UPPSALA
UNIVERSITET

IT 13 027
Examensarbete 30 hp
April 2013

A Distributed Lock Manager Using Paxos

Design and Implementation of Warlock, a Consensus Based Lock Manager

Sukumar Yethadka

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Distributed Lock Manager Using Paxos

Sukumar Yethadka

Locking primitives are one of the mechanisms used by distributed systems to synchronize access to shared data or to serialize their actions. Depending on the design, the locking service may constitute a single point of failure. This requires the manager itself to be distributed. Distributed solutions that address this using weak consistency models might lead to diverging states which in some cases are not possible to merge within acceptable effort. Solutions that are based on strong consistency models dictate the requirement of a static cluster.

We propose a design that combines Multi-Paxos algorithm with a reconfigurable state machine for a locking service. The primary goal of the service is strong consistency with availability and performance as secondary requirements.

We demonstrate the feasibility of such a design by implementing it in Erlang and testing it to check if it conforms to specified requirements. We demonstrate that it can provide the throughput required for a large web application while guaranteeing strong consistency.

Handledare: Paolo Negri
Ämnesgranskare: Justin Pearson
Examinator: Ivan Christoff
IT 13 027
Sponsor: Wooga GmbH, Berlin, Germany

Tryckt av: Reprocentralen ITC

CONTENTS

Context

Contribution

Summary

Appendices

LIST OF FIGURES

LIST OF TABLES

PREFACE

The idea behind Warlock was born out of a real project need at Wooga by Paolo Negri and Knut Nesheim. Paolo, as the thesis supervisor, helped me out with the project's conceptualization, feature set and figuring out some of the hard problems during the course of the project. Knut was very patient and helpful when trying to push Erlang to its limits. My deepest gratitude to them both.

Justin Pearson, Senior Lecturer at the Department of Information Technology, Uppsala University gracefully accepted to be the thesis reviewer and provided valuable input for the project. Olle Eriksson, Senior Lecturer at the Department of Information Technology, Uppsala University was really helpful as thesis coordinator with his excellent organization.

I would like to thank Anders Berglund, Ivan Christoff and staff of of the Information Technology Department for taking care of all the administrative formalities.

Wooga GmbH, Berlin hosted me for five months and provided me with all the resources for my thesis.

This project would not have been possible without the enthusiastic support from the people at Wooga, the Magic Land team and the Wooga back end team. Thanks to Marc Hämmerle for the lively project discussions and the name *Warlock*.

I would also like to thank Olle Gällmo from Uppsala University for the Project CS course. It allowed us to work on a large Erlang project and visit the Erlang User Conference 2011, Stockholm, where I was introduced to Wooga.

Thanks to Eivind Uggedal for compiling this beautiful LaTeX template.

Lastly, I would like to thank the scientific and open source community for all their contributions to further the progress in the field of distributed computing.

Sukumar Yethadka

Uppsala, Sweden
September, 2012

1 INTRODUCTION

Shared nothing architecture[1] (Stonebraker, 1986) is a popular distributed system design used for building high traffic web applications. The servers within such a system setup sometimes need to synchronize their activities or to access a shared resource. A service used for such a purpose could act as a single point of failure based on its design and the system architecture.

The online social game Magic Land uses a stateful application server architecture to handle millions of users. This architecture uses Redis as a global registry to keep track of user sessions. The registry acts as a locking mechanism allowing for at most of one user session at a time. Failure of this software, or the server it is deployed on can lead to application downtime. This creates a need for building a service that is itself distributed, fault tolerant and, primarily, consistent.

1.1 GOALS

The goal of this thesis is to build a customized locking service focused on being fault tolerant without compromising consistency or performance. Specifically we require,

- *Strong consistency*: The state of the system remains valid across all concurrent requests.
- *High availability*: The system should be operable at all times.
- *Performance*: The system should be able to handle a large number of requests.
- *Fault tolerant*: Server failures should be isolated and the system should continue to function despite these failures.

This locking mechanism can be realized by using state machines replicated across multiple servers which are kept consistent using distributed consensus algorithms. We investigate such algorithms and analyze similar projects. We then create a system design based on these observations and a working implementation.

[1] *Shared Nothing*: A type of architecture where the servers do not share any hardware resources between them.

1.2 OVERVIEW

Consensus in distributed systems is a large topic in itself. Consensus algorithms are notoriously hard to implement even though the pseudo code describing them can be relatively simple. Among the set of algorithms available for arriving at consensus in an asynchronous distributed system, we choose Paxos because of its literature coverage and usage in the industry.

The Erlang programming language is built with the primitives necessary for creating a distributed system, such as stable messaging, crash tolerance and well defined behaviours. The Actor model of Erlang processes maps very well to the algorithms pseudo code. Using Erlang allows us to separate the details of the algorithm from the implementation specifics. This enables us to ensure accuracy and makes debugging simpler.

The idea behind the system design is that we use Multi-Paxos[2] for consensus, reconfigurable state machine algorithms to make the cluster dynamic and implement this in Erlang.

The Erlang implementation of this thesis is named *Warlock*.

[2] *Multi-Paxos*: A variant of the Paxos algorithm designed to reach consensus wtih fewer number of messages and steps.

1.3 ORGANIZATION

The thesis is divided into three main sections over multiple chapters.

The first section provides the context and background information related to the project. It details the problem background, related projects, the research area and the set of requirements the project is based on.

The second section describes the analysis, design and implementation of the project, and the experiments performed on it.

The final section discusses the project results and provides an outline for future work.

1.4 SCOPE

The scope of the project is to implement a reasonably stable application that satisfies the above goals and can be used in production. The application needs to have good test coverage and documented code. The thesis scope does not cover creation of any new algorithms or techniques, but rather builds on the basis of well known ideas in the field.

PART I

CONTEXT

2 BACKGROUND

The concept area of this thesis is a mixture of distributed systems and lock management. In this chapter, we introduce the basics of these fields for a better understanding of the thesis. We also look at the factors that need to be addressed for an effective execution of the requirements.

2.1 CAP THEOREM

The CAP theorem or Brewer's conjecture states that it is not possible to achieve consistency[1], availability[2] and partition tolerance[3] at the same time in an asynchronous network model (Gilbert and Lynch, 2002). A choice of two of these attributes must be made when designing a system in such a network.

In this project, we focus mainly on consistency and partition tolerance[4] as primary goal with availability as a secondary goal.

2.2 CONSENSUS ALGORITHM

Lamport (1978) first suggested that distributed systems can be modelled as state machines. The system as a whole makes progress when these state machines transition between states based on events. The events are generated by passing messages between the networked machines. To ensure that all the servers in the system are at the same state, they need to agree on the order of these messages.

Consensus is the process of arriving at a single decision by the agreement of participants in a group. Consensus algorithms allow a group of connected processes to agree with each other, which is important in case of failures. Solving consensus allows us to use it as a primitive to solve more advanced problems in distributed computing such as atomic commits and totally ordered broadcasts. This is a primitive that is related to a lot of other agreement problems (Guerraoui and Schiper, 2001).

In an asynchronous network, consensus is simple when there are no faults, but gets tricky otherwise (Lampson, 1996). Further-more, in such a network, no algorithm exists that can reach consensus in the event of even one faulty process (Fischer et al., 1985).

Paxos is one of many available consensus algorithms, but its core is the best known asynchronous consensus algorithm (Lampson, 1996). It is covered in more detail in § 3.1 (p. 9).

[1] *Consistency*: Requests sent to a distributed system are said to be consistent if the result of the request is the same as compared to sending the request to a single node executing the requests one by one.

[2] *Availability*: Every request sent to the system must eventually terminate.

[3] *Partition tolerance*: Communication loss between a node sets in the network.

[4] Partition tolerance is an option that should always be a part of a distributed system. See Hale (2012).

2.3 DISTRIBUTED LOCKING

A Distributed System is defined by Coulouris et al. (2005, p. 2) as hardware or software components of networked computers performing activities by communicating with each other only by passing messages. Said definition gives it attributes such as concurrency[5], non usage of global clock[6] and the ability to handle independent failures [7].

One of the ways to co-ordinate concurrent processes in a loosely coupled distributed system is to use a distributed lock manager. It helps such processes to synchronize their activities and to serialize their access to shared resources.

A distributed lock manager can be implemented in many different ways. We use Paxos as our consensus algorithm since our system is asynchronous.

2.4 CHALLENGES

Building a distributed system has its own set of challenges. We identify the important ones below and address them in the design section Chapter 6 (p. 33).

2.4.1 *Scale*

Scaling in this context refers to an increase in throughput by increasing the number of machines in the network. However, in the case of a distributed consensus based locking system, the throughput is inversely proportional to the number of nodes in the network since it involves more messages and possibly more phases needed for agreement[8].

2.4.2 *Configuration*

Distributed systems need to plan ahead in terms of handling node failures and should provide ways to replace failed nodes. The system needs to support dynamic configuration to allow increasing and decreasing the cluster size as required.

2.4.3 *Failure Tolerance*

The set of servers in a distributed system is susceptible to failures. Machines occasionally fail, messages can be lost in transit, networks can be partitioned, disk drives can fail and so on. The system should be able to isolate the failures and make sure that it can function despite such failures.

One way to classify such failures is Byzantine[9] and non-Byzantine[10] depending on its origin. The software should be robust enough to tolerate such failures. This project only deals with non-byzantine failures.

[5] *Concurrency*: Simultaneous execution of processes (which may or may not be in parallel).

[6] *No global clock*: The computers in the distributed system are not co-ordinated using a single global clock and use other mechanisms such as vector clocks (Lamport, 1978) for it.

[7] *Independent failures*: Failure of individual computers does not lead to the failure of the entire system but, rather has other consequences such as degraded performance.

[8] Given x as the number of nodes, Du and Hilaire (2009) states that throughput of a system based on Multi-Paxos decreases as a $1/x$ function.

[9] *Byzantine Failure*: A faulty component sends conflicting messages to different parts of the system.

[10] *Non-Byzantine Failure*: A component either sends or doesn't send the message.

2.4.4 *Finding and Fixing Bugs*

Many things can go wrong in a distributed system (gal oz, 2006). The algorithms can be hard to implement, minute logical errors in the implementation might cause race conditions and it is hard to estimate time and message complexities in advance. This makes it hard to discover bugs, reproduce them, find their source and fix them.

2.4.5 *Testing*

Testing distributed systems is a difficult problem (Boy et al., 2004). Different types of tests such as unit tests[11], integration tests[12], system tests[13], load tests[14] are needed for a robust implementation. Furthermore, the implementation needs to be tested with different cluster sizes.

[11] *Unit Testing*: Testing individual "units" of code.

[12] *Integration Testing*: Testing that different components of the system work together.

[13] *System Testing*: Testing that the system as a while works well and meets specified requirements.

[14] *Load Testing*: Testing the amount of traffic the system can safely handle.

3 RELATED WORK

The design of Warlock is based on well known distributed algorithms. We discuss these algorithms here and list similar projects that meet few of our requirements (Chapter 4 (p. 21)).

3.1 PAXOS

Paxos is regarded as the simplest and most obvious of distributed algorithms (Lamport, 2001). It is a consensus protocol used for replication of state machines in an asynchronous environment (Lamport, 1998). We use Paxos in this thesis primarily since it has been shown that it has the minimal possible cost of any consensus algorithm in the presence of faults (Keidar and Rajsbaum, 2003).

A consensus algorithm tries to get a group of processes to agree on a value while satisfying its safety requirements[1]. These processes in the Paxos algorithm can be classified based on their roles without affecting its correctness:

- *Proposer*: A process that can propose values to the group.
- *Acceptor*: Acceptors form the "memory" of the algorithm that allows the algorithm to converge to a single value.
- *Learner*: The chosen values are "learned" by the other processes.

The algorithm proceeds in two phases with each phase having two sub phases.

- *Phase 1 a*: The proposer selects a number `n` and sends it as a `prepare` (`P1A`) message to all the acceptors.
- *Phase 1 b*: Each acceptor compares the `n` it receives and if it is greater than all previous numbers received as a part of `prepare`, it replies (`P1B`) with a promise not to accept any number lower than `n`. This response message also consists of the highest value `v` it has seen.
- *Phase 2 a*: If the proposer receives a response to its `prepare` messages from a quorum[2], it sends a reply (`P2A`) back to each of the acceptors with an `accept` message. The message also consists

[1] Safety requirements of a consensus algorithm (Lamport and Massa, 2004): (i) *Non triviality*: A value has to be proposed to be chosen. (ii) *Consistency*: Different learners cannot learn different values. (iii) *Conservatism*: Only chosen values can be learned and it can be learned at most once.

[2] *Quorum*: Majority agreement of processes. It is used to ensure liveliness in the system.

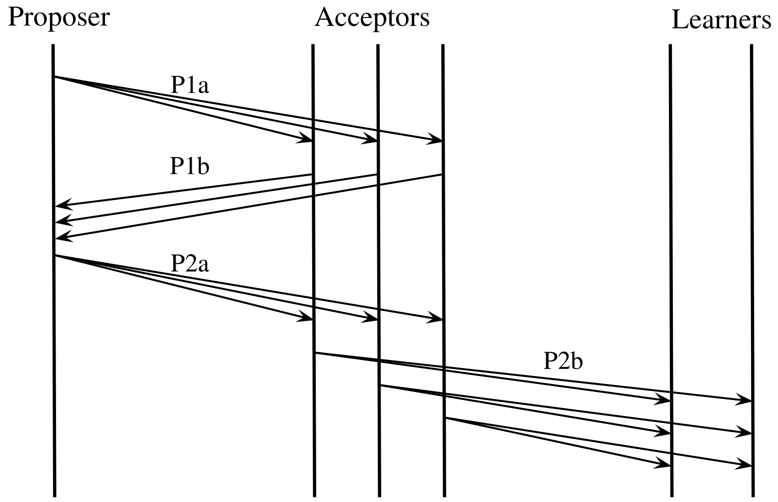

Figure 3.1: Basic Paxos Algorithm: Processes with roles – Proposer, Acceptor, Learner – send messages to each other illustrating the flow of the algorithm in a failure free instance.

of the value `v` which is the highest numbered proposal among all the responses from the acceptors. In case it is empty, the proposer is free to choose the value.

- *Phase 2 b*: If the acceptor has not received `prepare` request with a larger number when it receives an `accept` request from the proposer, it sends a message (`P2B`) to the learner with the accepted value `v`.

- *Learner*: If the learner receives a message from a quorum of acceptors, it concludes that the value `v` was chosen.

The algorithm makes progress when a proposed value is eventually learned by all the learners. However, a scenario where no progress can be made is possible when we have multiple proposers.

Consider two proposers issuing *prepare* requests with alternatively increasing *n*. They would keep preempting each other in a loop leading to no agreement being reached among the group. A solution is to create the role of *distinguished proposer* or *leader* where it becomes the only process that can issue new requests. Fischer et al. (1985) implies the "election" of this *leader* should use either randomness or timeouts.

Although the pseudo-code of the algorithm is relatively small, implementing it to create a stable, production ready system is non-trivial (Chandra et al., 2007). Different flavors of Paxos allows us to choose specific based on the project's requirements.

The family of Paxos algorithms differ from each other based on the topology of the process group, number of phases involved for one instance[3], amount of message delays and so on. We explore some of these Paxos variants.

[3] *Paxos Instance*: Single run of the algorithm starting from the value being proposed to the learning of the value by the learners.

3.1.1 *Basic Paxos*

Basic Paxos is the simplest version of Paxos and is the same as described previously in § 3.1 (p. 9). The algorithm proceeds over several rounds[4] with the best case taking two rounds. It is typically not implemented and used in production due to possible race conditions and relatively poor performance.

3.1.2 *Multi-Paxos*

We run one instance of Paxos algorithm for agreeing on a single value and multiple times for multiple values. We can batch together multiple values into a single instance, but this optimization does not reduce the message complexity.

Phase 1 of the algorithm become an unnecessary overhead if the *distinguished proposer* remains the same throughout. Multi-Paxos uses this as its basis to reduce the message count. The first round of Multi-Paxos (Du and Hilaire, 2009) is the same as Basic Paxos. For subsequent values, the same proposer starts directly with *Phase 2* halving the message complexity. Another proposer may take over at any point of time by starting with *Phase 1 a* overriding the current proposer. This is not a problem since the original proposer can start again with *Phase 1 a*.

van Renesse (2011) provides the imperative pseudo-code for Multi-Paxos and the details required for making it practical. We use it as a basis for implementing Paxos.

3.1.3 *Fast Paxos*

Fast Paxos (Lamport, 2005) is a variation of Basic Paxos, It has two message delays compared to four message delays of Basic Paxos and guarantees the round to be over in three message delays in case of a collision.

Clients propose the values directly to the *acceptors* and the *leader* gets involved only in case of a collision. Versions of Fast Paxos can be optimized further by specifying the collision resolution technique allowing the clients to fix collisions themselves.

However, according to Vieira and Buzato (2008) and Junqueira et al. (2007), Fast Paxos is not better than Basic Paxos in all scenarios. Basic Paxos was found to be faster in case of systems with small number of replicas[5] owing to the stability provided by its use of a single coordinator[6] and the variation of message latencies in practical networks. Fast Paxos also needs larger quorum sizes of active replicas for it to function.

3.1.4 *Cheap Paxos*

Basic Paxos requires a total of `2N+1` servers in a distributed system to tolerate `N` failures. However, `N+1` servers are enough (minimum) to make

[4] *Paxos Round*: A message round-trip between Paxos processes.

