Evaluating Durability of Distributed Databases:

Theory and Empirical Studies of MongoDB

Konstantin Dunn

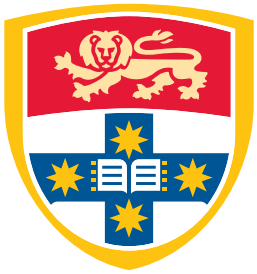
SID: 450303038

Supervisor: Prof. Alan Fekete

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

6 November 2018



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Konstantin Dunn

Signature: 

Date: November 6, 2018

Abstract

We present a study of durability properties in MongoDB focused on equipping users and developers of distributed systems with the necessary understanding and tools to increase the durability of their systems, understand the sources of data loss and empirically evaluate durability and performance tradeoffs of different solutions. As part of this, we contribute a comprehensive categorisation of cases that occur in MongoDB which result in write losses and present an experiment design which is capable of inducing failures in a way that causes write loss to occur. We then introduce the algorithm used to detect write loss using the execution history produced by the experiment and conclude with empirical results showing write loss frequency and quantity along with relevant performance metrics. Using these metrics, we detect (known) durability and performance problems in MongoDB 3.6-rc0 and demonstrate that these were fixed in MongoDB 4.0. We also offer the means to reason about the latency until the system reaches various levels of write durability, using a novel theoretical approach. The theory additionally allows users to estimate these quantities using client-accessible measurements. Using MongoDB as the foundation, we illustrate the theory and its practical application, by conducting an empirical study of durability in MongoDB based on the theory.

Acknowledgements

First and foremost, I would like to thank my supervisor Alan Fekete for his expertise and knowledge in the research discipline, providing me with plenty of guidance, feedback and always making time to meet with me. His critical eye and invaluable feedback right up until this thesis deadline have been crucial to the success of my thesis and the betterment to my understanding of the research community as a whole. Without his support, I would not have been able to achieve the level and quality of research I managed to attain.

I would also like to thank Michael Cahill from MongoDB for giving me the initial idea for the project and assisting me throughout the year with his indepth knowledge of MongoDB and how best to approach tackling this research. I would like to thank the Australian Research Council for providing the School of Information Technologies and MongoDB with a Linkage grant.

A thank you goes out to Julia Wong for persistently reaching out and making sure this thesis is on track. Your continuous proofreading, feedback and suggestions were an immense help throughout the year. Thank you to Alison Wong, Nicky Ringland and Shane Arora for proofreading my thesis and pointing out the improvements I could make. Your keen eyes were a huge help in getting this thesis to the finish line.

Super special thanks go to my fiancée Deanna Arora for her continual support in the form sweets, support and general motivation throughout the year. Thank you for being there for me through the thick and thin of writing code, running experiments, writing this thesis and proofreading it until its completion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

NoSQL databases are seeing ever increasing adoption rates due to their performance and scalability. Many such systems have the capacity to be deployed on premises or in the data center for use as Software as a Service. These systems are gaining reputation for their high performance and availability, outperforming relational databases of the same scale. These features make NoSQL databases a very attractive option for large established enterprises and small startups alike who wish their systems to be accessible around the globe while staying performant and (mostly) correct. As these databases are distributed systems, one of the properties they must provide is durability. Without this, companies may lose data. Further, the data stored must remain correct and updates must not be lost under failures. Unfortunately, all previous work testing the durability of these systems under failures has been performed in-house and no comparable metric between all major systems exists.

Durability was coined as part of the larger ACID acronym by Haerder and Reuter (1983). Durability is the guarantee that once a transaction has been committed, it will remain committed even in the case of a system failure. This means that a durable system can withstand arbitrary numbers of failures and will *never* lose an operation that it has already acknowledged. This is a vital property that is seen in practically *all* relational databases and hence is something that developers are familiar with and come to expect from their database systems. It is imperative that we gain a better understanding of durability in such distributed systems, as their use becomes ever more widespread. This must be done to ensure that developers and users of these systems are equipped with the knowledge of how to improve durability of their systems and diagnose durability failures if and when they appear.

MongoDB was first released is 2009 and has seen a high level of adoption in enterprise and small-scale deployments. As its use increased, so did the users' discovery that MongoDB and

other similar systems do not provide the same guarantees as relational databases. In particular, the makers of MongoDB had experience with durability and consistency issues in their previous version - MongoDB version 3 (Kingsbury, 2017; Patella, 2018). This is a clear indication that even cutting-edge database systems may still suffer from unexpected durability failures, further showing the need for a betterment of the community's understanding of durability.

Our work focuses on allowing systems designers and users to increase the durability of their systems by understanding and measuring the durability of writes in MongoDB under various configurations and failures. We provide a method for empirically measuring the durability of MongoDB configurations using only the operations that can be found in other major NoSQL document stores. We also present a detailed understanding of how certain configuration affect the durability of writes across the entire replica set. Throughout this thesis, we adopt a client-centric perspective towards write durability, where we observe the state of the data in the same way as it would be observed by a user and developer, as distinct from what can be observed by MongoDB itself. We focus on single-item operations as these tend to make up the vast majority of common document store workloads, while yielding the most granular level of insight into the durability properties of these systems.

1.1 Contributions

While the literature has presented many consistency and failure models, much less has appeared concerning durability. As such, we have chosen to focus our study on the evaluation of durability of writes of MongoDB during failures, applying a theoretical and empirical approach to this task.

The main objective of this thesis is to enhance the community's understanding of write durability. In particular, we seek to equip users and developers with the tools to measure and evaluate durability and performance properties of distributed storage systems. This can allow them to improve the durability of these systems and identify the sources of data loss. We also provide users with the means to reason about varying levels of durability at the individual write level and apply this reasoning estimate and evaluate the durability characteristics of their distributed storage systems. We do this by providing a theory and methodology of estimating write durability using rudimentary measurements from basic MongoDB configurations.

Overall, our contributions are outlined as follows:

(1) A categorisation and evaluation of the conditions needed to cause write durability failures.
(2) An experiment design capable of inducing a loss of writes under failures in a MongoDB replica set, producing an execution history of all events in the process.
(3) The algorithm used to process the execution history and quantify the amount of durability failures presented by the execution, while also reporting performance metrics for that execution.
(4) Empirical results to show that the experiment design and processing of its output give quantitative metrics of frequency and quantity of non-durable writes, along with performance metrics in the form of latency and throughput of operations in these experiments. In particular, we detect (known) durability and performance problems in MongoDB 3.6-rc0 and demonstrate that MongoDB 4.0 fixed these issues.
(5) A novel theoretical model for evaluating the time it takes for an acknowledged write to become durable, along with an estimation for when a write becomes durable on the primary.
(6) An empirical study of time-till-durability in terms of the time period until acknowledged writes become durable based on this theory.

Together, these contributions allow system users and designers to verify the durability of their distributed storage systems, understand the sources of data loss, empirically evaluate durability and performance tradeoffs of different solutions and reason about varying levels of durability in terms of the latency of when a write becomes durable.

1.2 Outline

The thesis is split into 4 parts, each addressing a core aspect of our research and providing a detailed account of our contributions.

Part 1 of the thesis outlines the background of the work, including a review of the academic literature relevant to this research in Chapter 2, followed by a more indepth exploration of relevant MongoDB internals in Chapter 3.

Part 2 builds on the understanding gained from the background and focuses on work regarding detection of durability failures. In Chapter 4 we present an analysis of the conditions that result

in a durability failure, categorising durability failures into three distinct types. We proceed with an experiment design and our method for measuring the quantity of durability failures given an execution history, in Chapter 5. We discuss the results, their implications and our approach in Chapter 6.

In Part 3 we use the results from Part 2 as motivation for a theoretical approach to analysing the time-till-durability. We begin by presenting a body of theoretical work regarding a granular and flexible form of write durability, where we focus on the time it takes for a write to reach a given level of durability. We then use this theory to derive a result for estimating durability in Chapter 7. This is followed up with an empirical study of time-till-durability in MongoDB based on this theory, along with a discussion of our findings in Chapter 8.

The thesis concludes with Part 4 with a discussion of the implications and significance of this research while providing suggestions for future work.

Part 1

Background

Chapter 2

Literature Review

Modern applications need to be able to grow quickly and support users across the globe. All such applications are dependent on a database backend in some shape or another. MongoDB and similar database systems promise low latency and high scalability, which are extremely attractive properties for today's application engineers. However, the distributed nature of these database systems may often result in highly unexpected behaviour - including data loss when a fraction of the participant machines experience failures. This literature review gives the reader an outline of the state-of-the-art methods for the design of distributed protocols, existing findings of database system durability failures and methods for measuring the durability of distributed database systems. In particular, we will investigate Jepsen - a tool for verifying consistency claims of distributed systems by inducing network failures. We will also inspect varying consistency models with a focus on MongoDB - an industry standard NoSQL distributed database and compare these to another well-researched system - Amazon's DynamoDB.

2.1 Replication, Consensus & Failure Models

The fundamental property of replicated distributed systems is the ability to withstand failures of individual components while simultaneously making decisions about the global state of the system. This is achieved through the use of consensus protocols as a basis for decision making. The most well-studied approach is the *Replicated State Machine* as first conceived by Schneider (1990). The mechanisms he described are flexible enough to be adapted to many different failure models, the most common of which are *Fail-Stop* (Schlichting and Schneider, 1983) and *Byzantine* (Lamport et al., 1982).

The Fail-Stop model assumes that if a component experiences a failure, it will crash and fail to respond to any subsequent requests until it recovers. After recovery, the component is to

behave normally. Handling failures under this model has been proven to be possible with $t + 1$ replicas for t failures, however no system has achieved this bound yet. This failure model has had much criticism levelled against it (Alagappan et al., 2016; Huang et al., 2017) for the oversimplified assumptions made about how a distributed system functions and the kinds of failures it experiences.

The Byzantine model, on the other hand, assumes that any component (or subset of components) can fail arbitrarily. More specifically, the model assumes that an adversary can arbitrarily partition the network for a malicious purpose and send inconsistent and conflicting messages to different components of the system. Lamport et al. (1982) showed that a minimum of $3t + 1$ replicas are required to handle t failures under the Byzantine model. This failure model lies on the opposite side of the spectrum to Fail-Stop, getting a lot of criticism for its unrealistic and overly pessimistic assumptions about how distributed systems operate and the environments in which they operate (Liu et al., 2016; Porto et al., 2015).

These failure models provide a strong basis for a whole body of research about protocols, measurements and algorithms in distributed systems. The base observations made in these papers prove to be valuable background knowledge when analysing and understanding the protocols which build upon these models.

2.2 Distributed Protocol Design

The basic failure models developed by Schlichting and Schneider (1983) and Lamport et al. (1982) have spawned much research into protocols and algorithms that allow distributed systems to function under such failures. Recently, the community has changed direction and numerous attempts have been made to refine the failure models with the aim of providing a more realistic failure model with theoretical and empirical backing.

2.2.1 Paxos

The seminal paper the field of distributed protocol design can be attributed to *Paxos* (Lamport, 1998) - the first protocol and algorithm that allows a distributed system to withstand Byzantine failures using $2t+1$ replicas. The algorithm uses a metaphor of a part-time parliament where the legislators and the messengers could leave the parliament at any time and rejoin later arbitrarily.

This metaphor maps nicely to computers, where the legislators are replicas and messages and messengers are the messages in the network. However this original algorithm is inefficient as it requires electing a new leader on every operation. Observing this, Lamport also proposed *multi-Paxos*, which only runs the leader election phase when the current leader fails.

Many improvements upon the classic Paxos algorithm have been made. *Fast Paxos* (Lamport, 2006) adds a new mechanism to reduce the messages needed for a decision. This comes at the cost of $3t + 1$ replicas instead of $2t + 1$ as with original Paxos. However Fast Paxos cannot replace the original protocol completely as there are possible scenarios where executing the "fast" variation of the protocol is impossible and a fallback to the original is required. Another improvement has been made in the form of *Vertical Paxos* (Lamport et al., 2009) which allows for performing *reconfiguration* - process of repairing and replacing a faulty replica with another - while simultaneously proceeding to decide on incoming commands and changes. The protocol achieves this by introducing an auxiliary master, which is in charge of deciding on reconfiguration operations and by separating the quorums for read and write operations. That is - machines in charge of deciding the value produced by a read operation will be different to machines deciding the writes. The main issues with all variations of paxos is the notorious complexity of the underlying algorithm and metaphors used to describe it. This makes the implementations of said algorithms prone to bugs and unexpected behaviours.

2.2.2 Alternatives & Improvements

An alternative to Paxos has been proposed in the form of *Raft* (Ongaro and Ousterhout, 2014). The authors of this protocol prove that it has the same bounds on the numbers of replicas, consistency and performance as Paxos but using techniques that are much more intuitive and have better real-world metaphors. Since MongoDB uses Raft as a basis for its consensus mechanism, it is vital to study this protocol to find how it can be leveraged to induce errors and inconsistencies in other parts of Mongo's consensus solution. Unfortunately, all follow-on work for these protocols tends to use Paxos as base, simply disregarding Raft.

For example, *XFT* (Liu et al., 2016) observes that Byzantine fault-tolerance is too strong and restrictive, while Crash fault-tolerance is too weak and unrealistic for the vast majority of distributed systems. They propose to weaken the byzantine fault tolerance limitation by assuming that errors occurring are not coordinated for malicious intent, which is expected from

managed systems. Using this perspective, the paper proceeds to provide a variation of Paxos called *XPaxos* which can withstand t byzantine failures using only $2t + 1$ nodes. It proceeds with a very extensive empirical analysis of all state-of-the-art consensus algorithms, presenting data on latency and consistency for scenarios under faults and without. The authors argue that this protocol is most useful in geo-replicated systems, as getting control of effectively half the replicas across the world is infeasible. However this is a very new and novel perspective on the problem of fault tolerance, with no commercial systems having implemented this protocol and no further papers written to build on this idea.

Further work in this field also includes *Visigoth* (Porto et al., 2015) which is a fault-tolerance protocol designed for data centers. The authors begin with the assertion that BFT models are unnecessarily conservative for data center environments, backing up the claim by observing that BFT is designed to cope with coordinated malice, which is unlikely to happen within the security perimeter of the data centre. Further, it observes that data centers have reliable and predictable network connectivity, meaning that the assumption of full asynchronicity is also too restrictive. Visigoth provides a framework to adjust the level of asynchrony and byzantine-ness of a replica set as to improve performance, reduce network overhead and reduce the number of replicas required to $2t+1$ for arbitrary errors. Since we know that MongoDB can be deployed on premises and in a data-center, so seeing its behaviour under both conditions and comparing it to VFT's guarantees and performance can offer insight into whether adopting this failure model or providing such an option would improve its performance and durability guarantees. Similarly to *XFT*, there are no papers that build on this framework, making it difficult to evaulate the impact of this framework.

Another paper that tries to tackle this issue is *ThriftyPaxos* (Shi and Wang, 2016) which observes that using *on-demand instantiation* (Lamport and Massa, 2004; Wood et al., 2011) and *lazy recovery* (Ladin et al., 1990, 1992) can allow distributed systems to handle byzantine failures more efficiently, using only $2t + 1$ replicas. The key observation driving this approach is that logical separation of *agreement nodes* and *execution nodes* allows for lazy recovery. The authors propose a variation of Paxos - ThriftyPaxos which incorporates the logical separation of nodes into its execution. The paper evaluates its performance in comparison to Paxos and some CFT algorithms for performance, availability and adaptive recovery, however provides no empirical data for live distributed systems.

Attempting to solve this problem in a completely different approach comes from *NOPaxos* (Li et al., 2016) which proposes an overhaul of how this problem should be tackled. Instead of trying to deal with unordered messages in an asynchronous environment at the application layer, like Paxos and Raft, the authors deal with these issues on different layers. Firstly, they design a new network primitive - Ordered Unreliable Multicast where all receivers are guaranteed to process multicast messages in the same order, but messages may be lost. The paper proceeds to show multiple ways of implementing this primitive, showing that it has no impact on network performance when implemented using datacenter hardware. Secondly, they introduce NOPaxos - a consensus algorithm utilising the OUM. It requires application-level coordination only to handle dropped packets, a fundamentally simpler problem than ordering requests. The resulting protocol is compared to other state-of-the-art consensus methods and is found to be simpler, achieving throughput ∼5x that of other protocols - comparable to an equivalent unreplicated system.

2.2.3 Summary

Observations made in Paxos and its alternatives make for a very extensive body of work on which to base how MongoDB will respond to failures. This knowledge will be vital in designing a standardised durability testing framework for measuring the guarantees MongoDB claims. This is especially important as many of the protocols proposed above do not provide empirical evidence in a comparable way, hence making it very difficult to evaluate their performance, benefits and drawbacks against one another.

2.3 Failure Analysis

Due to the very dynamic and fast-paced nature of this field, numerous papers have been published finding bugs and inconsistencies in the general operation of replicated database systems such as MongoDB or proposing new ways to observe, analyse and mitigate errors and failures in distributed systems.

One of the key papers in this field is *Correlated Crash Vulnerabilities* (Alagappan et al., 2016) which observes that, generally, replicas in a set will not all crash at once, and at least some of the replicas are correct at any point in time. It proceeds to consider a scenario in which

all replicas of a particular data shard crash at the same time and recover at a later point and calls this a *correlated failure*. It proceeds by introducing PACE - a novel framework for exploring correlated crash vulnerabilities. It also shows that different filesystems will affect the susceptibility of a system to correlated crash vulnerabilities. Using these findings, the paper analyses many popular distributed databases for correlated crash vulnerabilities, finding 1 on MongoDB using WiredTiger engine and 5 on MongoDB using Rocks engine. This paper provides a great foundation upon which to build further research in this area, specifically investigating other forms of correlated crashes. However, this paper only assumes a *Crash Failure Model* which is not fully representative of many production systems.

