EMPIRICAL STUDY OF UNSTABLE LEADERS IN PAXOS

BY

LONG KAI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Advisor:

Professor Nitin H. Vaidya

ABSTRACT

This thesis studies the effect of unstable leaders in Paxos protocol. Paxos algorithm is one of the most popular solutions for distributed consensus, and is often used for building replicated state machines. Safety is guaranteed by Paxos algorithm regardless of various machine and communication failures. However, the liveness is compromised when multiple Paxos leaders exist at the same time. Also, despite the extensive literature in the field, implementing Paxos algorithm for practical systems is still non-trivial. This thesis first studies the implications of multiple Paxos leaders in practical systems and provides an optimization by using leases. A complete specification of classical Paxos protocol is provided. We evaluate our implementation and show the effect of unstable leaders in practical systems.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Nitin Vaidya for his invaluable advice and guidance in preparing this thesis. I would also like to thank my family and friends for their suggestions and support. This thesis would not have been possible without them.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

It is well-known to use state machine replication [2, 10] to achieve distributed services that require high availability and high efficiency. Consensus algorithm [4] can be used to guarantee that all replicas are mutually consistent [7, 14, 10]. It is possible to apply a consensus algorithm on a sequence of inputs, so that each replica can build the same log of entries. If the entries of the log are deterministic operations on some application state, replicas that maintain the same sequence of entries can eventually reach the same state. For example, the log series could be a sequence of storage system operations. If all replicas apply the identical series of operations on the same initial state, they will eventually store the same information.

Reaching consensus is a fundamental problem in distributed systems. There are three conditions associated with consensus: agreement, validity, and termination. Given multiple replicas with the same initial state, solving a consensus among them implies that each replica will eventually reach the same output, while the three conditions are satisfied. Consensus problem has been studied extensively in the literature.

One of the most well-known approaches is Paxos algorithm [8]. The algorithm only assumes a partially synchronous system, and replica has direct communication channel with each other. Communication channel and machines may fail. Messages can be reordered, dropped, and duplicated. Machines may crash and restart from the stable storage. Safety is guaranteed in Paxos algorithm, while liveness is sacrificed.

Although the original Paxos algorithm was known in the 1980's and published in 1998, it is difficult to understand how the algorithm works from the original specification. Furthermore, the original specification had a theoretical flavor and omitted many important practical details, including how failures are detected and what type of leader election algorithm is used. Filling in these details is critical if one wishes to build a real system that uses Paxos as a replication engine.

As pointed out in [22], there exist significant gaps between theory and practice in Paxos, which need to be addressed by the fault-tolerant computing community. The Paxos algorithm is essentially designed for correctness while liveness and progress are not guaranteed. For example, in the classical Paxos algorithm, multiple leaders may be present in the system and block each other indefinitely. In practice, latency and throughput are critical for any system in production. Even though the performance of normal operations in Paxos is well understood theoretically and some of the benchmark results are available in the literature, the implications of multiple leaders and leadership transfer in real systems are still unclear.

We argue that leader election protocol and leadership management overall have critical impacts on the performance of Paxos in practical systems. The goal of this work is to clearly specify the Paxos algorithm for real implementation and study the liveness implications of Paxos in a practical setting. More specifically, we show how the system performance degrades due to multiple Paxos leaders and frequent leadership changes. We also provide optimizations for leadership management by using leases. We implemented our specification and provide performance results for our implementation.

The rest of this thesis is organized as follows. The next two chapters expand on the background and related work for this project and study Paxos protocol in general. In Chapter 4, we provide an optimization for leadership management and discuss our system design. In Chapter 5, the details of our implementation of the system are specified. In Chapter 6, we show evaluation results of our system. Finally, we conclude in Chapter 7.

CHAPTER 2: BACKGROUND

2.1. Replicated State Machines

Replicated state machines are usually used for services requiring both high availability and efficiency. Availability means that requests of applications are processed and replied quickly. It is straightforward to use replication to increase the availability of the system, because the service can be provided by several replicas instead of just one copy. When some machines are slow or failed, the service remains available.

Coordinating multiple servers is not trivial. The most general solution is to let replicas apply exactly the same operations on the same state, so that the output from each non-faulty replica would be the same. If f replicas hold the same output, then the system overall can tolerate $f - 1$ faults.

In this section we discuss about techniques to coordinate the replicas in a general and highly fault tolerant way. Then we provide background of an optimization called ‘leases’ that can make the coordination very efficient.

Lamport proposed deterministic state machines [3] to arrange each replica to do the same thing. Each replica can be built as a determinist state machine, which means that the transition relation is a function from *(state, input)* to *(new state, output)*. The essence of this approach is to let each replica have the same initial state and receive the same series of operations and apply them deterministically. This becomes the problem of consensus, which informally can be described as reaching the same output by multiple replicas.

When multiple replicas all agree on the values and order of input operations, they will deterministically go through the same process and eventually be in the same state. The failure of a single replica will not make the service unavailable. Note that this approach does not limit the kind of computation that each replica can do. To order the input operations, it is possible to make the ordering part of the input value. When replicas agree on the input value, the ordering of inputs is also contained.

The kind of inputs varies from service to service. For example, a replicated storage service might have $Read(x)$ and $Write(x, v)$ inputs, and a bank account system might have $Credit(x)$ and $Debit(x)$ inputs. Different clients usually generate their requests independently, so it is necessary to agree not only on what the requests are, but also on the order in which to apply them. One easy approach is to give sequence number to each input, so that they can be ordered according to their sequence number. The sequence number can be assigned by a single server to avoid conflicts. For example, the storage service will agree on $Input\ 1 = Write(x, 3)$ and $Input\ 2 = Read(x)$. [25]

2.2. Leases

Coordinating servers to achieve consensus is inefficient for many cases. It is much cheaper for a single replica to access a piece of state without interference from other replicas. Locks are often used for this purpose. However, lock-based schemes are not fault-tolerance. A failed server cannot release the lock it holds. And later no other server can access the piece of state protected by that lock anymore. A "lease" is a lock with timeout. The lock is automatically released when the lease expires. A server can only access that piece of state before the lease expires. If a server holds the lease, then it is the "owner" for that resource when the lease is valid. Other replicas cannot access that piece of state when they do not have the lease.

Leasing mechanism requires the system to have synchronized clocks. The skew of clocks needs to be bounded. For example, if the skew between any pair of clocks is ε. Then if the lease for one server is valid until time t, it is guaranteed that no other server will access the data protected by the lease before $t-\varepsilon$ on the leaser's clock. It is usually hard to provide such synchronized clocks in practical systems; we explore another solution to get around this requirement in chapter 4.

When the owner has a valid lease on a piece of data, it can access the data either by reading or writing. However, it is required for writes to finish within a bounded duration; therefore it is ensured that the operation is finished while the server still holds the lease.

Many transaction systems make extensive use of leases. A certain transaction can only proceed if the server holds a valid lease. If the lease expires before the transaction is finished, that transaction will be aborted, which means that the state is reverted to the exact state before the transaction. There are many subtleties involved when the leases are assumed outside of the transaction. The server may lose the lease anytime, and the application may need to provide mechanism to revert the impact of an aborted transaction. For instance, the application can use redo or undo operations based on logs [11]. It is often required if the resource being leased is itself replicated.

Because a lease is valid only for a period of time, the owner needs to periodically renew the lease. It is also possible for the owner to forsake the lease and let other servers acquire it immediately. Leasing mechanism is powerful, because a failed owner will release its lease automatically when the lease expires. And other servers can compete for the lease again to access that piece of data. Clearly, a tradeoff exists for the duration of the lease. If the duration of the lease is short, then the owner needs to frequently renew the lease. However, a longer lease makes the data inaccessible for a long period of time when the owner fails, because other servers need to wait for the lease to expire first.

Initially, leasing mechanism is proposed for caching shared data. For example, it can be used for memory cache server to process data from shared stable storage. It is possible to assign read lease or write lease to its holders to make the coordination more efficient. [25]

2.3. Consensus Problem

In a consensus problem, each process participated in the consensus can have a different input value, and the consensus is reached when all non-faulty processes agree on a single value that must be one of the inputs. If the agreed value can be anything, then it is easy to construct a trivial case where every server always agrees on “False”. The algorithm is terminated when all non-faulty servers decide on the value.

Standard two-phase commit algorithm selects a fixed leader to coordinate consensus among replicas. Also, the leader decides whether the value is committed or aborted. If some server fails during the process, the leader can abort the update. Generally speaking, in this approach, a single primary is elected to assign sequence number to updates and broadcast updates to other replicas. All initial inputs are sent to the leader, so that the leader has enough information to select a value. However, the algorithm does not work when the leader fails. It also becomes undetermined whether a value is committed.

An alternative approach is to leverage a group of servers to form a quorum. The agreed outcome is the value chosen by a majority of servers, or the quorum. It is ensured that no two values will be chosen because two quorums always have an intersection. There are also problems with this scheme. It is hard to coordinate a majority of servers to choose the same value. Also, the outcome becomes undetermined when a majority of servers fail at the same time.

It is proved [5] that there is no algorithm for consensus in an asynchronous system with faults, achieving both safety and termination. It may take an arbitrary amount of time to deliver any message in an asynchronous system, and it is impossible to separate delay from faults. There are algorithms for consensus in synchronous systems, but they are very inefficient for practical use [13].