[5] *Paxos Replica*: A node that participates in the protocol.

[6] *Paxos Coordinator*: A Paxos process that acts as a leader by coordinating the message transmission between the processes.

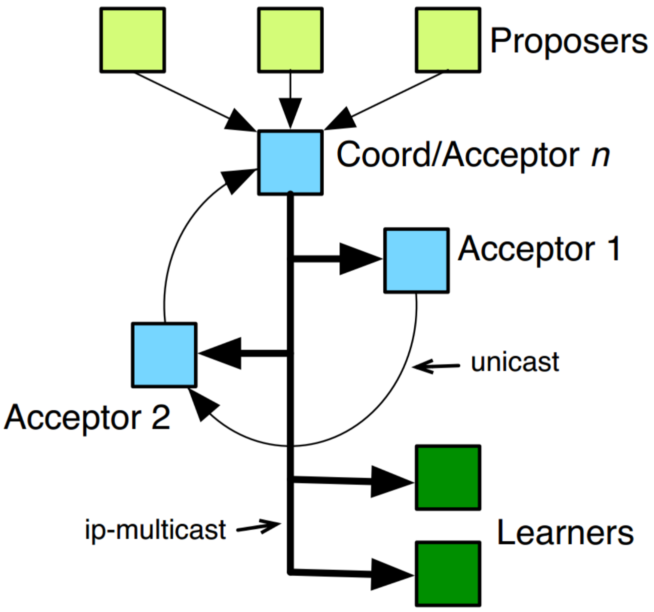

Figure 3.2: Processes and their roles in Ring Paxos. Figure courtesy (Marandi et al., 2010).

progress. Using servers that are slower or cheaper for the additional `N` servers allows us to reduce the total cost of the system. Cheap Paxos (Lamport and Massa, 2004) is designed along these lines.

Cheap Paxos uses `N` auxiliary servers along with `N+1` main servers which allows it to tolerate `N` failures. The idea is that the auxiliary server steps in to replace one of the main servers when it goes down temporarily. The main server takes back control once restored. The auxiliary servers thus act as a backup to the main servers without actively taking part in the protocol, merely acting as observers.

The downside of using Cheap Paxos is that it affects the liveliness of the system when multiple main servers fail at the same time since it takes time for the axillary servers to be configured into the system.

3.1.5 *Ring Paxos*

Ring Paxos (Marandi et al., 2010) is based on the observations that messaging using ip-multicast[7] is more scalable and provides better throughput compared to unicast[8] for a distributed system in a well-defined network. It has the property that it provides fixed throughput with variation in number of receivers. It claims to have the throughput of ip-multicast and low latency of unicast with the downside being that it provides weak synchrony[9].

Figure 3.2 illustrates the outline of Ring Paxos algorithm and shows the two communication protocols used between its processes.

[7] *Ip-Multicast*: The process sending messages to a group of receivers in a single transmission.

[8] *Unicast*: Transmission of message to a single destination.

[9] *Weak synchrony*: Message loss is possible.

3.1.6 Stoppable Paxos

The Basic Paxos algorithm is run under the assumption that all the participating processes are fixed and form a static system. However, the system should support cluster reconfiguration to be able to run for long periods of time in a practical environment. Reconfiguration includes adding new servers, removing/replacing old/faulty servers, scaling down the number of servers when lower throughput is acceptable and so on.

Stoppable Paxos (Lamport et al. (2008), Lamport et al. (2010)) is one such algorithm that allows us to reconfigure a Paxos based system. The algorithm defines a special set of *stopping* commands. A stopping command is issued as the `i`th command after which no new command at `i+1`the position can be issued. The system proceeds normally after it executes the `i`th command.

This thesis uses a variation of Stoppable Paxos for reconfiguration.

3.1.7 Other

Even though most of the Paxos papers detail the algorithm in pseudo code, it is non trivial to actually implement it. A few papers detail the fine points to consider from the implementation perspective.

Paxos for System Builders

Kirsch and Amir (2008) provides a detailed overview of Paxos from the implementation perspective. They list the necessary pseudo code required along with all the considerations needed to make the algorithm practical. Furthermore, they explore performance, safety and liveliness properties of the prototype they built.

Paxos Made Live – An Engineering Perspective

Chandra et al. (2007) details the learning in engineering the Paxos algorithm for use in Google Chubby Locks (Burrows, 2006).

The paper details the experience of engineers from Google in building a large scale system centered around Paxos. It details the major challenges faced and their solutions, performance considerations, Software Engineering techniques used and information of testing the setup.

3.1.8 Implementations

There are several implementations of Paxos and its variants. Listed below are implementations written mainly in Erlang. These projects serve as a good reference from the implementation perspective.

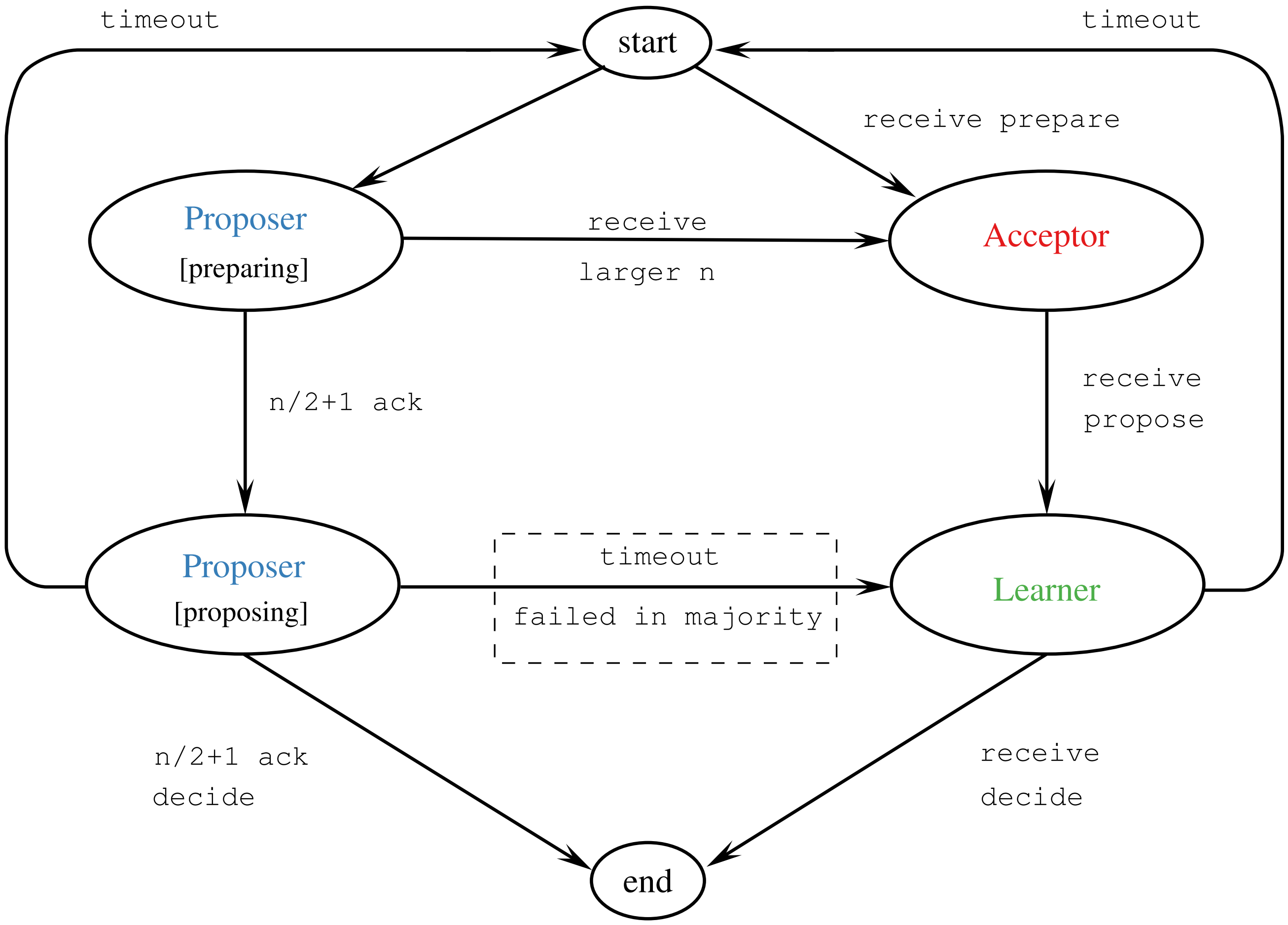

Figure 3.3: Figure shows Paxos algorithm when viewed as a finite state machine.

gen_paxos

(Kota, 2012) implements Paxos with individual processes modelled as finite state machines. Each process can be performing a different role based on what state it is on. This makes all processes equal and ready to take on different roles as required during runtime.

Figure 3.3 shows the view of Paxos algorithm as a Finite State Machine (FSM). The idea is that each process is run as an instance of this FSM, contrary to the regular view of process with a single well defined role. While this view of Paxos as a FSM is very helpful in the understanding of the protocol, we chose to use the protocol in the already established form of a single role per process.

LibPaxos

Primi and Others (2012) is a collection of open source implementations of Paxos created specifically for performance measurements in Marandi et al. (2010). It also includes a simulator written in Erlang to observe network behaviour.

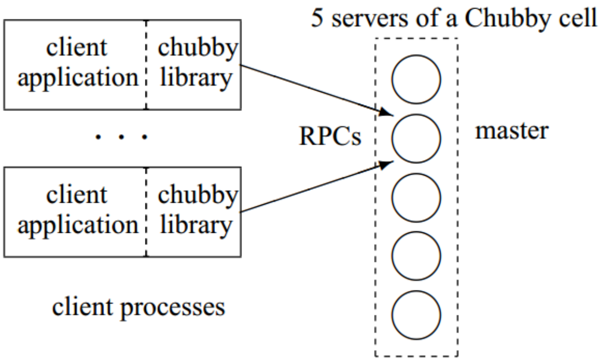

Figure 3.4: Figure shows the connection between Chubby cell and Chubby clients. Figure courtesy Burrows (2006).

gen_leader

Wiger et al. (2012) implements a leader election based on First In First Out (FIFO) basis. It is one of the notable implementations for leader election in Erlang.

3.2 GOOGLE CHUBBY LOCKS

Google created Chubby lock service (Burrows, 2006) for loosely-coupled distributed systems. It works as a distributed file system with advisory locks[10]. The goal of the project is to allow the clients to use the service to synchronize their activities. For example, Google File System (Ghemawat et al., 2003) and BigTable (Chang et al., 2006) use Chubby locks for co-ordination and as a store for small metadata (Chandra et al., 2007).

Chubby lock is made up of two components:

- *Chubby Cell*: Chubby cell is typically made up of five servers (termed replicas) that elect a master using Paxos protocol. The master server serves all the reads requests and co-ordinates the writes requests. The rest of the servers are for fault tolerance and are ready to replace the master should it fail.

- *Chubby Client*: Chubby client maintains an open connection with the Chubby cell and communicates with it via RPC[11] . The client maintains an in-memory write though cache that is kept consistent by the master using invalidations. The client is aware of the cell status using special requests[12] .

Figure 3.4 shows the network connection between Chubby cell and Chubby clients.

A Chubby replica, shown in Figure 3.5, mainly consists of a fault tolerant log that is consistent with other replicas in the cell by using Paxos

[10] *Advisory Locks*: Long term locks used specifically within an application.

[11] *RPC*: Remote Procedure Call: An inter-process communication technique where one process can run programs on remote processes.

[12] *KeepAlives*: Periodic requests used for indicating status.

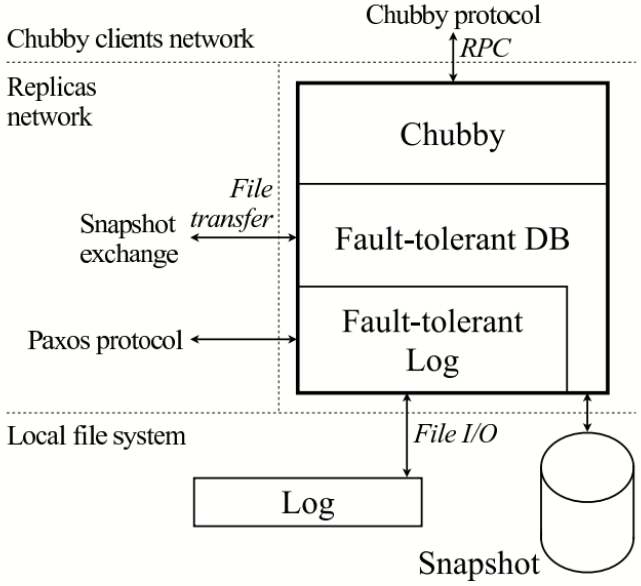

Figure 3.5: Figure shows the internal of a single replica inside the Chubby cell. Figure courtesy Chandra et al. (2007).

protocol. The rest of the replica is made of a fault tolerant database and an interface to handle requests from Chubby clients. The specific Paxos flavor used is Multi-Paxos with slight modifications such as having a "catch-up" mechanism for slower replicas.

The presence of the Chubby locks project and its use in some of the largest server installations is a testimony of the need for distributed lock managers. This thesis uses some of the ideas explored in Chubby locks, such as using the Paxos protocol to agree on a callback function that can eventually be run on the database component.

3.3 GOOGLE MEGASTORE

Megastore (Baker et al., 2011) is an ACID[13] compliant, scalable datastore that guarantees high availability and consistency. It uses synchronous replication for high availability and consistent views while targeting performance by partitioning the data.

Megastore users Paxos to replicate a write-ahead log, replicate commit records for single phase ACID transactions and as a part of fast fail over mechanisms. It provides fast local reads using a service called the *coordinator* which keeps track of the data version/Paxos write sequence over the group of replicas. It speeds up writes by pre-preparing optimizations and other heuristics.

Figure 3.6 shows the core architecture of Megastore. It illustrates the relation between different replicas and the coordinator.

[13] ACID properties are used to provide guarantees for database transactions. (i) *Atomicity*: A transaction is either executed completely or not executed at all. (ii) *Consistency*: The state of the database remains consistent after the transaction has been completed. (iii) *Isolation*: Transactions executed in parallel results in the same state as running all the transactions serially. (iv) *Durability*: All changes made by a transaction to a database is permanent.

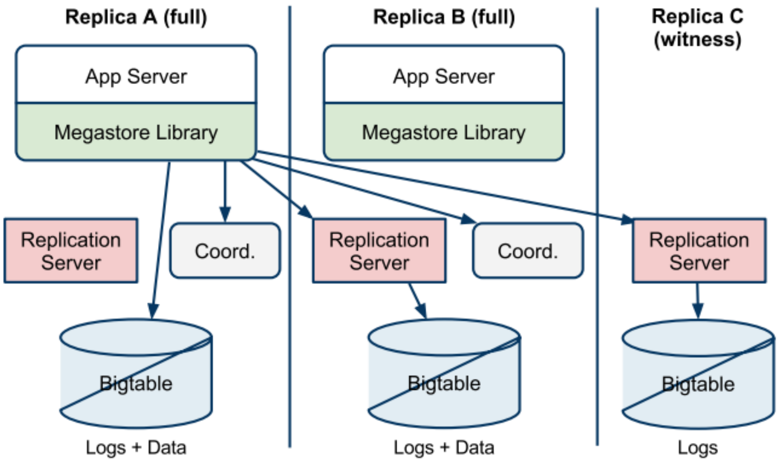

Figure 3.6: Figure shows the example architecture for Google Megastore. Figure courtesy Baker et al. (2011).

3.4 APACHE ZOOKEEPER

Zookeeper (Hunt et al., 2010; ASF and Yahoo!, 2012) is a open source consensus service written in Java that is used for synchronization in distributed applications and as a metadata store. It is inspired by Chubby lock § 3.2 (p. 15), but uses its own protocol Zookeeper Atomic Broadcast (ZAB) in place of Paxos.

3.4.1 *Zookeeper Atomic Broadcast*

Zookeeper Atomic Broadcast (ZAB) (Reed and Junqueira, 2008; Junqueira et al., 2011) is a totally ordered atomic broadcast protocol created for usage in Zookeeper. ZAB satisfies the constraints imposed by Zookeeper viz.,

- *Reliable delivery*: Message delivered to one server must eventually get delivered to all the servers.
- *Total order*: Every server should see the same ordering of the delivered messages.
- *Causal order*: Messages should follow causal[14] ordering.

ZAB is conceptually similar to Paxos, but uses certain optimizations and trade-offs. The service using ZAB has two modes of operation:

- *Broadcast mode*: Broadcast mode begins when a new leader is chosen using a quorum from the group. The leader's state is now same as rest of the servers and can hence start broadcasting messages. This mode is similar to two-phase commits (Gray, 1978), but with quorum.

[14] *Causal Ordering*: If message *a* is delivered before message *b* on a server then all other servers in the group should receive message *a* before *b*.

