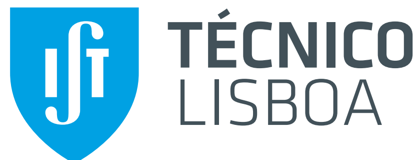


Dynamic Adaptation of Geo-Replicated CRDTs

Carlos Guilherme Crisóstomo Bartolomeu

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Alberto Manuel Rodrigues da Silva
Advisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member:	Prof. Nuno Manuel Ribeiro Preguiça

October 2015

Acknowledgements

Firstly, I would like to thank my advisor, Prof. Luis Rodrigues, for guiding and supporting me over this year, and also for patience. I would specially like to thank Manuel Bravo for the fruitful discussions and his helpful comments, he was constant source of guidance and support on every stage of the research work.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013, via the project PEPITA (PTDC/EEI-SCR/2776/2012).

Lisboa, October 2015

Carlos Guilherme Crisóstomo Bartolomeu

For my family and friends,

Resumo

Com a chegada da computação na nuvem, e com a necessidade de manter a informação replicada em centros de dados geograficamente distantes, tornou-se relevante procurar por estratégias que garantam coerência nos dados com o mínimo de sincronização entre as réplicas. Infelizmente, a maior parte dos tipos de dados requerem que as operações sejam ordenadas de forma total para garantir a coerência das réplicas.

Os tipos de dados replicados isentos de conflitos, do Inglês, "Conflict-free Replicated Data Types" ou simplesmente CRDTs, são tipos de dados cujas operações não entram em conflito umas com as outras e, portanto, podem ser replicadas com um custo mínimo na coordenação entre réplicas. Apesar de esta propriedade garantir que o sistema converge para um estado coerente quando fica em repouso, a escolha do melhor método para propagar as actualizações não é trivial. Abordagens diferentes, tais como o envio do estado ou a propagação de operações, foram propostos para propagar actualizações de forma eficiente, com diferentes contra-partidas tais como o uso da rede ou a desactualização da informação.

Esta tese propõe e avalia técnicas para o sistema escolher de forma automática qual o sistema de propagação de actualizações a usar, com base no desempenho observado e nos requisitos da aplicação. Estas técnicas foram integradas e avaliadas no SwiftCloud, um sistema que materializa o estado da arte na manutenção de CRDTs geograficamente distribuídas.

Abstract

With the advent of cloud computing, and the need to maintain data replicated in geographically remote data centers, searching for strategies to provide data consistency with minimal synchronization became very relevant. Unfortunately, most data types require operations to be totally ordered to ensure replica consistency.

Conflict-free Replicated Data Types (CRDTs) are data types whose operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. While it is easy to ensure that all replicas of CRDTs become eventually consistent when the system becomes quiescent, different techniques can be used to propagate the updates as efficiently as possible. Different approaches, such as state transfer and operation forwarding, have been proposed to propagate the updates as efficiently as possible, with different tradeoffs among the amount of network traffic generated and the staleness of local information.

This thesis proposes and evaluates techniques to automatically adapt a CRDT implementation, such that the best approach is used, based on the application needs (captured by a SLA) and the observed system configuration. Our techniques have been integrated in SwiftCloud, a state of the art geo-replicated store based on CRDTs.

Palavras Chave
Keywords

Palavras Chave

Sistemas Distribuídos

Replicação

CRDTs

Coerência Causal

Adaptação Dinâmica

Keywords

Distributed Systems

Replication

CRDTs

Causal Consistency

Dynamic Adaptation

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Contributions 4

1.3 Results 5

1.4 Research History 5

1.5 Structure of the Document 6

2 Related Work **7**

2.1 Replication and Data Consistency 7

2.2 Service Level Agreement (SLA) 9

2.3 Distributed Key-value Store Systems 10

2.3.1 Dynamo and Riak 10

2.3.2 SwiftCloud 10

2.4 Conflict-free Replicated Data Types (CRDTs) 12

2.4.1 Operation-based CRDT: Commutative Replicated Data Type (CmRDT) 13

2.4.2 State-based CRDT: Convergent Replicated Data Type (CvRDT) 13

2.4.3 Delta-based CRDT: Conflict-free Replicated Data Type (δ-CRDT) 14

2.5 Portfolio of basic CRDTs 14

2.5.1 Counters 15

2.5.1.1 Operation-based Counter: 15

2.5.1.2 State-based Counter: 16

2.5.1.3 Delta-based Counter: 16

2.5.2 Sets 17

2.5.2.1 Operation-based Set: 19

2.5.2.2 State-based Set: 19

2.5.2.3 Delta-based Set: 19

2.5.3 Registers 21

2.5.4 Graphs 21

2.5.5 CRDT use cases 22

3 Bendy **25**

3.1 State-Based vs Operation-Based Tradeoffs 25

3.1.1 Memory Stored in Each Replica 26

3.1.2 Complexity of Operations 27

3.1.3 Relevance of the Object Size 29

3.1.4 Impact of Buffering and using an SLA 30

3.1.5 Impact of Inter-Object Dependencies 31

3.2 Overview of the System 32

3.3 Taking SLAs Into Account 34

3.4 Propagation Time 35

3.5 Dynamic Adaptation 37

3.5.1 Reconfiguration 37

3.5.2 Selecting the Right Implementation 38

3.6 Implementation Issues 39

3.6.1 Implementation of the state-based approach 39

3.6.2 Combining operation-based and state-based approaches 40
3.6.3 Implementing a dynamic system 40

4 Evaluation **43**

4.1 Experimental Setup 43
4.2 Throughput 44
4.3 Bandwidth Utilization 46
4.4 Dynamic Behavior 46
4.5 Discussion 48

5 Conclusions **51**

5.1 Conclusions 51
5.2 Future Work 52

Bibliography **55**

List of Figures

3.1 Comparison of the memory stored in each replica to represent a counter 26

3.2 Comparison of the memory stored in each replica to represent a set 27

3.3 Time that takes to converge two replicas that diverge in N operations - either by merging states (state and delta) or by applying all operations (op) 28

3.4 Impact of the object size for both approaches 29

3.5 Relaxing data freshness requirements 30

3.6 Time required for the synchronization of an object. 32

3.7 Overview of the described system. 33

3.8 Propagation Time 36

4.1 Throughput vs Bandwidth 45

4.2 Throughput of one DC during the balancing process of top-k objects. 47

4.3 Percentage of objects in a top-k list that are different from the optimal top-k list. 48

List of Tables

2.1 Qualitative comparison between CRDT approaches 24

4.1 Coordination time (average) 44

List of Algorithms

2.1 Operation-based Counter 15
2.2 State-based Grow-only Counter 16
2.3 Delta-based Grow-only Counter 17
2.4 Operation-based Observed-Removed Set (OR-Set) 18
2.5 State-based Observed-Remove Set 18
2.6 Delta-based Optimized Observed-Removed Set 20
3.1 Process of migrating objects from one instance to the other. 38

Acronyms

CRDTs Conflict-free Replicated Data Types

δ-CRDT Delta Conflict-free Replicated Data Type

CmRDT Commutative Replicated Data Type

CvRDT Convergent Replicated Data Type

DC Datacenter

SLA Service Level Agreement

1 Introduction

This thesis addresses the limitation of only propagating operations of objects between datacenters. The propagation of operations is in most cases enough, but when the distribution of the access to the objects is not uniform this may become a bottleneck. For this purpose, the thesis proposes a combination of two different approaches to propagate updates and a dynamic mechanism that decides which approach is used by an object.

1.1 Motivation

With the advent of cloud computing, and the need to maintain data replicated in geographically remote data centers, searching for strategies to provide data consistency with minimal synchronization became very relevant. Unfortunately, most data types require operations to be totally ordered to ensure replica consistency. This means that operations are diverted to a single primary replica, incurring on long delays and availability problems, or that an expensive consensus protocol such as Paxos(Lamport 1998) is used to order the updates.

Conflict-free Replicated Data Types (CRDTs)(Shapiro, Preguiça, Baquero, and Zawirski 2011; Shapiro, Preguiça, Baquero, and Zawirski 2011; Letia, Preguiça, and Shapiro 2009) are data types whose concurrent operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. CRDTs are implemented in such a way that any two concurrent operations A and B are commutative and, therefore, even if they are executed in different sequential orders at different replicas, the final result is still the same. As a result, there are no conflicts among concurrent operations and replicas can often execute operations promptly, without synchronization with other replicas, i.e., operations may be executed locally first and shipped to other replicas only when it becomes appropriated. Using this approach, even if replicas diverge from each other, convergence is eventually reached due to CRDTs properties. Thus CRDTs, unlike other eventual consistency approaches, may strongly

simplify the development of distributed application such as social networks, collaborative documents, or online stores(Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and Preguiça 2013; Navalho, Duarte, Preguiça, and Shapiro 2013; Preguica, Marques, Shapiro, and Letia 2009).

Two main types of CRDTs have been proposed, that differ on the techniques they use to reach eventual consistency, namely *operation-based*(Shapiro, Preguiça, Baquero, and Zawirski 2011; Baquero, Almeida, and Shoker 2014) and *state-based*(Shapiro, Preguiça, Baquero, and Zawirski 2011) CRDTs. Operation-based CRDTs send to other replicas the operations that are executed locally; these are later executed remotely also. On the contrary, state-based CRDTs send the full state of the object (which includes the outcome of the operations), such that it can be merged with the local state at remote replicas. Both approaches have advantages and disadvantages as we will see later. A third type of CRDTs has also been proposed more recently, named *delta-based* CRDTs(Almeida, Shoker, and Baquero 2014), which combines features of the two basic approaches above. Delta-based CRDTs do not ship the full state but, instead, send a smaller state, labeled delta-state, that represents the operations performed between two instants.

The CRDT specification allows for the implementation to choose when it is more appropriate to exchange information among replicas, and allows to postpone eventual consistency to be reached in order to save communication and computation resources. For how long eventual consistency can be postponed depends on the application requirements. These requirements can be captured by a Service Level Agreement (SLA)(Terry, Prabhakaran, Kotla, Balakrishnan, Aguilera, and Abu-Libdeh 2013). By using SLAs, the client or the System Administrator can specify how the system should behave at a given situation.

1.2 Contributions

With this work, we aim at studying the possibility of automating the choice of the CRDT implementation based on the SLA that have been defined and based on the characteristics of the execution environment. These characteristics can be available processing power, observed network latency, timing of propagating messages, etc. More precisely, the thesis makes the following contributions:

- A practical study that compares an operation-based approach with a state-based approach.

- The architecture of a system that benefits from using both approaches simultaneously and dynamically.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- An analysis of the advantages and limitations of the SwiftCloud implementation.
- A prototype of the described architecture called Bendy.
- An experimental evaluation of the implemented prototype, which can have up to 50% more throughput when compared to a normal operation-based solution.

1.4 Research History

During my work, I benefited from the fruitful collaboration of Manuel Bravo.

In the beginning, the main focus of this work was to study how an operation-based solution or a state-based solution would behave in different environments. In particular, how different workloads, object distributions, object sizes and buffering updates would affect the throughput and/or staleness of the object. Because of the variety of variables, we focused our experiments on a single type of object: the set. A set is an interesting enough type of object because its size can vary and sometimes there are dependencies between an *add* operation and a *remove* operation. After an intense analysis of this data type, we designed and tested a prototype of a system that benefits from combining both solutions to optimise the propagation of updates. The architecture proposed in this thesis was inspired by the results obtained by the previous study. Therefore most of the work was testing different running environments to better decide the final architecture of our solution. Since the beginning we decide to build our solution on to of SwiftCloud. So part of the work was also understanding the system and most of the times trying to understand if SwiftCloud was capable of supporting our ideas.

