Hierarchical Chubby: A Scalable, Distributed Locking Service

Zoë Bohn and Emma Dauterman

Abstract

We describe a scalable, hierarchical version of Google's locking service, Chubby, designed for use by systems that require synchronization for a large number of clients accessing partitioned sections of data. Our service, termed Hierarchical Chubby, adds the ability to monitor overall system load, recruit additional machines as needed, and redistribute load when a machine becomes overwhelmed in a way that utilizes the locality hints provided by the client.

1. Introduction

Google's Chubby, a distributed, replicated locking service, provided a much needed mechanism for coarse-grained synchronization within Google developer systems, significantly reducing the amount of time and effort wasted building and debugging ad-hoc consensus protocols. Chubby was used by developers at Google to synchronize primary election, partition work among servers, and perform other routine, coarse-grained synchronization work within their systems.

chubby在伸缩性方面设计的不太够，因此通常用于内部的粗粒度的锁服务，比如用于选主。

One potential of Chubby not pursued in the original implementation, however, was the ability maximize scaling for large numbers of clients accessing partitioned sections of data. The authors of Chubby, aware of this, propose several methods for scaling Chubby, which we explore and expand upon in this paper.

本文介绍了一个多层级chubby设计来解决上述问题

This paper describes a *hierarchically scalable* version of Chubby designed to be used for large systems that require synchronization across a large number of clients accessing mostly partitioned data. Hierarchical Chubby (HC) adds several important features to the original Chubby design, inspired by those suggested in the original paper: 1) a means of monitoring overall system load, 2) a mechanism for recruiting additional clusters on demand, and 3) an ability to rebalance, or to redistribute locks to new clusters when a cluster becomes overwhelmed, in a way that utilizes the locality hints provided by the client.

HC由执行不同功能的多个集群构成

HC is composed of multiple clusters that perform different functions. A *cluster* is a set of machines that contain replicated state. In our implementation, each cluster consists of 3 machines running the Raft consensus protocol.

In order to enable each of the added features, HC uses a centralized *master cluster* to store metadata about overall system load. It uses this information to distribute work among its *worker clusters*. The master cluster recruits these worker clusters as demand on the system grows, and it redistributes the load among the worker clusters when necessary. When performing this redistribution, HC uses the locality hints it has received from the client to place locks likely to be accessed together in the same cluster. If a client uses locality hints well, this decreases the number of sessions a client must maintain with different worker clusters and thus maximizes the benefit of the redistribution.

Because of its ability to segregate locks based on client locality hints, HC provides a service best-suited to a system with large amounts of organized data that it expects to have many clients accessing in a segregated way; for example, a sharded database in which different sets of users perform transactions involving different sets of servers would potentially find our system useful.

Note that due to the proprietary nature of Chubby, we were unable to access its existing code base. Thus, before embarking on an implementation of HC, we first had to implement our own version of Chubby. Because of time constraints, we chose to implement only the simple locking service features of Chubby, as this was enough for a proof of concept. However, our system could easily be expanded to accommodate Chubby's other features, such as storing small amounts of metadata for a system.

The remainder of the paper is outlined in the following manner. In Section 2, we describe the client API of HC. In Section 3, we discuss the backend implementation of HC, highlighting where it diverges from Chubby. In Section 4, we discuss *rebalancing*, the mechanism that allows HC to scale as the number of locks and clients increases. In Section 5, we describe details specific to our implementation of HC. In Section 6, we evaluate our HC system. In Section 7, we describe potential future work. Finally, in Section 8, we conclude with a summary of what we have accomplished.

2. Chubby-Inspired API

Our client API is inspired by the Chubby API. Just as in Chubby, clients manage their locks in a hierarchical namespace. Locks are represented as pathnames, similar to the directory structure of a file system, where intermediate pathnames represent a domain (directory) in which the lock (file) is stored. HC has a single root domain '/' which can contain various locks ('/a') and subdomains containing other locks ('/a/b'). Domains not only allow the client to logically organize locks, but also allow the client to provide locality hints to HC. For example, HC expects '/a/c1' and '/a/c2' to be accessed together more frequently than '/a/c1' and '/b/c2'. Because of this, domains help determine the partitioning of locks in rebalancing.