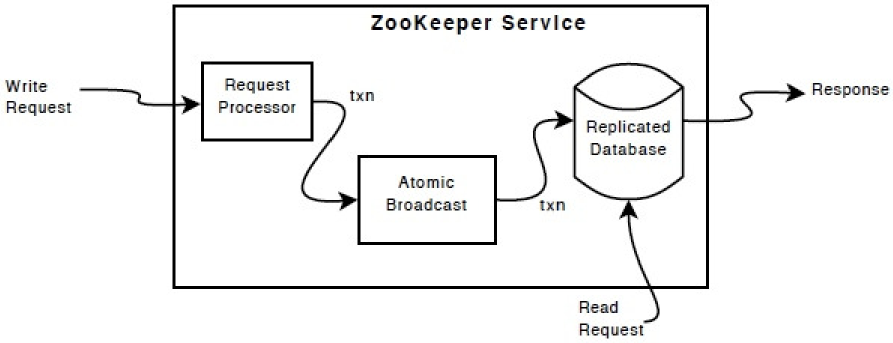

Figure 3.7: Figure shows the components of Zookeeper and the messaging between them. Figure courtesy ASF and Yahoo! (2012).

- *Recovery mode*: A new leader has to be chosen when the existing leader is no longer valid. The service is now in recovery mode until a new leader emerges using an alternative leader election algorithm.

Figure 3.7 shows the message flow between different components inside Zookeeper. We observe the optimization for *read* requests, which allows it to be much faster than the *write* requests.

Zookeeper uses the concept of *observers* to increase read throughput. They are a set of servers that monitor the Zookeeper service, but do not take part in the protocol directly thus acting as extended replicas. The write throughput is however inversely proportional to the number of servers in the group mainly due to the increased co-ordination required for consensus.

The data model of Zookeeper is that of a generic file system, which allows it to be used as a file system as well. It provides features such as access controls, atomic access, timestamps and so on.

Zookeeper worked on a static set of servers with no option to reconfigure the cluster at the time of writing. However, the feature was in the works (Shraer et al., 2012) and an initial release was available.

3.5 DOOZERD

Doozerd (Mizerany and Rarick, 2012) is a consensus service similar to Chubby locks § 3.2 (p. 15) and Zookeeper § 3.4 (p. 17) written in Go (Griesemer et al., 2012). It uses Paxos protocol internally for maintaining write consistency. Its use case is similar to that of Zookeeper and is mainly used as a fast name service. However, it is not as widely used or actively maintained as Zookeeper.

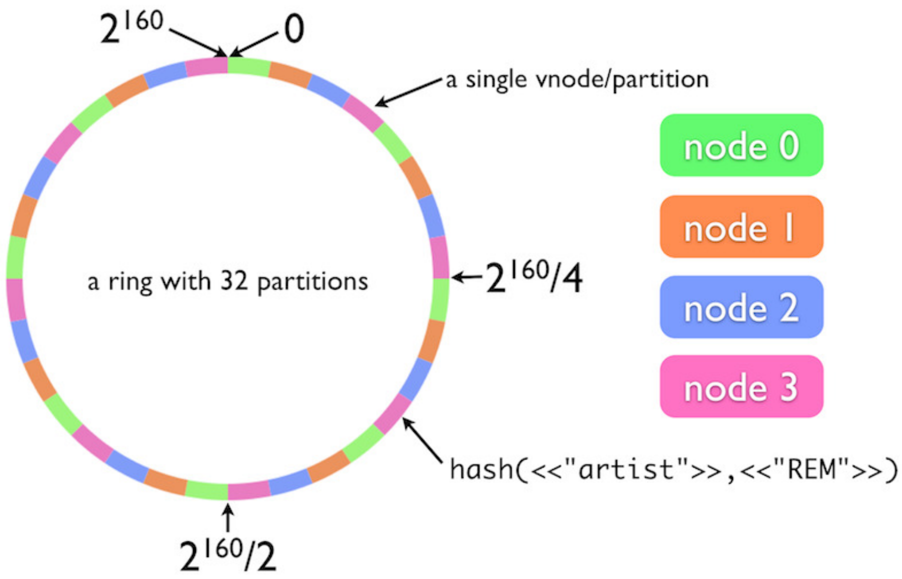

Figure 3.8: Figure shows the Riak Ring and details how the key space is divided among four nodes Figure courtesy Basho (2012c).

3.6 RIAK

Riak (Basho, 2012c) is a distributed eventually consistent datastore written in Erlang. It is based on the concepts from Amazon's Dynamo (DeCandia et al., 2007).

The primary use of Riak is as a distributed NoSQL[15]availability and partition tolerance while being eventually consistent. Figure 3.8 shows the distribution of the key space over multiple nodes using vnodes[16]. It might be possible to use this concept to scale key space and avoid having to store a complete copy of the data on all the nodes.

Riak is written in Erlang and is hence useful from this project's perspective for providing a good conceptual view of building distributed database applications in Erlang.

While the project is mature and satisfies all the other requirements to be used as a lock manager, absence of strong consistency makes its use untenable.

3.7 DYNAMO DB

Amazon DynamoDB (Amazon, 2012a) is proprietary key-value datastore based on DeCandia et al. (2007) available as a service. While providing the regular feature set of NoSQL data stores, it also provides strong consistency and atomic counters. While DynamoDB satisfies most of this project's requirements, usage of Warlock reduces the latency and provides quicker performance.

[15] NoSQL, generally called "Not Only SQL" or "Not Relational", are set of data stores that follow weaker consistency model than ACID and are characterized by their ability to scale their operations (Cattell, 2010).

[16] *Vnodes* or "Virtual Nodes" are a level of indirection used to map the key space so that any change in the status of the physical node does not affect the key distribution.

3.8 SCALARIS

Scalaris (Schütt et al., 2012) is distributed key-value database the supports consistent writes and full ACID properties. It is based on Distributed Hash Table (DHT)[17] and is implemented in Erlang. It also uses a non-blocking Paxos for consensus and provides strong consistency. It uses a Peer to Peer (P2P) protocol – Chord (Stoica et al., 2001) for its underlying data replication.

For our purpose however, Scalaris is a full featured database server with most of the features being left unused. Because of this, it affects its performance and needs more hardware for the necessary throughput.

3.9 SUMMARY

Warlock focusses on consistency and fault tolerance. The design of Warlock is influenced by the algorithms and architectures of the projects mentioned in this chapter.

Warlock differentiates itself from these projects by providing consistent operations while allowing for flexibility to add or remove nodes.

[17] *Distributed Hash Table*: DHT is a distributed system that provides hash table operations.

4 REQUIREMENTS

To explore the need for Warlock and to understand its requirements, we look at the architecture of the system and the derived requirements for Warlock in this chapter.

4.1 SYSTEM BACKGROUND

Magic Land (Wooga, 2012a) is a social[1] resource management game by Wooga[2] . The game is used by hundreds to thousands of users everyday resulting in thousands of HTTP requests every second. 90% of these requests are writes. This requires the back end to[3] handle lot of requests that are not cacheable. Traditional solutions which involves using stateless servers for application management in which all the state is managed by databases is not feasible for this access pattern. This led to the creation of an architecture comprising of stateful servers that handles all the user state changes and that uses database only for long term storage.

The system consists of the following components, as shown in the figure Figure 4.1.

- *Database*: The database is a persistent store used to store the user's session information for long periods when the user is offline.
- *Worker*: User sessions are run on the worker. Each user session consists of an stateful Erlang process that handles all the requests generated by the user for that specific session.
- *Coordinator*: The coordinator decides on which worker a user's session need to be started on.
- *Lock Manager*: Atmost one session of the user can be running at any given point in order to avoid creating conflicting states. This is achieved by using a lock service that needs to be checked before starting a new user session.

A typical user flow consists of the user trying to load the game. The request is sent to the coordinator which locates a suitable worker and asks it to start the session for the given user. The worker tries to create a lock on the users session by making a call to the lock manager with the user's id. On successful lock, the worker loads the user's state from the database and notifies that it is ready to accept requests.

[1] *Social Games*: Games running on a "social network" that allow interactions with users on the same network.

[2] Wooga (Wooga, 2012b) is a games company developing games on social networks and mobile platforms.

[3] *Back end*: The part of the system that handles all the user's actions and state and is not directly accessible to the user.

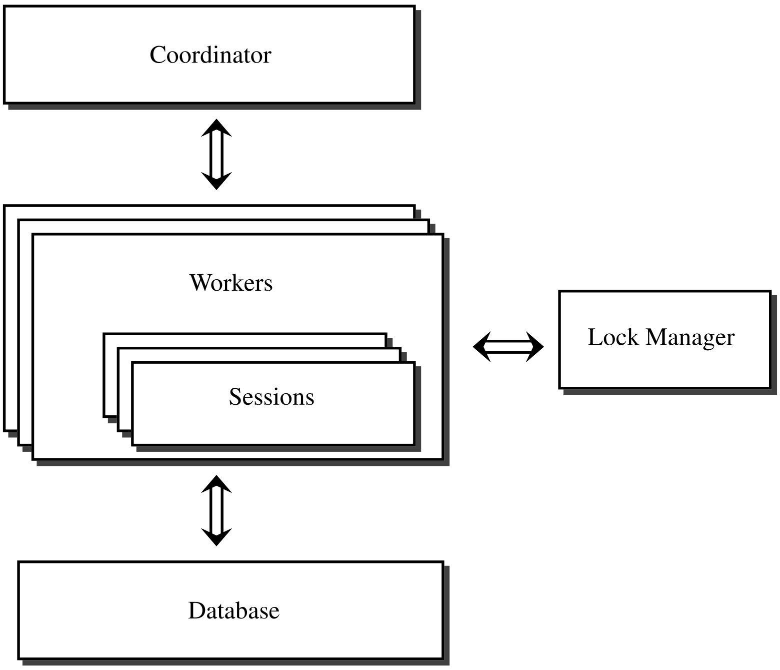

Figure 4.1: The figure depicts the high level view of the Magic Land system architecture.

The lock manager used is Redis (Sanfilippo and Noordhuis, 2012a)[4] . The worker uses the Redis command SETNX[5]. This makes sure that the worker can only start a new session if one is not already running. Redis also supports asynchronous replication allowing the data to be available in multiple locations.

While Redis is an excellent solution to the problem, it also becomes a single point of failure for the entire system since no new sessions can be created in the system if it is down. The goal of this thesis project is to try and replace Redis as a locking system while being fault tolerant.

4.2 PROJECT REQUIREMENTS

From the above background, we can now elaborate the requirements:

4.2.1 *Key Value Store*

The system should act as a key value store.

The user is referenced across the system uniquely using a numeric user id (UID[6]). A user's session present in any of the workers can be uniquely referenced across all the workers using the session's process id (PID[7]). The UID maps to PID and this mapping is stored in the lock manager. To support this, the lock manager should support the hash table primitives.

[4] *Redis*: A performance oriented key-value store with support for multiple data structures and atomic primitives.

[5] `SETNX key value` – is a command that sets the `key` to hold `value` in a hash table only if `key` does not already exist in the table.

[6] The UID is of the integer format.

[7] The PID is an Erlang process identifier.

4.2.2 *Strong Consistency*

All processes accessing the system concurrently should see the same results.

The lock manager will be accessed by multiple workers concurrently. This requires the manager to provide a consistent view to all the workers in order to avoid session duplication.

Without strong consistency, it is possible that multiple divergent user states exist at the same time. It might not be possible to merge these states within acceptable limits of effort.

4.2.3 *Maximize Availability*

The system should target for maximum possible availability.

Any downtime of the lock manager will translate to the game being unavailable for a lot of users. The manager should therefore target for high availability.

4.2.4 *High Read to Write Ratio*

The manager should be designed and optimized to handle large read to write ratio.

The UID to PID mapping is read by multiple workers many times during the life of a user session as compared to writes which happen only when the user tries to login. This means that the system can be optimized to handle a larger proportion of read requests in relation to write requests.

4.2.5 *Dynamic Cluster*

It should be possible to add/remove nodes from the system as long as a certain minimum number of nodes are available.

Individual servers within a distributed system are susceptible to failures. It should be possible to replace the failed servers without being forced to restart the system.

As the number of users in the game grows, so does the number of requests to the back end. The system should be able to handle additional load by allowing addition of nodes dynamically.

4.2.6 *Masterless / System with Master Election*

The system should not have a single point of failure.

The system should not use special nodes whose failure can lead to the entire system being out if service.

4.2.7 Fault / Failure Tolerant

The system should handle node failures gracefully.

Server failures in the system should be handled without affecting the service. Individual failures should not cascade to rest of the system.

4.2.8 Key Expiry

It should be possible to expire keys after a certain amount of time.

The system should support primitives that expires (deletes) the keys after a specified time. This feature allows us to clear the system even if a worker missed it during session cleanup.

4.2.9 Simple API

The system should have a simple API.

The system should have a simple interface and should be simple to communicate with.

4.2.10 Programming Language Support

The system should be written in Erlang.

Almost the entire back end stack of Magic Land is written in Erlang. Having the lock manager also implemented in Erlang will allow it to communicate with existing system in a more robust manner.

4.2.11 Performance

The system needs to have high throughput.

The system needs to be able to provide the high throughput required to handle millions of users.

PART II

CONTRIBUTION

5 CONCEPTS

In this chapter we introduce the important concepts required for this thesis.

5.1 PAXOS

5.1.1 *Terminology*

- *Proposal*: A request sent by the client that is to be executed on all the nodes in the cluster.
- *Decision*: A proposal that is successfully agreed upon by the cluster.
- *Master*: The node that is elected and is the only one who can handle proposals sent by the replica.
- *Master Leader*: The leader process running on the master node.
- *Master Replica*: The replica process running on the master node.
- *Slot*: Each of the decisions are discrete events. The events can be mapped to a log file. The indices of the transaction log file are termed *slot*s.
- *Node*: An independent Erlang runtime instance.
- *Membership*: The nodes in the cluster are called members. They can be a member of different groups based on their condition[1].
- *Lease*: The master node retains its status for the duration of the lease. Lease time dictates the maximum possible time for which data read could be stale[2].

5.1.2 *Algorithm*

We use Paxos § 3.1 (p. 9) as the consensus algorithm for the system. van Renesse (2011)'s "Paxos Made Moderately Complex" is the specific paper referred to for the implementation of Warlock. Firstly, we use this specific flavour since the detailed pseudo code specified maps very well to the Erlang's process oriented[3] design. Secondly, the paper discusses multiple optimizations required to make the algorithm practical.

[1] *Node Condition*: The condition of the node is a reference to the node's status in terms of handling the algorithm. Possible conditions are (i) *Valid*: A node that is ready to handle the messages and take part in the protocol. (ii) *Join*: A fresh node that is in the process of joining the cluster. (iii) *Down*: A node that was once a part of the cluster, but is currently inaccessible is moved to this state until it can be fixed manually.

[2] *Stale Data*: Data on the local node that is no longer consistent with the rest of the cluster.

[3] Erlang uses light weight processes for concurrency. These processes communicate using messages.

The processes of the group can be classified into different roles based on which part of the algorithm they are responsible for.

- *Replica*: Replicas are processes responsible for assigning a proposal number to an operation and handle decisions received.
- *Acceptor*: Acceptors are the "memory" of the algorithm. They keep track of which leader is currently in-charge to issue commands using ballots[4].
- *Leader*: Leaders receive proposals from replicas and try to coordinate the messaging to acceptors for that proposal. It uses ballots to track the execution order of proposals.
- *Scout*: A scout process is spawned by a leader to activate a specific ballot. It sends out prepare messages to the acceptors and tries to get a quorum acceptance for its leader.
- *Commander*: A commander process is spawned by the leader to try and get votes for a specific proposal.

Assuming the scout was already run and the current leader has its ballot as the largest one, lets see a typical flow of the proposal from its initiation to its execution skipping on the smaller details and corner cases.

1. The client creates a proposal based on the request. This proposal is uniquely identified by a ballot and contains the complete request information. This proposal is sent to the replica.
2. The replica checks if the proposal is a duplicate and if not it assigns a sequence number[5] to it before sending it off to the leader.
3. The leader, which has already run the scout, spawns a commander with the ballot and proposal information.
4. The commander sends out a message to all acceptors with the ballot information asking them to approve the proposal.
5. The acceptor respond positively if it has not seen a larger ballot and negatively otherwise.
6. The commander waits for a quorum (usually a majority of total acceptors). Once it has received majority of the approvals, the commander asks all the replicas to execute the proposal and exits.
7. The replica checks if the received decision is the next index on the consistent log and executes the proposal if it is.

The above use case constitutes a single Paxos instance among a set of processes. In general, the system consists of several of these processes running concurrently. In this scenario, the routing works as follows.

[4] *Ballots*: Monotonically increasing identifiers that are unique to a specific leader. Each leader has an infinite number of ballots.

[5] The replica is responsible for maintaining a consistent log of operations. This log is made up of slots with each of the slots indexed by a sequence number.

1. The client broadcasts its command to all the replicas.
2. Each of the replicas sends a *propose* message to all the leaders.
3. The leader sends the `P1A` message to all the acceptors via the scout.
4. The acceptor only replies to the sender with a `P1B` message.
5. On acceptance from a quorum of acceptors, the leader sends an `accept` message (`P2A`) to all the acceptors via the commander.
6. The acceptors respond with `P2B`, only to the sender.
7. The leader, on quorum response, broadcasts the decision to all the replicas.
8. Each of the replicas replies to the client.

The paper (van Renesse, 2011) details a few optimizations for state reduction and improving performance, making the implementation more practical. It also offers several suggestions from the implementation and deployment perspective.

5.2 ERLANG

Erlang (Ericsson, 2012d) is a general purpose functional programming[6] language built by Ericsson mainly to develop telephony applications (Armstrong, 2007). Erlang was build to handle large number of network requests with special attention directed towards handling failures.