1.5 Structure of the Document

The rest of this document is organized as follows. For self-containment, Chapter 2 provides an introduction to CRDTs and a description of its current implementations. Chapter 3 provides a study about two implementations of CRDTs along with the architecture and the algorithms used by Bendy. Chapter 4 presents the results of the experimental evaluation of the system. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

Introduction

This chapter surveys the relevant work that has been produced in the area of CRDTs. We start by discussing the problem of consistency when managing replicated data, the consistency/cost tradeoff, and how the notion of Service Level Agreement can be used to optimize the system performance. Then we move to introduce the general properties of CRDTs and to present the three main approaches that have been explored to implement that abstraction, namely: operation-based, state-based, and delta-based CRDTs. We survey some of the most common CRDTs, in particular counter and set datatypes. Finally, we compare how different implementation approaches perform in terms of memory, time to propagate updates and throughput.

2.1 Replication and Data Consistency

When data replicas are placed in geographically distant locations, such that the communication latency among replicas becomes significant, a tradeoff among performance and consistency emerges. In particular, the performance of reads with different consistency guarantees may be substantial. Strongly consistent reads generally involve multiple replicas or must be served by a primary replica, whereas eventually consistent reads can be answered by the closest replica.

Six levels of consistency for read operation have been defined in the literature(Terry, Prabhakaran, Kotla, Balakrishnan, Aguilera, and Abu-Libdeh 2013; Terry, Demers, Petersen, Spreitzer, Theimer, and Welch 1994): strong, causal, bounded, read-my-writes, monotonic, and eventual consistency, as explained below.

- **Strong:** A *read* returns the value of the last preceding *write* performed by any client.

- **Causal:** A *read* returns the value of a latest *write* that causally precedes it or returns some later version. The causal precedence relation $<$ is defined such that $op1 < op2$ if either (a) $op1$ occurs before $op2$ in the same session, (b) $op1$ is a *write* and $op2$ is a *read* that returns the version *writen* in $op1$, or ,by the property of transitivity, (c) for some $op3$, $op1 < op3$ and $op3 < op2$.

- **Bounded(t):** A *read* returns a value that is stale by at most t seconds. Specifically, it returns the value of the latest *write* that completed at most t seconds ago or some more recent version.

- **Read-my-Writes:** A *read* returns the value written by the last preceding *write* in the same session or returns a later version; if no *writes* have been performed to this key in this session, then the *read* may return any previous value as in eventual consistency.

- **Monotonic Reads:** A *read* returns the same or a later version as a previous *read* in this session; if the session has no previous *reads* for this key, then the *read* may return the value of any *write*.

- **Eventual:** A *read* returns the value written by any *write*, i.e. any version of the object with the given key; clients can expect that the latest version eventually would be returned if no further *writes* were performed, but there are no guarantees concerning how long this might take.

In a similar manner, in (Terry, Demers, Petersen, Spreitzer, Theimer, and Welch 1994) consistency levels have also been defined for write operations, namely *Writes Follow Reads* and *Monotonic Writes* as described below:

- **Writes Follow Reads:** *Writes* made during the session are ordered after any *Writes* whose effects were seen by previous *Reads* in the session.

- **Monotonic Writes:** *Writes* must follow previous *Writes* within the session.

Although the use of replication raises the problem of data consistency, it has many advantages. In first place, replication offers fault-tolerance. Also, by placing replicas close to the users, replication can provide fast access to data.

The tradeoffs among fault-tolerance and consistency have been captured by the well-known CAP Theorem (Brewer 2000), that states that in a shared-data system we can only have two of the following three properties: Consistency, Availability and Partition-tolerance.

These tradeoffs open a large solution space that has been explored in many different ways by different systems. In our work we weaken consistency, by allowing replicas having different states, to ensure availability and leverage from the CRDTs properties to simplify the replication management.

2.2 Service Level Agreement (SLA)

As we have seen in the previous section, it is possible to define different levels of consistency for read and write operations. Typically, the consistency level that must be enforced is a function of the application semantics and business goals. These requirements can be expressed in the form of a Service Level Agreement (SLA).

In most cases, applications prefer to use the stronger consistency, when the network conditions are favorable. However, when the network is unstable (higher latencies due to congestion, partitions, etc), different applications require different consistency guarantees. Actually, the preferred consistency guarantee may be even a function of the actual value of the network latency that is observed (for instance, an application may be willing to wait t seconds to get strong consistency but not more).

To cope with the fact that a given application may prefer different consistency levels for different operational conditions, the use of a multiple-choice SLA has been proposed in the context of the Pileus system(Terry, Prabhakaran, Kotla, Balakrishnan, Aguilera, and Abu-Libdeh 2013). This system allows developers to define which level of consistency should be used according to the response time of the system. This means, for a given SLA, if the system predicts that the response time of a strong read is more than the desired, then it should use a other level of consistency, previously specified, in order to achieve the desired response time with more gain. With such an SLA, a system is able to adapt to different configurations of replicas and users and to changing conditions, including variations in network or server load.

In the context of our project, we will not use the SLA to determine the consistency level to use. Instead, the max time that a user is willing to wait for an update, SLA time, will be used

by our system as a hint to determine which approach an object should use.

2.3 Distributed Key-value Store Systems

Nowadays, there is a need for a system that gives the "always-on" experience and that is able deal with failures in an infrastructure without impacting availability or performance. Also, there is a need for a more reliable and scalable system as several companies grow world-wide and serve millions customers around the world.

2.3.1 Dynamo and Riak

An example of a system that achieves those needs is Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007a). Dynamo gathers a well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing, and consistency is simplified by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. The gossip technique is used for a distributed failure detection and membership protocol. Dynamo is completely decentralized and storage nodes can be added and removed without requiring any manual partitioning or redistribution.

Another example is Riak (Klophaus 2010), inspired by Dynamo and many other similar systems, Riak has some new features and extended functionalities that better fits their vision of a geo-replicated and high available system. One feature is the use of CRDTs. By using this kind of data types, Riak is able to improve scalability and independence between storage nodes and removes the need for quorums and other expensive synchronization protocols.

2.3.2 SwiftCloud

SwiftCloud (Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and Preguiça 2013), which was the starting point of this thesis work, is another distributed key-value store system designed with the objective of proving the benefits of using CRDTs. SwiftCloud already embodies a number of interesting features, such as support for transactions, caching at the clients, propagation of operations among datacenters, etc. Therefore, we have opted to use SwiftCloud

as a testbed to get more insights on the advantages and disadvantages of operation based and state based CRDTs in practice. Because many decisions that were made in the architecture are related to how SwiftCloud was design, we describe below a few relevant aspects of the system.

In SwiftCloud, client applications do not contact datacenters directly; instead, they use a local proxy that keeps copies of the recently accessed objects and also maintains metadata that encodes dependencies among updates. The purpose of the metadata is to ensure that client application always observe states that are consistent with causality(Shapiro, Preguiça, Baquero, and Zawirski 2011). Every time a transaction needs to read or write an object, it first fetch the object from the local cache. If the object is not in the cache, or if the cache is disabled, the client proxy fetches it from any datacenter that has a clock equal or higher than the client's clock. When the transaction ends, if any update was generated, it is propagated to a datacenter asynchronously. Otherwise, it is a read-only transaction and the transaction ends without contacting a datacenter.

Each datacenter maintains a copy of the entire CRDT database. Datacenters contact each other to propagate updates received from clients and to ensure that their state eventually converge. There are two services that run in each datacenter: *surrogates*, that are in charge of interaction with clients, and a single *sequencer*, that is responsible for receiving and propagating updates among datacenters. A global single clock is used to record timestamps made by transactions. It is also used to ensure that transactions are only applied when dependencies are satisfied. Internally, CRDTs are encoded and stored using data structures similar to the ones design for operation-based approach (Shapiro, Preguiça, Baquero, and Zawirski 2011). Therefore, the propagation of updates among data centers follows exclusively an operation-based approach. One of the contributions of this work is to extend SwiftCloud to support also a state-based propagation of updates.

Regarding the propagation of updates among datacenters, they are dispatched asynchronously as soon as a client commits a transaction. SwiftCloud offers no support to take clients SLAs into consideration. When an update is received from a client in a datacenter, the corresponding operation is scheduled to be shipped to the remaining datacenters asynchronously, but as soon as possible. Although, in theory, the system could benefit from batching multiple updates in applications with weak freshness constraints, no attempt to perform such batching is implemented. When a replica receives an update from another replica, it first checks if the

dependencies are satisfied and then applies it. This is a sequential process; therefore, if more than one update is received from a remote datacenter, all updates are queued and processed serially, one by one. SwiftCloud makes no attempt to parallelize this process because the queue is usually short. However, if many updates are received close to each other, the processing of the queue may become a bottleneck. Unfortunately, it is not trivial to parallelize the applications of updates because updates can be only applied when the dependencies are satisfied, and updates coming from a given datacenter typically depend of each other, due to the way SwiftCloud maintains causality metadata.

2.4 Conflict-free Replicated Data Types (CRDTs)

A Conflict-free Replicated Data type (CRDT) is data structure, like a counter or a set, which can be deployed in a distributed system, by placing replicas of the data in multiple servers, in such a way that concurrent operations do not conflict. As a result, a client may contact any server, and execute a sequence of operations on that server, without being forced to wait for (potentially blocking) coordination with other servers.

Since CRDTs avoid explicit synchronization at every operation, the propagation of updates is often made asynchronously, in background. This means that, at a given time, different replicas may contain different states. However, if the system becomes quiescent, eventually all replicas converge to the same state. However, unlike in other forms of eventual consistency, operations in CRDTs never have to be cancelled or compensated, as a result of the synchronization among replicas (this derives form the fact that operations never conflict).

The state maintained by each replica is also named the *replica payload*. Client requests are modelled as *operations*. An operation contains the method to be executed and its arguments. The state of a replica after executing a client request is a deterministic function of its state before executing the request and of the operation received. The main difference among CRDT implementations is related with the way different replicas are synchronized. In this respect, there are three main approaches that have been proposed to implement CRDTs: operation-based CRDTs, state-based CRDTs, and an hybrid approach named delta-based CRDTs. These approaches are described in the following subsections.

2.4.1 Operation-based CRDT: Commutative Replicated Data Type (CmRDT)

Operation-based CRDTs ensure that replicas eventually converge by propagating all update operations to all replicas. Naturally, operations that are *read-only*, i.e. do not change the state of the object, can be executed locally at any replica, and have the result returned back to the client without the need for any other coordination. Operation-based CRDTs can be seen as implementing a form of active replication, given that all replicas must execute all the requests.

The algorithm used to disseminate the operations among all replicas is independent of the implementation of the CRDT, and several strategies may be used: broadcast, gossip, spanning-trees, etc. However, there are a number of properties that the dissemination process must preserve. In first place, the dissemination must be reliable, such that all operations are received by all replicas and must ensure exactly-once delivery. If a replica does not receive a given operation, or if it applies a given operation more than once, its state may never converge with the state of the remaining replicas. Furthermore, some operation-based CRDTs, such as the optimised observed-removed set(Bieniusa, Zawirski, Preguiça, Shapiro, Baquero, Balegas, and Duarte 2012), require causal delivery, which makes the dissemination process even more demanding.

2.4.2 State-based CRDT: Convergent Replicated Data Type (CvRDT)

State-based CRDTs ensure that replicas eventually converge by propagating the state of each replica to other replicas and by relying on a *merge* operation that combines the state received from a remote replica with the state of the local replica. When using state-based CRDTs, a request is sent to a single replica, that executes the operation locally. The operation is not propagated to the other replicas. Instead, the state of the replica that has executed the request will, eventually, be propagated and merged with the state of the other replicas.

The key to this approach is to encode the state of the replicas in such a way that the *merge* function becomes idempotent. Thus, if the same state update is applied twice to the same replica, the result should be the same as if it is applied only once. Furthermore, if two different states contain the effects of different, but overlapping, set of operations, the result of the merge function should still be equivalent to the state that would have been achieved by executing each

request only once in that replica. To ensure that states can be merged in an idempotent manner is not trivial and, usually, the state must be encoded in a manner that is less space efficient than in operation-based CRDTs.

As before, the approach is independent of the strategy used to decide when to propagate the state of one replica to the other replicas. A new state can be sent every time a client request is processed, or a new state can be sent periodically, and contain the result of many update requests. However, the idempotent property of the merge function puts much less constraints on the dissemination of state updates, when compared to the dissemination of operations. State messages may be delivered more than once or even lost; lost updates are masked by the next state transfer. In the previous case, the order by which state updates is applied is not relevant because the internal structure of the data type together with the *merge* operation ensures convergence of the replicas.

