Valter Balegas de Sousa

Licenciatura em Engenharia Informática

Key-CRDT Stores

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Nuno Manuel Ribeiro Preguiça,
Professor Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:	Prof[a]. Doutora Carla Ferreira
Arguente:	Prof. Doutor Luís Veiga
Vogal:	Prof. Doutor Nuno Preguiça

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2012

Key-CRDT Stores

Ao Mané.

Acknowledgements

I would like to thank my advisor Prof. Dr. Nuno Preguiça for the opportunity to participate in this project and to encourage me to proceed with the doctoring degree. This work was partially supported by project PTDC/EIA-EIA/108963/2008 and a Google Research Award, which i would like to thank for the grants given.

I want to mention the collaboration with other colleagues of this department and the members of Concordant Project. At last, i would like to thank to my mother, father and brother for their comprehension and patience with my bad mood when the work left me distressed.

Abstract

The Internet has opened opportunities to create world scale services. These systems require high-availability and fault-tolerance, while preserving low latency. Replication is a widely adopted technique to provide these properties. Different replication techniques have been proposed through the years, but to support these properties for world scale services it is necessary to trade consistency for availability, fault-tolerance and low latency. In weak consistency models, it is necessary to deal with possible conflicts arising from concurrent updates. We propose the use of conflict free replicated data types (CRDTs) to address this issue.

Cloud computing systems support world scale services, often relying on Key-Value stores for storing data. These systems partition and replicate data over multiple nodes, that can be geographically disperse over the network. For handling conflict, these systems either rely on solutions that lose updates (e.g. last-write-wins) or require application to handle concurrent updates. Additionally, these systems provide little support for transactions, a widely used abstraction for data access.

In this dissertation, we present the design and implementation of SwiftCloud, a Key-CRDT store that extends a Key-Value store by incorporating CRDTs in the system's data-model. The system provides automatic conflict resolution relying on properties of CRDTs. We also present a version of SwiftCloud that supports transactions. Unlike traditional transactional systems, transactions never abort due to write/write conflicts, as the system leverages CRDT properties to merge concurrent transactions. For implementing SwiftCloud, we have introduced a set of new techniques, including versioned CRDTs, composition of CRDTs and alternative serialization methods.

The evaluation of the system, with both micro-benchmarks and the TPC-W benchmark, shows that SwiftCloud imposes little overhead over a key-value store. Allowing clients to access a datacenter close to them with SwiftCloud, can reduce latency without requiring any complex reconciliation mechanism. The experience of using SwiftCloud has shown that adapting an existing application to use SwiftCloud requires low effort.

Keywords: CRDT, Replication, Cloud-Computing, Eventual Consistency, Optimistic Replication, Key-Value Store, Snapshot-Isolation, Evaluation.

Resumo

A Internet criou oportunidades para criar serviços de escala global. Estes sistemas requerem alta disponibilidade e tolerância a falhas sem prejudicar a latência das operações. A replicação é uma técnica amplamente utilizada para providenciar essas características. Diferentes mecanismos de replicação têm sido propostos ao longo dos anos, mas para fornecer essas propriedades em serviços à escala global é necessário balancear a consistência. Nos modelos de consistência futura, é necessário lidar com possíveis conflitos provenientes de actualizações concorrentes. Neste trabalho, propomos o uso de tipos de dados sem conflitos (CRDTs) para enfrentar esse problema.

Os sistemas de Cloud são utilizados para fornecer serviços à escala mundial, muitas vezes baseados em bases de dados Chave-Valor. Estes sistemas particionam e replicam os dados por diversas máquinas que podem estar espalhadas pela rede. Para lidar com conflitos, estes sistemas empregam soluções que permitem a perda de operações (e.g. ultima escrita ganha) ou delegam a resolução de conflitos para o nível aplicacional. Para além disso, estes sistemas fornecem pouco ou nenhum suporte para transacções.

Nesta dissertação, apresentamos o desenho e implementação do SwiftCloud, um sistema de armazenamento Chave-CDRT desenvolvido a partir de um sistema Chave-Valor existente que adiciona CRDTs ao seu modelo de dados. O sistema disponibiliza resolução automática de conflitos baseando-se nas propriedades dos CRDTs. Apresentamos também uma versão do sistema com suporte transaccional. O nosso suporte transaccional não obriga que transacções com conflitos de escrita abortem, devido às propriedades dos CRDTs, que permitem a convergência das replicas nestas situações. Neste trabalho, introduzimos algumas novas técnicas como os CRDTs com versões, a composição de CRDTs e métodos alternativos de serialização de objectos.

A avaliação do sistema, com micro-benchmarks e o TPC-W benchmark, mostram que o SwiftCloud produz um pequeno impacto na performance do sistema Chave-Valor original. A replicação de dados perto dos clientes com o SwiftCloud pode aumentar significativamente a performance sem utilizar mecanismos de reconciliação complexos. As experiências mostram ainda que adaptar aplicações para utilizarem o SwiftCloud requer pouco esforço.

Palavras-chave: CRDT, Replicação, Cloud-Computing, consistência futura,Replicação optimista, Sistemas de armazenamento chave-valor, Snapshot-Isolation, Avaliação

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Context

Internet allows providing services for users across the globe with great impact on people's life. Examples of these services are social networks, documents and media hosting services, on-line shopping, etc.. Besides accessing information provided by content providers, such as newspapers, users are sharing posts, documents, images and videos with each other. An important property for on-line services, is low response time. Recent studies show that an increase in latency has a direct influence in the usage of the systems [Nah04].

The latency and scalability of on-line systems is related to its underlying infrastructure. To deal with high volume of data and requests, a centralized solution may become very expensive and present high latency. An alternative approach is to build a systems composed by a network of cheaper machines geographically disperse, that distributes and replicates data to address performance, availability and reliability issues.

Several companies have this kind of global infrastructure for hosting their services, usually named cloud computing infrastructures [CRS+08, Klo10, LM09]. Some of them, e.g. Microsoft [Li09], Amazon [DHJ+07], etc., provide similar services to third party companies, allowing them to have access to global infrastructures, without the cost of deploying and maintaining one.

With Internet becoming ubiquitous and wireless communications having better quality, people want to be in constant contact with their on-line services. This usage of the internet led to increased demand of cloud infra-structures. These infra-structures are composed by several data center geographically disperse that host multiple services. The content may be stored anywhere, services may be integrated with each other to enhance their functionalities and users can access the cloud from various types of devices.

Cloud systems must provide high availability, fault tolerance and low latency. For providing

these properties, cloud systems strongly rely on replication [CDK05]. Replication consists in maintaining multiple copies of the same object in different machines. When a replica fails, another can continue to deliver the same content. It also helps reducing requests latency by allowing a replica, closer to the origin of the request, to handle the request. Replication is not trivial and it has some cost associated. Systems managing multiple replicas of the same objects must maintain them consistent over time, which may have an high cost.

Some services, such as banking usually require data to be consistent at all time. To provide always consistent view of data, replication systems must rely on some synchronization mechanism. Other systems, such as in social networks, the information is not very sensitive and it is tolerable that the content displayed to users diverges for a small period of time.

Allowing replicas to diverge increases the availability of the system, because replicas can continue to deliver responses to clients even with stale data. On the other hand updating replicas without synchronization, leads to divergence between replicas which requires the application to fix this divergence. For example, in a social networking application, all replicas of a user's wall must contain all posts. The drawback of this solution is that it adds complexity to the development of applications.

Cloud computing systems tend to favour availability over consistency [FGC+97a, DHJ+07, LM09]. This requires having to handle concurrent updates, which can be done by requiring applications to solve conflicts.

1.2 Motivation

Key-Value stores[DHJ+07, CRS+08] were developed to create large scale databases. They are used extensively in Cloud infrastructure, with a large number of proprietary (e.g. [DHJ+07, CRS+08]) and open-source solutions (e.g. [LM09, Klo10]). The data-model for these systems is simplified, when compared with traditional DBMS systems, providing a simple *put*, *get* API and a non-relational data-model. These systems are decentralized and replicated for providing high availability and no single point of failure. Since it is difficult to deliver strong consistency and availability, these systems can be configured to favour one of those properties.

The different Key-Value stores have different approaches for dealing with replication. Typically, they select a set of nodes to replicate content and use some heuristic to make them geographically disperse. For providing high availability, these systems allow concurrent updates to occur, leading to the execution of concurrent operations. The techniques used for conflict resolution also vary. Many systems employ a *last-writer-wins* strategy [LM09], where in case of concurrent updates, the last write will overwrite the previous ones, leading to lost updates. Other systems maintain the multiple concurrent updates [Klo10], providing them to the application for conflict resolution. This has the advantage of not losing updates, but it is much more complex for programmers.

Replication has been studied extensively in many contexts[SJZ+98, LLSG92, TTP+95, KS92, PMSL09]. In particular, optimistic replication systems [SS05, CDK05] are the most relevant approach used in cloud computing. A large number of conflict resolution approaches have been

proposed in the past, but they tend to be complex.

Recently, conflict-free replicated data types, CRDTs[PMSL09, SPBZ11b], have been proposed as a new approach to guarantee consistency without any synchronization mechanism. CRDTs are based on commutativity of operations, allowing replicas to apply operations in different orders with the guarantee that the replicas will converge. This approach offers the promise of simpler solutions which can be interesting for data management in cloud computing environments.

1.3 Proposed Solution

The objective of this work was to develop a distributed Key-CRDT store. This key-CRDT store should combine the high availability and fault tolerance properties of key-value stores, with automatic merge of concurrent updates. Building a Key-CRDT store from scratch would require a great effort concerning all the components of the system: the data storage, the topology of the network, the membership of nodes, etc. To avoid this effort, we decided to build the system as a middleware on top of an existing Key-Value store, transforming it into a Key-CRDT store while keeping all other characteristics of the system. This allowed us to study the overhead of adding conflict free replicated data types to an existing system without developing our own.

Our Key-CRDT store is called SwiftCloud, and it is built on top of Riak [Klo10]. However, unlike a key-value store, all values stored in the system are CRDTs. It has a simple interface similar to the the interface of underlying Key-Value Store.

Many CRDT designs were proposed in the literature [SPBZ11b, PMSL09] with two different methods to achieve consistency, state based and operation based. SwiftCloud used state based replication as it is more suitable to the data-model we are addressing. State based replication allows operations to be executed in our replicas of the object. Strong eventual consistency [SPBZ11c] between replicas is achieved by merging the states of the replicas.

We provide a version of the middleware that supports a form of transactions with a semantic that we call mergeable Snapshot Isolation. The transactional system is scalable, and supports intra-cluster transaction. We are not able to provide inter-cluster transactions consistency on top of Riak inter-cluster replication mechanism.

As in SI, our transactions access a snapshot of the data that can span multiple CRDTs. Unlike SI, transactions always commit, relying on the properties of CRDTs for merging concurrent updates. With the semantic of mergeable Snapshot Isolation we cannot force strict data invariants, as it is done by the solution of Aguilera et. al. [SPAL11]. For example, if two withdrawals from the same account are executed concurrently, we do not lose any of the withdrawals, however, we could let the balance go negative. Even with this limitation, our system can support a large number of applications, such as social networks.

For supporting access to a data version we have designed a new type of CRDTs, which we have called Versioned CRDTs. This new CRDT type allows us to select which elements are visible according to a requested version of the object, which is used to provide access to a consistent snapshot in SwiftCloud transactions.

In our work we present a qualitative and quantitative evaluation of SwiftCloud. Our objective

is to show that CRDT can simplify application development and impose low overhead over weaker solutions.

To evaluate the performance of our system we have developed micro and macro benchmarks. The micro-benchmarks evaluate specific parts of the system, namely the serialization overhead and the performance some CRDT implementations. The Macro-benchmark intends to evaluate the performance of CRDTs in an application and the effort of porting an application to use CRDTs. To this end, we used the TPC-W benchmark. We used an existing open-source implementation of this system in Cassandra and ported it to Riak. Then, we have modified this version to use SwiftCloud. Changing the benchmark to use SwiftClound required only minimal interface changes.

SwiftCloud has support for composing CRDTs, where a CRDT object is composed of other CRDTs objects. Our solution allows CRDTs to be stored in a different value in the underlying Key-Value store, using a special serialization mechanism. Our solution supports late-reading for improved performance. Results shows that CRDT composition is important for supporting large CRDT objects.

1.4 Contributions

The contributions of this work are:

- The design and implementation of a lightweight Geo-Replicated Key-CRDT store, the first Key-Value store with automatic conflict resolution based on CRDTs;
- The implementation of a CRDT library that can be used as a building block for applications;
- The design and implementation of a mechanism for CRDT composition.
- The design of data types that support versioning control over CRDTs;
- A Transactional system on top of an eventual consistency replication model;
- The development of a benchmarking tool based on TPC-W.

1.5 Organization

The document is organized as follows: In this section we made an introduction of the problem we are trying to solve; we motivated the need to add replication to current Cloud Systems and presented CRDTs; In chapter two we cover the state of the art. Presenting the classic replication models, describing some large scale databases and making a brief overview of transactional systems and Causality tracking literature; In chapter three we do a broad study on CRDTs already studied in the literature and propose some new designs; The following two chapters describe the solution. In chapter four we present the design of the proposed System and chapter six explains the detail of the implementation; Chapter 6 evaluates the solution and we present our conclusion in the last chapter, chapter seven.

2

Related Work

Replication is a fundamental technique to provide high data availability, fault-tolerance and shorter response time on globally available services. With services being accessed all over the world, it is fundamental that data is stored in different places close to end-users. Replication became ubiquitous to serve those purposes.

The web is being used to store all kind of information, with strong emphasis in high volume of low sensitivity and short messages, as those exchanged in social networks. Simpler storage systems, such as Key-Value stores [DHJ+07, LM09], are being used and studied to store this kind of information, since traditional database systems introduce large overhead in these environments. In this chapter, we will start by presenting replication models, architectural requirement and some selected systems. Later, we will describe Key-Value stores, which are an alternative to databases to store high volumes of data in a geographically distributed network.

2.1 Replication

Replication consists in having multiple instances of the same object in several machines, over a local or distributed network, and maintaining these objects' copies consistent over time. This is fundamental for performance of systems, since some objects can have a heavy access load and the machine hosting them might not have enough power to serve all the requests. Replication allows the workload to be distributed over the available set of replicas. The objects might be hosted in geographically disperse areas making the access latency shorter for computers nearer to those replicas. Furthermore availability and fault-tolerance increases because, if one replica is inaccessible, another replica can deliver the requests. Formally, if the probability of a replica failure is p, then the probability of the system to be unavailable is $1 - p^n$, assuming independent faults.

2.1.1 Models

Systems may have different requirements, for instance, some applications have an high ratio of updates over reads, while others are mostly read-only, one application might tolerate stale data, as some social networks, other might demand that data is always consistent over time, as in banking systems. We now address different dimensions of replication systems.

2.1.1.1 Pessimistic vs. Optimistic Replication

In pessimistic replication, the system simulates that there is a unique copy of the object despite all client requests to different replicas (*one copy serializability* [Raz93]). This method requires synchronization among replicas to guarantee that no stale data is retrieved.

In wide area networks this technique may not be adequate because the synchronization requirements would imply a great latency overhead. But, in local area networks, it can be acceptable because the latency to contact another replica is low.

Optimistic solutions [KS92, TTP+95, DHJ+07, LM09, CRS+08, SPAL11] allow replicas to diverge. This means that clients can read/write different values, for the same object, if they contact different replicas. When using such level of consistency the application must be aware of this fact.

These solutions are more adequate to large scale systems providing good fault-tolerance and responsiveness since every replica can reply to a read or write request without coordinating with others.