The process is the concurrency primitive of Erlang. Each of the processes are isolated and have access to their own private memory. This allows building large scale applications with the Actor model[7] (Clinger, 1981). These processes are light weight since they do not map onto to the operating system's process structure. This allows Erlang to run thousands of processes concurrently. The process based concurrency also allows taking advantage of multi-core processors when parallelizing computations.

Erlang also supports hot code loading[8] which allows us to upgrade the system without restarting or disrupting the service.

The ideas of concurrency, fault tolerance, distributability and hot code loading behind Erlang maps well on to building large web applications.

5.3 OPEN TELECOM PLATFORM

Open Telecom Platform (OTP) is a collection of Erlang libraries. The OTP code is well tested, robust and provides design patterns allowing us to quickly build Erlang applications. We take a look at few of the OTP principles that is used in the project.

[6] *Functional Programming*: is where programs are run by evaluating expressions as opposed to imperative programming where statements are run to change state. The data used in this type of programming is typically not mutable.

[7] *Actor*: A process that can (i) Send messages to other actors. (ii) Spawn new actors. (iii) Specify the behaviour to be used when it receives its next message.

[8] *Hot code loading*: Dynamically updating running software without restarts.

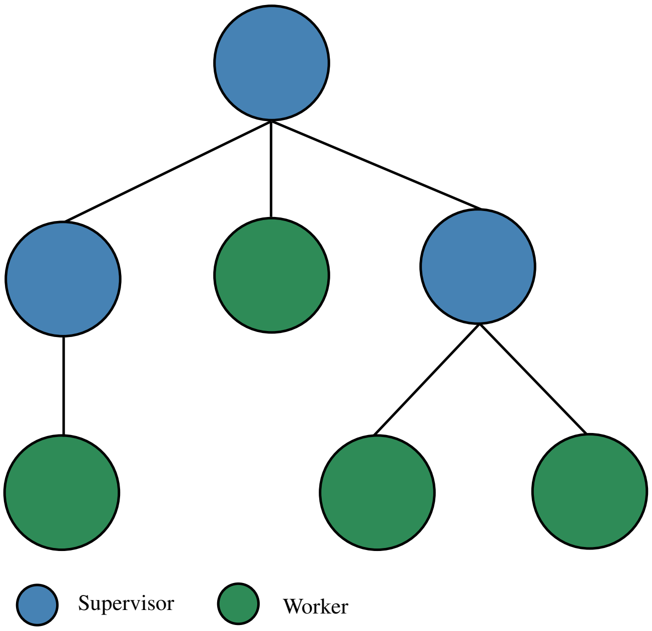

Figure 5.1: Supervision Tree: Structure of a typical Erlang supervision tree.

5.3.1 *Supervision*

Supervisors are processes that monitor worker processes and restart them based on predefined sets of rules. An application can be designed in the form of a tree with fine grained control to handle crashing processes. It can also localize such crashes.

A typical Erlang supervisor tree looks like Figure 5.1. The processes are connected to each other using links[9] . The links propagate upwards and are "trapped" by the processes that trap exits[10]. This allows them to detect and restart crashed processes.

A process as per the supervision tree can either be a supervisor[11] or a worker[12]. Processes that crash are restarted by the supervisor as per its restart strategy[13] . The supervisor also provides several other options such as child specifications, restart frequency, restart policy and so on to provide fine grained control on the restart procedure.

5.3.2 *Behaviours*

Behaviours are commonly used Erlang design patterns. They contain a predefined set of functions necessary to implement specific design patterns allowing for quick implementation.

The three main behaviours provided by the OTP library are

- *gen_server*: A process that waits for incoming events, performs actions based on the events and responds to the request.

- *gen_event*: A process that acts as an event manager waiting for

[9] *Links*: Bidirectional connections between processes with a maximum of one link per pair of processes.

[10] *Trap Exit*: When links are broken due to crashed processes, the failure propagates to the linked processes who themselves shutdown. Processes which should not stop even when links to it are broken can set a flag to do so. Such a process is said to trap exits.

[11] *Supervisor*: An Erlang process responsible for creating, terminating, monitoring and restarting processes as defined.

[12] *Worker*: A worker process in this context is any other process started by the supervisor that is not a supervisor itself.

[13] *Restart Strategy*: defines how the supervisor should restart crashed children. (i) *One for one*: Only the crashed process is restarted. (ii) *One for all*: All the children under the supervisor are restarted if any of the process terminates. (iii) *Rest for one*: Similar to one for all, but children restart is based on just the first process. (iv) *Simple one for one*: Same as one for one, but the children are created dynamically.

events to happen and runs specific event handlers subscribed to that specific type of event.

- *gen_fsm*: A process that acts as a finite state machine.

gen_server

gen_server is based on the typical architecture of client-server model (Birman, 2005). It supports two types of requests namely, synchronous calls[14] and asynchronous calls[15].

It also provides other features such as handling other types of messages (such as TCP/UDP messages), hot code loading and sending requests to other processes.

We use gen_servers to model the roles described in the algorithm.

5.3.3 *Applications*

Logical group of components can be grouped together to form applications. This allows us to start and stop a group and define orders for it. It also makes the code modular, promoting code reusability. Applications have well defined roles and Erlang provides convenient ways to manage them.

5.3.4 *Releases*

A release is the complete system packaged for deployment. Erlang provides modules necessary to create packages for deploying new code and upgrading existing code. This helps in rapid development and maintenance of the code.

[14] *Synchronous calls*: Also called blocking calls, has the caller wait till the process can provide a response.

[15] *Asynchronous calls*: Also called non-blocking calls, is received by the process and handled when it has processed all the messages it had received before this request.

ANALYSIS AND DESIGN

6

Warlock is a distributed consensus service custom made to be used as a lock manger. In this chapter, we discuss the design of the system based on the requirements detailed in Chapter 4 (p. 21). We explain the structure of Warlock and then detail how it maps on to the specified requirements.

6.1 ARCHITECTURE

The architectural goal for Warlock is to,

- Satisfy all the requirements specified in Chapter 4 (p. 21).
- Implement the system in Erlang while following OTP principles.
- Create a modular design to allow for customization for other projects.

The Warlock system can be divided into different components based on their functionality as shown in figure Figure 6.1. This figure illustrates the dependencies and communication paths between the internal components of Warlock. In terms of the data flow and interaction between the components, the system design is quite close to that of Chubby lock service (Burrows, 2006) as in Figure 3.4.

With the design goal of keeping the system modular, we separate logically distinct parts of the system into Erlang applications § 5.3 (p. 29). These applications interact with each other by function calls if they are libraries or by message passing if they are distinct processes. Below we detail the purpose and functionality of each of these applications.

6.1.1 *Utilities*

The utilities component provides the rest of the Warlock components with commonly used modules. The library consists of

- *Configuration Helper*: Reads configuration files to be used as settings for Warlock.
- *Hash Table*: A hash table implementation based on ETS[1] and dict[2]

[1] *Erlang Term Storage (ETS)* An in-memory storage provided by the Erlang Virtual Machine. It supports multiple data structures and operations over them, some of which are atomic.

[2] *dict*: Erlang's in-built dictionary implementation. Unlike ETS, it is immutable.

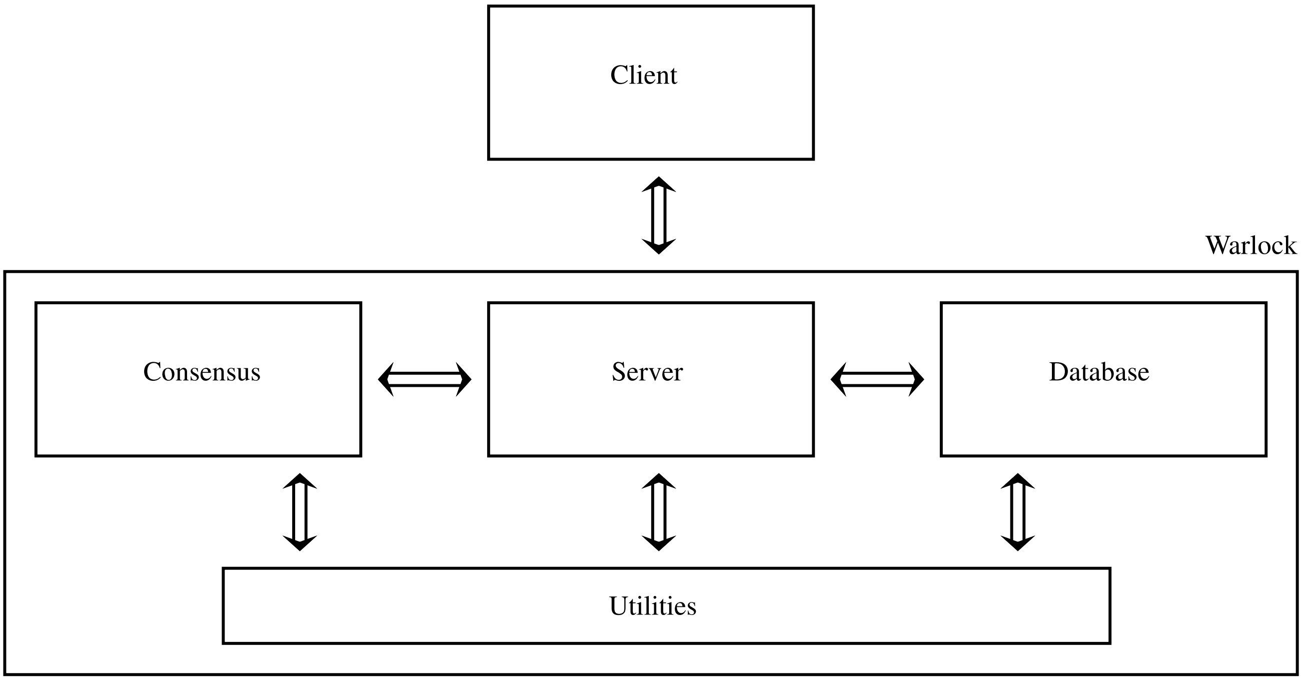

Figure 6.1: The figure shows the high level view of Warlock with its components and the messaging/dependencies between them.

Utilities is used as a dependency in rest of the Warlock components. It also defines Erlang macros[3] for enabling different levels of logging.

6.1.2 *Server*

The server component of Warlock ties all the other components together and indirectly routes messages between them. The main functionalities of this component are,

Handle client connections

Interaction with Warlock can be done in three different ways.

1. Embedding Warlock with the client application.
2. Accessing the system using RPC.
3. Using a well defined binary protocol.

The first two of the options are trivial to implement. For the last option, the server manages the incoming client connections which can then send requests. The client connections are over TCP/IP and use the Redis binary protocol (Sanfilippo and Noordhuis, 2012b)[4]. Once a connection is setup, it becomes an individual process unaffected by other connections, making it more robust.

[3] Erlang macros are similar to C macros. They allow us to define small functions and constants that is taken care of by the preprocessor during code compilation.

[4] The reasoning behind using the Redis binary protocol is that it is well defined and has a good set of features. It is also implemented in multiple languages allowing for usage from a much wider audience.

Replication

For the requirement § 4.2.5 (p. 23) of enabling a dynamic cluster we need the functionality to copy/replicate data between servers. This is done by assigning a seed node to the connecting node and transfer data between them. The steps for this are,

- All the nodes in the member group listen on a predefined port.
- A console command is executed on the new node that is to be added to the cluster. The address of a seed node[5] is passed to it as a parameter.
- The seed node sends the address information of the source node[6] to the target node (new node).
- The target node sets up a TCP connection with the seed node and sends a `SYNC`[7] signal.
- On the reception of the signal, the seed node does the following:
  - Change the status of the callback module to *passive*.
  - Ask the database component to backup the entire data to a predefined file.
  - Transfer the file to the target node using binary file transfer.
  - Once transfer is complete and the callback module has synced the data of the two nodes, request for a state reconfiguration via consensus.
  - Once callback queue is processed, reset it back to *active* state.
- On the reception of the data file, the target machine does the following:
  - Reset the local database.
  - Load the transferred file into the database.
  - Queue commands from the source node into its local queue until the file is loaded into the database and then execute the all the command in the queue, maintaining the order.
  - Get added to the members group and closes connection with the source node.

Callback

The callback module provided by the server component is executed once the request is processed. This allows us to keep the core of Warlock independent of the implementation of database module and to treat the commands as simple messages. This helps increase the robustness of the

[5] A *seed node* in this context is one that provided all the necessary information to setup a data transfer connection.

[6] Since data transfer is resource intensive, we do not use the master node as the seed. So the *source node* is picked to be a health cluster member other than the master.

[7] The `SYNC` signal is used to indicate that the target node is ready to receive the data.

consensus component by isolating the command execution and limiting its exposure only to the database component.

To enable replication § 6.1.2 (p. 35), the callback module needs to provide few additional features. The database component of Warlock has to be in read-only mode to avoid the risk of data corruption when it is a source node being replicated from. To allow this, callback has two modes of operation.

1. *Active*: The callback module's normal mode of operation where it accepts decision requests and forwards them to the database component.

   Change from active to passive mode can be done immediately.

2. *Passive*: The callback module does does not forward the decision requests, but adds them to a queue.

   When changing from passive to active mode, the queue is first processed asynchronously and mode change only happens when the queue is empty.

Console

The server component provides an interface that allows us to interact with Warlock from the command line. This allows us to start, setup and modify the Warlock cluster from the command line. Command line access helps us use automation tools such as Chef (Opscode, 2012) to rapidly deploy it on cloud servers (Armbrust et al., 2010; Amazon, 2012b).

6.1.3 Database

The database component is used to store all the data. It receives the decisions from the callback module and executes them. A typical database implementation is a key value store.

Besides the simple database interface, this component can provide data backup and restore functions that can be used by the server component to enable the Warlock feature of adding fresh nodes dynamically. The important characteristics of the database component are,

Pluggable back ends

As per the requirement § 4.2.1 (p. 22), the database needs to a key value store. However, any type of database can be used since Warlock by itself does not execute the commands. The commands are passed on to the database driver after a successful round allowing us the flexibility to use other kinds of database types. We create a specific behaviour[8] for this purpose. Drivers can then be written for different databases implementing the functions specified in the behaviour.

[8] Erlang behaviours (Ericsson, 2012b) are a set of specifications that can be used to create a generic reusable component.

Key Expiry

The option that allows stored objects to expire after a certain period of time enables the system to remove entries which were left behind when the cleanup process of the client application fails. However, key expiry in is a complex process in itself which can potentially lead to poor performance. For our system, we store the expiry time, but cleanup is only done when it gets accessed the next time. This allows us some flexibility while keeping the system performance predictable.

6.1.4 Consensus

The consensus component is the core of Warlock. It uses a modified version of the Paxos algorithm from van Renesse (2011) as described in Chapter 5 (p. 27) for its implementation. It also uses ideas from Lamport et al. (2008) and Lamport et al. (2010) to allow for a dynamic cluster. However, the algorithm needs some modifications § 6.3 (p. 38) before it can be used.

6.2 RECONFIGURABLE STATE MACHINE

In the regular Paxos protocol, the cluster nodes that take part in the it must be static. We need to explicitly design the system to handle the requirement § 4.2.5 (p. 23) of having a dynamic cluster. This can be done by following certain ideas from Lamport et al. (2008) and Lamport et al. (2010).

We can achieve this by treating the consensus state itself as a separate state machine along with the data state machine in parallel. Now we can use the Paxos protocol itself to reconfigure the system state which consists of the list of valid member nodes among other data. This modification allows us to add new nodes, remove existing nodes, replace nodes as required at any point of time.

For the implementation, instead of using the slots assigned for regular consensus operations, we use slots with alphabetic indexes allowing for better control during reconfiguration. This also allows us to make the reconfiguration immediately instead of blocking requests like in Lamport et al. (2008).

By default, the ballot[9] is made up of the leaders unique id and a monotonically increasing integer that increments whenever there is a change in the leader. It is possible that messages tagged with the old ballot is still in transit after a reconfiguration. We introduce a new variable in the ballot called *view* to avoid conflicts caused by this. *view* is a monotonically increasing integer that increments whenever there is a cluster configuration change.

The downside of this specific design is that if the master changes during reconfiguration, any of the proposals making progress will time out. It should be possible to fix this by transferring such proposals to

[9] *Ballots*: Monotonically increasing identifiers that are unique to a specific leader. Each leader has an infinite number of ballots.

the new master as new proposals by having the clients resend proposals that have timed out.

6.3 CONSENSUS OPTIMIZATION

The direct implementation of Paxos is not very efficient owing to the overhead that comes with duplicate messages, duplicate states and the detection of these duplicates at every stage. The default implementation also requires the entire history of the state be stored, making it impractical. This section describes the optimizations performed in this project, some of which are from van Renesse (2011).

6.3.1 *Backoff and Preemption*

Consider the following situation in a Paxos system with two leaders `L1` and `L2`.

1. `L1` initiates the protocol by issuing `P1A` message.
2. `L1` receives quorum acceptance and all the acceptors accept its ballot.
3. `L2` initiates the protocol by issuing `P1A` message.
4. `L2` receives quorum acceptance and all the acceptors accept its ballot.
5. `L1` initiates `P2A` phase only to be preempted with a higher ballot.
6. `L1` retries by issuing a `P1A` message again.
7. `L1` receives quorum acceptance and all the acceptors accept its ballot.
8. `L2` initiates `P2A` phase only to be preempted with a higher ballot.

