2019

ORLease: Optimistically Replicated Lease Using Lease Version Vector For Higher Replica Consistency in Optimistic Replication Systems

Diaa Fathalla
Nova Southeastern University, diaa@hotmail.com

ORLease: Optimistically Replicated Lease Using Lease Version Vector For Higher Replica Consistency in Optimistic Replication Systems

By

Diaa Fathalla

A dissertation submitted in partition fulfillment of the requirements for the degree of Doctor of Philosophy
in
Computer Science

College of Engineering and Computing
Nova Southeastern University
2019

We hereby certify that this dissertation, submitted by Diaa Fathalla, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Gregory E. Simco, Ph.D.
Chairperson of Dissertation Committee

6/6/2019
Date

Francisco J. Mitropoulos, Ph.D.
Dissertation Committee Member

6/6/2019
Date

Junping Sun, Ph.D.
Dissertation Committee Member

6/6/2019
Date

Approved:

Meline Kevorkian, Ed.D.
Interim Dean, College of Engineering and Computing

6/6/2019
Date

College of Engineering and Computing
Nova Southeastern University

2019

ORLease: Optimistically Replicated Lease Using Lease Version Vector For Higher Replica Consistency in Optimistic Replication Systems

By
Diaa Fathalla
June 2019

There is a tradeoff between the availability and consistency properties of any distributed replication system. Optimistic replication favors high availability over strong consistency so that the replication system can support disconnected replicas as well as high network latency between replicas. Optimistic replication improves the availability of these systems by allowing data updates to be committed at their originating replicas first before they are asynchronously replicated out and committed later at the rest of the replicas. This leads the whole system to suffer from a relaxed data consistency. This is due to the lack of any locking mechanism to synchronize access to the replicated data resources in order to mutually exclude one another.

When consistency is relaxed, there is a potential of reading from stale data as well as introducing data conflicts due to the concurrent data updates that might have been introduced at different replicas. These issues could be ameliorated if the optimistic replication system is aggressively propagating the data updates at times of good network connectivity between replicas. However, aggressive propagation for data updates does not scale well in write intensive environments and leads to communication overhead in order to keep all replicas in sync.

In pursuance of a solution to mitigate the relaxed consistency drawback, a new technique has been developed that improves the consistency of optimistic replication systems without sacrificing its availability and with minimal communication overhead. This new methodology is based on applying the concurrency control technique of leasing in an optimistic way. The optimistic lease technique is built on top of a replication framework that prioritizes metadata replication over data replication. The framework treats the lease requests as replication metadata updates and replicates them aggressively in order to optimistically acquire leases on replicated data resources. The technique is demonstrating a best effort semi-locking semantics that improves the overall system consistency while avoiding any locking issues that could arise in optimistic replication systems.

Acknowledgments

First, I would like to express my sincere gratitude to my advisor Dr. Greg Simco for his continuous support of my research and for his patience and motivation. His guidance helped me in all the time of research and the writing of this thesis. I would also like to thank the rest of my committee: Dr. Junping Sun, and Dr. Francisco Mitropoulos, for their insightful comments and encouragement.

Last but not least, I would like to thank my family for all their love and encouragement. My parents and sister for supporting me spiritually throughout my research and my life in general. My mother who encouraged me to pursue my Ph.D and supported me all the time. And my loving and patient wife whose support during the final stages of this Ph.D. is so appreciated.

Thank you!

Table of Contents

Abstract iii
Acknowledgments iv
List of Tables vi
List of Figures vii

Chapters

1. **Introduction 1**
   Background 4
   Problem Statement 7
   Goal 13
   Relevance and Significance 19
   Delimitations 23

2. **Literature Review 26**
   Overview 26
   Replicated Data Stores and Consistency Levels 27
   Optimistic Replication Systems 32
   Tracking Changes and Conflict Detection 34
   Conflict Mitigation 36
   Summary 40

3. **Methodology 42**
   Overview 42
   PRACTI Overview 43
   ORLease Overview 45
   ORLease Architecture 47
   Metadata 50
   ORLease Runtime Service 62
   Application Model 66

4. **Results 68**
   Testing and Evaluation 68
   Experiments 72

5. **Conclusions, Implications, Recommendations and Summary 77**
   Conclusions 77
   Implications 79
   Recommendations 80
   Summary 83

References 86

List of Tables

Tables

1. Experiment I 75

List of Figures

Figures

1. System Overview 46

2. ORLease Architecture 47

3. The Lease Acquisition and Break Process 56

4. Lease Break Scenario for Connected Network 58

5. The Scenario of Disconnected Replicas 60

6. The Lease Operation Flowchart 62

7. The Lease Operation Flowchart (Continued) 63

8. The Lease Operation Flowchart (Continued) 65

9. Stale Access Comparison I 76

10. Stale Access Comparison II 76

Chapter 1

Introduction

The tradeoff between consistency and availability of optimistic replication systems has been studied quite extensively in the literature (Saito and Shapiro, 2005). Earlier optimistic replication research attempted to improve the consistency level of these systems but they either introduced other problems such as the communication overhead in Pangaea (Saito, Karamonolis, Karlsson, and Mahalingam, 2002) or provided a good substrate with an incomplete solution such as *Partial Replication, Arbitrary Consistency, Topology Independent* (PRACTI) replication framework (Belaramani, Dahlin, Gao, Nayate, Venkataramani, Yalagandula, and Zheng, 2006). This research demonstrates a new technique that builds on PRACTI's solution in order to achieve a higher consistency level for replication systems without sacrificing their high availability while controlling the communication overhead.

The newly introduced technique is based on optimistic concurrency (Kung and Robinson, 1981) and leasing (Cary and David, 1989) in order to allow replicas to have semi-mutually exclusive access to their data resources by applying the leases optimistically as a best effort. This capability is an extension to the PRACTI replication framework (Belaramani et al., 2006), which separates the metadata replication from the data replication, so that lease requests can be propagated as metadata. The replicas receiving the lease request metadata lock their copy of the replicated data resource for the lease owner replica. The replicated data resources will stay locked until the lease owner replica sends

out a lease release request or the replicas receiving the lease request forcefully break the lease.

This new methodology is also leveraging the aggressive propagation methodology of Pangaea (Saito et al., 2002) in order to propagate the metadata aggressively so that lease requests are accelerated to all replicas. Since leases are applied optimistically, accelerating their propagation increases the likelihood of mutually excluding any data resources that are concurrently updated and lowering the possibility of reading stale data and introducing data inconsistency.

This research studies and analyzes this novel optimistic leasing approach in the context of object store replication. However, it can be applied to systems that optimistically replicate objects, files, or database records. The remainder of this paper is organized as follows. The rest of this chapter, which is "Introduction", is divided into 5 subsections. The first subsection gives a background overview about replication systems and their different types. The second subsection elaborates on the data consistency problem in optimistic replication systems. It's followed by the "Goal" subsection that outlines the approach pursued in order to mitigate the rise of data inconsistency in these systems. The fourth subsection demonstrates the significance of this research and its impact on improving the data consistency of these systems. The fifth subsection is the "Delimitations" subsection. It identifies the imposed constraints on the scope of the study in order to make it manageable.

Chapter two, which is "Review of the Literature", supports the context of the problem, the goal, and the significance of this research by reviewing early studies from the literature on replication systems. The third chapter "Methodology" focuses on the approach and the

leveraged methodologies used in order to tackle the problem and achieve the desired goal. It gives more details about the architecture and design of the solution. The fourth chapter "Results" describes the testing and evaluation metrics of the approach followed by the outcome of this research and the experiments' results. The paper is then concluded with the fifth chapter "Conclusions, Implications, Recommendations, and Summary". The chapter is concluding the research with an emphasis on the effectiveness of optimistic leases and its impact on optimistic replication systems. The chapter then identifies future research recommendations that build on optimistic leasing and is concluded with a full summary for the paper.

Background

Replication has been adopted in many distributed storage systems as a form of data redundancy to improve the storage availability and performance (Baker, Bond, Corbett, Furman, Khorlin, Larson, Leon, Li, Lloyd, and Yushprakh, 2011). Availability is improved by keeping the data accessible in the presence of some replica failures while performance is enhanced through reduced latency and increased throughput (Saito et al., 2005). Reduced latency is attained based on spatial locality by allowing users to access nearby replicas; increased throughput is achieved by having all replicas provide the same data simultaneously to multiple users (Saito et al., 2005). However, based on the following properties: consistency (C), availability (A) and tolerance to network partitions (P), the CAP theorem (Brewer, 2000) demonstrates a tradeoff between consistency and availability since network partitions is inevitable.

According to the CAP theorem, when two partitions are disconnected, replicas belonging to each partition can be made accessible during replica updates. The updates could forfeit the consistency property when they modify the same data on disconnected replicas. Likewise, if consistency needs to be preserved, then only one replica or a quorum of replicas could be made accessible for write operations in order to accept data updates, thus forfeiting the availability property. Therefore, data replication systems are implemented with either strong consistency and lowered availability or relaxed consistency and higher availability (Yu and Vahdat, 2006).

Data replication strategies have been divided into *eager replication* and *lazy replication* (Gray, Helland, O'Neil, and Shasha, 1996). The former requires all of its replicas to be synchronized so that any data update is propagated as an atomic transaction while the latter

has its replicas sending their data updates asynchronously (Saito et al., 2005). Eager replication is sometimes referred to as *pessimistic replication* because the techniques used in implementing its systems prevent read and write access to all replicas that are not up to date (Davidson, Garcia-Molina, and Skeen, 1985).

Pessimistic replication systems have a tradeoff in terms of reduced availability and scalability due to the coordination required across replicas in order to achieve strong consistency for replicated data with no concurrency anomalies (Saito et al., 2005). In these systems, replica updates over wide area networks incur high latency due to the coordination required in order to achieve strong consistency (Shankaranarayanan, Sivakumar, Rao, & Tawarmalani, 2014). This is due to the communication overhead required to maintain consistency across replicas by using consensus or even quorum protocols (Shankaranarayanan et al., 2014).

On the contrary, lazy replication achieves higher availability in the presence of network outages and increased latency. Lazy replication offers eventual consistency for the data through a communication mechanism that works in the background to propagate the data updates in order to get all replicas to converge (Demers, Greene, Hauser, Irish, and Larson, 1987). The eventual consistency is accomplished by using techniques, such as *Direct mail*, *Anti-entropy* or *Rumor mongering* (Demers et al., 1987). These techniques eventually drive all replicas toward a consistent state when the updated replicas propagate their updated metadata and data to all other replicas. This can take place at any time in the future based on replication schedule, network latency or when network connectivity is restored in case of network disruption.

When networks are partitioned or replicas are disconnected, lazy replication still allows data read and write access at every individual replica; hence achieving higher availability (Davidson et al., 1985). Lazy replication is also known as *optimistic replication* due to its optimistic approach for concurrency control which assumes all replicas can be updated simultaneously without locking their data resources (Kung and Robinson, 1981).

Problem Statement

Optimistic replication system faces two costs due to the lack of resource locking; reading of stale data and producing conflicting versions of the same data object (Demmer, Du, and Brewer, 2008), (Heidemann, Goel, and Popek, 1995) and (Yu et al., 2006). Locking in distributed systems can be accomplished through a distributed locking service, such as Google's Chubby system (Burrows, 2006) and Microsoft's Boxwood (Maccormick, Murphy, Najork, Thekkath, and Zhou, 2004), or a coordination service, such as Yahoo's ZooKeeper (Hunt, Konar, Junqueira, and Reed, 2010). These systems use the Paxos protocol (Lamport, 1998) for asynchronous consensus in order to elect a leader from a pool of servers. Once a leader is elected, clients can contact the leader to coordinate access to their shared resources.

Nevertheless, leveraging distributed locking or coordination services in optimistic replication systems will violate the asynchrony and autonomy of these systems (Saito et al., 2005). For instance, the elected leader will become the bottleneck of the replication system since all replicas have to contact it first before proceeding with their operations. This will have a negative impact on both the availability and the latency of the replication system. The availability will be impacted if the network is partitioned and the leader node is unreachable to coordinate access to the replicated resources while the latency can be affected if replicas are geographically dispersed and distantly located from the leader.

These distributed locking issues are due to the fact that there is a tradeoff between the availability and consistency properties of any distributed replication system (Brewer, 2000). Optimistic replication favors high availability over strong consistency so that the replication system can support disconnected replicas as well as high network latency

between replicas (Saito et al., 2005). Optimistic replication improves the availability of these systems by maximizing the number of writes accepted by its replicas relative to the number of writes submitted by its clients (Yu et al., 2006).

Consequently, the highest replica availability is attained by imposing an update anytime policy on every replica, even if the replica is disconnected from the rest of the system (Saito et al., 2005). This update policy relies on the asynchronous replication methodology used by optimistic replication in order to bring all replicas back in sync when network connectivity is restored. However, this leads the whole system to suffer from a relaxed data consistency due to the lack of any locking mechanism to synchronize access to the replicated data resources and mutually exclude one another (Saito et al., 2005).

When consistency is relaxed, there is a potential of introducing mutual inconsistencies that would arise when the same data object is updated on multiple replicas without mutually excluding one another. These mutual inconsistencies are considered conflicts even if the same modified data objects have the same changes (Parker, Popek, Rudisin, Stoughton, Walker, Walton, Chow, Edwards, Kiser, and Kline, 1983). This is due to the conflict detection algorithm that is based on logical clocks (Lamport, 1978) to capture causality between different versions of the same object.

The conflict detection algorithm flags any concurrent changes modifying the same data object on different replicas (Parker et al., 1983). For instance, if two data objects are modified on two different replicas, the conflict detection algorithm does not have the syntactic knowledge of the data objects or the semantic operations that were done on them in order to identify the differences between them (Parker et al., 1983). Therefore, resolving

these mutual inconsistencies is a complex problem whether it is done manually or automatically (Parker et al., 1983).

To manually fix the conflicting updates, the time consuming process would usually involve multiple users who updated the replicas with conflicting data along with a reintegration process (Kawell, Beckhart, Halvorsen, Ozzie and Greif, 1988). Other replication systems proposed automatic conflict resolution, such as Ficus (Heidemann et al., 1995), Coda (Kumar and Satyanarayanan, 1995), and Bayou (Terry, Theimer, Petersen, Demers, Spreitzer, and Hauser, 1995). These systems usually require writing complex application specific resolvers that are capable of understanding the syntax of the replicated data objects in order to automatically fix their conflicting data (Reiher, Heidemann, Ratner, Skinner, and Popek, 1994).

It is expected that the number of conflicting updates is on the rise as it was found that the *Write* access patterns have increased significantly relative to the *Read* patterns based on a study by Leung et al. (Leung, Pasupathy, Goodson, and Miller, 2008). This is due to the increase of actively changing document files when compared to system data objects that were used for sequential read access in the past (Baker, Hartmart, Kupfer, Shirriff, and Ousterhout, 1991), (Ellard, Ledlie, Malkani, Seltzer, 2003), and (Roselli, Lorch, and Anderson, 2000).

Therefore, the higher rates of *Write* access patterns will increase the likelihood of producing more conflicts that could impact the usability of optimistic replication systems (Gray et al., 1996). Consequently, it will lower the *quality of service* (QoS) of these systems due to the increase in conflicting updates and data staleness which are factors in evaluating their QoS (Kuenning, Bagrodia, Guy, Popek, Reiher, and Wang, 1998).

It was demonstrated that lowering the conflicting updates and data staleness can be achieved by aggressively propagating the data updates (Yu et al., 2006). When data updates are aggressively propagated, replicas will synchronize more rapidly; hence lowering any possibility of reading stale data or conflicting with other updates. For instance, if a replication system has a data object that is modified by a client on one replica and then replicated to the other replicas as soon as the data object updates are committed, other replicas will update the most recent updated data object and lower the chance of introducing a conflict. However, this does not scale well in write intensive environments because replicas have to go through some catch up time that is proportional to the size of the replicated data objects even if just the modified data chunks are replicated instead of the data objects in their entirety (Saito et al., 2005).