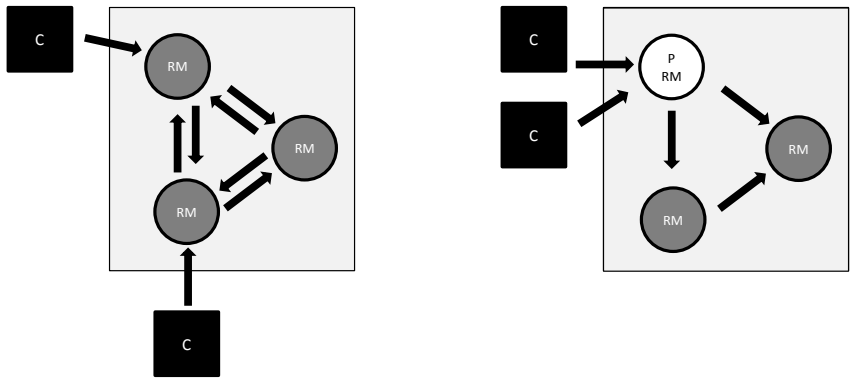
2.1.1.2 Active vs. Passive Replication

In active replication (or synchronous) [KA00, Ora99], all replicas are contacted inside a transaction to get their values updated. This replication mechanism can be implemented using a peer-to-peer approach, where all replicas have the same responsibilities and can be accessed by any client. Alternatively, a primary replica may have the responsibility of coordinating all other replicas.

Passive replication (or asynchronous) model assumes a replica will propagate the updates to the other replicas outside the context of a transaction. As usual, replicas must receive all the updates.

2.1.1.3 Operation Based vs. State Based replication

There are two fundamental methods to propagate updates among replicas. In State based replication [KS92], updates contain the full object state (or in optimized versions, a delta of the state). In operation based [TTP+95, KRSD01], the updates contain the operations that modify the object and must be executed in all replicas. The size of an object is typically larger than the size of an operation. Transmitting the whole state of an object can introduce a large overhead in message size. On the other hand, if the number of operations is high it can be better to transmit the whole state instead of all operations. Also, it can be simpler to update the state of an object than applying the operations on it, as discussed in 2.1.5.



(a) Peer-to-Peer replication (b) Client-Server replication

Figure 2.1: Replication architectures

2.1.2 Architecture

There are two main architectures to deploy a replication system, client-server and peer-to-peer, each with some advantages and drawbacks.

In client-server replication, there is a primary-replica that is responsible for receiving the requests and initially process the updates. Every client contacts the primary replica that works as a sequencer defining the order of the operations. The primary replica is a bottleneck of the system and a point of failure. To overcome this problem, when the primary replica is down, a secondary replicas takes the place of the primary.

In Peer-to-Peer systems every node share the same responsibilities and can be accessed to read or update values by any client. These solutions do not have a single point of failure. However, algorithms that require strong coordination of nodes are more complex than client/server solutions.

Client-server architectures are good to deploy pessimistic replication solutions, where clients contact the primary-replica for processing updates. The secondary replicas can be used to read values, when they are up-to date. On the other hand, peer-to-peer infra-structures are good to deploy optimistic solutions, because they rely on a completely distributed network and every replica can be used to read or execute updates. Figure 2.1 shows a diagram representing both architectures.

2.1.3 Correctness Criteria

Pessimistic replication systems usually impose strong correctness criteria such as linearizability or serializability. The concurrent execution of operations in a replicated shared object is said to be *linearizable* if operation appear to have executed atomically, in some sequential order that is consistent with the real time at which the operations occurred in the real execution [AW94].

According to the definition, for any set of client operations, there is a virtual canonical execution against a virtual single image of the shared object and each client sees a view of the shared object that is consistent with that single image.

The concurrent execution of operation in a replicated shared object is said to be *serializable* if operations appear to have executed atomically in some sequential order.. With this definition, we must maintain the order of operations within a transaction, but concurrent transactions can be executed with different interleaves, as long as their outcome conforms to a serial execution.

For optimistic replication, several weaker correctness properties have been defined [SE98, SS05]:

Eventual convergence property: copies of shared objects are identical at all sites if updates cease and all generated updates are propagated to all sites.

Precedence property: if one update O_a causally precedes another update O_b, then, at each site, the execution of O_a happens before the execution of O_b.

Intention-preservation: for any update O, the effect of executing O at all sites is the same as the intention of O when executed at the site that originated it, and the effect of executing O does not change the effect of non concurrent operations.

Assuming eventual consistency, a client can get stale data from replicas but the system will be very responsive since any replica can deliver the request. This criteria is often adopted in systems, susceptible to node failures, partition or long message delays.

2.1.4 Event Ordering

In this section we present some definitions and observations that we consider useful to better understand the techniques described in the following sections.

Event ordering is an important concept in distributed systems. Lamport introduced the happens-before relation [Lam78] to characterize the relation between events, on other words, to determine if an event can be responsible for another. We define the *happens-before* relation, between two events, a and b, as follows:

Definition 2.1 *The relation "$\rightarrow$" between two events is the smallest relation satisfying the following three conditions: 1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$; 2) if a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$; 3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.*

This definition induces that if $a \nrightarrow b$ and $b \nrightarrow a$, neither was generated before the other and they are concurrent "$||$". Happens-before defines a partial order on the events in a system.

It is often useful to define a total order on events in a system. A *total order* is a binary relation, $\leq_t$ with the following properties, where a and b are events in the set E:

Antisymmetry if $a \leq_t b$ and $b \leq_t a$ then $a = b$

Transitivity if $a \leq_t b$ and $b \leq_t c$ then $a \leq_t c$

Totality either $a \leq_t b$ or $b \leq_t a$

Reflexivity $a \leq_t a$

The totality property makes all events comparable and the Reflexivity property of total order states that all events have a relation to themselves.

A timestamp [Lam78] is composed by *(pId, counter)* where *pId* is the identifier of the process that generated the event and *counter* the number of operations generated at that process. Timestamps are totally ordered by the relation $(p_1, c_1) \leq (p_2, c_2)$, *iff* $c_1 < c_2 \vee (c_1 = c_2 \wedge p_1 < p_2)$. Timestamps do not allow to verify causal dependencies.

A Lamport clock is a variant of a timestamp, where when receiving a message with timestamp *(p,c)*, the *counter* of the local node is updated to *max(c, counter)* (if to the message reception is assigned an event id, this id is generated after updating the local clock). In this case, it is known that $a \rightarrow b \Rightarrow (P_a, C_a) \leq (P_b, C_b)$. However, given two timestamps, it is impossible to know if they are causally dependent.

Version Vectors are logical clocks that capture the causality relation of data versions in storage systems. A version vector is an efficient representation of a causal history [SM94]. A causal history, H_a, of an event a, contains the set of identifiers of events that precede event a. The partial order of causality can be precisely tracked by comparing these sets with set inclusion. An history H_a causally precedes H_b *iff* $H_a \subset H_b$. Two histories are concurrent if none includes the other: $H_a \| H_b$ *iff* $H_a \nsubseteq H_b \wedge H_b \nsubseteq H_a$. Two histories can be merged by computing the set union. When events are identified by Timestamps, causal histories can be efficiently represented by a version vector that maps the process identifier to the largest counter of events from that site.

The rules for causal histories can be adapted in a straightforward way for version vectors: $V_1 \leq V_2$ *iff* $\forall_p V_1[p] \leq V_2[p]$ and $V_1 \| V_2$ *iff* $V_1 \not\leq V_2 \wedge V_2 \not\leq V_1$. Two version vectors can be merged by taking the max counter of all entries. $\forall_p V_3[p] = max(V_1[p], V_2[p])$

2.1.5 Case studies

In this section we will present a few selected systems that use replication. We will focus on optimistic replication system because those are more related to our work.

2.1.5.1 Coda

Coda [KS92] is a distributed file system originated from AFS [CDK05] and developed with the purpose of improving availability and allowing disconnected operations by using optimistic replication. In this system, clients can act as replicas. The set of servers that replicate a volume is named the *volume storage group*(VSG).

When a client opens a file he accesses one replica from the *available volume storage group*(AVSG). The system tries to predict which files a user is interested, while he is on-line, and cache them locally to enable disconnected operation. A user is said to be executing disconnected operations when AVSG, for the file he is using, is empty. During this period, updates are kept on the client. When the replicas for the files are again available, conflicts may have occurred. The system uses version-vectors for every file, which stores the number of modifications at each replica, for conflict detection. The system automatically solves conflicts on directories. For solving conflicts

in files the system keeps the divergent versions and allows users to manually solve the conflict. Additionally, users may specify programs to automatically merge the conflicting versions.

Even if the heuristic for predicting the files a user will be interested is very good, the user probably will generate accesses to files that the system did not predicted he would. To improve the user experience, Coda enables the user to select which files he wants the local replica to store and maintain consistent.

2.1.5.2 Bayou

Bayou [TTP+95] is a replicated database that provides high data availability. The system adopts an eventual consistency model where replicas can receive updates independently. Bayou allows disconnected operations and enables domain specific conflict detection and resolution. The system propagates updates among replicas during anti-entropy session [PST+97]. Updates are totally ordered by a primary server. The system keeps two versions of the database: a committed version, which contains only the operations ordered by the primary server, and a tentative version, which reflects all known updates. While an update is not committed, the system may undo and reapply it to produce a consistent state. When an update is committed it is placed in a canonical order.

Bayou system differs from other systems by exposing replication to the application. The programmer shall specify for each operation the dependencies for execution and a conflict resolution procedure to be executed if a conflict is found. For example, in a calendar application, when an operation commits, other concurrent request might have already committed. Thus, the dependency check is used to verify if the original update is still possible - in a calendar operation this can check if the time slot is still available. If the dependency check succeeds, the original update is executed. Otherwise, a merge procedure is called that is responsible to try an alternative update. In the example of the calendar, this might be scheduling the meeting for a different time slot.

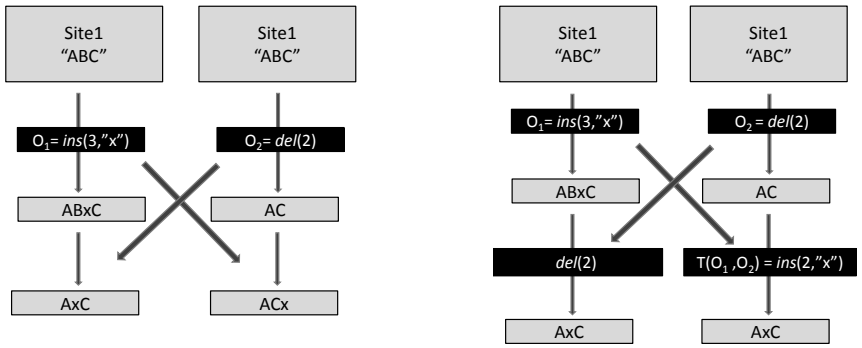
2.1.5.3 Operational Transformation

Operational transformation (OT) [SE98] is a model developed to achieve convergence in replicated systems through an algorithm that executes operations in different orders in different replicas. To achieve eventual convergence, the algorithm transforms the operation to be executed, so that all replicas converge to the same state.

The model is very generic and it can be implemented by different algorithms and transformation functions. OT assumes a set of nodes that maintain replicas of the data, each replica can receive updates without coordination with other replicas. The main idea of OT is to avoid delaying the execution of operations, and, in particular, to allow local operations to execute immediately. Thus, in different replicas, operations can be executed in different orders.

To provide eventual convergence, precedence and intention-preservation, OT frameworks are based on two main components:

1. An integration algorithm that is responsible for interchange and executing operations received locally and from other replicas.



(a) Divergence Problem

(b) Convergence using transformation function

Figure 2.2: Operation Transformation example

2. A set of transformation functions that are defined to ensure consistency among replicas while executing operations in different orders.

More formally, Transformation functions must guarantee that the execution of two operations out of order, on an object with state S, produce the same state S'. In particular, algorithms usually require transformation functions to obey the following properties, where Ox is an operation and $Ox \circ Oy$ is the sequence of operations Ox followed by Oy.

Property 1 *(TP1)* $Oa \circ T(Ob, Oa) \equiv Ob \circ T(Oa, Ob), \forall O_a, O_b, O_a \| O_b$

Property 2 *(TP2)* $T(Oc, Oa \circ T(Ob, Oa)) = T(Oc, Ob \circ T(Oa, Ob)), \forall O_a, O_b, O_c, O_a \| O_b \| O_c$

TP1 expresses an equivalence(denoted by $\equiv$) between two sequence of operations. This means that both sequences, when applied to an object with state S, would produce an equivalent state S'. TP2 defines an equality between the transformation of one operation against equivalent sequences of operations.

The development of a transformation function that respects these two properties is not trivial and furthermore it is very difficult to prove its correctness, mainly because there are a great number of possible cases of execution. Ressel et al. [RNRG96] showed that transformation functions satisfying TP1 and TP2 ensure convergence whatever reception order of concurrent operations.

To illustrate the need of applying transformations to make replicas converge, figure 2.2a illustrates an example of two replicas diverging by executing operations in their initial form. Operation O_1 refers a position in site 1, but in site 2 the operation is executed after O_2, thus the position no longer refers to the original position where the user of site 1 intended to insert the character. In figure 2.2b, the insert operation was transformed to adjust the insert position after the execution of O_2.

In the remainder of this section we will present some systems developed using the OT model. These systems address the problem of cooperative text editing. In this context, we assume the objects being replicated are documents represented as character sequences. The allowed operations

are *insert(position, char)* and *remove(position)*. The systems must guarantee eventual convergence, intention preservation and causal precedence, as defined in 2.1.3.

GROVE [EG89]: Groove includes four main elements: A state-vector to ensure precedence based on time stamping; an algorithm, called dOPT, to transform operation to assure convergence; a transformation function which is used in the algorithm; and a log of processed operations. The transformation function, $T()$, is defined to satisfy TP1 and it includes an auxiliary parameter that indicate priority between sites by their identifiers. The pre-condition to transform and apply an operation is that it must conform to causal order relation. The operation to be integrated is transformed against concurrent operations in the log of previous operations and then executed and stored in the log.

GROVE was pioneer in OT Model implementation. However, it was developed without respecting TP2 condition and it could not guarantee TP1 in every cases [SE98]. In figure 2.3 we show an example of TP1 violation, that problem could not be solved by refining the priority property and constitutes a violation of intention preservation. Thus, GROVE does not respect both convergence and intention preservation.

REDUCE [SJZ+98]: Reduce uses an *undo/do/redo* scheme and a transformation algorithm to maintain intention preservation and achieve consistency. It uses a History Buffer, similar to GROVE's operation log, for achieving causality preservation and make operations conform to total order. Operations are applied as soon as the causal order is verified, but the algorithm uses an *undo/do/redo* scheme to enforce total order with operations transformed to guarantee intention preservation. Two transformations are used. The first transformation assures that the effect of a transformed operation Oa' in a document that contains the impact of Ob is the same as the effect of Oa in a document that does not contain the effect of Ob. The second transformation assures that the transformation of Oa against Ob excludes the effect of Ob. The *undo/do/redo* scheme guarantees replica convergence, as all replicas eventually execute all operation in the same total order. The transformations are used to preserve users intentions. Transformation function in REDUCE respect TP1 and TP2.

Jupiter [NCDL95]: Jupiter, adopted in Google Wave, uses a central server to connect replicas. The algorithm for maintaining consistency is based on the dOPT algorithm modified to deal with a central server. Replicas have a copy of the documents being replicated but they are also maintained in the central server. Operations from clients are executed in their local replica and then transmitted to the server. The server transforms the operation, if it is required, and then broadcast the operation to other replicas, on reception the operations may be transformed again. This solution simplified the transformation function but relies on a central server which can be a bottleneck for scalability.

The development of algorithms that conform to the OT Model are very error prone [OUMI05a]. For this reason, researchers started to think on alternative techniques. In the following sections we will present WOOT and CRDTs, two alternative solutions for replicating data. WOOT is appropriated for cooperative text editing and CRDTs are more generic, addressing different data

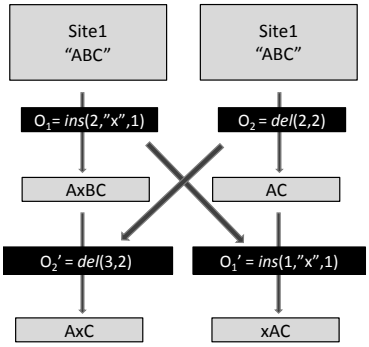


Figure 2.3: Counter-example violating TP1 for GROVE algorithm.

types.