2.4.3 Delta-based CRDT: Conflict-free Replicated Data Type (δ-CRDT)

Delta-based CRDTs combine features of operation based and state-based CRDTs. The idea is that, as a result of applying an operation, a *delta-state* is produced. A *delta* only captures changes caused by the associated operation but has the mergeable properties of state-based CRDT. When a *delta A* is merged with other *delta B*, a new *delta C* is created, which represents the *delta A* and *delta B* merged. One delta state is comparable to an operation but has the property of being able to capture multiple operations as a result of multiple merges. Since a delta states are mergeable, it can be sent to replicas without any requirement because the final state of the replica will always be consistent. All the replicas converge when all the replicas have seen, directly or indirectly, all the deltas states.

2.5 Portfolio of basic CRDTs

In this section we describe and specify some of the basic data types that have been proposed, such as the *counter* and the *set*, which are the basic blocks for more complex data types like graphs. The study of these concrete data types helps in understanding the challenges, benefits, and limitations of CRDTs.

About the algorithms that will be presented, the communication is not described. Instead, we assume the replicas have communications channels and what they send to each other is returned by the operation performed.

2.5.1 Counters

A counter is a replicated integer that supports three operations, namely *increment*, *decrement*, and *value* (the first two operations change the state of the counter and the third operation returns its value). It is straightforward to extend the interface to include operations for *adding* and *subtracting* any value. The semantics of the counter are such that its value converges towards the global number of increments minus the number of decrements. A counter is useful in many applications, for instance for counting the number of currently logged-in users.

2.5.1.1 Operation-based Counter:

An operation-based counter is the simplest CRDT we can find. Its *payload* is an integer i and supports two basic operations: *increment* and *decrement*. It can be extended to support increments of any value as can be infered just by looking at the specification depicted in Alg. 2.1. This is possible because any increment or decrement are operations that commute.

Algorithm 2.1 Operation-based Counter

1: **payload** integer i
2:     **initial** 0
3: **function** INCREMENT
4:     $i := i + 1$
5:     **return** $+1$ ▷ Operation
6: **function** DECREMENT
7:     $i := i - 1$
8:     **return** -1 ▷ Operation
9: **function** VALUE
10:     **return** i
11: **function** UPDATE(operation k)
12:     $i := i + k$ ▷ it will increment or decrement depending on k being positive or negative

2.5.1.2 State-based Counter:

A state-based counter requires a more complex data structure. To simplify the problem, let us specify a grow-only counter. Suppose that we have a payload, like in the operation-based approach, which is an integer i, and the merge operation does the *max* of each payload. Consider two replicas, with the same initial state of 0; at each one, a client originates increment. They converge to 1 instead of the expected 2. Suppose instead that the payload is an integer and that merge adds the two values. This implementation does not have the properties of a CvRDT, given that the merge operation is not idempotent. In (Bieniusa, Zawirski, Preguiça, Shapiro, Baquero, Balegas, and Duarte 2012) the following solution has been proposed to implement a state-based counter: the payload is stored as a vector of integers, with one position per replica (inspired by vector clocks). To increment, each replica adds 1 to its position in the vector. The value is the sum of all entries of the vector. Merge takes the maximum of each entry. The specification can be seen in Alg. 2.2. To implement a counter that supports *increment* and *decrement* operations we need two vectors, one for increments and one for decrements. This is because if we use only one vector the max function in the merge operation will not take into account the decrements. The value of the counter is increments minus decrements.

Algorithm 2.2 State-based Grow-only Counter

```
1: payload integer[n] P                          ▷ One entry per replica
2:     initial [0, 0,...,0]
3: function INCREMENT
4:     r := myID()                               ▷ r: source replica
5:     P[r] := P[r] + 1
6: function VALUE
7:     return ∑_i P[i]
8: function MERGE(State X)
9:     P[i] := max(P[i], X.P[i]) : ∀i ∈ [0, n − 1]
```

2.5.1.3 Delta-based Counter:

The delta-based counter is inspired by the specification of the state-based counter. After executing an increment it produces a delta which is basically the replica's entry of the vector. In Alg. 2.3 shows that the payload is the same and the merge operation is the same as well. The only difference in the merge operation is that is most likely for a delta to have the entry of one replica's counter instead of the full state that have the counter's entries of all the replicas. This

Algorithm 2.3 Delta-based Grow-only Counter

```
1:  payload integer[n] P                                  ▷ One entry per replica
2:      initial [0, 0,...,0]
3:  function INCREMENT
4:      r := myID()                                       ▷ r: source replica
5:      P[r] := P[r] + 1
6:      return (r, P[r])                                  ▷ Delta
7:  function VALUE
8:      return ∑_i P[i]
9:  function MERGE(Delta X)
10:     P[i] := max(P[i], X.P[i]) : ∀i ∈ X
```

small difference allow the replicas to send a delta or a small group of deltas merged into one instead of the full state. More details about the performance of this implementation are given in Section 3.1.

2.5.2 Sets

A set is a data type that stores elements, without any particular order, and with no repetitions. It has two operations: *add* and *remove*, where *add* adds an element to the set and *remove* removes the element from the set. This data type is the basic block for other complex data structures like maps and graphs as we will see later in this report. So, it is essential to have a specification of this data type for the different approaches of CRDTs. Unfortunately, the semantics of a set under concurrent operations it is not trivial. To introduce the problem, imagine that initially the set contains only one item $\{A\}$ and that its state is maintained by three replicas (1, 2, 3) that are consistent. If, concurrently, replica 1 removes A and then adds A again, replica 2 removes A and replica 3 does nothing, then, there are different serialization orders for these three operations and different orders may provide different final outcomes. A discussion of the possible semantics and valid outcomes of the concurrent set is provided in (Shapiro, Preguiça, Baquero, and Zawirski 2011). In this report we will only consider the *Observed-Removed Set* semantics because it is, at the moment, the best apporach in terms of space and with less limitations.

Algorithm 2.4 Operation-based Observed-Removed Set (OR-Set)

1: **payload** set S ▷ set of pairs { *(element e, unique-tag u), ...*}
2: **initial** $\emptyset$
3: **function** LOOKUP(element e)
4: **boolean** $b = (\exists u : (e, u) \in S)$
5: **return** b
6: **function** ADD(element e)
7: $\alpha := unique()$ ▷ $unique()$ returns a unique tag value
8: $S := S \cup \{(e, \alpha)\}$
9: **return** (add, e, α) ▷ representation of the add operation
10: **function** REMOVE(element e)
11: $R := \{(e, u) | \exists u : (e, u) \in S\}$
12: $S := S \setminus R$
13: **return** $(remove, e, R)$ ▷ representation of the remove operation
14: **function** UPDATE(operation (op, e, u)) ▷ executes operations from other replicas
15: **if** $op = add$ **then**
16: $S := S \cup \{(e, u)\}$
17: **if** $op = remove$ **then**
18: $S := S \setminus u$

Algorithm 2.5 State-based Observed-Remove Set

1: **payload** set E, ▷ E: elements, set of triples { *(element e, timestamp c, replica i)*}
2: vect v ▷ v: summary (vector) of received triples
3: **initial** $\emptyset, [0, ..., 0]$
4: **function** LOOKUP(element e)
5: **boolean** $b = (\exists c, i : (e, c, i) \in S)$
6: **return** b
7: **function** ADD(element e)
8: $r := myId()$ ▷ r = source replica
9: $c := v[r] + 1$
10: $O := \{(e, c', r) \in E | c' < c\}$
11: $v[r] := c$
12: $E := E \cup \{(e, c, r)\} \setminus O$
13: **function** REMOVE(element e)
14: $R := \{\forall c, i : (e, c, i) \in E\}$
15: $E := E \setminus R$
16: **function** MERGE(State B)
17: $M := (E \cap B.E)$
18: $M' := \{(e, c, i) \in E \setminus B.E | c > B.v[i]\}$
19: $M'' := \{(e, c, i) \in B.E \setminus E | c > v[i]\}$
20: $U := M \cup M' \cup M''$
21: $O := \{(e, c, i) \in U | \exists e, c', i \in U : c < c'\}$ ▷ Old and duplicated elements
22: $E := U \setminus O$
23: $v := [max(v[0], B.v[0]), ..., max(v[n], B.v[n])]$

2.5.2.1 Operation-based Set:

To solve the previous problem using Operation-based CRDTs each element needs a unique tag. Therefore, the payload of the data type will be a set of tuples (*element, tag)*, where the *tag* is a unique identifier associated with each insert operation. This *tag* is needed to support the remove operation: when removing an element, a replica will send to others the element that it wants to remove and all the tags that it sees. This way, when *add* and *remove* operations are concurrent the *add* operation will always win because it is adding an element with a new tag that is not in the set of tags from the *remove* operation. When considering the resulting specification, provided in Alg. 2.4, the concurrent scenario described before will always has the same outcome, which is a set with the element A. This solution imposes a constraint on the communication pattern: because a *remove* operation always depends on an *add* operation, the operations exchanged between all replicas must respect causal delivery.

2.5.2.2 State-based Set:

The state-base approach requires a more complex solution to support the merge operation. Thus, the payload needs to maintain a causal vector, where each entry has a timestamp that belongs to a specific replica. The set's payload consists of a set of tuples with three components: *(element, timestamp, replica ID)*. Like the state-based counter, the only information that is sent to other replicas is the state, and the merge operation ensures convergence. The trick in this solution is in the merge operation. An element should be preserved in the merged state only if: either it is in both payloads (set M in Alg. 2.5), or it is in the local payload and not recently removed from the remote one (set M') or vice-versa (M") - an element has been removed if it is not in the payload but its identifier is reflected in the replica's causal vector. This approach does not impose causal delivery constraints on the communication pattern. The drawback is the space for storage and the cost to send the whole state instead of few operations.

2.5.2.3 Delta-based Set:

The solution for the delta-based set (Alg. 2.6) is inspired by the state-base set. This time, instead of the causal vector, we have a set with tuples *(timestamp, replica)* and we call this causal context. The causal context set is used in the merge operation to determine if an element

was added or removed from the set. If a replica makes an *add*, then the delta contains the set with the element added and the causal context has the tag *(timestamp,replica)* associated to that element. If a replica makes a *remove*, then the delta contains an empty set and the causal context has the tag *(timestamp,replica)* associated to element that was removed. In the merge operation their causal contexts are simply unioned; whereas, the new tagged element set only preserves: (1) the triples present in both sets (therefore, not removed in either), and also (2) any triple present in one of the sets and whose tag is not present in the causal context of the other state. This approach is amenable for buffering operations before engaging in communication, because we can compress the result of multiple operations in a single delta state. The drawback is that the causal context will always increase dependently on the number of adds an never decrease. One possible solution to manage the space is by using a distributed garbage collector, that cleans the tags from the causal context of elements that where removed by all replicas. This, obviously, requires some sort of synchronization.