The *Gray Failure* (Huang et al., 2017) paper introduces a new perspective on how distributed systems misbehave. The authors observe that most severe system outages are caused when a component of a system does not notice a fault but still gets affected by it. It derives a definition for such failures as Gray Failures and observes that their key property is *differential observability*, where one entity is negatively affected by the failure and another entity does not perceive the failure. Arguing that Byzantine Fault-Tolerant solutions are overkill, the authors use the differential observability definition to define practices on how to handle gray failures. Specifically, it notes that gray failures can only be detected in a distributed fashion - as different components have different views of the system and aggregating those results would provide a more holistic view of the system health. Unfortunately, the paper lacks empirical evidence to back up its claims. Its observations of gray failures could become a very valuable tool to test how MongoDB would handle such failures.

2.3.1 Summary

While this field of research tends to lack in depth and breadth, the observations made within these papers provide a vital perspective on further analysing and discovering theoretical and practical failures in industry-standard replicated database systems. Specifically, the lack of a standardised measure or framework makes analysing the severity of these failures difficult to analyse and compare.

2.4 Durability Metrics

There are few metrics designed specifically to measure the durability of a distributed system. Most similar metrics revolve around measuring the probability of data loss of RAID arrays. This only further signifies the need for a standard benchmark of reliability.

The standard way of measuring and reporting the reliability *MTTDL* (Gibson, 1990). This measurement uses the scale of the system, the reliability of its components and the rate of recovery to determine how long the system may last without data loss. Unfortunately, this measurement has come under a lot of criticism for its overly optimistic estimates and incomparable results. For example, (Greenan et al., 2010) show that modern archive storage systems can achieve MTTDL of 1400 years, which is obviously well beyond the lifetime of any system. They further propose NOMDL - Normalised Magnitude of Data Loss. This measurement attempts to compensate for the shortcomings of MTTDL by measuring the amount of data loss in bytes over a fixed period of time t per usable terabyte of storage. The authors argue that NOMDL is more meaningful, understandable and comparable than MTTDL, BHL (Rosenthal, 2010) and DDF pKRG (Elerath and Pecht, 2007) while being insensitive to technological change.

Further work in this field includes *Network Aware Reliability Analysis* (Epstein et al., 2016) which notes that the task of accurately measuring reliability of a distributed system is intractable using current methods. The main reason behind this is because very low-probability failure events create a large impact on the reliability measure. The authors further claim that existing approaches suffer from unrealistic expectations regarding network bandwidth. The paper proceeds by proposing a new measurement framework with the focus on the cumulative effect of simultaneous failures on repair time, taking the network bandwidth into account. They also note that a node's repair time is a function of disk bandwidth *and* network bandwidth, which - when taken into account - increases the variance in the probability of data loss estimates by two to four orders of magnitude, hence providing a more accurate perspective on the problem of data loss in a distributed replicated system. The observations in this paper present a vital perspective which can be used to tackle similar questions and perform relevant measurements when attempting to induce errors in MongoDB.

2.4.1 Summary

The metrics found in the literature tend to be centered around single machines with multiple disks configured in a RAID array. There is a gap in these metrics when distributed systems are introduced, making the development of a consistent, comparable and meaningful durability metric for a replicated storage system an important task which would open many pathways for further research.

2.5 Consistency Models

Consistency models provide ways to verify whether a storage system is behaving correctly under the limitations and guarantees imposed by the model.

The most successful model for consistency originated as a result of work in relational databases. Haerder and Reuter (1983) define the ACID consistency model, which states that every database transaction must be *Atomic, Consistent, Isolated & Durable*. This means that each transaction must complete in full or not at all; updates to data must keep the database in a valid state; the effects of concurrent transactions will be the same if those transactions were serialized; and once a transaction is committed, the change will not be lost.

Brewer (2000) conjectured that it is impossible for a distributed data storage system to satisfy *Consistency, Availability & Partition Tolerance* simultaneously. This was later proved by Gilbert and Lynch (2002). The resultant CAP theorem therefore limits the capabilities of a theoretically optimal distributed system to either guaranteeing consistency or availability since we cannot sacrifice partition tolerance in a distributed system. Brewer (2012) revisits the theorem and admits that CAP is misleading as it oversimplifies the relation between the three properties as binary - either the property is satisfied completely or not at all.

As a result, Abadi (2012) builds on the CAP theorem, creating PACELC. He adds that further tradeoffs need to be made between latency and consistency even when no network partitions exist. The main advantage of this model is its more complete portrayal of how distributed systems operate in the presence of network partitions and without.

BASE (Pritchett, 2008) is the distributed systems' alternative to ACID. It is the conceptual equivalent to the CAP tradeoff of Consistency for Availability. The model states that the

system should be *Basically Available, Soft state & Eventually Consistent*. Vogels (2009) defines eventual consistency as servers *converging* to the same state in the absence of updates. As such, BASE provides us with a perspective on the favourable properties and behaviors of a distributed system.

2.5.1 Summary

The consistency models of data storage systems have seen many iterations through the lifetime of the field. The shift in perspective from strongly consistent systems to eventually consistency has given rise to highly available, scalable and performant distributed systems. While much work has been done on defining the models and studying their theoretical limits, little progress has been made in the empirical evaluation of the models, especially how the durability of a system affects its consistency and latency guarantees.

2.6 MongoDB

MongoDB is an industry-standard NoSQL document store with capabilities for distributed sharding and replication. It uses a custom method for consensus on replica sets using a modification of the Primary-Backup replication model via a combination of Raft for leader election and an operations log (Oplog).

The Oplog keeps a record of all operations that modify the data stored in the database. The elected leader will push its oplog changes to the secondary MongoDB nodes in order to make the data consistent across all replicas.

It is commonly found in literature in comparisons with other NoSQL databases, mostly for performance. The work regarding durability guarantees of MongoDB or other NoSQL document stores is severely lacking, highlighting the need for a standard benchmark to test the claims these systems make about durability.

MongoDB has long been a source of critique and research of database properties in distributed systems. Kingsbury (2013) was the first to perform an evaluation of MongoDB's consistency properties, finding that it lost all acknowledged writes at all consistency levels when a network failure occurs. MongoDB has since been fixing these errors and improving their guarantees,

however some violations still persist (Patella, 2018). Considering the seeming complexity of ensuring consistency properties of MongoDB, we should also address any possible failures present in its durability properties.

2.7 Dynamo

Amazon's Dynamo (DeCandia et al., 2007) is a classic distributed key-value storage system that has fundamentally changed the way we reason about data storage at large scale. Its influential ideas and principles paved the way for the entire field of NoSQL database research and development and resulted in the development of more complex AWS services such as DynamoDB (Sivasubramanian, 2012) and Aurora (Verbitski et al., 2017).

The authors of this paper introduced a revolutionary multi-master replication scheme which traded strong consistency for availability and performance. The basic principle of the scheme involves making each node responsible for a chunk of the overall data and being able to reroute a request to the correct replica in a single step. To satisfy availability guarantees, data is replicated across multiple nodes across the entire system, with synchronisation of state between replicas being handled by a quorum-based protocol.

These design tradeoffs allowed Amazon to achieve performance levels that would have been impossible with traditional relational databases, providing for a compelling case-study of the limitations and tradeoffs between relational and NoSQL storage solutions.

2.8 Jepsen

Jepsen is an open-source software library designed to test consistency claims of distributed systems. Jepsen is also used to simulate certain extreme failure scenarios and observe how the system tolerates and recovers from these. Jepsen achieves this by performing a series of operations on the distributed system and analysis whether the execution of this operation satisfies certain consistency properties.

The most notable work with Jepsen on MongoDB was done by Kingsbury and Patella (Kingsbury, 2017; Patella, 2018), who performed several analyses of MongoDB's consistency claims and found a variety of violations caused by inducing network errors. Unfortunately, Jepsen does

not have the capability to induce local errors on a single replica, which prevents it from being used to test and measure the durability of these systems.

2.9 Conclusion

We have studied the literature regarding distributed systems with a focus on replication & consensus protocols, failures, durability metrics and consistency models. We have investigated MongoDB and how it achieves replication. We have also observed that the literature in these fields is very extensive on theory but empirical analysis is lacking. There is a clear absence of a standardised metric or framework for durability of distributed data storage systems which makes it impossible to fully evaluate the tradeoffs between different systems.

Chapter 3

Background

This chapter is dedicated to equipping the reader with the understanding of the properties of MongoDB that make it susceptible to durability failures and how these are addressed in the MongoDB architecture by providing them with relevant background knowledge of MongoDB and its replication mechanisms. We expand upon the Section 2.6 focusing on the technical details relevant to our contributions. The material is taken from the online documentation, code and explanations from experts on MongoDB.

3.1 MongoDB

A MongoDB database holds *collections* of *documents*, where a collection is the equivalent of a table and a document is equivalent to a row in a relational database. Each document may contain an arbitrary arrangement of fields and values - not being limited by a schema.

MongoDB is able to provide high availability and redundancy via *replica sets*. A MongoDB replica set is comprised of an odd number of MongoDB instances, each maintaining a copy of the same set of data. Through an automatically orchestrated election, one of these instances is determined as the *primary* replica, while the remaining nodes are labeled as *secondary* replicas. Only the primary replica will receive write operations propagating those writes to the secondaries via the *Oplog*. The secondary replicas will continuously read and apply the changes in the Oplog on their own data set as to maintain identical data to the primary.

MongoDB's default storage engine is WiredTiger. This engine regularly creates snapshots of the data and writes them to disk in a consistent way across all data files. These snapshots are therefore durable and can act as recovery checkpoints in case of data corruption. During the writing of a new snapshot, the previous snapshot remains valid. As such, if MongoDB

encounters an error while writing the snapshot, it can use the previous one to recover. By default, WiredTiger is configured to create snapshots every 60 seconds.

To complement long-term durability guarantees of snapshots, WiredTiger implements a write-ahead transaction log to persist all data modifications between checkpoints. WiredTiger creates one journal record for each client-initiated write operation and assigns it a unique identifier. This includes any internal writes that are initiated by the database as a consequence of the client operation (e.g. updating indexes). WiredTiger buffers these writes in-memory, persisting them to disk every 50 milliseconds.

3.2 Replication

A functioning MongoDB replica set requires that a primary node to always be up. This is because only the primary can accept write operations and propagate those writes to the secondary nodes. When a primary receives a write operation from a client, at minimum, it will perform the operation on its copy of the data before responding to the client with an acknowledgement of this operation. The primary will proceed by journaling the write and waiting for the journal to flush to disk. After the flush, the primary will add the operation into its Oplog.

The secondary replicas then read the Oplog and apply these operations to their own copy of the dataset. Once an operation is applied, the secondary sends an acknowledgement of this operation to the primary for it to track how many nodes have applied each operation to their copy of the data. This process insures that a secondary replica has data that is consistent with the primary up to a fixed point in time. As such, in the event of a failure in the primary, a secondary node can take over its role while suffering minimal data loss.

To maximise the speed of replication, each secondary has a cursor that points to the end of the primary's Oplog. When a new write operation is performed, the primary adds the operation to the Oplog and notifies the replicas that the Oplog has been updated. The replicas proceed to move their cursors forward, applying the new operations and updating their own Oplog.

As all secondary nodes have their cursors pointing to the same Oplog, they will all process the operations in the same order. This means that at any point, all replicas will have a *common*

prefix of all write operations submitted to the primary node i.e. if there are two writes w_1 and w_2, it is *impossible* for one replica to have *only* w_1 and another to only have w_2.

3.3 Failure Detection & Election

In order to maximise availability, MongoDB uses a simple heartbeat scheme to automatically detect when a replica set member fails. Each node sends heartbeats (pings) to all other nodes every two seconds. If the node fails to reply to a heartbeat within a preconfigured amount of time (10 seconds by default), the replica member sending the heartbeat will mark it as inaccessible.

If the inaccessible node is a secondary replica member, no further work is performed by the replica set, as the functioning primary can still process write operations and queries. If a primary is marked as inaccessible, the member that detected the failure will call an election to select a new primary. The elected primary is guaranteed to have the most up-to-date view of the data. That is, if secondary A saw an operation o but secondary B hasn't, then secondary A will become the new primary.

3.4 Write Concern

The *write concern* setting in a MongoDB client allows each client to configure the strength of write acknowledgement independently. Upon issuing a write operation, the client is blocked until it receives an acknowledgement of the operation. Our study uses the *w : 1, journaled and majority* settings of write concern.

The most permissive, and hence performant, write concern *w : 1* (also referred to as primary write concern) acknowledges the write just after the primary applies the operation on its data. This value can be modified to be *w: k* where the primary will wait for $k-1$ secondary replicas to acknowledge the write before the primary sends an acknowledgement to the client. An example of a w:1 execution can be seen in Figure 3.1

The *journaled* setting builds on this, delaying the acknowledgement until after the primary persists the operation into the journal. The journal does not immediately write persist the

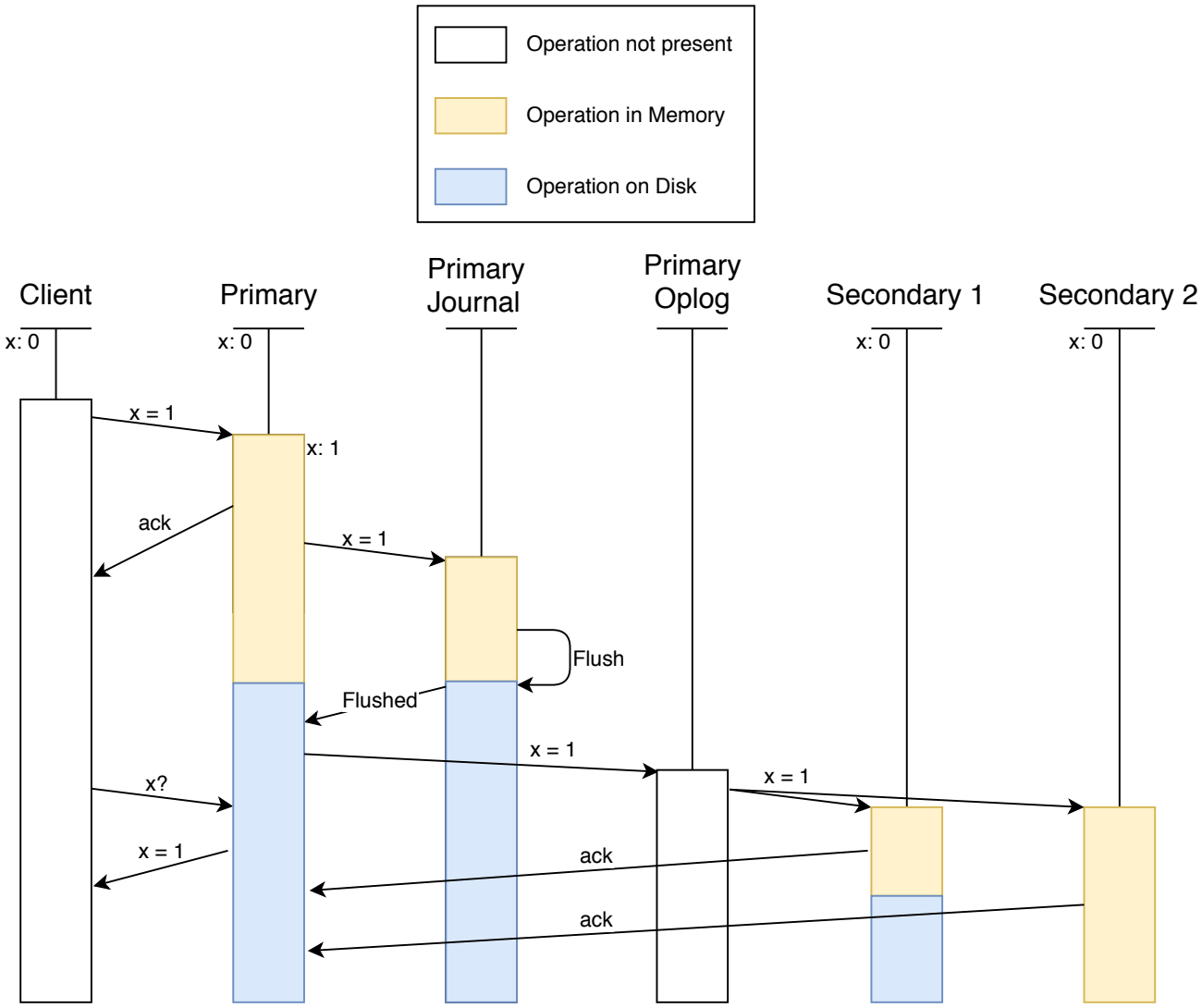


Figure 3.1: Time-space diagram of how a write operation behaves with *primary* write concern

operation on disk, instead buffering multiple operations before persisting them to the journal as a batch as seen in Figure 3.2. An example of journaled execution can be seen in Figure 3.3.

Finally, *majority* will not acknowledge the write until more than half of the nodes have also acknowledged it. An example of this is seen in Figure 3.4. We should bring your attention to the fact that the only change between figures 3.1, 3.3 and 3.4 is the time at which the acknowledgement is sent back to the client.