2.1.5.4 WOOT

WOOT [OUMI05b] uses a different approach to achieve the same goal of previously shown solutions. This technique solves successfully the problem of collaborative text editing and also intends to solve some scalability issues observed in other systems, in which the message size grows with the number of replicas.

The communication model is similar to the used before. An operation is applied locally when it is created, broadcast to the other replicas and then integrated locally. The authors say that the algorithm can be used with linear structures, such as XML trees but only present the edition of a character sequence.

Consistency model The consistency model requires convergence, intention and causality preservation. However, causality is defined to captures the semantic meaning of dependency [LLSG92], i.e., two operations are only dependent if one could not be instantiated without the execution of the other.

Data-Model The atoms in WOOT are *w-characters*$< id, \alpha, v, id_{cp}, id_{cn} >$ where: id is the identifier for this *w-character*; α is the alphabetical value of the character; $v \in True, False$ indicates if the character is visible; id_{cp} is the identifier of the previous *w-character* of c, in the sequence; id_{cn} is the identifier of the next *w-character* of c, in the sequence;

The identified is a pair *(numSite, counter)* and the operations for inserting and removing characters refer the *w-characters id* instead of its position. The operations that modify the characters sequence are *ins(c)* , *del(c)*.

Algorithm WOOT approach relies on commutativity of pairs of operations to deliver convergence. *(ins,del)* and *(del,del)* are commutable, however *(ins,ins)* is not commutable because of the semantic of the *ins* operation, and so there must be some additional ordering criteria.

When some *del(c)* operation is executed, the *w-character* is not removed from the sequence, it becomes a tombstone and it is marked as invisible. When a user insert a *w-character* c at pos

x, the framework must translate it to the position in the sequence considering invisible characters, pos *y*, and the operation will insert it between c_p and c_n which are the *w-characters* at pos *y* and pos *y+1*.

The receiver buffers the insert operations until the pre-conditions for execution are verified, which are that id_{cp} and id_{cn} *w-characters*, referenced in the operation, are in the local sequence. When receiving delete operations, the precondition is that c must be present at the local sequence. The execution of a received insert operation requires the location of the characters id_{cp} and id_{cn}. If there are no other *w-characters* in-between, c can be executed without further conflicts. Otherwise, it is necessary to order concurrent inserts before executing them.

This solution is easy to understand and guarantees every property of the consistency model. The fact that there is always a tombstone for every delete impose a space overhead.

2.2 Key-Value stores

A key-value store is a storage system with a simple API consiting only in *get(key)* and *put(key,value)* operations. Key-Value stores are used in large-scale, highly available systems, such as Facebook[1] [LM09] or Amazon[2] [DHJ+07] online store. These systems have an important role on the Internet nowadays, as they are being adopted for many cloud services. A Key-Value Store can be composed from hundreds to thousands of machines connected to each other and be geographically disperse.

This section presents the main concepts associated with these systems. We will focus on different design decisions and their advantages and drawbacks. Later on, we will present some concrete implementations of Key-Value stores.

2.2.1 Organization

In key-value stores, each node may share responsibilities or play different roles. In Dynamo [DHJ+07], every node has the same responsibilities in a typically peer-to-peer organization. An advantage of this approach is that there is no central point of failure but, on the other hand, routing can be more expensive. Several peer-to-peer organizations have been adopted in these systems.

Chord [SMLN+03] is a peer-to-peer overlay network that became popular for its ring topology. Every node has a connection to its successor and for a small set of more distant nodes in the ring, to minimize routing time. To enable better load balancing, the node that replies to a user request is not necessarily the one he sent the request to. For minimizing the overhead of routing in peer-to-peer networks, one-hop DHT have been proposed [DLS+04, GLR03]. In these systems every node knows every other node in the system.

In some systems, nodes have different responsibilities. For example in PNUTS [CRS+08], the network is composed of Routers, Storage Units and message brokers, making a layer for routing requests and another for storing data. With this approach, the actual location of data is hidden from the clients and the system maintains directories, with the mapping of data to nodes.

[1] `http://www.facebook.com`
[2] `http://www.amazon.com`

2.2.2 Data-model

The typical data-model for Key-Value stores is very simple, and it is composed by one or more tables, each one with a set of opaque values indexed by a key. The data-model supports no constraints over data. The interface for Key-Value stores is very simple with support for *put* and *get* operations on a single table.

2.2.3 Data Storage and Partitioning

Key-Value stores are used to store data partitioned among different nodes. The general strategy for partitioning is horizontal partitioning, because it requires only to contact one node to retrieve an entire record. The partitioning algorithm should distribute the data equally among all replicas to prevent node overload.

As tables may have millions of records, it is important to store data in an efficient way. The typical approach is to use hash functions (or consistent hash functions [KLL+97]) to distribute the data evenly across nodes (e.g. Dynamo) or use ordered tables indexed in a directory, which favour range queries(e.g. PNUTS, BigTable [CDG+06]).

2.2.4 Data Consistency and Replication

One reason for choosing a Key-Value store instead of a traditional database system is the high volume of data it must handle and the demand for quick operations. It would be desirable to deliver ACID properties [GR92], unfortunately it is difficult to offer data consistency and high data availability simultaneously [FGC+97b]. To guarantee data consistency when updating values, a simple solution is to make data unavailable until object convergence is verified in every replica. But, to increase performance, the system should allow uninterruptedly data requests to continue. For this, the consistency requirements must be relaxed and replicas must continue to serve data requests even in the presence of concurrent updates or failures. Key-Value stores typically provide eventual consistency, allowing updates to always proceed with the need of dealing with data conflicts and stale values.

In an eventual consistency context, conflicts may arise when different replicas are concurrently updated by different users. To reconcile the data, there are many strategies. One simple solution is to keep only the latest update(e.g. Cassandra [LM09]). Another solution is to deliver every conflicting values when requested and apply an application level conflict resolution(e.g. Riak [Klo10], Dynamo). Other important aspect is when to do the conflict checking and data reconciliation: it can be done at write or at read time. In Dynamo, conflict resolution is made on the application level, at read time. PNUTS delivers timeline consistency, this technique ensures that there are no conflicting updates, as discussed in 2.2.7.2.

2.2.5 Dynamo

In this section we will present Dynamo [DHJ+07], a Key-Value store from Amazon. In the Amazon's business it is very important do deliver fast response and consistent view of users updates:

when shopping on-line, one wants to see every update to the shopping cart quickly and correctly. If an user update is lost the user experience is decreased and the trust on the service is compromised.

In this section, we will present the main design decisions of Dynamo and how it addresses the availability and consistency requirements.

2.2.5.1 Partitioning algorithm

Nodes in the network are connected in a ring infrastructure. To store data, items keys are hashed yielding their position in the ring. Every node is responsible for a range of values that fall between their position and their predecessor's position. Hashing of keys is done through consistent hashing [KLL+97]. This technique is better than traditional hashing because it will only require to redistribute a small set of keys when a new node is added or removed. However, the original consistent hashing technique provides less effective load distribution. To allow better distribution of data, nodes are assigned to multiple regions of the ring [SMLN+03] by allowing them to run multiple instances of virtual nodes.

To route requests in the ring, nodes have a list with the other nodes and their hash-ranges. This is an extension of Chord routing protocol to reduce the number of hops to reach a node to one.

2.2.5.2 Replication

Data distribution in the ring is not enough to increase overall system's availability. For instance, if some object is very popular and it is being heavily accessed, the node hosting that content might not be able to handle all requests. Replication is necessary to make the system responsive in those situations. Dynamo's approach to deal with replication is to elect a coordinator who is responsible for the object, and replicate it at N of its successors. The identification of nodes that store that object are stored in a preference list and the algorithm that computes that list excludes multiple virtual nodes hosted by the same physical node to forbid redundant replication in the same node.

2.2.5.3 Data Versioning

Dynamo associates a vector clocks to every object to capture causality relations and detect conflicting versions. If there are conflicts, the system stores the conflicting values to deliver them to the application, which in turn will solve the conflict (application level reconciliation).

Vector clocks present a way to deal with concurrency but this technique has scalability issues, as vector size grows with the number of replicas that update the same content. In Dynamo, it is unlikely that many different nodes update the same values, since most of the updates will be executed by the preferred nodes to do the operation. To avoid that vector clocks grow immeasurably, for every pair *(nodeId,counter)*, if the difference between the counters surpasses a threshold then the pair with the lower counter is discarded. This is unsafe, but the authors report that, in practice, it poses no problems.

2.2.5.4 Operation execution

If the client requests operations to the network through a load balancer, he may contact any node in the system, either for *get* or *put* operations. When a request is sent, if the receiver node is not one of the top N preferred nodes to handle the query it will forward the operation to one of those nodes. The node that is handling the query is called the coordinator. It is necessary to specify how many replicas are necessary to handle a read (R) or write (W) request (Read/Write *Quorum*). The minimum number of replicas that participate in a read(R) or write(W) can be parametrized. If those values are defined such that R + W $>$ N and W $>$ N/2 , then there cannot be concurrent read and write operations on the same value and only one write persists. This behaviour is described in [Gif79] and it assures one copy serializability inside the cluster. Under this configuration it is still possible to generate conflicts.

When doing a write, the coordinator generates a new vector clock and sends the operation to N-1 nodes in the preference list. If W-1 nodes reply then the operation is considered committed. If there is already one object stored with a version vector that is not older then the vector that we are writing, then both values are kept. On get operations, the protocol is similar but if there are different vector clocks in the answers among the R replicas, then the content must be reconciled and written back. The availability of the system can be parametrized through R and W values. When W=1 the system require the commitment of the operation on only one replica, and hence make it more available.

To maintain durability and consistency among replicas, the system implements an anti-entropy protocol to keep replicas synchronized. The system uses a Merkle tree [JLMS03] to detect inconsistencies between replicas. One advantage of using a Merkle tree is that it is not required to download the whole tree to check consistency, on the downside, whenever a node enter or leaves the system the tree must be recomputed.

2.2.6 Dynamo inspired Systems

Cassandra [LM09]: Cassandra is a popular Key-Value Store used in large scale systems such as Facebook or Twitter[3]. This Key-Value store merges the design of Dynamo and the data-model from BigTable.

Nodes are organized in a ring topology, and objects are partitioned among nodes using consistent hashing. To address the non-uniform distribution problem, the system moves slightly loaded nodes in the ring to alleviate the load from other hotspot regions, as described by Stoica et. al. [SMLN$^+$03].

Replication is similar to Dynamo but can be configured to be rack or data-center aware, i.e., guaranteeing that replicas are stored in a way that the failure of a rack/data-center will not make the system unavailable and that there exist enough replicas in a rack to obtain a *Quorum*.

Secondary replica updates can be synchronous or asynchronous depending on the requirements of the application for performance or consistency. Conflict resolution is executed using *last-writer-wins* rule, which implies that some updates are lost.

[3] `http://www.twitter.com`

Data is stored in tables and every row has a key. Any operation under a single row is atomic per replica no matter how many columns are being read or written into. Table's columns are organized in families, which can be of two types: Simple or Super Column families. The Super Columns act like column families within a column family; columns can be sorted either by name or by date. Figure 2.4 illustrates the column organization in Cassandra.

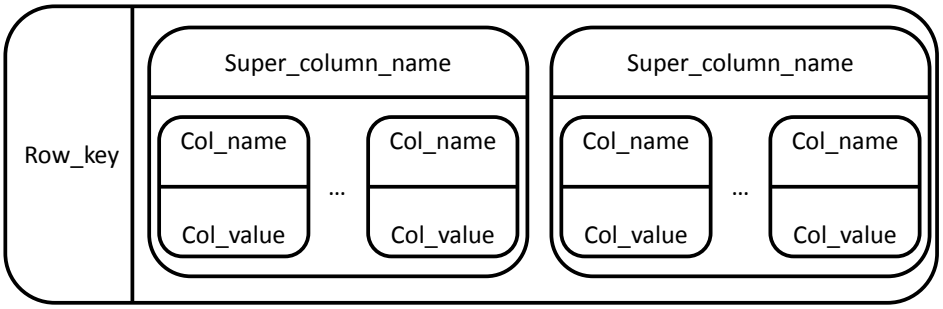


Figure 2.4: Cassandra table model

Riak Riak [Klo10] is a Key-Value store developed by Basho and it is a more faithful implementation of Dynamo than Cassandra. The data-model is very simple. Content is stored as binary data and it is identified by (bucket, key) pairs. A bucket is like a table and it can be created on the fly, while Cassandra requires the cluster to restart when the data-model changes. It also supports Map-Reduce operations [DG08] that is useful for distributed operations.

Conflict resolution can be done by last-writer-wins or by returning conflicting values to the application for domain-specific conflict resolution. Objects have a version vector associated for conflict resolution. The ring is organized in partitions and nodes can run multiple virtual nodes for better load distribution.

Riak Enterprise edition uses optimistic state based replication with a primary/secondary server interaction model. Each cluster, whose data is to be replicated, must specify a node to be the *listener*. A cluster that wants to replicate *listener's* data must define a *site* node that will be responsible for replicating the content to its cluster. It is possible to add multiple listeners and sites in each cluster, but this will only add redundancy, because replication is handled by one node for each cluster. Also, the system supports symmetric replication, i.e., both data-centers act as primary and secondary servers.

When the secondary server connects to the primary, they start a *full-synchronization* process. This process consists in computing hashes for all the values in the primary-server and sending them to the secondary server. The secondary server computes the hashes of its values and requests the values of the hashes that are missing or are different from the received. New written values on the primary cluster will be streamed to the secondary periodically.

2.2.7 PNUTS

PNUTS [CRS+08] is a Key-Value store developed by Yahoo![4] to store data for many of their web services. This system has major architectural differences when compared with Dynamo. Two of the major differences is that it supports ordered tables, which increases range queries performances, and it uses a *publish/subscribe* mechanism for many features such as replication.

2.2.7.1 Data Storage and partitioning

The system is divided into regions, each region contains a complete copy of every table. Replication between regions is carried out by a reliable *publish/subscribe* system. Tables are partitioned horizontally into tablets. Tablets are scattered across many servers and each tablet is stored only once in each region.

The data can be stored ordered or in a hash organization. When storing a table ordered, the primary-key space is divided into intervals and then each interval is associated to a tablet. When using an hash organized table, the hash space is partitioned and each partition is associated to a tablet. It uses the hash value of the key to determine the tablet for a record. Tablets can be associated to different storage units over time, to distribute the system load.

2.2.7.2 Data versioning

PNUTS uses a per-record timeline consistency model. It is a technique that provides conflict free access to data. There is a master for every record that is responsible for applying every update to that record. This imposes a sequential ordering of operations and for that reason there are no conflicting replicas. The write is complete when the master applies the operation locally, then it transmits the operation to other replicas. During this process, while replicas are not updated, they can continue to deliver their stale values. The master, for each record, is adaptively changed to be the node that receives the most number of write requests.

By using this model users can consult information with different levels of consistency:

Read-any: returns the value available from any replica, this value can possibly be stale. This can be used for favouring performance over consistency.

Read-Critical(required_version): Reads a record with a version equal or newer than required_version. This can be used when the user wants to read a value that contains his changes, for example to implement monotonic reads session guarantees [TDP+94, TTP+95].

Read-latest: delivers the newest version of the record.

Write: On the presence of multiple concurrent writes, the last-writer will win.

Test-and-set-write(required_version): This operation only writes the value if, the current version is equal to required_version. This allows to avoid a concurrent update from being overwritten.

[4] `http://www.yahoo.com`

This semantic is slightly different from the traditional serializable, repeatable read, read committed and snapshot isolation. In particular, this approach provides isolation over one record only.

2.2.7.3 Operation execution

To access a record the client must reach the tablet that contains that record. To find it, the client contacts a router that will determine what tablet stores the record and which storage unit stores the tablet. The router contains the ranges of the tablets and a cache of the mapping to the storage units. The mapping between storage units and tablets is managed by the tablet controller. This controller is responsible for managing the load balance of the system and is periodically contacted by routers to check for changes in the map.