Chubby allows clients to interact with it through a handle abstraction that appears as a pointer to an opaque data structure. We use a similar abstraction: an HC client calls *CreateLockClient()* to generate an opaque handle, which we refer to as a Lock Client, that can be used to perform locking operations. *DestroyLockClient()* can similarly be called when the client finishes. Clients can use the Lock Client handle to issue the following requests:

CreateLock(lock path)
CreateLockDomain(domain path)
TryAcquire(lock path) -> Sequencer*
ValidateLock(Sequencer*) -> bool
Release(lock path)

*The sequencer returned in *TryAcquire* and *ValidateLock* is modelled after a technique used by Chubby to make operations initiated under the protection of a client-held lock robust to client failure (more detail in section 3.2.1).

3. Normal-Case Backend Operation

HC使用一个集中式的master集群来维护系统整体元数据，并按需派发任务给worker集群

Where HC differs from Chubby most is in its backend implementation. Unlike Chubby, which uses a single cluster to service client requests, HC uses both a central *master cluster* and additional as-needed *worker clusters*. The master cluster stores metadata about the system and recruits worker clusters as needed to manage different lock domains. This design allows HC to scale as needed to service a higher volume of client requests. Furthermore, despite the greater spread of lock placement, under the assumption that locks under the same domain are more likely to be accessed together (and under our design, which attempts to place locks in the same domain on the same cluster), clients accessing disjoint sets of data should only need to maintain sessions with a few worker clusters.

The *LockClient* abstraction allows us to hide the master/worker cluster distinction from the client for ease of use, despite the fact that different client requests are serviced by different cluster types. The following sections describes the responsibilities of the master and worker clusters, respectively.

3.1. Master Cluster

In normal case operation, the master cluster has three main responsibilities: 1) allocating locks to worker clusters in response to client *CreateLock* requests, 2) allocating *domains* to worker clusters in response to client *CreateDomain* requests, and 3) locating the corresponding worker cluster for a lock that a client wishes to acquire. To service these requests, the master maintains a mapping of both locks and lock domains to the worker clusters that manage them.

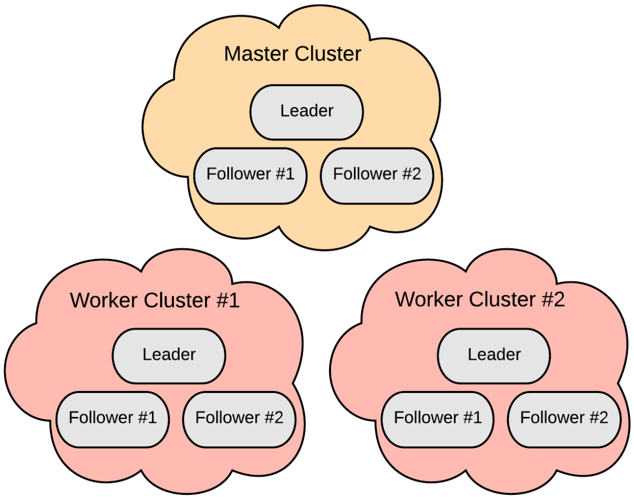


Figure 1. Architecture of Hierarchical Chubby: Master and worker clusters are each made up of three machines running the Raft consensus protocol.

3.1.1 Lock Creation and Allocation

Figure 2 shows the RPC sequence initiated when a client attempts to create a lock. Upon receiving a *CreateLock* request, the master checks its list of existing locks (to prevent duplicate lock creation). Once it determines that the lock does not already exist, it checks the lock's path to find its domain. It then uses its domain mappings to determine which worker cluster to allocate the lock to, placing it with other locks in the same domain. Finally it alerts the worker, waiting for confirmation before responding to the client.

3.1.2 Domain Creation and Allocation

The sequence for domain creation is even simpler than that for lock creation because the master does not need to alert the worker cluster to which it assigns the new domain. Domains are used solely as locality hints for lock placement, which the master is wholly responsible for.