2.4. The Paxos Algorithm

It is well-known that Paxos [8] is a powerful tool to achieve consensus. A similar idea was proposed by Liskov and Oki [7]. In this section, we introduce and analyze this algorithm. Paxos is usually perceived as a very complicated algorithm to describe and understand.

Paxos protocol is resilient to various infrastructure failures. The servers can fail and recover from stable storage. The messages can be delayed, dropped, duplicated, and reordered. But safety is still guaranteed. Logically, the algorithm has two kinds of processes, leader and acceptor. There could be multiple leaders in the system coordinating a quorum of acceptors for consensus.

For background, we explain the classical version of Paxos in this chapter. Then, we provide a discussion of the optimizations in literature in Chapter 3. In later chapters, we propose to use leases to make the protocol more available and efficient for practical use.

The algorithm is composed of leader processes and acceptor processes. The behavior of an acceptor is deterministic; an acceptor faithfully follows what the protocol specifies. An acceptor has persistent storage that survives crashes. The set of acceptors is fixed for a single run of the algorithm. The leader processes coordinate acceptors for consensus. Leaders can come and go freely. They are not deterministic, and they may not have persistent storage.

The core of Paxos is based on consensus among a majority of acceptors. As described before, majority-based consensus algorithm stops working when a value cannot be chosen by a majority of acceptors; or if a majority of replicas fail so that it is undetermined what the outcome is. A series of *views* is used to get around this problem in Paxos. There is a single leader for each view *v* and it tries to get a majority to accept a single value *vp*. If the protocol is not successful in one view, it starts another view to propose a value from the beginning. If a value *vp* is chosen by a majority of acceptors in one view, then *vp* will eventually become the decided outcome.

To avoid conflicts, if two values are chosen by a majority of acceptors in two views, they must be the same. The core of Paxos is to solve this problem.

Essentially, there is only one leader for each view, and that leader asks a majority of acceptors to provide information about the previously proposed values. Then, the leader proposes a value for the current view, based on the information from acceptors. If a majority of acceptors agree on the proposed value, the leader finally broadcasts a commit message to all servers.

There are two phases of the protocol and the last half-round to broadcast the results. Thus, without faults, in total $2\frac{1}{2}$ round trips are needed for a successful round. If the leader fails repeatedly, or several leaders are in conflict with each other, it may take many rounds to reach consensus.

Views in Paxos are totally ordered, and each view can only result in two possible states. A view is *dead* if a majority of acceptors vote No (which means vote for no value) for that view, or *successful* if a majority of acceptors have casted votes for a $Value$. The algorithm needs to maintain two properties for safety:

Property 1: At most one value can be proposed in a view.
If a set of acceptors accept some values in the same view, they must be the same value. If a majority of acceptors accept that value, it eventually becomes the output of the consensus, even though it may not be visible at the time.

Property 2: If two views are both successful, they must have the same value.

We introduce the concept of an anchored view as the following:

A view *n* is *anchored* if only if for any view *m* before *n*, either view *m* is dead or the value of view *m* is the same as the value for view *n*. If all preceding views are dead, *n* is anchored no matter what its value is. [25]

The goal of the leader is to anchor the view by finding a proper value to propose. First of all, the leader needs to gather enough information from acceptors to learn about the previous views. In Paxos, the leader broadcasts a prepare message with a new view number *v* to all acceptors. Upon receiving this message, an acceptor responds with all the votes that it has casted for views before *v*. If there are views before *v* in which the acceptor has not casted votes, the acceptor sets those votes to *No*, indicating that it will not accept any value for those views. When the leader receives responses from a majority of acceptors, it can find a proper value to propose in its view.

Because the leader receives promises from a majority of acceptors, it can decide that a previous view is dead if a majority of acceptors voted *No* for that view. The leader ignores all the dead view. When it encounters a view *v'* with a value, it chooses that value *vp* to propose in view *v*. The logic is that *v'* is anchored by assumption and since no view between *v'* and *v* is successful, view v is also anchored if the value of view *v* is the same as the value of view *v'*.

Only in one condition that the leader is freely to propose any value. It is when the leader discovers that no earlier view is successful. In this case, this is the first view to have a value status, thus it is also anchored no matter what the value is.

After the first round, the leader proposes the value *vp* to all acceptors in the system. Upon receiving the propose message, an acceptor casts its vote for the value if it has not set the vote for view *v* to *No*, which implies that the acceptor has not promised any leader with a higher view number. If an acceptor has set the vote for view *v* to *No*, it ignores the accept message or responds with a rejection. When the leader receives positive acceptance from a majority acceptors, it can decide that a consensus has been reached. Finally, it notifies all the servers by broadcasting a commit message. Thus, in total, the algorithm requires 2 and half rounds to finish one instance.

It is important to realize that a consensus is reached when a majority of acceptors cast votes in the same view. Even though at the time, the leader may not learn the fact and it may fail before receiving acceptance from a majority of acceptors. The view becomes successful when a majority of acceptors accept the value in that view, and no future leader will propose a different value. [25]

To illustrate this algorithm, we discuss two concrete cases in Table 2.1. In here, we have three acceptors and three previous views. In the first instance of the algorithm, when a leader establishes the fourth view, all previous views are dead because a majority of acceptors voted *No*. If the new leader receives information from all acceptors, the leader can learn that all views are dead and it is ok to choose any value to propose in the fourth view. However, if only acceptor *a* and *c* successfully delivered their promises to the leader, the leader must choose 23 to propose because it does not know whether view 3 is dead.

In the second instance illustrated in Table 2.1, only view 2 is successful. Even though both view 1 and view 3 are dead, the new leader in view 4 will always learn about the value in view 2, which is 23. The leader will not consider view 2 dead and will propose the same value as in view 2. As this case illustrated, a successful view will mandate all future views to propose the same value. And thus when a majority of acceptors accept a value in the view, a consensus is reached.

The Paxos algorithm is robust, because the only requirement for the underlying communication channel is that the contents of messages are intact. The communication network can drop, delay, reorder, and duplicate messages, but the algorithm is still correct. Servers can fail and recover from stable storage to restore previous state.

2.5. Tables

Table 2.1 Examples for Paxos

	Acceptor a	Acceptor b	Acceptor c
View 1	53		
View 2			53
View 3			23

	Acceptor a	Acceptor b	Acceptor c
View 1	53		
View 2		23	23
View 3		23	

CHAPTER 3: RELATED WORK

3.1. Alternative State Machine Replication Protocols

As described in Chapter 2, state machine replication [2, 10] is used to establish several replicas that maintain the same data structures and are updated in the same way. This is done by applying the same deterministic operations in the same order on all replicas. It is also assumed that these replicas have the same initial state, thus they will end up having the same outcome. Replicated state machine is often used for distributed storage system or distributed database system. The abstraction is powerful, which allows the application to perceive the several replicas as one copy of the data. The system may allow applications to generate concurrent operations on multiple replicas as the same time, but the operations are serialized, which means that the parallel execution of multiple operations has the same effect of the serial execution on a single copy.

Several protocols have been proposed for achieving this goal. One of the most popular algorithms is the two-phase commit protocol [1]. As briefly discussed in Chapter 2, the two-phase commit protocol (2PC) relies on the primary leader to assign sequence numbers to updates. The protocol stops working if the leader fails. When the leader fails, the transaction cannot be simply aborted, because the outcome is undetermined. This can become a serious problem, because all the replicas need to block until the leader is restarted and restored.

The Three-Phase Commit (3PC) protocol [24] adds another around to the classical 2PC protocol to get round the problem of leader failures. In 3PC, when the leader fails, a majority of replicas can still make progress. However, if more than one server fails, 3PC may still need to block a majority of replicas to wait for the servers to restart and restore. There is another version of the protocol called the enhanced three-phase commit (e3PC), which allows any set of servers that constitutes a majority to make progress, even though other servers may fail. However, the protocol is even more expensive than 3PC. [26]

3.2. Alternative Paxos Specifications

Several papers have attempted to specify and clarify the Paxos algorithm since its original presentation by Lamport.

3.2.1. Paxos revisited

De Prisco, Lampson, and Lynch [15] specify and prove the correctness of the protocol using Clocked General Timed automata to model the processes and channels in the system. The Clock GTA is an extension to General Timed Automaton with the notion of a Clock that models the local time at each process. Clock GTA provides a systematic way to describe partially asynchronous systems. These types of systems normally exhibit synchronous behavior but occasionally will violate those timings bounds.

The authors introduce automaton for failure detection, leader election and then combine those into a system which performs Paxos. The BASICPAXOS automaton is further split into three components: BPLEADER, BPAGENT, and BPSUCCESS.

The algorithm is then explained using info-quorums, accepting-quorums and the invariants that the algorithm keeps. They emphasize that the key points in the algorithm are that 1) the info-quorums and the accepting-quorums always have one process in common and 2) the leader who selects the value to propose maintains the consistency of the round.

Then they assume that when the system stabilizes that actions by processors are bounded by l time and messages are bounded by d time. With these assumptions, they prove that Paxos in the worst case terminates in $24l + 10nl + 13d$ time and $10n$ messages when there are n processors.