There is also a communication cost incurred when doing aggressive propagation. Some replication systems try to save on the network bandwidth by holding their modified data object replication for some time assuming that it may get modified again later; hence it is going to be replicated only once for multiple modifications (Saito et al., 2005) and (Yu et al., 2006). On the contrary, aggressive replication requires the updates to be sent out to other replicas instantaneously and that will negatively impact the optimistic replication protocol's ability to amortize the communication cost. For instance, the percentage of file reopens that are temporally related to the previous close and could occur in less than one minute can be as high as 71.1% (Leung et al., 2008). According to the same study, the ratio of write:read is 2:1 which implies that a bit over 47% of reopened files will be modified in less than a minute; hence, modified data will be replicated again if aggressive replication is used.

PRACTI replication framework (Belaramani et al., 2006) separated data replication from metadata replication so that it can perform partial replication and reduce the communication cost. Partial replication enabled PRACTI to replicate modified data object's metadata first in order to mark the data objects themselves as invalid on other replicas and then replicate the actual data lazily or on demand. By holding off the instantaneous data replication, the communication cost is amortized since the replication metadata is usually considered negligible when compared to the actual replicated data (Wang, Alvisi, and Dahlin, 2012). For instance, the communication cost for a data object that is frequently modified but marked for replication on demand is going to be just proportional to the size of its replication metadata multiplied by the number of modifications it incurred.

Belaramani et al. also claim that the PRACTI framework is capable of providing a tunable consistency that can be weakened or strengthened (Belaramani et al., 2006). However, the acclaimed strong consistency that can be achieved by this framework has its own limitations. For instance, when a client opens a file for write with strong consistency, the replica will block the write so that it synchronizes the write operation with other replicas. This implies that every replica has to communicate with at least a quorum of its peer replicas; hence, rendering the whole system to be limited in scalability and availability.

Consequently, in a cloud deployment, object stores avoid strong consistency for objects that are replicated across multiple geographical regions due to the increased latency (Shankaranarayanan et al., 2014). For instance, SCFS (Bessani, Mendes, Oliveira, Neves, Correia, Pasin, & Verissimo, 2014) proposed a cloud of clouds backed file system that

provides strong consistency by using a coordination service that Bessani et al. called it the consistency anchor. However, the latency of file create, open and close operations was almost four orders of magnitude higher due to the extra communication with the consistency anchor in order to coordinate between different clients to operate on shared files.

On the contrary, latency was not an issue in BlueSky (Vrable, Savage, & Voelker, 2012) due to the lack of coordination between the different proxies accessing the same file; hence, leading to potential update conflicts. BlueSky is a network file system that is backed by cloud storage providers such as Amazon S3 ("Amazon S3", n.d.) and Microsoft Azure ("Microsoft Azure", n.d.) where clients may access the files on the cloud storage through a proxy. Similarly, the cloud based file system Coral (Chang, Sun, and Chen, 2016) favored the low latency and high availability over data consistency and uses the latest-version-wins mechanism (Thomas, 1979) to resolve conflicts.

Therefore, strong consistency was either considered to be complex to achieve in BlueSky as stated by Vrable et al. (Vrable et al., 2012) or will affect the performance and availability of Coral as stated by Chang et al. (Chang et al., 2016). This became apparent in SFCS (Bessani et al., 2014) as the latency was negatively impacted by the consistency anchor. Therefore, these shortcomings paved the way to explore and introduce a novel locking mechanism that improves the consistency level of optimistically replicated object stores that have a reliable network connectivity to be as close as possible to strong consistency without sacrificing its high availability and low latency.

Goal

Objects in a replication system are concurrently accessed from multiple replicas and a mutual exclusion mechanism is required in order to achieve strong consistency. However, mutual exclusion has a negative impact on the availability and the latency of these systems as explained earlier. Therefore, the main goal of this research is to introduce a new technique for optimistic replication systems that improves their consistency level without compromising their availability and latency.

The new technique is based on optimistic concurrency (Kung et al., 1981) and leasing (Cary et al., 1989) in order to acquire a lease on data objects by optimistically replicating the lease request; hence, the technique has been called ORLease or Optimistically Replicated Lease. Leveraging leases to mutually exclude data objects should provide a locking mechanism that could be forfeited by allowing any replica to break acquired leases.

The lease forfeiture has the benefit of not locking a data object indefinitely if the lease's owner replica is disconnected from the network since network partitioning is not uncommon in distributed systems. The lease is optimistically applied by asynchronously broadcasting the lease request to all other replicas as a metadata update without waiting for leasing acknowledgments from them. This allows an optimistic lease to achieve a best effort locking semantics without compromising the latency and availability of the replication system.

In order to propagate the lease request, the object's replication metadata needs to be separated from its data similar to what has been accomplished in PRACTI (Belaramani et al., 2006) and Gnothi (Wang et al., 2012). The metadata separation allows lease requests to be sent out as metadata similar to the replication metadata being exchanged between

replicas as explained in further details in chapter 3 “Methodology”. The lease metadata can then be aggressively propagated similar to Pangaea (Saito et al., 2002) so that the acquired leases are accelerated to all replicas. Accelerated leases are required to minimize the window gap that might give a chance to other replicas to acquire leases on the same data object leading to potential conflicts.

PRACTI (Belaramani et al., 2006) has already demonstrated that the metadata separation allows the metadata of changes to propagate aggressively leading to improved consistency. However, the metadata is propagated after the object has been updated similar to all optimistic replication systems (Saito et al., 2005). Since leases in ORLease are optimistically issued when objects are opened with the intent to update but before changes are committed, accelerating the lease propagation increases the likelihood of mutually excluding objects from being concurrently updated from other replicas and lowering the possibility of introducing data inconsistency. Thus, ORLease demonstrates a better consistency improvement over PRACTI (Belaramani et al., 2006) for systems with larger window of time between requesting object updates and committing them.

The accelerated lease requests that are sent out as metadata updates have a minimal communication overhead for two reasons. First, the lease requests are going to be considered as control messages (Belaramani et al., 2006) that are broadcasted in one direction with no need for acknowledgements. Secondly, the lease request is metadata that has negligible size when compared to the actual object data size (Wang et al., 2012). Consequently, the aggressive replication of the added lease requests should improve the consistency of connected replicas in a replication system by providing a window of

metadata staleness guarantees that equals to the maximum latency between any two replicas.

Objects in replication systems that separate the data and metadata are also impacted by data staleness. The data staleness affects the availability because its window of staleness is unbounded as data is asynchronously replicated on demand similar to PRACTI (Belaramani et al., 2006). However, the metadata replication is the one affecting the consistency because it allows any replica to convey the state of a modified object to other replicas so that they can react accordingly. Once the metadata update is received by other replicas, the object is considered to be logically up to data even if its data is not yet available.

ORLease's new technique to replicate the lease request is an addition to the metadata update that improves its consistency over any other optimistic replication system. The reason is that ORLease's lease request is broadcasted to all replicas before an object is accessed for modification. Other replication systems are either replicating out the data and metadata after the object is closed (Saito et al., 2005) or aggressively replicating out metadata invalidation requests after the object has been modified as in PRACTI (Belaramani et al., 2006). Therefore, replicating out the metadata before the object is modified reduces the metadata staleness window when compared to all other replication systems that replicate the metadata after the object has been modified and closed.

The addition of ORLease's optimistic lease requests to the metadata updates does not change the behavior of optimistic replication systems in case optimistic leases are not required. For instance, ORLease systems allow concurrent reads and writes with no locking semantics for applications that do not require optimistic lease and the behavior should be

similar to other optimistic replication systems (Saito et al., 2005). However, ORLease provides semi-strong consistency semantics for applications that require such semantics (Belaramani et al., 2006) and (Yu and Vahdat, 2000).

ORLease's technique allows applications to issue an optimistic lease in a best effort to lease the object for read or write operations. The optimistic leases would attempt to grab leases on replicated objects so that applications that are accessing the objects for read or write operations can have a consistent view of their data objects. The applications can also get notifications when they attempt to lease objects that other replicas are holding optimistic leases against them. This allows applications to either cancel their lease request or revoke other replicas' leases. Revoking the leases of other replicas is necessary in the case of network partitioning so that leases are not held indefinitely by disconnected replicas. However, revoking leases have the potential of introducing conflicts if data objects are modified by both the leasing replicas and the lease revoking replicas.

Estimating the window of stale access for data objects in ORLease has been very crucial in order to evaluate its success. Stale access gives information about the replication system divergence time from the ideal semantics when all replicas are in sync. The divergence time starts from the time an object is opened to be modified and not when it is committed. For instance, if an object is opened on a storage system for update at time t, modifications committed to storage at time t+x, replicated out and reached the furthest replica at time t+y, then the stale access time is y and not (y - x) where $y > x$.

Therefore, the evaluation has been based on a modified version of the *stale-access* metric. The *stale-access* metric is considered as one of the proper quality of service metrics for optimistic replication systems (Kuenning et al., 1998). The modified metric has been

identified in this research as the *leased stale-access* metric. The *stale-access* metric is the difference in time between two replicas when an update is started at one replica and when the updated object is available at the other replica (Bermbach and Kuhlenkamp, 2013). However, the *leased stale-access* metric is the difference in time when a replica receives a lease for an object, that has been leased by another replica, and when the object is updated and available at the same replica. The metric name is prefixed with the word *leased* because it identifies a portion of the *stale-access* period where the object is leased for a specific replica and is protected from being modified by other replicas. Even though the object is still stale during that period of time, it cannot be modified by any replica other than the lease owner.

The optimistic replication systems that are enhanced with the optimistic lease technique have reduced their window of *stale-access* of data objects by excluding the *leased stale-access* period of time from it. This is due to the fact that the opening of the object with the intent to update it triggers an optimistic lease request in ORLease. This request is then honored by all replicas which in turn locks the object from being modified by them. Hence, it allows replicas to maintain an optimistic consistent view of the metadata using the optimistic leases but without having all of the objects' data contents.

PRACTI (Belaramani et al., 2006) was successful in reducing the *stale-access* by replicating the metadata of the update once the object is updated and committed. ORLease has reduced the window of *stale-access* even further by replicating the metadata for optimistic leases when the replica opens a data object with the intention to update it. However, ORLease's effectiveness in reducing stale access should be better demonstrated in replication systems that has their data objects updated over a large window of time such

as the replication of files in Distributed File System Replication ("Microsoft Distributed File System Replication (DFSR)", n.d.) and Azure File Sync ("Microsoft Azure File Sync (AFS)", n.d.). Files in DFSR and AFS can be opened for hours before changes are committed and replicated. Otherwise, if the difference between the object start update time and update committed time is close to zero, then PRACTI should demonstrate a better performance because it does not need to broadcast metadata updates for leasing its data objects.

Relevance and Significance

Optimistic replication is widely used in different types of products, including wide area applications, mobile device applications, data distribution and data collaboration (Saito et al., 2005). It is implemented in some well-known systems for high availability and fault tolerance, such as the internet Domain Name Service DNS (Mockapetris and Dunlap, 1988). DNS is the standard hierarchical distributed naming service for the internet and it manages the naming within its zones through a single master replication system in each zone. The zone's master replica maintains the authoritative naming database that is updated by the system administrator and then replicated to the slave replicas. This is a highly available system that is capable of fulfilling query requests coming from multiple servers. However, the master replica is the only one that can be updated in order to avoid introducing any conflicting data.

Another well-known system that leveraged optimistic replication is the wide area bulletin board system Usenet (Lin et al., 1999). It is a system that replicates articles between its sites so that users can read the articles from their nearest neighboring site. It is a multi-master optimistic replication system that was designed to be conflict free. It used the simplest approach for resolving conflicts based on Thomas's write rule (Thomas, 1979) which is the last writer wins. Users might have found it confusing to find articles disappearing after they have been updated or resurfacing after they have been deleted due to the side effect of applying Thomas's writer rule (Thomas, 1979). However, it was a reasonable cost in exchange of the system high availability (Saito et al., 2005).

The design of DNS (Mockapetris et al., 1988) and Usenet (Lin et al., 1999) systems realized that the adoption of optimistic replication systems brings some interesting

challenges for replica consistency. These systems avoided dealing with conflicts by either implementing a single master replication as in DNS or by adopting Thomas's writer rule (Thomas, 1979) for conflict resolution as in Usenet. To completely prevent conflicts from happening, a recent study (Shapiro, Pregui_ca, Baquero, and Zawirski, 2011) leveraged simple mathematical properties, such as commutativity. For instance, a counter data type, which can be incremented or decremented, will converge because its increment and decrement operations commute. However, this requires building new systems and applications based on these data types which is not practical for replication systems that replicates generic data objects.

Unfortunately, stale reads and conflicting updates are not uncommon in other optimistic replication systems because coordination between replicas is done asynchronously in the background to propagate the data updates (Saito et al., 2005) and (Yu et al., 2006). Conflicting updates is an accepted cost in some commercial environments, such as banks' ATM machines and airline reservation systems, because the availability and performance of these systems outweigh the need for strong consistency (Heidemann et al., 1995). Nevertheless, the number of conflicting updates in optimistic replication systems affects its quality of service. This led to implementing systems with pluggable modules to resolve conflicts such as Ficus (Reiher et al., 1994) and Coda (Kumar et al., 1995). Other systems, such as TACT (Yu et al., 2000) and PRACTI (Belaramani et al., 2006), incorporated some techniques into their replication framework in order to reduce stale access and lower the number of conflicts.

Coda file system (Kumar et al., 1995) and the Ficus file system (Reiher et al., 1994) considered conflicts as rare events and provided automated conflict resolvers. For instance,

Coda has provided a framework that invokes *application-specific resolvers (ASRs)* (Kumar et al., 1995) to handle the conflict resolution process. Conflicts will then get resolved automatically for applications that have an implemented ASR and the rest will require user intervention. However, these conflict resolver are complex to implement because they have to understand the syntax of replicated files in order to automatically fix their conflicting data (Reiher et al., 1994).

TACT (Yu et al., 2000) and PRACTI (Belaramani et al., 2006) attempted to lower conflicting updates by incorporating different technique that ties to the system replication framework. For instance, TACT implemented a middleware layer that controls client read/write access to replicas as well as the data propagation between replicas based on some defined consistency bounds that can be targeted for certain applications. The authors of the TACT framework also demonstrated in a later study that aggressive write propagation can achieve the highest levels of consistency for replication systems (Yu et al., 2006). However, aggressive propagation does not scale well in write intensive environments because replicas have to go through some catch up time that is proportional to the data size even if modified data object chunks are replicated instead of data objects in their entirety (Saito et al., 2005). Aggressive propagation also incur a communication cost as explained earlier in the "Problem Statement" subsection.

To overcome the aggressive propagation issues, the PRACTI replication framework (Belaramani et al., 2006) separated the data replication from the metadata replication so that it can perform partial replication. Partial replication enabled PRACTI to replicate out the modified file's metadata first in order to mark the data objects as invalid on other replicas and then replicates the actual data lazily or on demand. Therefore metadata

replication can be replicated aggressively while the communication cost is amortized since the replication metadata is usually considered negligible when compared to the actual replicated data (Wang et al., 2012).

All this strongly suggests that the aggressive replication of replica knowledge or metadata should get the replica consistency to be closer to strong consistency since the metadata is negligible in size (Wang et al., 2012). The availability of the system is slightly affected since the communication overhead is minimal. The replicas are brought in sync faster since the metadata is quicker to propagate (Wang et al., 2012). However, this approach is missing the locking semantics that could potentially reduce the staleness window dramatically.

Consequently, ORLease introduced a semi-locking methodology by leveraging the existing system's communication methodology and causality capturing techniques. It optimistically broadcasts lease requests by utilizing the same metadata replication channels that are already used by optimistic replication systems in order to achieve semi-locking for the replicated objects. It is considered a continuation for PRACTI as it decreases the staleness window by removing the system dependency on the replicated object size and its commit time. It reduces the factors that affect the staleness window to only the network latency and replica connectivity.

Delimitations

This research focused on leveraging the newly introduced technique ORLease in optimistic replication systems in order to enhance their consistency. The focal point was to optimistically broadcast the lease requests of replicated objects in these replication systems through their metadata replication mechanism in order to shorten their *window of inconsistency* (Bailis and Ghodsi, 2013). Therefore, a few simplifications have been undertaken in order to expedite the research outcome and to simplify its evaluation process.