3.1.3 Lock Location

When a client wishes to acquire a lock, it must first ask the master cluster which worker cluster to send the request to. Figure 3 shows the subsequent sequence of RPCs. As

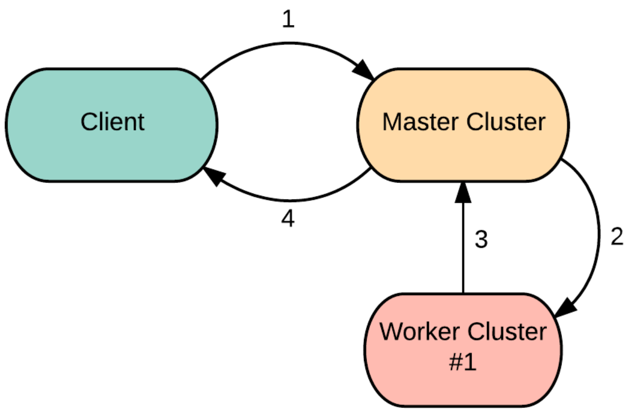


Figure 2. Sequence of RPCs involved in creating a lock:
1) CreateLock Request
2) ClaimLocks Command
3) ClaimLocks Response
4) CreateLock Response

an optimization, once the client receives the locational information from the master, it adds it to a local cache that it can service subsequent requests from.

3.2. Worker Clusters

Worker clusters are responsible solely for the locks given to them by the master cluster. They maintain state for each of these locks to ensure that only unheld locks may be acquired and that only the client that acquired a lock may release it.

3.2.1 Sequencers

For each lock, the worker keeps a corresponding sequence number (introduced in Section 2). Like Chubby, in order to allow clients to deal with failures while holding locks, HC maintains a sequence number for each lock, which it increments upon each successful acquire of that lock. It returns the current sequence number in its response to the client *TryAcquire* request. The client can then include this number with requests to other remote servers. These servers can validate the sequence number before processing the client request. If the sequence number has changed, then the server knows the client failed between the issuing of the remote request and its processing for long enough that its client session with HC timed out. Because workers release locks held by timed-out clients, another client was then able to acquire the lock (and increment the sequence number).

Note that if a client fails to acquire a lock, HC will return a sentinel value as the sequencer to show that the lock was not acquired.

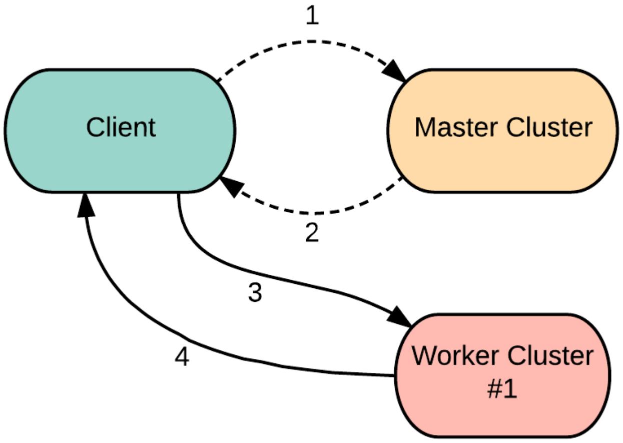


Figure 3. Sequence of RPCs involved in acquiring a lock:
1) LocateLock Request
2) LocateLock Response
3) AcquireLock Request
4) AcquireLock Response

4. Recruitment and Rebalancing

In order to scale as the number of locks maintained by HC grows, HC occasionally needs to recruit additional workers. The master can move locks from an overwhelmed worker cluster to a newly recruited one in a process we call *rebalancing* (outlined in Figure 4).

The need for recruitment is determined by the number of locks a worker cluster is currently maintaining, a number that grows as more locks are created under the domain(s) it manages. The master tracks the number of locks maintained by each worker cluster, and when that number exceeds a threshold, the master recruits another cluster and begins the rebalancing process in order to share load between the overloaded cluster and the newly recruited one.

First, the master determines which of the locks should be moved, using their pathnames to maximize locality when partitioning them between clusters. Next, the master issues a *RebalancingCommand* to the overwhelmed worker specifying the locks that should be moved.

Note that locks currently held by clients cannot be moved. This is because workers maintain sessions with each client holding locks so that they can release these locks when clients fail. These sessions cannot be transferred between workers. Thus currently held locks are not safe to move: we refer to these locks as *recalcitrant locks*. By extension, all other locks are safe to move.

Therefore, upon receiving the *RebalancingCommand*, the worker disables locks that are safe to move to prevent clients from acquiring them during rebalancing. The worker marks the remaining recalcitrant locks to be moved upon their release.