3.2.2. Deconstructing Paxos

[18] decomposes Paxos into three abstractions: round-based register, round-based consensus, and weak leader election. [18] also introduces four variants of Paxos, one of these is FastPaxos and will be discussed in Section 3.3.2.

The round-based register supports $read()$ and $write()$ operations with an integer round number and may commit or abort those operations. Round-based consensus is a shared object which supports the $propose()$ operation. $propose()$ can commit a round or abort. The weak leader election abstraction is a shared object that elects a leader among a set of processes and supports the $leader()$ operation which returns the process identifier for the current leader.

[18] then deconstructs the Paxos algorithm using the three abstractions, along with totally ordered broadcast and deliver primitives. Round-based consensus captures the sub-protocol used in Paxos to agree on a total order and uses the round-based register. Weak leader election encapsulates the way Paxos chooses a process that decides on the ordering of the messages.

3.2.3. Paxos Register

[23] introduces the Paxos Register to explain Paxos and other similar protocols, such as Byzantine Paxos. The Paxos Register is a write-one register that exposes two abstractions: 1) write and read operations which correspond to proposing and deciding and 2) tokens that guarantee agreement despite partial failures.

Processes play any of three roles: proposer, acceptor, or learner. Proposers propose values by writing to the register while acceptors implement the register abstraction. Learners decide values by observing what have been written to the register.

3.3. Theoretical Optimizations for Paxos

3.3.1. Cheap Paxos

[19] introduces an optimization to Paxos that reduces the number of active processors needed to achieve state machine replication in a system. The main idea is that f of the $2f + 1$ processors which are usually a part of Paxos can be used as auxiliary processors. They can remain idle until one of the main processors fails. Once a main processor fails, the auxiliary processors take part in reconfiguring the system to replace the failed processor. The recovery step involves the auxiliary processors as acceptors to allow the system to continue progress. However, the algorithm can only guarantee liveness when the set of main processors does not change quickly. This is a weaker liveness condition than the Classic Paxos algorithm.

The intuition of being able to not use f of the processors is that in the Classic Paxos algorithm if we have all of the working processors form a quorum, the processors not in the quorum do not have to do anything in order for there to be progress.

3.3.2. FastPaxos

The FastPaxos algorithm of [18] reduces the communication during stable periods of the system. The protocol is optimized to provide one round-trip communication to agree on a given order for a request.

The round-based consensus consists of a read and write phase. When the system is in a stable state, the read phase can be skipped.

As shown in [17] the optimal message delay that is needed to solve asynchronous consensus is two rounds. However, [21] points out that typically the processors in Paxos that propose values are not the same that choose them. One example is that typical Basic Paxos deployments usually have remote clients contacting a group of Paxos servers. This extra communication from the client to the server introduces an extra message delay.

[21] is an algorithm which removes the extra message delay in most instances. [17] showed that it is not possible for a general consensus algorithm to learn a value in two message delays in all instances. When there are concurrent competing proposals, extra communication must be had in order to maintain safety.

In the paper, there are also two recovery protocols describes, coordinated and uncoordinated recovery. Coordinated recovery involves the acceptors sending their phase $2b$ messages to the coordinator and having the coordinator detect that there might have been a collision. The coordinator will treat those messages as the next rounds phase $1b$ messages instead. Uncoordinated recovery involves the acceptors broadcasting their phase $2b$ messages to the rest of the acceptors and then appropriately picking a value.

The recovery cost can potentially add four extra message delays. Additionally the fast rounds require that the acceptors quorum size be larger than the simple majority of Classic Paxos.

3.4. Practical Study of Paxos

The most closely related work is [22], which also takes a practical approach. Paxos algorithm is used as the base for a framework that implements a fault-tolerant database. This building block is used as the heart of Chubby [20]. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- “While Paxos can be described with a page of pseudo-code, the complete implementation contains several thousand lines of C++ code. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.
- “The fault-tolerant algorithms community is accustomed to proving short algorithms (one page of pseudo code) correct. This approach does not scale to a system with thousands of lines of code. To gain confidence in the “correctness” of a real system, different methods had to be used.
- “Fault-tolerant algorithms tolerate a limited set of carefully selected faults. However, the real world exposes software to a wide variety of failure

modes, including errors in the algorithm, bugs in its implementation, and operator error. Designers had to engineer the software and design operational procedures to robustly handle this wider set of failure modes.

- “A real system is rarely specified precisely. Even worse, the specification may change during the implementation phase. Consequently, an implementation should be malleable. Finally, a system might “fail” due to a misunderstanding that occurred during its specification phase.” [22]

Essentially, their work demonstrates the significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol. Another interesting factor is that they provided insights about testing fault-tolerant systems.

The core algorithms work remains relatively theoretical and many practical tools and shortcomings still need to be addressed. [22]

CHAPTER 4: SYSTEM DESIGN FOR IMPROVEMENT

4.1. System Overview

4.1.1. System Model

For the system model, our assumption is that the system contains multiple replication servers and they communicate with each other over unreliable communication network. The only assumption about the network is that the content of the message is not corrupted. Messages can be delayed, reordered, dropped, or duplicated. Network partitioning may happen, which means a set of servers can be cut off from the rest of the system.

Servers in the system may stop completely and recover from stable storage. We assume there is no Byzantine failure and servers faithfully perform operations according to the protocol. Servers can be slow and crash for arbitrary times, but a stable storage must be available to maintain some server state.

Operations in servers are all deterministic. If two servers apply the same operation on the same state, they should end up in the same state as well. As described in previous chapters, we use Paxos to order application updates and deliver the same set of updates to replication servers. Servers in the system should have the same initial state. And when they apply the same set of operations ordered by Paxos, they should eventually be in the same state.

An application proxy is implemented locally in each Paxos server. Application is responsible for implementing the communication channel of its choosing between remote application clients and local application servers. Clients introduce updates for execution by initiating them on Paxos servers. Updates should have different and unique identifiers. Update id should contain two values, the id of the client that issues the update and a monotonically increasing sequence number that is maintained by the client. A single client cannot issue multiple concurrent updates. A client can only issue a new request after all its previous updates are acknowledged.

4.1.2. Architecture and API

We implement our service as a replicated log service and use Paxos algorithm as the heart of the protocol to add and order entries in the log. The system consists of multiple replication servers and each replica contains a copy of the entire log. Applications issue entries to be added to the log. Replicas communicate with each other through Paxos protocol, so that each replica can record the same sequence of entries.

We present the API of our service in Figure 4.1. Application submits requests by invoking a local function in replicas and the service will automatically add that entry to all replicas. When the entry is added, replicas invoke a local callback function to notify the application.

Multiple requests from different clients and applications can be submitted at the same time. The system can process multiple concurrent requests.

4.2. Problems with Classical Paxos Protocol

The core Paxos protocol does not specify the algorithm for leader election. As analyzed in previous chapters, the core Paxos protocol is designed to guarantee safety. In fact, the existence of multiple leaders does not compromise the correctness of Paxos. However, the performance of the replication system can still benefit significantly from having a single leader in the system. In the following sections, we analyze three motivations for having a single leader in the system and propose a solution for improvement.

4.2.1. Multiple leaders

As discussed in the Paxos paper, “it is easy to construct a scenario in which two leaders each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Leader p completes phase 1 for a proposal number $n1$. Another leader q then completes phase 1 for a proposal number $n2 > n1$. Leader p’s phase 2 $accept$ requests for a proposal numbered $n1$ are ignored because the acceptors have all promised not to accept any new proposal numbered less than $n2$. So, leader p then begins and completes phase 1 for a new proposal number $n3 > n2$, causing the second phase 2 $accept$ requests of leader q to be ignored. And so on.” [16]

The only way to ensure liveness in Paxos is to select a single leader in the system and only allow that server to propose values. If the system contains only one leader and a majority of replicas are alive, the system is guaranteed to make progress. When the leader is elected, it can learn the highest committed sequence number from replicas, and propose values with higher sequence numbers. Without conflicts from other leaders, replicas will accept the values from the single leader.

From the discussion above, the system needs a leader election protocol to make sure there is at most one leader to guarantee liveness of Paxos. However, if the election protocol fails to guarantee the single leader, the safety property of Paxos is not compromised.

4.2.2. Multi-Paxos

A complete instance of Paxos requires five writes to the stable storage, for each of the prepare, promise, propose, accept, and commit messages. The server needs to finish the write to the stable storage before proceeding to the next step. Without any optimization, the Paxos protocol requires five consecutive writes to disk for each update. This can easily become the bottleneck of the system and be the major contributor to the overall latency.

A popular optimization for the basic Paxos protocol is called multi-paxos [16]. Multi-paxos lets the leader to propose multiple values concurrently, which means the first phase of the protocol (prepare and promise) is run only once for multiple Paxos instances. As long as the leader does not change, a new value only requires the second phase (propose and accept) to be delivered to replicas. The safety property of the basic Paxos algorithm is not compromised, because a view with higher sequence number can still be established by another server. With this optimization, the disk write of prepare and promise messages are ignored for most of the updates. And the remaining disk write of propose and accept can be done in parallel because they are performed in different servers (leader and acceptor). However, to benefit from this optimization, a single leader should be elected and maintained for long periods of time.