Algorithm 2.6 Delta-based Optimized Observed-Removed Set

1: **payload** set E, ▷ E: elements, set of triples { *(element e, timestamp c, replica i)*}
2: set V ▷ V: { *(timestamp c, replica i)*} causal context
3: **initial** $\emptyset, \emptyset$
4: **function** LOOKUP(element e) : boolean
5: **boolean** $b = (\exists c, i : (e, c, i) \in S)$
6: **return** b
7: **function** ADD(element e) : Delta
8: $r := myId()$ ▷ r = source replica
9: $c := 1 + max(\{k | (r, k) \in V\})$
10: $E := E \cup \{(e, c, r)\}$
11: $V := V \cup \{(c, r)\}$
12: **return** $(\{(e, c, r)\}, \{(c, r)\}$ ▷ Delta
13: **function** REMOVE(element e) : Delta
14: $R := \{\forall c, i : (e, c, i) \in E\}$
15: $R' := V \cap R$
16: $E := E \setminus R$
17: $V := V \setminus R'$
18: **return** $(\{\}, R')$ ▷ Delta
19: **function** MERGE(Delta B)
20: $M := (E \cap B.E)$
21: $M' := \{(e, c, i) \in E | (c, i) \notin B.V\}$
22: $M'' := \{(e, c, i) \in B.E | (c, i) \notin V\}$
23: $E := M \cup M' \cup M''$
24: $V := V \cup B.V$

2.5.3 Registers

A register is a data type that stores one value in memory. It only has two operations: *assign* - to change the value; and *get* - to obtain the value. The problem to solve, in a distributed register, is the concurrent updates to the register because that operation does not commute. Two major approaches have been identified to address this problem(Shapiro, Preguiça, Baquero, and Zawirski 2011): in the first one operation takes precedence over the other (LWW-Register) and, in the second one, both operations are retained (MV-Register). The *Last-Writer-Wins Register* (LWW-Register) tries to solve concurrency using timestamps to create total order. Timestamps are assumed unique, totally ordered, and consistent with causal order; i.e., if assignment 1 happened-before assignment 2, the former's timestamp is less than the latter's (Lamport 1978). This way, older updates with lower timestamps are discarded and updates with higher timestamp are preserved. Due to its simplicity, both the operation-based approach or the state-based approach look similar. This means that there are no advantage of one approach over the other in terms of storage space or cpu consumption. The *Multi-Value Register* (MV-Register), instead of deciding which value is the correct one for concurrent updates, takes all concurrent values and store them in a set. Therefore, it leaves up to the application to decide which value is the correct one.

2.5.4 Graphs

A more complex structure is a graph. A graph is a pair of sets (V, E) (called vertices and edges respectively) such that $E \subseteq V \times V$. Any of the Set specifications described above can be used for V and E. Because of its nature ($E \subseteq V \times V$), operations on vertices and edges are not independent. An edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge(Shapiro, Preguiça, Baquero, and Zawirski 2011). However, if concurrently a edge is added while a vertex is removed there should exist a deterministic way to preserve the invariant $E \subseteq V \times V$. There are two intuitive forms to solve this without using synchronization: (i) Give precedence to *removeVertex*(u): all edges to or from u are removed as a side effect. This it is easy to implement, by using tombstones for removed verteces. (ii) Give precedence to *addEdge*(u, v): if either u or v has been removed, it is restored. A good example of a use of a graph is a Replicated Growable Array (RGA). This is a linked list (linear graph) that provides Partial Order to its elements and supports the operation

$addRight(v, a)$. Each element (vertex) is connected (edges) to other element. With this concept it is now possible to make co-operative text editing systems. To solve the problem of concurrent adds to the same vertex we can either timestamps like in registers or use other deterministic tie breaker.

2.5.5 CRDT use cases

A CRDT is ideal for a Key-value store that provides at least eventual consistency and that demands conflict free operations like in Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007a), Riak (Klophaus 2010) or SwiftCloud (Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and Preguiça 2013).

In terms of applicability, any system that can tolerate some staleness for read operations and/or perform disconnected updates:

- web search
- social networks
- e-mail
- calendaring programs
- news readers
- shopping cart
- co-operative text editing

For some of this applications a weak consistency like eventual consistency is enough, as long as the user waits the least possible for a response. For others, some stronger consistencies may be required, like monotonic reads or read my writes, but still tolerate some data staleness. For example, in a social network, a user don't want some posts to disappear after they were seen, but a user in Europe can tolerate for a late post made in America.

Regarding the shopping cart solution the CRDT set fits perfectly. As it was described before, with CRDT sets we can guarantee that concurrent changes to the set do not conflict with

each other, i.e., we can avoid the anomalies of the Amazon shopping cart(DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007b).

For the co-operative text editing we can take the advantages of the Replicated Growable Array (RGA). Because with RGA we can insert elements in a relative position to other elements, we can design a system that supports multiple users writing text in the same page and even in the same word. The system WOOT(Oster, Urso, Molli, and Imine 2006) is one of many systems that do this.

Summary

This chapter starts by introducing some concepts, like Replication, Data Consistency and Service Level Agreement (SLA). Then it describes some Distributed Key-value Store systems that are the state of the art. In particular SwiftCloud, which was the starting point for the prototype developed in this thesis. Finally, we present in detail the three approaches to implement CRDTs, and we also describe how to implement some data types for each approach.

After making an analysis of what was described and based on the literature, the Table 2.1 summarizes the most relevant features of the three approaches.

In terms of memory stored in the replica the operation-based is the best one because it has almost no metadata. On the contrary, the delta-based requires a large amount of memory for storing the causal history, for the set specification. In terms of computational power, the merge operation has much more overhead when comparing to apply N operations at once due to the complexity of the merge operation. These two factors are where the operation-based approach has big advantages. Now in terms of communication, the operation-based approach must send its operations to every replica and it requires causal delivery. On the other hand, we have the state and delta approaches that do not require causal delivery and only need to send its state/delta to R replicas. The more replicas we send the state, the faster it will converge. In terms of operations that are buffered, once again, the operation-based approach is the worst because it needs an array that grows with the number of operations while the delta-based approach can represent several operations in one object. Finally we have the garbage collector, this is fundamental for the delta-based approach remain useful, otherwise the memory in the replica would grow uncontrollably. We could also use garbage collection in the buffer of operations, in the operation-

based approach, to reduce its size, i.e., search for operations like $add(A)$ followed by $remove(A)$ that can be removed because they cancel each other. About the delta-based approach, we found out that it costs at least as much as a state-based approach. This is dependent on the object type. For example the counter, we will see later that the worst case for the delta approach is as good as using the state approach. While in the case of the set the delta-approach can easily become worse than state-based approach.

Table 2.1: Qualitative comparison between CRDT approaches

	Operation-based	**State-based**	**Delta-based**
Memory in replicas	low	average	high
Computational power	low	high	high
Communication	contact all replicas	contact R replicas	contact R replicas
Memory of buffers	low	-	average
Garbage collection	low	-	high
constrains	causal delivery	none	none

The next chapter starts by analysing, in more detail, each approach and compare them in practice. Then we will introduce the architecture and implementation details of our system, derived from the practical study.

3 Bendy

From the previous chapter, we identified three different approaches to solve the CRDT problem. Each approach has its advantages and disadvantages. However, we need to see in practice if there a system that could benefit from using an approach over the other, and if a dynamic system can be a viable solution.

The Section 3.1 reflects the initial work made for the thesis. It is also in this section, that we've began to understand the real pros and cons of each approach with actual data made from early experiments. This data was important to better design the architecture of the final system and was also used statically for the dynamic decisions.

After this study, we present a global view of the system in Section 3.2, and then we describe in more detail the features of the system in Sections 3.3, 3.4 and 3.5. Finally in Section 3.6, we discuss some issues we had during the development of the system.

3.1 State-Based vs Operation-Based Tradeoffs

In this section, we study the tradeoffs between the use of state-based and operation-based CRDTs in geo-replicated settings. In particular, we use scenarios where the application may have different requirements in terms of data freshness, a feature that is extremely relevant in cloud scenarios(Terry, Prabhakaran, Kotla, Balakrishnan, Aguilera, and Abu-Libdeh 2013). This requirements are captured by SLAs. For this study, we have created an alternative version of SwiftCloud, where the method to propagate updates among the datacenters has been changed to use a state-based approach. We have used both versions, under similar workloads to compare these approaches, varying object sizes and using different SLAs.

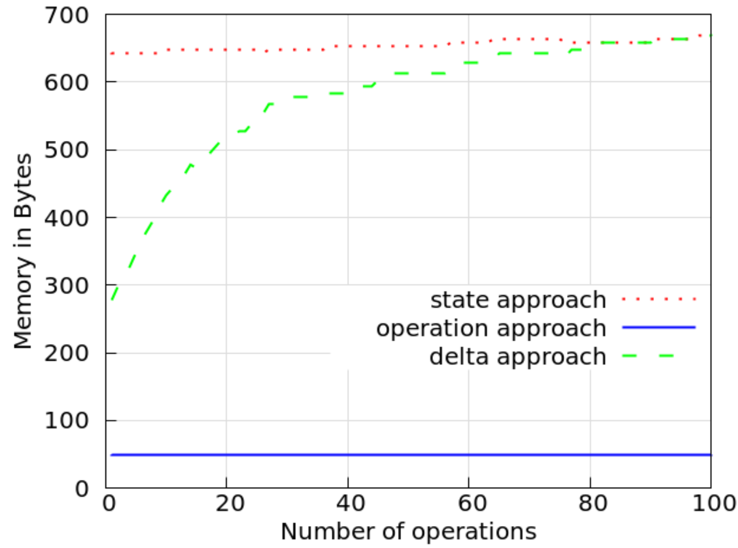


Figure 3.1: Comparison of the memory stored in each replica to represent a counter

3.1.1 Memory Stored in Each Replica

Regarding the Counter, we did a small test to compare the memory usage, see Figure 3.1. This plot depicts the evolution of the memory after N operations in a system with 15 replicas. We compare three implementations/approaches (i) operation-based, (ii) state-based, and (iii) delta-based.

We instantly notice that both operation-based and state-based version are constant in memory size. For the operation-based, it does not have any metadata. So in terms of memory it's the same as having a normal centralized counter which has constant memory size. For the state-based and delta-based approaches, both depend on the number of replicas. However, the state-based depends on the number of replicas that exists in the system, while the delta-based depends on the number of replicas that made an operation. That explains why at the beginning the delta-based uses less memory. But after some time, eventually all replicas will make at least one operation and the memory spent to represent the counter will be the same as if we were using the state-base approach.

Regarding the Set, we did a similar test with 15 replicas but the chance of adding a new element was 70% while the remove chance was 30%, meaning the size of set will increase with more operations made. We can observe that the memory increases as we expected with the increase of the number of elements in the set. In particular, the state-base approach has,

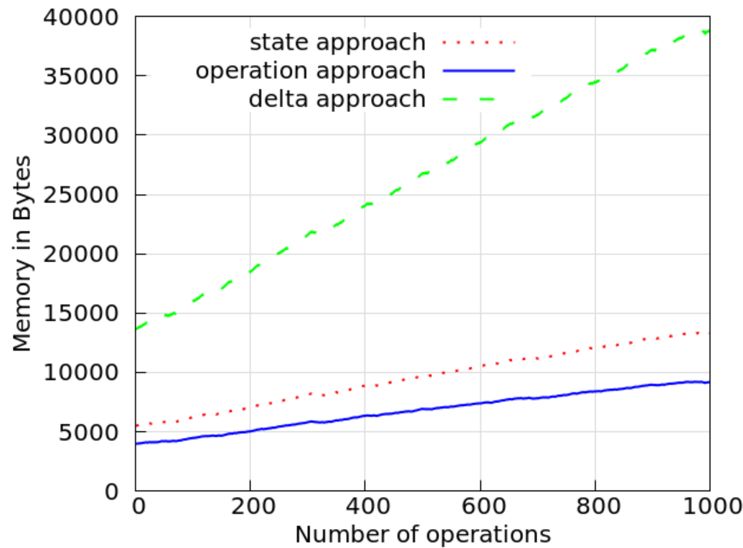


Figure 3.2: Comparison of the memory stored in each replica to represent a set

besides the internal set, a fixed sized vector that depends on the number replicas, and the delta-base approach, instead of a vector, it has a second set that depends on the number of operations (causal context). As we can see in Figure 3.2, the difference between the memory used in operation-based and state-based is small, because the state-base approach has a fixed sized vector and requires more metadata per element. The delta-based approach has the worst results because the memory also depends on the number of operations, and with no garbage collection it grows quickly.

