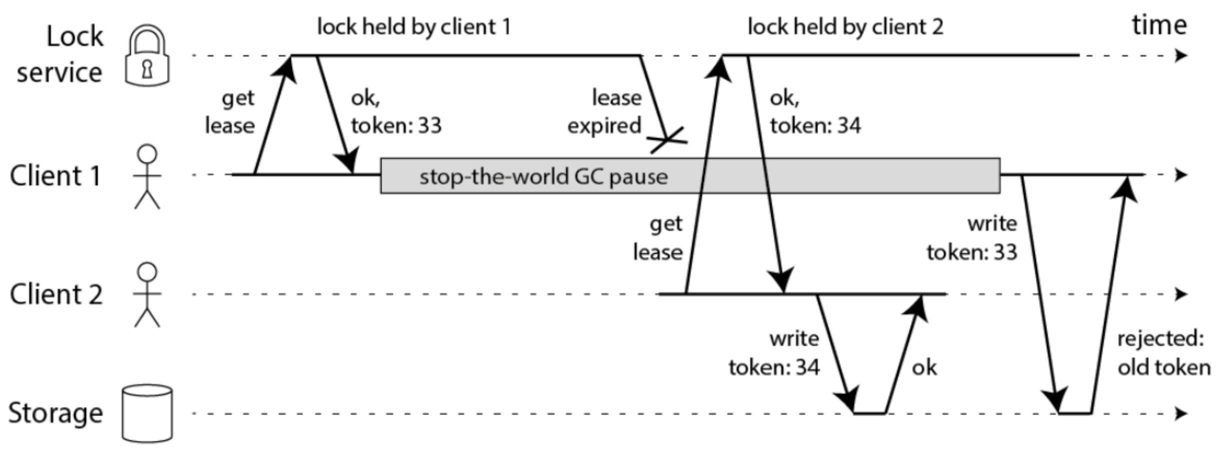

Figure 3: **Safe Lock with Fence Token**

- The file update request may get delayed in the network before reaching the storage service. As a result, it may so happen that the request reaches the storage server only after the lease has already expired

The solution to this problem involves the use of a 'fence token', which is a monotonically increasing value issued by the lock service on every lock acquisition, whose purpose is to disambiguate acquisition operations. To understand how using a fence token mitigates the problem discussed above, let's assume that $client_1$ acquires a lock and obtains a fence token with value 33. Immediately after it acquires the lock, it goes into a long pause and its lease expires. Since its lease expires, the lock server reclaims the lock, thus making it available for acquisition by other client. Now, a different client, $client_2$ acquires the same lock with a token of 34 (since the token monotonically increases), and then sends its write to the storage service with its token. On receiving this write, the storage server knows that the most recent node to have acquired the lock has a fence token of 34. Therefore, when $client_1$ comes back to life and sends its write to the storage service with token value 33, the storage service remembers that it has already processed a write with a higher token number (34), and so it rejects the request. This is illustrated in Figure 3. Similarly, Zookeeper's [6] zxid (or znode version number) can be used as a fence token to solve the same problem.

2.5 Failures

Since we use Raft for replication, each Chubby cell consisting of 5 servers can continue operation as long as a majority (i.e. 3) servers are up and running. Since Raft implements leader election and has built in mechanisms to handle failures, we rely on the library for fault tolerance.

2.6 API

The API exposed by our lock service is as follows:

`create_session()` - Used by clients to establish a session with the Chubby server. The server responds with a session ID and a lease for the session. The client must renew this lease periodically using `keep_alive()` failing which the server assumes the client has crashed and releases all its locks.

`delete_session()` - Used by clients to voluntarily terminate a session with the Chubby server. On receiving this request, the server releases all locks held by the client session.

`keep_alive(session_id)` - As discussed in section 2.3, the client is responsible for periodically invoking this API to inform the server about its liveness and renew its session's lease. The server blocks on this API and returns only when the session lease is close to expiring.

`open(session_id, path)` - Before being able to acquire a lock file, the file must be created. This API is used to create a new lock file at the specified path. Our implementation deviates from Chubby in that we do not return a file descriptor on a successful call to `open()`. Each request must instead include the full path of the lock file being operated on. This choice reduces extra state that the server needs to maintain.