In the classical Paxos algorithm, when the leadership is transferred to another process, the new leader only needs to rerun the last instance of Paxos for completion and notification to other replicas. With the optimization of Multi-Paxos, however, a new leader needs to detect all unfinished Paxos instances of the previous leader and rerun them all.

4.2.3. Expensive Read

The core Paxos algorithm does not differentiate reading operations and writing operations, and reads need to be globally ordered in the same manner as writes. Otherwise, the consistency guarantee does not hold. In a simplified example, if replicas are allowed to serve reads locally without coordination with other replicas, a client can read from a more up-to-date replica first and then read from a stale replica. Clearly, the sequential ordering of the replicated state is violated.

The classical Paxos protocol does not consider the reading operations, which often dominate in a typical storage system. If the classical Paxos algorithm is used directly to implement a replicated storage system, reads need be serialized to ensure the no stale state is returned to the application. Especially, the leader cannot serve the reads locally because it may not contain the most up-to-date information. In the classical Paxos algorithm, the leader is not aware whether it still holds the leadership. When a new leader is selected in the classical Paxos, there is no way for acceptors to notify the previous leader. Thus, if the leader can serve the reads locally, it is possible that an old leader may return stale state to the application. Reads in classical Paxos protocol are very expensive.

4.3. System Optimizations

The solution is to guarantee that there is at most one leader in the system at any time, and the leader always holds the most up-to-date information locally. In this way, the progress of Paxos is not suspended by the conflicts of multiple leaders. Even though the progress still cannot be guaranteed in practice because a majority of replicas can be failing at the same time, or the network can be partitioned, but these infrastructure failures are much less likely. Another benefit from having a single leader is that the leader can store the most up-to-date state and serve reads locally. Because all updates are sequenced by the single leader, conceptually it can insert reads into the global order and guarantee consistency without extra coordination with other replicas.

To guarantee there is at most one leader in the system at any time. The leader election protocol needs to be strong, which means that at most one leader can be elected in each instance of the election protocol. However, leader election alone is not sufficient to provide the guarantee of a single leader, because it does not concern the condition of leadership transition. For example, a new instance of leader election can be initiated while the old leader is still valid. Because fault-tolerant election protocols do not require the participants of all replicas, it is possible to elect a new leader regardless of the existence of a valid old leader.

To ensure that no concurrent leaders are valid when a leader is elected, a leasing mechanism is used in our system. When the current leader is valid, a majority of replicas promise that they will not initiate or participate in any leader elections. And thus, no leader can be successfully elected while the current leader is still valid. In short, the election protocol selects at most one leader in each election round, and the leasing mechanism guarantees that no election will be successful when the old leader is still valid.

4.3.1. Leader election

As described above, Paxos is coordinated by a leader server, which assigns sequence numbers to client updates and proposes the assignments for global ordering. The servers use a leader election protocol to elect this leader. To simplify the system and reuse the development structures, we see the leader election as a consensus problem, which is solved by Paxos itself.

The inputs of the election are server id of each participant. Eventually, replicas reach consensus on a single input and a new view is subsequently installed in the system. It is straightforward to use a variant of the basic Paxos protocol to achieve the required election protocol. From a high-level perspective, each participant proposes a new view in Paxos. But multiple participants can propose at the same time and block each other with repeated attempts. To avoid conflicts in a best-effort manner, each participant timeouts before making another proposal with higher view number when its previous proposal is not successful. An unsuccessful participant will wait long enough before retrying to reduce the likelihood of being conflict with the current leader, if exists. Once a value is successfully committed in this election protocol, no other participants can establish new views, except the elected leader.

The Paxos-based leader election faces the same liveness challenge as we discussed before. We briefly discuss an optimization in this section. But for simplicity of reasoning, this optimization is not realized in our implementation. A more sophisticated solution is to use a weak election protocol to elect a coordinator for leader election. A weak election protocol does not guarantee single leader for each election, but the protocol is guaranteed to proceed with reasonable performance. Multiple coordinators may result from the weak election protocol, but a weaker election protocol has more flexibility to be efficient and avoid conflicts. Only the coordinator(s) can propose new leadership contestants to replicas. Because the number of coordinators is small, and most of the time one, the blocking conflicts for the leader election is greatly reduced.

Because of the leasing mechanism which is discussed in the next section, the leader election can only be initiated when a majority of Paxos servers consider the current leader to be failed and therefore invalid. Intuitively, when the leader election is initiated, it is impossible for the old leader to be valid at the same time, because a majority of replicas deny its leadership. Thus, the new leader elected from the election will not have conflict with the old leader. And at most one leader exists in the system at any time. The leasing mechanism is discussed in the next section.

4.3.2. Leader Lease

Leases are a well-known solution usually used for ensuring at most one server at a time can access a piece of data [20]. Originally, lease is a contract that gives the holder certain rights over some property for a bounded amount of time. In our context, a lease grants its holder leadership control within the term of the lease. A server must first acquire the leader lease before proposing values to other replicas in the system.

Before processing reading requests locally and responding to writing requests, it is required for the leader to obtain a valid leader lease from replicas. When a server is elected as leader during the election, other replicas also grant a lease ensuring that they will not elect a new leader during the lease term unless the leaseholder forsake its lease. After the lease expires, an operation on the shared state requires that the server first extends the lease for leadership, refreshing the local copy if the leadership has been transferred since the lease expired. When a client sends a request about the shared state, the server must defer the request until the leadership lease is obtained and forward the requests to the server that holds the lease.

The semantics for our implementation of the leader leases [9] are the following: while the server has a valid leader lease, it is impossible for other servers to successfully propose a value. When a server becomes the leader, it catches up with the leading replicas. Thus, it is guaranteed that while a server has a valid leader lease, it also has the most up-to-date information in its local copy of the shared state. This copy can be used to serve reads locally without interacting with multiple replicas in the system. We let the leader extend its leader lease periodically before the lease expires, so that the leadership does not need to change for long periods of time.

Acceptors in our system only process the messages from the server that they believe holds a valid leader lease. Any message sent from a server without a valid lease will be ignored or rejected. To ensure there is no problem resulted from clock drift, the lease term in the leader is shorter than that granted by the acceptors. Thus, the lease always expires sooner in the leaseholder. The leader periodically extends its lease by submitting a renewal request to Paxos.

4.3.3. Leader Library

The leader initiates a Paxos round trying to extend leader leases every 10 seconds from Paxos replicas. In our system, we set the leader lease to be 30 seconds. Thus, a majority of acceptors have to miss two consecutive renewal requests before the lease is spuriously expired. We assume that if the leader is able to send out renewal request, then it is alive to process application requests. In our implementation, we prevent a restarted leader from continuously holding the lease it previously held. If the lease is still valid, the leader must invalidate the lease and compete for the leadership again. The reason is that during the crash and recovery, the memory state in the leader is lost. To ensure that a restarted leader does not mistake lease extension by acceptors from lease grant, the grant message is different from extension message. When a restarted leader receives lease extension from other servers, it rejects the extension and notify other servers that the leadership is free to compete.

4.3.4. Dealing with Clocks

Implementing leases usually requires loosely synchronized clocks in the system. However, even this is hard to achieve in practical systems. To get around this problem, we use a technique demonstrated in Figure 4.2, and proposed by Liskov [12]. The assumption for this mechanism is clock rate synchronization, which means that if the clock in one server advances by T, there is no other server whose clock advances by more than $T(1+\varepsilon)$. The exact clock drift can be undetermined and unbounded. As the Figure 4.2 demonstrated, the leader holds the lease for less amount of time than that granted by other replicas.

The leader acquires the leader lease when it receives acknowledgements from other servers. However, the lease period on the leader server starts before it sends out the request for lease. Thus, the lease period in the leader starts earlier with shorter length, compared to other replicas. As long as the rate synchronization assumption is satisfied, the lease always expires sooner in leader than in other replicas.

4.3.5. Reconciliation

Reconciliation is important for replicas to actually execute the updates, instead of just agreeing on the order. The classical Paxos algorithm only solves the problem of reaching consensus on the set of updates and their order. However, during the protocol, updates are not guaranteed to reach all replicas. A replica cannot execute any update if an early update is not delivered. To solve this problem, we need to add a reconciliation mechanism to get the lagging replicas up-to-date.

For simplicity of design and reasoning, in our system, the reconciliation happens only between the leader and replicas. The complete set of decided sequence numbers are piggybacked on the messages between the leader and replicas during the lease request and renewal. When the leader lease is granted by a majority of replicas, the leader also receives a complete set of decided sequence numbers from the same majority of replicas. The leader starts reconciliation with corresponding replicas if the leader is lagging behind. When the leader holds the most up-to-date information, it can begin to serve reads locally. On the other hand, during lease renewal, the replicas learn the complete set of decided sequence numbers from the leader periodically. If the replica is lagging behind, it initiates reconciliation with the leader to catch up.

It is possible to extend the protocol to allow reconciliation happen between peer and peer. This may be useful because the leader can easily become the bottleneck of the entire system. By issuing reconciliation with peers, the load on the leader is reduced. For example, the peers can randomly gossip with each other to learn about the missing updates and request committed updates from each other. However, this is beyond our implementation.