The first simplification was to evaluate the ORLease replication technique in a replicated object store that is based on a flat namespace with no directories or subdirectories. This is similar to the simulation framework developed by Wang et al. (Wang, Reiher, and Bagrodia, 1997) in order to evaluate their optimistic replication system. Their system was based on a flat namespace instead of a hierarchical one in order to simplify their prototype implementation and its evaluation process.

The second simplification was to have a coarse lease and replication granularity where data objects in the replicated object store are leased in their whole entirety. The modified data objects are then replicated lazily as a whole instead of replicating just the modified parts of the object similar to some replication systems (Yu et al., 2006). In order to support partial replication of objects, different replicas will have to optimistically lease different parts of the same object. However, this approach will just complicate the implementation and evaluation process and was considered an enhancement to be deferred for future work.

The third simplification was to conduct the ORLease experiments on a single machine. Consequently, all replication processes and their replicated object stores coexisted on the same machine. Since Simulation frameworks have been previously developed to simplify

the evaluation of optimistic replication systems (Wang et al., 1997), the network latency has been simulated between the replication processes. Therefore, latency could be simulated in the communication between the replication processes in order to reflect the actual measured delay between regions ("LAN performance on the WAN", n.d.) and (Shankaranarayanan et al., 2014).

The fourth simplification was to evaluate ORLease for a limited number of object store operations since the optimistically replicated lease is considered a semi-locking operation itself that precedes all object's data read or write operations. An optimistic lease is replicated out before the actual read or write operation is executed and then released when the operation is complete. Therefore, the evaluation was based only on object creation operations while other operations such as object read, update, delete and rename have not been implemented or evaluated as part of this research due to their similarities in their lease requirements to the create operations.

The fifth simplification was to optimistically replicate leases without prioritizing the data replication for the incomplete objects. As explained later in the "Methodology" chapter, an object can be marked as incomplete similar to PRACTI (Belaramani et al., 2006) because of the metadata replication prioritization and the object can still be optimistically leased by any replica. The optimistically replicated lease request can trigger data replication prioritization from replicas that have the complete data objects. However, this was considered an enhancement that will not provide additional value to this research and has not been evaluated. Therefore, this research has implemented and evaluated ORLease based on holding the lease on the object until the data is fully downloaded and available. It then releases the lease instead of depending on the data replication

prioritization to block any new incoming request if data is not yet available until it is fully downloaded and available.

Chapter 2

Review of the Literature

Overview

This chapter elaborates on the major areas that build the foundation of this research. The first subsection discusses the replicated data stores and their different levels of consistency. The second subsection discusses the significance of optimistic replication systems and their need due to the advancement of mobile systems. The third subsection describes the mechanism used for tracking changes and detecting conflicts in optimistic replication systems. The fourth subsection will then elaborate on the conflict detection and mitigation techniques leveraged by related studies in order to alleviate the conflicted updates problem in these systems. The fifth subsection is a summary for the techniques leveraged by this research in order to improve its overall consistency.

Replicated Data Stores and Consistency Levels

Data stores are repositories that persist and manage collections of data which can be organized in the form of a complex relational database or as simple as a collection of files. The data can then be stored on single node or more than one node in order to have a distributed data store for fault tolerance and high availability. In these distributed data stores, the data is replicated between the nodes either synchronously or asynchronously (Saito et al., 2005). The replication methodology used determines how the replica state and data get updated and how the clients subsequently observe these updates; hence, dictating the consistency of the replicated data.

One important aspect of any replicated data store is the consistency level guarantees that it provides to its clients (Saito et al., 2005). Different systems provide different consistency levels in order to manage different latency and availability requirements. For instance, a system might provide a strong consistency guarantees to its client so that all clients always have a consistent view of the data objects. However, strong consistency levels require coordination between the replicas for the execution of operations which can lead to higher latency. Therefore, systems may relax these guarantees in order to have better availability and lower latency.

There are two different perspectives on consistency; data-centric and client-centric (Bermbach et al., 2013). These perspectives are based on how the data state of the replicated store is internally viewed or externally observed. The data-centric consistency perspective views the internal data of the system based on the synchronization protocol used between its replicas while the client-centric perspective is the externally observed consistency behavior of such system.

Both data-centric and client-centric consistency perspectives have two dimensions for the consistency guarantees; staleness and ordering (Yu and Vahdat, 2002). Staleness describes how much a given replica is lagging behind for a specific update while ordering describes how many requests executing on a secondary replica have deviated from the chronological order of requests that are being executed on the primary replica. Based on those two perspectives and their two consistency dimensions, there are various consistency models that can be provided by the replicated data store (Coulouris, Dollimore, Kindberg, and Blair, 2011).

The data-centric consistency perspective describes the consistency level of the replicated data storage based on the synchronization algorithms that are internally used (Bermbach et al., 2013). The most common consistency levels provided by a distributed replicated storage system are either strong consistency (Bermbach et al., 2013), per-object sequential consistency (Cooper, Ramakrishnan, Srivastava, Silberstein, Bohannon, Jacobsen, Puz, Weaver, and Yerneni, 2008), causal consistency (Ahamad, Neiger, Burns, Kohli, and Hutto, 1995), or eventual consistency (Saito et al., 2006).

The strictest consistency level to be provided to a client is the strong consistency level as it provides clients with a consistent view for all objects at all times (Coulouris et al, 2011). It is known as single copy consistency because it provides clients with replica views as if there is only a single server in the distributed storage system. There are two semantics for strong consistency; linearizability and serializability (or sequentially consistent) (Coulouris et al, 2011). Linearizability ensures that any interleaved sequence of operations from different clients are executed in the same order as if they were executed on one replica and the order of operations are consistent with the real times at which the operations

executed (Herlihy, and Wing, 1990) and (Coulouris et al, 2011). Similarly, serializability guarantees the same order of execution but without appealing to real time (Coulouris et al, 2011).

The aforementioned consistency levels suffer from high latency issues and do not scale in geographically replicated systems (Shankaranarayanan et al., 2014). Therefore, these consistency levels were relaxed in order to avoid global ordering of clients operations as in the per-object sequential consistency level. It guarantees serializability just at an object level; hence, it ensures that clients operations are serialized and ordered per object (Cooper et al., 2008). Its synchronization algorithm guarantees that each replica applies the same updates in the same order for every object in the system. However, its algorithm has no guarantees for global or even partial ordering of clients operations across multiple objects.

Another weaker consistency than sequential consistency is the causal consistency. It guarantees that clients operations are always executed after the execution of earlier client operations which they are causally dependent on. Its algorithms ensure partial ordering between causally dependent clients operations (Ahamad et al., 1995). Both the per-object sequential consistency and causal consistency guarantees partial ordering of operations. The former guarantees partial ordering of all clients operations per object while the latter guarantees partial ordering for causally dependent clients operations across objects.

The weakest consistency level of all previously introduced consistency levels is the eventual consistency level. The reason is that eventual consistency does not have a formal definition for the order of its clients operations. An eventually consistent system is relaxed in terms of concurrency control and the only guarantees it provides to its clients is that all replicas will eventually converge. Its clients would have inconsistent views of the system

and stale data at times when replicas are disconnected or when they suffer from high latency connections between themselves. Therefore, it is leveraged by optimistic replication systems because they do not block client access when data are inconsistent or stale (Saito et al., 2005).

Aside from the strong consistency algorithms, each of the weak consistency synchronization algorithms might have a different client-centric perspective consistency level. The client-centric consistency levels were proposed as session guarantees for application in order to manage the weakly consistent replicated data in their replicated storage systems (Terry, Demers, Petersen, Spreitzer, Theimer, and Welch, 1994). A session is an abstraction used to represent the application's view for a sequence of read and write requests that are performed during the execution of the application. It has four different consistency models; *Monotonic Read Consistency (MRC), Read Your Writes Consistency (RYWC), Monotonic Writes Consistency (MWC), and Write Follows Read Consistency (WFRC)* (Terry et al., 1994), (Vogels, 2008) and (Bermbach et al., 2013).

The first model, *Monotonic read consistency (MRC)*, guarantees that if a client reads version n of an object, it will thereafter always read versions greater than or equal to n for the same object. Basically, if a client reads a certain version of an object from a specific replica, it will not go back in time reading older versions for the same object from any replica. Similarly, the second model or *Read Your Writes Consistency (RYWC)*, guarantees that if a client updates an object to be of version n on a specific replica, then it will always read versions that at least equal to version n for the same object from any replica. It ensures that a client will not go back in time reading older versions than its latest object update even if the read operation is taking place on any of the peer replicas.

The third model; *Monotonic Writes Consistency (MWC)*, guarantees that multiple updates from the same client to the same object are serialized in order. Basically, if updates are taking place on different replicas, then a subsequent write operation will only be allowed to execute on replicas that have the latest preceding write operation. Similarly, the fourth model or *Write Follows Read Consistency (WFRC)*, guarantees that an update following a read of version n will only execute at replicas that have at least version n of the object being updated.

Optimistic Replication Systems

Pessimistic replication systems tend to have strong consistency guarantees; hence, they require reliable connections and low latency between their replicas in order to manage the synchronously exchanged messages during data updates (Zhao, 2014). On the contrary, optimistic replication systems allow data updates to be applied at the local replica and asynchronously schedule the required messages in order to replicate out its updates to its peer replicas; hence, providing clients with weaker consistency levels (Saito et al., 2005). Nevertheless, optimistic replication systems are driven by the need for data replication over the internet and wide area networks because they do not require reliable connections or low latency networks. They are becoming increasingly popular due to advancement of mobile computers and the need to handle their intermittent connectivity (Zhao, 2014) and (Coulouris et al., 2011).

Optimistic replication systems achieve higher availability than their pessimistic counterpart in the presence of network outages and increased latency (Saito et al., 2005). The asynchronous nature of propagating data updates between their replicas ensures that disconnected replicas are reconciled with the rest of the system when network is restored back (Saito et al., 2005). Their replicas will eventually converge and reach a consistent state because of their epidemic communication mechanism; however, the convergence process could take some time (Demers et al., 1987).

The convergence time depends on factors such as the replication schedule of each replica and the network latency between replicas. Less frequent replica schedule and higher network latency implies a longer stale window; hence, more potential write conflicts and stale reads that could occur in an optimistic replication system with multi-master replicas.

For instance, it has been demonstrated by one of the file system studies (Leung et al., 2008) that more than 55% of shared opens from different clients occur within one minute of each other. Therefore, it is expected that delaying the propagation of data updates will increase the likelihood of reading stale data and introducing conflicted updates in optimistic replication systems. This is due to the increased chance of shared opens between different clients and the lack of locks to mutually exclude one another.

Tracking Changes and Conflict Detection

Tracking changes in distributed systems relies on logical clocks (Lamport, 1978) in order to capture causality between different versions of the same object. Vector clocks are used in distributed environments to order events based on a logical clock that captures the relations between distributed events (Lamport, 1978). Even though there could be a time gap between two distributed events and one of them *happens-before* the other, they could still be considered concurrent events. This would be the case if both events occur on two different replicas but cannot be related to a third event in order to correlate their event ordering (Lamport, 1978) and (Saito et al., 2005). Similar to Vector clocks, version vectors (VV) captures relations among distributed replicas to relate replica states instead of replica events. They were introduced by that name to track object's modification history in LOCUS (Parker et al., 1983).

LOCUS assigns each replica a unique identifier and a counter that acts as a logical clock and keeps incrementing with every object change. When an object is modified on multiple replicas, its version is assigned the replica's unique identifier and the current counter value. For example, if an object is replicated between 3 different replicas R1, R2 and R3, the version vector of the object would take the form of (R1: i, R2: j, R3: k) where i, j, and k are the last update number that R1, R2, and R3 have applied to the object respectively. Therefore, the version vector is of variable length depending on the number of replicas in the system. It can be represented as N number of (replica id, last update number) pairs where N is the number of replicas that have updated the object.

Subsequently, when two replicas R1 and R2 exchange their version vectors VV1 and VV2, it is said that both version are compatible if one version vector dominates the other.

For example, if VV1 is (R1: 7, R2: 5) and VV2 is (R1: 4, R2: 5), then VV1 dominates VV2 because R1 was the last replica to apply an update to the object. Therefore, VV1 and VV2 are not conflicting because Thomas's write rule can be applied in order to copy R1's object to R2 and make both R1 and R2 consistent. On the contrary, a conflict could be detected if neither of the version vectors is dominating the other. From the previous example, if VV1 is (R1: 7, R2: 5) but VV2 is (R1: 4, R2: 6), then neither VV1 is dominating VV2 nor VV2 is dominating VV1 and a conflict would occur if Thomas's write rule is used to copy the object either from R1 to R2 or vice versa. Hence, R1 and R2 are not consistent and in order to manually or automatically reconcile them, objects have to be merged or one of the objects has to be picked to win the conflict.

Conflict Mitigation

Many studies and systems have focused on the conflict detection algorithms as explained earlier in order to manually or automatically resolve conflicts (Heidemann et al., 1995), (Wang and Amza, 2009), (Saito et al., 2005), Coda (Satyanarayanan, Kistler, Kumar, Okasaki, Siegel, & Steere, 1990), Dynamo (DeCandia et al., 2007), Ficus (Reiher et al., 1994), and Bayou (Terry et al., 1995). There were also other studies that focused on improving the consistency of optimistic replication systems either by enforcing read/write ordering (Terry et al., 1994), by bounding replica divergence (Yu et al., 2000), by leveraging aggressive propagation (Saito et al., 2002), by using probabilistic techniques (Lawrence, Rowstron, Bishop, and Taylor, 2002), or by separating the data and metadata (Belaramani et al., 2006).

Improving the consistency of optimistic replication systems is a reaction to the eventual consistency drawbacks. Users of such systems used to sometimes see older replicated objects after they have already updated them as if the objects were going back in time. This problem was addressed by enforcing the read and write ordering of objects based on some predefined policies using session guarantees (Terry et al., 1994). The policies used were based on the four consistency models observed by clients; *Monotonic Read Consistency (MRC), Read Your Writes Consistency (RYWC), Monotonic Writes Consistency (MWC), and Write Follows Read Consistency (WFRC)* (Terry et al., 1994), (Vogels, 2008) and (Bermbach et al., 2013).

Session guarantees were implemented using a *session* object carried by each user (Terry et al., 1994) and (Saito et al., 2005). The session object has two kinds of information; *write-set* and *read-set*, represented in a compact form as a version vector (Saito et al., 2005). The

g*write-set* preserves the past objects versions of the write operations submitted by the user and the *read-set* preserves the past objects versions read by the same user. This information is used to ensure that; for example, the *Read Your Writes Consistency (RYWC)* policy is enforced and the user would always read his last written information.

Another way improve the consistency of optimistic replication systems was to bound the replica divergence as demonstrated by TACT (Yu et al., 2000). TACT implemented a middleware layer that controls client read and write access to replicas as well as the data propagation between replicas based on some defined consistency bounds that are user specified. For instance, a replica would stop accepting updates from clients once it detects that the number of its uncommitted operations on the site exceeds the user specified limit (Saito et al., 2005). TACT used to deduce the number of uncommitted operations by exchanging metadata with peer replicas and calculating the difference.

Improving the consistency in optimistic replication systems has also been demonstrated by the aggressive propagation of data updates in the Pangaea file system (Saito et al., 2002). The aggressive propagation has also been evaluated by a later study (Yu et al., 2006) that corroborated the aggressive propagation effect on achieving the highest availability and consistency for optimistic replication systems; thus, leading to lower conflicting updates and reduced staleness. However, it was concluded that these techniques do not scale well in write intensive environments (Saito et al., 2005). Replicas will have to go through some catch up time that is proportional to the data size even for divided data objects (Saito et al., 2005). In addition, there is a communication cost that is a result of the traffic incurred to aggressively propagate the data (Yu and Vahdat, 2001).