As we can see, this leads to a race condition. To fix this, whenever a leader is preempted, it waits for a predefined period of time called the *backoff* time[10]. This provides the leader with the higher ballot sufficient time to make progress, thus avoiding the deadlock.

6.3.2 *Timeouts*

The leaders launch scouts and commanders depending on the phase the protocol is currently running in. These processes have to wait for the response from a quorum of acceptors after sending out their initial message. It is possible that not all of these acceptors respond due to reasons such as a network partition[11].

[10] *Backoff Time*: Backoff time can either be something constant or based on a multiplicatively increasing number. In most cases, using a constant backoff time should suffice, which is what Warlock uses.

[11] *Network Partition*; In a connected group of nodes, the connection between the nodes can be severed in such a way that it creates two separate smaller groups termed "partitions".

Introduction of timeouts in scouts and commanders allows us to timebox the protocol creating a maximum bound for response time to the client. The client itself can use timeouts, but this should always be greater than the internal Warlock timeout to maintain consistency.

In case a timeout occurs, the message is aborted. The client is notified of this which can then decide if it needs to resend the proposal.

6.3.3 State reduction

In the basic version of Paxos, the acceptor maintains all the accepted values for all the slots. This can be reduced by having the acceptor store only the value corresponding to the maximum ballot for each slot resulting in smaller acceptor state and `P1B` message size sent to the leader.

The acceptors are notified at the end of a successful operation. This allows them to store the values for only the slots that are currently making progress resulting in reduced `P1B` message size.

Similarly the replica only needs to maintain the data for slots that are making progress allowing it to purge data for decided slots. The leader can also purge the completed proposal data it maintains to spawn commanders.

All the above optimizations are possible by colocating replica, leader and acceptor inside each node. While it is possible to use shared data structures between these processes residing on the same node to improve performance , we avoid it to allow for a cleaner implementation.

6.3.4 Master Node and Leases

A leader that is not preempted can proceed directly with phase 2 of the protocol for a proposal cutting the message latency in half. We use the concept of lease to use this to our advantage.

The initial node whose leader manages to complete one round of Paxos is termed the master node. We define a period of time called the master lease during which leaders from other nodes cannot spawn scouts to push for a higher ballot. The master tries to renew its lease before it expires.

Failure detectors[12] are used to detect if the master node goes down, to accelerate the election of the next master node. The leaders from other nodes themselves try to get elected as soon as the lease of the current master expires. If the failure detector takes a long time, then the system as a whole stops making progress as long as the lease lasts. By tweaking the lease time, we can restrict the system to have an upper bound on the down time due to master node failure.

Usage of lease introduces a timing requirement into the system. The system assumes that there is a definite bound on clock drift[13] between the nodes.

[12] Erlang has built in failure detectors called monitors (Ericsson, 2012c). Using this feature it is possible to monitor the failure of local or remote processes.

[13] *Clock drift*: Due to physical and mechanical limitations in building clocks, the time maintained by individual clocks tend to drift away from each other. Many mechanisms can be used to correct the drift, allowing us to set a bound on this drift.

6.3.5 *Message Reduction*

Chapter 5 (p. 27) discussed the Paxos protocol used, detailing the number of messages sent between processes. By co-locating sets of processes and using a master node, we can reduce this number by a large extent.

In the reduced version, the client sends the message directly to the replica on the master node (master replica). The master replica only sends the proposal to the local leader (master leader). On receiving quorum acceptance, the leader directly responds to the client.

The above optimization allows us to reduce the number of messages from $O(n^2)$ to $O(n)$, where n is the number of messages transferred between the processes for a single consensus.

6.3.6 *Monitoring*

All the member nodes[14] keep a list of nodes that are participating in the protocol. The leader of the master node monitors the leaders of rest of the members‘ of the cluster. When the master leader detects any failures, it moves the node from the list of available nodes to the list of down nodes via the consensus protocol.

Failure of the master leader itself will eventually trigger the master election leading to a new master node. This node will then take over the role of monitoring rest of the members.

6.4 API DESIGN

We try and keep the Application Programming Interface (API) of Warlock as simple as possible. A typical command is of the format Source Code Listing 6.1

```
war_server:x(<location>, <command>)
```

Source Code Listing 6.1: Warlock’s generic command format

The parts of the format are:

- `war_server:x` is the module and function name which acts as the external interface.
- `<command>` is the operation requested by the client that has to be eventually executed by the database component.
- `<location>` indicates where `<command>` has to be run.

`<location>` has two possible values:

[14] *Member nodes*: Nodes which are in sync with rest of the cluster and actively take part in the consensus protocol.

```
1> (warlock@127.0.0.1)> war_server:x(clu, [set, key, value]).
{ok,success}
```

Source Code Listing 6.2: Warlock's cluster command format

```
2> (warlock@127.0.0.1)> war_server:x(loc, [get, key]).
{ok,value}
```

Source Code Listing 6.3: Warlock's local command format

- *Cluster*: Cluster (`clu`) indicates that the full consensus protocol must be run so that the command can be executed on all the servers.

  The commands are run on `clu` whenever its execution changes the database's state. For example, Source Code Listing 6.2

- *Local*: Local (`loc`) indicates that the command only needs to be run on the local replica the client is connected to.

  The commands that have no effect on the database's state can be run on `loc`. For example, Source Code Listing 6.3

The burden of deciding where the command has to be run rests on the client. Warlock has no way of knowing whether a specific command can change the database state. A command that changes the state and is sent as `loc` will be executed without any errors. However, this leads to creation of divergent states on the replicas, equivalent to data corruption.

6.5 READ WRITE SEPARATION

As per the requirement § 4.2.4 (p. 23) of supporting a high read to write ratio, we need to design the system to support read load that is a multiple (> 5 times) of the maximum possible write load. It also follows from the system design that each node (replica) has the entire dataset available and that we do not need consensus to access it.

Since the system is quorum based, not all of the nodes in the group take part in every round of consensus. This means that there are possibilities of a few slow nodes existing in the system. Reading data directly from these nodes might possibly get stale data.

The solution is to use a compromise between having fresh data and accepting chances of slightly stale data in lieu of performance improvement. The use of lease described in § 6.3.4 (p. 39) allows us to minimize the chances of stale data. The lease is renewed by the master node as per preset time (typically 5 seconds). We use this lease time at the individual node level to check if we are in sync with the rest of the nodes. This allows us to serve read-only data from individual nodes directly with a predictable limit on the age of data. In case the lease has expired, we return an error response to the client in line with the goal of serving consistent data.

Such a compromise allows us to reduce the message complexity of the system allowing for faster reads.

6.6 FAILURE RECOVERY

The system should be able to withstand problems such as node failures and data corruption. This is a hard problem to solve in system with static configuration since they have to implement complex error correction and conflict resolution mechanisms. However, since we support dynamic configuration, we can replace the problem node with a fresh node. Granular control allows us to detach the problem nodes with the intention of debugging the error.

The current design supports resetting a node's data and replicating from a member with good data. Improvements are possible in this area in terms of performance and usage of less resources as discussed in Chapter 9 (p. 61).

IMPLEMENTATION

7

We detail the implementation of the system as per the requirements (Chapter 4 (p. 21)) and design (Chapter 6 (p. 33)) in this chapter. We also explain how the design goals matches on to the features of Erlang programming language and OTP libraries.

The choice of Erlang as the programming language for implementation was motivated by the factor that the Magic Land back end was already implemented in Erlang. The idea is that a new system in the same language will be much easier to integrate with the existing architecture.

The development of the system was done in iterations[1], roughly a week in size. The target of each iteration was to extend the existing base or to develop new features. This allowed us to quickly identify the part of the system that forms the core and other features that are just good to have.

[1] *Iteration*: refers to the repeated execution of a development process. A typical process involves requirement analysis, design, implementation and testing. This is along the lines of Agile development (Kinoshita, 2008).

7.1 PROTOTYPE

The core of the consensus component is a variation of the Paxos protocol formulated in van Renesse (2011). While the algorithm is straight forward, implementation oriented and covers more edge cases than Lamport (2001), a working prototype helps to get a more through understanding of the protocol's intricacies.

We built a prototype that is a straight one-to-one implementation of the pseudo code from van Renesse (2011). It supports a configurable number of replicas, leaders and acceptors, allowing us to get an idea of how the protocol behaves with different configurations using simple unit tests. Building the prototype helps us single out the core of the system and make sure the protocol works correctly before getting to the other parts.

7.2 ALGORITHM IMPLEMENTATION

Each of the processes described in § 5.1 (p. 27) has some state and has to send and receive messages to and from other processes. We use *gen_servers* § 5.3.2 (p. 31) to implement these processes since process roles map on to this behaviour.

*gen_server*s can internally maintain state that can be accessed using all of its callback functions. It also provides functions to send messages[2] to one or more processes. The callback functions *handle_cast*[3], *handle_call*[4] and *handle_info*[5] are used to handle incoming messages.

Warlock uses *handle_cast* instead of *handle_call* since the protocol is for a system that communicates with asynchronous messaging. This also allows us to decouple the processes and take a step towards creating a lock-free system.

Let us take a look at the individual *gen_server* processes. Note that all of these processes reside on the same node inside an Erlang application. For each of the process roles, we describe the state it maintains and its the main responsibilities.

7.2.1 Replica

The replicas are responsible for maintaining the state of the system. They represent the state machine that updates based on the events sent to it. We have state machine replication by making sure we execute the events in the same sequence across all the nodes in the cluster.

In the original algorithm, the replica maintains a transaction log. The goal of the protocol is to keep this log consistent across multiple nodes. Instead of maintaining such a log, we execute the agreed commands (using callback functions).

State

- *Current slot number*: The strictly increasing property of slots can be used to ensure that the set of decisions taken by the algorithm are sequential and ordered.

  The current slot number is the next open slot available for a decision.

- *Minimum slot number*: The protocol allows for concurrent proposals. Each of the incoming proposals are allotted the next available slot and kept there until they are decided upon and committed.

  The minimum slot number is the next available slot for a new incoming proposal.

- *Proposals*: A table that maps proposals to slot numbers. It is implemented as a hash table.

- *Decisions*: It is sometimes possible that we receive decisions for higher numbered slots before the lower ones because the protocol is asynchronous and concurrent. Since it is also sequential, we have to wait until we commit all the lower numbered slots before we can proceed.

  Decisions is a table that maps decided proposals to slot numbers and is implemented as a hash table.

[2] *gen_server* messages can either be call (synchronous) or cast (asynchronous).

[3] *handle_cast*: *gen_server* callback function used to handle asynchronous messages. It cannot reply to the caller directly, but must use *gen_server:reply/2* function if it wants to do so.

[4] *handle_call*: *gen_server* callback function used to handle synchronous messages. It has the caller's address and blocks the caller until it sends a reply.

[5] *handle_info*: *gen_server* callback function used to handle all the other types of messages, e.g. plain Erlang messages, TCP, UDP messages and so on.

Role

- Receives proposals the leader, assigns slot numbers to them and forwards them to the leader.
- Receives decisions from the leader and commits them sequentially.

7.2.2 Acceptor

The acceptors form the *memory* of the protocol and ballots is the mechanism used for it.

State

- *Ballot Number*: The maximum ballot number it has seen in all of the messages it has received.
- *Accepted*: The leader sends a proposal to the acceptor along with its ballot for a specific slot. The acceptors stores the proposal for the maximum ballot it has seen for that specific slot. It is implemented as a hash table.

Role

- Responds to `P1A` messages with the `P1B` message, maximum ballot and the entire accepted table.
- Responds to `P2A` messages with the `P2B` message along with maximum ballot. If the ballot in the `P2A` message was equal to or great than the one in its state, it adds it as an entry in the accepted table.

7.2.3 Scout

The scout is a process spawned by the leader to handle the first phase of the protocol.

State

- *Leader*: Address of the leader that spawned it.
- *PValues*:[6] Unique set of PValues returned by the acceptor. It is implemented as a hash table.

Role

- Initiates the first phase of the protocol by sending `P1A` message to all the acceptors in the group.
- Waits for replies from majority of acceptors while collecting all the PValues sent by them. It only maintains the PValues for slots with maximum ballots.

[6] *PValue*: A tuple consisting of the ballot number, slot number and the proposal.

- Replies to the leader with the set of PValues if it gets majority acceptance.

7.2.4 *Commander*

The commander is a process spawned by the leader to handle the second part of the protocol. It is very similar to the implementation of the scout. A commander is spawned for every proposal.

State

Same as that of the scout, but instead of storing all the PValues, it only stores the PValue corresponding to the proposal it is working on.

Role

- Initiates the second phase of the protocol by sending `P2A` messages to all the acceptors.
- It waits for replies from a majority of the acceptors.
- On receiving the required acceptance, it sends a message to all the replicas indicating that the proposal has been accepted.

7.2.5 *Leader*

The leader process co-ordinates votes for proposals using scouts and commanders.

State

- *Ballot*: A ballot is unique to a leader and is used to tag all proposals being sent to the acceptors.
- *Active*: In multi-paxos, once the leader is in the second phase of the protocol, it does not need to run first phase of the protocol again.

  The *active* flag determines what phase of the protocol the leader is in. If it is set to true, the leader is in active phase allowing it to directly spawn commanders for every incoming proposal.
- *Proposals*: A mapping from slots to proposals. It is implemented as a hash table.
- *Timer Reference*: The leader uses a single timer based on its current status. This flag keeps track of it. Different possible values for the flag are
  - *renew*: The leader needs to renew its lease before it expires if it is in the master node.

    - *backoff*: When the leader's scout gets preempted, instead of immediately re-spawning the scout, the leader waits for a prespecified *backoff* time.
    - *master_check*: Periodic check by the leader to see if the existing master node's lease has expired. This is the check that is run before spawning a new scout.
    - *membership*: A new node being added to the cluster has to wait until it gets accepted. This flag is used to poll its acceptance.

- *Monitors*: Monitors are used by the master leader to keep track of the process status of leaders of the other nodes.

Role

- Receives proposals from the replica and either queues or spawns a commander for it based on its active flag's value.
- Handles communication between scout and commander, spawning them when required and acting on the results sent by them.
- Handles all ballot related activities such as incrementing the ballot when preempted and incrementing the view with the change in the system configuration.
- Initiates and participates in master node elections.
- Receives and acts on the messages resulting from monitoring of other nodes.
- Uses a single timer based on its current status to poll for the reasons mentioned.

7.2.6 Client

The consensus client process is only used for reconfiguration requests. The client behaves as a code library for the actual proposals. The replica uses the client as a execution point for the decided proposals.

7.3 SYSTEM RECONFIGURATION

As discussed in § 6.2 (p. 37), we use a parallel state machine to track the node's configuration. This state machine is implemented as a hash table. The operations we can support with this design choice are:

- *join*: A fresh node can join the consensus cluster without copying any data from the other nodes. This command is used to setup the initial cluster.

- *add_repl_member*: A new node can be added to the system by first copying data into it and then added to the cluster once it is in sync with rest of the member nodes.
- *leave*: A member node can leave the cluster.
- *remove*: Any of the members nodes can remove any of the other member node from the cluster.
- *transfer_master*: Nodes are designed to retain their master status for as long as possible. This command can be used to transfer the master status to any other node.

The reconfiguration requests are treated as proposals allowing us to use the same set of functions, albeit with different slots.

7.4 APPLICATION STRUCTURE

As described earlier, the code is split into different applications according to their functionality. A tree view of the source structure is provided in Appendix A (p. 71) for reference.

7.5 BUILDING AND DEPLOYMENT

A build[7] process is necessary for Warlock since it is made up of multiple Erlang applications. We also need to create a small cluster of Warlock nodes during development for observation and testing. This requires us to have an automated procedure to generate and deploy builds.

The final build consists of a portable Erlang runtime system, log paths setup and all the required libraries. rebar (Smith, 2012), reltool (Ericsson, 2012f) and make files (GNU, 2012) are the tools used to enable this automation.

[7] *Software Build*: Compiled set of source code ready to be run.

7.6 TESTING

The importance of testing increases when building a system that is developed rapidly in small increments. Even though we start with a fixed set of requirements, we might need to changes different components of the system in order to improve performance or add features. Having a solid set of tests allows us to make bold changes to the code while minimizing the chances of breaking working code.

7.6.1 Unit Testing

Testing individual units of code is unit testing. While we did not write tests for each and every part of the code as required by test driven development methodology (Beck, 2002), we wrote tests in a way that covered the critical code path. Some parts of the code that act more as a library has unit tests for it.

7.6.2 *Development Cluster*

The system needs distributed system tests. We automate the creation of a test cluster allowing us to quickly inspect, reproduce and trace bugs that appear only when the system is running as a part of the cluster.

7.7 LOGGING

Logging of events in the system is essential since it is one of the primary methods of debugging in a distributed system. It also allows us to track rare failures and helps during system implementation.

We use lager (Basho, 2012b), a library for logging. It has the flexibility of providing many log levels[8] and multiple outputs (console logging, file based logging).

The logging systems are designed with performance in mind. Even logging can have noticeable effects even if the log level is higher. The key is to balance the points where events are logged and to keep them at a bare minimum when used in live systems.

[8] *Log levels*: Log levels determines the severity of the event that is being logged allowing the developer to take appropriate action.