PNUTS uses asynchronous replication to ensure low-latency updates by relying on the Yahoo! Message Broker (YMB). This is a *publish/subscribe* system and it guarantees that no message is lost even in presence of network failures.

There are many YMB servers to serve PNUTS, each with some local subscribers. When a message is sent to one server, the message is also delivered to the other YMB servers. Messages delivered to one YMB are replayed in order, but between YMB servers any order can occur.

To implement replication using this system, the system defines a master for each record. When an update for a record occurs, it is forwarded to the master of the replica and then sent to the YMB for propagation to other replicas. When the update is delivered to YMB, the record update is committed.

2.3 Transactions

Transactions are a well known technique commonly used in databases [GR92]. Transactions group a set of operations that must executed in the database and either the effect of all operations is visible or none. Transactions are commonly characterized by the ACID properties (Atomicity, Consistency, Isolation and Durability):

Atomicity Atomicity is the described effect of viewing all the effect of all operations in the transaction or none.

Consistency Consistency means that the database must always move from one consistent state to another. In the context of database systems, this means that the constraints of the database will be valid after the execution of a transaction.

Isolation Isolation is the property that no transaction can interfere in the work of another, either by reading an uncommitted value or overlapping write operation.

Durability Durability property is defined to assure that a committed value will always belong to the database even in the presence of failures.

It is complex to provide ACID properties in transactional systems, allowing concurrent accesses. In the case of DBMSs, common solutions rely on table row locking to assure that transactions

do not interfere with each other. This can degrade performance in the presence of transactions accessing the same data item.

To get better performance systems often allow to reduce the isolation guarantees. The drawback of reducing isolation level is that applications will experience phenomena that depart from strict serializability.

The ANSI SQL-92 [X3.92] defines the isolation levels in terms of observable *phenomena*:

Dirty Reads A dirty read means that a transaction will read a value that is not committed.

Fuzzy Reads A fuzzy read happens when a transaction reads a value more than once, and sees that the value has been modified by another transaction.

Phantoms A phantom read is when a transaction reads a set of values answering a query and a posterior execution of that query returns different values. This happens if another transaction that creates data items that satisfy that query commits between the two queries.

The defined isolation levels are: Serializable, Repeatable Read, Read Committed, Read Uncommitted. The isolation levels are characterized according to the observable *phenomena*. Table 2.3 shows the phenomena allowed for each isolation level.

Isolation Level	Dirty Read	Fuzzy Read	Phantom
Read Uncommited	allow	allow	allow
Read Committed	not allowed	allow	allow
Repeatable Read	not allowed	not allowed	allow
Serializable	not allowed	not allowed	not allowed

The loosest (that disallows less phenomena) isolation level is the Read Uncommited. There can be defined intermediate isolation levels until Serialializable isolation level, which is the strictest. In Read Uncommited, values that were not committed and can possibly be rolled back, can be read in a transaction. In Serializable, the execution of a concurrent set of transactions has to be equivalent to the sequential execution of them. Berenson et al. [BBG+95] discuss the problems with the description of the isolation levels and propose some fixes.

A large number of systems have implemented distributed transactional systems using primary-servers (e.g. Ganymed [PA04]) or decentralized solutions (e.g. Postgres-R [WK05], Tashkent [EZP05, EDP06]). Supporting transactions in a cloud system that provides geographic replication has been proposed recently in some systems (e.g. PNUTS [CRS+08], Walter [SPAL11], [LFKA11]).

Cloud systems with transactions present some limitations. For instance, in the PNUTs system, write transactions include a single operations. In Walter, there are two execution models for transactions. The slow commit protocol allows transactions to span multiple objects, replicated in different nodes, but it uses a two-phase commit protocol. The fast commit protocol allows transactions to commit locally, if either the preferred replica of all objects is the local node or if the object is a C-Set (an object where all operations commute).

Our system provides transactions in a cloud system with a weaker consistency level. We rely on the commutativity of operations [SPBZ11a] to provide a transactional system in which transactions never abort due to write-write conflicts. This allows a greater level of concurrency, allowing transactions to execute without the need of coordination among replicas.

3

CRDTs

The Conflict-Free Replicated Data Types (CRDTs) are a family of data types that can be updated without synchronization and do not require consensus on replica reconciliation. Their asynchronous nature make them very suitable to provide replication in eventual consistency environments, allowing to provide scalability and fault-tolerance in large scale distributed systems.

The CAP theorem [GL02] states that is not possible to archive, simultaneously consistency, availability and partition-tolerance, in a distributed system. However, it is possible to pick two of those properties without huge prejudice to the latency. The eventual consistency model sacrifices consistency to provide both availability and partitioning-tolerance. However, eventual consistency poses an important drawback: executing operations without coordination between replicas, can originate conflicts that must be resolved. CRDTs tackle that problem in a systematic, theoretical proven approach, based on simple mathematical rules, by providing automatic reconciliation. Furthermore, they satisfy Strong Eventual Consistency Model [SPBZ11c] and can be used as building blocks of other data types that are suitable for programmer's applications.

In this section, we overview previous work on CRDTs that we consider relevant to this work. We specify the SEC model, how CRDTs satisfy it and two different interaction models, CmRDTs an CvRDTs. We present a portfolio of known CRDTs and extend that collection by introducing Versioned CRDTs.

3.1 System Model

We consider a single object replicated at a fixed set $\{p_0, p_1, ..., p_n\}$ of processes that can fail and recover. A process that has not crashed is considered working correctly. The $source$ of an operation is the client that invokes an operation on an arbitrary replica.

In our system, CRDTs are mutable objects that store one or a more atoms (*payload*), contain

an initial state and a specific API to access and modify them. CRDTs can be replicated in different processes. We consider temporary network partitioning and no Byzantine behaviour. Operations can be executed in any replica and synchronization between replicas occurs periodically.

The Strong Eventual Consistency Model [SPBZ11c] guarantees consistency without operations roll-back. Conflict free operations ensures safety and liveliness despite any number of failures. These properties pose a solution for the CAP theorem [SPBZ11c]. On the other hand, eventual consistency only guarantees that all updates are executed in the replicas, requiring consensus to ensure that conflict resolution produces the same final state in different replicas.

To provide SEC, the outcome of an update operation must be deterministic and independent of the context. We can define strong eventual consistency as follows:

Definition 3.1 *(Strong eventual consistency) A system providing replicated objects is Strongly Eventually Consistent iff: (**Eventual Delivery:**) An update delivered at some correct replica is eventually delivered to all correct replicas:* $\forall_{i,j} : u \in c_i \Rightarrow \diamondsuit_u \in c_j$. *(**Termination:**) All method executions terminate. **Strong Convergence:** correct replicas that have delivered the same updates have equivalent state:* $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$

(c_i represents the current state of the replicated object at process i and goes through a sequence of states $c_i^0, ..., c_i^k$)

3.2 State based CRDTs (CvRDT)

State based objects are tuples (S, s^0, q, u, m). A replica at process p_a has state $s^i \in S$, called the payload of the object, , s^0 is the initial state, q, u are sets of query and update operations, respectively, and m is a merge operation. Query operations read the current state of the object, update operations modify the state of the object and merge reconcile the state of two objects.

A process p_a that executes an update on a replica changes the object's state from s^i to s^{i+1}. In Background, a replication agent sends the payload — the current state of the object — to process p_b, that also replicates the object, and, when it is received, the process merges the received and local states.

Definition 3.2 *(Causal-History (state-based)) We define the object's causal history* $C = \{c_1, ..., c_n\}$ *as follows: Initially,* $c_i^0 = \varnothing$*, for all* i*. If the* k^{th} *method execution at* i *is: (i) a query* q*: the causal history dos not change, i.e.,* $c_i^k = c_i^{k-1}$*; (ii) an update (noted* $u_i^k(a)$*: it is added to the causal history, i.e.,* $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$*;(iii) a merge* $m_i^k(s_{i'}^{k'})$*, then the local and remote histories are unioned together:*$c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$.

A join semi-lattice is a partial order $\leq$ equipped with a *least upper bound* (LUB) $\sqcup$ for all pairs: $m = x \sqcup y$ is a LUB of $\{x, y\}$ under $\leq$ iff $\forall m', x \leq m' \wedge y \leq m' \Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$. It follows that $\sqcup$ is: commutative: $x \sqcup y = y \sqcup x$; idempotent: $x \sqcup x = x$; and associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

Definition 3.3 *(Monotonic semi-lattice object) A state-based object, equipped with partial order* $\leq$*, noted* $(S, \leq, s^0, q, u, m)$*, that has the following properties is called a monotonic semi-lattice:*

(i) Set S of payload values forms a semi-lattice ordered by $\leq$. (ii) Merging state s with remote state s' computes the LUB of the two states, i.e., $s \bullet m(s') = s \sqcup s'$. (iii) State is monotonically non-decreasing across updates, i.e., $s \leq s \bullet u$

Theorem 3.1 *(Convergent Replicated Data Type(CvRDT)) Assuming that every update is propagated to all replicas and they cease, any state-based object that satisfies the monotonic semi-lattice property is SEC.*

Proof of this theorem as been presented by Shapiro et al. [SPBZ11b].

3.3 Operation based CRDTs (CmRDT)

An op-based object is a tuple (S, s^0, q, t, u, P), where S, s^0 and q preserve the same meaning as before and the update operation is split into a pair (t, u). t is a *prepare-update* method and u is an *effect-update* method. The prepare-update executes at the replica where the operation is invoked, before the effect-update method u, i.e., $f_i^{k-1} = t \Rightarrow f_i^k = u$. The effect-update t executes also at all remote replicas (*downstream*). Operations to downstream are delivered according to the delivery precondition P, specified for that type of objects, e.g. causal delivery (operations delivered in causal order). We say that an operation is *enabled* when it can be executed in the receiving replica. q and t do not produce side effects, i.e., $s \bullet q \equiv s \bullet t \equiv s$.

Definition 3.4 *(Causal History(op-based)) An object's causal history $C = \{c_1, ..., c_n\}$ is defined as follows: Initially, $c_i^0 = \varnothing$, for all replica i. If the k^{th} method execution is: (i) a query q or a prepare-update t, the causal history does not change; (ii) an effect-update $u_i^k(a)$, then $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$.*

An update is said delivered at a replica when the effect-update is included in the replicas' causal history. Update (t, u) happened-before (t', u') iff the former is delivered before the latter executes.

Operations can naturally be instantiated concurrently. Instead of arbitrating an order for concurrent operations, some delivery orders allow those operations to be delivered in different orders at different processes. Operations that are delivered in different orders must be commutative, otherwise the state of the replicas that applied those operations may diverge.

Definition 3.5 *(Commutativity) Update (t, u) and (t', u') commute, iff for any reachable replica state s, where both u and u' are enabled, u(respectively u') remains enabled in state $s \bullet u'$ (respectively $s \bullet u$), and $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$*

Theorem 3.2 *(Commutative Replicated Data Type (CmRDT)) Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates and whose delivery precondition is satisfied by causal delivery is SEC.*

Proof of this theorem as been presented by Shapiro et al. [SPBZ11b].

3.4 CRDT Examples

In this section we present a collection of known CRDTs. We describe them formally and the presented specifications are the simplest possible. In the specifications we are not concerned with operations complexity. In this work we only work on CvRDTs as these were more suitable to our system.

3.4.1 Counters

A counter is an integer with two operations: increment and decrement. The value reflects the difference between the number of increments and decrements[1] on it.

A CmRDT counter is very simple to implement. If we unequivocally identify every operation and the delivery pre-condition guarantees that all operations are delivered and executed once, replicas will converge whatever order operations are applied. On the other hand, a CvRDT monotonic counter is not trivial to implement: Consider a counter that maintains the number of increments. To merge the value of two replicas with value 1, we would have to sum the values, or select the maximum value of the merging replicas, but neither would return the correct value in every case, as shown next: If the replicas had concurrent updates, then the value should be 2. However, if the two replicas are synchronized, the final value should be 1. This make it impossible to use either *max* or *sum* as the merge procedure.

A good solution is inspired by vector clocks, we call it *increment only counter*. We store the number of increments for each replica indexed by position in a vector. The query operation retrieves the sum of every vector position and the merge procedure selects the maximum value for each index in the vector. To allow decrement operations we can combine two *increment only counters*, one for counting the increments and other to count the decrements. In this case, the value is the difference of the two counters. The pseudo-code for the implementation of this CvRDT is shown in algorithm 1.

If we want the counter to be non-negative then this would not be a possible solution because two concurrent decrements would make a counter with value 1 to become -1. We would have to restrict the number of decrements of one replica to be smaller or equal to the number of its increments but this could be a too restrictive solution in some environments.

3.4.2 Sets

Sets are abstract data types that are used in many applications. For example, sets can be used to store a shopping cart, to store your friends in a social network or in many other situations. The minimal interface of a set is composed by the following operations: *add(e)*, *remove(e)*, *contains(e)* and *elements*. *add* and *remove* have the typical meaning, *contains* verify if an element *e* belongs to the set and *elements* retrieve the elements of the set.

In the literature, there are some implementations of replicated sets that try to enable convergence without synchronization [WB84, SPAL11]. Some of these implementations have erroneous