To overcome the communication cost in replication, both the Gnothi system (Wang et al., 2012) and the PRACTI replication framework (Belaramani et al., 2006) separated the metadata from the actual replicated data in order to propagate the metadata first. Gnothi, which is a pessimistic replication system, leveraged this approach to ensure that the latest data updates are synchronized amongst all replicas by replicating the metadata to all replicas while replicating the actual data to only a subset of the replicas. This improves the recovery speed of the Gnothi system and maximizes its I/O throughput because it executes the write operations on subsets of replicas. For instance, the random I/O performance of write operations in Gnothi is 40 to 64% faster than Gaios (Bolosky, Bradshaw, Haagens, Kusters, and Li, 2011) when using 3 replicas because Gnothi writes data to only 2 replicas while Gaios writes to all 3 replicas like any other pessimistic replication system.

Gnothi is considered a pessimistic replication system that requires all data to be synchronized before any data update can take place. However, allowing data updates for a subset of the replicas has also improved the overall system availability because it shortened the period of time the system would take in order to be ready to accept data updates from clients. Even replicas with incomplete data could participate in replication with other replicas while some of its data blocks are not available and are being lazily downloaded. This was accomplished by including a *complete/incomplete* metadata flag that is set when the data is available and fully downloaded; otherwise the flag is not set. In addition to the *complete/incomplete* flag, there is a data block version that determines whether the data is stale or not.

PRACTI replication framework (Belaramani et al., 2006) has also leveraged the metadata separation technique for optimistic replication systems in order to do partial

replication. Partial replication enabled PRACTI to replicate out the modified file's metadata first in order to mark the data objects as invalid on other replicas and then it replicates the actual data lazily or on demand. This reduced PRACTI's communication cost and improved its availability; however, PRACTI came short in addressing the consistency problems by leveraging a distributed locking mechanism.

It is well understood that data locking and synchronization is not a property of optimistic replication systems; otherwise, these systems will lose their availability edge over pessimistic replication systems (Saito et al., 2005). However, it is postulated that leveraging locks to be acquired optimistically (Kung et al., 1981) reduces the conflicting updates even though they will not be completely eliminated due to concurrent updates and disconnected replicas.

Summary

Optimistic concurrency (Kung et al., 1981) has been used in database systems. It assumes that concurrent database transactions can complete without conflicting with each other. Based on that assumption, optimistically concurrent transactions use the database resources without acquiring locks. However, before optimistic transactions are committed, they verify whether the data they modified has been updated by other transactions or not. If a data update is detected, they rollback their transaction and restart.

The same analogy of optimistic concurrency is applied to optimistic replication systems in order to improve their consistency level. ORLease's idea is about optimistically replicating leases (Cary et al., 1989) as metadata updates in order to achieve a semi locking semantics for the replicated data. The leasing metadata is aggressively replicated out in order to ensure that objects can be leased quickly. This reduces the staleness dimension of its eventual consistency level even though the data is not replicated out along with the lease metadata. The reason is due to the leased objects being blocked by other clients from read and/or write until the lease is over. Blocking other clients' reads avoid reading stale data and blocking their writes is a safeguard against introducing conflicts.

For instance, if a client attempts to open a data object for a write operation on one of the system replicas, the client request then triggers the replica to send out a lease request to the other replicas in the system in order to lock the data object as a best effort. Once the lease is received by the peer replicas, the object is considered locked and the peer replicas can block clients from accessing the leased object. Therefore, if the lease request conflicts with other lease requests coming from other replicas, the optimistic replication system will allow the data resources to be modified by multiple replicas. Even though the number of

conflicted updates should be lowered when using ORLease, a conflict resolution mechanism is still required in order to resolve the conflict as explained later in the "Methodology" chapter.

Chapter 3

Methodology

Overview

This section elaborates on the architecture and design of the ORLease framework. The first subsection gives an overview about the PRACTI replication system (Belaramani et al., 2006) since it is considered the baseline for this research. The second subsection gives an overview about ORLease and its functionality. The third subsection demonstrates ORLease's architecture showing its key design elements and main building blocks. The fourth subsection shows how metadata replication is leveraged to optimistically replicate leases. The fifth subsection demonstrates the ORLease runtime and how all the building blocks interact together to achieve a semi locking semantics for replicated data objects. Finally, the sixth subsection elaborates on the application model and the effect of updating and leasing the data objects that are being replicated by the ORLease replication framework.

PRACTI Overview

PRACTI (Belaramani et al., 2006) is a replication architecture that separates the data and metadata replication in order to replicate the metadata first. This makes PRACTI capable of doing *Partial Replication* by replicating the metadata of all of its data objects while replicating just a subset of the data itself. It maintains a flag in its metadata to indicate whether the data is *VALID* or *INVALID*. If it is *INVALID* and there is an attempt to read or write to an *INVALID* data object, then it goes through the process of blocking the request until it retrieves the data from other replicas. Once the data is retrieved, it marks the data object as *VALID* and allows the request to go through. This helps in getting the replicas to converge sooner because the metadata is negligible in size when compared to the actual object data size (Wang et al., 2012).

PRACTI also provides *Arbitrary Consistency* that allows a range of consistency guarantees to the caller by providing a control interface to specify the consistency requirements for the read and write operations. It provides a continuous range of consistency guarantees such as sequential consistency (Coulouris et al, 2011) or eventual consistency (Saito et al., 2006). For instance, a data read request will not block unless the caller specifies that it requires sequential consistency. The level of consistency would require the replica to gather more updates from other replicas before it can proceed with the read in order to ensure that the read operation satisfies the sequential consistency requirement. Similarly, all write requests will be applied right away unless the caller requests a sequential consistency.

PRACTI's *Partial Replication* and *Arbitrary Consistency* are provided in a *Topology Independent* environment where each replica can exchange its data and metadata with any

other replica. All these features provides a replication framework that can fit the needs of any large-scale replication system. However, PRACTI's range of consistency is lacking semi-locking as a feature that should improve the eventual consistency guarantee level. ORLease's semi-locking is provided as optimistically replicated leases that shorten the window of stale access which is one of the two dimensions of consistency guarantees (Yu et al., 2002). Semi-locking neither needs synchronization between replicas as in pessimistic replication nor it needs to gather more metadata from peer replicas in order to satisfy some session guaranteed consistency levels such as *Monotonic Read Consistency* (Terry et al., 1994)

ORLease Overview

The ORLease framework replicates data objects optimistically between multiple replicas where any replica is permitted to share updates with any other replica. Each participating replica has an object store and runs two services; a replication service and an object store frontend service. Both services are cooperating in order to manage the replicated object stores and is implemented as Windows platform executables ("Microsoft Windows Executable Files", n.d.). However, ORLease's framework is not limited to any specific implementation and can be incorporated into any replication framework that can separate and prioritize its metadata replication such as PRACTI (Belaramani et al., 2006).

The replication service manages replicating the object store updates to its peer replicas while the object store frontend manages uploading and downloading the data objects to and from the object store. Its object store frontend provides a local interface for applications to read, write, delete and lease the data objects as depicted in Figure 1 below. The application interface is similar to the PRACTI interface (Belaramani et al., 2006) that provides read, write and delete functionality for the data objects. However, ORLease also provides a lease interface for making explicit optimistic lease requests on its data objects. Additionally, it provides implicit optimistic leasing capabilities for the read and write operations on its data objects that can be configured based on the consistency requirements of the replication system.

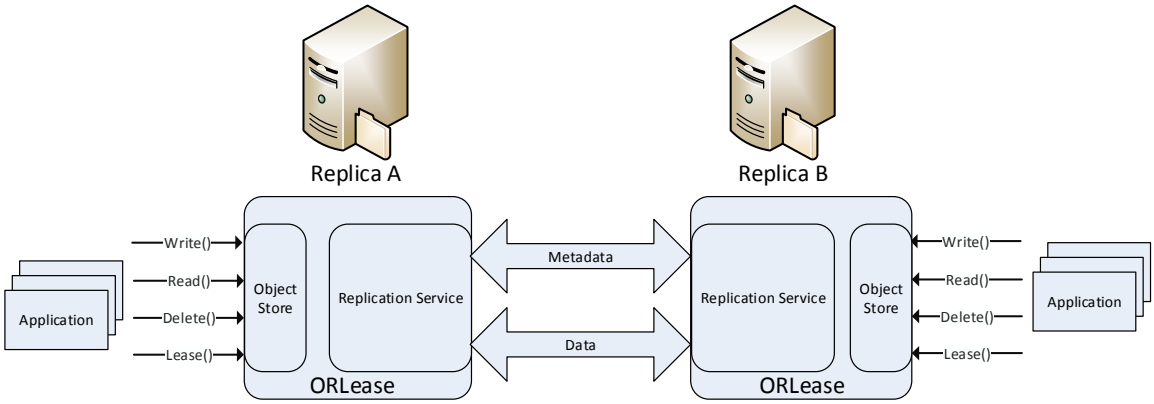

Figure 1

The ability to acquire optimistic leases on data objects, whether explicitly or implicitly, as a semi-locking mechanism distinguishes the ORLease framework from its optimistic replication framework counterparts. The lease functionality gives applications the ability to acquire a lease in an optimistically replicated fashion. If the data object is not marked in the replicated object store as leased by any other replica, then the replication service grants the lease request and sends it out in the form of a data object metadata update to other replicas as explained in more details later in the "Metadata" subsection. However, if the data object is marked as leased by another replica, then the replication service denies the request but then provides the calling application the ability to break the lease in case the lease owning replica is offline. This is explained in further details in the "Application Model" subsection.

ORLease Architecture

The ORLease framework is composed of a frontend object store service and a replication service as was depicted earlier in Figure 1. The frontend object store service communicates with the application through a well-defined interface as explained earlier. The object store will then relay the message to the replication service in the form of metadata through its metadata transport which is depicted in Figure 2 below as the *Store Metadata Transport*.

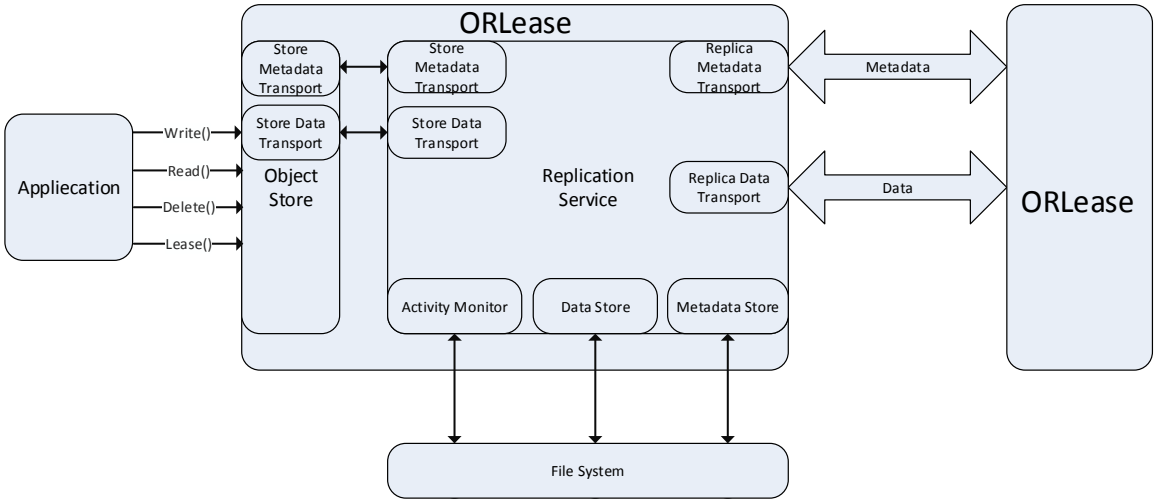

Figure 2

All data and metadata in ORLease are transferred through its transport layers. Each transport layer implements both the sender and receiver functionalities on each side of the communicating components. ORLease has two transport layers; one between the object store and the replication service and the other is between the replication service and its peer replication service. For instance, the *Store Data Transport* layer has two participating components; the object store and the replication service. If the object store is writing to an

object then the *Store Metadata Transport* on the object store side will act first as the metadata sender and the *Store Metadata Transport* on the replication service side will act as the receiver. Once the object's metadata is received by the replication service, the *Store Data Transport* on the object store side will act as the data sender and the *Store Data Transport* on the replication service side will act as the receiver.

Assuming that the application making a request to create a new object in the object store. The object store will then forward the request to the replication service through the *Store Metadata Transport* which will then check and update the replica's *Metadata Store.* Since the object is newly created and not leased, the *Metadata Store* will get updated and will allow the create operation to go through. This will trigger the *Replica Metadata Transport* to send out the lease request for this object to its peer replica. It will also create a thread ("Microsoft Windows Threads", n.d.) that will handle the data upload request through the *Store Data Transport* into a temporary location.

When the sending replica sends out the lease request through *Replica Metadata Transport*, the receiving *Replica Metadata Transport* adds the objects metadata to the *Metadata Store*. This prevents any application on the receiving replica from accessing the object because it is marked as leased. Once the object is fully uploaded on the sending replica, it then moves the object into its final destination in the replicated object store. The moving of the object into the replicated object store triggers the *Activity Monitor* to create a thread to handle the completion of uploading the object. This in turn updates the *Metadata Store* which will trigger the *Replica Metadata Transport* to send out metadata updates to its peer replica to release the leased object. Once the metadata is sent out, another thread

will be created in order to get the *Replica Data Transport* to transfer the data asynchronously from the data store to its peer replica.

When the sending replica sends out the release request for the leased object through the *Replica Metadata Transport*, the receiving replica will then remove the lease of the object from its *Metadata store* but will keep the object marked as not available because its data is not fully downloaded yet. Also, when the sending replica creates a thread to get the *Data Transport* to transfer the data asynchronously from the sending replica's data store to its peer replica, the receiving replica will in turn create a thread to receive the data. Once the data is fully downloaded and transferred into the receiving replica's replicated data store, the object will then be marked as available in its metadata store.

Metadata

The replication service provides a metadata store similar to PRACTI (Belaramani et al., 2006). The data store is used to store the objects' data while the metadata store is used to store the objects' versions, their lease versions, and their states. The data store organizes the objects' data as files on the file system while the objects' metadata should be stored separately in a database[1]. This gives the replication service control over the data access based on the metadata state.

There are two types of metadata that are associated with the actual data being replicated in optimistic replication systems; one describes the object's version while the other describes the replica's knowledge state (Parker et al., 1983). The former will be referred to as the data object's metadata and the latter as the replica's metadata. The data object's metadata represents the data object's version and its lease version at the replica that it is residing on while the replica's metadata represents a collection of all data objects' versions and their lease versions for a specific replica in the form of version vectors (Parker et al., 1983), (Saito et al., 2005) and (Wang et al., 2009).

Each object in the object store is represented with a metadata object in the metadata store and has the following data structure:

```
WCHAR   ObjectName[MAX_PATH];
DWORD   ReplicaId;
DWORD   Version;
DWORD   LeaseReplicaId;
DWORD   LeaseVersion;
BOOL    Lease;
BOOL    Available;
DWORD   Action;
HANDLE  Handle;
```

[1] The metadata store is currently stored in memory and not persisted in a database.

The ObjectName represents the name of the stored object while the ReplicaId and Version are both used to represent the object's version. Similarly the LeaseReplicaId and the LeaseVersion are both used to represent the object's lease version and owner. The Lease flag is used to identify whether the object is leased or not. For instance, assuming two replicas, replica A and replica B, and an application is running on replica A requested a lease on a data object named "F1". The lease request will replicate out to replica B and will be stored in its metadata store. The LeaseReplicaId will then be set to A, the lease version will be set to the version provided by replica A, and the Lease flag will be set to TRUE. If another application running on replica B requests a lease on "F1" or tries to write to it, it will get access denied from the replication service because the metadata store will indicate that it is already leased to replica A.

The Available flag is used to identify whether the object is *VALID* and available or not. This is similar to PRACTI's *VALID* flag and is set to false once the object is available to be updated or read. For instance, assuming two replicas, replica A and replica B, and an application is running on replica A leased a data object named "F1" to update it. Replica A will then send a metadata update that will set the Lease flag on replica B for object "F1" to TRUE. Once the write operation is complete on replica A, it will remove the lease and will send a metadata update to replica B which will set the Lease flag to FALSE and the *VALID* flag to FALSE because the object is not leased anymore but has been updated on replica A and the data did not reach replica B yet.