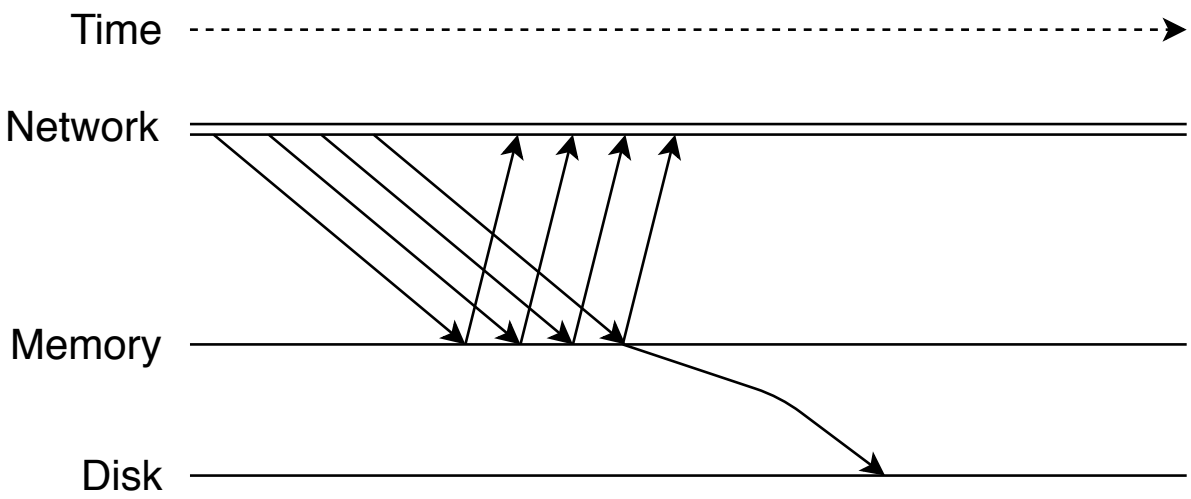


Figure 3.2: Time-space diagram of MongoDB buffering writes from the network before journaling.

3.5 Read Concern

The MongoDB *read concern* option allows each client to control the consistency properties of the data read from a replica set. Our study uses the *local, majority and linearizable* settings for the read concern.

Clients making a query with a *local* read concern will retrieve the most recent data from the replica that received the read request. A query with *majority* read concern will return data that has been acknowledged by a majority of the replica set members, while *linearizable* concern will return data that reflects all majority-acknowledged writes completed prior to the start of the query.

3.6 Read Preference

The *read preference* setting specifies how a client will route queries to the MongoDB replica set. By default, all reads are sent to the primary, however this can be changed to increase availability. Our experiments use the *primary, primary preferred and secondary* settings.

The *primary* setting is the default and will only query the primary replica for data. The *primary preferred* setting is effectively the same as default, but uses secondary replicas as a fallback if the primary appears to be down. The *secondary* preference will never query the primary replica.

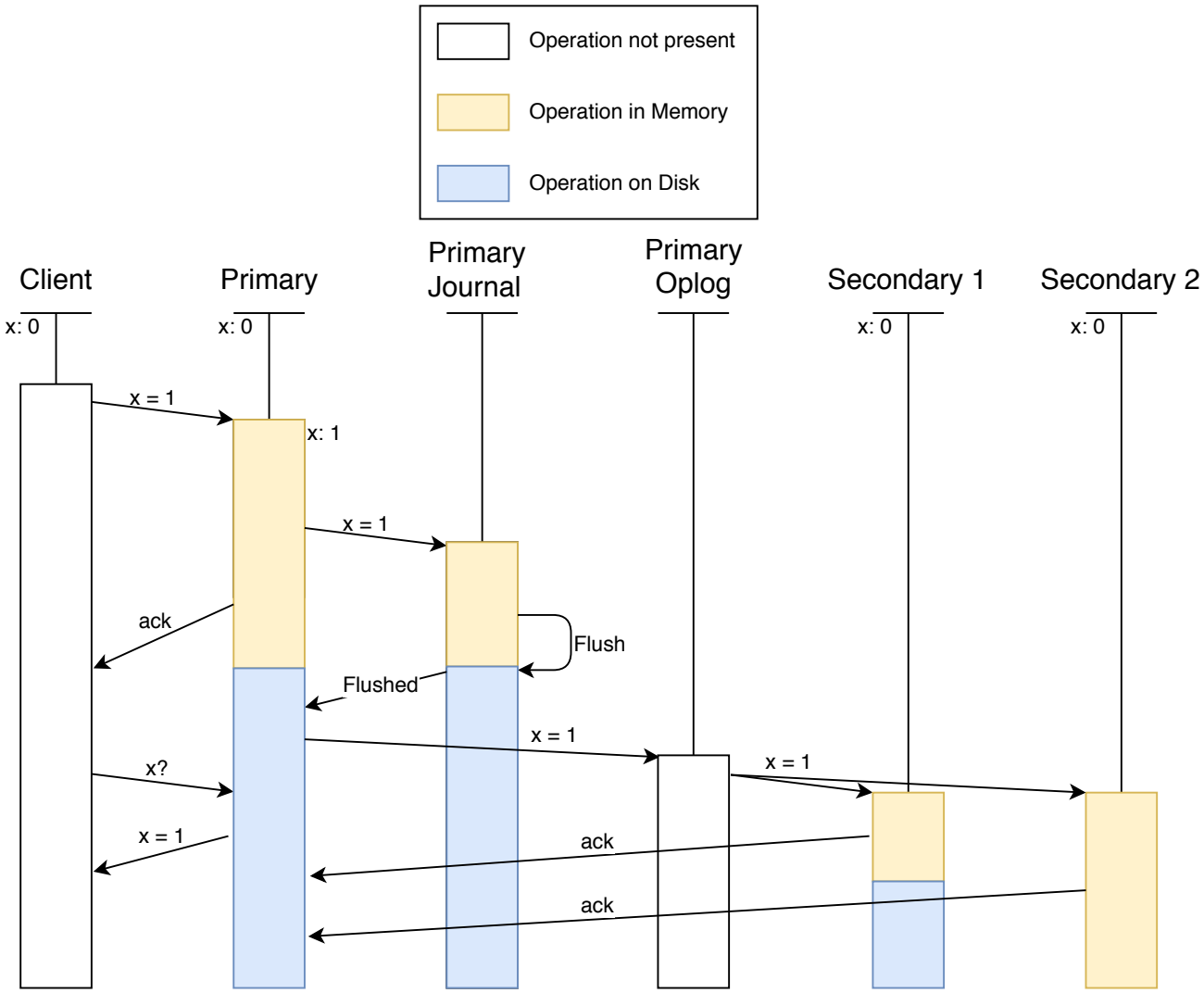


FIGURE 3.3: Time-space diagram of how a write operation behaves with *journaled* write concern.

3.7 MongoDB vs Dynamo

In the previous chapter, we discussed MongoDB and Dynamo as two types of NoSQL databases, with MongoDB acting as a document store, while Dynamo being a simpler key-value store. We now bring your attention to the differences in the way these two systems perform replication and sustain high availability.

While MongoDB uses a Primary-Secondary replication scheme and requires the primary to first acknowledge an operation before propagating it to the secondaries, Dynamo utilises a multi-master scheme. In Dynamo's scheme, every replica is in charge of a chunk of the total data

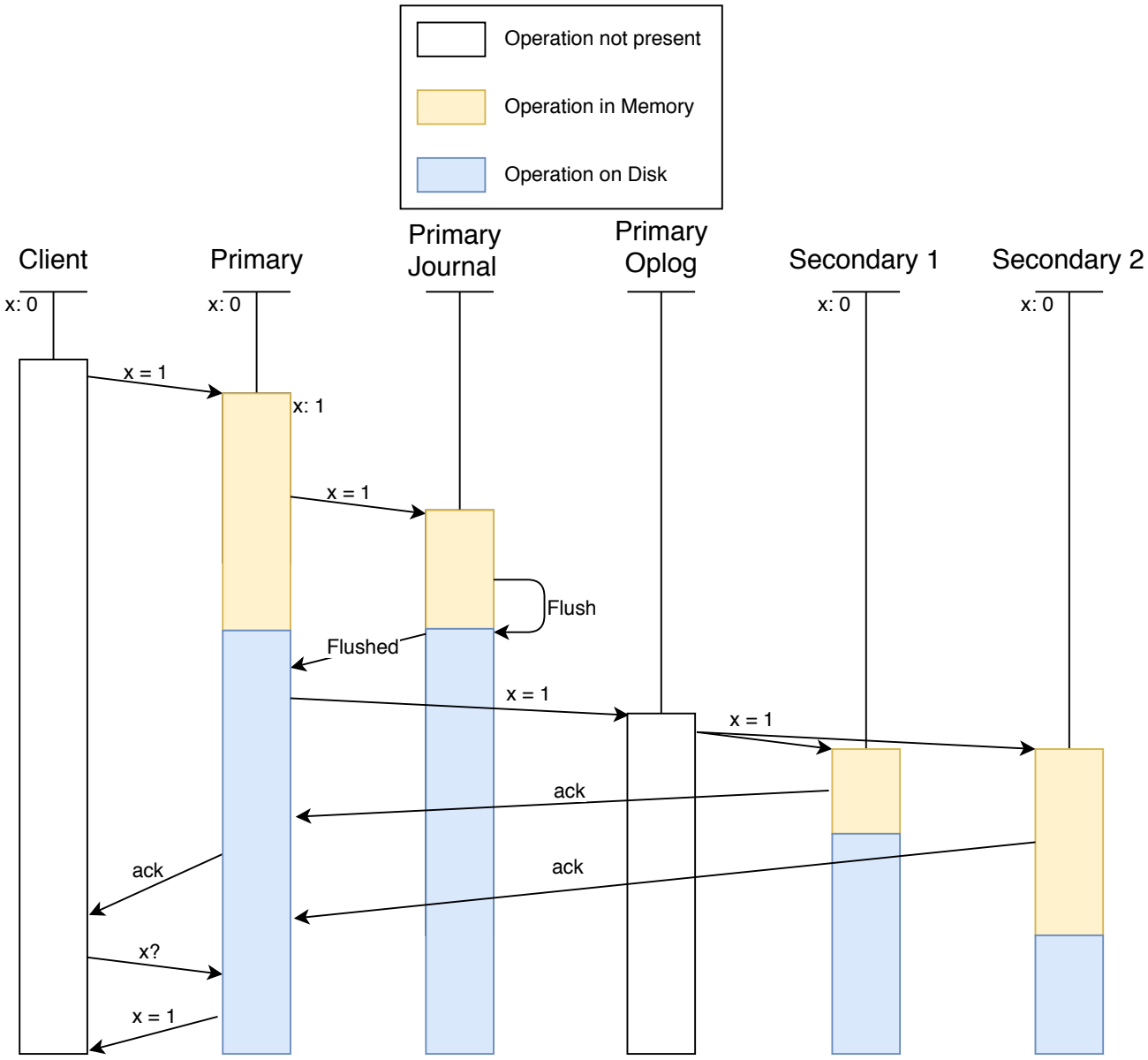


FIGURE 3.4: Time-space diagram of how a write operation behaves with *majority* write concern.

and every replica can accept a write operation. Upon receiving a write operation, a Dynamo replica will apply the operation to its copy of the data if the key is in its range, and forward the operation to other nodes in charge of this data item. This way, Dynamo ensures high availability of its data and failure tolerance.

We should also note that MongoDB's Primary-Secondary replication scheme requires an election if the Primary fails. An election can potentially reduce the availability of the system, as no

write operations can be accepted while a Primary is unavailable. On the other hand, Dynamo's multi-master strategy utilises a gossip-based protocol for identifying failed replicas, which allows it to stay completely available while replicas rearrange themselves to spread out the load and data of the failed replica, using replication as the means of ensuring durability.

While Dynamo's properties have been extensively researched, the fundamental differences between the designs of these two systems warrant a in-depth look into MongoDB.

Part 2

Detecting Durability Failures

Chapter 4

Theoretical Analysis

In this chapter we show the reader that MongoDB cannot always guarantee write durability. We do so by establishing the conditions required for a write to be lost, using a theoretical approach based on the the background information of how MongoDB processes and persists write operations. We conclude with a categorisation of MongoDB's write durability failures in terms of these conditions. This theory will then motivate the measurements described in Chapter 5 to help users evaluate the risk of lost writes.

4.1 Introduction

Recall from Chapter 3 that MongoDB replication scheme works in a primary-secondary configuration. That is, a single primary node is elected and becomes the *arbiter of all write operations*. This means that the primary will receive all write operations before informing the secondary nodes of the operations' existence. From this we can intuitively conclude that a crash can result in a lost write if that write has been acknowledged by the primary but has not been persisted to disk.

However, MongoDB replica sets also support elections when a primary crashes, in order to elect a new primary and minimise downtime. This creates a situation where a secondary can get promoted to the role of a primary without having received all operations from the crashed primary. In these circumstances, when the crashed node goes back online, it must *roll back* any operations that it acknowledged but are not on the new primary. This can directly result in writes being lost despite being acknowledged and persisted.

```
write x = 1
ack   x = 1
...
read  x -> x = 0
```

Figure 4.1: Sample scenario depicting write loss

4.2 Write Loss

Recall that, in a durable system, all acknowledged writes must remain committed even in case of system failure. As such, it is a violation of durability if an acknowledged write is not reflected in the reads following the acknowledgement. Therefore, we define a write as lost when a read on the document returns a version older than the latest acknowledged write. An example of this can be seen in Figure 4.1.

We observe that a write loss may be permanent or it may merely be transient. Permanent write loss involves an acknowledged write being erased or rolled back such that its effects are completely removed from the data and *all* subsequent reads return a version older than the lost write. On the other hand, a transient write loss involves a read returning an older version temporarily, with the latest version of the document eventually being returned.

4.3 Write Loss Scenarios

On the assumption that the only way a machine could fail is to crash, the conditions for permanently losing an acknowledged write will fall broadly into two categories:

(1) Rollback due to re-election
(2) Primary crashed before persisting

Observe that category 1 losses are a direct consequence of how long it takes for the node to receive a write while category 2 losses are related to the time it takes to make the write durable.

We also find that a transient write loss can occur if the client's read preference is configured as "Primary Preferred".

We now present each of these scenarios in detail.

4.3.1 Rollback due to re-election

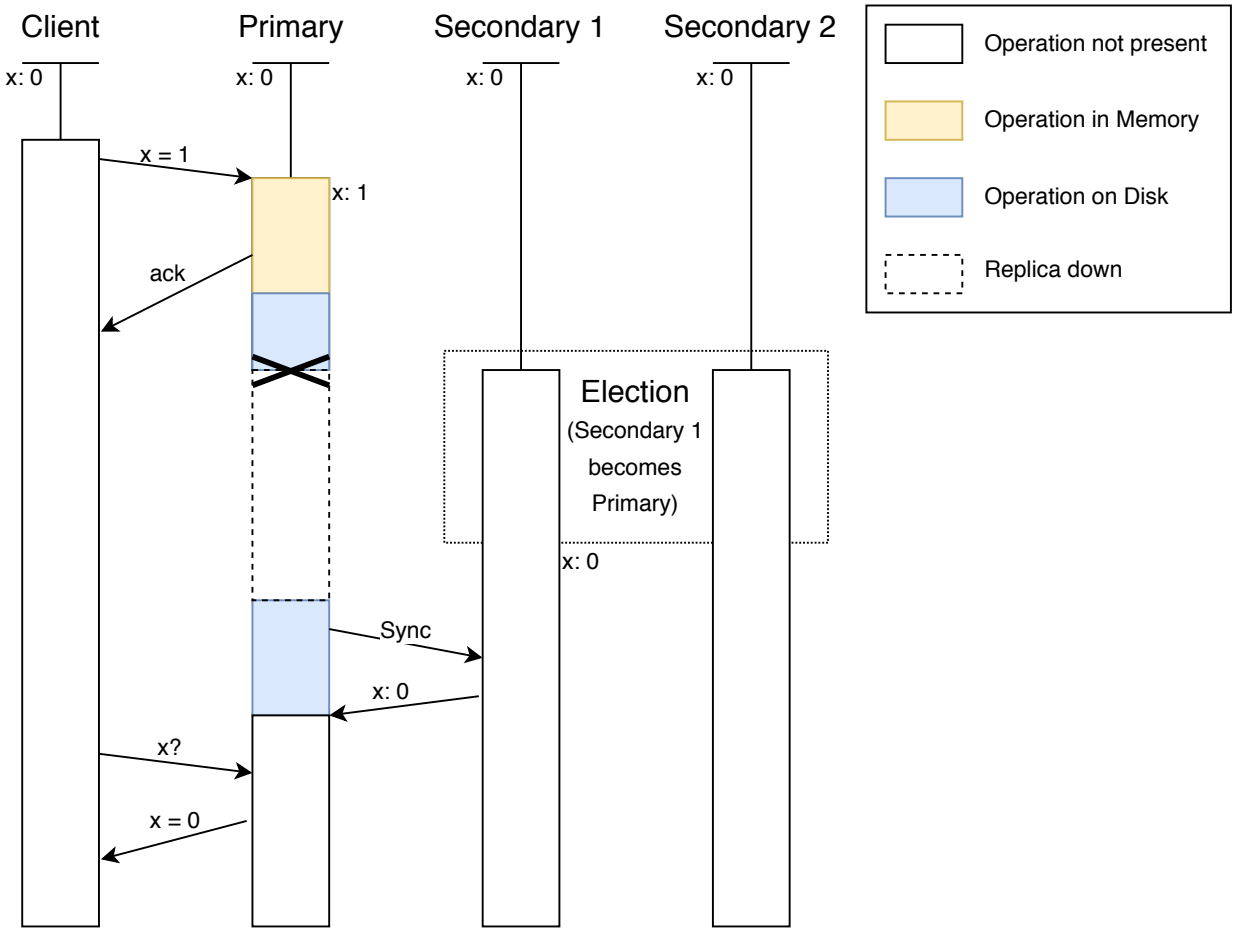


Figure 4.2: A time-space diagram of a scenario where a write gets lost due to a re-election

We define the entire sequence of operations currently in the primary's Oplog as O. This failure is induced when the replica set notices that the primary node (p) has crashed. At this point, it is possible that *none* of the secondaries managed to read the entire Oplog. This results in the existence of a sequence of one or more operations that have been committed to the primary but left completely unknown to the rest of the replica set. For MongoDB, this sequence *must* be a suffix of O (see Section 3.2), which we will denote as $\hat{O}$.