3.1.2 Complexity of Operations

The normal operations of each data type usually do not have a high computational complexity, instead, the merge operation in the state-based and delta-based approach may become expensive. Regarding the counter, the *increment* and *decrement* operations have constant time in any approach. But the merge operation has to make a max function for each entry of the vector and it costs $O(R)$ where R is the number of replicas. Regarding the set, the *add* operation is constant and the for *remove* operation we always have to search for the tags to be removed. When we implement the specifications of the set we can always use some optimizations like hash sets or maps, for fast access to the element and its tags, but we cannot avoid the overhead of the merge operation. The merge operation, either the state-based or delta-based approach, has to make intersection of the payload to see the common elements and then check the elements

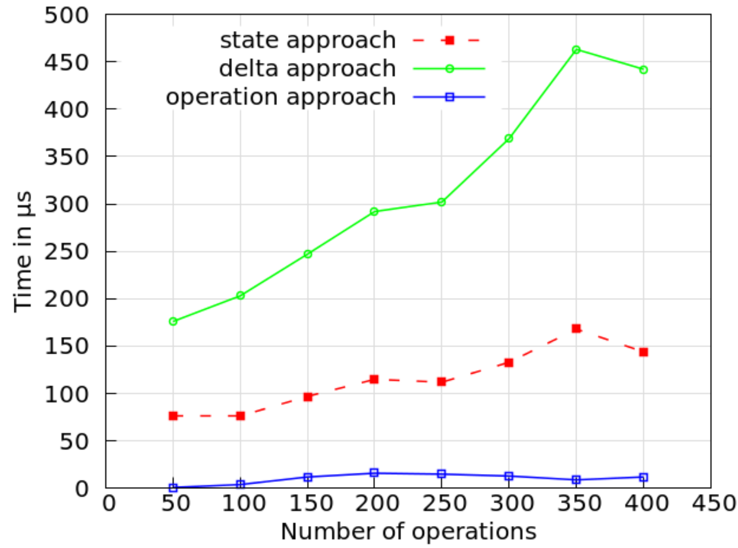


Figure 3.3: Time that takes to converge two replicas that diverge in N operations - either by merging states (state and delta) or by applying all operations (op)

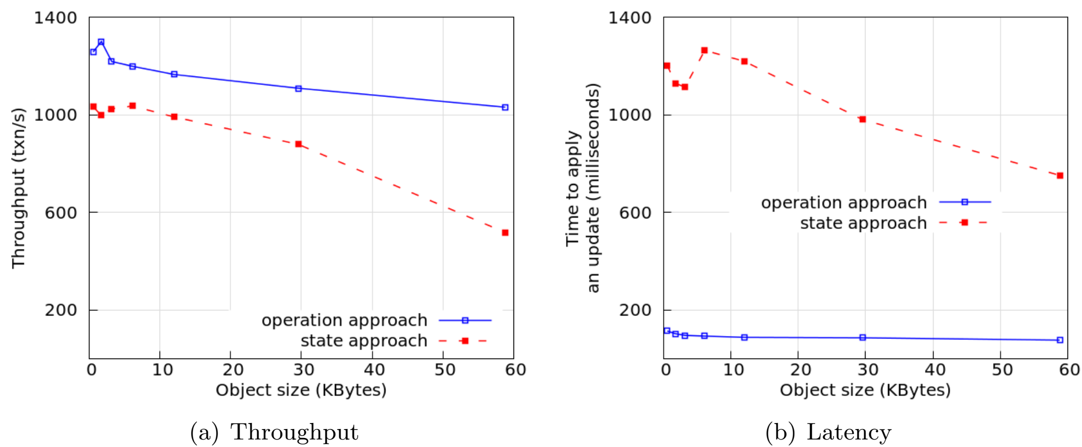
that are new in both of the payload to not remove them. The state-base approach demands some computational power for this operation comparing to the operation-base approach, and it cost more as the size of the set increases. However, this is not too bad when we compare to the delta-base approach with no garbage collection. Because, in the last approach, the causal context grows with the size of the operations made the merge operation takes a huge amount of time searching for a tag in the causal history to know if the element should exists or be removed.

With this analysis we can explain the results obtained (see Figure 3.3) for the merge operation. The test consists in two replicas with an equal initial state that make different N operations an in the end they converge. For the initial state both replicas made 300 equal operations. Then each replica executed N different operations to simulate divergence. The operations performed were *add* and *remove* with 75% chance of adding, which means the size of the set should increase. In the end we measured the time it took to converge, either by applying all the N operations at once (operation-based approach) or by merging the two states/deltas. The graph shows the time it takes for one replica to apply N operations made by the other replica. As we can see, applying a buffer of operations is not so expensive as making one merge operation, for example, applying 400 operation costs more or less $12\mu s$, one merge of two states that diverge in 400 operations costs $144\mu s$ and one merge of two deltas that diverge as well in 400 operations costs $442\mu s$.

These two results, from memory stored and from complexity, indicate that the operation-based approach is always better. But, if we take into account the memory that takes to buffer the N operations, maybe the other approaches are worth to use.

3.1.3 Relevance of the Object Size

In this first experiment, clients access a single object, whose size varies, and propagate updates immediately, i.e., there will be no SLA. The goal is to assess for which object sizes one approach is better than the other.

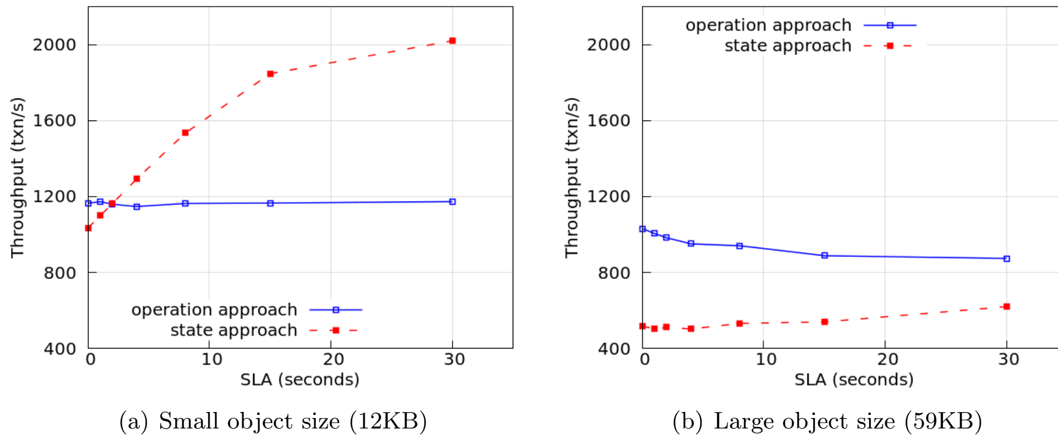


(a) Throughput (b) Latency

Figure 3.4: Impact of the object size for both approaches

Figure 3.4 shows the impact of the object size when propagating updates among two different datacenters using both operation-based (dotted lines) and state-based approaches (continuous lines). The plot on the left shows the system throughput (measured in number of transactions per second). The plot on the right shows the time it takes for operations to be applied in the remote preferred datacenter. The results are somehow unsurprising, and support the design decisions of the SwiftCloud developers. When SLAs are not taken into account, the operation-based approach always offers the best performance, both in terms of throughput and latency.

About the time to apply an update, we noticed that there is a drop in this value. At a first look it may look contradictory, because the object size is increased. However, the main reason is the drop in the throughput. Because the clients now take more time to perform transactions, the server has more cpu available (less thread scheduling) to perform the merge operation. Yet,



(a) Small object size (12KB)

(b) Large object size (59KB)

Figure 3.5: Relaxing data freshness requirements

the server is not able to support more clients due to higher activity in the network.

3.1.4 Impact of Buffering and using an SLA

We now study the impact of buffering updates and delaying its propagation to other datacenters. In this experiment, we fix the object size and vary the SLA. The impact of the SLA on the operation-based approach is that multiples updates can be batched and sent in a single message to other datacenters. This may yield some improvement on the network utilisation. For the state-based approach, this means that a single state-update is sent instead of multiple operations. The goal of this experiment is to understand, for different object sizes, how many updates need to be buffered to compensate the additional overhead of sending the entire state. The intuition is that the weaker the requirements of the application (i.e. more updates can be batched), the better the state-based approach performs.

The results for a database of objects with a fixed size of 12KB are depicted in Figure 3.5. The y axis shows the overall system throughput for different SLAs. The larger the SLA constraint on the x axis, the more relaxed the SLA is, and more updates can be batched. The initial value of SLA=0 corresponds to the original SwiftCloud implementation without any batching.

Interestingly, even for very small relaxations of the SLA, the state-based approach outperforms the operation-based approach. In fact, significant throughput improvements can be achieved if the application can tolerate a certain amount of staleness in the observed data. For

instance, for an SLA of 15s the throughput of the system can be 50% higher if the state-based approach is used.

An additional conclusion extracted from this experiment is that no advantages can be obtained due to relaxed SLAs in the operation-based approach. Although the relaxed SLA allows to batch multiple updates in a single message, the fact that all operations need to be applied serially, via a computationally expensive procedure, counterbalances this potential benefit. One potential avenue for research, that has not been explored in this work, is to find techniques to merge operation-based updates based on semantic information and combine this with techniques to apply updates in parallel.

3.1.5 Impact of Inter-Object Dependencies

In the previous experiments we have used a single object. We now experiment with different objects, each with a different SLA. In particular we will take a look at the operation-based solution. The goal is to understand how potential causal dependencies that are established by the way SwiftCloud manages the objects metadata may interfere with the staleness of the object.

More precisely, we consider a SwiftCloud system with two different sets of clients. The first set of clients accesses exclusively object O_1, which has an SLA of 5s (i.e., clients tolerate reading data that is 5s stale). The second set of clients accesses exclusively object O_2, which has an SLA of 8s. In our experimental setting, with two datacenters, and 2 clients per DC, we are able to batch 476 operation for object O_1 and 728 operations for object O_2.

Figure 3.6 plots all measured times that took to process the queue of batched updates, after they are received from a remote datacenter. Without inter-objects dependencies and with such low amount of updates, the processing time for any of the objects is also low. However, as it may be observed from the results, the time it takes to empty O_1's operation queue can be as large as 8s, i.e., as large as the SLA for the object O_2. The same happens in the other way, object O_2 can take up to 5s, the SLA time of the object O_1. This behaviour is easily explained by the false interdependencies created by SwiftCloud metadata: some of the operations for object O_1 depend on operations for object O_2. Since the propagation of O_2 operations can be postponed by 8s, O_1 operation may be stalled for an amount of time that depends on the SLAs of other objects. This experiment shows that it may be very hard to support clients with different SLAs

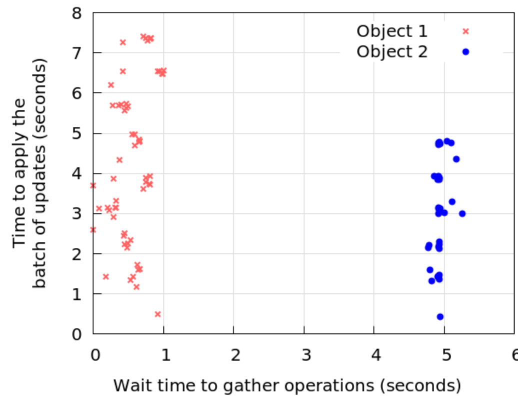


Figure 3.6: Time required for the synchronization of an object.

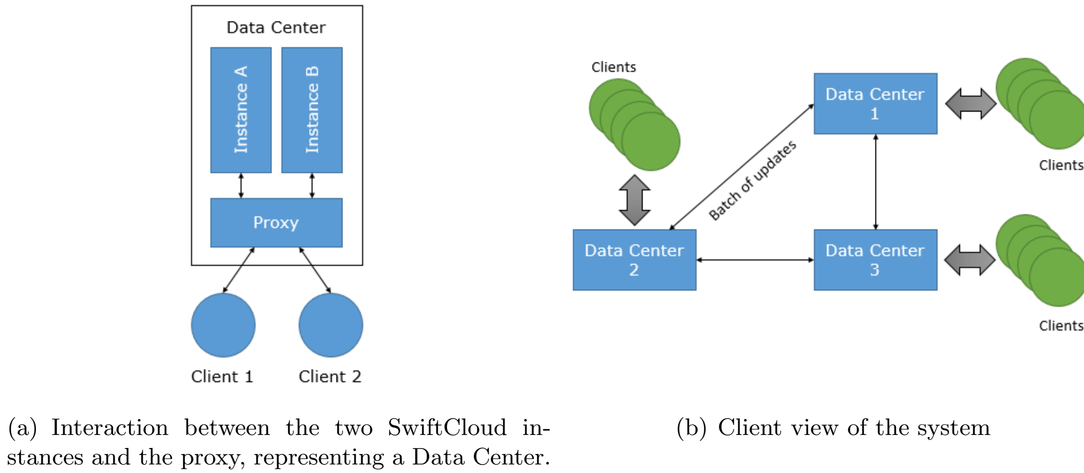
in the current implementation of SwiftCloud.