Data replication in optimistic replication system is downloaded asynchronously (Saito et al., 2005). Therefore, the Action and Handle variables are used as part of the replication service in order to keep track of the object activities and its asynchronous replication

mechanism with other replicas. The Action variable determines the current activity on the object and triggers the proper replication activity with other replicas while the Handle object is to manage the data replication with another replica. Once the object is downloaded by the peer replica, the Available flag will be set to TRUE and the handle will be reset.

When the ORLease's framework updates the data object metadata to reflect a data update or a lease request, it updates the replica metadata based on version vectors similar to other replication systems (Parker et al., 1983) and (Saito et al., 2005). For example, assuming two replicas A and B started to replicate with each other. If a data object "F1" is created on replica A, the object's metadata will be updated to have a new object for "F1" that has the following information:

```
ObjectName     = "F1";
ReplicaId      = A;
Version        = 1;
LeaseReplicaId = A;
LeaseVersion   = 1;
Lease          = TRUE;
Available      = FALSE;
Action         = OBJECT_ACTION_LEASE;
Handle         = INVALID_HANDLE_VALUE;
```

The replica ID is set to A and the version is going to be 1 since it is the first change on replica A. Similarly, the lease replica ID will be set to A and its version to 1. This implies an object metadata version of {A-1} and lease version of {A-1}; hence, the replica metadata will get assigned a VV of {A-1} and a lease VV of {A-1}. Replica A will then notify its peer replica B that it has a replica metadata update. This will trigger the metadata of object F1 to be replicated out to replica B and both its VV and lease VV will stay as {A-1}. However, its Available flag on B will be set to false until all of the data object's data is received from replica A. The object metadata on replica B will be initially set to be:

```
ObjectName     = "F1";
ReplicaId      = A;
Version        = 1;
LeaseReplicaId = A;
LeaseVersion   = 1;
Lease          = TRUE;
Available      = FALSE;
Action         = OBJECT_ACTION_LEASE;
Handle         = INVALID_HANDLE_VALUE;
```

Once the "F1" object is fully created on replica A, its metadata object will be updated to be the following:

```
ObjectName     = "F1";
ReplicaId      = A;
Version        = 1;
LeaseReplicaId = A;
LeaseVersion   = 2;
Lease          = FALSE;
Available      = TRUE;
Action         = OBJECT_ACTION_RELEASE;
Handle         = INVALID_HANDLE_VALUE;
```

This will implicitly update the replica metadata to have a lease VV of {A-2}; however, the object's VV stays the same as {A-1} because it has not changed. The Lease flag will be dropped to FALSE and Action will be set to OBJECT_ACTION_RELEASE since replica A is not leasing or accessing the object anymore. The Available flag will be set to TRUE since the object is fully created. Updating the replica metadata on A will trigger the exchange of VV with replica B. Replica B's metadata has the VV of {A-1} and a lease VV of {A-2}; hence, replica B will realize that it needs to get the lease change of {A-2}. The lease update indicates the release of the object's lease. Once replica B gets the metadata changes, it will release the object and update its metadata object to be:

```
ObjectName     = "F1";
ReplicaId      = A;
Version        = 1;
LeaseReplicaId = A;
LeaseVersion   = 2;
Lease          = FALSE;
```

```
Available       = FALSE;
Action          = OBJECT_ACTION_RELEASE;
Handle          = REPLICA_A_HANDLE;
```

Since the data is not yet fully downloaded on replica B, its Available flag will still be set to FALSE. However, replica B establishes a data connection with replica A in order to download the data locally. The connection is controlled by the handle value which is given the pseudo value above as Replica_A_Handle.

When "F1" is opened later to be modified on replica B, it will have a lease VV of {A-2, B-1} and replica B metadata is going have its lease version updated to {B-1}:

```
ObjectName      = "F1";
ReplicaId       = A;
Version         = 1;
LeaseReplicaId  = B;
LeaseVersion    = 1;
Lease           = TRUE;
Available       = FALSE;
Action          = OBJECT_ACTION_LEASE;
Handle          = REPLICA_A_HANDLE;
```

This will trigger metadata replication to replica A which will then lease the object to replica B. However, the application opening the data object will be blocked until F1 is fully downloaded. Once downloaded, its Available flag will be set to TRUE and the data handle will be set to INVALID_HANDLE_VALUE because it is fully downloaded and no need to keep a communication handle with replica A. At this point, the data object can then be modified by the application and the metadata will be set to the following:

```
ObjectName      = "F1";
ReplicaId       = A;
Version         = 1;
LeaseReplicaId  = B;
LeaseVersion    = 1;
Lease           = TRUE;
Available       = TRUE;
Action          = OBJECT_ACTION_LEASE;
Handle          = INVALID_HANDLE_VALUE;
```

Once the object is fully modified by the application, it will then release the object's lease. The object's version will be updated to {B-1} and the lease version will be updated to {B-2}. This implies that the VV will be updated to {A-1, B-1} and the lease VV will be updated to {A-2, B-2}. The object's metadata will also have the Lease flag set to FALSE and Action set to OBJECT_ACTION_RELEASE. The object's metadata will be updated to be as follow:

```
ObjectName     = "F1";
ReplicaId      = B;
Version        = 1;
LeaseReplicaId = B;
LeaseVersion   = 2;
Lease          = FALSE;
Available      = TRUE;
Action         = OBJECT_ACTION_RELEASE;
Handle         = INVALID_HANDLE_VALUE;
```

As demonstrated in the previous example, ORLease introduced two metadata version updates in order to propagate lease acquire and release. Once a lease acquire request is sent out, every replica receiving the request will honor the lease. The lease on the data object will stay indefinitely until the lease owner replica sends a lease release request. However, leases should not be held indefinitely on any data object in case the lease owner's replica gets disconnected from the other replicas and never connect back again.

Therefore, ORLease also introduced the concept of a third metadata version update in order to propagate a lease break request. This request is considered a lease acquisition request that is expected to introduce a lease conflict and a possible data conflict if the lease owner updates the object. This lease break request can be sent out by any replica in order to take the lease ownership of any leased object in its metadata store.

The lease acquisition, release and break process is depicted in Figure 3 with two replicas; replica A and replica B. When a data object is created on replica A, it gets assigned

a lease VV of {AL-1} and a data VV of {AD-1}; where the letter "L" and "D" are appended to the replica ID to just distinguish between a lease VV and a data VV respectively. The {AL-1} indicates a lease acquisition request and the {AD-1} indicates a data change which is the creation of a new object. Both VVs are replicated right away to replica B as metadata to ensure that replica B does not create a similar data object with the same name. Once replica A finishes updating the object's data, it creates a lease VV of {AL-2} that indicates a lease release request which will get replicated to replica B. The data will then be replicated asynchronously from replica A to replica B.

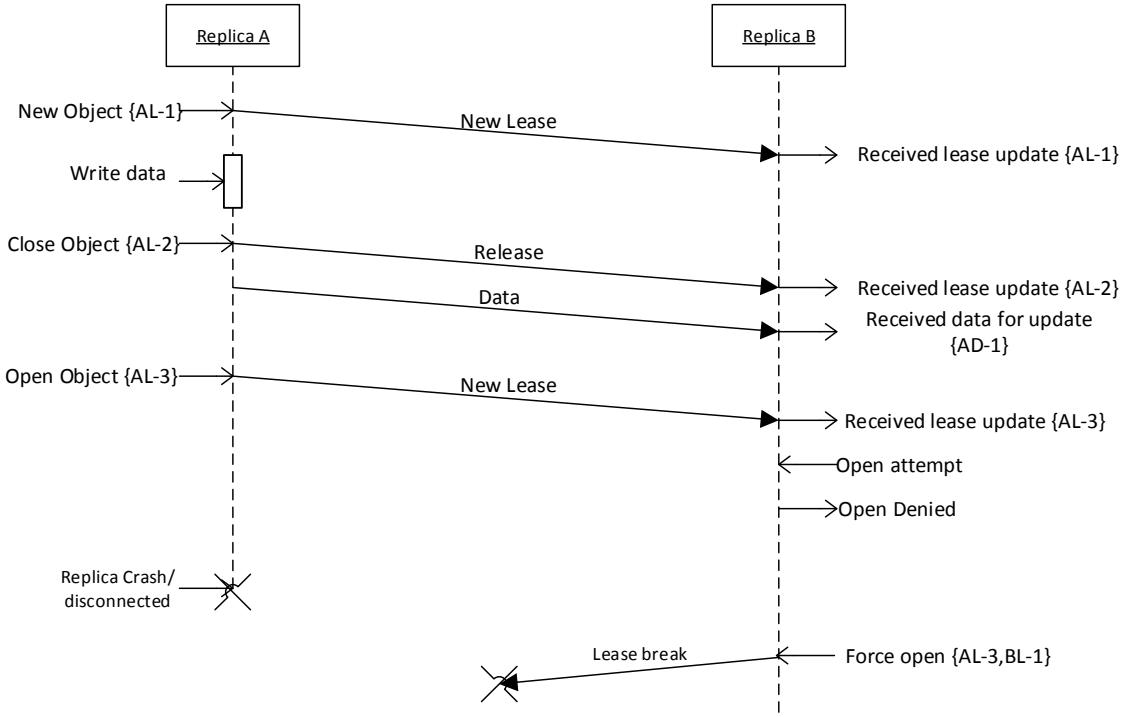

Figure 3

Assuming replica A will then open the object again which will create a lease VV of {AL-3} that indicates a lease acquisition request. Now if replica B decides to open the

same data object, its open request will get denied because of the lease request sent out by replica A as lease VV of {AL-3}. The VV of {AL-3} received by replica B should have updated its metadata store to indicate that object is leased for replica A. However, if replica A gets disconnected from the network, the availability of the object will go down due to the lease that is being held by its peer replica in response to the lease VV of {AL-3} sent out by replica A. Therefore, in order to avoid the situation of having the object being blocked indefinitely due to a disconnected replica, replica B can issue a lease break request. It updates the lease VV to be {AL-3, BL-1} and then proceed with accessing and updating the object.

Another lease break scenario is depicted in Figure 4 below between two replicas; A and B. In this example, replica A is still connected to replica B but starts updating the data object. This will generate a data VV of {AD-2}; however, replica B decides to break the lease and open the object for modification. Replica B sends a lease acquisition to replica A which will lead replica A to detect the request as a lease break. Replica A will then honor the lease break and update the lease owner to be replica B and will update its lease VV to be {AL-3, BL-1}.

Replica A then decides to close the object which will update the lease VV to {AL-4, BL-1}. When Replica B receives the lease VV update, it will just update its object lease metadata to be {AL-4, BL-1}. However, the object will still be leased to replica B since it forced breaking the lease with its explicit object lease VV update of {BL-1}. Replica A will then send out its object data update which has the data VV as {AD-2}; however, replica B will initially reject the update because the file is still opened by replica B. It will then

conflict with the data update on replica B which will have the data VV of {BD-1} after the object is closed.

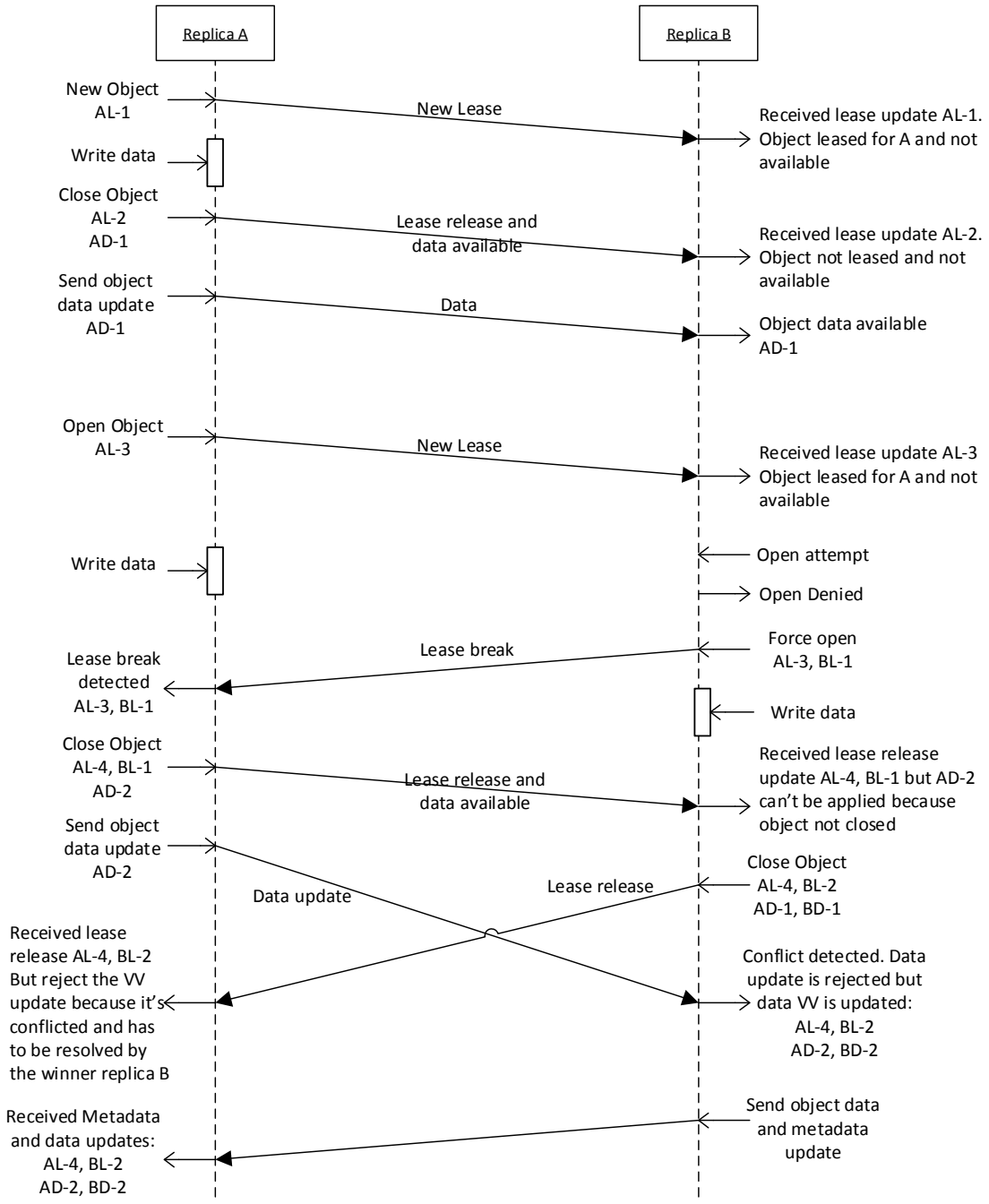

Figure 4

The conflict will be resolved and will result in replica B winning the conflict because replica B is the last object writer based on Thomas write rule (Thomas, 1979). The data VV will be updated to reflect the conflict reconciliation to ensure that the new VV is compatible and include all previous VV (Zhao, 2014). The data VV will also have replica B's VV count incremented by 1 from BD-1 to BD-2 so that its data VV represents a new version that will include the replica A's VV of {AD-2} and have its own data VV update of {BD-2} (Zhao, 2014). The new data VV will then be {AD-2, BD-2} which will dominate replica A's data VV of {AD-2, BD-1} and that will trigger data replication of the winning object from replica B to replica A.

Replica A should have also received replica B's data VV update of {BD-1} earlier before replica B detected and resolved the conflict. The conflict would have also been detected on replica A but will be not be reconciled because the winning replica is the only replica responsible for the object reconciliation. If more than one replica is reconciling the object, then the replication system could suffer from multiple conflicts because each reconciling replica is creating a new version of its own by incrementing its version counter (Zhao, 2014); hence, newly created data VV will conflict with each other resulting in more reconciliation and more conflicts.

Another conflicting scenario is depicted in Figure 5 below where two replicas A and B were forced to lease an object and make updates to an object because the replicas were disconnected. A data object is concurrently leased and updated by both Replica A and B but the network was down during the lease acquisition request, data update request and the lease release request. Therefore, each replica updated its replica knowledge according to its own changes while being unaware of the other replica's changes.