`acquire(session_id, path, mode)` - Used by clients to acquire access to a distributed lock. The client specifies the path to the lock file, (which must have been created using `open()` before the call to `acquire()`) as well as the mode in which the lock must be acquired. If the client requests an acquisition in exclusive mode, the call only succeeds if no other client has acquired the lock (either in shared or exclusive mode). If the client requests an acquisition in shared mode, the call succeeds if no other client has acquired the lock in shared mode. Shared locks are reference counted to keep track of the number of concurrent readers.

`release(session_id, path)` - Released a lock file. If the lock being released is a shared lock, its reference count is decremented.

`get_contents(session_id, path)` - This API is used by clients to obtain the contents of a lock file. For instance, a newly elected leader could write its ID to a lock file, which is then read by other clients to obtain information about the leader's identity. Similarly, the cluster's configuration could be stored in a different lock file, which could be read by any node.

`set_contents(session_id, path, contents)` - This API is used by clients to update the contents of a lock file. Nodes in the system read the contents of lock files using `get_contents()` as described above.

3 Use Cases

In this section, we show how Chubby's API can be used to implement primitives that could be of use to distributed applications.

Locks Being a lock service, Chubby is naturally able to allow distributed clients to acquire and release locks, which can then be used for mutual exclusion or efficiency. A client first creates a lock file with the `open()` API call by passing in a path name as an argument. Once the file has been opened, the client can then acquire the lock in either shared or exclusive mode with the `acquire()` API call. Locks are released either explicitly using the `release()` API call or when the client's lease expires. Unfortunately, locking in Chubby suffers from the herd effect, where if there are many clients waiting to acquire a lock, they will all contend to acquire the lock even though only one of them will eventually succeed.

Leader Election One of Chubby's primary uses is to elect a leader among a group of nodes and allow nodes to discover which node among themselves is the current leader. A simple way to do this using Chubby would be to run an election such that the first node to acquire a pre-determined lock file becomes the leader. This leader could then update the lock file with its identity or other metadata that nodes in the network might need.

Configuration Management More often than not, when a new leader is elected, it must change a number of configuration parameters and notify the other nodes of these changes. Doing this successfully requires two things: (1) We don't want nodes to start using a configuration that is currently being modified until it has been completely updated and (2) If the new leader dies before the configuration has been fully updated, we do not want other nodes to use a partially updated configuration. Chubby only helps enforce the first requirement. To update configuration, a leader would acquire an exclusive lock for the configuration file, write its updates and then release the lock. Therefore, while the configuration is being changed, no other node can read this configuration.

Group Membership Assigning specific nodes to a group is a common operation in distributed systems. This can be performed in Chubby by first creating a directory file to represent the group and then adding files under this directory for every node that wishes to be a member of the group.

Name Service Although we do not implement client-side caching in our project, it is worth noting that Chubby's caching uses explicit invalidation messages as opposed to time-to-live (TTL) fields used in systems such as DNS. As a result, clients need not keep polling name servers when their cache TTL values expire, thereby reducing the load on name services.

4 Implementation

We implement Chubby in Rust with gRPC using the tonic [3] crate. We use TiKV [2], which is a fault-tolerant key-value database that uses Raft for consensus. Our code can be found at `https://github.com/abhishekvijeev/chubby`.

5 Evaluation

The goal of our evaluation is twofold. First, we attempt to measure how well Chubby performs in the common case (i.e. for coarse grained locking where locks are acquired and release infrequently). Second, since Chubby has a single leader which could become a performance bottleneck with increasing lock acquisition rates, we study its scalability by sending hundreds of acquisition requests. In the experiments, we use 5 servers for Chubby.

Methodology We obtain measurements using Rust's time API. We measure wall clock time for every `acquire()` request by obtaining the system's timestamp