[1] This can be easily extended to support add/subtract operations

Algorithm 1 CvRDT Integer Counter

```
1: payload integer[n] P, integer[n] N
2:     initial [0, ..., 0], [0, ..., 0]
3: update increment ()
4:     let g = myID()                                   ▷ g: source replica
5:     let P[g] := P[g] + 1
6: update decrement ()
7:     let g = myID()
8:     let N[g] := N[g] + 1
9: query value () : integer v
10:    let v = Σ_i P[i] − Σ_i N[i]
11: compare (CvRDT X, CvRDT Y) : boolean b          ▷ CvRDTs must be Integer Counter
12:    let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i]) ∧ (∀i ∈ [0, n − 1] : X.N[i] ≤ Y.N[i])
13: merge (CvRDT X, CvRDT Y) : CvRDT Z
14:    let ∀i ∈ [0, ..., n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
15:    let ∀i ∈ [0, ..., n − 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

behaviour or are very restrictive.

In this section, we introduce *concurrent set specifications*, a model to describe the behaviour of sets by specifying the resulting state when a set of operations are applied concurrently. We describe some Set CRDTs specifications.

3.4.2.1 Concurrent Set Specification

We specify the sequential semantics of update operations using Hoare logic [Hoa69]:

$$\begin{array}{ccc}
\{true\} & add(e) & \{e \in S\} \\
\{true\} & remove(e) & \{e \notin S\} \\
\{e \in S\} & add(f) & \{e \in S\} \\
\{e \in S\} & remove(f) & \{e \in S\} \\
\{e \notin S\} & add(f) & \{e \notin S\} \\
\{e \notin S\} & remove(f) & \{e \notin S\}
\end{array}$$

Whatever the initial state, the set contains (respectively does not contain) *e* immediately after the return of *add(e)* (respectively *remove(e)*). An operation on *e* does not affect the state regarding other elements, if they are different elements.

Concurrent specification of an object is an axiom that describes the outcome of a set of concurrent updates: $\{P\}\,[u_0 \| ... \| u_{n-1}]\,\{Q\}$. We describe concurrency semantics with permutations of operations. Consider some pair of operation op, op': if both sequences $op;op'$ and $op';op$ have the same effect, then the concurrent execution $op \| op'$, should also have that same effect. This can be generalized to any number of concurrent operations and all their permutations. We call this *principle of permutation equivalence*. Thus we require a set to satisfy the following (where $e \neq f$):

$$
\begin{array}{lcr}
\{true\} & add(e) \| add(f) & \{e, f \in S\} \\
\{true\} & add(e) \| remove(f) & \{e \in S \wedge f \notin S\} \\
\{true\} & remove(e) \| remove(f) & \{e, f \notin S\} \\
\{true\} & add(e) \| add(e) & \{e \in S\} \\
\{true\} & remove(e) \| remove(e) & \{e \notin S\}
\end{array}
$$

The pair $add(e) \| remove(e)$ is ambiguous and can produce different results. To satisfy SEC, we have to specify an outcome that can be produced in every replica, whatever the execution order of both operations. If operations are totally ordered, for instance by $<_{CLK}$, which orders elements by a CausalityClock associated to them, the produced outcome is to keep the greater element according to $<_{CLK}$. Another solution is to specify precedence of operations; *add-wins* ($\{e \in S\}$) or *remove-wins* ($\{e \notin S\}$). All the proposed solutions are correct since they conform to sequential semantics and the sufficient conditions to satisfy SEC [SPBZ11c].

The procedure adopted may depend on the application. For example, if we are implementing a shopping cart, the vendor may prefer an add-win strategy over remove-wins, in order to guarantee that items are kept in the cart on conflict situations.

3.4.3 Set specifications

3.4.3.1 C-Sets

C-Set [SPAL11], or counter set, are a type of CRDTs that implements the Set abstract data type. C-sets were introduced in [SPAL11] and, although the authors did not refer it, C-Set is a CvRDT. C-Sets store elements in a normal set and count the number of *add* and *remove* operations for each element. An element is visible (respectively invisible) if the number of adds (resp. removes) is greater than the number of the remove (resp. add) operation. The merge procedure will sum the number of adds and removes, for each element, in both replicas.

C-Set is a CRDT but it cannot guarantee *permutation equivalence* defined for set operations, as shown in figure 3.1. If two C-Set replicas, with state S_A and S_B, *add(e)* the same element *e* and then $merge(S_A, S_B)$, they resulting C-SET will have a counter 2 for the added element. If they *remove(e)*, they must *remove(e)* twice, to make it effectively invisible. If they *add(e)* and then they $merge(S_A, S_B)$ again, the element will not be visible, because the sum of add and removes is four for both elements. However, as in any permutation, the *add* would be the last operation, the element should be visible.

3.4.3.2 OR-Set

The idea of OR-Sets, or Observed-Remove Set, is to control the visibility of an element according to the precedence of *add* or *remove* operations, when concurrent operations are issued for the same element. To enable it, *add* and *remove* operations associate identifiers that have the total order relation (e.g. Timestamps) to the elements.

There are two distinct implementations of OR-Sets: one with *add-wins* and the other with *remove-wins* precedence.

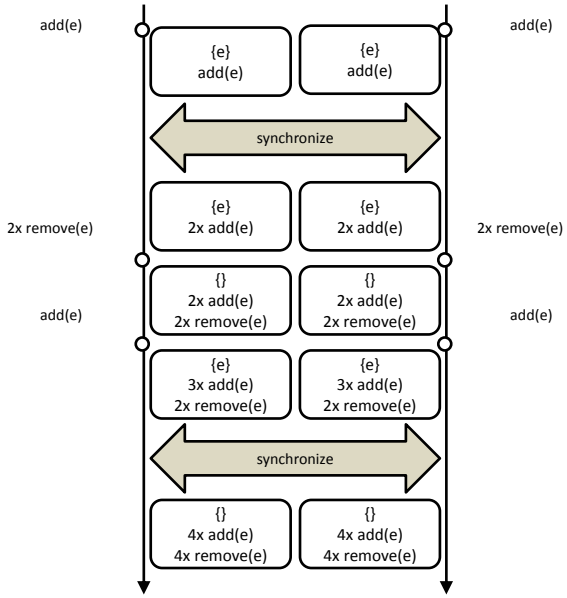


Figure 3.1: C-Set Anomaly

Add-Wins OR-Set The CvRDT Add-Wins OR-Set gives precedence to the *add* operation, i.e., concurrent operations over the same element *e* will make the element belong to the set if at least one of the concurrent operations is an *add(e)*. The concurrent specification $\{P\}\ [u_0 \| ... \| u_{n-1}]\ \{Q\}$ that describes this behaviour is defined as follows:

- $\forall i, u_i = remove(e) \Rightarrow Q = (e \notin S)$
- $\exists i; u_i = add(e) \Rightarrow Q = (e \in S)$

To implement a CvRDT Add-Wins OR-Set we need a Set *S* of elements and a set *T* of tombstones. The Set *S* stores (element, identifier) pairs and *T* is just a set of identifiers. When an element is added to the set, it is stored with a new identifier. If an element *e* is removed, the identifiers associated to that element are moved to the tombstones and the element removed from the set. During merge, all elements of *S* that have their identifier in the Tombstones set are removed. A specification for CvRDT Add-Wins OR-Set is given in algorithm 2:

Remove-Wins OR-Set The CvRDT Remove-Wins OR-Set is similar to the Add-Wins. The difference is that we promote the *remove* instead of the *add* operations. The concurrent specification $\{P\}\ [u_0 \| ... \| u_{n-1}]\ \{Q\}$ for *Remove-wins* OR-Set is defined as follows:

- $\forall i, u_i = add(e) \Rightarrow Q = (e \in S)$
- $\exists i; u_i = remove(e) \Rightarrow Q = (e \notin S)$

The implementation is also similar to the previous. The payload has the set *S* of pairs, as the previous specification, and the set *T* of tombstones also stores pairs. When an element is added to the set, a new identifier is created if none exists before, or if the element has been previously

Algorithm 2 OR-Set Add-Wins specification

```
1: payload set E, set T                    ▷ E: elements; T: tombstones; element (e, identifier)
2: initial ∅, ∅
3: update add (element e)
4:     let n = unique()                    ▷ generates a event that identifies the operation and process that generated it
5:     E := E ∪ {(e, n)}
6: update remove (element e)
7:     let R = {(e, n)|∃n : (e, n) ∈ E}
8:     E := E \ R
9:     T := T ∪ R
10: query contains (element e) : boolean b
11:    let b = (∃n : (e, n) ∈ E)
12: query elements () : set S
13:    let S = {e|∃n : (e, n) ∈ E}
14: compare (CvRDT X, CvRDT Y) : boolean b
15:    let b = ((X.E ∪ X.T) ⊆ (Y.E ∪ Y.T)) ∧ (X.T ⊆ Y.T)
16: merge (CvRDT A, CvRDT B) : CvRDT Z     ▷ CvRDTs must be OR-Set Add-Wins
17:    Z.E := (A.E \ B.T) ∪ (B.E \ A.T)
18:    Z.T := A.T ∪ B.T
```

removed, the identifiers are added to the set *S* again. A specification for the Remove-Wins OR-Set is given in algorithm 3.

OR-Set without tombstones The previous OR-Set specifications have an unbounded tombstone growing set. This poses a problem, because the space complexity is linear in the number of *remove* operations and not the number of elements in the set, as one would expect. This data structure would benefit with garbage collection of tombstones that have been propagated to all replicas.

CRDT OR-Set without tombstones can be implemented using a Version Vector that stores the identifiers of executed operations. In the former specifications, we used identifiers to keep track of executed operations. Since we had no way to verify that two replicas have seen the same operations, we had to keep the identifiers. This implementation relies on Version Vectors to encode the executed operations per replica. When a *add* operation is instantiated, a new identifier for that operation is created and added to the version vector.

On merge, if both replicas' version vectors have the same events, but they do not belong to the element set of one of them, then the elements have been removed in that replica. If all the timestamps associated to an element have been removed, then the element no longer belongs to the set. The specification for this set is shown in algorithm 4.

Although we can see the tombstones set as a Causal History Set, using version vectors has great performance benefit comparing to the solutions with tombstones because there are implementations of version vectors that have constant time complexity to verify if an event belongs to the vector.

Algorithm 3 OR-Set Remove-Wins specification

1: payload set E, set T ▷ E: elements; T: tombstones; element (e, unique-tag)
2: initial $\varnothing, \varnothing$
3: update add (element e)
4: let $R' = \{(e, n) | \exists n : (e, n) \in T\}$
5: **if** $R' = \varnothing$ **then** $R = \{(e, unique())\}$
6: **else** $R = R'$
7: $E := E \cup R$
8: $T := T \setminus R$
9: update remove (element e)
10: let $n = unique()$ ▷ generates a event that identifies the operation and process that generated it
11: $E := E \setminus \{(e, n) \in E\}$
12: $T := T \cup \{(e, n)\} \setminus E$
13: query contains (element e) : boolean b
14: let $b = (\exists (e, n) \in E \wedge \nexists n' : (e, n') \in T\}$
15: query elements () : set S
16: let $S = \{e | \exists n : (e, n) \in E \wedge \nexists n' : (e, n') \in T\}$
17: compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDTs must be OR-Set Remove-Wins
18: let $b = ((X.E \cup X.T) \subseteq (Y.E \cup Y.T)) \wedge (X.E \subseteq Y.E)$
19: merge (CvRDT A, CvRDT B) : CvRDT Z
20: $Z.E := (A.E \setminus B.T) \cup (B.E \setminus A.T)$
21: $Z.T := A.T \cup B.T$

3.4.3.3 Implementation

The specifications presented in this section are were written with simplicity in mind. A direct implementation of these specifications would produce data structures with low performance. For instance, the concrete implementation of the OR-Set uses a map instead of a set to store its data. The keys of the map represent the values of the set, and the values are the set of timestamps associated to that value.

3.4.3.4 Map CRDT

A Map CRDT can be implemented using the Set specification. The Map CRDT maps keys to sets, using any of the specifications presented in this section. The merge procedure creates a new map with all keys from the merging maps. If a key belongs to both maps, then the sets are merged.

3.4.4 Registers

We can see registers as memory cells that store elements. The operations over a register are *read* and *assign*. The same principle of concurrent specifications used in the previous section can be applied here to define the outcome of concurrent operations. The only ambiguous case is what happens when two *assign* operations are executed concurrently. Again, we define a deterministic outcome to make registers satisfy SEC, which origins the following Register designs.

Algorithm 4 OR-Set without Tombstones specification

payload Set E, VersionVector V ▷ E: elements; element (e, timestamp) V Version vector of timestamps
initial $\varnothing$
update add (element e)
    let $t = unique()$
    $V := V \cup t$
    $E := E \cup (e, t)$
update remove (element e)
    $E := E \setminus \forall_i (e, t_i)$ ▷ remove all pairs with element e
query contains (element e) : boolean b
    let $b = (\exists (e, t) \in E)$
query elements () : set S
    let $S = \{e | \exists_t : (e, t) \in E\}$
compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDTs must be OR-Set Without tombstones
    let $R = \{t | t \in X.V \wedge \nexists e : (e, t) \in X.E\}$
    let $R' = \{t | t \in Y.V \wedge \nexists e : (e, t) \in Y.E\}$
    let $b = X.V \leq Y.V \wedge R \subseteq R'$ ▷ Compare the remove sets
merge (CvRDT A, CvRDT B) : CvRDT Z ▷ CvRDTs must be OR-Set without Tombstones
    let $E' := \{(e, t) | (e, t) \in B.E \wedge \exists_t \notin A.V\}$ ▷ V the version vector associated to the Set
    let $E'' := \{A.E \setminus \{(e, t) | (e, t) \in A.E \wedge \exists_t \in B.V \wedge (e, t) \notin B.E\}$
    $Z.E := E' \cup E''$
    $Z.V := A.V \cup B.V$

3.4.4.1 LWW-Register

CRDT LWW-Register, or *Last Writer Wins Register*, is a simple implementation of the Register abstract data type. LWW-Register stores the last written element according to the total order of operations. Each operation has a unique identifier associated, consequently, when there are concurrent *assign* operations, they are ordered according the operations' identifiers, hence it is always possible to determine the last assigned element. The payload of this CRDT is an element and the respective identifier. The *merge* of two LWW-Register clones the register with the greatest identifier.

3.4.4.2 MV-Register

In MV-Register, or *Multi Value Register*, instead of establishing a total order of elements and keep the greatest one, the idea is to rely on causal order and keep conflicting values, instead of losing some of them. Keep conflicting values may be a benefit in some applications, however the programmer will have to deal with conflicts.

To implement this, the payload is a set S of pairs *(value,Version Vector)*, we use a set to allow storing concurrent values and a version vector to track causality. When an *assign* operation executes it overwrites the existing payload. If concurrent *assign* operations execute, all concurrent values are kept on *merge*, this is detected by comparing the version vectors in the payload.

A complete specification of this CvRDT is shown in algorithm 5. The concurrent specification for conflicting assign operations is $\forall u_i = assign(e) \Rightarrow e \in S$.

Algorithm 5 Multi-Value Register

1: payload set S ▷ set of (x, V) pairs; $x \in X; V$ its version vector
2: initial $\{(\perp, [0, ...0])\}$
3: update assign (element e)
4: let $V := V \cup unique()$ ▷ Generates a event that identifies the operation and process that generated it
5: $S := \{(e, V)\}$
6: query value () : set S'
7: let $S' = S$
8: compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDTs must be MV-Register
9: let $b = (\forall (x, V) \in X, \forall (y, V) \in Y) : V \leq V'$
10: merge (CvRDT A, CvRDT B) : CvRDT Z ▷ CvRDTs must be Multi-Value Register
11: let $A' = \{(x, V) \in A | \forall (y, W) \in B : V \| W \vee V \geq W\}$
12: let $B' = \{(y, W) \in B | \forall (x, V) \in A : W \| V \vee W \geq V\}$
13: $Z.S := A' \cup B'$

3.4.5 Treedoc

Treedoc is a CvRDT designed to manage data in a collaborative text editing tool. Treedoc was originally presented in [PMSL09]. In the original version, to replicate the object, sites propagate the operations executed in the Treedoc. In this version the state is propagated instead.

Treedoc is inspired by a binary tree, where nodes store atoms (the text units, which can be characters, paragraphs, etc.) and each atom is identified unequivocally. The interface of the Treedoc allows users to insert new atoms in the tree by position, making it very suitable to develop a text editor.

Whenever a user inserts a new atom, the position is converted to an identifier which will be used to identify the node in the tree. Similar to what is done in WOOT, Treedoc maintains an immutable unique identifier for each node and operations are idempotent.

3.4.5.1 Paths as unique identifiers

Identifiers domain must be dense to allow insertion of atoms in any position. This means that between any two identifiers there can be created another identifier. They must be unique and constant over time for each node. Paths are used as identifiers for tree nodes generated at different sources. Paths are represented by a bit string and satisfy the density requirements. The total order of identifiers is given by travelling the tree, from the root, in infix order.

Using simple binary trees, two different sites can generate the same tree path. To preclude different sites from generating the same identifier, each identifier has an associated unique disambiguator. Nodes must be placed in the same position on the tree and still have to be ordered between them. To make it possible, the tree structure was extended with major-nodes which store the conflicting nodes for the same identifier, called mini-nodes. In figure 3.2, node with identifier 100 is an example of a major node containing two mini-nodes.

Disambiguators are unique identifiers to distinguish tree nodes. These disambiguators are represented as pairs of *siteID* and *counters*. The $<$ is a total order relation that compares disambiguator as follows: $(c_1, s_1) < (c_2, s_2)$, *iff* $c_1 < c_2$ or $c_1 = c_2 \wedge s_1 < s_2$. Using this disambiguators, sites can discard deleted nodes as soon as they are removed because they are unique and no replica can generate the same identifier.

Disambiguators are also used to track causality of operations and each Treedoc maintains a version vector that includes all the generated disambiguators.

3.4.5.2 CvRDT Treedoc

Treedoc operations are *insert(pos,atom)* and *remove(pos)*, *merge(A,B)*, *compare(A,B)*, as shown in algorithm 6. When a client adds an atom at a given position x, the insert algorithm will generate a new identifier between node occupying pos and its precedent. The new node will be the child of one of those selected nodes which has no descendent in the tree. As an example, in figure 3.2, node with content d was inserted between nodes with identifier root (*nil*) and 1. It was appended to the left of 1 with identifier 10.

The remove operation receives a position of the atom the client intends to remove and looks for the node with that position, in the atom buffer, to remove it. Nodes are discarded immediately.

The merge operation receives two Treedoc CRDTs and merge them, creating a new one that reflects the executed operations in both. To handle removed elements, Treedoc requires a version vector. Similarly to what is done in OR-Set without tombstones, if a Treedoc contains a node

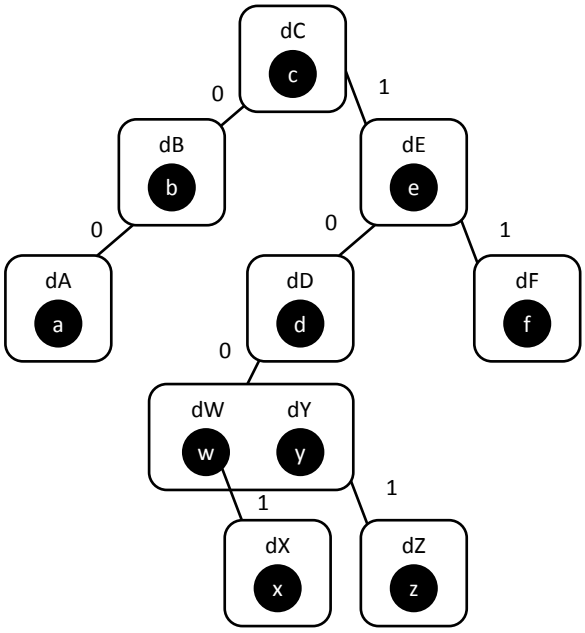


Figure 3.2: Example of a Treedoc tree

that does not belong to the other merging tree but the node's disambiguator belongs to the Version Vector, then the node was removed in the other Treedoc and it must be discarded. Merge must deal with concurrent operations also, if two clients generated the same identifier, a new major-node must be created and the conflicting nodes are insert in the major-node ordered by disambiguator.

Atom W and Y, in figure 3.2, illustrate the execution of two concurrent operations: Two different clients generated the same node identifier 100 and each assigned a disambiguator dW and dY. When the replicas were merged, a new major-node was created do store both mini-nodes. Since the disamguigators have a total order relation (dW < dY), they were stored with the same order across different clients.

Treedoc trees are also comparable, they have a partial order relation and the merge operation create a new Treedoc that is the LUB of the merged Treedocs.

3.5 CRDTs with Version Control

In the previous CRDT designs, we observe that elements are tagged with unique identifiers and, in some implementations, tombstones are kept when elements are removed. It is possible to reconstruct the evolution of a CRDT or undo operations with the unique identifiers stored.

To read and update previous object versions it was necessary to change the interface of CRDTs. The interface now requires users to specify the version vector of the state that the user wants to access, both for query or update operations. If the version that a user requests is newer than the object's version, then the most recent state of the object will be delivered.

This is a new usage for CRDTs which allowed us to implement transactions with CRDTs on

Algorithm 6 Treedoc CvRDT specification

payload OrderedList Q, VersionVector V ▷ Ordered list of elements $((path, disambiguator), atom)$, elements ordered by *id* $(path, disambiguator)$
  initial nil
update $insert$ ($int\ pos$, $any\ atom$)
  let $prev = Q$[pos-1]
  let $next = Q$[pos]
  let $id = newID(prev, next)$
  $Q.put(id, atom)$ ▷ *put* inserts the element *atom* ordered by *id*
  $V = V \cup \{disamb(id)\}$ ▷ *disamb* return the disambiguator of the identifier *id*
update $remove$ ($int\ pos$)
  $Q.remove(pos)$ ▷ removes the pos^{th} element from Q
compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDTs must be Treedoc
  let $R = \{id | t \in X.V \wedge \nexists atom : (id, atom) \in X.Q\}$
  let $R' = \{id | t \in Y.V \wedge \nexists atom : (id, atom) \in Y.Q\}$
  $b = X.V \leq Y.V \wedge R \subseteq R'$ ▷ Compare the remove sets
merge (CvRDT A, CvRDT B) : CvRDT Z ▷ CvRDTs must be Treedoc
  let $new = \{(id, atom) | (id, atom) \in B.Q \wedge disamb(id) \notin A.V\}$
  let $rem = \{(id, atom) | (id, atom) \in (A.Q \setminus B.Q) \wedge disamb(id) \in B.V\}$
  let $Z.Q = (A \cup new \setminus rem)$ ▷ union maintains the queue ordered
  $Z.V[i] = max(A.V[i], B.V[i]), \forall_i$
query $newID$ ($id\ uid_p$, $id\ uid_n$) : id id
  Require: $uid_p < uid_n$
  if $\exists$ a mini-node with id such that $uid_p < uid_m < uid_n$ **then return** $newID(uid_p, uid_m)$
  else if $uid_p /^+ uid_n$ **then**
  let $uid_n = c_1 \odot ... \odot (p_i : u_i)$; **return** $c_1 \odot ... \odot (p_i) \odot (0 : d)$
  else if $uid_n /^+ uid_p$ **then**
  let $uid_p = c_1 \odot ... \odot (p_i : u_i)$; **return** $c_1 \odot ... \odot (p_i) \odot (1 : d)$
  else if $MiniSibling(uid_p, uid_n)$ **then return** $uid_p \odot (1 : d)$
  else
  let $uid_p = c1 \odot ... \odot (p_i : u_i)$; **return** $c1 \odot ... \odot p_i \odot (1 : d)$

top of an eventual consistent system. This system will be later discussed. In this section we present the specification for CvRDT counter, register, set and Treedoc with version control support.

3.5.1 Versioned Counter

The versioned Counter implementation tags every increment/decrement operation with an identifier. When a user requests the value, with a version vector VV, the return value will only contain the increments/decrements whose identifiers belong to VV. The operations that update the state now generate a new unique identifier that is associated to the operation.

Algorithm 7 CvRDT Integer Counter with Versions

1: payload Set inc, Set dec ▷ Set of elements (integer,id)
2: initial $\varnothing, \varnothing$
3: update $increment$ ()
4: let $t = unique()$
5: $inc := inc \cup \{(1, t)\}$
6: update $decrement$ ()
7: let $t = unique()$
8: let $dec := dec \cup \{(1, t)\}$
9: query value (*VersionVector* VV) : $integer\ v$
10: let $incs = count(\{(1, t)|\forall_t(1, t) \in inc \wedge t \in VV\})$
11: let $decs = count(\{(1, t)|\forall_t(1, t) \in dec \wedge t \in VV\})$
12: $v = incs - decs$
13: compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDT must be Integer Counter with Versions
14: let $V = \{t|(1, t) \in X.inc\} \cup \{t|(1, t) \in X.dec\}$
15: let $V' = \{t|(1, t) \in Y.inc\} \cup \{t|(1, t) \in Y.dec\}$
16: let $b = V \subseteq V'$
17: merge (CvRDT A, CvRDT B) : CvRDT Z
18: let $Z.inc := A.inc \cup B.inc$
19: let $Z.dec := A.dec \cup B.dec$

3.5.2 Versioned MV-Register

The versioned Register maintains the history of values assigned to the register. Given a version vector that represents a point in the past, the CRDT returns the state in that moment. To deliver this behaviour it is necessary to store every value that has been assigned to the MV-Register alongside with the Version Vector that contains all the operations up to that moment.

To assign a new value to the register it is necessary to indicate a version vector. To read a register, the CRDT requires that the programmer specifies which version he wants to read by providing a Version Vector. The set of values to be delivered contains the values that have the greatest version vector and its concurrents that are smaller, equal or concurrent with the given one.

Algorithm 8 Versioned MV-Register specification