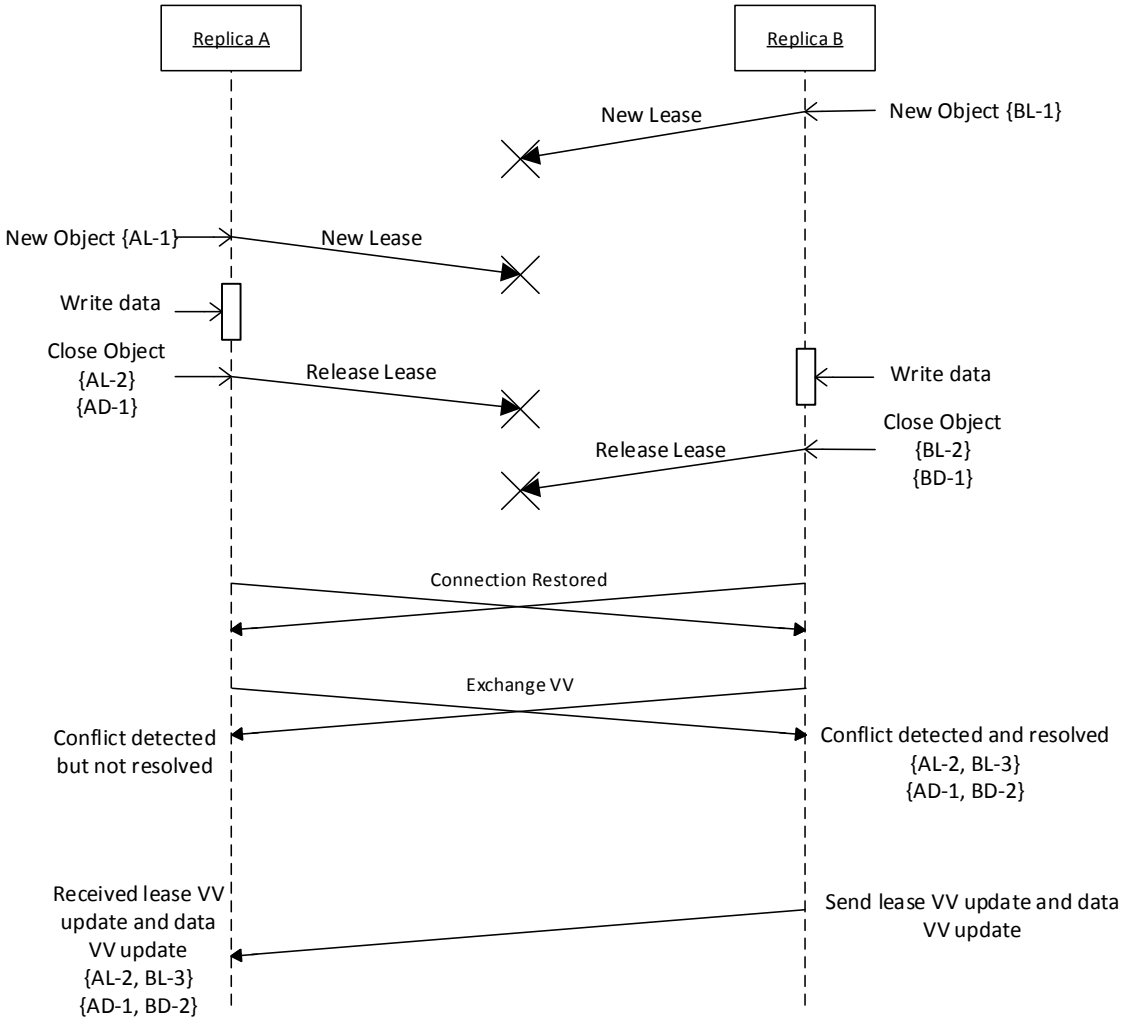

Figure 5

When both replicas get their network connection restored back and start to exchange their version vectors, they will both detect that the same data object has different lease and data versions leading to a detected conflict. Since the conflict resolution in ORLease is based on Thomas's write rule (Thomas, 1979), which is the last writer wins, the data object with the latest data VV update will win the conflict. For instance, in the flow diagram of

Figure 5, replica B closes the object and release the lease after replica A. Therefore, when network connectivity is restored between the replicas, the data object of replica A loses the conflict.

The conflict will be resolved on the winning replica which will generate a new version for both the lease VV and data VV as explained earlier. Therefore, the lease VV will become {AL-2, BL-3} instead of just {AL-2, BL-2} because the winning replica has to generate a new version (Zhao, 2014). Similarly, the data VV will become {AD-1, BD-2} instead of just {AD-1, BD-1}. The winning replica B will then send out the metadata update along with the data update to replica A. This would complete both replica reconciliation and replica A would then move its conflicting object out of the replication folder. The conflicting object can be preserved in a special location in the object store in case it should be accessed later for a manual merge or override by the end user.

ORLease Runtime Service

Once the replication service starts, it registers its *Replica Transport* layer for both data and metadata with its peer replicas' *Transport* layers. As explained earlier, the *Replica Transport* layer is used to transfer the data and metadata to and from its peer replica(s) asynchronously similar to PRACTI's core module functionality (Belaramani et al., 2006). The replication service then starts monitoring for two types of metadata updates; local updates and peer replica updates as depicted in the following flowchart in Figure 6.

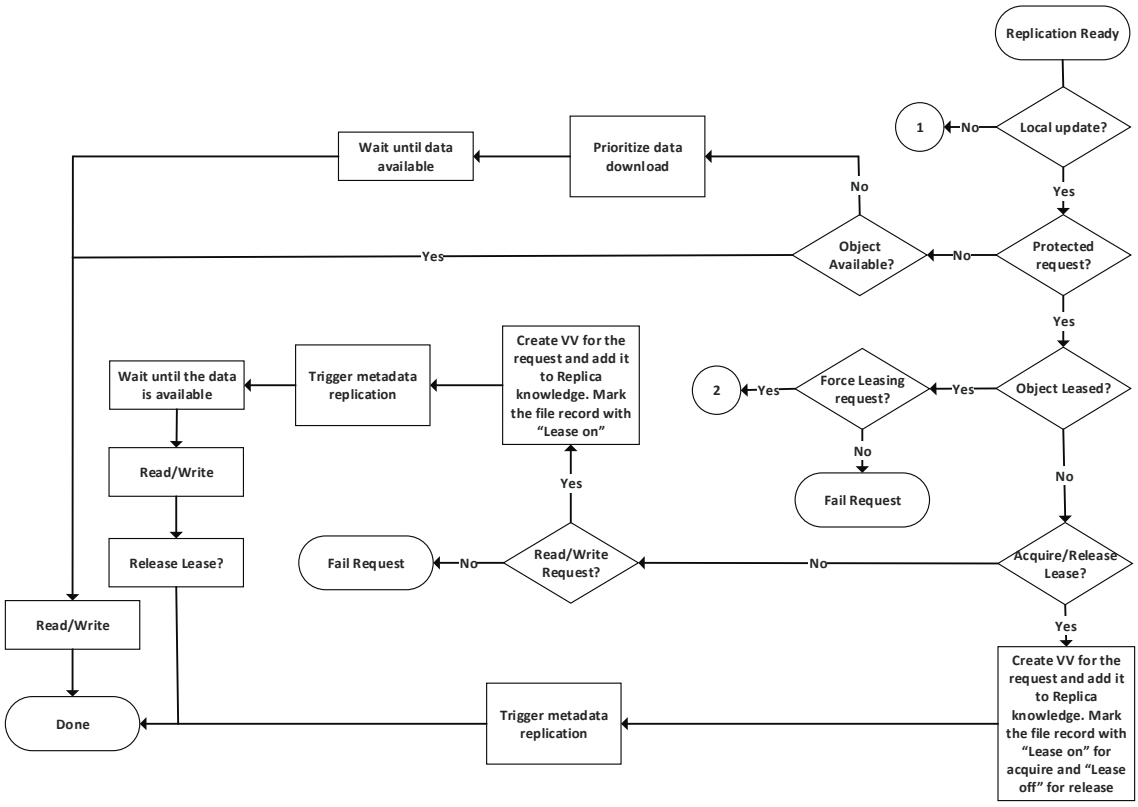

Figure 6

For the local updates, the service checks first whether it is a protected request that requires leasing or not. If it is not protected and the object is available, then the request goes through with data access. Otherwise, if the object is not available, then the request is

in a pending state until the data object is completely downloaded and available. However, if it is an optimistically protected read/write operation, the replication engine will have to check first whether the data object is leased by another replica or not. If it is leased, then the request is denied unless it is an explicit lease operation as depicted in Figure 7 below. Explicit lease requests are used to override an existing lease as explained earlier so that a data object is not locked indefinitely if the lease's owner replica is disconnected from the network since network partitioning is not uncommon in distributed systems.

Figure 7

If the object is not leased, then the requested object operation is checked. It can either be a lease/release operation or a protected read/write operation. Other operations are not supported in this research as mentioned earlier in the "Delimitations" section. If the requested operations is an optimistically protected read/write operation, then an implicit lease is acquired in order to optimistically lease the object while the read/write operation is in flight. Once the operation is complete, the lease is implicitly released as depicted in both Figure 6 and 7.

In addition to the implicitly requested leases during the protected read/write operations, the ORLease framework also provides explicit leasing. A client can request a lease on the data object and the replication engine will check whether the data object is leased and available or not. Whether the object is leased or not, the lease will be granted right away

and the data object's lease version vector will be updated in the metadata store triggering a metadata replication with the replica's peers. If the data object is not available, then any following read/write request will be in a pending state until the data object is completely downloaded and available.

For the second type of updates, which is the peer replica update, the replication service follows the flowchart path referenced by the circled number 1 as depicted in Figure 6. The continuation of this flowchart is depicted in Figure 8 and represents two types of peer replica metadata update; data VV update and lease VV update. For the first type of peer metadata update, which is data VV update, the replication service compares the received data VV in order to check whether the object is currently in a conflict or not. If there is no conflict detected, then the local object's data VV will be updated which in turn will trigger the object's data to be replicated if data VV difference is due to a data update. However, if a conflict is detected, then it will be reported and resolved only on the replica that most recently updated the object as explained earlier. A conflict could have occurred due to concurrent optimistic lease requests and write operations by multiple replicas when the replicas were disconnected.

For the second type of peer metadata update, which is lease VV update, a lease request can be a new lease, release lease or an override lease request that got propagated from peer replica. All lease operations add the lease VV of the request to the replica knowledge and take the proper actions based on the lease operation type. The new lease request; whether it is an explicit lease request or an implicit request acquired during an optimistically protected read or write operation, will set the metadata Lease flag to TRUE and the LeaseReplicaId to the Id of the replica owner of the lease. The lease release request does

the opposite as it sets the metadata Lease flag to FALSE and reset the LeaseReplicaId to NULL since the object is not leased and no replica owns any lease on the object. Finally, the lease break request keeps the Lease flag set to TRUE but changes the LeaseReplicaId to the Id of the last lease replica owner set in the metadata request.

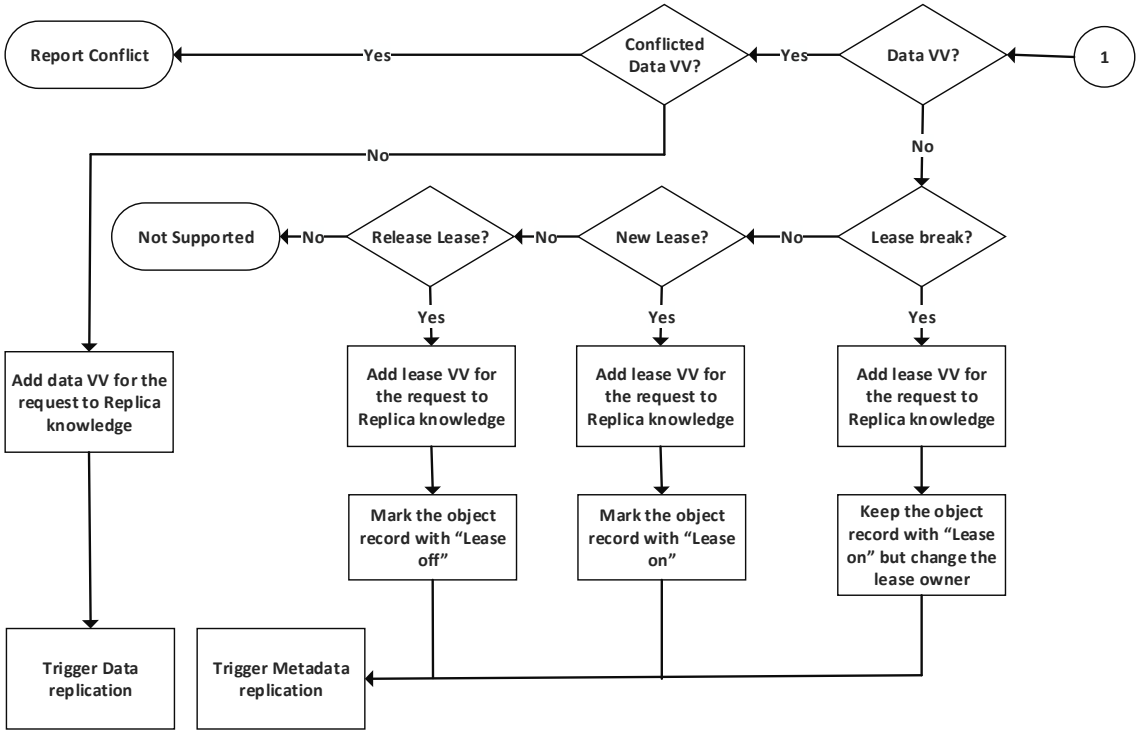

Figure 8

Application Model

The ORLease framework provides a programming model which empowers the application developers to handle conflicts in their replicated data stores. This is similar to MDCC or Multi-Data Center Consistency (Kraska, Pang, Franklin, Madden, and Fekete, 2013) which is a wide area replication system that also provided a programming model to handle long and unpredictable latencies caused by the inter-data center communication. The interface provides read, write and delete methods for reading and modifying the data. This is similar to the PRACTI interface (Belaramani et al., 2006); however, ORLease's methods acquires implicit leases and the interface also provide an extra lease method. The lease method can be used by the applications in order to acquire an explicit lease on the data object that is being replicated.

The write and delete methods provide an implicit lease functionality as explained earlier in order to achieve the semi locking semantics for the replicated data. If the object is leased by another replica, these operations will fail and the application will get an error that the object is already leased. However, the application can explicitly lease an object to override the leased object. The application developers will need to show the proper error message to the end user when a write or delete operation fails. They could also provide the proper methodology that allows the end user to issue an explicit lease on the leased object in order to break the existing lease and complete the write or delete operation.

Therefore, conflicts in ORLease will not be uncommon due to the optimistic nature of its replication framework. Conflicts will still occur due to disconnected replicas, explicit lease override as well as the concurrent lease acquisition. In order to mitigate this issue, an application can also register to receive notifications when conflicts are detected at the lease

level as explained earlier. Conflicts in PRACTI (Belaramani et al., 2006) are detected when the metadata is exchanged after the data object has been updated. However, ORLease can detect conflicts even before data is committed because conflicts can be detected when lease metadata requests are exchanged.

For instance, assuming two replicas A and B are issuing a write operation with an implicit lease on the same object at the same time. The lease requests will result in a conflict that will be detected on both replicas. When a conflict is detected, the application developer could raise a warning to the end users on both replicas. The warning could identify the conflicting replicas since the LeaseReplicaId in the metadata store is holding this information. This would give the end users the proper information to the end users in order to communicate with each other and orchestrate a conflict resolution.

Chapter 4

Results

Testing and Evaluation

The ORLease framework has been evaluated based on the modified *stale-access* metrics (Kuenning et al., 1998) as explained earlier. The *leased stale-access* metric is the difference in time when a replica receives a lease for an object that has been leased by another replica, and when it receives a lease release or a lease break. The *stale-access* has been evaluated based on an external (client-centric) or internal (data-centric) methodology (Kuenning et al., 1998) and (Bermbach et al., 2013). The external evaluation methodology is achieved by using external data writers and readers in order to detect the propagation time of individual updates between replicas. It is leveraged in cases where it is hard to have access to the source code or its detailed logging as in the evaluation of Amazon's S3 ("Amazon S3", n.d.) that was attempted by Bermbach et al. (Bermbach and Tai, 2014).