Upon a re-election, the replicas will announce the last operation they applied (o). MongoDB's election procedure will elect the primary, such that it minimises the size of $\hat{O}$, however since there are *some* operations have have been seen by *none* of the replicas, $|\hat{O}| > 0$ must hold.

After being elected, the new primary (p') will establish its Oplog as the source of truth. This Oplog will contain operations $O - \hat{O}$. When the original node p goes back online, it still has the operations applied as per Oplog O. Since p is no longer the primary, it must synchronize its state with p', which will involve *rolling back* all operations in $\hat{O}$. This is precisely the loss of writes as a result of a rollback.

As such, this failure scenario can occur with primary write concern and journaled, but it will not occur (in a properly implemented MongoDB system) with write concern majority.

4.3.2 Crash before persisting

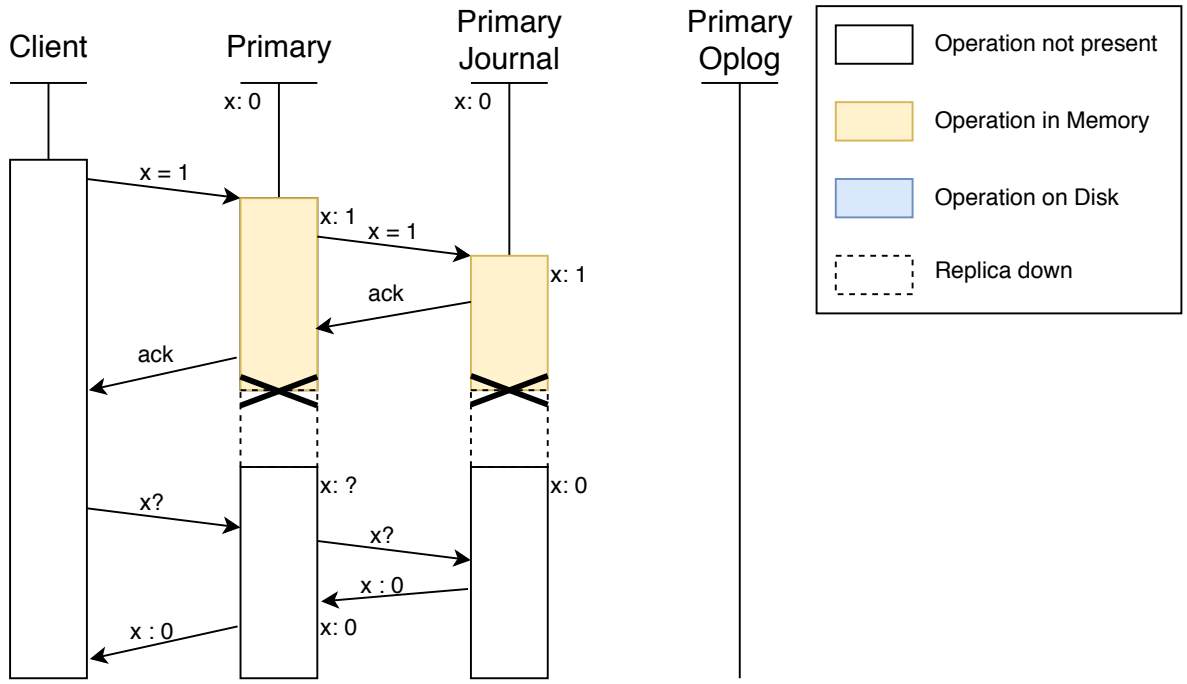


Figure 4.3: A time-space diagram of a scenario where a write gets lost due to a failure to persist it on disk

This failure relies on two core properties of MongoDB - acknowledging a write before persisting it and only adding a write to the Oplog after flushing it to disk.

Take an operation o, with write concern *journaled*, which has just been received by the primary. Currently, the primary stores this operation in memory, performs the operation on the in-memory copy of the data and buffers the write to the journal, but does not yet flush it to disk. Recall that a write is added to the Oplog only *after* it has been flushed to disk. This creates a situation where secondary nodes do not know about a write that has been acknowledged by the primary. As such, if a primary is to crash it will "forget" about o as it was only stored in memory. None of the secondary replicas would have seen o either. A client querying the primary will force it to read the last value of the document operated on from disk. The value returned from the read will not account for the latest operation, which demonstrates the operation has been lost.

Thus, this failure scenario occurs with primary write concern but may also occur with journaled, if the write is buffered instead of being immediately persisted. It will not occur (in a properly implemented MongoDB system) with write concern majority.

4.3.3 Transient Loss via Primary Preferred Read Preference

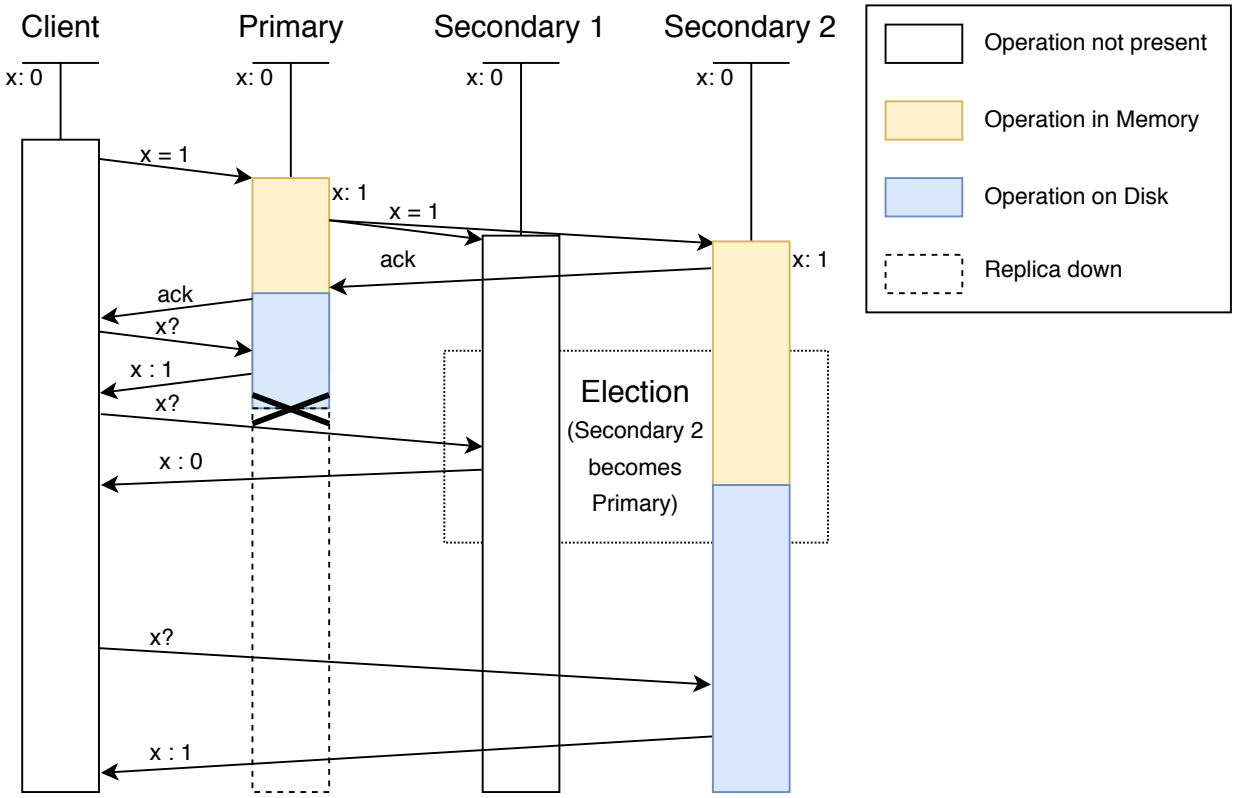


Figure 4.4: A time-space diagram of a scenario where a Majority write concern write gets temporarily lost as a result of the MongoDB client choosing the wrong Secondary to read from.

Recall from Section 3.6 that the Primary Preferred read preference will query the secondary if the primary appears to be down, instead of waiting for a primary to respond. The MongoDB client will only query one secondary. This can create a situation where a write will seem to be available, lost and available again.

Specifically, consider an operation o that was submitted with a majority write concern to the replica set. The primary will acknowledge the write when $\lfloor \frac{|R|}{2} \rfloor$ secondaries also acknowledge the write. This leaves $\lfloor \frac{|R|}{2} \rfloor$ secondaries that may not have acknowledged or even received the write. As such, if the primary is to fail immediately after acknowledging the operation, the MongoDB client has $\frac{\lfloor \frac{|R|}{2} \rfloor - 1}{|R|}$ chance of querying a secondary that has not seen o. Intuitively,

this probability reduces with time as more replicas apply the operation to their copy of the data.

If the client is to query the primary, the data returned will have operation o applied to it. Once the primary fails, the MongoDB client has a chance of querying a secondary that has not applied the operation, hence the data returned from that query will not have the operation o applied. Querying the new primary after an election will once again return data with operation o applied.

As such, Primary Preferred read preference can create instances of transient write loss, where acknowledged operations seem to disappear and reappear during a failure. This scenario can occur in all write concerns, but "all".

Chapter 5

Experiment Design & Methodology

In this chapter we offer a design and methodology of an experiment and analysis that can quantitatively evaluate the durability of a distributed storage system by measuring the frequency and quantity of write loss under failures. The experiment induces the conditions we identified in Chapter 4, and Chapter 6 will report on running the experiments on MongoDB.

5.1 Database Contents

Each document has a simple ID & value structure as seen in Algorithm 1.

Algorithm 1 Example document

```
{
    Id:  5,
    Val: 7
}
```

5.2 Methodology

The experiment is set up by creating three identical virtual machines with the database system installed and configured to run in a replica set. We then run a stress test on the replica set using a single client on the host machine which initiates many operations, on separate threads, in parallel. The operations performed are reads, writes and updates on randomly chosen documents with randomly chosen values.

During the run of the experiment, the harness records an execution history, keeping track of each operation performed and error returned by the replica set. Each read, write and update operation is individually recorded into the execution history along with the document id corresponding to the document the operation is performed on, the timestamp and duration of the

```
R,5b9f25ef2644855182a9201c,1042661835,66.689,1537156641143
U,5b9f25ef2644855182a9201c,705768886,1.132,1537156641288
ERR,W,5b9f25f82644855182a92ff2,834995024,22.04,1537156641795
R,5b9f25ef2644855182a9201c,705768886,58.42,1537156641830
```

Figure 5.1: Sample excerpt of an execution history

operation. Each read record contains the value returned by the database, while the write and update records contain the new value written to the document. Any operations that result in an error are also recorded with the same data.

The experiment is broken into three stages with each stage occupying a third of the experiment time:

Standard operation: The virtual machines are started in "Headless" mode and run normally.

Operating under failure: A failure is induced on the primary. The primary replica disconnects from the replica set initiating an election.

Recovery: The failure is reverted and the failed machine is connected back to the replica set.

5.3 Failures

The failures induced during the experiments involve shutting down the primary replica and observing how the secondary nodes behave immediately after the shutdown. The timestamps for when a failure is induced and fixed are also entered into the execution history. The following failures are induced in our experiment:

Graceful ACPI Shutdown: Sends an ACPI shutdown signal to the virtual machine to trigger a graceful shutdown.

Hard Poweroff: Simulates an abrupt power failure, which turns the VM off immediately.

Since we focus on crash-tolerant systems, we chose these failures they are supposed to be handled gracefully and correctly by the replica set. This will allow us to find concrete durability failures in the system, that should be possible to prevent within the failure assumptions the system

makes. Additionally, they are common system administration scenarios of needing to restart a machine or cutting the power. The recovery flows following these failures will need to re-elect the primary after the failure and, once the failed machine is back up, allow it to "catch up" on the writes which it has not seen.

5.4 Workload

The workload of each experiment consists of three types of operations:

Create Document: This operation adds a new document to the collection in the replica set. A new document is created with a value retrieved from a random number generator. Upon acknowledgement of the write from the replica set, its document ID is added to the list of IDs with acknowledged writes.

Read document: This operation queries the replica set for a document, given its key. The key for the query is chosen randomly from a list of IDs with acknowledged writes. This means, we expect that every query should return a document. Any missing documents are therefore as a result of an error in the replica set.

Update document: This operation modifies the value of an existing document. The key for the query is chosen randomly from a list of IDs with acknowledged writes and the value is generated using a random number generator.

The primary setting used to vary the workload of the experiment is *write probability*, which determines how often a write operation is issued to the replica set. There are two types of write operations in this experiment - *create document* and *update document*. The decision between which operation to perform is random with equal likelihood.

In addition, we set the Read Concern, Write Concern and Read Preference. This configuration varies between experiments, allowing us to explore the tradeoffs between these configurations and understand which configuration is best for a given scenario.

5.5 Execution Analysis

The execution history produced by each run of the experiment is analysed by Algorithm 2 to produce the following data:

Throughput: The number of successful operations performed during the entire run of the experiment.

Errors: The number of operations that did not receive a positive acknowledgement, either they got a negative acknowledgement, or they were time out without any acknowledgement.

Lost Writes: The number of write operations which were lost either temporarily or permanently.

In particular, the algorithm detects when the expected value of the document and value returned from a read are not the same or where the value is missing (denoted by a value of -1). This indicates that a write has either been lost, or that it has been committed but an acknowledgement was not received by the client. We make a distinction between these two cases by keeping track of the write operations that came back as errors to the client which allows us to handle the committed-but-not-acknowledged write and update the state of our analysis as needed.

Additionally, we also use the execution history to plot the number of different measures against the time of the experiment, bucketed using bins of 1 second.

Algorithm 2 Algorithm used for evaluating the execution history

$H \leftarrow$ execution history
$t_{fail} \leftarrow$ timestamp of failure induction
$t_{fix} \leftarrow$ timestamp of failure repair
$D \leftarrow \{\}$ {Lookup of every document's current value}
$D_e \leftarrow \{\}$ {Lookup of every write that errored for a document}
$E \leftarrow \{\}$ {Set of all errors produced}
$M \leftarrow \{\}$ {Set of missing writes}
$U \leftarrow \{\}$ {Set of unacknowledged writes}
$o \leftarrow 0$ {Number of successful operations}
$s \leftarrow$ Normal {Current state of the history (Normal, Failure or Recovery)}
for all $h \in H$ **do**
  if $h.time \leq t_{fail}$ **then**
    $s \leftarrow$ Normal
  else if $t_{fail} < h.time \leq t_{fix}$ **then**
    $s \leftarrow$ Failure
  else
    $s \leftarrow$ Recovery
  end if
  if $h.op =$ Write **or** $h.op =$ Update **then**
    $o \leftarrow o + 1$
    $D[h.id] \leftarrow h.val$
  else if $h.op =$ Read **then**
    $o \leftarrow o + 1$
    if $h.val \neq D[h.id]$ **then**
      if $h.id \in D_e$ **and** $h.val \in D_e[h.id]$ **then**
        $U \leftarrow U \cup \{(h.id, h.val, s)\}$
        $D[h.id] \leftarrow h.val$
        delete $h.val$ from $D_e[h.id]$
      else
        $M \leftarrow M \cup \{(h.id, h.val, state)\}$
      end if
    end if
  else if $h.op =$ Induce **then**
    $s \leftarrow$ Failure
  else if $h.op =$ Recover **then**
    $s \leftarrow$ Recovery
  else if $h.op =$ Error **then**
    $E \leftarrow E \cup \{(h.err\_op, h.id, h.val, s)\}$
    if $h.err\_op =$ Write **or** $h.err\_op =$ Read **then**
      $D_e[h.id] \leftarrow D_e[h.id] \cup \{h.val\}$
    end if
  end if
end for

Chapter 6

Results

In this chapter, we show that our experiment design detects expected write losses in MongoDB 4.0 and 3.6-rc0, while also finding (known) durability and performance bugs in 3.6-rc0, which have been fixed in 4.0. We present the results of our experiments and give quantitative metrics for the frequency and quantity of non-durable writes, along with performance metrics for varying MongoDB configurations. We proceed to evaluate the results of our experiments and conclude with a discussion of their limitations.

6.1 Environment

The environment used for experiments consisted of 3 virtual machines running on a single host. The host has a 2.2Ghz Intel Core i7 processor, with 16GB RAM and a 256GB SSD. It runs Ubuntu 16.04, and uses VirtualBox 5.1.28 to host the virtual machines.

The three virtual machines were each dedicated 2GB RAM and 20GB of pre-allocated disk space, running Ubuntu Server 16.04.5. They are networked together via a Host-Only network that allows the host and the VMs to communicate with each other at very stable sub-millisecond latencies. Each virtual machine was configured to honour all disk flushes as to ensure parity between which writes MongoDB expected to be persisted and which writes are indeed persisted on disk. The three virtual machines were all installed with the same version of MongoDB and configured to run in a single replica set.

Two of these environments were generated for two versions of MongoDB. The MongoDB versions used were 4.0 and 3.6-rc0. MongoDB 4.0 is the current LTS version of MongoDB and will act as our control measurement, while MongoDB 3.6-rc0 is a version with known durability issues.

The reason for testing both is to see whether our experiment can find durability failures and whether the current version of MongoDB has any such failures.

Each failure was induced via a call to VirtualBox's `VBoxManage`.