```
1: payload set S                                   ▷ set of (element x, VersionVector V) pairs;
2: initial {(⊥, [0, ...0])}
3: update assign (element e, VersionVector VV)
4:     let V = unique()
5:     S := S ∪ {(e, V)}
6: query value (VersionVector VV) : set S''
7:     let S' = {(x, V)|(x, V) ∈ S ∧ (V ≤ VV ∨ V‖VV)}
8:     let S'' = {(x, V)|(x, V) ∈ S' ∧ ∀V' : (y, V') ∈ S' : V ≥ V' ∨ V‖V'}
9: query unique (VersionVector VV) : VersionVector V'
10:    let p = myId()
11:    V'[i] = max({V[i]|(x, v) ∈ S}), ∀i
12:    V'[p] = V'[p] + 1
13: compare (CvRDT X, CvRDT Y) : boolean b        ▷ CvRDT must be MV-Register Versions
14:    let V = {V|(x, V) ∈ X.S}
15:    let V = {V|(x, V) ∈ Y.S}
16:    let b = V ⊆ V
17: merge (CvRDT A, CvRDT B) : CvRDT Z
18:    Z.S := A.S ∪ B.S
```

3.5.3 Versioned Set

The original OR-set implementations have an identifier associated to each value. The versioned implementation has a new identifier placeholder to indicate that the element has been removed. To remove an element e from the set, the user must specify the version vector VV of the version that he is modifying. Every insertion of e with an identifier reflected in VV is marked as deleted with a new identifier. The version is used on remove operation to avoid marking as removed elements that were not yet inserted according to VV. When assessing the state of the set, we compare the given version vector with the identifiers associated to each element, to include or exclude it from the current state. As before, it is possible to implement a version of a CRDT Set with add-wins or remove-win semantic on concurrent operations.

A versioned Map can be implemented in similar way.

3.5.4 Versioned Treedoc

Treedoc was also extended to include versioning support. The versioning support in Treedoc allows to implement the undo operation in a collaborative text editor. To read the state of the tree, the interface requires the programmer to specify the state he wants to read. As in the previous implementations we use a version vector to indicate the state that the programmer wants to read.

To allow reading previous states, nodes are no longer discarded when a remove operation is executed. Instead, the nodes of the Treedoc were extended to include a set of remove identifiers. When a remove operation is issued, a new unique identifier is added to the set. It is necessary to store a set of removed identifiers to keep all the generated identifiers of concurrent remove operations. The merge operation, instead of discarding the elements that were removed, joins the

Algorithm 9 Add-Wins Versioned Set specification

1: payload set E ▷ E: elements; element (e, insertID, removeID)
2: initial $\varnothing$
3: update add (element e, VersionVector VV)
4: let $id = unique()$
5: $E := E \cup \{(e, id, nil)\}$
6: update remove (element e, VersionVector VV)
7: let $rem = unique()$
8: let $E' = \{(e, id, nil) \in E | id \in VV\}$
9: let $E'' = \{(e, id, rem) | (e, id, nil) \in E \wedge id \in VV\}$
10: $E = (E \setminus E') \cup E''$
11: query contains (element e, VersionVector VV) : boolean b
12: let $b = \exists (e, id, r) \in E : id \in VV \wedge r \notin VV \wedge (\forall_{r'} : (e, id, r') \in E \wedge (r' = nil \vee r' \notin VV)$
13: query elements () : set S
14: let $S = \{e | (e, id, r) \in E \wedge id \in VV \wedge r \notin VV \wedge (\forall_{r'} : (e, id, r') \in E \wedge (r' = nil \vee r' \notin VV)\}$
15: compare (CvRDT X, CvRDT Y) : boolean b ▷ CvRDT must be Add-Wins Versioned Set
16: let $ids = \{ins | (e, ins, rem) \in X.E\} \cup \{rem | (e, ins, rem) \in X.E \wedge rem \neq nil\}$
17: let $ids' = \{ins | (e, ins, rem) \in Y.E\} \cup \{rem | (e, ins, rem) \in Y.E \wedge rem \neq nil\}$
18: let $b = ids \subseteq ids'$
19: merge (A, B) : Z
20: let $remSet = \{(e, ins, nil) \in A.E \cup B.E | (e, ins, rem) \in A.E \cup B.E \wedge rem \neq nil\}$
21: $Z.E := (A.E \cup B.E) \setminus remSet$

set of removed elements of each node.

3.5.5 Permanent Rollback

The CRDTs shown in this section allow the programmer to control the values that he wants to read. For instance, it is possible to hide every action from a certain replica, or, by using the information provided by version vectors, read the CRDT according to a certain isolation level.

In transactional systems, it is often necessary to rollback updates due to transactions aborts. In CRDTs where the values are tagged with unique identifiers, it is possible to undo the update that generated them, by removing the unique tags from the payload. The rollback operation depends on the CRDT implementation, since values may be stored in different ways. In algorithm 10 we show the rollback operation for the Set Versioned CRDT.

Algorithm 10 Set Versioned Rollback

1: update rollback (identifier id_i)
2: $S := S \setminus \{(e, id_i, \_)\}$
3: let $S' = \{(e, id, r) \in S | r = id_i\}$
4: let $S'' = \{(e, id, nil) | (e, id, rem) \in S'\}$
5: $S = S \cup S''$

Removing the effect of an operation can make the CRDT inconsistent, because the effect of

a rollback can break causality, i.e., if we execute an operation that generates an identifier id_i and then we execute an operation that generates identifier id_{i+1} and we only rollback the first one, this would break causality, because the object state would reflect an operation that depends on other operation that is not present in the object's payload.

3.5.6 Generic Version Control

The support for versioning originated new CRDT designs. A specific implementation for each CRDT structure may deliver the best performance, however the programmer may not be interested in developing specific CRDTs to deliver version support. A generic approach is interesting because it allows us to deploy versioning in any CRDT implementation.

We have seen that a new identifier is generated for each executed operation in a Versioned CRDT. This is what enables hiding the effect of an operation when we access a previous state of the CRDT using a Version Vector. If we store the state of the object for each executed operation, then we can recover the state of the object in the moment the operation was executed. To reconstruct a previous state of a CRDT, we merge all the CRDTs associated to identifiers that belong to the Version Vector that reflects the requested state. If we use an implementation of a version vector that encodes all the identifiers generated at the same source in one entry, then the number of merges is linear with the number of sources.

In the previous algorithms, we specified the comparison of CRDTs. However, we can also do a generic implementation based just on the comparison of the Version Vectors associated to each CRDT with the rules to compare Version Vectors [Fid88].

4

SwiftCloud Design

The purpose of this work was to develop a Key-CRDT Store. As it was simpler not to create our own Key-Value store, we built SwiftCloud as a layer on top of an existing one. We have chosen Riak as our base Key-Value store. Unlike existing Key-Value stores, SwiftCloud presents a data-model with automatic conflict resolution that merges concurrent streams of activity. Furthermore, we implemented a transactional version of the system that allows a set of updates to be executed atomically.

In this section we will present the system architecture, discuss the relevant design decisions and detail the design of the transactional system. We start this section by doing an overview of Riak, since we built SwiftCloud on top of it.

4.1 Riak

Riak is a distributed Key-Value store. A Riak cluster is composed by a set of nodes that can be installed in the same cluster, or be geographically disperse. Nodes form a ring and distribute values among them. Each node contains a local storage on which it stores data. Data is replicated among the ring to handle network partitioning and server failures. In our prototype, we use the default storage system, Bitcask[1]. We use the Riak enterprise edition to have multi-cluster replication.

Our implementation does not change any components of Riak. SwiftClient's view of the system is limited by the functionalities of the Java Riak Client. Developing our system using this middleware approach can hurt performance since we do not have direct access to replicas, or stored objects, but results show that it is not significant. On the positive side, we rely on an unmodified product and we can benefit from any improvement in the system.

[1] https://github.com/basho/bitcask

Listing 4.1: Establishing connections to Riak nodes with Riak Java Client