3.2 Overview of the System

As it was described in the previous chapter, when updates need to be propagated as soon as possible, the operation-based approach outperforms the state-based approach; this justifies the original SwiftCloud implementation. However, when clients are willing to tolerate reading information that is slightly stale, significant gains can be obtain by using the state-based approach. Also, for clients that can tolerate some latency, as long as it is small, the best approach might depend on the size of the object.

Our conjecture is that benefits may be achieved by supporting *both* approaches simultaneously, such that for some realistic SLAs, an operation-based approach is used for larger objects and a state-base approach is used for smaller objects. Notice that the number of updates received by each object within a SLA time window also plays an important role deciding between approaches. To validate our hypothesis, we have developed a prototype of an hybrid system, that we have named Bendy, that is able to adapt the approach used for each object dynamically. This section reports on the design of this prototype and on the experimental results that validate our assumption.

In order to reduce the amount of metadata that is stored by clients, SwiftCloud maintains vector clocks that encode updates for multiple objects. This makes extremely hard to support simultaneously state-based and operation-based implementations for different objects, due to the interdependencies among both implementations (Subsection 3.1.5). Since the goal of this thesis



(a) Interaction between the two SwiftCloud instances and the proxy, representing a Data Center.

(b) Client view of the system

Figure 3.7: Overview of the described system.

was not to perform a major redesign of SwiftCloud but, instead, explore the advantages of using both approaches (and, in this way, provide experimental evidence to support such redesign) we have used a pragmatic approach when architecting our prototype. Bendy is, internally, composed of two independent SwiftCloud instances, which are hidden from the client by an extended SwiftCloud proxy.

One instance is the original, operation-based SwiftCloud implementation as described in (Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and Preguiça 2013). Another instance is the version that we have produced that uses exclusively state-based CRDTs. Bendy makes dynamic decisions about which approach is more favourable for a given object, based on its size, on the associated SLA, and on the workload characterisation; the object state is stored in one of the instances accordingly. If, at some point, the implementation of an object needs to be changed in runtime, the state of the objects is transferred from one instance to the other, and the proxies updated such that the current implementation is used. Figure 3.7(a) show the interactions between the two instance with the proxy, and figure 3.7(b) represents the architecture what was described above.

Although this approach is somehow rudimentary, and more efficient switchover between implementations could be supported by redesigning the metadata structure of SwiftCloud from scratch, we have opted not to do so for two reasons: first, the entire redesign of SwiftCloud is a challenge on its own, and outside the scope of this work; second, by using the current approach

it is possible to make direct comparisons with the original implementation (i.e, all observed benefits derive from the use of dynamic adaptation, and not from other optimisations of the SwiftCloud middleware).

Note that the extended proxy maintains the metadata for both instances such that all accesses to a given object respect causality, regardless of the instance where the object is currently stored. Also, Bendy is focused only on the propagation of updates on the server side, and because of that, the remaining implementation of the clients is preserved from the original SwiftCloud system.

3.3 Taking SLAs Into Account

In Bendy, one SLA will be associated for all object. However, due to different access ratio of some objects, some updates are propagated as soon as possible, as in the original SwiftCloud implementation. The SLA is used mainly to decide when updates need to be propagated.

When deciding if an object should use an operation-based or a state-based approach, Bendy takes into account the global SLA, the object size, and the workload characterisation for that object, more precisely how many updates are expected to be performed during a period that corresponds to the SLA. If the ratio between the number of expected updates and the object size is above a given threshold, a state-based approach is used. Otherwise the default operation-based approach is used.

From the results presented in Subsection 3.1.4, we could observe that no significant advantages can be extracted from batching operation-based updates. Also, from Subsection 3.1.5 we have seen that delaying operation-based updates may cause other updates to be stalled. Thus, for objects that are selected to use an operation-based approach, updates are propagated as soon as possible, exactly as in the original SwiftCloud implementation.

Also from Subsection 3.1.4, one could observe that significant gains can be obtained by delaying the propagation of updates in the state-based approach. Therefore, in Bendy, we use the following strategy. Let δ_o be the latency tolerated for object o according to the corresponding SLA. Let μ_o be *propagation time*. Let t_o^u be the time at which some update u_o on object o has been performed at a given datacenter. To ensure that the object SLA is not violated, Bendy

forces the propagation of that update (and all subsequent updates that have been buffered) at the following instant.

As it will be discussed in the next section, the propagation time μ_o depends not only on the size of the objects, but also on the number of updates that are propagated simultaneously. Thus, as discussed before, the value μ_o is adjusted in runtime, based on values measured during the last synchronizations.

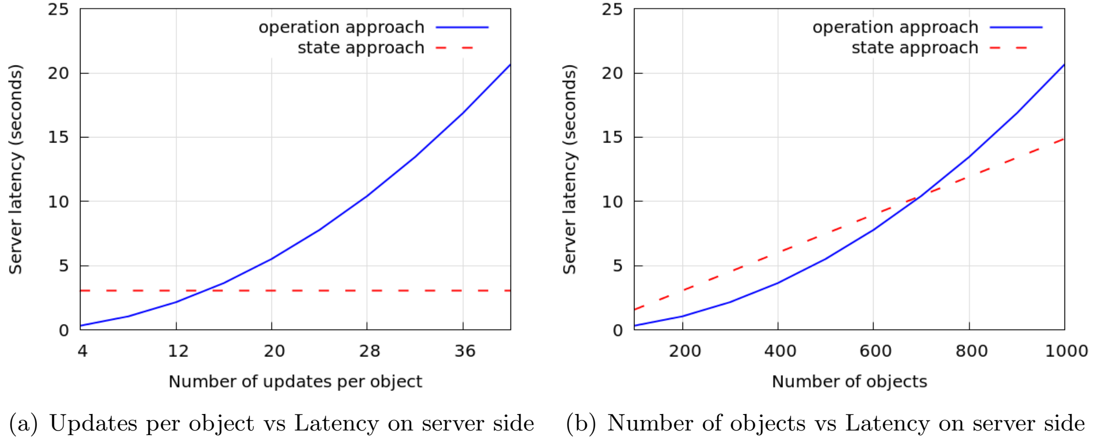
3.4 Propagation Time

SLAs can be exploited to batch updates, a technique that can bring significant advantages for state-based approaches. However, batched updates need to be propagated *before* the SLA expires, with enough slack to transmit and deploy the updates at remote sites without violating the SLA. Therefore, the assessment of the time needed to propagate and apply updates, that we denote the *propagation time*, is of critical importance for any system that aims at exploring relaxed SLAs.

From our experience with SwiftCloud, we observed that the number of objects and the number of updates per object are the main variables that affect the propagation time. This is illustrated by the following experiments where we have fixed the SLA and object size and vary two other parameters, namely the number of objects and the number of updates batched: (i) in the first experiment, we fixed the number of objects to 200 and we vary the number of updates; (ii) in the second, we fixed the number of updates to 8 per object and we vary the number of objects.

Figure 3.8(a) depicts the results from the first experiment. As expected, the number of batched updates are irrelevant in the state-based approach (as a single state per object is sent before the SLA expires). On the contrary, the propagation time increases significantly with the number of batched updates, in the operation-based approach. Interestingly, it can be observed that the propagation time grows super-linearly, because the system resources become exhausted in two many updates are queued to be applied at once. This further reinforces the observation that, at least in the current implementation of SwiftCloud, it is hard to extract benefits from relaxed SLAs with the operation-based approach.

Results for the second experiment are depicted in Figure 3.8(b). Unsurprisingly, the prop-



(a) Updates per object vs Latency on server side (b) Number of objects vs Latency on server side

Figure 3.8: Propagation Time

agation time increases with the total number of updates but the interesting aspect is that the increase is linear with state-based approach while the operation-based approach depicts, as above, a super-linear degradation. However, these figures clearly show that, when computing the propagation time, one need to have a estimate of the total number of objects that are using batching, in order to compensate for the burst of updates that can result from having SLAs for multiple objects expiring simultaneously.

Since the parameters identified above, that influence the propagation time, are hard or impossible to predict offline, in Bendy we keep in runtime statistics for the propagation time for the subset of the top-k objects that use a state-based approach.

In this prototype, the propagation time t_x^i in round i, for each of the objects O_x that use state-based, is computed dynamically as follows:

$$t_x^i = t_{u_x}^i + \delta - \mu_x^i$$

where $t_{u_x}^i$ is the time at which O_x received the first update operation u_x for this round; δ is the visibility delay tolerated by the application (specified in the SLA) and; μ_X^i is a moving average over a window of the time that the last n coordination procedures took.

3.5 Dynamic Adaptation

Recent work on adaptive storage systems(Couceiro, Chandrasekara, Bravo, Hiltunen, Romano, and Rodrigues) provides evidence that substantial gains can be achieved without performing fine-grain adaptation of every and single object in a storage system. In fact, many realistic workloads follow a zipfian distribution, where some objects are accessed much more often than the others. Thus, a large fraction of the gains can be achieved by adapting only the implementation of those popular objects.

3.5.1 Reconfiguration

Based on these observations, Bendy performs a top-k analysis of the workload and only adapts the implementation of the most popular object. All other objects just use the default SwiftCloud implementation. The state-of-the-art stream analysis algorithm (Metwally, Agrawal, and El Abbadi 2005) permits to infer the top-k most frequent items of a stream in an approximate, but very efficient manner. Given that workloads may change in run-time, the top-k analysis is repeated periodically. The Algorithm 3.1 describes the whole process. At the end of each period, Bendy first reverts back to the default (operation-based) implementation all objects that are no longer part of the top-k. For those objects in the top-k. Bendy selects the target implementation using the criteria described above. Finally, Bendy reconfigures the implementation of those objects for which the target implementation differs from the current implementation in use.

Given that Bendy is implemented as a wrapper, the reconfiguration of a given object is implemented by migrating that object from one SwiftCloud instance to the other. In this process, N objects are migrated, and to ensure consistency, we lock the access to an object that is being reconfigured to the other implementation. The action of locking access to an object might have impact on the throughput, specially if we lock the access to all the objects, that are being migrated, at the same time. To avoid such drawback, the system can migrate m objects at a time until all N objects are migrated. Being m a fraction of N. More details are provided in section 4.4 with experiments that compare different values for m.

Algorithm 3.1 Process of migrating objects from one instance to the other.

```
1: function Migrate
2:     O :=GetOldTopK()
3:     E :=GetNewTopK()
4:     N := (O \ E) ∪ (E \ O)                    ▷ N = all objects to be migrated
5:     while N ≠ ∅ do
6:         m ⊆ N
7:         LockObjects(m)
8:         MigrateObjects(m)
9:         UnlockObjects(m)
10:        N := N \ m
11: function MigrateObjects(m)
12:     for id in m do
13:         o :=FetchObject(id, replicaA)
14:         o :=Reconfigure(o)    ▷ Each type of object implements its own mechanisms to
        change from one approach to the other
15:         StoreObject(id, o, replicaB)
16: function LockObjects(m)
17:     for id in m do
18:         LockClientAccess(id)
19:     FlushPendingUpdates()
20: function UnlockObjects(m)
21:     FlushPendingUpdates()
22:     for id in m do
23:         UnlockClientAccess(id)
```

3.5.2 Selecting the Right Implementation

Bendy requires several statistics about the objects and the workload to be maintained, such as the object size, the update ratio, and the time it takes to propagate and apply state updates. To keep those statistic for every object in the storage system may cause an unnecessary overhead. Also, client proxies need to be aware of which instance stores a given object.