All experiments were run from the host, with the host being the sole issuer of queries and write operations to the replica set. The experiment harness is written in .NET Core, using the MongoDB Driver 2.7.0.

We achieve parallel execution by having multiple threads perform these operations, each tracking their own documents and values.

6.2 Write Durability

We ran each experiment for 5 minutes, including the failure at the 100 second point and fixing the failure at the 200 second point.

Tables 6.1-6.8 present the summative results of these experiments. Overall, we see that MongoDB 4.0 performs as predicted by our theoretical analysis. Any write concern that ensures acknowledgement only on the primary does not guarantee durability if the primary fails. This applies to the Primary and Journaled write concerns.

On the other hand, MongoDB 3.6-rc0 is shown to have durability failures even when majority of the replica set have acknowledged the operation. This implies that MongoDB 3.6-rc0 has software flaws in the way it measures acknowledgement from secondary replicas and how it handles recovery and elections after a failure. It should be noted that these durability failures were only seen in experiments utilising a very heavy write load onto the replica set, which means that these durability failures may also be a result of flawed persistence mechanisms. We should add that such write-heavy workloads are unlikely to appear in real world applications, which tend to involve more read operations from the database.

6.2.1 Primary Preferred Read Preference

We also make a note of Table 6.4, where the Primary Preferred, Majority, Majority case resulted in 9 lost writes. Upon inspection of the execution history, we find that *all* write losses in this

```
write x = 1
...
read  x -> x = 1
...
read  x -> x = -1
...
read  x -> x = 1
...
```

Figure 6.1: Example of a transient loss observed with Primary Preferred read preference

experiment were transient. Specifically, they followed the pattern seen in Figure 6.1. This was the only experiment that exhibited transient losses, all other experiments either showed no write loss, or permanent write loss - where a lost write was never seen after being detected as lost.

This type of behaviour is expected, given that this read preference will force the MongoDB client to read from the secondary if the primary is unavailable, instead of waiting for the primary's response. If the client then reads from a secondary that had not yet seen the operation, the data returned from the query will not include it - leading to a loss. However, once a new primary is elected, the client will query only the primary, hence seeing the write again. Fundamentally, this shows that a Primary Preferred read preference removes all guarantees about the data queried from the replica set during a failure, *unless* the write has propagated to all secondaries. Given these results, we recommend that a configuration with a Primary Preferred read preference should always have the write concern configured to "all" - that is, all secondary replicas must acknowledge the write before it is acknowledged to the client.

6.2.2 Journaled vs Primary Write Concern

We observe that for non-majority-majority experiments, using the Journaled write concern does not always reduce the number of lost writes. The cases which see a reduction are only the write-heavy workloads (write probability 0.7). This implies that Journaled write concern reduces the chances of a write loss as a result of not persisting the write.

However we see that Journaled write concern experiences increased write-loss compared to experiments with a Primary write concern in a standard write-load workload (write probability 0.3). This is a strong indication that these write losses occur as a result of a rollback due to a reelection, as the operation itself is more likely to be persisted to disk.

As such, we conclude that Journaled Write Concern acts as a tradeoff - it reduces the likelihood of persistence failures but increases the time to send the operation to the secondaries, hence increasing the probability and magnitude of a rollback failure. In contrast, the Primary write concern attempts to minimise communication delay at the cost of not providing guarantees about the persistence of an operation.

6.2.3 Write Loss distribution

Having established the principal causes for write loss, and finding examples of these during the experiments, we now turn our attention to determining *when* write loss occurs. Figures 6.2 and 6.3 plot the number of lost writes against the time within the experiment when the write was submitted.

From Figures 6.2 & 6.3 we see that only writes which are acknowledged immediately before or while a failure is being induced are prone to loss. This is intuitive as those operations are the ones in the process of being persisted and being communicated to secondary nodes.

We should observe the relationship between the write loss distribution and error distribution (Figures 6.4, 6.5) of the experiments. Specifically, we note that there are few to no errors present at the time of the failure. This is a direct result of the MongoDB client waiting for responses. The timeout for a response was configured to be 5 seconds. As such, errors spike at around 105 seconds and continue to persist for approximately 20 seconds. This represents the influx of timeout errors from the MongoDB client during the election process for a new primary.

We note that during all experiments, all write losses were found on *unique* documents - no document suffered multiple write losses during a single run of the experiment.

Furthermore, we see a unique pattern of errors in Figure 6.5, where the main peak of errors takes place immediately after the failure is induced. However smaller peaks of errors then tail-off until the failure is fixed and the failed node recovers. The reason for this error can be found by looking at the pattern of write operations for this experiment in Figures 6.6 & 6.7.

We notice that there are *no successful write operations* during the entire failure phase of the experiment. This is even more peculiar considering that *some* read operations succeeded during this phase, returning correct values the vast majority of the time. This indicates that MongoDB

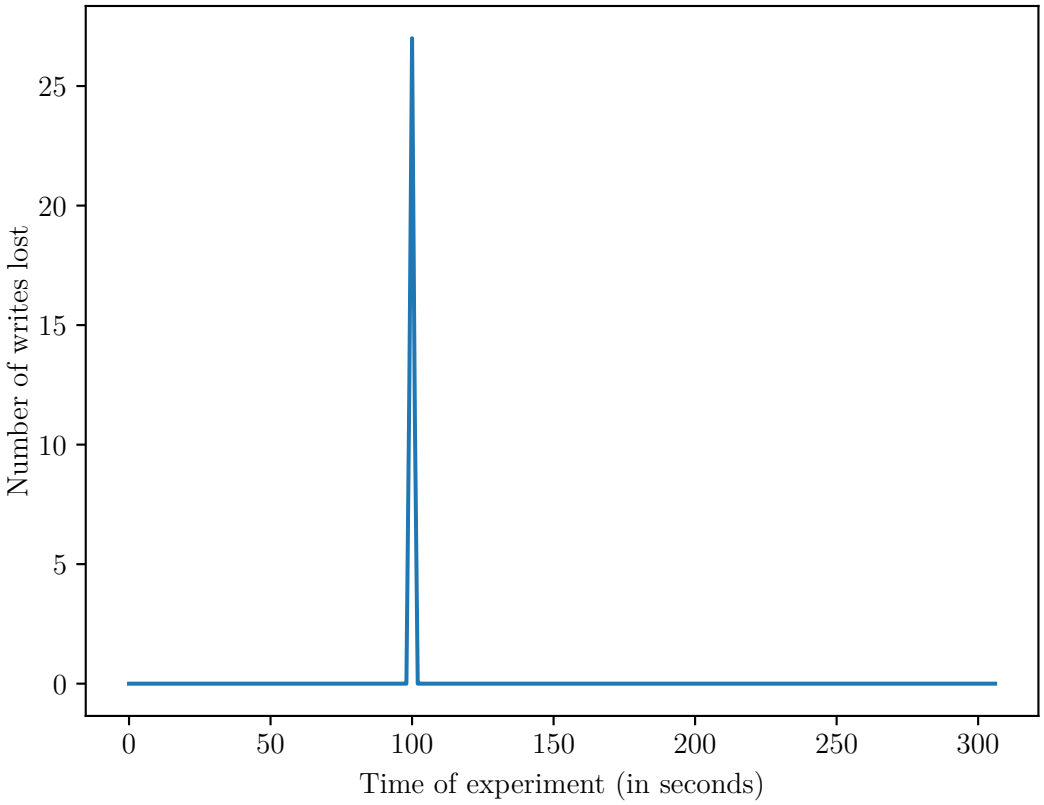


Figure 6.2: Distribution of write loss for every second of experiment 3 in Table 6.6

3.6-rc0 has a bug in its recovery flows which causes the new elected primary to ignore all write operations, but still accept some read operations. Considering that this behaviour was not seen in any other experiment - either in 3.6-rc0 or 4.0, we can be confident that the bug was fixed in the 4.0 version of MongoDB.

Using these results, we can now explain the tailing-off of errors in Figure 6.5. Soon after the failure is induced, all operations that were submitted to the replica set time out and return as a massive spike in errors. After this initial surge, a new set of write operations and queries is sent to the replica set. Some of the queries return successfuly while all write operations time out. This results in a reduction of total errors reported. The cycle keeps repeating, each time all write operations timing out and only some read operations doing the same. When the failure is fixed and the failed node joins the replica set, all operations start returning successfully, hence dropping the number of errors back to 0.

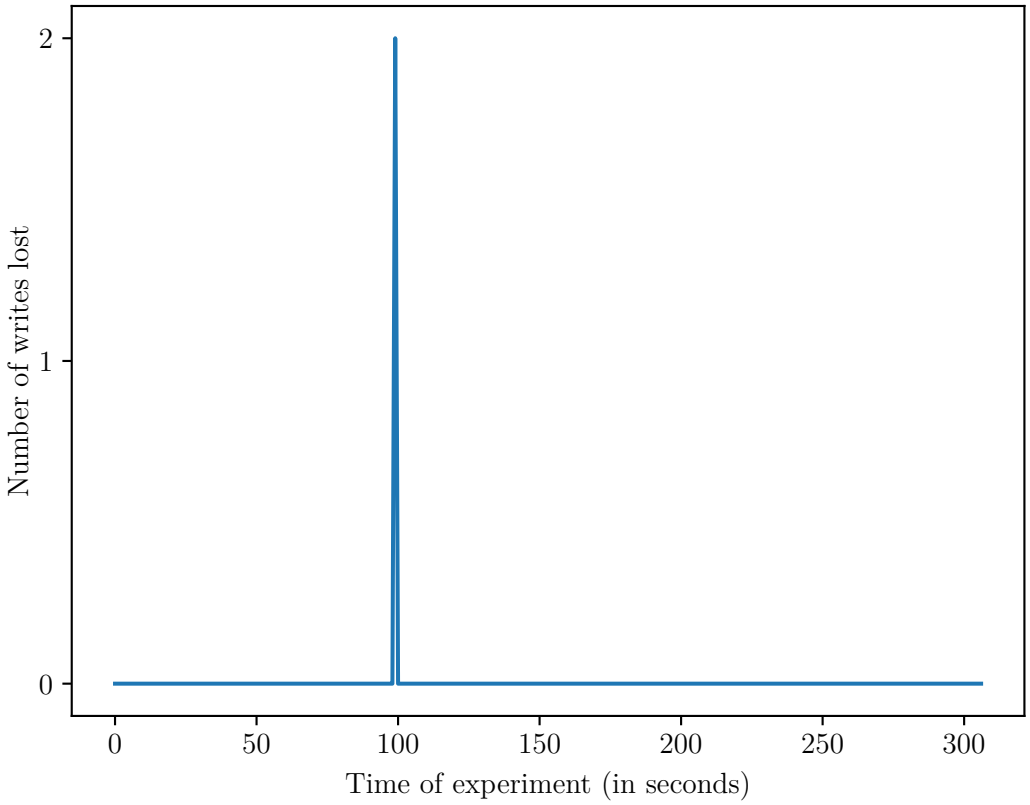


Figure 6.3: Distribution of write loss for every second of experiment 3 in Table 6.8

6.2.4 Performance

Having established the durability properties of the MongoDB configurations used in these experiments, we will now address performance as the main tradeoff to the durability guarantees these configurations provide.

6.2.4.1 Write Concerns

Figures 6.8, 6.9 & 6.10 show the average latencies of write operations for the first three experiments performed in Table 6.4 . We will consider any operation with latency of 2000ms or more as timed out. We found that read latencies were not affected by write concerns and so we did not include them in this analysis.

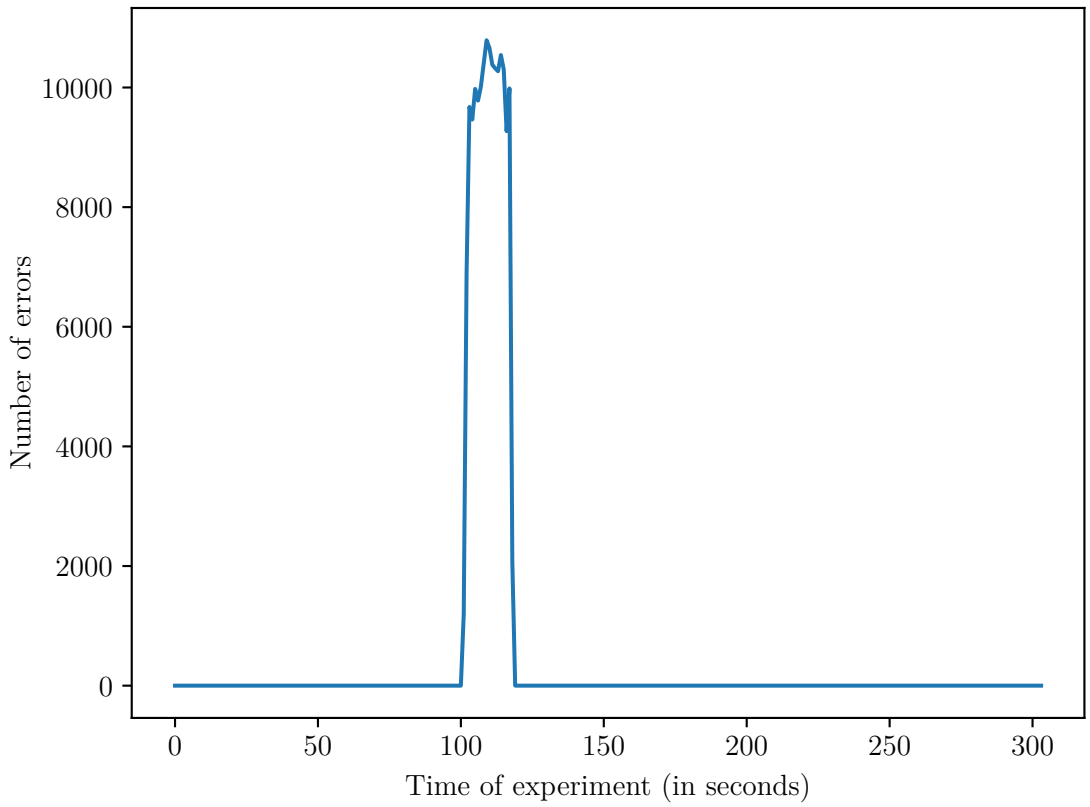


Figure 6.4: Distribution of errors for every second of experiment 3 in Table 6.6

We can immediately observe that as we increase the strength of write concern (where Primary is the weakest and Majority is strongest), we also increase the average latency of the operations performed with that write concern. Recall from Section 3.4 that as the strength the write concern increases, MongoDB has to perform more actions prior to acknowledging the operation. This is exactly the reason for the increasing average latencies.

However we make another observation in these results. Figures 6.9 and 6.10 show a noticeable but temporary increase in latency soon after the 200 second mark - the recovery phase of the experiment. In our experiments, the recovery phase is signified by turning on the failed replica and letting it rejoin the replica set as a secondary, which would also involve performing synchronisation and rollbacks to be in sync with the new primary. It is likely that this extra load on the primary replica reduces the resources available for it to respond to queries and update its copy of the data. Another possibility is that synchronisation may require that certain documents

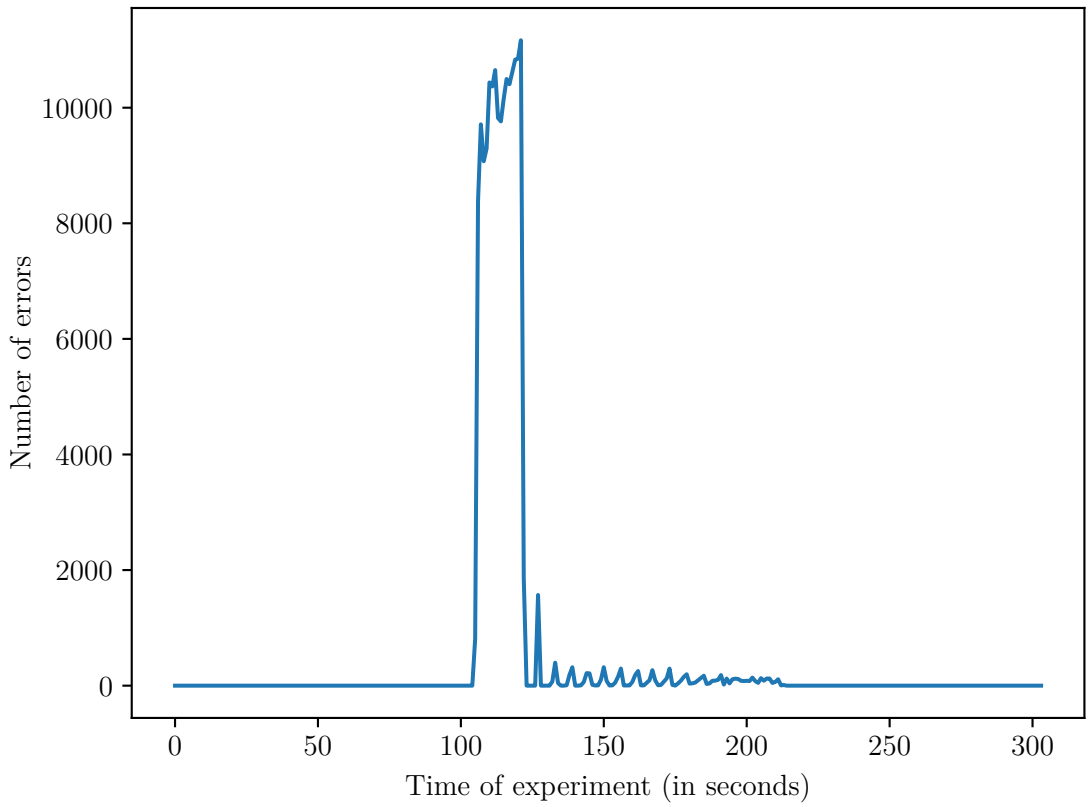


Figure 6.5: Distribution of errors for every second of experiment 3 in Table 6.8

be locked while they're being synchronised, causing a delay to acknowledging any operations on those documents until after the primary has confirmed that they've been synchronised with the secondary.