```
1 //Protobuf Client
2 RiakClient pbcClient = new RiakClient("127.0.0.1");
3 // OR HTTP client
4 com.basho.riak.client.http.RiakClient httpClient = new RiakClient("http
      ://127.0.0.1:8098/riak");
5 RawClient rawClient = new PBClientAdapter(pbcClient);
```

4.1.1 Riak Java Client

Riak provides a REST and a protobuf[2] API, which the programmer can use to fetch and store data directly in the Key-Value store. However, these interfaces are verbose and difficult to use without any library that generates the messages automatically. There are many client implementations in different languages to simplify the communication. We use Riak Java Client[3] as our abstraction to communicate with the Key-Value store.

The interface of the Key-Value store is very simple, its purpose is to store and retrieve byte arrays from the database. A *(bucket, key)* pair has an associated value and the programmer must specify a read *Quorum* when reading a value. The read *Quorum* establishes the number of replicas that reply to a read operations. To store data, the programmer must specify the number or replicas he wants to write before the write is complete, a write *Quorum*. Besides the read/write *Quorum*, Riak requires programmer to specify the number of replicas for an object. By default the number of replicas is three. Riak can be configured to store multiple values for the same key and if an operation fails on a certain node, the programmer can overload the default retry function. We now present a subset of the API that we consider relevant to understand the interaction model of the client. Riak also provides map-reduce and search capabilities that are not relevant to this work.

4.1.2 Riak API

The API provides functionalities that allow querying and storing data, while hiding communication details. The API contains methods to connect to clients, create buckets and interact with objects. There are two different interfaces for the Riak client, the High level and low level API. In our work, we use the low level API, which provides direct access to messages with lower overhead.

Connecting to the Cluster To create a connection to a Riak node, we declare a new client, whose type corresponds to one of the available communication protocol, http or protobuf. A connection pool is created for each node to provide parallel access to the node. Connections can be configured with the number of attempts to do an operation, the time limit to get a connection and buffer size. Listing 4.1 shows an example of how to establish a connection to a Riak node.

Database operations The available operations in the Riak Key-Value store are very simple, giving programmers the ability to fetch and store data in buckets. Basically, to fetch an object, the

[2] http://code.google.com/p/protobuf/

[3] https://github.com/basho/riak-java-client

Listing 4.2: Store an object with Riak Java Client

```
IRiakObject riakObject = RiakObjectBuilder.newBuilder(bucketName, "key1").
    withValue("value1").build();
rawClient.store(riakObject, new StoreMeta(2, 1, false));
RiakResponse fetched = rawClient.fetch(bucketName, "key1");
IRiakObject result = null;
if(fetched.hasValue())
    if(fetched.hasSiblings())
        //resolve conflicts
    else
     result = fetched.getRiakObjects()[0];
// ...code...
RiakResponse stored = rawClient.store(result, writeQuorum);
rawClient.delete(bucketName, "key1");
```

user execute the *fetch* operation on a Riak node, specifying the Bucket and Key which he wants to read. The replied messages are represented by *RiakFetchResponse* object, which wraps the retrieved values and meta-data. This object may contain conflicting stored values and users must provide a conflict resolution handler if that case. Programmer may also specify the read *Quorum* during the fetch operation. The read and write *Quorums* must intercept to guarantee that the client reads the most recent value for that key. To store an object, first the programmer must create a new *IRiakObject* and specify the data and the *(bucket, key)* in which it will be stored. The *store* operation requires the *IRiakObject* that has been just created and the write *Quorum* to store the data on the database. Listing 4.2 demonstrates how to use the Riak client to fetch a key from a bucket, store it again and delete it.

Buckets are created automatically when the programmer stores a key in a bucket that does not exist. Yet, the programmer can also specify a new bucket and define its properties. The parameters to create a bucket are the replication factor for each key and the strategy for dealing with conflicts that can be last writer wins or keep multi versions. When using the keep multi-versions conflict resolution, it is important to always fetch an object before storing it, even when we want to overwrite it. The system internally stores a version vector to order the operations, if the programmer was creating a new value, but the key already contains data, and the new value is stored blindly, the system is not able to detect that the new value is newer than the previous one and stores both.

4.2 SwiftCloud Architecture

SwiftCloud is a middleware system that implements a Key-CRDT on top of Riak. Thus, SwiftCloud transforms a Key-Value store in a Key-CRDT store, with a data-model that uses CRDTs to provide strong eventual consistency. The system allows clients to execute updates concurrently without any coordination and guarantees that the client sees no conflicts on concurrent updates, by applying automatic conflict resolution defined in CRDTs. Programmers that want to use SwiftCloud must use CvRDT implementations for their objects, e.g., they can use the CRDT Library that we have implemented. Other CvRDTs can be added by the programmer.

Figure 4.1 shows the architecture of SwiftCloud. The SwiftCloud middleware was developed over Riak, by interacting only with the Java Riak client. The interface of the SwiftClient works as a wrapper for Riak Java Client methods. Thus, it allows to fetch and store CRDT objects and do automatic merge of conflicting CRDTs. Applications use our system as a client to access the SwiftCloud system.

In the transactional version of SwiftCloud, we have another component to control transactions, the transaction server (TxServer). The transaction server is responsible for managing the version of the database and handle the executing transactions, as we explain later. The TxServer may have to roll-back modification on the CRDTs, so it requires the SwiftCloud middleware to contact the Riak Cluster.

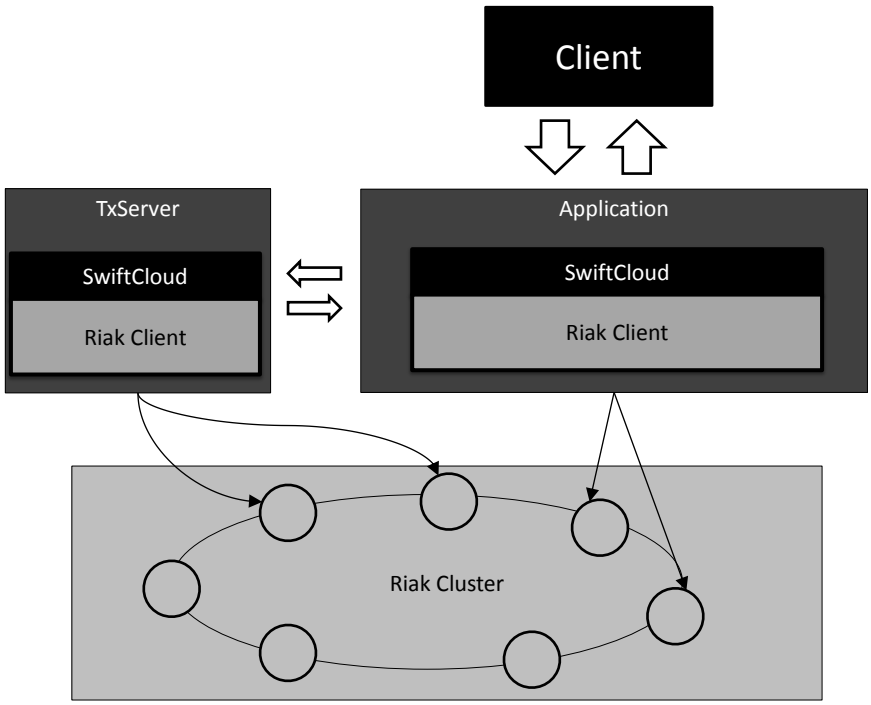


Figure 4.1: SwiftCloud Architecture

A deployment of SwiftCloud will include a set of nodes running an unmodified version of Riak. Applications can run in the same or different machines. In a typical environment for serving web applications, the client of SwiftCloud will run in application servers processing web requests executed by users.

4.3 Data-model

The SwiftCloud data-model is inherited from the Riak data-model. Data is organized in buckets that have a name. Inside each bucket, objects are indexed by key. Buckets are similar to tables in a database system, and objects correspond to rows indexed by keys. Objects are simple arrays of bytes. Each object has the serialized CRDT, meta-data describing the type of the object and its version (a CausalityClock). If an object is concurrently modified, conflicting values and their

meta-data are kept in the underlying Riak system, until a new version is stored.

Fetch and *Store* operations access stored CRDTs and serialize/deserialize them automatically. To deliver a consistent version of the CRDT to the application, the *fetch* operation merges automatically the conflicting updates, when it is necessary.

The system requires all objects to be CRDTs, to provide conflict-free replicated objects to programmers. We have seen in chapter 3 that some designs require a Version Vector to do conflict resolution. For this reason, a Version Vector (CausalityClock) is stored in the meta-data of CRDTs to identify its version.

Objects are stored as binary data in Riak, for this reason we have to serialize objects before storing them. Our system has an interface to handle CvRDT serialization to allow programmers to provide their own serialization support. SwiftClouds provide a default JSON serializer which supports CRDT composition, allowing the distribution of CRDTs among different keys in the Riak database. This feature can avoid fetching a large CRDT just to access a part of it. Instead of storing all values, a lazy CRDT contains only references to other CRDTs, which are stored elsewhere, being fetched only if the client requests them.

4.4 Interface

SwiftCloud API provides functionalities to store and fetch data, list keys and create buckets as it is common in Key-Value Stores. Listing 4.3 shows the interface of SwiftClient and the SwiftObject. SwiftObject is the interface that is responsible for managing the CRDT meta-data, as well as merging conflicting values.

To fetch a CRDT it is necessary to specify the *(bucket, key)* pair and the type of the object being accessed. The result is wrapped in a *SwiftFetchResponse* object. This object contains the raw data fetched from the server, which consists in the native vector clock, the CRDT meta-data, the value and its siblings (if any). To get the CRDT, first the programmer must get a *SwiftObject* from the *SwiftFetchResponse*. To build this wrapper, the CRDT value and its siblings (if any) are deserialized and merged pairwise until there is only a single CRDT that contains all concurrent updates. The clocks associated to CRDTs are used during the *merge* operation. Clocks are also merged and the resulting clock is associated to the CRDT, before delivering it to the client.

To store a new object, the programmer uses the *createObject* method to create a SwiftObject, by specifying the CRDT value and its type. When creating a SwiftObject, the system creates a *RiakObject* with the serialized CRDT and all data associated to it. If the programmer indicates an interface or a generic type, then the class name of the object is stored alongside with the value. To store an updated CRDT, the programmer must call the *update* on the corresponding SwiftObject before storing it.

CreateBucket creates a new Bucket in the Riak database. By default, buckets keep multiple values for conflicting keys. If the Bucket already exists, instead of creating a new bucket, the *createBucket* operation overrides the configuration of the specified bucket and enables the keep multi versions conflict resolution technique. *Delete* eliminates the object associated with a given key from the specific bucket. *getKeys* retrieves an iterator with all keys from the given bucket.

Listing 4.3: SwiftClient Interface

```
1  public interface SwiftClient {
2    /**
3    * Creates a new bucket in the database
4    * @param bucketName - the name of the bucket
5    */
6      void createBucket(String bucketName) throws RiakRetryFailedException;
7    /**
8    * Stores an object in the database
9    * @param obj - the object being stored
10   */
11   <V extends CvRDT> void store(SwiftObject<V> obj) throws IOException,
         SwiftException;
12   /**
13   * Fetches the object at the given bucket and key, from the database.
14   * @param bucket - the bucket where the object is stored
15   * @param key - the key where the object is stored
16   * @param type - the type of the object being fetched
17   */
18   <V extends CvRDT> SwiftFetchResponse<V> fetch(String bucket, String key,
         TypeToken<V> type) throws IOException;
19   /**
20   * Deletes the object at the given bucket and key.
21   * @param bucket - the bucket where the object is stored
22   * @param key - the key where the object is stored
23   */
24   void delete(String bucket, String key) throws IOException;
25   /**
26   * Get the list of keys from a bucket
27   * @param bucketName - the name of the bucket
28   */
29   Iterable<String> getKeys(String bucketName) throws IOException;
30   /** Creates a new SwiftObject with the given CRDT
31   * @param bucketName - the name of the bucket
32   */
33   <V extends CvRDT> SwiftObject<V> createObject(String bucket, String key, V
         crdt, TypeToken<V> type) throws SwiftException;
34 }
35
36 public interface SwiftObject<V extends CvRDT> {
37
38   /** Gets the CRDT object with conflicts merged. */
39   V getValue();
40
41   /** Updates the version of the object */
42   void updated() throws CvRDTSerializationException;
43 }
```

Since the number of keys can be large, the latency of this operation can become high. To avoid that, the set of keys is delivered to the programmer using chunked transfer encoding, allowing the programmer to start reading the keys before they are totally delivered.

To access the Riak Key-Value store it is necessary to specify the Read/Write *Quorums*. The programmer can specify whichever configuration that best suits his application. However, SwiftCloud does not require to read or write any version of the object, as CRDTs are used to assure data converge and conflict resolution. This way, if a CRDT updates stale version of a CRDT, the operation will be added to other versions of the CRDT after synchronizing.

4.4.1 Execution Protocol

SwiftCloud client implements the API by simply calling the underlying Riak API. Objects must be serialized/deserialized on store/fetch operations. SwiftCloud specific meta-data is stored in Riak object meta-data.

4.5 CRDT Composition

CRDT composition allows the creation of a CRDT that is composed by a set of other CRDTs. The composition of CRDTs is still a CRDT [SPBZ11a]. Examples of composite CRDTs are a Set of Registers or a Shopping cart composed by a CRDT Register, to store the billing information, and an OR-Set pairs of item and quantity.

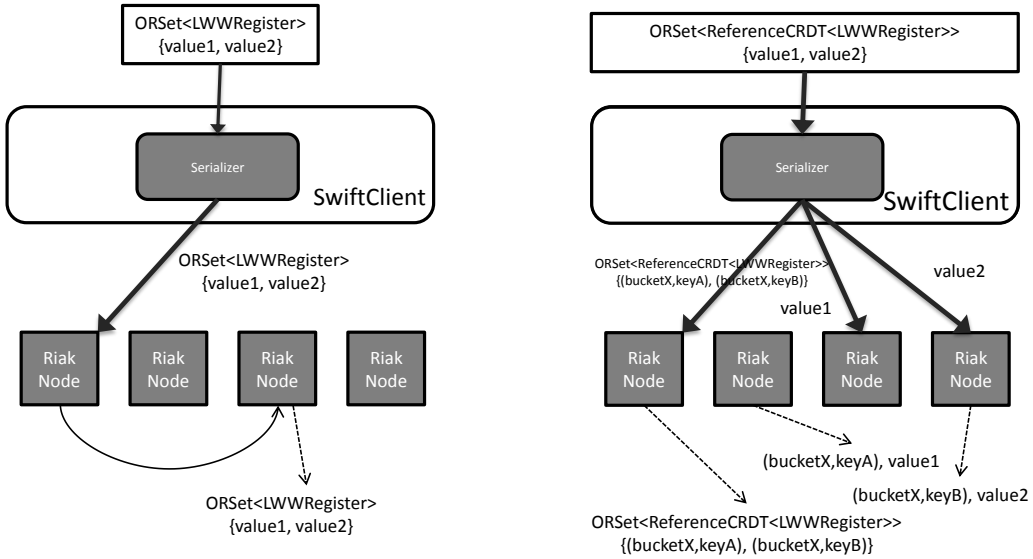
In some situations, the size of the data stored in the CRDT can be very large. For instance, if we are creating an image gallery using a CRDT, we are not interested in fetching all the images to display just one of them. To address this problem, we extended the client and the serializer to allow storing CRDTs with a level of indirection. The purpose of the devised solution is to enable composite CRDTs to be scattered over the storage system. A requirement of the system is that the scattered values must also be CRDTs. This way, the set of images would become a set of references to images and the binary data of the images would be stored in different *(bucket, key)* pairs.

To add the level of indirection we had to create a new CRDT type called ReferenceCRDT. This CRDT is a wrapper for CRDTs, which stores the *(bucket, key)* pair of the location where the wrapped value will be stored. Two conflicting ReferenceCRDTs will have their values merged and the *(bucket, key)* pairs updated according to last-writer-wins[4] strategy.

Figure 4.2 illustrates how a CRDT is stored with and without references. In figure 4.2a, a Set is stored without scattering its values. The object is completely stored in the same key, assigned to a node selected by Riak. In figure 4.2b, the serializer creates a reference for each value in the Set and substitute the values with the references in its payload. The Set is stored in the given key and the values are stored in different keys, possibly in different Riak nodes.

To retrieve the stored references we provide two different strategies, the eager and lazy strategies. These strategies differ in the moment they fetch the stored references. In the eager approach,

[4]This may lead to objects that are no longer referenced. For collecting the space used, a garbage-collection mechanism would have to be implemented



(a) Using simple JSON Serializer

(b) Using CRDTReference

Figure 4.2: Examples of storing a CRDT

when fetching a CRDT, all its values are also fetched from the Key-Value store. Conversely, in the lazy approach, values are only fetched when they are accessed.

This mechanism would not be very simple to use if the programmer had to specify the location of the references by himself when storing objects. To address this issue, we have implemented an automatic mechanism in which the fields of the CRDT that uses References are annotated with the rules to define the bucket and keys for the reference values. Listing 4.4 shows how to define a shopping cart composed by a map of item identifiers and their information.

The programmer may specify the bucket in which he wants to store the values or use the default, which is to use the parent bucket name extended with the suffix "_REFERENCE". The key generation has two different methods, public and private.

The public rule stores the objects in a bucket that can be globally accessed, and that keys may be shared by different objects. The private rule allow references to be stored in a bucket accessible by other clients, however the keys are particular to the objects that stores them, hence only that object access those keys. The keys are computed in order to be unique for the CRDT that stores the reference, in both strategies. However, with the private method, a suffix is added to the key in order to make it also unique for the client that stored the object.

To better understand the two strategies, consider the example of listing 4.4, which represents a shopping cart composed by a register that stores the cart information and a map to store the items in the cart. Using the private strategy, if two users have concurrently added an item to the cart with the same identifier, the two items would have been stored in different keys. Thus, the last writer wins policy of the reference would be used. In the public method, both lines would be stored in the same key leading to the use of the CRDT rules for merging concurrent updates.

Listing 4.4: CRDT Shopping Cart implementation using References