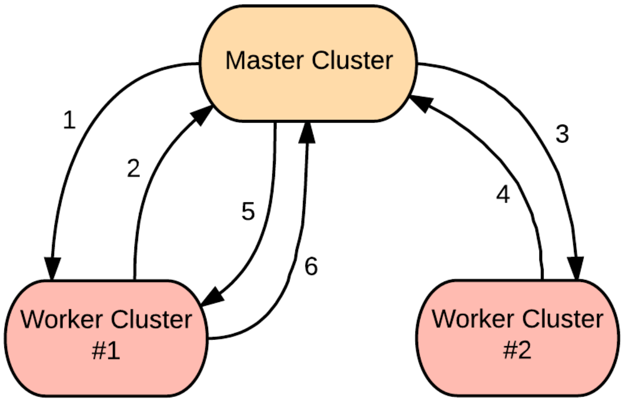


Figure 4. Sequence of rebalancing RPCs:
1) Rebalance Command
2) Rebalance Response
3) ClaimLocks Command
4) ClaimLocks Response
5) DisownLocks Command
6) DisownLocks Response

Finally, the worker responds to the master with the locks that are safe to move. The master subsequently transfers these locks to a new worker cluster. Once the new worker cluster acknowledges ownership of these locks, the master updates its internal mappings to reflect the new lock locations and sends an RPC to the original worker cluster, letting it know that it is safe to delete these locks.

4.1. Recalcitrant Locks

Recalcitrant locks remain at the old worker cluster after rebalancing is complete, but should be moved to the new worker as soon as possible. When a recalcitrant lock is released, the worker cluster immediately responds to the client, disables the lock, and alerts the master that the lock can now be moved. As before, the master then transfers the lock to the appropriate cluster, updates its internal state, and alerts the worker that it can delete its copy. The full process is outlined in Figure 5.

4.2. Locality Hints

Locality hints are central to the process of rebalancing because they help HC make informed decisions about where to place locks. The client gives HC domain hints by creating domains and placing related locks in the same domain. By trying to keep locks with the same domain on the same cluster, HC minimizes the number of worker clusters a client needs to communicate with (assuming a client groups all the locks it plans to use in a domain). This is important because clients maintain sessions with workers, which incurs overhead (see Section 5.2). In the future, this could also allow for more batching and caching.

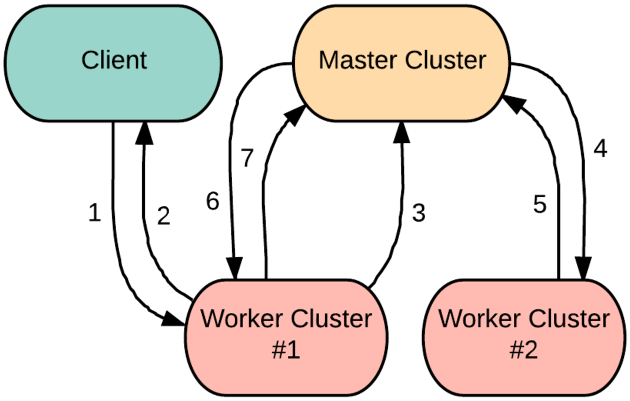


Figure 5. Sequence of RPCs involved in rebalancing a recalcitrant lock:
1) ReleaseLock Request
2) ReleaseLock Response
3) ReleaseRecalcitrant Alert
4) ClaimLocks Command
5) ClaimLocks Response
6) DisownLocks Command
7) DisownLocks Response

5. Implementation

Because the proprietary nature of Chubby precluded extension, we first implemented our own simplified version of Chubby using details in the paper. Our implementation is available at *https://github.com/zoebohn/Hierarchical-Chubby* and is approximately 2,700 lines of code (excluding the Raft library).

5.1. Consensus

To replicate state across machines in the master and worker clusters, we used the Raft consensus protocol. Master and worker operations are executed in master and worker finite state machines (FSMs) replicated with Raft. We used the hashicorp Golang implementation of Raft[1], which required us to implement a basic client library to allow the *LockClient* to communicate with HC clusters.

5.2. Client Library

Our client library supports communication with clusters via a single RPC or a client session. In a session, the client maintains a connection with the active leader by sending periodic *KeepAlive* messages. In a single RPC, the client library opens a connection, sends the RPC to the machine it thinks is the Raft leader, and closes the connection, repeating this until it knows that the RPC has reached the active leader. It is important to find the leader because the *ClientRequest* can only be handled correctly and replicated across all FSMs at the leader.