On the contrary, the internal evaluation methodology is leveraged when source code is accessible or detailed logging is available. It is based on calculating the difference of timestamps logged by replicas for propagated updates in order to detect the propagation time. This is the methodology that ORLease followed since the evaluation of staleness has been extracted from the logs generated and displayed in the command window by the replication services and the frontend applications communicating with them. For instance, when an object (obj1) is added to one replica (replica A), the logs shows the time when the object is initially added to the replication store and completely uploaded:

```
The object (obj1) metadata is added at: 19:52:16.316
```

```
..
The object (c:\RF1\obj1) data is added at: 19:52:16.414
```

Then the log for the remote replica will display when the remote replica got the object's lease, the object's metadata and when the object was completely downloaded:

```
The object (obj1) is leased and metadata is added at: 19:52:16.320
..
The object (obj1) metadata is updated at: 19:52:16.430
..
The object (c:\RF2\obj1) data is added at: 19:52:16.555
```

ORLease's prototype has been implemented in order to evaluate ORLease and compare its results against other common replication techniques that have been simulated by the same prototype. The prototype provided the proper logging that showed the timestamps at which both the source replica A and the destination replica B are reacting to different events as shown in the previous logging snippets. For instance, the logging of the destination replica B showed different timestamps at which a lease is applied, replicated object's metadata is received and replicated object's data is received.

These three logging events were very crucial in evaluating ORLease because the first one shows the timestamp when the destination replica B receives and applies the lease metadata for a potentially replicated object. This lease metadata is used by ORLease in order to block the receiving replicas from accessing the object; hence reducing the *stale-access* time window. The second logging event indicates the receiving of the object's creation or update metadata. That metadata is used by PRACTI (Belaramani et al., 2006) in order to determine if an object is created or updated. It also reduces the *stale-access* time window because the receiving replicas will block object access until the object's data is fully downloaded. The third logging event indicates the completion of the object's data

download to the destination replica B. This is used by most replication systems (Saito et al., 2005) to indicate that an object has been created or updated and ready to be accessed.

As a measure of success, the optimistic replication system that is enhanced with the optimistic lease technique showed a reduction in the *stale-access* time window by the *leased stale-access* time. This is due to the fact that ORLease will allow replicas to maintain an optimistically consistent view of an object's metadata using the optimistic lease's metadata but without necessarily having other object's replication metadata or its data contents available. Additionally, the semi-locking lease ensures that the leasing replica will optimistically have a lease to the object even before the data is modified. This is what gives ORLease an edge over other replication systems that replicate metadata after the objects are modified (Saito et al., 2005) and (Belaramani et al., 2006). However, if the source replica fails or aborts the object update or creation, then the replication system would have incurred an extra metadata request for leasing that is unnecessary.

In the previous logging snippets, the *stale access* period is the difference in time between the time when obj1 got uploaded and added to replica 1 and the time when it is fully downloaded and received by replica B which is 19:52:16.316 – 19:52:16.555 = 239 millisecond. However, the *leased stale access* period is the difference in time between the time when obj1 lease is received by replica B and the time when it is fully downloaded which is 19:52:16.320 – 19:52:16.555 = 235 millisecond. The difference in time between *stale access* and *leased stale access* is just 4 milliseconds. Therefore, a conflict will only occur if replica B adds an object that is named (obj1) during this time window.

PRACTI improved the consistency by reducing the window of inconsistency to be the difference in time between receiving the object metadata on replica B after it is committed

on replica A and when the data is received which is 19:52:16.430 – 19:52:16.555 = 125 millisecond. Therefore, by comparing ORLease to PRACTI, the *stale-access* window for introducing conflicts will be reduced from 239 milliseconds in common replication systems such as Coda (Kumar et al., 1995) to 125 milliseconds in systems such as PRACTI (Belaramani et al., 2006) to just 4 millisecond in ORLease.

Experiments

To evaluate the efficacy of ORLease, the proper experiments have been conducted in order to evaluate the *stale access* of replicated objects for different object sizes. ORLease's results have been compared against the expected results of replication systems that keep their replicated objects stale until the objects are committed. Once an object is committed on any replica, it is replicated to its peer replicas and the object will still be stale until the whole object's data is available at the peer replicas in systems such as Coda (Kumar et al., 1995) or just until the object's metadata is available at the peer replicas in systems such as PRACTI (Belaramani et al., 2006).

The evaluation was conducted on a single machine that has been configured to have two replication folders; RF1 and RF2. Each replication folder had a replication service instance that is responsible of monitoring changes in its replication folder. Each replication folder also had a store service that communicates with the replication service acting as a client that uploads objects to the replication folder. When the store service uploads a file to the replication folder, it does that by making a request to the replication service which in turn issues a lease request to its peer replication service. Once the lease is received by the peer replica, the object is considered leased and locked for the originating replica until the object is fully replicated and the lease is dropped.

The replication service is a prototype that has been developed in order to evaluate the expected *stale access* of replicated objects in ORLease, PRACTI (Belaramani et al., 2006) and other common replication systems such as Ficus (Reiher et al., 1994) and Coda (Kumar et al., 1995). The prototype is capable of mimicking the different replication approaches while emitting the proper logging information for evaluation as mentioned earlier. It is

capable of communicating with other instances of the replication service as well as clients that create and modify different objects.

The goal of the experiments was to demonstrate ORLease's reduction in *stale access* which implicitly lowers the replication system's conflict rate (Saito et al., 2005). The experiments were conducted in the absence of network partition since ORLease systems do not function when replicas are disconnected from the network. Nevertheless, ORLease's behavior due to network partitioning faults and its recovery by breaking leases was left for future work.

ORLease's experimental results were in line with the expectations. It was expected that ORLease will have a smaller constant stale window access when compared to other systems such as PRACTI (Belaramani et al., 2006) and Coda (Kumar et al., 1995). Its metadata propagation latency results were almost constant and negligible for different objects' sizes. This is due to the fact that ORLease sends the lease metadata right when the object is created or opened for update and the metadata is negligible in size when compared to the data size (Wang et al., 2012).

On the contrary, PRACTI (Belaramani et al., 2006) sends its metadata right after the object is committed on the sending replica while Coda (Kumar et al., 1995) does that when the object is committed at the receiving replica. Therefore, both PRACTI (Belaramani et al., 2006) and Coda (Kumar et al., 1995) depends on the object's committing time which can take hours or even days depending on the users as in DFSR ("Microsoft Distributed File System Replication (DFSR)", n.d.). In addition, replication systems such as DFSR depends on the data propagation latency which grows proportionally relative to the size of the object's update.

Consequently, few experiments have been conducted using the ORLease prototype in order to evaluate the data and metadata propagation latency for newly created objects. The setup for this experiment had 2 replicas (replica R1 and replica R2) running on the same machine as 2 different processes. Then different objects sizes (1k, 100k, 1MB, 4MB, 10MB, 113MB, 1GB, 2.4GB, and 14GB) have been uploaded from the frontend application to replica 1 which in turn replicated them to replica 2.

Based on the conducted experiments, the results have been reported in table 1 below. The table rows represent the different object sizes while the table columns represents collected log data as well as the calculated *stale access* window for ORLease, systems similar to PRACTI (Belaramani et al., 2006) and systems similar to Coda (Kumar et al., 1995). The first two columns represents the timestamps from the logs for when the object was opened or created and then closed on replica 1 (R1). The following three columns represents the timestamps from the logs for when the object is leased, has its metadata added and data added on replica 2 (R2).

Regardless of the object size being uploaded to the ORLease replica, the metadata propagation time stayed constant as expected. The results are also depicted on two charts; one for small objects as shown in Figure 9 and the other for medium to large objects as shown in Figure 10. The results for all object sizes could be presented on one chart but the chart was not clearly showing the difference between the stale access of ORLease, PRACTI and regular replication systems for small objects.

Object size	Open object time (R1) mm:ss.ms	Close object time (R1) mm:ss.ms	Lease Added (R2) mm:ss.ms	Metadata Added (R2) mm:ss.ms	Full data received (R2) mm:ss.ms	ORLease Stale Access mm:ss.ms	PRACTI Stale Access mm:ss.ms	Stale Access mm:ss.ms
1 KB	13:15.537	13:15.566	13:15.549	13:15.569	13:15.574	00:00.12	00:00.32	00:00.37
100 KB	19:50.790	19:50.856	19:50.804	19:50.858	19:50.914	00:00.14	00:00.72	00:00.124
1 MB	52:10.061	52:10.158	52:10.076	52:10.161	52:10.224	00:00.15	00:00.100	00:00.163
4 MB	18:50.950	18:51.095	18:50.959	18:51.096	18:51.160	00:00.9	00:00.146	00:00.210
10 MB	21:31.485	21:32.388	21:31.500	21:32.397	21:32.548	00:00.15	00:00.912	00:01.063
133 MB	25:21.467	25:23.288	25:21.479	25:23.294	25:24.693	00:00.12	00:01.827	00:03.226
1 GB	29:33.271	29:42.367	29:33.282	29:42.368	29:46.922	00:00.11	00:09.097	00:13.651
2.4 GB	33:11.042	33:33.098	33:11.060	33:33.099	33:56.368	00:00.18	00:22.057	00:45.326
14 GB	37:41.403	40:05.101	37:41.410	40:05.101	42:40.461	00:00.7	02:23.698	04:59.058

Table 1

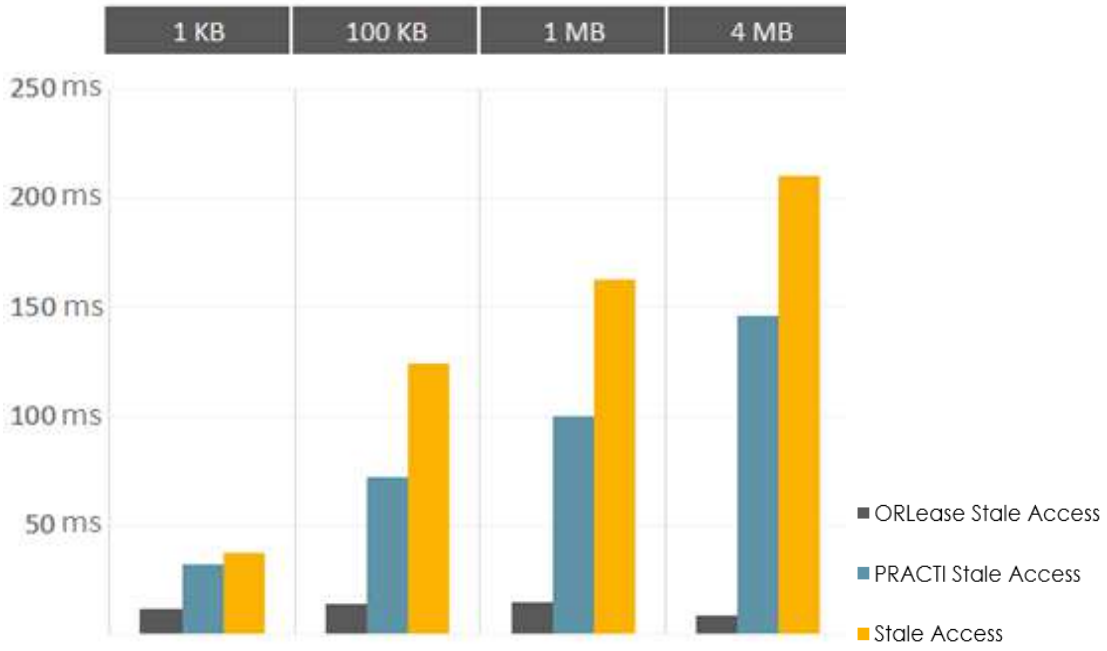

Figure 9

Figure 10

Chapter 5

Conclusions, Implications, Recommendations and Summary

Conclusions

Optimistic replication systems that are enhanced with ORLease's optimistic lease technique demonstrate a reduction in the *stale-access* time window by the *leased stale-access* time. It has been demonstrated that ORLease's *stale-access* doesn't depend on the replicated object size and the time taken to commit an object similar to PRACTI (Belaramani et al., 2006) and Coda (Kumar et al., 1995). ORLease's evaluation showed constant time in stale access that only depends on the network latency and its connectivity between replicas.

Consequently, ORLease is about conflict reduction and not conflict prevention because it provides a best effort leasing and requires a well-connected network between its participating replicas. Therefore, conflicts will occur if the network latency between two replicas of a replication system is T milliseconds and the time difference of opening an object on both replicas is less than T. Conflicts will also occur if the network is partitioned and same objects are modified on multiple replicas at the same time. Conflicts should then be resolved either manually or automatically as demonstrated in the optimistic replication survey by Saito and Shapiro (Saito et al., 2005).

ORLease's current implementation had few simplifications in order to expedite its implementation and attenuate the evaluation process. These simplifications did not influence ORLease's expected successful outcome. For instance, ORLease's prototype was based on a flat namespace with no directories or subdirectories similar to the simulation

framework developed by Wang et al. (Wang, Reiher, and Bagrodia, 1997). Their system was also based on a flat namespace instead of a hierarchical one in order to simplify their prototype implementation and its evaluation process.

Another simplification was to have ORLease's prototype operations limited to just the object creation operation. The optimistically replicated lease is considered a semi-locking operation that precedes all object's operations and is applied similarly to all of them; hence, evaluating one operation should suffice. The objects were also leased, created, and replicated in their whole entirety. The lease is then released when the object is fully created and replicated without depending on the metadata replication prioritization to do partial replication on demand. These simplifications were necessary in order to accelerate the implementation process and they are left for future research.

Implications

ORLease introduced an innovative technique to achieve semi-locking of data objects in optimistic replication systems by leveraging the existing system's communication methodology and causality capturing techniques. It has been demonstrated that the object size and its commit time did not have an impact on staleness in optimistic replication systems that are enhanced with ORLease. Network latency and connectivity are the only factors that affects staleness in these systems.

Optimistic replication systems that are augmented with the optimistic lease technique should reduce their window of *stale-access* for data objects by the *leased stale-access* time that was introduced with ORLease's lease acquisition. PRACTI (Belaramani et al., 2006) was successful in reducing the *stale-access* by replicating the metadata of the update once the object is committed. ORLease has reduced the window of *stale-access* even further by replicating the metadata for optimistic leases when the replica opens a data object with the intention to create it or update it.

However, ORLease's effectiveness in reducing stale access is better demonstrated in replication systems that have their data objects updated over a large window of time such as the replication of distributed file systems' files in DFSR ("Microsoft Distributed File System Replication (DFSR)", n.d.) and Azure File Sync ("Microsoft Azure File Sync (AFS)", n.d.). Files in DFSR and AFS can be opened for hours or even days before changes are committed and replicated.

Recommendations

This research focuses on leveraging the newly introduced technique ORLease in optimistic replication systems in order to enhance their consistency. The focal point is to optimistically broadcast the lease requests of replicated objects in these replication systems through their metadata replication mechanism in order to shorten their *window of inconsistency* (Bailis and Ghodsi, 2013). Therefore, a few simplifications have been undertaken in order to expedite the research outcome and its evaluation process.

ORLease's implementation was limited to handle object creation due to time constraints but not for technical reasons. It was also based on a flat namespace instead of a hierarchical one in order to simplify the prototype implementation and its evaluation process. Therefore, a more extensive study is required in order to validate ORLease's functionality for all file operations in a multi-master distributed replication system. The study should also evaluate hierarchical namespaces and the move operations of files and folders between different folders.

The prototype was also implemented as a very primitive platform executable ("Microsoft Windows Executable Files", n.d.) that leveraged named pipes ("Microsoft Windows Named Pipes", n.d.) as the data transport layer. It utilized the Windows directory management functions ("Microsoft NTFS file system directory management functions", n.d.) in order to detect the file system modifications. Therefore, other technologies are also worth the investigation like using RPC ("Microsoft Windows Remote Procedure Call (RPC)", n.d.) instead of named pipes and leveraging the Update Sequence Number (USN) change journal of the Microsoft NTFS file system ("Microsoft NTFS file system Update Sequence Number (USN) change journal", n.d.) instead of using the Microsoft directory

management functions ("Microsoft NTFS file system directory management functions", n.d.).