4.4. Figures

Figure 4.1 System Architecture

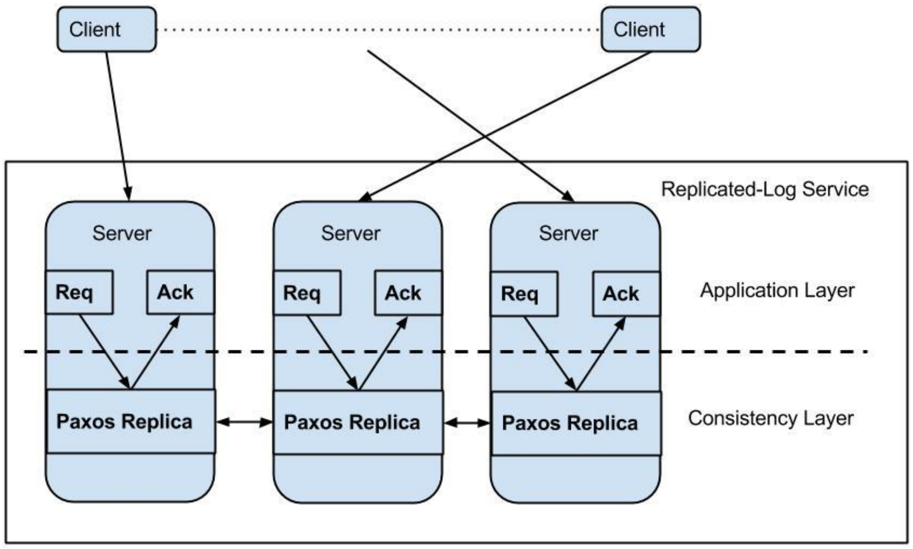
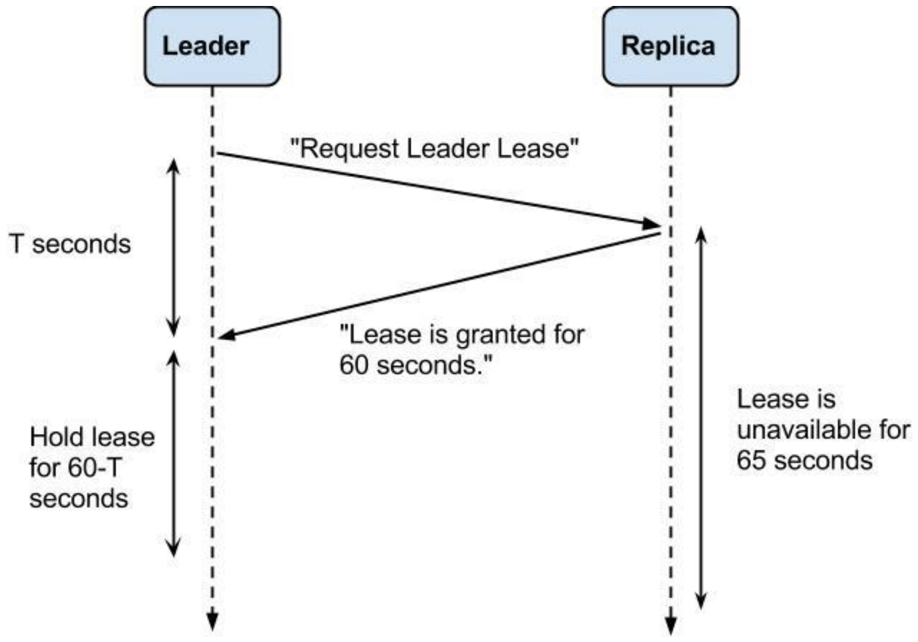


Figure 4.2 the Lease Protocol between the Leader and Replicas


Leader
Replica
"Request Leader Lease"
T seconds
"Lease is granted for 60 seconds."
Lease is unavailable for 65 seconds
Hold lease for 60-T seconds

CHAPTER 5: SPECIFICATION FOR CLASSICAL PAXOS

5.1. Overview

For evaluation, we implemented two versions of the fault-tolerance log system. The simple version is faithful to the classical Paxos algorithm [8] with minimum modification for development. We also implemented a complete version with leases and reconciliation as described in Chapter 4. However, the specification for the complete version is rather complicated and we understand that many alternative implementations exist to develop leases and reconciliation. Therefore, in this Chapter we only present the specification for classical Paxos algorithm. We use pseudo code to show the important variables and basic structures of the actual code. All variables and procedures in section 5.5 are taken from the code, but greatly simplified.

As described in previous chapters, a replication system built from Paxos lets replicas agree on the same set of updates and their ordering. By starting with the same initial state, replicas execute the same sequence of operations and eventually reach the same state. The fault-tolerance log service runs multiple Paxos instances in parallel, and each Paxos update is assigned a unique sequence number. When a replica receives consecutive updates, it executes them in the order of their sequence numbers.

Leaders are selected by successfully establishing views among acceptors. Each view has a unique view sequence number and there is only one leader associated with each view. A view is established in the first phase of the Paxos protocol, by exchanging prepare and promise messages between the leader and acceptors. A leader considers itself elected when it receives $\lfloor N/2 \rfloor$ promise messages. Subsequently, it orders application updates and begins to propose values by broadcasting propose messages. A consensus is reached when the leader receives $\lfloor N/2 \rfloor$ accept messages.

To make the protocol more efficient, a leader concurrently proposes multiple values to acceptors. The first phase of the Paxos protocol is performed only when the leadership needs to be changed. While the leader is not changed, each new update directly invokes the second phase of the protocol.

Actions in servers are triggered by events. Essentially, an action can be triggered either by a received message or a timer running out. The leader receives requests from applications and messages from acceptors. It retries after timeout when certain actions are not successful. Acceptors only receive messages from the leader. They try to compete for the leadership when the current leader is not responsive for long enough.

The following sections are organized in this way. We first discuss the data structures and message types used in our implementation. The Paxos protocol is divided into prepare phase and propose phase, and we present them separately.

5.2. Data Structures and Message Types

Data structures reflect the state of each server in the system. The state of a server is generally divided into two categories, memory state and disk state. The data structures presented in this section are all stored in memory. In Figure 5.1, we lay out the essential variables associated with prepare phase, and propose phase. The updates are ordered, delivered, and stored in propose phase. Especially, the variable History contains all accepted and committed updates received by this replica. History is indexed by the update sequence number, and it is also maintained in the disk. Lastly, we also show the variables we use for client handling. This is useful for update execution.

The leader and acceptors communicate by exchanging messages. The message types are showed in Figure 5.2. Clients invoke the protocol by sending Update messages to the leader. In the prepare phase of the protocol, the leader sends Prepare message to acceptors, who reply with Promise message. In the propose phase of the protocol, the leader broadcasts Propose message to acceptors, while acceptors send back Accept messages. In the end, the leader sends out Commit message to notify that an update is committed.

Upon receiving a message, a server first checks whether the message should be processed, as specified in Figure 5.3. Because the underlying communication infrastructure is unreliable, the messages could be reordered and delayed. It is possible for servers to receive stale or duplicated messages. Thus all incoming messages need to be checked before being processed. Messages failed the check are ignored by the receiver. If the check is passed, the server proceed to apply the message to its local state according to the rules described in Figure 5.4.

5.3. Prepare Phase

The Prepare phase is the first round of the Paxos protocol. In classical Paxos, this round is used to establish a new view associated with one leader. The pseudo code for this phase is showed in Figure 5.5. During this phase, the leader establishes a new view identified by a view sequence number. To guarantee that this view has no conflict with previous views, the leader needs to have enough information from the acceptors to learn about the previous views. This information is provided when the acceptor replies the leader with a promise message. In our implementation, a leader can propose multiple values in the same view.

In the protocol, a new view sequence number needs to be selected by the leader. In our implementation, this view sequence number contains two values, the identifier of the leader and a monotonically increasing local sequence number. The local sequence number is the same as the system timestamp when the view is generated. Thus this number must be monotonically increasing. The view sequence numbers are ordered first by their local sequence number. When their local sequence numbers are the same, they are ordered by their server identifiers. Besides the view sequence number, the prepare message also contains a committed sequence number for updates. This update sequence number indicates the highest number among all consecutively committed updates received by the leader. After broadcasting the Prepare message to acceptors, the leader waits for their responses.

When an acceptor receives a prepare message from the leader, it checks whether the contained view sequence number is the highest it has encountered so far. A prepare message without the highest sequence number is rejected. If the sequence number is indeed the highest, the acceptor modifies the local state and constructs a Promise message. A Promise message contains all the information that this acceptor can provide to help the leader learns about previous views. The acceptor provides information for updates that the leader are not aware of. If the acceptor received Propose message or Commit message beyond the committed sequence number provided by the leader, these messages are included in the Promise message replied to the new leader.

When at least $\lfloor N/2 \rfloor$ Promise messages are delivered to the leader for the view it prepares, the leader successfully establishes the view. From the information provided in those Promise messages, the leader is able to find the highest used update sequence number and catch up with the leading replica. The leader shifts to the Propose phase subsequently.

5.4. Propose Phase

The leader only responds to client requests when it is shifted to the propose phase. The updates received from clients are added to *pendingUpdates* queue. The client requests are ignored in non-leader servers. In this phase, the leader coordinates acceptors to reach consensus on the values added to the log and their order. The pseudo code for this phase is showed in Figure 5.6.