6.2.4.2 Read Preferences

Another dimension of our experiments concerns the effect of read preference on how the client reacts to failures in the replica set. Figures 6.11 & 6.12 show the average latencies of read operations for two equivalent experiments, only varying the read preference between Primary and Primary Preferred. During this analysis, we will consider any operations that took 2000ms or longer as timed out.

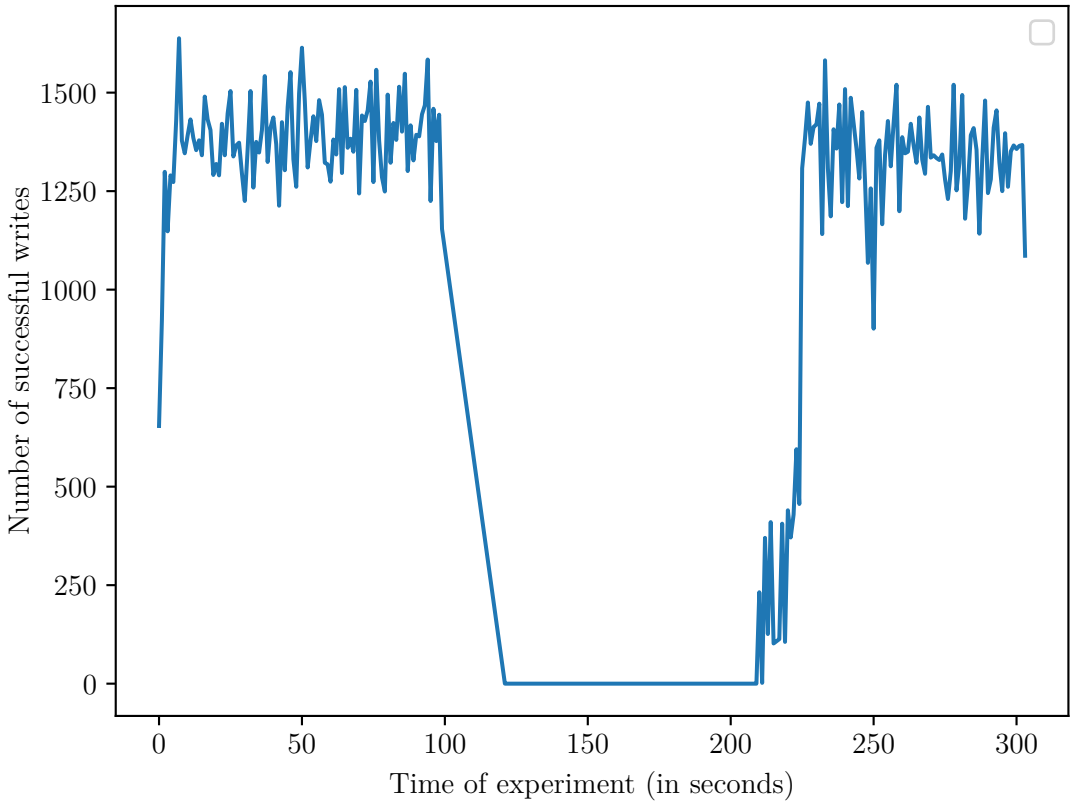


Figure 6.6: Distribution of successful write operations of experiment 3 in Table 6.8

We immediately observe that Primary Preferred and Primary reads behave differently under a failure. Primary reads exhibit the behaviour indicative of timeouts for approximately 20 seconds after a failure is induced and return to a relatively stable state afterwards.

On the other hand, Primary Preferred reads tend to return very quickly after a failure for only 10 seconds after it is induced, returning back to normal operation soon after. The small spike immediately at the point of failure implies that the client was trying to query from the Primary while it was failing, resulting in a small number of timeouts. This implies that the client was able to identify the primary replica as failing and start querying secondaries as per its read preference. The reasoning behind the significant drop in latency is unknown but may be attributed to the significantly reduced load on the replica set, as no write operations are being applied.

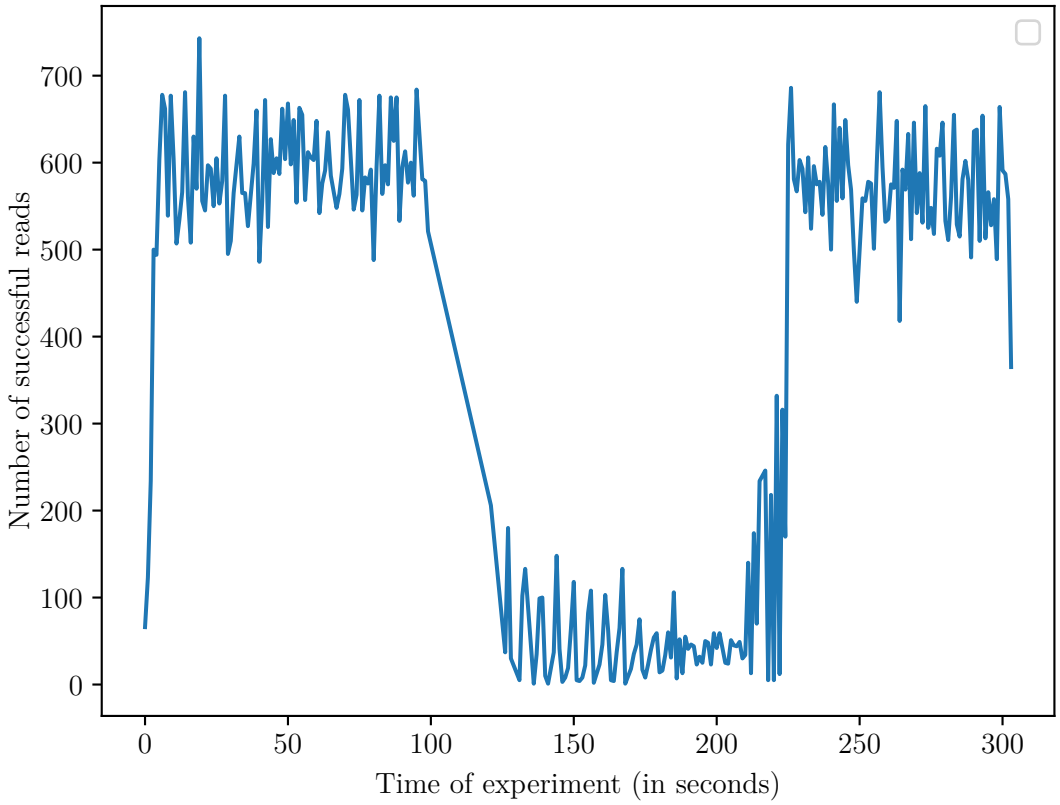


Figure 6.7: Distribution of successful read operations of experiment 3 in Table 6.8

The more peculiar result is found around the time the failed node rejoins the replica set, between 180 seconds and 230 seconds into the experiment. During this time frame, the execution exhibits a very unstable latency pattern, with latencies jumping between 100ms at the lowest and 900ms at the highest. We could not determine the source of this anomaly, but postulate that it may be affected by the process in charge of adding a new secondary to the replica set.

6.3 Discussion

Much of the results we obtained were as we expected. Nonetheless, we recognise that our method had limitations. These limitations spanned across our methodology and resources. The main limitations are as follows:

(1) All failures were induced *only* on the primary.

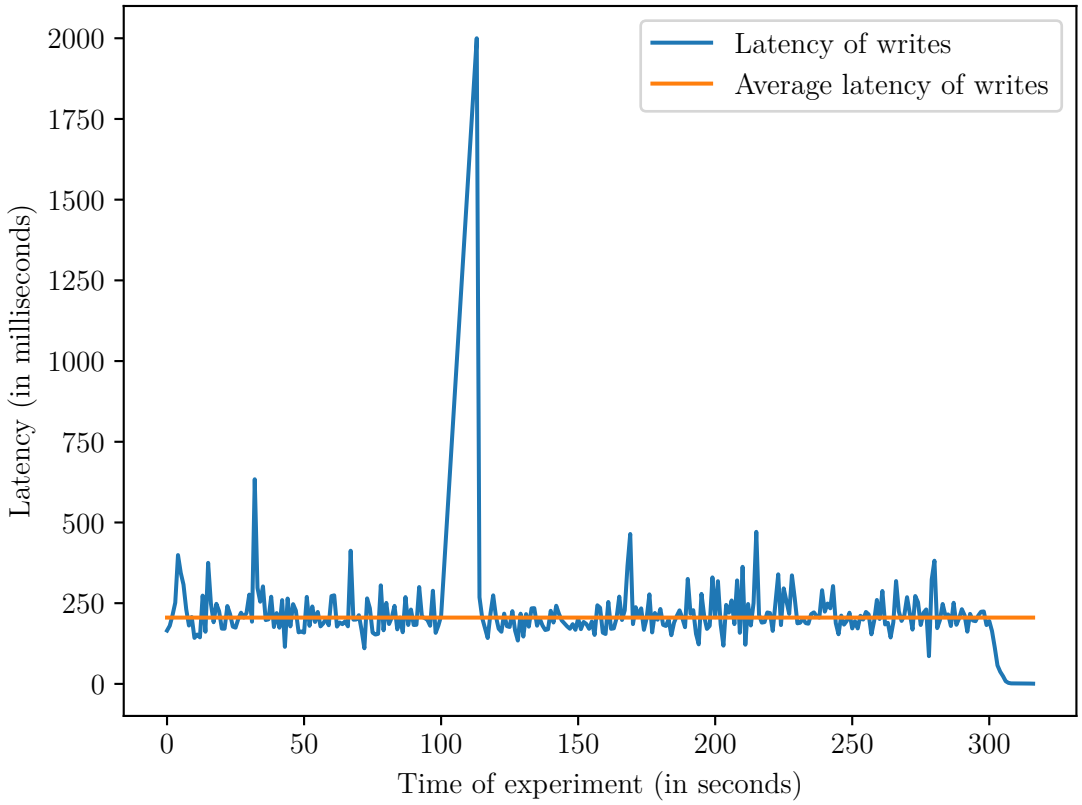


Figure 6.8: Latency of write operations for every second of experiment 1 in Table 6.4

(2) Only one failure was induced per experiment.

(3) Timeout errors created uncertainty about document state.

(4) Uniformly random document selections.

(5) Analysis was based only on the execution history, not the internal replica logs.

6.3.1 Failures exclusively on the Primary

The decision to induce failures exclusively on the primary arose from recognising that the primary replica is the arbiter of all write operations in a replica set. As such, by making the primary fail, we ensure that the replica set *must* go through some kind of recovery flow. Unfortunately, this means our methodology did not account for any behaviours that would arise from when a secondary replica (or multiple secondaries) fail and how this would impact the durability and performance of the replica set.

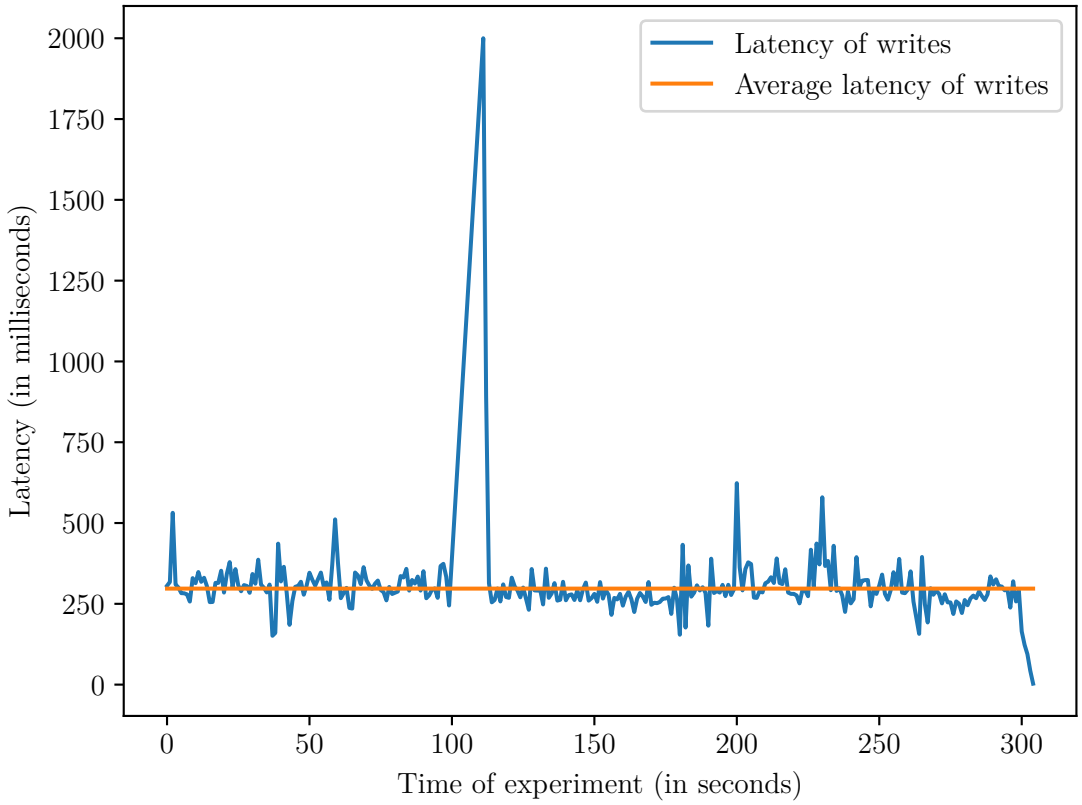


Figure 6.9: Latency of write operations for every second of experiment 2 in Table 6.4

6.3.2 One failure per experiment

Our experiment was designed to be analysed in three segments - normal operation, failure and recovery. In order to ensure a distinguishable cutoff point for each segment, we only induced a single failure and fixed it after a specified time period. This fundamentally limited what kinds of errors and scenarios we could induce as part of the experiment. Specifically, we did not investigate a case of repeated or ongoing failures of a node, such as repeated restarts due to misconfiguration. This could stress how elections handle an unstable, but functional, primary.

6.3.3 Timeout Errors

Having inspected the results, we found a large number of errors being reported by the analysis script. The majority of these errors are simply timeouts and are mostly present during the

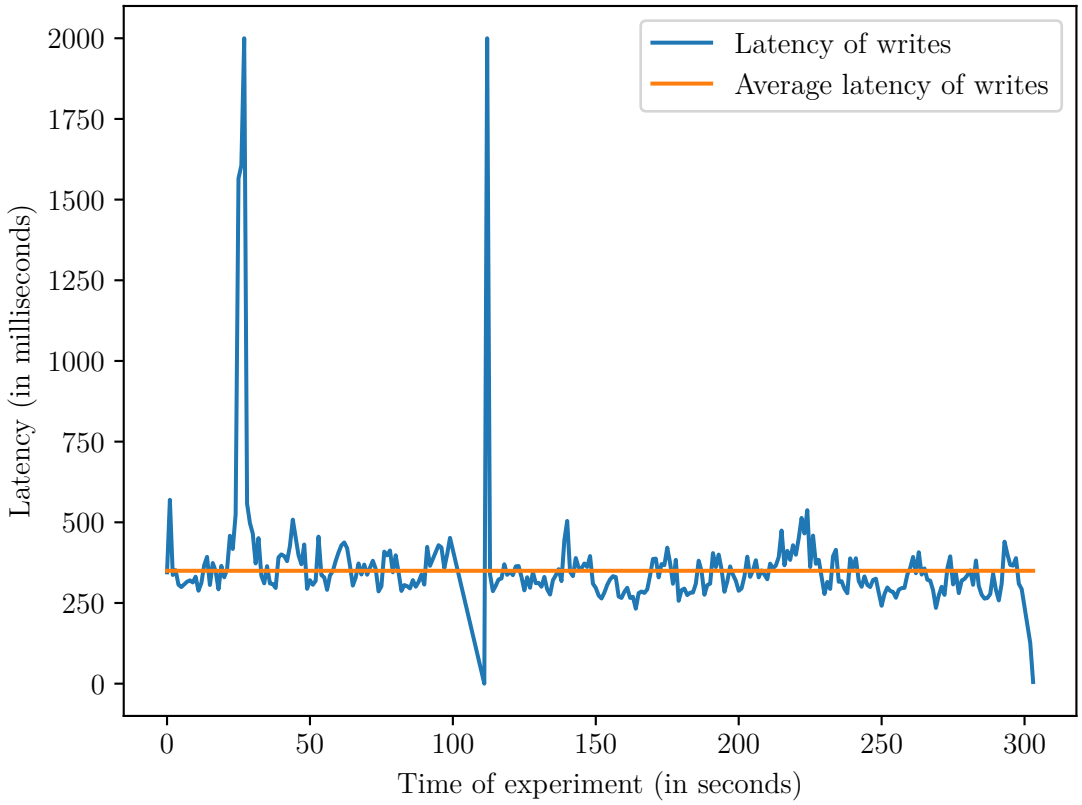


Figure 6.10: Latency of write operations for every second of experiment 3 in Table 6.4

time that the failure is induced in the primary. These timeouts are a result of the election procedure that must take place when the primary crashes in order for MongoDB to select a new primary replica. As there is no arbiter for write operations present during this time, no write operations can be acknowledged. If the read preference is set to *Primary*, no read operations can be acknowledged either. This causes those operations to wait until the primary is available, which may take longer than the configured timeout, resulting in an error.