Both methods of communication use *ClientRequest* and

ClientResponse RPCs. A *ClientRequest* contains log entries to apply to the replicated FSMs, as well as state for maintaining a session, if necessary. To ensure correctness, all log entries applied to the FSMs are idempotent. A *ClientResponse* contains the success of the operation as well an address hint for the current leader. This leader address hint is useful because if the client contacts a Raft server that is not the leader (either when first contacting the cluster or in a session when another server is elected), the client can be redirected to the leader.

If the active leader fails to receive a *KeepAlive* message from a client that it has an open session with over a certain interval, it ends the session. The cluster must then update its state accordingly; for a worker cluster, this involves releasing all locks held by the client. To support this, we allow a *ClientRequest* to contain a command that is applied to cluster FSMs when a session ends. Note that clients do not maintain sessions with the master cluster, but simply send single RPCs because the master cluster does not need to be informed if the client fails. This helps to avoid a potential master bottleneck because clients do not need to continuously send *KeepAlive* messages to the master.

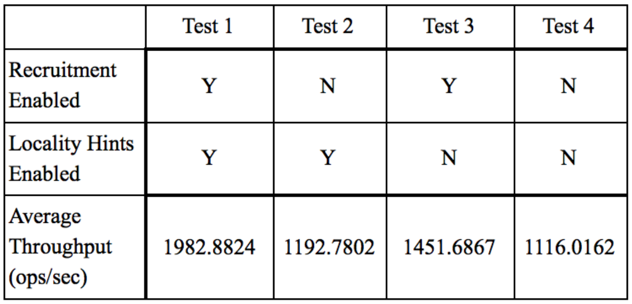
5.3. Callbacks

Our implementation of HC requires master and worker clusters to communicate directly. HC cannot do this communication within its replicated logs for several reasons:

1) Possible deadlock: Because communication needs to be done synchronously before responding to the client, communication must be blocking. This can lead to deadlock if two clusters simultaneously send messages from within their FSMs and each block waiting for the other's response.

2) Performance optimization: If each cluster has 3 machines (as in our implementation) and cluster A sends a message to cluster B and B responds, not using callbacks would require 3 + (3*3) = 12 messages, instead of just 2. Because our RPCs are idempotent, this would not cause the log entries to be incorrect, but would lead to a log entry being applied times instead of just once.

For these reasons, we added support for callbacks. After applying a log entry to the replicated FSMs, the FSMs may return a callback. This callback is run at the active leader, and while state from the FSM is used to generate the callback, the callback itself does not change FSM state. However, this callback may return a command that can then be replicated to all Raft servers, updating all FSMs in the cluster based on the execution of the callback. Because the leader executes the callback synchronously before replying to the client, if the client hears that its operation has been applied, the callback is guaranteed to have been executed in its entirety. Because all callbacks are idempotent, this ensures correctness.



	Test 1	Test 2	Test 3	Test 4
Recruitment Enabled	Y	N	Y	N
Locality Hints Enabled	Y	Y	N	N
Average Throughput (ops/sec)	1982.8824	1192.7802	1451.6867	1116.0162

Table 1. Results of tests varying recruitment and locality hints.

6. Evaluation

6.1. Correctness

Before evaluating the performance of our system, we first evaluated correctness through 14 end-to-end tests that verified that HC's behavior matched the Chubby-inspired API described in Section 2.

6.2. Performance

Our evaluation varied recruitment and locality hints to examine their effect on throughput. We deployed HC on the Stanford myth cluster with each machine using 2 Intel Core2 Duo CPUs and running Ubuntu 14.04. We chose to focus on throughput because our locking system is meant to be used by a large number of clients. By recruiting additional clusters, we hoped to be able to use these additional resources (particularly network bandwidth) to perform more client operations, allowing our locking system to scale. We focused on the throughput of acquiring and releasing locks because we expect these to be the operations clients continually use. The cost of creating a lock, which might include rebalancing, is only incurred once, and so, over thousands of locking operations, is amortized away. We expect that clients will use a fairly small working set of locks and repeatedly acquire and release these locks, and so we structured our tests to evaluate this type of workload.