When deciding if an object should use an operation-based or a state-based approach, Bendy takes into account the SLA, the object size, and the workload characterisation for that object, more precisely how many updates are expected to be performed during a period that corresponds to the SLA. If the ratio between the number of expected updates and the object size is above a given threshold, a state-based approach is used. Otherwise the default operation-based approach is used.

3.6 Implementation Issues

During the development of the system, many decisions were made to implement the features that were described above. The main features are: the implementation of state approach, the combination of the two approaches operation- and state-based and the dynamic system.

3.6.1 Implementation of the state-based approach

As it was discussed before, we started our project with an implementation of SwiftCloud that only supports operation-based CRDTs. The first step was to implement a state-based solution to make the first comparisons between the two approaches. Thus, we based our implementation on the algorithms described in Section 2.5 and managed to implement the merge function. For that, we've extend the CRDT interface and changed the internal representation of the objects like in the algorithms. However, we've only changed how the updates where propagated on the server side, because we assumed that one client will never make enough updates that justifies the sending of a state. Instead, we look at the client as an entity that makes operations remotely and not as a replica, like in the original SwiftCloud. We've kept all the features of the Client and focused only on the server side.

The second part of implementing the state-based approach was the propagation of updates. At the same time we did the propagation of updates we started to add the notion of SLA. Since, in the original SwiftCloud, the updates were sent asynchronously, it was not hard to implement the SLA. Basically we've added a delay between synchronizations to the synchronization thread. Later, we've implemented the formula described in Section 3.4 to ensure that the SLA was satisfied. About the propagation of updates, we had to implement the merge of the replica's clock to ensure that clients were able to see updates from other replicas. Although in the specification of the state approach it is not required to have a causal clock outside of the object, we've found out that it was harder change completely the SwiftCloud implementation than keeping the clock. Besides that, we intended to integrate with the operation-based approach which needs that global clock.

3.6.2 Combining operation-based and state-based approaches

After the implementation of the state approach, we tried to put together both approaches in the instance. However, that was impossible due to the complexity of the global causal clock and because, when merging an object, the merge of the clock sometimes it contain the timestamps of other operations. This means that after making a merge of an object and its clock, the operations that were on hold to be applied will be discarded because the new clock already includes the timestamp of those operations. We've looked at this problem for a long time, and the best solution was to separate the two implementations into two instances. Otherwise, we wouldn't have time to work on the dynamic part of the system.

To hide the two instances from the client we created a proxy. This proxy basically forwards the client's request to the instance that has the object. For the client there were no changes. We kept the same interface so that the client can make operations regardless of the propagation method that each instance uses.

3.6.3 Implementing a dynamic system

Regarding this topic, most of the details are explained in the Section 3.5. Nevertheless, there is one aspect that needs to be explained with more detail. Regarding the migration of an object from one approach to the other, we had to make it in a simple manner. Because we have two instances, the causal clocks of each instance will grow independently. However, we need to move one object from one side to the other and changing the clocks was not easy. The best solution was to momentarily lock the object on the two instances for all clients except one. There is one client that belongs to the system and it is in charge of migrate objects. This client, contacts directly the instance instead of contacting the proxy like the other clients. It reads the value on the original instance, removes the object from that instance (by removing its the elements in the case of the set) and applies that value on the other instance. Finally we flush all buffers to ensure that all replicas have migrated the object and we unlock the object to all clients.

Summary

In this chapter we have been through the design and implementation of Bendy. First we showed an overview of the system, comparing with SwiftCloud. Then, we presented more details about the particularities of the system, like the adding of an SLA, the propagation of updates and the dynamic adaptation.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

This chapter presents an evaluation of Bendy. We first present our experimental setting, where we list the systems we use to compare Bendy with. Then we present experiments comparing the throughput and the bandwidth used by each of the systems. We finally evaluate how Bendy is able to adapt to changes in the workload.

4.1 Experimental Setup

Each experiment uses 3 datacenters. Each datacenter replicates the full key-space which is composed by 1000 objects for the experiments in Subsections 4.2 and 4.3, and 50000 objects for the experiments in Subsection 4.4. We run a total 600 clients, having 200 client associated with each datacenter. In order to generate the workload, we use a zipfian distribution. All objects have associated a SLA of $10s$. We first run a warm-up phase were the database is populated. Then, each experiment runs for 5 minutes.

In our experiments, we compare the following three systems:

- An unmodified version of SwiftCloud (**op-based** hereafter) that implements an operation-based dissemination process. Updates are propagated immediately without batching them.

- A modified version of SwiftCloud (**state-based** hereafter) that implements a state-based dissemination process. Updates are batched based on the formula presented in Subsection 3.4.

- Bendy which is a mixed approach that follows the specifications presented in Chapter 3. It is approximately 2200 lines of Java. Bendy is built on top of SwiftCloud.

4.2 Throughput

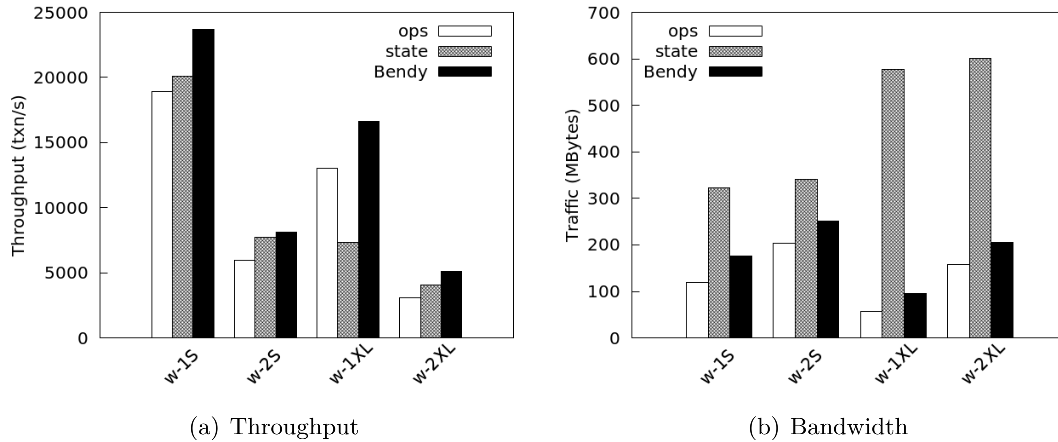
In the following experiment we aim at comparing the three systems in terms of throughput. Since Bendy runs two independent instances of SwiftCloud, one implementing operation-based and other one implementing state-based, bridged by an extended proxy, we opted for running the experiments in two instances also for the other systems. This allows us to achieve comparable results; even though op-based and state-based systems could run the whole experiment using a single instance since they do not mix approaches.

Figure 4.1(a) show the results for two different workloads: (i) a read-dominated workload with only 5% of updates (Workload-1); and (ii) a balanced workload with an equal number of reads and updates (Workload-2). These are two of the workloads specified by the widely used YCSB(Cooper, Silberstein, Tam, Ramakrishnan, and Sears 2010) benchmark and are representative of the workloads imposed by typical applications of geo-replicated storage systems. In addition, for each of the workloads, we experiment with two different object sizes: S (12KB) and XL (30KB). Bendy, due to the parametrization of the zipfian distribution generator and our top-k analysis, optimizes the dissemination process of 1% of the total number of objects. In these experiments, we are forced to limit the number of objects to 1000 since the state-based approach starts struggling and becoming very unstable with a large number and size of objects, due to the amount of information that needs to ship in every *coordination procedure*. Operation-based and Bendy do not suffer from this problem but in order to have comparable results across systems we limit the number of objects also for them. In the experiment in Subsection 4.4 we use a more realistic amount of objects (50000) to demonstrate that our system does not suffer from this problem.

The results show that Bendy always outperforms the other two systems (up to 127% in some cases). The reason is because our system does not reach the bottleneck stage that the

	ops	**state**	**Bendy**
Workload-1S	1 523	11 640	267
Workload-2S	377	5 290	237
Workload-1XL	1 238	55 119	799
Workload-2XL	492	63 120	729

Table 4.1: Coordination time (average)



(a) Throughput (b) Bandwidth

Figure 4.1: Throughput vs Bandwidth

other solutions have to face. The state-based solution has to struggle with large-sized objects while the operation-based approach struggles with the high amount of operations issued in a few objects. This experiment validates our hypothesis, and demonstrates that benefits can be achieved by having a hybrid system.

In this experiment, the state-based approach seems to, most of the times, outperform the operation-based approach. However, for the state-based approach, the SLA of $10s$ was only satisfied once as we can see in the Table 4.1. The main reason for not being able to satisfy the SLA was the large amount of states that are processed at the same time and its size. Nevertheless, as Section 3.1 unveils, this directly depends on the workload, the SLA, and the size of the objects.

Finally, one more subtle conclusion one can extract from this experiment is that, with the state-based approach, one must take special care to avoid violating the SLA. We notice that, as soon as the size and the number of objects increases, the accumulated processing time for all state-updates also increase. Thus, a naively configured state-based approach can easily start violating the target SLA. For instance, Table 4.1 lists the average time that the *coordination procedures* took for each of the experiments. One can see that the state-based solution is only able to satisfy the SLA of $10s$ for the one of the experiments (Workload-2S), violating in the other three, e.g. for Workload-2XL, it takes more than $60s$ on average. This reinforces the importance of applying the state-based approach just to a small number of top-k objects, that have the bigger impact on the system performance.

4.3 Bandwidth Utilization

With the same experiment, we also measured the bandwidth usage of the different approaches. Figure 4.1(b) shows the amount of bandwidth, in MB, used by each of the approaches for the entire run of five minutes. The experiment configuration and the used workloads are equivalent to the ones presented in the previous subsection.

The results match our analysis. As expected, the pure state-based system, is by far the worst solution. In many cases this system sends the state of objects that only received 2 or 3 operations, which is not efficient. Regarding Bendy, it uses more bandwidth than the pure operation-based system. Nevertheless, as demonstrated before, we reach better throughput mostly because of (i) the benefit of batching updates, and (ii) the inter-object dependencies problem of the operation-based system described in Subsection 3.1.5.

4.4 Dynamic Behavior

In previous experiments, we have compared the throughput of all the approaches at a stable point. However, in a real setting, the workload may change dynamically. Therefore, in this subsection, we describe an experiment where we induce a dynamic change in the workload by changing the most accessed objects. Our goal is to assess how well Bendy adapts to the changes and how the adaptation penalizes the throughput provided by Bendy.

For this experiment, we use a balanced workload with an equal number of reads and updates. The experiment goes as follows: during the first 100 seconds, the system is stable, meaning that no changes in workload are introduced; then the most accessed objects are changed. This forces Bendy to migrate a total of 700 objects between instances in order to optimize the newly identified top-k objects. We compare four variations of Bendy:

i. A version that does not adapt to the new workload (*baseline*);

ii. A version that migrates all objects at once (*All at once)*;

iii. A version that migrates the objects in groups of 50 (*50 by 50*);

iv. A version that migrates the objects in groups of 10 (*10 by 10*).