ORLease should be capable of handling objects that are marked as incomplete similar to PRACTI (Belaramani et al., 2006) because of the metadata replication prioritization. Incomplete objects can still be optimistically leased by the replicas that have them marked as incomplete. The optimistically replicated lease request can then trigger data replication prioritization from replicas that have the complete data objects. However, this was considered an enhancement that has not been evaluated and should be considered for future research. Therefore, this research has implemented and evaluated ORLease based on holding the lease on the object until the data is fully downloaded and available. It then releases the lease instead of depending on the data replication prioritization to block any new incoming request for any object until it is fully downloaded and available.

Another area that has not been explored is the breaking of leases held by disconnected replicas. Disconnected replicas are key aspect of optimistic replication systems (Saito et al., 2005). For instance, Coda (Satyanarayanan, & Kistler, 1990) focused on the disconnected replicas and their reconciliation with the replication system. ORLease's semi-locking mechanism is in the form of an infinite lease that is optimistically replicated out from the lease requesting replica to all other participating replicas. The leasing mechanism introduced the concept of a lease break request so that any replica can attempt to break the lease and ensure that any leased object is not locked indefinitely in case the lease owner's replica gets disconnected. However, the impact of disconnected replicas and side effect of breaking leases on ORLease is a pivotal topic that should be addressed in future research.

Consequently, the study of disconnected replicas and its side effect on the lease behavior should pave the way for further research around the application model of applications that are interacting with ORLease. The application model should encompass the handling of conflicts in their replicated data stores and the lease break operations. Other replication systems have studied application models for their system such as MDCC or Multi-Data Center Consistency (Kraska et al., 2013). MDCC provided a programming model to handle long and unpredictable latencies caused by the inter-data center communication.

Summary

Optimistic replication favors high availability for its replicas at the cost of stale reads and potential conflicts that could occur due to concurrent update requests from different replicas for the same data object. The conflicts happen due to the lack of a mutual exclusion mechanism between replicas to serialize the update requests. However, enforcing mutual exclusion will defeat the purpose of optimistic replication because it either requires synchronization between replicas similar to Paxos protocol (Lamport, 1998) or communicating with a resource locking entities similar to Yahoo's ZooKeeper (Hunt et al., 2010) which are key aspects of pessimistic replication.

ORLease introduced a semi-locking mechanism between the different replicas. It is an extension to optimistic replication systems where conflicts are not uncommon. ORLease's methodology is based on optimistic concurrency (Kung and Robinson, 1981) and leasing (Cary and David, 1989) in order to allow replicas to have semi-mutual exclusive access to their data objects. It is leveraging the aggressive propagation methodology of Pangaea (Saito et al., 2002) in order to accelerate the lease requests to all replicas. It also leverages logical clocks (Lamport, 1978) in order to exchange the lease metadata between replicas similar to the exchanged metadata of optimistic replication system to reconcile their data objects.

ORLease's semi-locking requests are piggybacked on the already existing metadata exchanging mechanism of the optimistic replication system. This semi-locking mechanism is in the form of an infinite lease that is optimistically replicated out from the lease requestor replica to all other participating replicas. The leasing mechanism also introduces a lease break request so that any replica can attempt to break the lease and ensure that any

leased object is not locked indefinitely in case the lease owner's replica gets disconnected or becomes unavailable.

The semi-locking mechanism does not completely eliminate conflicts since the lock acquisition of any data object is not synchronized between replicas. It also requires replicas to be fully connected and the metadata to be instantaneously replicated in order to reduce the possibility of introducing conflicts. A conflict will still occur if two lease requests are placed by two different replicas for the same data object within a period of time that is smaller than the time it takes to propagate and apply the lease request from one of the lease requesting replicas to the other. Therefore, a conflict resolution mechanism, whether it's done manually or automatically (Parker et al., 1983), is still required in order to resolve conflicts that might occur.

ORLease's framework is not limited to a specific implementation and can be incorporated into any replication framework that can separate and prioritize its metadata replication such as PRACTI (Belaramani et al., 2006). Its framework is capable of optimistically locking data objects while the replication framework is replicating the data objects optimistically between multiple replicas. It is also topology independent where any replica is permitted to share updates with other replicas while maintaining optimistic locks during updates.

The current ORLease implementation has each participating replica configured with an object store and running two services; a replication service and an object store frontend service. Both services are cooperating in order to manage the replicated object stores and is implemented as Windows platform executables ("Microsoft Windows Executable Files", n.d.). The replication service manages replicating the object store updates to its peer

replicas while the object store frontend manages uploading and downloading the data objects to and from the object store. Its object store frontend provides an interface similar to the PRACTI interface (Belaramani et al., 2006) that provides read, write and delete functionality for the data objects. However, ORLease also provides a lease interface for making explicit optimistic lease requests on its data objects. Additionally, it provides implicit optimistic leasing capabilities for the read and write operations on its data objects that can be configured based on the consistency requirements of the replication system.

ORLease's experimental results showed smaller constant stale window access when compared to other systems such as PRACTI (Belaramani et al., 2006) and Coda (Kumar et al., 1995). PRACTI (Belaramani et al., 2006) sends its metadata right after the object is committed on the sending replica while Coda (Kumar et al., 1995) does that when the object is committed at the receiving replica. Therefore, both PRACTI (Belaramani et al., 2006) and Coda (Kumar et al., 1995) depends on the object's committing time which can take a long period of time depending on the replication system.

References

Ahamad, M., Neiger, G., Burns, J., Kohli, P., and Hutto P. (1995). Causal memory: Definitions, implementation, and programming. Distributed Computing, Volume 9 Issue 1, pp. 37-49.

Amazon S3 (n.d.). https://aws.amazon.com/s3

Bailis, P. and Ghodsi, A. (2013). Eventual Consistency today: Limitations, extensions, and beyond. Communications of the ACM Journal, Volume 11 Issue 3, pp. 55-63

Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Leon, J., Li, Y., Lloyd, A., and Yushprakh, V. (2011). Megastore: providing scalable, highly available storage for interactive services. In Proceedings of the Conference on Innovative Data System Research (CIDR), pp. 223-234.

Baker, M., Hartmart, J., Kupfer, M., Shirriff, K., and Ousterhout, J. (1991). Measurements of a distributed data object system. In Proceedings of the Symposium on Operating Systems Principles (SOSP), pp. 198-212.

Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., and Zheng, J. (2006). PRACTI replication. In Proceedings of the Third Conference on Networked Systems Design and Implementation (NSDI), Volume 3, pp. 5-5.

Bermbach, D. and Kuhlenkamp, J. (2013). Consistency in distributed storage systems: An overview of models, metrics and measurement approaches. In Proceedings of the First International Conference on Networked Systems (NETYS), Springer (2013), pp. 175-189.

Bermbach, D. and Tai, S. (2014). Benchmarking eventual consistency: lessons learned from long-term experimental studies. In Proceedings of the Second International Conference on Cloud Engineering (IC2E), IEEE (2014), pp. 47-56.

Bessani, A., Mendes, R., Oliveira, T., Neves, N., Correia, M., Pasin, M., and Verissimo, P. (2014). SCFS: a shared cloud-backed file system. In Proceedings of the 2014 USENIX Annual Technical Conference (ATC), pp. 169-180.

Bolosky, W., Bradshaw, D., Haagens, R., Kusters, N., and Li, P. (2011). Paxos replicated state machines as the basis of a high-performance data store. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 141-154.

Brewer, E. (2000). Towards robust distributed systems. (Invited Talk) Principles of Distributed Computing (PODC), Portland, Oregon, July 2000.

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), pp. 335-350.

Cary, G. and David C. (1989). Leases: an efficient fault-tolerant mechanism for distributed data object cache consistency. In proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP), pp. 202-210.

Chang, C., Sun, J., and Chen, H. (2016). Coral: A Cloud-Backed Frugal File System. IEEE Transactions on Parallel and Distributed Systems, Volume 27 Issue 4, pp. 978-991.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H. A., Puz, N., Weaver, D., and Yerneni, R. (2008). PNUTS: Yahoo!'s hosted data

serving platform. In Proceedings of the VLDB Endowment, Volume 1 Issue 2, pp. 1277-1288.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). Distributed Systems: Concepts and Design. Fifth Edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011. ISBN: 0132143011.

Davidson, S., Garcia-Molina, H., and Skeen, D. (1985). Consistency in partitioned networks. ACM Computing Surveys (CSUR), Volume 17 Issue 3, pp. 341-370.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pp. 205-220.

Demers, A. J., Greene, D. H., Hauser, C., Irish, W., and Larson, J. (1987). Epidemic algorithms for replicated database maintenance. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 1-12.

Demmer, M., Du, B., and Brewer, E. (2008). Tierstore: a distributed filesystem for challenged networks in developing regions. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), Article No. 3, pp. 32-38.

Ellard, D., Ledlie, J., Malkani, P., Seltzer, M. (2003). Passive NFS tracing of email and research workloads. In Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST), pp. 203-216.

Gray, J., Helland, P., O'Neil, P., Shasha, D. (1996). The dangers of replication and a solution. In Proceedings of the ACM SlGMOD Conference, pp. 173-182.

Heidemann, J., Goel, A., and Popek G. (1995). Defining and measuring conflicts in optimistic replication. Technical Report UCLA-CSD-950033, University of California, Los Angeles, Sept. 1995.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. In ACM Transactions on Programming Languages and Systems, Volume 12 No. 3, pp. 463-492.

Hunt, P., Konar, M., Junqueira, F., and Reed, B. (2010). Zookeeper: Wait-free coordination for Internet scale services. In Proceedings of the 2010 USENIX Annual Technical Conference (ATC), pp. 145-158.

Kawell, JR. L., Beckhart, S., Halvorsen, T., Ozzie, R., and Greif, I. (1988). Replicated document management in a group communication system. In Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW), pp. 395.

Kraska, T., Pang, G., Franklin, M., Madden, S., Fekete, A. (2013). MDCC: Multi-data center consistency. In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 113-126.

Kuenning, G., Bagrodia, R., Guy, R., Popek, G., Reiher, P., Wang, A. (1998). Measuring the Quality of Service of Optimistic Replication. In Proceedings of the European Conference on Object Oriented Programming (ECOOP) Workshop on Object-Oriented Technology, pp. 319-320.

Kumar, P. & Satyanarayanan, M. (1995). Flexible and safe resolution of file conflicts. In Proceedings of the USENIX Technical Conference (TCON), pp. 8-8.

Kung, H. and Robinson, J. (1981). On optimistic methods for concurrency control. In Proceedings of ACM Transactions on Database System (TODS), pp. 213-226.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, Volume 21 Issue 7, pp. 558-565.

Lamport, L. (1998). The part-time parliament. In ACM Transactions on Computer Systems (TOCS), pp. 133-169.

LAN performance on the WAN (n.d.). Retrieved from https://www.masv.io/lan-performance-on-the-wan-part-1

Lawrence, N. D., Rowstron, A. I. T., Bishop, C. M., and Taylor, M. J. (2002). Optimizing synchronization times for mobile devices. In Advances in Neural Information Processing Systems, T. G. Dietterich, S. Becker, and Z. Ghahramani, Eds. Volume 14. MIT Press, Cambridge, MA, USA, pp. 1401-1408.

Leung, A., Pasupathy, S., Goodson, G. & Miller, E. (2008). Measurement and Analysis of Large-Scale Network Data object System Workloads. In Proceedings of the 2008 USENIX Conference on Annual Technical Conference (ATC), pp. 213-226.

Maccormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. (2004). Boxwood: Abstractions as the foundation for storage infrastructure. In the 6th Conference on Operating Systems Design and Implementation (OSDI), pp. 105-120.

Microsoft Azure (n.d.). Retrieved from: http://azure.microsoft.com/en-us/

Microsoft Azure File Sync (AFS) (n.d.). Retrieved from: https://docs.microsoft.com/en-us/azure/storage/files/storage-sync-files-monitoring

Microsoft Distributed File System Replication (DFSR) (n.d.). Retrieved from: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/dfsr/distributed-file-system-replication--dfsr-

Microsoft NTFS file system directory management functions (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/fileio/directory-management-functions

Microsoft NTFS file system Update Sequence Number (USN) change journal (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/fileio/change-journals

Microsoft Windows Executable Files (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/msi/executable-files

Microsoft Windows Named Pipes (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/ipc/named-pipes

Microsoft Windows Remote Procedure Call (RPC) (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/rpc/rpc-start-page

Microsoft Windows Threads (n.d.). Retrieved from: https://docs.microsoft.com/en-us/windows/desktop/procthread/creating-threads

Mockapetris, P. and Dunlap, K. (1988). Development of the Domain Name System. In Newsletter ACM SIGCOMM Computer Communication Review, pp. 123-133.

Parker, D. S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C. (1983). Detection of mutual inconsistency in distributed systems. IEEE Transactions on Software Engineering, Volume 9 Issue 3, pp. 240-247.

Reiher, P., Heidemann, J. S., Ratner, D., Skinner, G., and Popek, G. J. (1994). Resolving file conflicts in the Ficus file system. In Proceedings of the 1994 USENIX Annual Technical Conference (ATC), pp. 183-195.

Roselli, D., Lorch, J., and Anderson, T. (2000). A comparison of file system workloads. In Proceedings of the 2000 USENIX Annual Technical Conference (ATC), pp. 4-4.

Saito, Y., Karamonolis, C., Karlsson, M., and Mahalingam, M. (2002). Taming aggressive replication in the Pangaea wide-area file system. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), pp. 15-30.

Saito, Y. and Shapiro, M. (2005). Optimistic replication. ACM Computing Survey, Volume 37 Issue 1, pp. 42-81.

Satyanarayanan, M. and Kistler, J. (1990). Disconnected operation in the Coda file system. ACM Transactions on Computer Systems, Volume 10 Issue 1, pp. 3-25.

Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. (1990). Coda: A highly available file system for a disconnected workstation environment. IEEE Transactions on Computers, Volume 39 Issue 4, pp. 447-459.

Shankaranarayanan, P., Sivakumar, A., Rao, S., and Tawarmalani, M (2014). Performance sensitive replication in geo-distributed cloud datastores. In Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 240-251.

Shapiro, M., Pregui_ca, N., Baquero, C., and Zawirski, M. (2011). Conflict-free replicated data types. In the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS), pp. 386-400.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. (1994). Session guarantees for weakly consistent replicated data. In the Third International Conference on Parallel and Distributed Information Systems (PDIS), pp. 140-149.

Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), pp. 172-182.

Thomas, R. (1979). A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems (TODS), Volume 4 Issue 2, pp. 180-209.

Vogels, W (2008). Eventually consistent. Queue 6, pp. 14-19 (October 2008)

Vrable, M., Savage, S., and Voelker, G., M. (2012). Bluesky: a cloud-backed file system for the enterprise. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST), pp. 19-19.

Wang, A., Reiher, P., and Bagrodia, R (1997). A simulation framework for evaluating replicated filing environment. Technical Report CSD-970018, University of California, Los Angeles.

Wang, W. and Amza, C. (2009). On optimal concurrency control for optimistic replication. In Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 317-326.

Wang, Y., Alvisi, L., and Dahlin, M. (2012, June). Gnothi: separating data and metadata for efficient and available storage replication. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC), pp. 38-38.

Yu, H. and Vahdat, A. (2000). Design and evaluation of a continuous consistency model for replicated services. In the 4th Symposium on Operating System Design and Implementation (OSDI), pp. 305-318.

Yu, H. and Vahdat, A. (2001). The costs and limits of availability for replicated services. In the 18th Symposium on Operating System Principles (SOSP), pp. 29-42.

Yu, H. and Vahdat, A. (2002): Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Transactions on Computer Systems (TOCS), Volume 20 Issue 3, pp. 239-282.

Yu, H. and Vahdat, A. (2006). The costs and limits of availability for replicated services. ACM Transactions on Computer Systems (TOCS), Volume 24 Issue 1, pp. 70-113.

Zhao, W. (2014). Building dependable distributed systems. Scrivener Publishing LLC, Beverly, MA. ISBN: 978-1-118-54943-8.