We must also address that our experiment harness only queries and modifies documents that have their ID's in the list of acknowledged writes. This has the observable consequence that an update operation could be committed on the replica set but the acknowledgement not be received by the client. We can therefore reason that *Create Document* operations may suffer from the same issue. This kind of ambiguity can easily mask durability failures as it prevents us from having a complete view of all documents in the replica set, in particular preventing us

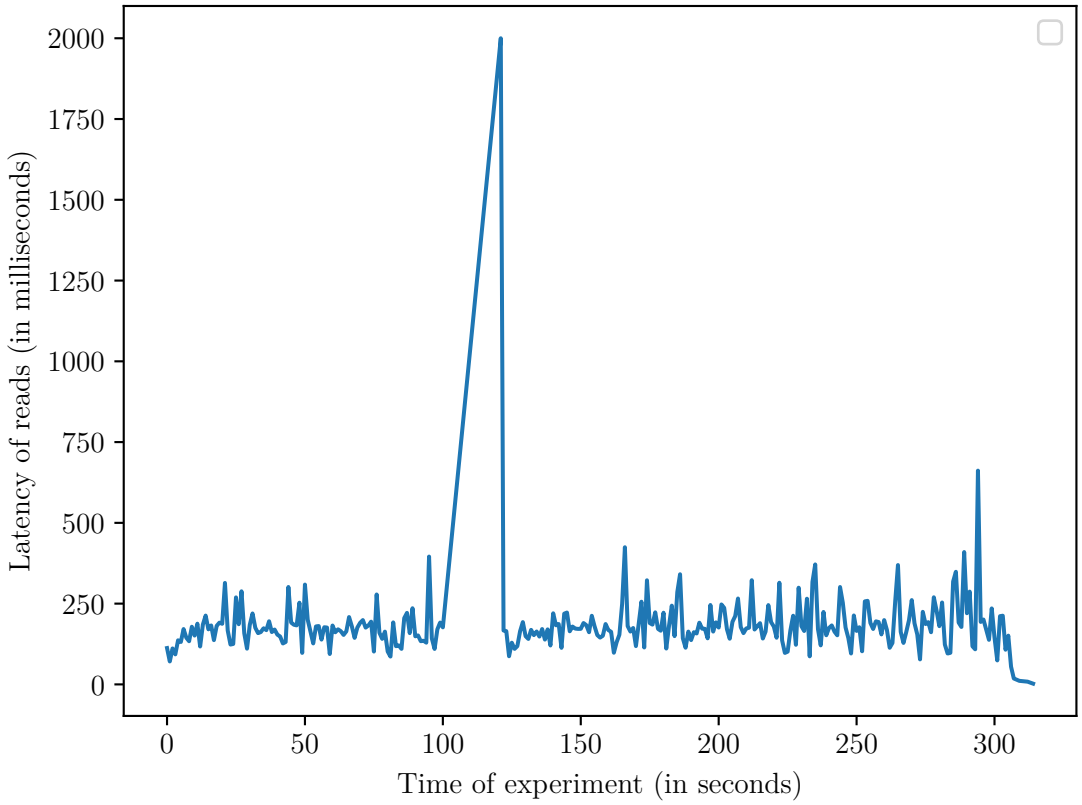


Figure 6.11: Latency of read operations for every second of experiment 1 in Table 6.3

from querying documents that have been committed to the replica set but not entered into the list of acknowledged writes because of a timeout. This is especially problematic since durability failures tend to happen *during* a failure - at the same time as the timeouts.

6.3.4 Uniformly random document selection

It is common knowledge that data is not accessed in a uniform manner. Our experiment used a simple random number generator to decide which document to read or update. As such, our workload did not represent a realistic scenario of operations performed on the database. For example, the uniform selection ensures minimal contention for documents in the queries at the database-level, meaning that we are not observing how MongoDB handles high contention during failures.

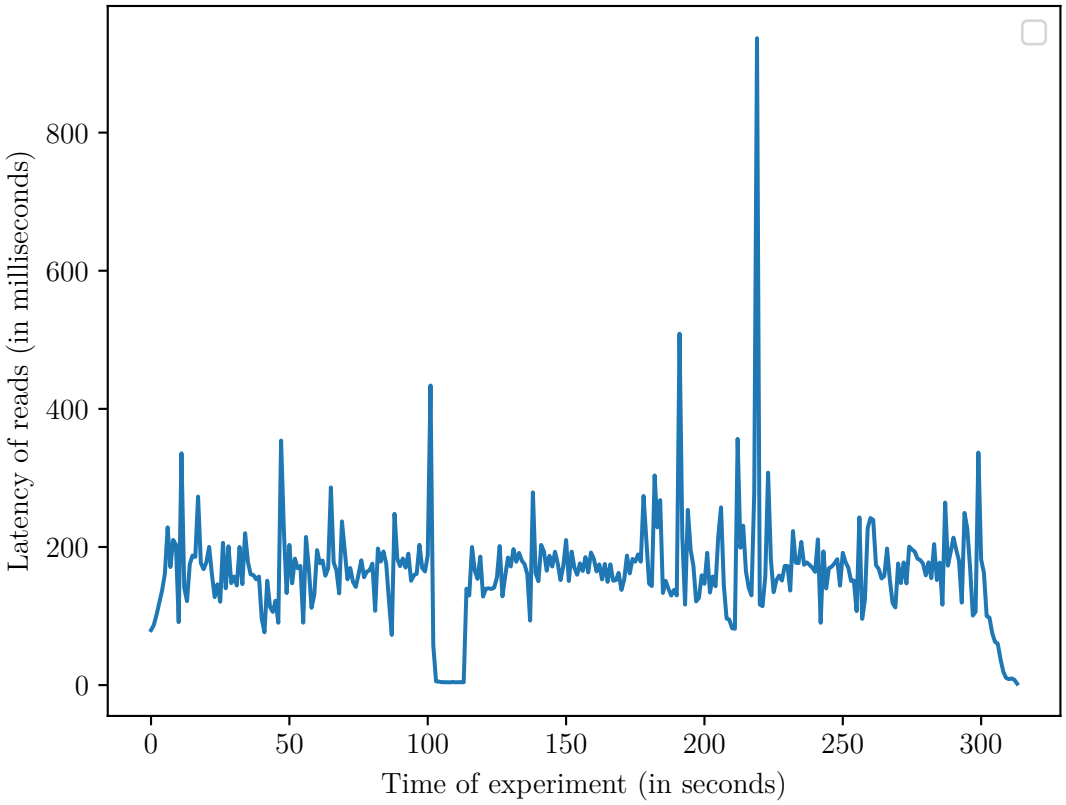


Figure 6.12: Latency of read operations for every second of experiment 4 in Table 6.3

6.3.5 Execution history, not replica logs

Our methodology concentrated on detecting write losses by looking at the execution history. We did not inspect the native MongoDB logs as part of our study because we concentrated on finding immediately visible durability failures in MongoDB, from the view of the a user/client of the database. This limitation meant we could not pick up certain durability failures, such as write rollbacks on secondary nodes.

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	763824	1177	400
Primary	Local	Journaled	828721	28760	331
Primary	Majority	Majority	798183	31212	0
Primary Preferred	Local	Primary	846215	8142	141
Primary Preferred	Local	Journaled	840277	9181	421
Primary Preferred	Majority	Majority	765270	7499	0

Table 6.1: Experiment results MongoDB 4.0 with *Shutdown* failure on 0.3 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	693570	101310	2223
Primary	Local	Journaled	647880	89464	1733
Primary	Majority	Majority	641685	86279	0
Primary Preferred	Local	Primary	782852	66952	1800
Primary Preferred	Local	Journaled	705703	50119	180
Primary Preferred	Majority	Majority	646299	51933	0

Table 6.2: Experiment results MongoDB 4.0 with *Shutdown* failure on 0.7 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	738601	2135	261
Primary	Local	Journaled	759628	25906	2834
Primary	Majority	Majority	757109	21838	0
Primary Preferred	Local	Primary	805828	1037	706
Primary Preferred	Local	Journaled	770923	11868	171
Primary Preferred	Majority	Majority	777713	18867	0

Table 6.3: Experiment results MongoDB 4.0 with *Poweroff* failure on 0.3 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	767995	59474	1763
Primary	Local	Journaled	649970	49109	1217
Primary	Majority	Majority	566651	50420	0
Primary Preferred	Local	Primary	712128	6144	4540
Primary Preferred	Local	Journaled	730408	74884	238
Primary Preferred	Majority	Majority	534430	33748	9

Table 6.4: Experiment results MongoDB 4.0 with *Poweroff* failure on 0.7 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	605527	666	70
Primary	Local	Journaled	613437	33398	352
Primary	Majority	Majority	628248	35591	0
Primary Preferred	Local	Primary	672897	9760	393
Primary Preferred	Local	Journaled	664797	8093	581
Primary Preferred	Majority	Majority	688463	10962	0

Table 6.5: Experiment results MongoDB 3.6-rc0 with *Shutdown* failure on 0.3 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	569716	85219	210
Primary	Local	Journaled	546575	953	87
Primary	Majority	Majority	412634	62548	40
Primary Preferred	Local	Primary	595135	44845	1097
Primary Preferred	Local	Journaled	549167	51327	387
Primary Preferred	Majority	Majority	536845	74019	0

Table 6.6: Experiment results MongoDB 3.6-rc0 with *Shutdown* failure on 0.7 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	663695	17754	1319
Primary	Local	Journaled	663009	15741	1815
Primary	Majority	Majority	628013	15211	0
Primary Preferred	Local	Primary	706243	5633	1189
Primary Preferred	Local	Journaled	676734	709	335
Primary Preferred	Majority	Majority	676025	5204	0

Table 6.7: Experiment results MongoDB 3.6-rc0 with *Poweroff* failure on 0.3 write probability

Read Preference	Read Concern	Write Concern	Throughput	Errors	Lost Writes
Primary	Local	Primary	472444	110326	5589
Primary	Local	Journaled	597266	65285	457
Primary	Majority	Majority	338798	126149	2
Primary Preferred	Local	Primary	581263	32181	255
Primary Preferred	Local	Journaled	608816	36625	2782
Primary Preferred	Majority	Majority	571397	34555	0

Table 6.8: Experiment results MongoDB 3.6-rc0 with *Poweroff* failure on 0.7 write probability

Part 3

Estimating Durability

Chapter 7

Durability Theory

In this chapter we use the results from Part 2 and background information of how MongoDB persists write operations to equip the reader with the tools to reason about varying levels of durability in a flexible manner by developing a novel theoretical model of when write durability occurs. This model will then allow the reader to understand the theoretical foundation on estimating when a write becomes durable using simple measurements present with each individual write operation.

We will begin with a discussion of the circumstances required to make a write durable and proceed to loosen those restrictions to introduce a more flexible form of write durability. We then compare this model to MongoDB's write concern and show that MongoDB's journaled write concern provides weaker guarantees compared to this model, concluding by showing how this model can be used to estimate when writes become durable.

7.1 Introduction

Upon closer inspection of the execution histories which contained lost writes, we notice that the writes that tended to be confirmed *closer* to the time of failure were more likely to be lost. This leads us to believe that these writes were lost because no secondaries were informed of the operation prior to the failure.

7.2 Total Durability

In order to be absolutely certain that a write will not be lost even if *all* nodes crash, we must allow the write to be propagated to and persisted by every node in a replica set R.

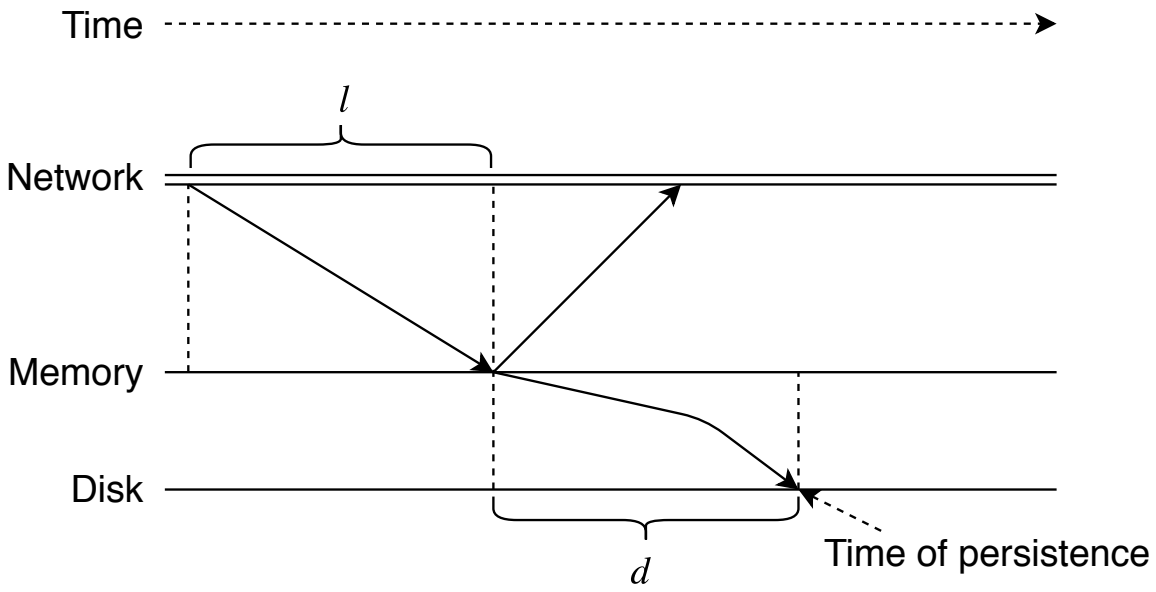


Figure 7.1: A time-space diagram of a write being persisted onto a single node

Let us define t as the time a write operation was submitted and t_a as the time at which it has been persisted to all nodes. The time it takes for a write to be persisted is comprised of the time it takes for the node to receive the write and the time it takes for the write to be written into memory and flushed to disk. We will define these as l and d respectively. As such, for an individual node r, the time at which it persists the write to disk is $l_r + d_r$. An illustration of this process is seen in Figure 7.1. Assuming that all replicas will retrieve the write in parallel, we can say:

$$t_a = \max_{r \in R}(l_r + d_r) + t \tag{7.1}$$

7.3 K-Durability

We now make a vital observation - if total durability is defined when *all* replicas persist a write, we can define a measure for when k replicas persist a write.

We define a write as *k-durable* if it has been persisted onto k replicas. A k-durable write can sustain $k-1$ crashes and still be visible to the replica set after an election. Using this definition, and equation (7.1) we derive the time at which a write becomes k-durable as:

$$t_k = \max_{r \in R_k}(l_r + d_r) + t \tag{7.2}$$

Where R_k is the set comprised of the first k nodes that have persisted the write.

We can now see that a totally durable write is $t_a = t_k$ where $k = |R|$. Intuitively, $t_k \leq t_a \ \forall k$ as we simply ignore the $|R| - k$ worst durations. This means that it is quicker for a write to become k-durable than totally durable.

This model can be used to predict the durability of any write as a function of the duration since the write was acknowledged. Specifically, if we can measure the probability distributions of l and d, we estimate how many nodes have persisted the write at a given point in time, hence estimating the write's durability level.

7.4 K-Durability & Write Concern

Recall from Section 3.4 that MongoDB's write concern configuration can specify how many replicas must acknowledge the write before the primary acknowledges it to the client. The write concern configuration can also specify whether a write is to be journaled, which should ensure that the operation is persisted before being acknowledged. Journaled write concern k should then be a good approximation of k-durability - the operation is expected to have reached k nodes and been persisted to the journal on all k nodes prior to acknowledgement.

However MongoDB buffers write operations before persisting them as a batch. The buffering behaviour compromises all durability assumptions as we *cannot* guarantee that a write operation has been persisted when we receive an acknowledgement.

7.5 Estimating 1-durability

Recall the definition of k-durability from equation 7.2. Substituting $k = 1$, we get the following:

$$t_1 = \max_{r \in R_1}(l_r + d_r) + t$$

Now we observe that R_1 is a set consisting only of the first node that persists the operation. Recall that the primary replica will add the operation to its Oplog only *after* persisting it.

This means the secondary nodes will not even be aware of an operation until after the primary persists it. As such, the primary *must* be the first node to persist the operation. Hence $R_1 = \{primary\}$ and our equation for t_1 can be simplified as follows:

$$t_1 = l_{primary} + d_{primary} + t \tag{7.3}$$

From here onwards, the *primary* subscript will be left out for brevity.

As a result, to estimate 1-durability of an operation, it is sufficient to estimate when the primary persists the operation.

Recall that journaled write concern is supposed to act as a method for ensuring a write is persisted before acknowledgement. While this is not MongoDB's exact behaviour, as MongoDB buffers journaled operations before flushing them to disk, it is a good approximation. We define the time a journaled write gets acknowledged as:

$$j = 2l + d_{est} + t$$

We notice this is very similar to our definition of t_1 in equation 7.3. Therefore we define an estimate of 1-durability as:

$$t_{1\_est} = j - l \tag{7.4}$$

We will now show that t_{1_est} is a valid approximation of t_1 using the measurements of ping latency l and latency j of a journaled write:

$$\begin{aligned} t_{1\_est} &= j - l \\ &= l + d_{est} + t \\ &\approx l + d + t \\ &\approx t_1 \end{aligned}$$

As such we prove that we can use equation 7.4 to estimate 1-durability of an operation.

Chapter 8

Results

In this chapter, we present an empirical exploration of time-till-1-durability in MongoDB based on the theory developed in Chapter 7 and proceed to evaluate the results in light of how MongoDB handles write operations.

8.1 Environment

The experiments were performed on the same harness as the one developed in Chapter 6, using the *Local* read concern and *Primary* read preference. The harness was configured to *not* induce a failure and perform only write operations (using write probability 1).