When the *pendingUpdates* queue is not empty, the leader takes a value from the queue and proposes the value through *starts_propose* call. The leader picks the first uncommitted sequence number and checks the information sent by acceptors in the Prepare phase. If a value has been accepted in previous views for that sequence number, the leader will propose that value. Otherwise, the value from the *pendingUpdates* queue will be proposed for that sequence number.

When an acceptor receives a Propose message from the leader, it checks weather the message belongs to the view that it promised. If the view sequence number is valid, the acceptor will store the message in disk and reply with an Accept message.

Upon receiving $\lfloor N/2 \rfloor$ accept messages for the value it proposes, the leader is certain that a consensus is reached for this value. The leader notifies acceptors about the consensus by broadcasting a Commit message. This allows the acceptor to execute the update. An acceptor cannot execute an update if it has not received a commit message for that update.

5.5. Figures

Figure 5.1: Data Structures

```
Leader:
/* Server State variables */
1. int myServerId - a unique identifier for this server
2. State - one of {UNSTARTED, PREPARING, PROPOSING, CLOSED}

/* View State variables */
1. ViewSeq attemptedView – the view that the leader attempts to
       establish

/* Prepare Phase variables */
1. boolean[] promises - array of promises received from acceptors,
       indexed by serverId

/* Propose Phase variables */
1. UpdateSeq committedSeq - the highest sequence number of all
       consecutively committed updates
2. UpdateSeq availableSeq - the lowest sequence number that has not
       been proposed and committed
3. Map history - map of updates, with sequence number as the key, and
       each slot contains:
4.   String value – the value of this update, if any
5.   boolean committed – whether the value of this update is committed
6.   ViewSeq viewOfValue – the highest view in which this value is
         proposed
7.   boolean[] acceptVotes – array of acceptance received from
         acceptors, indexed by serverId

/* Client Handling variables */
1. Queue<String> pendingUpdates – queue of client update messages that
       have not been processed

Acceptor:
/* Server State variables */
1. int myServerId - a unique identifier for this server

/* View State variables */
1. ViewSeq promisedView - the highest view this server promised, all
       messages of lower views are rejected

/* Prepare Phase variables */
1. (disk) Message prepare - the Prepare message from the last promised
       view, if received

/* Propose Phase variables */
1. UpdateSeq committedSeq - the locally highest sequence number of all
       consecutively committed updates
2. Map history - map of updates, with sequence number as the key, and
       each slot contains:
3.   String value – the value of this update, if any
4.   boolean committed – whether the value of this update is committed
```

Figure 5.1 (cont.)

```
5.   ViewSeq viewOfValue – the highest view in which this value is
         proposed
6. (disk) Message Propose – the propose messages accepted by this
       server
7. (disk) Message Commit – the commit messages received by this server

/* Client Handling variables */
1. UpdateSeq appliedSeq – the sequence number of the last applied
       update
2. (disk) AppStateManager applicationState – An object that applies
       committed updates to the application by writing updates to the
       disk
```

Figure 5.2: Message Types

```
1. Update - contains the following fields:
2.   clientId - unique identifier of the sending client
3.   clientUpdateSeq - client sequence number for this update
4.   value - the update being initiated by the client

1. Prepare - contains the following fields:
2.   senderId - unique identifier of the sending server
3.   view - the view number being prepared
4.   committedSeq - the locally committed sequence number of the leader

1. Promise - contains the following fields:
2.   senderId - unique identifier of the sending server
3.   promiseView - the view number that the acceptor promised
4.   committedSeq - the highest sequence number of all consecutively
         committed updates
5.   info – a portion of the acceptor history that helps to bring the
         leader up-to-date

1. Propose - contains the following fields:
2.   senderId - unique identifier of the sending server
3.   view - the view in which this Propose is being made
4.   updateSeq - the sequence number of this Propose
5.   value - the update value being bound to updateSeq in this Propose

1. Accept - contains the following fields:
2.   senderId - unique identifier of the sending server
3.   view - the view for which this message applies
4.   updateSeq - the sequence number of the associated Propose
5.   isAccepted – a false value notifies the leader of a higher view,
         so the leader can quit

1. Commit - contains the following fields:
2.   senderId - unique identifier of the sending server
3.   updateSeq - the sequence number of the update that was ordered
4.   value - the update value bound to updateSeq and globally ordered
```

Figure 5.3: Conflict checks to run on incoming messages. Messages for which a conflict exists are discarded or rejected.

Leader:

```
boolean Conflict(message):
// The leader only handles promise message when its state is
// PREPARING. This method also checks whether the promise message is
// meant for the same view.
1. Promise(senderId, promiseView, committedSeq, info):
2.   if state ≠ PREPARING
3.     return TRUE
4.   if promiseView ≠ attemptedView
5.     return TRUE
6. return FALSE

// The leader only handles accept message when its state is PROPOSING.
// The leader also checks the view sequence number and update sequence
// number to rule out stale messages.
1. Accept(senderId, view, updateSeq, isAccepted):
2.   if state ≠ PROPOSING
3.     return TRUE
4.   if view ≠ attemptedView
5.     return TRUE
6.   if history[updateSeq] does not contain an update
7.     return TRUE
8. return FALSE
```

Acceptor:

```
boolean Conflict(message):
// Checks whether this acceptor has promised a higher view. If so,
// ignores this message.
1. Prepare(serverId, view, committedSeq):
2.   if view < promisedView
3.     return TRUE
4. return FALSE

// Checks whether this acceptor has promised a higher view. If so,
// ignores this message.
1. Propose(senderId, view, updateSeq, value):
2.   if view < promisedView
3.     return TRUE
4. return FALSE
```

Figure 5.4: Rules for updating the local state.

Leader:

```
Update Data Structures(message):
// Records that the acceptor with senderId has promised, and
// updates information about previous views based on info.
1. handle_promise_msg(senderId, promiseView, committedSeq, info):
2.   if promises[senderId] is true
3.     ignore this message
4.   promises[senderId] ← true
5.   for each entry e in info
6.     Apply e to local data structures

// If the update has not been committed, records that the acceptor with
// senderId has accepted this update.
1. handle_accept_msg(senderId, view, updateSeq, isAccepted):
2.   if history[updateSeq].committed is true
3.     ignore this message
4.   if history[updateSeq].acceptVotes already contains ⌊N/2⌋ Accept
       messages
5.     ignore this message
6.   if history[updateSeq].acceptVotes[senderId] is true
7.     ignore this message
8.   history[updateSeq].acceptVotes[senderId] ← true
```

Acceptor:

```
Update Data Structures(message):
// Updates local promiseView variable.
1. handle_prepare_msg(serverId, view, committedSeq):
2.   promiseView ← view

// Updates promiseView variable if this message has a higher view.
// Records this update if updateSeq is not already committed.
1. handle_propose_msg(senderId, view, updateSeq, value):
2.   if promiseView < view
3.     promiseView = view
4.   if history[updateSeq].committed is true
5.     ignore this message
6.   else
7.     history[updateSeq].value ← value
8.     history[updateSeq].viewOfValue ← view

// Updates local history variable.
1. handle_commit_msg(senderId, updateSeq, value):
2.   history[updateSeq].value ← value
3.   hisotry[updateSeq].committed ← true
```

Figure 5.5: Prepare Phase

Leader:

```
// Updates local state varaible. Constructs a new prepare message and
// broadcasts to all acceptors.
1. starts_prepare ()
2.   state ← PREPARING
3.   attemptedView ← {myServerId, System.currentTimeMillis}
4.   prepare ← Construct_Prepare(attemptedView, committedSeq)
5.   broadcast to all acceptors: prepare

// If promise messages are received from a majority of acceptors, the
// leader shifts to propose phase.
1. Upon receiving Promise(senderId, promiseView, committedSeq, info)
2.   Apply to data structures
3.   if getMajorityPromises(attemptedview)
4.     availableSeq ← committedSeq.getNextSeq()
5.     state ← PROPOSING
6.     starts_propose()

// Helper function to count whether this leader has received promise
// from a majority of acceptors.
1. bool getMajorityPromises(ViewSeq view)
2.   if promises[] contains ⌊N/2⌋ + 1 entries, p, with p.view = view
3.     return TRUE
4.   else
5.     return FALSE
```

Acceptor:

```
// Replies the leader with a promise message.
1. Upon receiving Prepare(serverId, view, committedSeq)
2.   if promiseView < view /* Establish the view */
3.     Apply Prepare to data structures
4.   Promise ← Construct_Promise(view, committedSeq, history)
5.   {Write to disk}
6.   SEND to leader: Promise
```