6.2.1 Test Setup

We ran 4 tests varying recruitment and locality hints (see Table 1). In each test, 3 clients simultaneously acquired and released locks repeatedly for approximately 50 seconds. Each client had a set of 100 locks, and the lock sets were disjoint to prevent contention. The locks and domains were created before the test was started to exclude the cost of creation and rebalancing and focus on the most common operations.

To test the effectiveness of recruitment, we ran tests with recruitment (Tests 1 and 3) and without recruitment (Tests 2 and 4). With recruitment, we recruited 2 additional worker clusters to spread the load evenly across 3 worker clusters. Without recruitment, the load was concentrated at a single worker cluster.

To test the effectiveness of locality hints, we ran tests with locality hints (Tests 1 and 2) and without locality hints (Tests 3 and 4). With locality hints, we gave each client its own domain and configured the rebalancing threshold (when the master cluster triggered rebalancing) so that each worker cluster contained all the locks held in a single domain. Without locality hints, the locks were all in the root domain. However, in Test 3, each client's lock set was spread between all three worker clusters.

Because we had a limited number of machines, we ran each cluster (representing 3 machines) on a single machine and gave each client its own machine. Thus in Tests 1 and 3, we used 7 machines (1 master cluster, 3 worker clusters, 3 clients), and in Tests 2 and 4, we used 5 machines (1 master cluster, 1 worker cluster, 3 clients). While the service would not be deployed this way in practice, using a machine for each cluster still allowed us to test scalability. Because of variable delay caused by other workloads running on the myth machines, Raft consensus operations (e.g. snapshotting), and the network, we ran each test 5 times and took the average: these are the numbers shown in Table 1.

6.2.2 Analysis

From the results in Table 1, we can see that recruitment with locality hints allows HC to scale by almost a factor of 2. Recruitment with or without locality hints (Tests 1 and 3) allows some scaling because the system has more resources, most notably bandwidth, to process client requests. The master is not a bottleneck because once the client locates a lock, the client can cache the result and talk directly to the worker from then on, and the worker does not need to talk to the master for simple acquire and release operations.

Recruitment with locality hints (Test 1) further improves performance by reducing the overhead of sessions: the client only needs to send periodic *KeepAlive* messages to a single worker cluster as opposed to many worker clusters. We saw the overhead of sessions in the logs of worker clusters after tests without rebalancing: it was not uncommon to see that a client session ended and had to be reestablished. From this, we concluded that the worker cluster was too overwhelmed to process *KeepAlive* messages quickly enough, incurring the additional overhead of cleaning up and then reestablishing a connection both at the worker and client. Recruiting additional clusters while limiting the number of clusters the client needs to communicate with helps to solve this problem.

We were unsurprised to see no difference between using locality hints and not using locality hints when recruiting was disabled (Tests 2 and 4) because if no clusters are recruited, all locks are stored in a single worker cluster, regardless of the lock's name.

From these tests, we conclude that recruitment substantially improves throughput, and that locality hints allow HC to take advantage of recruiting most effectively.

7. Future Work

We plan to experiment with more sophisticated lock partitioning during rebalancing. Rather than always splitting locks into two equal-size groups, HC may sometimes benefit from keeping all the locks in a domain at one cluster, even if this gives one cluster more locks than another. This change could help further reduce session overhead.

HC could also benefit from taking into account the frequency at which a lock is being accessed in order to more accurately balance load between workers. This would require additional communication between the master and worker during rebalancing because although the master makes partitioning decisions, only the worker knows which locks are accessed most frequently.

HC could also avoid using unnecessary resources by joining clusters with low load. This feature would require first implementing a function to delete a lock in the Client API.

8. Conclusion

Hierarchical Chubby is a scalable, hierarchical locking service designed for use by systems that require synchronization for a large number of clients accessing disjoint sets of data. In addition to an implementation of a simplified version of the original Chubby, it also contains features that allow it to scale as the number of clients grows and to rebalance load among worker clusters in a way that preserves the benefit of the locality hints provided by clients. We plan to expand upon these benefits in our future work.

References

[1] Hashicorp golang implementation of raft *https://github.com/hashicorp/raft*.

[2] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006.

[3] Ongaro, D. and Ousterhout, J. K.. In search of an understandable consensus algorithm. In USENIX Annual Technical Conference, 2014.

[4] Ongaro, Diego. Consensus: Bridging Theory and Practice. Stanford University. 2014.

[5] Raft consensus algorithm website. *http://raftconsensus.github.io*.