7.8 PLUGGABLE BACK ENDS

The Warlock project was designed as a lock manager to replace Redis (Sanfilippo and Noordhuis, 2012a). While the default Warlock system supports the operations required for the Magic Land back end, other projects using Warlock might need some other commands. A better solution than re-implementing the commands is to support multiple back ends.

So Warlock acts as a consensus layer and its callback can be used to run commands on different back ends. We have implemented back ends in ETS, Redis and Judy arrays (Baskins, 2012). It is possible to use other software as back end since Warlock is transparent to the commands.

7.9 PERFORMANCE

While consistency is the primary goal, the system also needs to focus on performance to handle systems with millions of users. Apart from the many optimizations described in Chapter 6 (p. 33), we use some tools to locate the bottlenecks and attempt to fix it without affecting the primary goal.

7.9.1 Profiling

Profiling is analyzing the code dynamically based on different measurements such as CPU time used, memory consumed, number of function calls made, number of messages sent and so on. It allows us to discover the parts of the source code that take the longest to run or is run the most number of times.

Erlang has a number of built in specialized profilers available (Ericsson, 2012e). It helped us identify that at one point logging was responsible for degrading upto 30% of performance even though it was set to a lower log level.

7.9.2 Benchmarking

Profiling allows us to look though the performance of the code minutely. Benchmarking allows us to see how the system performs from an external point of view. It is the act of testing the performance of a system by repeatedly running it or interacting with it.

Basho bench (Basho, 2012a) is the benchmarking tool used for the project. It allows us to define a number of options such as the number of concurrent workers, functions to call, rate of calls, system status measurements and so on. However, it was unable to generate enough traffic to saturate the system on a network while running from a separate server. While distributed benchmarking can be done to generate more calls, the coordination becomes harder and it gets difficult to merge the results.

7.9.3 Pluggable Hash Tables

Hash tables are used across the implementation of Warlock making its implementation choice a crucial factor from the performance perspective. Erlang provides many options:

- *ETS*: Erlang Term Storage is a built-in storage module in Erlang used mainly for storing tuples. It provides the option of multiple data structures and provides many atomic operations on them.

- *Dict*: Dict is Erlang's implementation of an immutable dictionary data structure.

- *gb_trees*: Erlang's implementation of General Balanced Trees (Andersson, 1999). It is an efficient tree structure that can be used as an hash table.

By using generalized hash tables in our code, we were able to evaluate these data structures finally choosing ETS for its performance and features.

7.10 ERLANG LIBRARIES

A few external projects were used for building this system. We describe the roles of these project.

7.10.1 lager

Lager (Basho, 2012b) is a logging framework for Erlang applications. It provides features such as fine grained log levels, multiple back end support, optimized logging and relatively better performance.

7.10.2 eredis

eRedis (Nesheim, 2012) is a non-blocking Erlang client library for Redis.

eRedis is used in this project to support the Redis back end and is used by the project to provide Redis protocol support.

7.10.3 Ranch

Ranch (Hoguin, 2012) is a socket acceptor pool for TCP protocols.

Ranch is used in this project for accepting and managing multiple client TCP connections for the Redis protocol implementation.

EXPERIMENTS AND PERFORMANCE

8

Warlock has to be performant in order to handle lot of traffic. We also need to detect side effects such as memory leaks when running the application for prolonged periods of time. Testing how the system scales with different parameters is also of interest. This chapter discusses the set of tests used for this and analyzes the results.

8.1 EVALUATION PROCESS

The test setup consists of a cluster of servers hosted on Amazon EC2[1]. We can test our system on different hardware configurations using the different *instance* types provided. The configuration of the two instances used in our tests are shown in Table 8.1

The test cluster consists of 3, 5, or 7 servers running Warlock and 1 server running Basho Bench (Basho, 2012a). The different cluster configurations tested are

1. 3 *m1.large* servers
2. 5 *m1.large* servers
3. 7 *m1.large* servers
4. 3 *c1.xlarge* servers

We have created different tests to test the performance variations with different parameters. We first test raw performance of the system with

Type	Attribute	Value
	CPU	4 EC2 Compute Units
m1.large	Memory	7.5 GB
	Platform	64-bit
	CPU	20 EC2 Compute Units
c1.xlarge	Memory	7 GB
	Platform	64-bit

Table 8.1: Amazon instances used for testing.

[1] *Amazon Elastic Compute Cloud (EC2)* A computing platform that allows us to provision servers on demand as a web service. For details, see http://aws.amazon.com/ec2/

different cluster configurations. Then we test the change in performance of the system with change in input parameters while keeping the cluster static. Specifically, the tests are

1. Write only requests only to master node.
2. Read only requests only to master node.
3. Write only requests to all nodes.
4. Read only requests to all nodes.
5. Mixed requests to cluster.
6. Variation in number of concurrent client connections.
7. Variation in read to write ratio.
8. Variation in payload size.
9. Different back ends.
10. Change in location of the bench client.

We also measure the system health in terms of CPU usage, memory usage and process count providing an idea about the resource consumption.

8.2 RESULTS

8.2.1 General Performance

Figure 8.1 shows the performance of Warlock based on type of requests when run for an hour. All the servers were of *c1.xlarge* type. The requests were generated by a single bench server with 10 concurrent connections[2] and handled by a 3 node Warlock cluster. The division of the requests based on the type was $^2/_3$ read , $^1/_6$ write and $^1/_6$ delete.

The results show that the Warlock can handle ≈ 5,000 writes/second, ≈ 14,000 reads/second and ≈ 8,000 mixed requests/second. The performance remains consistent throughout the test. This also helps us acknowledge that Warlock can handle the load generated in Magic Land.

Figure 8.2 shows us the change in request latency[3] with change in request types. We observe that the *GET* requests have the lowest latency. This is expected since these requests do not have to go through the protocol. The *PUT* and *DEL* requests are close together since they are very much similar. We also observe that *PUT* and *DEL* requests sent to the master have lower latency as expected since we save a message trip.

[2] The connection refers to the link between the client and the Warlock cluster. Each of these connections can be used to send thousands of commands/proposals.

[3] *Latency*: The total time taken to issue a request, process it and receive a response from the client's perspective.

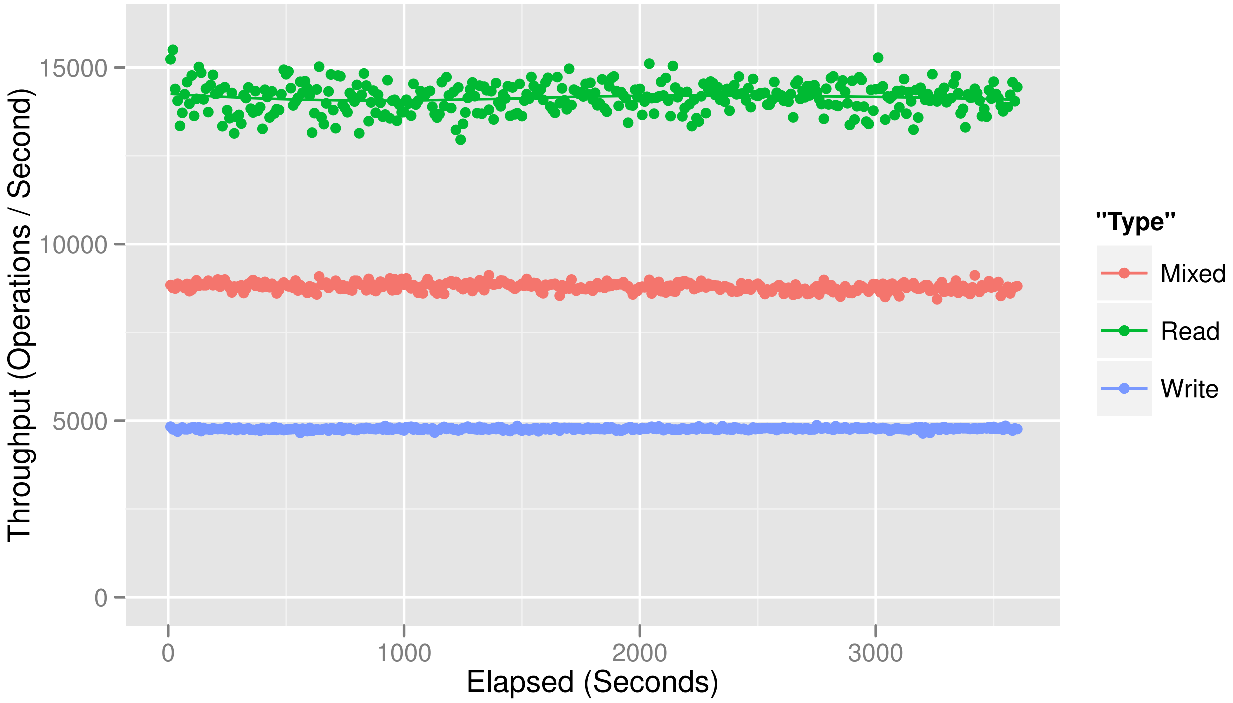

Figure 8.1: Performance test to evaluate throughput with different request types.

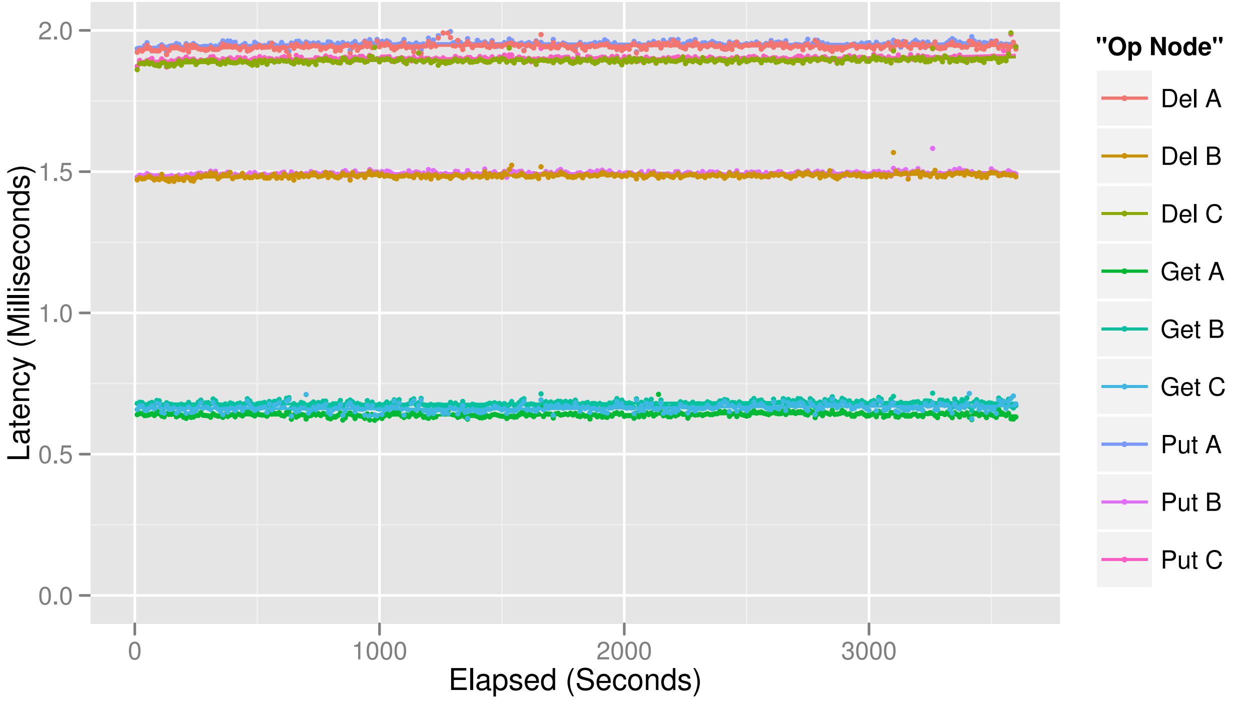

Figure 8.2: Performance test that shows us the change in latency based on the request types.

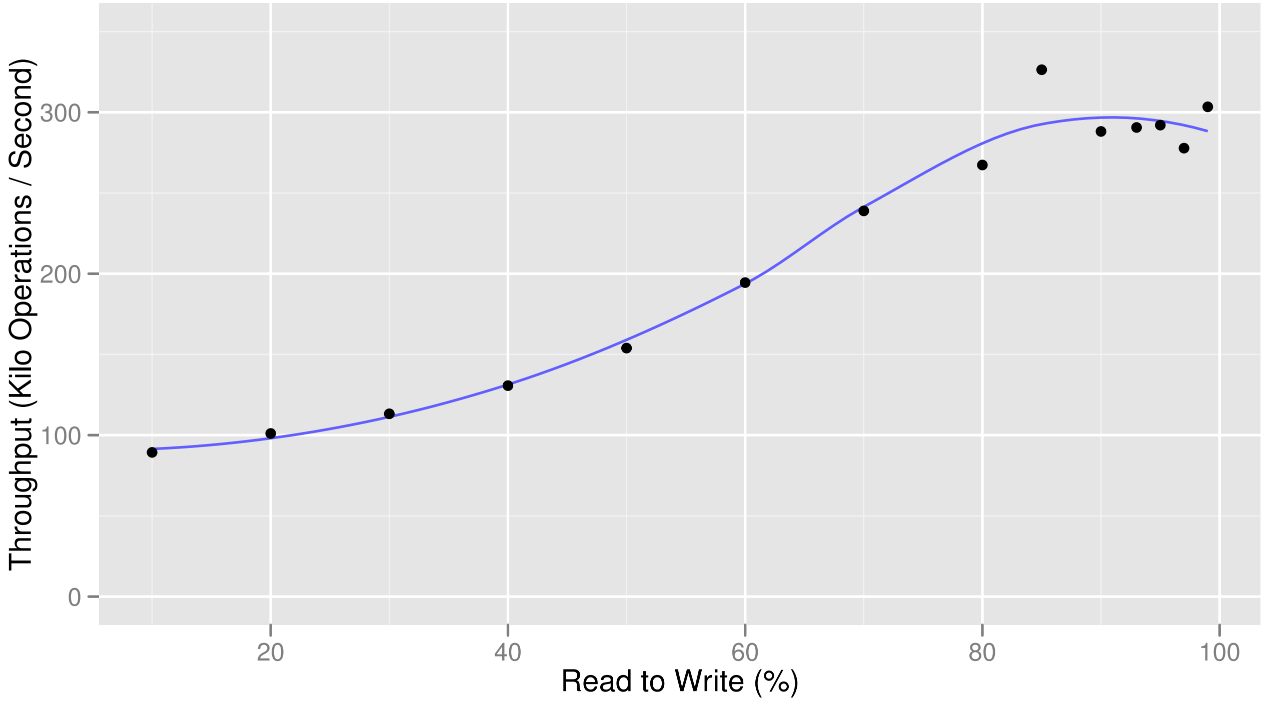

Figure 8.3: Test depicting the variation of performance with increase in read to write ratio.

8.2.2 *Read to Write Ratio*

We observe that the performance of the system increases with the increase in the read to write ratio, as shown in Figure 8.3. This is expected since the system is tuned to handle more number of reads than writes.

The test was run with a single bench server with 100 concurrent connections to a 3 node Warlock cluster. All the machines were of the *c1.xlarge* type.

8.2.3 *Concurrency*

We can connect to a Warlock cluster in two ways – RPC and Redis protocol as discussed in § 6.1.2 (p. 34). The red line in Figure 8.4 indicates the successful requests using the Redis protocol and the blue line indicates those using RPC.

We observe that the throughput increases up to 64 concurrent connections and stays constant after that. Warlock cluster timed out on some of the requests when using RPC with 1024 connections. Also, we were unable to test the Redis protocol with concurrency level of 1024 due to Basho Bench becoming unresponsive.

The distribution of read and write requests for this test was $^4/_5$ and $^1/_5$ respectively.

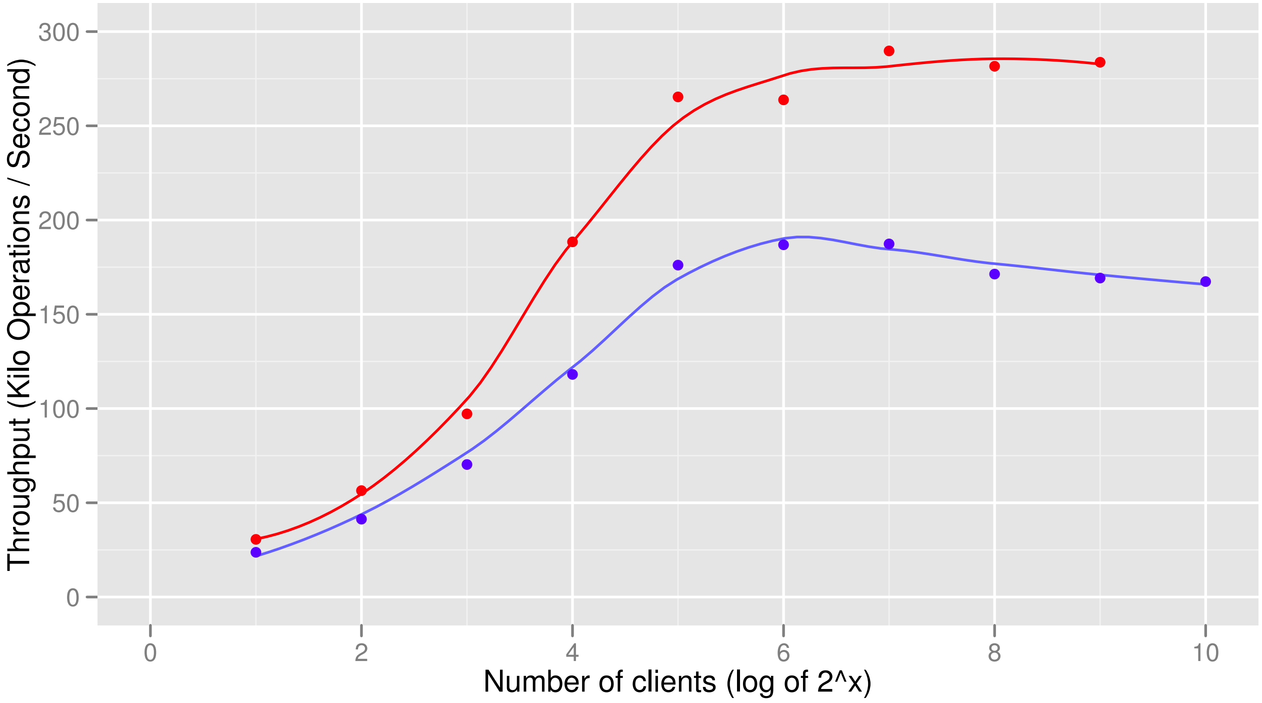

Figure 8.4: Test shows the performance variation of the system with exponential increase in number of client connections.