Figure 5.6: Propose Phase

Leader:

```
// Adds the update request to the pendingUpdates queue.
1. Upon receiving Client_Update(clientId, clientSeq, value):
2.   pendingUpdates.enqueue(value)
3.   starts_propose()

// Keeps proposing updates until the pendingUpdates is empty.
1. starts_propose()
2.   while(pendingUpdates.isEmpty() = false)
3.     if (history[availableSeq].value = null)
4.       history[availableSeq].value ← pendingUpdates.pop()
5.     do_propose(availableSeq)
6.     availableSeq ← availableSeq.next()
7.     while(history[availableSeq].committed = true)
8.       availableSeq ← availableSeq.next()

// Constructs a propose message for updateSeq and broadcasts to all
// acceptors.
1. do_propose(updateSeq)
2.   Propose ← Construct_Propose(myServerId, view, updateSeq, value)
3.   Apply Propose to data structures
4.   broadcast to all acceptors: Propose

// Records that the acceptor with senderId has accepted updateSeq. If
// accepts from a majority of acceptors are received, commits updateSeq
// and broadcasts the commit message.
1. Upon receiving Accept(senderId, view, updateSeq, isAccepted):
2.   if (isAccepted = false)
3.     state ← CLOSED
4.     quit this view and return
5.   Apply Accept to data structures
6.   if getMajorityAcceptVotes(updateSeq)
7.     history[updateSeq].committed ← true
8.     while(history[committedSeq.next].committed)
9.       committedSeq ← committedSeq.next
10.    Commit ← Construct_Commit(updateSeq)
11.    broadcast to all acceptors: Commit

// Helper function to check whether the leader has received accepts
// from a majority of acceptors.
1. bool getMajorityAcceptVotes(UpdateSeq seq)
2.   if history[seq].acceptVotes contains ⌊N/2⌋ Accepts from the same
       view
3.     return TRUE
4.   else
5.     return FALSE
```

Acceptor:

```
// Replies the leader with an accept message.
1. Upon receiving Propose(senderId, view, updateSeq, value):
2.   Apply Propose to data structures
```

Figure 5.6 (cont.)

```
3.   accept ← Construct_Accept(senderId, view, updateSeq)
4.   {Write to disk}
5.   SEND to the leader: accept

// Records that updateSeq is committed. Executes the update if all
// previous committed updates are received.
1. Upon receiving Commit(senderId, updateSeq, value):
2.   Apply Commit to data structures
3.   while(history[committedSeq.next].committed = true)
4.     committedSeq ← committedSeq.next
5.     apply history[committedSeq].value to the application
6.     {Write to disk}
```

CHAPTER 6: EXPERIMENTAL RESULTS

6.1. System Setup

We built a fault-tolerance log service based on Paxos algorithm, as described in Chapter 4 and Chapter 5. In this section we evaluate our implementation and demonstrate how the system performance is affected by various factors.

For our tests, we ran the replicated log service on multiple physical machines (typical Pentium-class machines). The servers are connected via a switch. We tested various number of machines, but for each run, the set of servers is fixed. All communication is point-to-point via UDP. Our implementation incorporates disk operations to tolerate crashes. Servers recover states from the stable storage on start.

We are especially interested in two metrics, latency and throughput, as they generally reflect the performance of a distributed service. The latency is measured as the duration from the time a client request is acknowledged by our system to the time the corresponding update is executed. Since an update is eventually executed on all replicas, we only count the first execution of such update as the ending point of the corresponding request. Throughput is measured as the number of successful Paxos operations per second. A successful Paxos operation is a Paxos instance that successfully writes an entry to the replicated log.

We used a single process to simulate a large number of clients to generate load on the servers. The client requests are made locally, so that no request is dropped in the network. Requests are made in each server, but only the leader will acknowledge the requests and proceed to process them. However, we always generate overload requests to test the performance of the system. Our system has a flow control mechanism at the leader. The leader has a window with preconfigured window size, which limits the number of outstanding proposals at any given time. We tuned this window size for each configuration to achieve the highest throughput without overloading the physical machine. Requests made beyond the service capabilities are dropped by the replicated log service automatically, before they can overload the underlying infrastructure.

The clients only make write requests to the service, because read requests are handled by the leader locally and do not invoke Paxos algorithm or any communication with other servers. Each client keeps writing entries to the log service continuously. We intentionally keep the size of each update small, typically 100KB for each. Therefore the overall latency reflects the overhead of Paxos instead of the delay for transmitting large files over the network.

Our system is by no means optimized for performance. For example, our system does not aggregate requests. Aggregation is a technique to bundle multiple requests into one Paxos update and disseminate the update in one Paxos instance.

Without aggregation, the throughput reflects the number of updates that can be ordered per second.

6.2. Evaluation

Figure 6.1 shows the latency achieved by Paxos in configurations ranging from 2 to 5 servers with a single leader. The single leadership is guaranteed by leases as described in Chapter 5. The results are demonstrated as cumulative distribution function. In the experiment, servers order 10,000 updates in each trial. The window size in this set of tests is 20.

It can be observed that the latency increases steadily when more servers join the system. The average latency with 2 servers is 85.1 milliseconds, while the average latency with 5 servers is increased to 273.1 milliseconds. We tuned the size of window to achieve highest throughput in the system. When the system is reaching the maximum throughput, the leader usually becomes the bottleneck of the system because it needs to exchange messages with every acceptor in the system. The degradation in performance with more servers in the system verified this idea.

This effect can also be demonstrated in terms of throughput. In Figure 6.2, we show the throughput of the replication service with the number of servers ranging from 2 to 5. The window size is set to 20. Again, the performance is negatively affected when more servers join the system. This also shows that the leader is saturated. When there are more servers, the bottleneck effect in the leader becomes more severe.

If the load on the leader is reduced, can the latency of requests be improved? In this set of experiments, we change the size of rate window in the leader to test latency under various load. Figure 6.3 shows the latency with different rate window size. Rate window is our mechanism to control the rate of updates. The window size is the maximum number of outstanding requests that the leader can generate. If the window is full, the leader can only issue new requests when some requests within the window are finished. In this experiment, 5 servers participated in Paxos, and we also use lease to guarantee a single leader in the system. The lease duration is 30 seconds, but the overhead for lease renewal is very small and the leader never loses its lease.

From Figure 6.3, it is clear that the latency improves dramatically with smaller window size. Especially, when the window size is 5, the average latency is 80.05 millisecond, which is even smaller than the average latency with 3 servers (85.1 millisecond) in Figure 6.1. When the load on the leader is reduced, the system can achieve similar latency even with more servers in the system.

To examine the impact of multiple leaders, in this experiment, we disable the lease and allow two servers to compete for leadership freely. In Paxos, a new leader with a greater view number always supersedes the current leader, thus the two servers in our system become leader in rotation. The server competes for the leadership again roughly 2 seconds after it loses the leadership. 3 acceptors participate in Paxos.

Figure 6.4 shows the latency of updates in this experiment. It can be easily observed that the latency increased dramatically compared to the latency in Figure 6.1. For 3 servers, the average latency is increased from 144.7 milliseconds in Figure 6.1 to 1389.4 milliseconds in Figure 6.4. The throughput, as measured, is dropped to 14.9 ops/s, from 43 ops/s in Figure 6.2.