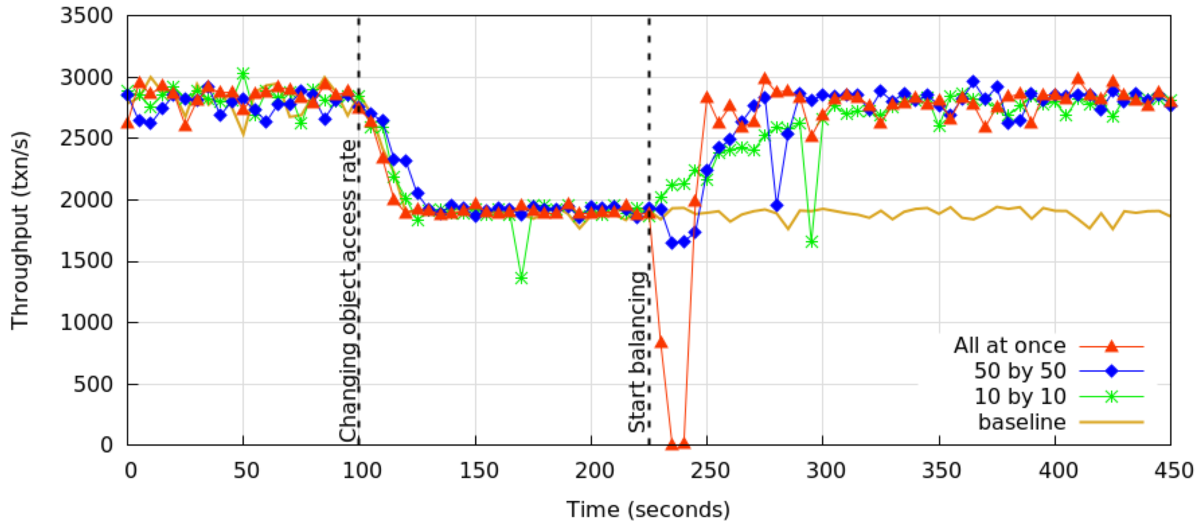


Figure 4.2: Throughput of one DC during the balancing process of top-k objects.

Figure 4.2 shows the throughput of each of the variations. One can see that, after changing the workload (100th second), all variations of Bendy degrade their performance reducing its throughput almost 33%. This is because the objects being optimized are not highly accessed anymore and the new hot objects are using basic operation-based approach, which does not benefit from batching. Then, after 125 more seconds (225th second), the migration process is started. As expected, if we move all objects at once the throughput drops drastically until the balancing process ends. However, if we migrate a small amount of objects at a time, the process may take longer but the throughput loss is minimal. We can conclude that moving by groups of around 50 objects is a reasonable solution. Although the throughput initially drops (about 20%), it recovers quite fast, achieving maximum throughput in less than 50 seconds since the balancing process started.

Another aspect of our dynamic system is the ability to detect that the top-k list has changed. As in all autonomic system, there is a tradeoff between how fast the system reacts to changes and the likelihood of the new state to be stable. Since we consider this problem orthogonal to this thesis, we decided to adopt a simple approach, in which we wait for some extra time once the change has been detected, before starting the adaptations procedure of Bendy. Of course, we could adopt more sophisticated techniques in order to make Bendy more robust to transient workload oscillation, such as techniques to filter out outliers (Hodge and Austin 2004), detect statistically relevant shifts of system's metrics (Page 1954), or predict future workload trends (Kalman 1960).

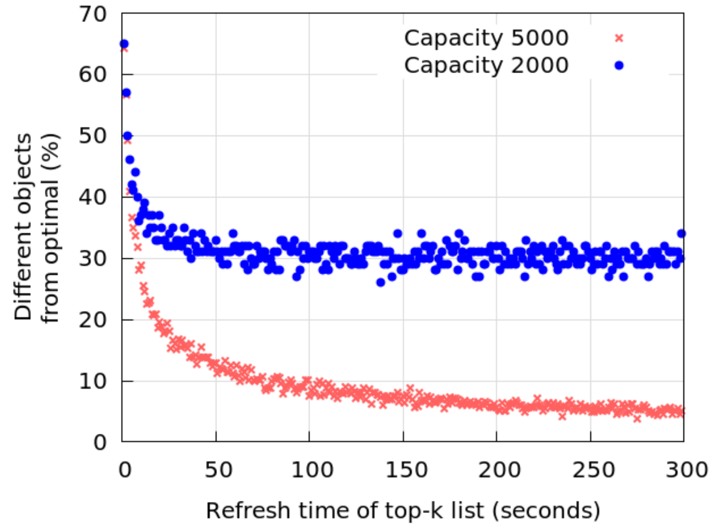


Figure 4.3: Percentage of objects in a top-k list that are different from the optimal top-k list.

In order to get some insights on the time we have to wait until considering a change in the workload stable, we evaluate the time that our top-k analysis needs in order to propose a top-k list close to the optimal. For the implementation of top-k that was used, there is a variable called *capacity* which is the number of events that happened in the past. In our solution, each event is an access to an object. This means, if the algorithm uses a large capacity then we will have more accuracy in the result. However, if the capacity is extremely large, we will waste a lot of resources in terms of memory and it should take more time to process a list of top-k objects. Figure 4.3 shows the results of the experiment. As one can see, a larger capacity brings more accuracy, and the top-k analysis rapidly starts proposing almost an optimal list of hot objects (only 10% of error) in barely 75 seconds. This justifies the 125 seconds time window used in the previous experiment before adapting the system.

4.5 Discussion

The results obtained with Bendy, clearly show that significant benefits can be achieved if multiple CRDT implementations are supported in a single system, and the best implementation is used according to the user SLA and the workload characterization. This opens the door for new avenues of research, in the design and implementation of systems that can provide such functionality in an much more integrated manner, than that provided by the wrapper approach followed in the Bendy design. In the following paragraphs, we discuss a number of insights

and guidelines, that we gained from our experience with the evaluation SwiftCloud and the implementation of Bendy, that may be useful when building future systems:

- If updates need to be pushed as soon as possible, the current SwiftCloud operation-based approach excels. However, if clients can tolerate stale data, significant gains could be achieved by supporting state-based propagation of updates.
- With the current SwiftCloud implementation, no significant advantages can be extracted from batching multiple operation-based updates. This happens because SwiftCloud applies all the batched operations serially and independently (for instance, releasing and grabbing locks for every single operation in the batch). There is room to optimize SwiftCloud for more efficient processing of batched updates. Also, semantic compression of batched updates, as suggested by in (Almeida, Shoker, and Baquero 2014), would also improve the system.
- The way SwiftCloud compresses client metadata, namely by using a single clock shared by all objects, makes hard, if not impossible, to support multiple distinct SLAs in an effective manner, as the interdependencies that are created among updates may cause SLAs to be violated. Techniques that are more costly, but that allow for more fine-grain tracking of dependencies (such as using per-object clocks (Lloyd, Freedman, Kaminsky, and Andersen 2011)) are needed to effective support multiple SLAs.

Taking into consideration our experience, implementations that aim at outperforming Bendy should use the following guidelines:

- To rely on CRDT implementations that support both approaches, like the Optimized OR-Set (Bieniusa, Zawirski, Preguiça, Shapiro, Baquero, Balegas, and Duarte 2012).
- To use metadata maintenance techniques that avoid creating false dependencies among objects using different SLAs.
- To use metadata maintenance techniques that ensure that causality information regarding objects using different implementations is not compressed together, as this creates undesirable dependencies among both implementations.

- Embed in the transaction processing engine sensors that may simplify the task of extracting the workload characterisation, namely the update rate, given that the benefits of the state-based over the operation-based critically depend on the possibility of aggregating multiple updates before the SLA expires.

Summary

In this chapter we introduced the experimental evaluation made to Bendy and its results. First we started by comparing the system with the two generic solutions: operation-based and state-based solutions. The comparisons where in terms of Throughput, Bandwidth usage and taking into account if the SLA was satisfied. Then we performed some analysis to our system in a simulated environment. In that environment we could simulate changes in the top-k elements and observe the system balancing itself to recover the maximum throughput. Finally, we made some micro-benchmark to analyse the time for the system to properly detect a change in the top-k distribution.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions

5.1 Conclusions

In this thesis we have analyzed the cost performance tradeoffs between the two main approaches that can be used to propagate updates in geo-replicated stores using CRDTs, namely operation-based and state-base approaches. Our work shows that node of the approaches outperforms the other in absolute terms, and that an hybrid system may yield the best results. In particular, we have show that, if the application is willing to tolerate small amounts of staleness when reading objects, significant throughput gains can be achieved by encoding multiple updates in a single state-update.

In order to validate our hypothesis, we have presented and evaluated Bendy, a CRDT-based geo-replicated storage system that supports both operation- and state-based approches. Bendy is able to optimize object-wise for a bounded number of objects (hot objects) the dissemination process. Plus, Bendy is able to react to changes in the workload by relying on an approximate, but very efficient, state-of-the-art stream analysis algorithm. Our results have shown that Bendy outperforms solutions that use only one of the two approaches, and that is capable of rapidly self-adapt to variations in the workload.

In the process of implementing and experimenting with both approaches, using SwiftCloud, a state of the art CRDT-based geo-replicated storage system, we were also able to get interesting insights that may help in driving future implementations of similar systems. In particular, designers need to take special care on providing support for efficient processing of batched operation-based updates, fast state-updates, and carefully crafted metadata structures that avoid undesirable false causal dependencies among objects.

5.2 Future Work

As future work, there is the possibility of using machine learning to better decide the approach to use for each object, instead of using static analysis. However, this is not a trivial solution because in order to make a decision correctly, a accurate data model must be produced. This means, the training data should have a representative amount of possible configurations and the best approach for each configuration.

Another aspect for future work is to remove the two instances of CRDTs and use only one that supports both approaches. For this solution, a deep change in the metadata of the system is needed to correctly support both approaches.

References

Almeida, P. S., A. Shoker, and C. Baquero (2014). Efficient state-based crdts by delta-mutation. *CoRR abs/1410.2803*.

Baquero, C., P. Almeida, and A. Shoker (2014). Making operation-based crdts operation-based. In K. Magoutis and P. Pietzuch (Eds.), *Distributed Applications and Interoperable Systems*, Lecture Notes in Computer Science, pp. 126–140. Springer Berlin Heidelberg.

Bieniusa, A., M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte (2012). An optimized conflict-free replicated set. *CoRR abs/1210.3368*.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, New York, NY, USA, pp. 7–. ACM.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, pp. 143–154. ACM.

Couceiro, M., G. Chandrasekara, M. Bravo, M. Hiltunen, P. Romano, and L. Rodrigues. Q-opt: Self-tuning quorum system for strongly consistent software defined storage. In *Proceedings of the 16th International Middleware Conference*, Middleware '15, New York, NY, USA. ACM.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels (2007a). Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, pp. 205–220. ACM.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels (2007b, October). Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev. 41*(6), 205–220.

Hodge, V. and J. Austin (2004). A survey of outlier detection methodologies. *Artificial Intelligence Review 22*(2), 85–126.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering 82*(1), 35–45.

Klophaus, R. (2010). Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, New York, NY, USA, pp. 14:1–14:1. ACM.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*(7), 558–565.

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

Letia, M., N. M. Preguiça, and M. Shapiro (2009). Crdts: Consistency without concurrency control. *CoRR abs/0907.0929*.

Lloyd, W., M. J. Freedman, M. Kaminsky, and D. G. Andersen (2011). Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, pp. 401–416. ACM.

Metwally, A., D. Agrawal, and A. El Abbadi (2005). Efficient computation of frequent and top-k elements in data streams. In T. Eiter and L. Libkin (Eds.), *Database Theory - ICDT 2005*, Volume 3363 of *Lecture Notes in Computer Science*, pp. 398–412. Springer Berlin Heidelberg.

Navalho, D., S. Duarte, N. Preguiça, and M. Shapiro (2013). Incremental stream processing using computational conflict-free replicated data types. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, New York, NY, USA, pp. 31–36. ACM.

Oster, G., P. Urso, P. Molli, and A. Imine (2006). Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, New York, NY, USA, pp. 259–268. ACM.

Page, E. S. (1954). Continuous inspection schemes. *Biometrika 41*(1/2), pp. 100–115.

Preguica, N., J. M. Marques, M. Shapiro, and M. Letia (2009). A commutative replicated

data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, Washington, DC, USA, pp. 395–403. IEEE Computer Society.

Shapiro, M., N. Preguiça, C. Baquero, and M. Zawirski (2011, January). A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506.

Shapiro, M., N. Preguiça, C. Baquero, and M. Zawirski (2011). Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, Berlin, Heidelberg, pp. 386–400. Springer-Verlag.

Terry, D., V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh (2013, November). Consistency-based service level agreements for cloud storage. In *Proceedings ACM Symposium on Operating Systems Principles*. ACM.

Terry, D. B., A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch (1994). Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, Washington, DC, USA, pp. 140–149. IEEE Computer Society.

Zawirski, M., A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça (2013). Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR abs/1310.3107*.