8.2.4 *Payload Size*

Figure 8.5 shows us the change in system throughput with increase in payload size. We observe a near linear decrease in performance which is expected since the algorithm is $O(n)$, where n is the number of messages.

Figure 8.6 shows the latency result for the same test. We again observe a nearly linear increase in latency.

The amount of read and write requests for this test was $^4/_5$ and $^1/_5$ of the total number of requests respectively.

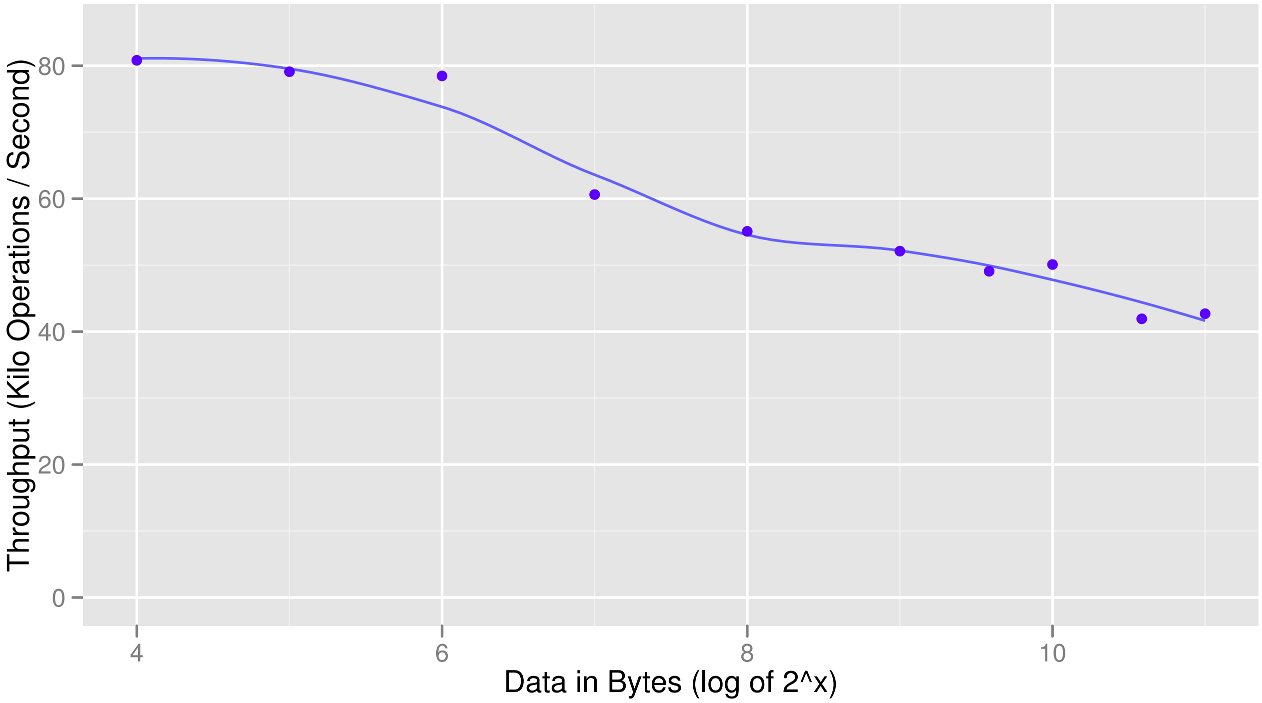

Figure 8.5: Test indicates the variation of system throughput with exponential increase in message size.

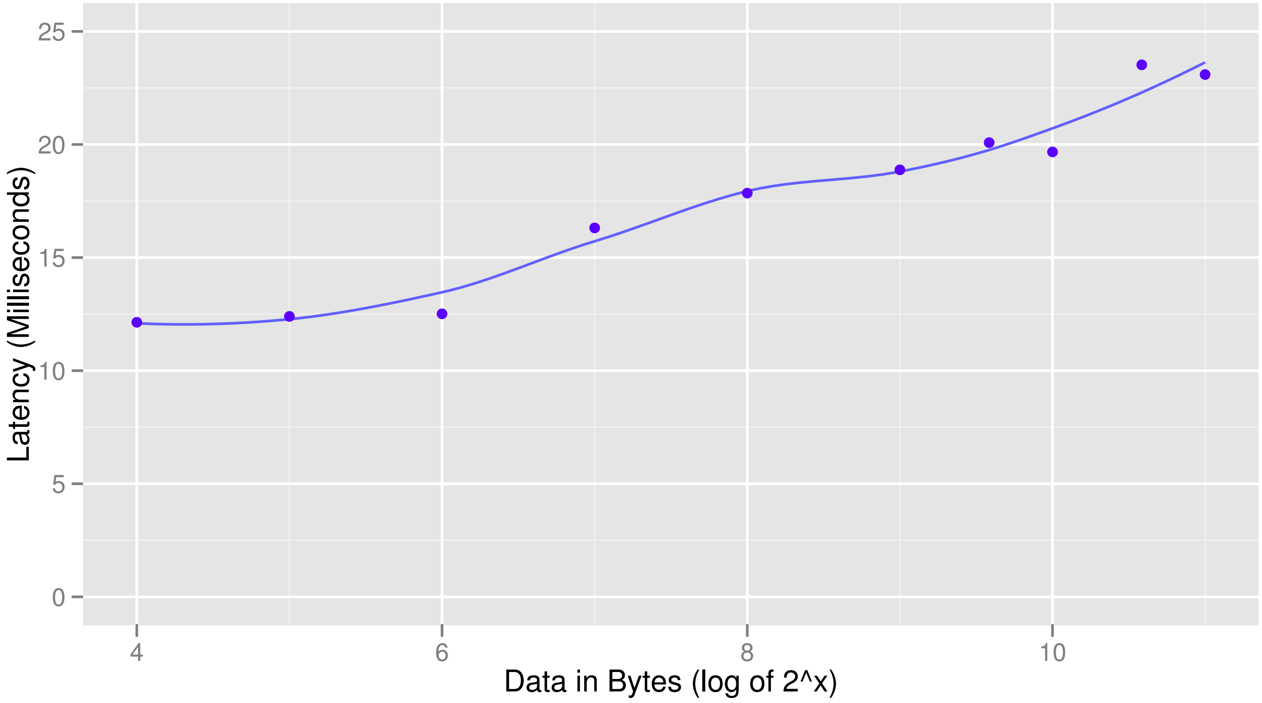

Figure 8.6: Test depicts the change in latency with exponential increase in payload size.

PART III

SUMMARY

9 FUTURE WORK

The Warlock project in its current state is quite stable and ready to be used.[1]. As of writing this report, a test run of Warlock is being done on Magic Land production. Though the project satisfies the requirements (Chapter 4 (p. 21)), Warlock still has scope for improvement which is detailed in this chapter.

9.1 SECURITY

Warlock is meant to be run within an internal network and it is still in terms of security. While reasonably safe to run within a secure network, improved security will allow it to run in more diverse environments.

The client based on Redis protocol supports basic authentication. However, it needs improvements in terms of the password storage, configuration and transmission.

9.2 TESTING

As discussed in § 7.6 (p. 48), testing is essential for distributed projects. A basic set of tests only focus on the common use cases and flows, whereas more tests will help us cover corner cases better.

Warlock can benefit from PropEr[2] by providing automated tests and can also help in improving the overall system stability. By just specifying the structure of the program and the range of input values, PropEr can generate the required number of tests needed for us to gain confidence in our implementation. This tool can then be made part of compile phase itself, allowing us to catch discrepancies at a much earlier stage.

System tests[3] become necessary in distributed systems to make sure changes made to the code does not break any previous features. Common test (Ericsson, 2012a) is a system testing tool that is a part of the Erlang distribution. It is feature rich and allows us to implement system and integration tests. This tool can be used to run system tests during the build phase, thus helping to make sure the build works are expected on multiple nodes.

[1] As per load tests listed in Chapter 8 (p. 53).

[2] *PropEr*(Manolis Papadakis, 2012): A property based testing tool for Erlang.

[3] *System Testing*: Testing the system as a whole, usually treating it as a black box (ignoring the implementation details).

9.3 PERFORMANCE

Performance is crucial for projects serving requests for millions of users. While Warlock is tuned to provide necessary performance for Magic Land, it can be tweaked to push the performance even further.

9.3.1 *Code Profiling*

Profiling, as explained in § 7.9.1 (p. 50), can help us identify the bottlenecks in the codebase. Erlang comes with lots of different profilers which can be used to identify bottlenecks from different perspectives. This should allow us to focus more on the most frequently used functions and try to optimize the most common messages.

9.3.2 *Latency*

While latency is not a major concern for Magic Land[4], improving it is necessary since it blocks the client accessing it. Latency can be improved by reducing the number of processes the proposal/message has to travel through. However, this is quite tricky to optimize without breaking the algorithm.

[4] The front end of Magic Land (written in Adobe Flash) takes several seconds to load. A difference of few milliseconds does not make an overall impact on the user experience.

9.4 SCALABILITY

Scalability for us is to increase the throughput of the system by increasing the number of nodes in the cluster.

9.4.1 *Reads*

Reads are performed locally from the replicas itself and we do not use the consensus protocol for it. This allows us to read directly from any of the replicas.

It should be possible to create additional nodes that just subscribes to all the decisions without taking part in the consensus protocol. We can then use these nodes as read-only sources. This setup should allow us to scale reads to a fairly large extent.

9.4.2 *Writes*

Increasing the number of nodes in the cluster decreases its throughput since more nodes are now involved in approving a proposal. This makes increasing the number of writes a much harder task.

A few of the possible areas for improvements are:

- Changing the algorithm to reduce the total number of messages exchanged.

- Using different ways to broadcast messages. Such as ip-multicast as described in Marandi et al. (2010)

- Batching multiple requests together to decrease individual round trip times.

9.5 RECOVERY TECHNIQUES

Currently if a Warlock node fails, we wipe all its data and restore it from another node in the cluster using replication. While this method has its advantages, there are cases where this might be a problem.

- Certain back ends may have a full blown disk recovery system. Allowing for conflict resolution between nodes can decrease the recovery time for a node drastically, especially for clusters with lot of data.

- Certain back ends may provide the data backup and loading mechanism required by Warlock.

9.6 SMART CLIENT

The current Warlock client is quite simple. It just connects to the server and issues commands to it. Improving the client can allow us to decrease the number of Warlock nodes and improve availability. Some suggestions for such an improved client are:

- The client can store the list of Warlock nodes when it initiates connection. This will allow the client to connect to other Warlock nodes (e.g., using round robin) in case the one it is connected to goes down.

- Clients can have a certain timeout as provided by a Warlock node. This allows us to control client timeouts from the server side.

- The Warlock nodes currently route all the proposals they receive to the Master node. An alternative would be to have the client broadcast the request to all the nodes resulting in reduction of a single message latency when the request is being sent to a node that is not the master.

- The client could queue proposals when the Warlock service is down for reasons such as leader election in progress or temporarily failed quorums.

- A technique used in Burrows (2006), the client could cache the data and the master could actively invalidate cached data on updates. However, this would require a large change in the current

Warlock client server architecture, possibly making the system more complex.

9.7 OTHER

9.7.1 *Consistency Checks*

We treat the database part of Warlock as a state machine and update it based on the proposals that make it through the protocol. However, it might be the case that a few of the nodes become slow in executing these decisions, falling behind the rest of the cluster. If the decision queue grows too large, some of the decisions may even be skipped. While, this has not been observed even under heavy loads, it is a good idea to have a consistency checksum based on the data which is shared periodically with the cluster, providing us with an idea of how many nodes are in sync and how many are catching up.

9.7.2 *Back End Support*

Warlock's current back ends are key value stores. Since Warlock is transparent to the commands, we can potentially use any data store as a back end. Adding more types of back ends to Warlock can help bring it to a wider audience.

9.7.3 *Code Improvement*

Several parts of the code that are open to improvement are tagged with "TODO"s. These are possible performance enhancement points, feature and stability improvement points.

For example, all the ETS tables used currently are directly running under the processes that created them. If these processes crash, the linked ETS table would get lost. Using a table manager would help survive these crashes.

9.7.4 *Expire*

Key expiry has many benefits, as discussed in § 6.1.3 (p. 37). However, testing a few types of implementation led to having severe trade-offs with performance. Hence the need for performant implementation of key expiry.

9.7.5 *More OTP*

Certain portions of the code skip some OTP practices to trade for performance. For example, the commander is spawned directly by the leader instead of using a supervisor. This makes it harder when it comes to hot code loading. In this specific issue, it not much of a problem since commanders are short lived process. A high level tuning from the OTP perspective can help improve code stability and maintainability.

9.7.6 *Timeouts*

Warlock relies heavily on timers which are currently static. Making these timeouts as a function of, say system load, can make the system more tolerant of high loads.

9.7.7 *Reconfiguration*

The selection mechanism of nodes when transferring the master or when allocating the seed node for replication is currently very simple. By using things such as a priority list to keep track of the most stable nodes will allow us to ensure more stable operation.

9.7.8 *Shared Data*

Certain data within the Warlock consensus application is repeated between processes. For example, replicas and leaders share the same proposal map. We currently do not share such data, making it simpler for us to reason about the system and allowing for localizing the failures. It might be possible to improve performance while trading for more complex code.

9.7.9 *Backups*

Warlock is designed to be an in-memory service with each node having a full copy of the dataset. This allows the database module to directly access backup functions of the back end.

10 CONCLUSION

The work that has gone into this project can be classified into four areas.

1. We have surveyed a number of papers and projects related to distributed lock management and compared them in the context of our requirements.

2. We designed a system based on the literature survey and prototyping.

3. We implemented a working system in Erlang.

4. We tested the system to make sure it satisfies the specified requirements and can sustain it.

We were able to select a specific flavour of Paxos and optimize it, use other state machine algorithms, pick ideas from other projects in the field and build a scaffolding around it to create the lock manager.

Separating the implementation from the algorithm details turned out to be essential for ensuring the correctness of the implementation. Erlang provided us with robust and well tested distributed programming primitives. These led to the creation of a modular architecture which enables us to extend it to be used in wider application domains.

The graphs indicating performance over time are all simple and remain flat across signifying steady operation. The results of this study indicate that we were able to achieve our goal of creating a strongly consistent distributed lock manager.

APPENDICES

A SOURCE CODE

This project is open sourced and available on GitHub at https://github.com/wooga/warlock.

Below we see the source tree providing an overview of the application structure.