6.3. Figures

Figure 6.1 Update Latency with Single Leader

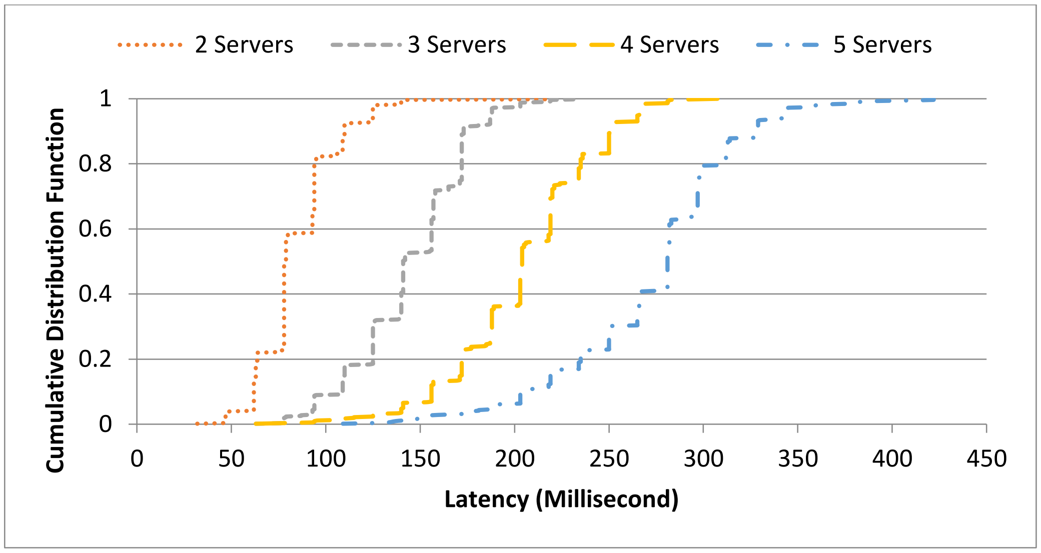


Figure 6.2 Throughput with Single Leader

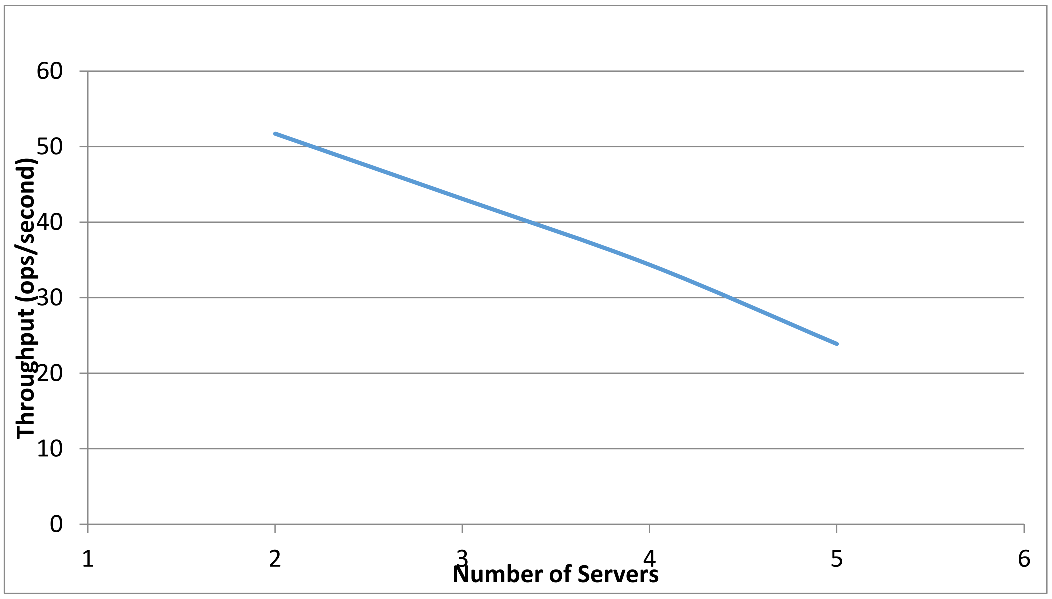


Figure 6.3 Latency with Different Request Rate

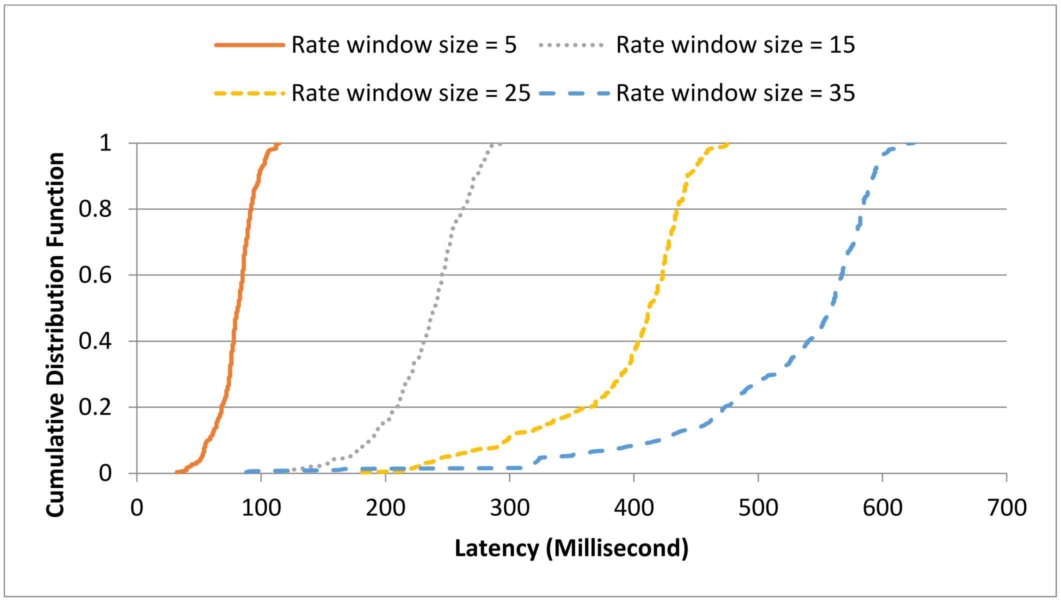
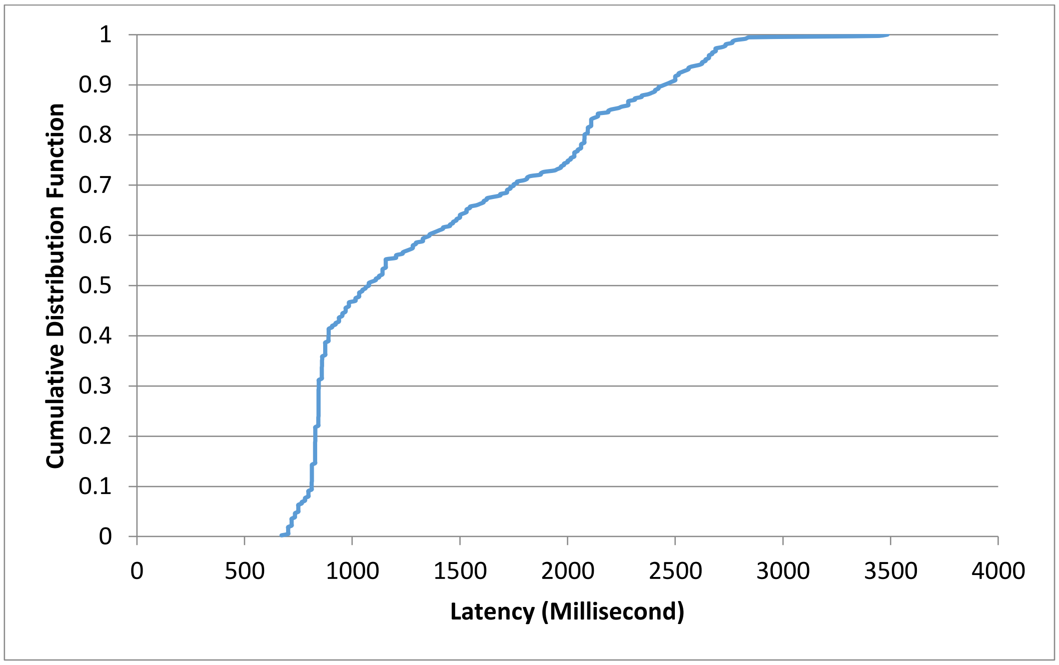


Figure 6.4 Latency with Two Leaders



CHAPTER 7: CONCLUSION

In this thesis, we studied the liveness implications of multiple leaders in Paxos. We argued that the performance of a practical system built from Paxos can be significantly improved if there is at most one leader at any given time, for three reasons:

1) Theoretical progress condition is satisfied when there is a single leader in the system;
2) The system benefits from multi-paxos optimization when the leadership does not change for longer periods of time;
3) The single leader can maintain the most up-to-date information and serve reads locally;

We proposed an optimization to achieve the single leader guarantee in Paxos by using leases. A leader is elected through Paxos and replicas grant the leader a lease during the election. A server can only function as the leader when it holds a valid lease. The lease is used like a lock with time-out to ensure there is at most one server holding a valid lease at any given time. Therefore, with this optimization, the system can have at most one leader.

We also provided a specification for classical Paxos algorithm and developed a replicated log service using Paxos. During the development, we found that there are significant gaps between theoretical study of Paxos and its practical deployment. The implementation of Paxos needs to address various problems, like leader election, reconciliation, rate control, and client handling. The choices for these protocols can greatly impact the guarantees and performance of Paxos.

Lastly, we showed the evaluation of our system and experimental results. It is observed that the leader can easily become the bottleneck of the system, especially when more servers participate in the consensus. It is also verified in experiments that the performance is significantly worsened when multiple leaders are present in the system.

REFERENCES

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (November 1976), 624-633.

[2] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558-565.

[3] Leslie Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks 2,* (1978), 95-114.

[4] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382-401.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1983. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems* (PODS '83). ACM, New York, NY, USA, 1-7.

[6] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (April 1988), 288-323.

[7] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (PODC '88). ACM, New York, NY, USA, 8-17.

[8] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133-169.

[9] C. Gray and D. Cheriton. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (November 1989), 202-210.

[10] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (December 1990), 299-319.

[11] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Barbara Liskov. 1991. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing* (PODC '91). ACM, New York, NY, USA, 1-9.

[13] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133-169.

[15] Roberto De Prisco, Butler Lampson, and Nancy Lynch. 2000. Revisiting the PAXOS algorithm.*Theor. Comput. Sci.* 243, 1-2 (July 2000), 35-91.

[16] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) 51-58.

[17] Leslie Lamport. 2003. Lower bounds for asynchronous consensus. In *Future directions in distributed computing*; Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao (Eds.). Lecture Notes In Computer Science, Vol. 2584. Springer-Verlag, Berlin, Heidelberg 22-23.

[18] Romain Boichat, Partha Dutta, Svend Frlund, and Rachid Guerraoui. 2003. Deconstructing paxos. *SIGACT News* 34, 1 (March 2003), 47-67.

[19] Leslie Lamport and Mike Massa. 2004. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks* (DSN '04). IEEE Computer Society, Washington, DC, USA, 307-.

[20] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (OSDI '06). USENIX Association, Berkeley, CA, USA, 335-350.

[21] Leslie Lamport. Fast paxos. *Distributed Computing*. Oct. 2006, 79-103.

[22] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (PODC '07). ACM, New York, NY, USA, 398-407.

[23] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. 2007. The Paxos Register. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems* (SRDS '07). IEEE Computer Society, Washington, DC, USA, 114-126.

[24] Dale Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report. Cornell University, Ithaca, NY, USA.

[25] Butler W. Lampson. 1996. How to Build a Highly Available System Using Consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms* (WDAG '96), Özalp Babaoglu and Keith Marzullo (Eds.). Springer-Verlag, London, UK, UK, 1-17.

[26] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: an overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (LADIS '08). ACM, New York, NY, USA, Article 3, 6 pages.