8.2 Estimating 1-durability

We ran two experiments, each for 5 minutes. One with write concern *w:1* and one with *journaled*. The execution histories were processed to count the latencies of each operation. The raw results of this collection can be seen in Figure 8.1.

Given that there is a significant difference between the patterns of *w:1* and *journaled* histories, we use the journaled history and apply equation (7.4) on every operation to derive the time a write becomes 1-durable. Given that VirtualBox networking is very stable, we subtract the average ping latency to the primary from every operation to determine the time that operation became 1-durable. A comparison between 1-durability and *w:1* write acknowledgement can be found in Figure 8.2.

To further clarify the differences between *w:1* and 1-durability, we present a chart of the difference between the two cumulative plots in Figure 8.3.

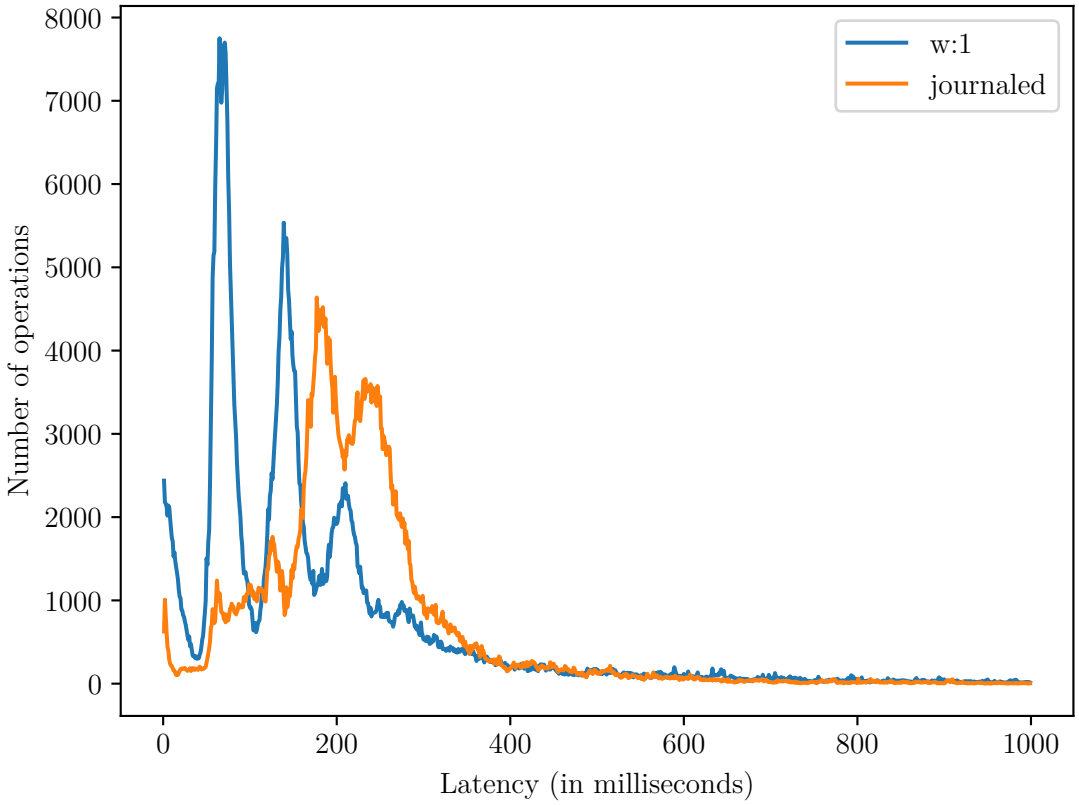


Figure 8.1: A frequency graph of the number of write operations acknowledged at latencies of 1-1000ms.

8.3 Discussion

We observe a distinct difference in the pattern of plots in Figure 8.1. In particular, w:1 is shown to have multiple peaks on the latencies of which the operations tend to be acknowledged. This is likely to be due to MongoDB's need to persist the operations with w:1 write concern separately from acknowledging them, which results in certain operations being blocked due to needing to wait until the batch is persisted, before the operation can be applied to the data and acknowledged. It is likely that due to this "waiting" that peaks in latency tend to form, as MongoDB forces operations to wait while it attempts to persist a batch of older operations. This behaviour is not seen in the journaled write concern because every write is offloaded onto the WiredTiger journal, which is then solely in charge of deciding when to persist those operations,

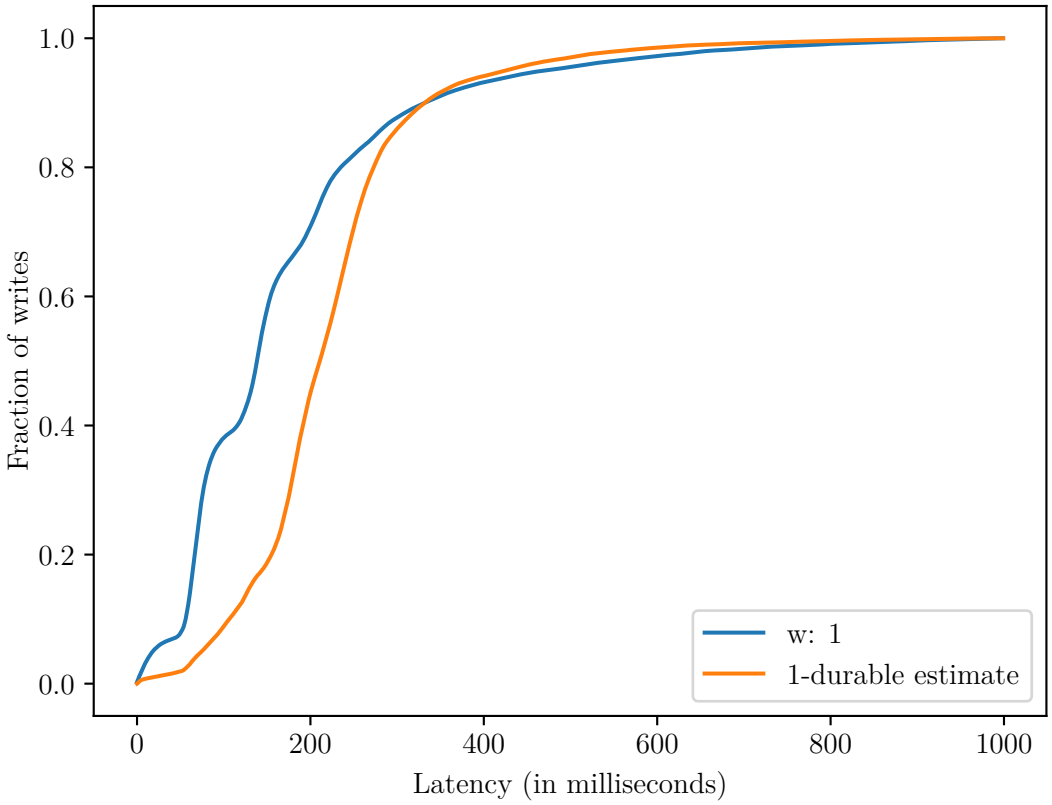


Figure 8.2: A cumulative frequency graph of the writes which become acknowledged and 1-durable at latencies 1-1000ms.

separate from MongoDB. We should emphasize that this is conjecture and more research is required to fully understand this behaviour.

Additionally, we see that the peaks and troughs of w:1 and journaled plots respectively align from 170ms to 300ms. We do not know why these alignments occur, but postulate once again that the WritedTiger journal may be playing a role in this behaviour. This further highlights the need for further research, specifically into the intrinsics of WiredTiger and its interaction with MongoDB.

We also note that the journaled execution has a higher average latency, clumping around 200ms, which is expected given the additional work required by the journaling process before operation can be acknowledged. However, journaled execution also seems more stable than its w:1

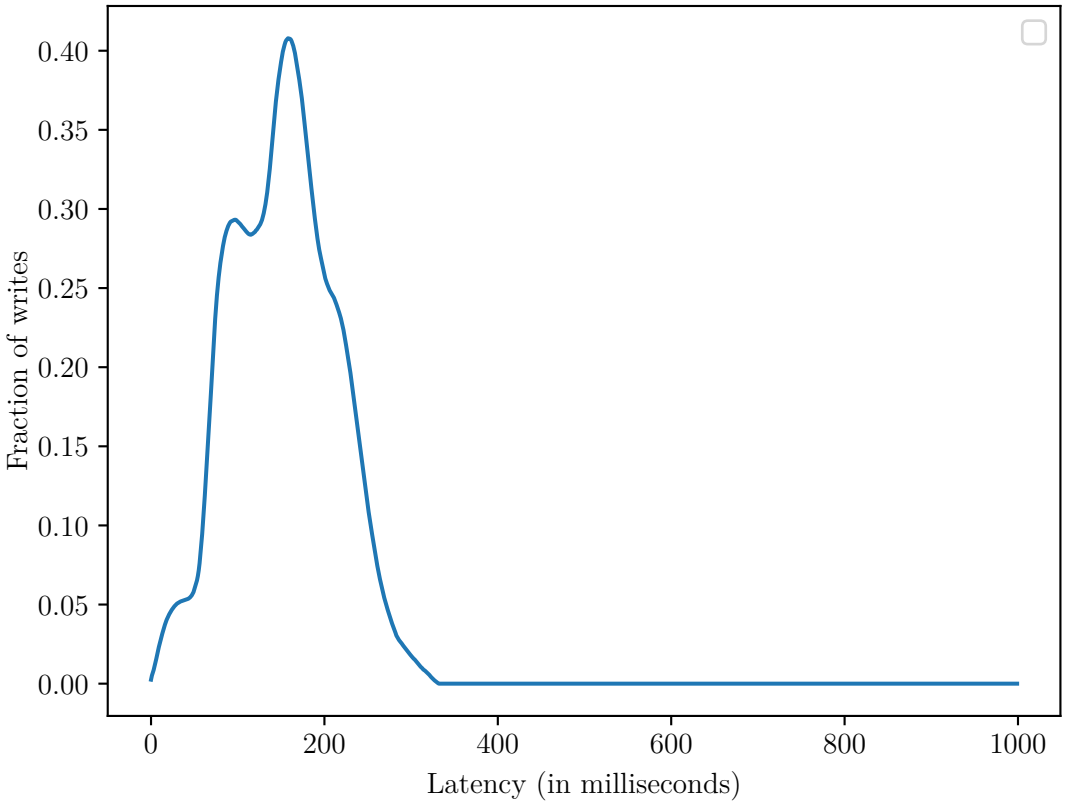


Figure 8.3: Difference in plots of Figure 8.2.

counterpart, with fewer operations appearing with more than 500ms latency. This stability can also be attributed to WiredTiger, as MongoDB can offload the responsibility of persisting writes onto a separate application. The instability of w:1 is therefore a consequence of MongoDB needing to decide when to persist operations, forcing the incoming operations to wait. Specifically, if an incoming operation is found to be dependent on an operation currently being persisted, the operation will need to wait until after the flush to disk has been performed in order to be applied to memory and acknowledged.

Moving on to the cumulative distribution of w:1 acknowledgements and estimated 1-durability in Figure 8.2, we find very predictable results, as w:1 operations get acknowledged long before those operations become 1-durable. We see an interesting result after 300ms, where the fraction of 1-durable writes overtakes w:1. This is likely due to our estimation technique, as we used the

execution history of journaled writes to estimate 1-durability. As such, the increased instability of operations with the w:1 write concern in the higher latencies is the cause of this behaviour.

The differences between the two plots as shown in Figure 8.3 show a similar pattern. The difference is small early on as most operations are not acknowledged by either write concern in the earlier latencies but increases rapidly until 200ms as the majority w:1 operations get acknowledged but few of the journaled operations do. The difference then drops quite sharply towards zero by 300ms as most of the journaled operations are now being acknowledged. We chose not to graph the negative difference (where 1-durable overtakes w:1).

Part 4

Conclusion

Chapter 9

Conclusion

This thesis explored the concept of durability in MongoDB with the aim of providing system users and designers with the understanding of the sources of data loss and the means to quantitatively measure the frequency and quantity of non-durable writes. This makes it possible to empirically evaluate the tradeoffs between highly-durable and highly-performant configurations. We also equip developers with the tools to reason about the time it takes for a write to reach varying levels of durability and a method by which they can estimate durability using client-accessible measurements.

Through this thesis, we have contributed a categorisation and evaluation for the conditions needed to cause a write durability failure. We found two scenarios that result in a permanent loss of at least one write, one related to MongoDB's election process and another as a consequence of its persistence mechanism. We also uncovered a scenario that leads to a transient loss of a write, where the write appears to be applied, then disappears and reappears once more - all due to the use of the Primary Preferred read preference in a failure scenario.

We have outlined an experiment design capable of inducing the conditions required for write loss and an analysis that can quantitatively evaluate the durability of a distributed storage system by measuring the frequency and quantity of write loss under failures. Having applied this experiment to MongoDB 3.6-rc0 and 4.0, we were able to detect expected write losses in both versions, while also finding durability and performance failures that we did not expect in MongoDB 3.6-rc0. These failures were attributed to known bugs in this version of MongoDB and have since been fixed in 4.0.

Additionally, we defined the concept of k-durability and derived a formula for the time a write becomes k-durable. We showed that we can estimate 1-durability of an operation using client-accessible measurements and ran experiments to explore when an acknowledged write becomes 1-durable in MongoDB, finding that 1-durability is achieved with probability of 0.9 by 300ms.

9.1 Limitations

We recognise that our work provides only a limited insight into the field of durability in MongoDB. Our experiment harness only induced one failure per experiment, exclusively on the primary replica which prevented us from observing the effects of a failing secondary or the behaviour of the replica set under multiple failures. Our experiment observed and reported only the measurements available to the client, ignoring MongoDB's own logs. This inhibited our capacity to analyse MongoDB's behaviour in detail and to directly observe the types of write failures that occured in the experiment.

Our estimation experiments concentrated on finding time-till-durability, exclusively for the case of 1-durability. This has allowed us to understand the pattern of when a write becomes durable, however limiting the scope to 1-durability meant that we could not find a generalised estimation for a k-durable write.

9.2 Future Work

We believe the work outlined in this thesis can be used as the foundation for furthering the exploration of durability in distributed systems. Extending upon our experiments to induce multiple failures on primary and secondary replicas would provide a more holistic view of MongoDB's durability guarantees. Additionally, exploring the behaviour of MongoDB internals by inspecting MongoDB logs could offer a greater insight into how and why certain failures occur, perhaps finding a way to minimise them.

We also believe there is great benefit in exploring how k-durability can be estimated for an arbitrary k. This can give users and developers a better understanding of how durable their operations are, equipping them with the means to design applications and systems that can account for these failures.

9.3 Closing Statements

Currently, the field of durability in distributed systems is fairly limited in scope and exploration. In this thesis, we have contributed to the greater understanding of durability in theoretical and empirical dimensions. We have offered theoretical and empirical ways of measuring how many failures occur and which categories they fall into. We have contributed a framework that allows users to evaluate tradeoffs between durability and performance. We have provided ways of reasoning about and estimating the time until a given level of durability occurs. Overall, we have equipped developers with the means the ensure their applications can be protected from these durability failures, signalling a change in the way developers think and work around this critical aspect of data storage and safety.

Bibliography

Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42.

Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167. USENIX Association, Berkeley, CA, USA.

E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2):23–29.

Eric A Brewer. 2000. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7—-. ACM, New York, NY, USA.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.

Jon G. Elerath and Michael Pecht. 2007. Enhanced Reliability Modeling of RAID Storage Systems. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 175–184. IEEE.

A. Epstein, E. K. Kolodner, and D. Sotnikov. 2016. Network aware reliability analysis for distributed storage systems. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 249–258.

Garth A Gibson. 1990. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. Ph.D. thesis, EECS Department, University of California, Berkeley.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51.

Kevin M. Greenan, James S. Plank, and Jay J. Wylie. 2010. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, pages 5–5. USENIX Association, Berkeley, CA, USA.

Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317.

Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155. ACM, New York, NY, USA.

Kyle Kingsbury. 2013. Jepsen: Mongodb https://aphyr.com/posts/284-call-me-maybe-mongodb.

Kyle Kingsbury. 2017. Mongodb 3.4.0-rc3 https://jepsen.io/analyses/mongodb-3-4-0-rc3.

Rivka Ladin, Barbara Liskov, and Liuba Shrira. 1990. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 43–57. ACM, New York, NY, USA.

Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391.

Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Leslie Lamport. 2006. Fast Paxos. *Distributed Computing*, 19:79–103.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. Technical report.

Leslie Lamport and Mike Massa. 2004. Cheap paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 307–. IEEE Computer Society, Washington, DC, USA.

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 467–483. USENIX Association, Berkeley, CA, USA.

Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500. USENIX Association, Berkeley, CA, USA.

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319. USENIX Association, Philadelphia, PA.

Kit Patella. 2018. Mongodb 3.6.4 https://jepsen.io/analyses/mongodb-3-6-4.

Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 8:1–8:14. ACM, New York, NY, USA.

Dan Pritchett. 2008. BASE: An Acid Alternative. *Queue*, 6(3):48–55.

David S. H. Rosenthal. 2010. Bit Preservation: A Solved Problem? *International Journal of Digital Curation*, 5(1):134–148.

Richard D Schlichting and Fred B Schneider. 1983. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238.

Fred B Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319.

Rong Shi and Yang Wang. 2016. Cheap and available state machine replication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 265–279. USENIX Association, Berkeley, CA, USA.

Swaminathan Sivasubramanian. 2012. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730. ACM, New York, NY, USA.

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052. ACM, New York, NY, USA.

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM*, 52(1):40–44.

Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. 2011. Zz and the art of practical bft execution. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 123–138. ACM, New York, NY, USA.