A.1 ROOT TREE

```
.
|-- apps
|    |-- war_consensus
|    |-- war_db
|    |-- war_server
|    |-- war_util
|-- deps
|    |-- eredis
|    |-- lager
|    |-- ranch
|-- Makefile
|-- priv
|    |-- warlock.term
|-- README.md
|-- rebar
|-- rebar.config
|-- rel
```

Source Code Listing A.1: Root source tree of the project

A.2 APPLICATIONS

```
apps/war_util
|-- include
|    |-- war_common.hrl
|-- Makefile
|-- rebar.config
|-- src
|    |-- war_util.app.src
|    |-- war_util_conf.erl
|    |-- war_util_dict_ht.erl
|    |-- war_util_ets_ht.erl
|    |-- war_util_ht.erl
|-- test
     |-- war_util_dict_ht_tests.erl
     |-- war_util_ets_ht_tests.erl
```

Source Code Listing A.2: Utilities source tree

```
apps/war_consensus/
|-- include
|    |-- war_consensus.hrl
|-- Makefile
|-- rebar.config
|-- src
|    |-- war_consensus_acceptor.erl
|    |-- war_consensus_app.erl
|    |-- war_consensus.app.src
|    |-- war_consensus_client.erl
|    |-- war_consensus_commander.erl
|    |-- war_consensus.erl
|    |-- war_consensus_leader.erl
|    |-- war_consensus_msngr.erl
|    |-- war_consensus_rcfg.erl
|    |-- war_consensus_replica.erl
|    |-- war_consensus_scout.erl
|    |-- war_consensus_state.erl
|    |-- war_consensus_sup.erl
|    |-- war_consensus_util.erl
|-- test
     |-- war_consensus_test.erl
     |-- war_consensus_util_test.erl
```

Source Code Listing A.3: Consensus source tree

```
apps/war_server
|-- include
|   |-- war_server.hrl
|-- Makefile
|-- rebar.config
|-- src
|   |-- war_server_app.erl
|   |-- war_server.app.src
|   |-- war_server_callback.erl
|   |-- war_server_console.erl
|   |-- war_server.erl
|   |-- war_server_receiver.erl
|   |-- war_server_sender.erl
|   |-- war_server_sup.erl
|   |-- war_server_worker.erl
|-- test
    |-- war_server_callback_test.erl
        |-- war_server_test.erl
```

Source Code Listing A.4: Server source tree

```
apps/war_db
|-- include
|   |-- war_db.hrl
|-- Makefile
|-- rebar.config
|-- src
|   |-- war_db_app.erl
|   |-- war_db.app.src
|   |-- war_db_backend.erl
|   |-- war_db.erl
|   |-- war_db_ets_backend.erl
|   |-- war_db_kingdom_backend.erl
|   |-- war_db_redis_backend.erl
|   |-- war_db_sup.erl
|   |-- war_db_worker.erl
|-- test
    |-- war_db_test.erl
```

Source Code Listing A.5: Database source tree

BUILDING AND DEPLOYMENT

B

This appendix discusses the steps necessary to get Warlock running.[1]

B.1 BUILDING

Download, build and start Warlock.

```
# Get the source
$ git clone https://github.com/wooga/warlock

# Build the code
$ cd warlock
$ make
$ make rel

# Create release
$ cd rel/warlock
$ bin/warlock start
```

Source Code Listing B.1: Download and build Warlock.

B.2 DEVELOPMENT CLUSTER

Build a simple 3 node development cluster.

```
# Create dev release
$ make devrel

# Start all the nodes
$ dev/dev1/bin/warlock start
$ dev/dev2/bin/warlock start
$ dev/dev3/bin/warlock start

# Connect any of the two node with the other node, creating the cluster
$ dev/dev2/bin/warlock-admin join warlock1@127.0.0.1
$ dev/dev3/bin/warlock-admin join warlock1@127.0.0.1
```

Source Code Listing B.2: Create Warlock dev cluster.

[1] Warlock was not open source at the time of writing the thesis, though there were plans around it. The interface and commands will most probably remain the same, allowing us to use this section.

B.3 COMMAND EXECUTION

Execute commands on Warlock with ETS as back end.

```
# Connect to dev node 1 and run a set command
$ dev/dev1/bin/warlock attach

(warlock1@127.0.0.1)1> war_server:x(clu, [set, key, value]).
{ok,success}

# Connect to dev node 2 and try to get the value set before
$ dev/dev2/bin/warlock attach

(warlock2@127.0.0.1)1> war_server:x(loc, [get, key]).
{ok,value}
```

Source Code Listing B.3: Executing commands on Warlock.

BIBLIOGRAPHY

Amazon. 2012a. *Amazon DynamoDB*. http://aws.amazon.com/dynamodb/. [Online; accessed 16-August-2012]. Cited on p. 19.

———. 2012b. *Amazon Web Services*. http://aws.amazon.com/. [Online; accessed 16-August-2012]. Cited on p. 36.

Andersson, Arne. 1999. *General Balanced Trees*. In ALGORITHMS: Journal of Algorithms, vol. 30. Cited on p. 50.

Armbrust, Michael; Fox, Armando; Griffith, Rean; Joseph, Anthony D.; Katz, Randy; Konwinski, Andy; Lee, Gunho; Patterson, David; Rabkin, Ariel; Stoica, Ion; and Zaharia, Matei. April 2010. *A view of cloud computing*. In Commun. ACM, vol. 53, no. 4, pp. 50–58. ISSN 0001-0782. doi:10.1145/1721654.1721672. Cited on p. 36.

Armstrong, Joe. 2007. *History of Erlang*. In ACM HOPL III. Cited on p. 29.

ASF, Apache Software Foundation and Yahoo! 2012. *Zookeeper*. http://zookeeper.apache.org/. [Online; accessed 16-August-2012]. Cited on pp. 17 and 18.

Baker, Jason; Bond, Chris; Corbett, James; Furman, J. J.; Khorlin, Andrey; Larson, James; Leon, Jean-Michel; Li, Yawei; Lloyd, Alexander; and Yushprakh, Vadim. 2011. *Megastore: Providing Scalable, Highly Available Storage for Interactive Services*. In CIDR, pp. 223–234. Cited on pp. 16 and 17.

Basho. 2012a. *Basho bench, Erlang based benchmarking tool*. https://github.com/basho/basho_bench/. [Online; accessed 16-August-2012]. Cited on pp. 50 and 53.

———. 2012b. *Lager, Erlang Logging Framework*. https://github.com/basho/lager. [Online; accessed 16-August-2012]. Cited on pp. 49 and 51.

———. 2012c. *Riak*. http://wiki.basho.com/Riak.html. [Online; accessed 16-August-2012]. Cited on p. 19.

Baskins, Doug. 2012. *Judy arrays*. http://judy.sourceforge.net/. [Online; accessed 16-August-2012]. Cited on p. 49.

Beck. 2002. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0321146530. Cited on p. 49.

Birman, Kenneth P. 2005. *Reliable Distributed Systems, Technologies, Web Services and Applications.* Springer. ISBN 10 0-387-21509-3. Cited on p. 31.

Boy, Nathan; Casper, Jared; Pacheco, Carlos; and Williams, Amy. May 2004. *Automated Testing of Distributed Systems.* Final project report for MIT 6.824: Distributed Computer Systems. Cited on p. 7.

Burrows, Michael. 2006. *The Chubby Lock Service for Loosely-Coupled Distributed Systems.* In Bershad, Brian N. and Mogul, Jeffrey C., eds., OSDI, pp. 335–350. USENIX Association. ISBN 1-931971-47-1. Cited on pp. 13, 15, 33, and 63.

Cattell, Rick. 2010. *Scalable SQL and NoSQL data stores.* In SIGMOD Record, vol. 39, no. 4. Cited on p. 19.

Chandra, Tushar Deepak; Griesemer, Robert; and Redstone, Joshua. 2007. *Paxos made live: an engineering perspective.* In Gupta, Indranil and Wattenhofer, Roger, eds., PODC, pp. 398–407. ACM. ISBN 978-1-59593-616-5. Cited on pp. 10, 13, 15, and 16.

Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C.; Wallach, Deborah A.; Burrows, Michael; Chandra, Tushar; Fikes, Andrew; and Gruber, Robert. 2006. *Bigtable: A Distributed Storage System for Structured Data.* In Bershad, Brian N. and Mogul, Jeffrey C., eds., OSDI, pp. 205–218. USENIX Association. ISBN 1-931971-47-1. Cited on p. 15.

Clinger, William D. May 1981. *Foundations of Actor Semantics.* Ph.D. thesis, MIT. Cited on p. 29.

Coulouris, G.F.; Dollimore, J.; and Kindberg, T. 2005. *Distributed Systems: Concepts And Design.* International Computer Science Series. Addison-Wesley. ISBN 9780321263544. Cited on p. 6.

DeCandia, Giuseppe; Hastorun, Deniz; Jampani, Madan; Kakulapati, Gunavardhan; Lakshman, Avinash; Pilchin, Alex; Sivasubramanian, Swaminathan; Vosshall, Peter; and Vogels, Werner. October 2007. *Dynamo: Amazon's highly available key-value store.* In Proceedings of the 21st ACM Symposium on Operating Systems Principles (21st SOSP'07), pp. 205–220. ACM SIGOPS, Stevenson, Washington, USA. Cited on p. 19.

Du, H. and Hilaire, D.J.S. 2009. *Multi-Paxos: An Implementation and Evaluation.* In . Cited on pp. 6 and 11.

Ericsson. 2012a. *Common Test, A framework for automated testing*. http://www.erlang.org/doc/man/common_test.html. [Online; accessed 16-August-2012]. Cited on p. 61.

———. 2012b. *Erlang Behaviour*. http://www.erlang.org/doc/design_principles/des_princ.html#id56732. [Online; accessed 16-August-2012]. Cited on p. 36.

———. 2012c. *Erlang Monitor*. http://www.erlang.org/doc/reference_manual/processes.html#id82613. [Online; accessed 16-August-2012]. Cited on p. 39.

———. 2012d. *Erlang Programming Language*. http://www.erlang.org/. [Online; accessed 16-August-2012]. Cited on p. 29.

———. 2012e. *Profiling Erlang Applications*. http://www.erlang.org/doc/efficiency_guide/profiling.html. [Online; accessed 16-August-2012]. Cited on p. 50.

———. 2012f. *reltool, Erlang release tool*. http://www.erlang.org/doc/man/reltool.html. [Online; accessed 16-August-2012]. Cited on p. 48.

Fischer; Lynch; and Paterson. 1985. *Impossibility of Distributed Consensus with One Faulty Process*. In JACM: Journal of the ACM, vol. 32. Cited on pp. 5 and 10.

Ghemawat, S.; Gobioff, H.; and Leung, S.-T. 2003. *The Google file system*. In SOSP, pp. 29–43. Cited on p. 15.

Gilbert, Seth and Lynch, Nancy A. 2002. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. In SIGACT News, vol. 33, no. 2, pp. 51–59. Cited on p. 5.

GNU. 2012. *Unix make files*. http://www.gnu.org/software/make/manual/make.html. [Online; accessed 16-August-2012]. Cited on p. 48.

Gray, JN. 1978. *Notes on Database Operating Systems*. In Bayer, R, ed., Operating Systems - An Advanced Course, pp. 393–481. Springer-Verlag. ISBN 3-540-09812-7. Cited on p. 17.

Griesemer, Robert; Pike, Rob; and Thompson, Ken. 2012. *Google Go Language*. http://golang.org/. [Online; accessed 16-August-2012]. Cited on p. 18.

Guerraoui, R. and Schiper, A. January 2001. *The Generic Consensus Service*. In IEEE Transactions on Software Engineering, vol. 27, no. 1, pp. 29–41. Cited on p. 5.

Hale, Code. 2012. *You Can't Sacrifice Partition Tolerance*. http://codahale.com/you-cant-sacrifice-partition-tolerance/. [Online; accessed 16-August-2012]. Cited on p. 5.

Hoguin, Loïc. 2012. *Ranch, Socket Acceptor Pool.* https://github.com/extend/ranch/. [Online; accessed 16-August-2012]. Cited on p. 51.

Hunt, Patrick; Konar, Mahadev; Junqueira, Flavio P.; and Reed, Benjamin. 2010. *ZooKeeper: wait-free coordination for internet-scale systems.* In Proceedings of the 2010 USENIX conference on USENIX annual technical conference, USENIXATC'10, pp. 11–11. Cited on p. 17.

Junqueira, Flavio; Mao, Yanhua; and Marzullo, Keith. 2007. *Classic Paxos vs. fast Paxos: caveat emptor.* In Proceedings of the 3rd workshop on on Hot Topics in System Dependability, HotDep'07. USENIX Association, Berkeley, CA, USA. Cited on p. 11.

Junqueira, Flavio Paiva; Reed, Benjamin C.; and Serafini, Marco. 2011. *Zab: High-performance broadcast for primary-backup systems.* In DSN, pp. 245–256. IEEE. ISBN 978-1-4244-9233-6. Cited on p. 17.

Keidar, Idit and Rajsbaum, Sergio. 2003. *On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial.* In de Lemos, Rogério; Weber, Taisy Silva; and Jr, João Batista Camargo, eds., LADC, vol. 2847 of *Lecture Notes in Computer Science*, pp. 366–368. Springer. ISBN 3-540-20224-2. Cited on p. 9.

Kinoshita, Fumihiko. 2008. *Practices of an Agile Team.* pp. 373–377. Cited on p. 43.

Kirsch, Jonathan and Amir, Yair. 2008. *Paxos for System Builders.* Cited on p. 13.

Kota, Uenishi. 2012. *gen_paxos, Paxos in Erlang as FSMs.* https://github.com/gburd/gen_paxos. [Online; accessed 16-August-2012]. Cited on p. 14.

Lamport, L. 1978. *Time, clocks, authornd the ordering of events in a distributed system.* In Communications of the ACM, vol. 21, no. 7, pp. 558–565. Cited on pp. 5 and 6.

Lamport, Leslie. 1998. *The Part-Time Parliament.* In ACM Trans. Comput. Syst, vol. 16, no. 2, pp. 133–169. Cited on p. 9.

———. 2001. *Paxos Made Simple.* In SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory), vol. 32. Cited on pp. 9 and 43.

———. July 2005. *Fast Paxos.* Tech. Rep. MSR-TR-2005-112, Microsoft Research (MSR). URL ftp://ftp.research.microsoft.com/pub/tr/TR-2005-112.pdf. Cited on p. 11.

Lamport, Leslie; Malkhi, Dahlia; and Zhou, Lidong. 2008. *Stoppable Paxos.* In . Cited on pp. 13 and 37.

———. 2010. *Reconfiguring a state machine.* In SIGACT News, vol. 41, no. 1, pp. 63–73. Cited on pp. 13 and 37.

Lamport, Leslie and Massa, Mike. 2004. *Cheap Paxos.* In DSN, pp. 307–314. IEEE Computer Society. ISBN 0-7695-2052-9. Cited on pp. 9 and 12.

Lampson, B. W. 1996. *How to Build a Highly Available System Using Consensus.* In Lecture Notes in Computer Science, vol. 1151. ISSN 0302-9743. Cited on p. 5.

Manolis Papadakis, Kostis Sagonas, Eirini Arvaniti. 2012. *PropEr, Property based testing tool for Erlang.* https://github.com/manopapad/proper. [Online; accessed 16-August-2012]. Cited on p. 61.

Marandi, Parisa Jalili; Primi, Marco; Schiper, Nicolas; and Pedone, Fernando. 2010. *Ring Paxos: A high-throughput atomic broadcast protocol.* In DSN, pp. 527–536. IEEE. ISBN 978-1-4244-7501-8. Cited on pp. 12, 14, and 63.

Mizerany, Blake and Rarick, Keith. 2012. *Doozerd.* https://github.com/ha/doozerd/. [Online; accessed 16-August-2012]. Cited on p. 18.

Nesheim, Knut. 2012. *eRedis, Redis Erlang Client.* https://github.com/wooga/eredis. [Online; accessed 16-August-2012]. Cited on p. 51.

Opscode. 2012. *Chef, open-source systems integration framework.* http://www.opscode.com/chef/. [Online; accessed 16-August-2012]. Cited on p. 36.

gal oz, Arnon Rotem. 2006. *Fallacies of Distributed Computing Explained.* Cited on p. 7.

Primi, Marco and Others. 2012. *LibPaxos.* http://libpaxos.sourceforge.net. [Online; accessed 16-August-2012]. Cited on p. 14.

Reed, Benjamin and Junqueira, Flavio P. 2008. *A simple totally ordered broadcast protocol.* In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08, pp. 2:1–2:6. ACM, New York, NY, USA. ISBN 978-1-60558-296-2. doi:10.1145/1529974.1529978. Cited on p. 17.

van Renesse, Robbert. 2011. *Paxos Made Moderately Complex.* Cited on pp. 11, 27, 29, 37, 38, and 43.

Sanfilippo, Salvatore and Noordhuis, Pieter. 2012a. *Redis.* http://redis.io/. [Online; accessed 16-August-2012]. Cited on pp. 22 and 49.

———. 2012b. *Redis Protocol.* http://redis.io/topics/protocol. [Online; accessed 16-August-2012]. Cited on p. 34.

Schütt, Thorsten; Schintke, Florian; and Alexander Reinefeld, Nico Kruber. 2012. *Scalaris, a distributed key-value store*. http://www.zib.de/de/pvs/projekte/projektdetails/article/scalaris.html. [Online; accessed 16-August-2012]. Cited on p. 20.

Shraer, Alexander; Benjamin Reed, Dahlia Malkhi; and Junqueira, Flavio. 2012. *Dynamic Reconfiguration of Primary/Backup Clusters*. In . Cited on p. 18.

Smith, Dave. 2012. *Rebar, Erlang build tool*. https://github.com/basho/rebar. [Online; accessed 16-August-2012]. Cited on p. 48.

Stoica, Ion; Morris, Robert; Karger, David R.; Kaashoek, M. Frans; and Balakrishnan, Hari. 2001. *Chord: A scalable peer-to-peer lookup service for internet applications*. pp. 149–160. Cited on p. 20.

Stonebraker, M. March 1986. *The Case for Shared-Nothing*. In IEEE Data Eng. Bull., vol. 9, no. 1. Cited on p. 1.

Vieira, Gustavo M. D. and Buzato, Luiz E. 2008. *The Performance of Paxos and Fast Paxos*. Cited on p. 11.

Wiger, Ulf; Svensson, Hans; Fayram, Dave; and Thompson, Andrew. 2012. *gen_leader*. https://github.com/abecciu/gen_leader_revival. [Online; accessed 16-August-2012]. Cited on p. 15.

Wooga. 2012a. *Magic Land*. http://www.wooga.com/games/magic-land/. [Online; accessed 16-August-2012]. Cited on p. 21.

———. 2012b. *Wooga GmbH*. http://www.wooga.com/about/. [Online; accessed 16-August-2012]. Cited on p. 21.