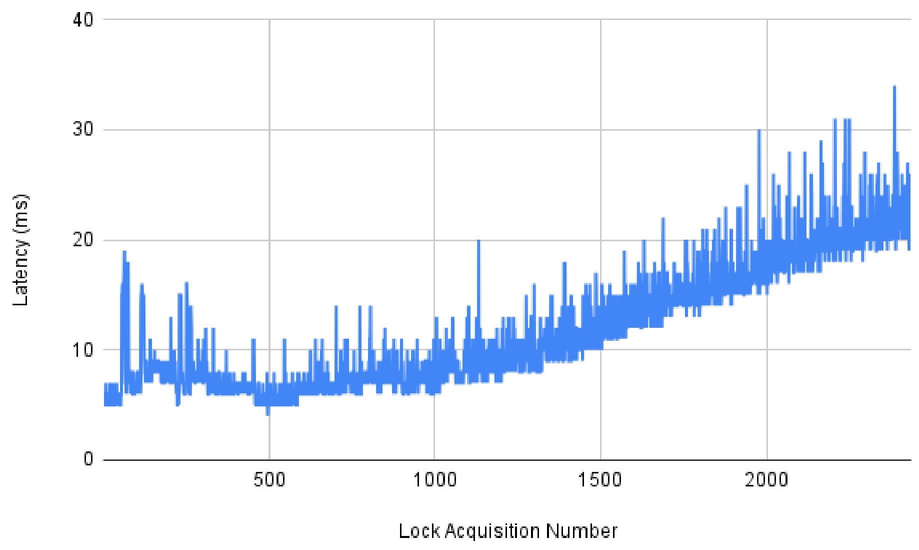

Figure 4: **Scalability**

at the beginning of the function definition and calculating the amount of time elapsed from this instant before the call returns.

Hardware Our evaluation was performed on a machine with an 8 core Intel Core i5-8250U CPU with a base clock frequency of 1.60GHz and 8 Gigabytes of DRAM.

5.1 Common Case Performance

In the common case, Chubby is able to respond to an acquisition request with an average latency of 6ms. We attribute a latency in the order of milliseconds to the fact that each request must be replicated on a majority of servers in the Chubby cell with Raft consensus before returning to the client.

5.2 Scalability

Since every lock service operation goes to a single Chubby leader, we expect this leader to become a performance bottleneck as acquisition rates increase. To measure Chubby's scalability, we measure request latencies by overloading the server with 2500 concurrent acquisition requests. The results are show in Figure 4.

As visible from the graph, the average latency per request gradually increases, starting from around 6ms and reaching about 20ms for the 2500th request. There are a couple of reasons for this observed behaviour. First, the server maintains state for every client session and therefore, the amount of state information increases with increasing load. Second, our implementation spawns a background thread for every session to monitor its lease and respond to KeepAlive requests. Therefore, the number of threads spawned grows linearly with the number of active sessions.

6 Additional Features

In this section we list out a set of features (from the original paper as well as other lock services) that have not been implemented in this project:

Access Control At present, we do not implement access control lists for locks, thereby allowing any client to acquire/release a lock that has been opened. We'd like to attach access control lists with every lock object to restrict the set of operations that clients can perform.

Caching Chubby's original design uses consistent client side caching, which allows clients to cache data without the need for specifying explicit timeouts. Instead, the Chubby server triggers event notifications (which clients can subscribe to) when cached data becomes stale. This mechanism makes it easy to use Chubby as a name service.

Events Clients should be allowed to subscribe to different kinds of events, that will be delivered asynchronously by the server. Events include file content modification, leader fail-over, primary election and conflicting lock requests from other clients.

Herd-free Locks Chubby's locks suffer from the herd effect, which basically means that when a lock is released, all clients that are currently waiting to acquire the lock contend for acquisition even though only one of them would succeed. Zookeeper mitigates this problem by using sequential counters on znodes and allowing clients to only watch for lock release events on the znode immediately preceeding them. As a result, only one client is awoken when a lock is released.

Atomic Broadcast We'd like clients to be able to talk to any node in the Chubby cell to balance load, instead of solely relying on the master. Zookeeper uses atomic broadcasts to correctly order write operations despite the fact that clients are allowed to talk to any replica. It also optimizes for read-heavy workloads by allowing clients to read from their local replica. Reads are partially ordered with writes by using a *zxid*, which corresponds to the last transaction seen by the server.

References

[1] How to do distributed locking. `https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html`.

[2] Tikv. `https://github.com/tikv/tikv`.

[3] Tonic. `https://docs.rs/tonic/latest/tonic/`.

[4] BURROWS, M. The chubby lock service for Loosely-Coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)* (Seattle, WA, Nov. 2006), USENIX Association.

[5] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP '89, Association for Computing Machinery, p. 202–210.

[6] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (USA, 2010), USENIXATC'10, USENIX Association, p. 11.

[7] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

[8] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1988), PODC '88, Association for Computing Machinery, p. 8–17.

[9] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)* (USA, 2014), USENIX Association, p. 305–320.