```
1
2 public class ShoppingCart implements CvRDT {
3   private ICRDTRegister<ShoppingCartInfo> info;
4   //The Key is the identifier of the items and SCLine is the associated
        information.
5   @StoreReference(ReferenceBucket = "", privateKey = true)
6   private ORMap<Integer, CvRDTReference<SCLine>> lines;
7   ...
8 }
```

4.6 Transactional SwiftCloud

This section describes Transactional SwiftCloud, a version of our middleware that supports transaction in a system with only eventual consistency guarantees. Since we rely on a weak consistency model, it is not possible to provide the exact ACID guarantees of transactional systems.

The semantics of the transactions is the following: transactions access a consistent snapshot of the database, i.e., a transaction does not see an intermediate state of any other transaction, similar to the guarantees given with Snapshot Isolation. Unlike Snapshot Isolation, transactions always commit. This can be achieved relying on the the properties of CRDTs that allows concurrent operations to be merged automatically. We call our isolation level mergeable snapshot isolation. Next, we explain how we achieve it.

4.6.1 Consistent Snapshots with Versioned CRDTs

The operations of a versioned CRDT generate unique identifiers. A version of a versioned CRDT can be identified by the set of unique identifiers generated by all executed operations. As explained before, this set can be efficiently stored in a causality clock if identifiers have the form *(Counter, SiteId)*. If the same source of identifiers is used for operations in all CRDTs, it is possible to use the causality clock to represented a version that spans all objects in the system.

A transaction consists of multiple operations. To guarantee that all operations in a transaction are visible or none, a causality clock must include identifiers of all operations in a transaction or none. To address this problem in a simple way, we have assigned to each transaction, a transaction identifier *(Counter, SiteId)*. Unique identifiers generated inside a transaction have the form *(Counter, SiteId, TxCounter)*, with *TxCounter* generated using a counter for each transaction. The causality clock only records the transaction identifiers, thus making visible all operations or none.

Transaction identifiers can be generated at client side, or in a server. Generating identifiers at the client requires having one entry for each client in the causality clock. Instead, we use a server to generate the transactions identifiers. When clients execute operations within transactions, they generate events using the transaction identifier from the server and a transaction local sequence number.

4.6.2 Transaction Servers

To maintain the version of the database and generate transaction identifiers, we use a transaction server (TxServer). This server has a CausalityClock that summarises all the finished transactions in the system, thus it reflects the current version of the database. When a client starts a new transaction, the TxServer sends the current version of the database and the identifier of the new transaction. When the transaction finishes, the TxServer includes its identifier in the CausalityClock. Any transaction using the database version CausalityClock to read Versioned CRDTs will not see operations from transactions that not have yet committed. TxServer will also intervene if a client fails.

There can be more then one TxServer running, to balance the load of the system. TxServers contact each other to update the global version of the database. To synchronize the different versions of the database, TxServers exchange their CausalityClocks and merge them.

Our system was designed in order to deploy a key-CRDT store that could be replicated and updated across data-centers. This allows us to provide better latency for users that are physically distant from the data-center (e.g. across ocean). Unfortunately, the underlying Riak cluster replication system does not provide guarantees on the order in which updates are propagated. For this reason, when relying on Riak inter-cluster replication for synchronization, we cannot guarantee that a data-center has already received all updates of a transaction when updating the data-center version. A possible heuristic is to delay the update of data-center version for some time. In order to guarantee a correct behavior in all situations, it would be necessary to handle inter-cluster replication at TxServer level to only include updates from a different cluster in the CausalityClock, after the updates are persistent in the database.

4.6.3 Interface

The interface of the client is restricted to start transactions and execute operations which are not transactional. The *fetch*, *store* and *delete* operations are now executed in the TransactionHandler, which is responsible for accessing the correct version of the objects, manage the CausalityClock updates, *commit* transactions and rollback them when necessary. This version of SwiftCloud can only be used with Versioned CRDTs. Listing 4.5 shows the interface of the SwiftClientTransactional and the TransactionHandler.

In the transactional version of SwiftCloud, we have to read a consistent state of the database, hence we must avoid reading stale replicas of the objects. To guarantee that the most recent version of the database is always read, the Riak replication parameters are configured such that $R + W > N$, with R, W being the Read, Write *Quorums* and N the replication factor. No update is lost because we keep conflicting values. This is necessary to ensure that, when the client requests a value, its version is at least as new as the CausalityClock of the transaction.

It is possible to access keys that existed in a bucket for a certain version of the database, but that may have been deleted in a posterior version of the database. For this reason, the *delete* operation does not remove keys, instead a tombstone of the value is kept and all the previous versions.

Listing 4.5: SwiftClientTransactional Interface

```
1  public interface SwiftClientTransactional {
2
3    /** Creates a new bucket in the database
4    * @param bucketName - the name of the bucket  */
5      void createBucket(String bucketName) throws RiakRetryFailedException;
6
7    /** Get the list of keys from a bucket
8    * @param bucketName - the name of the bucket  */
9    Iterable<String> getKeys(String bucketName) throws IOException;
10
11   /** Starts a new transaction at a client, by fetching the database version
         CausalityClock and the identifier of the new transaction. */
12   public TransactionHandler begin() throws SwiftException;
13 }
14
15 public interface TransactionHandler extends Serializable {
16   /** Commits the current transaction */
17     TRANSACTION_STATUS commit() throws SwiftException;
18
19   /** Abort the current transaction and rolls back its updates
20   * on the database, if any. */
21   void transactionRollback() throws SwiftException;
22
23   /** Fetches the object at the given bucket and key, from the database.
24   * @param bucket - the bucket where the object is stored
25   * @param key - the key where the object is stored
26   * @param type - the type of the object being fetched  */
27   <V extends VersionedCvRDT> SwiftFetchResponse<V> fetch(String bucket, String
         key, TypeToken<V> type) throws IOException;
28
29   /** Deletes the object at the given bucket and key.
30   * @param bucket - the bucket where the object is stored
31   * @param key - the key where the object is stored */
32   void delete(String bucket, String key) throws IOException;
33
34   /** Creates a new bucket in the database
35   * @param bucketName - the name of the bucket  */
36   <V extends VersionedCvRDT, X extends V> SwiftObject<V> createObject(String
         bucket, String key, X crdt, TypeToken<V> type) throws SwiftException;
37   /** Generates the identifier for the next operation at the given
38   * CRDT and marks the crdt to be stored when the transaction commits.
39   * @param crdt - The CRDT on which the operation will occur. */
40   EventClock nextOperation(CvRDT crdt) ;
41
42   /* Gets the CausalityClock that reflects a consistent version of the
43   * database that the transaction is allowed to access.*/
44   CausalityClock getClock() ;
45 }
```

Listing 4.6: Example showing SwiftClient Transactional interface

```
1    TransactionHandler th = swiftClient.begin();
2    SwiftObject<ShoppingCart> cartObj = th.fetch("shopping_cart", cart_id,
         shoppingCartTypeToken).getObject();
3    ShoppingCart cart =cartObj.getValue();
4    cart.addItem(item, th);
5    th.commit();
```

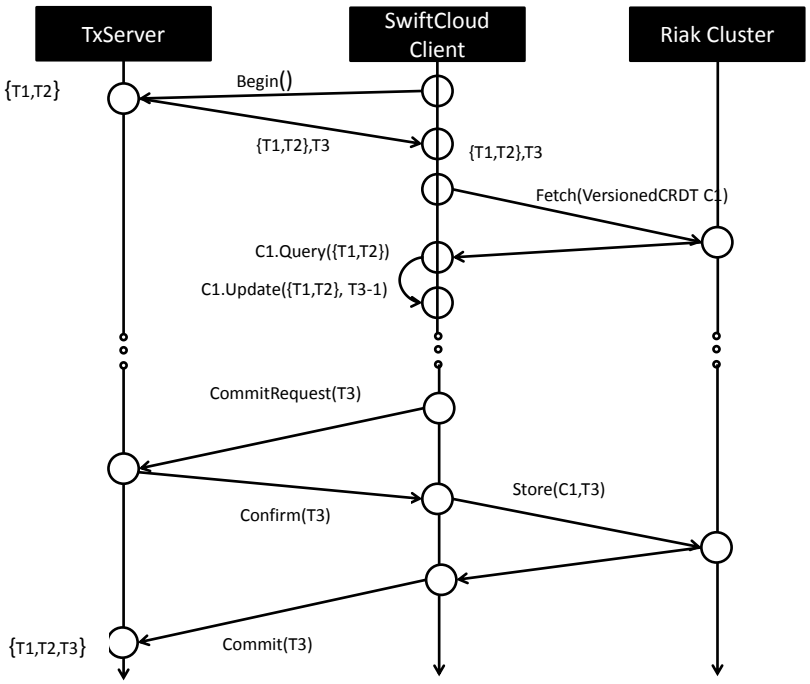


Figure 4.3: Transactions Protocol

Listing 4.6 shows an example of a program that adds an item to a shopping cart using SwiftCloud transactional. To start the transaction, the programmer executes the *begin* operation in SwiftClient interface. The client contacts a TxServer to get the version of the database and the identifier of the new transaction. Then, the client executes operations over the shopping cart, which is a Versioned CRDT. The *addItem* operation receives the transaction handler to access the state of the cart that corresponds to the version of the database that is visible to that transaction and to generate an identifier corresponding to the current transaction. The transaction ends with a *commit*.

4.6.3.1 Transaction protocol

The protocol for executing transactions is straightforward. The messages exchanged in the protocol are shown in figure 4.3. The client starts by sending a *begin* message to the TxServer to start a new transaction. The transaction server replies with the current database version and the new transaction identifier. In the example of the figure, the state reflects two executed transaction, T1

and T2 and the new transaction identifier is T3.

With the database version causality clock, the client accesses a consistent state of the Versioned CRDTs from the Riak database (updates are executed only locally and their propagation is delayed until the commit of the transaction). Whenever the client executes an update operation, the client generates a new event that identifies that operation ordered with the previously executed. The client contacts the TxServer to commit the executed transaction.

When the client wants to commit the transaction, he sends a commit request to the TxServer that contains the log of keys of modified CRDTs. This log allows the TxServer to abort the transaction if the client fails during the commit. The TxServer acknowledges the transaction and the client starts updating the database with the new transaction. This involves writing the new version of all modified CRDTs. When the client finishes, it informs the server that the transaction is persistent and the TxServer includes the transaction identifier in the version of the database by updating the causality clock.

Note that with this protocol, transactions might not complete in the order they have started. Thus, the used causality clock must allow adding identifiers in any order — we use version vectors with holes for supporting this functionality. This protocol also guarantees that the data-center version is only updated after the updates have been applied to the replicas. This guarantees that a client can always access the correct version of the database.

The protocol to update the transaction servers guarantee that if the client always use the same TxServer, the system will prove the usual session guarantees, such as read your writes, monotonic reads, writes follow read and monotonic writes [TDP+94]. The same does not occur if different servers are contacted. A client could assure some of these properties at the application level. For instance, to provide read your writes session guarantee, while allowing clients to access different TxServers, the application could store the last committed transaction identifier, and only access TxServers that contain that identifier. Monotonic reads could be achieved by maintaining the CausalityClock of the accessed version in the last transaction and, when starting a new transaction, propagate this information to the TxServer, where it would be merged with the local clock before returning the version to be accessed. This would make clients intermediates for synchronizing the CausalityClock in the different TxServers.

4.6.4 Handling Failures

We now explain how we address failures. We assume only fail-stop failures. The following cases must be considered.

First, the client can detect the failure of the TxServer before sending the commit request. In this case, the transaction aborts in the client and updates are discarded.

Second, the TxServer can detect the failure of the client before receiving the commit request (by timeout). In this case, the server simply includes the transaction identifier in the current data-center version to minimize the holes in the version vector.

Third, the TxServer can detect the failure of the client after receiving the commit request (by timeout). In this case, the client can have written some values in the Riak servers. To handle this

situation, the TxServer uses a log of modified keys sent by the client to undo the executed changes. To this end, it uses the Permanent Rollback procedure described in section 3.5.5 that navigates through the payload of a CRDT and removes the effects of an operation. Since operations are marked with the transaction identifier, we can request the permanent rollback using that identifier. This requires that all Versioned CRDTs support this procedure.

Finally, the client can detect the failure of the TxServer after the commit request. In this case, it is the client that rollbacks the objects written to the cluster until that moment.

When a transaction is aborted during commit, other clients could have fetched objects containing the values that were already written by the aborted transaction. If they commit their transactions, the removed values would appear again in the database, and it would be impossible to safely garbage collect the holes. To prevent this, when a TxServer aborts a transaction and other client sends a commit request that has not seen the identifier from the aborted transaction, the client is informed that he must rollback locally the transaction before storing the values to Riak.

4.6.5 Extensions

Our system implementation does not rely on persistent logs to recover the state of TxServers. In practice, it would be necessary to record the causality clock to recover the data center version.

The presented system always creates a new transaction identifier when a transaction starts. The number of ongoing transactions may pose a problem to the size of the database version vector (because this creates many holes in the CausalityClock). A solution that requires low effort to implement and minimizes the problem, would be to support identified *read-only* transactions. This type of transaction would receive a CausalityClock to read a consistent state of CRDTs but would not create a new transaction identifier. Depending on the application, we can expect the number of generated identifiers to decrease.

5

Implementation

In this section we will details SwiftCloud implementation. We have two versions: the non-transactional version allows fetching and storing values from the database regardless of the version read. The transactional version supports operation execution over a consistent state of the database and provides isolation from concurrent transactions executing in the system.

SwiftClient must handle connections to the Riak Cluster. In our implementation, we assume that it is done on top of Riak Java Client, but in practice, any underlying interface could be used to connect to Riak.

5.1 SwiftClient

We start by describing the non-transactional solution. In this solution, every operation is handled independently and it is immediately executed in the underlying Riak storage system.

SwiftClientWithMeta is the implementation of the SwiftClient interface in our system. To initialize the client it is necessary that the programmer specifies the location of a Riak node.

A *fetch(bucket,key)* request on Riak retrieves a RiakObject object. This object contains the binary value, or multiple values, if there were concurrent updates, an opaque version vector and the meta-data for each value. It is not possible to modify or read the Riak version vector of the object, so we store or own CausalityClcok in the meta-data to track the versions of values. If multiple clients have executed concurrent updates on the same *(bucket,key)* pair, the conflicting values are automatically merged in the SwiftClient before delivering them to the application.

The system can be configured in its degree of replication as explained in section 4.1.1. The read/write Quorum is essential to view objects state consistently. For example, if we set the number of written replicas on store operations to one, request only one replica on read and have more than one replica for the each object, it is not guaranteed that the returning value is the most recent

one. By default, we require accessing a *Quorum* of replicas. Some applications are fine with this and the convergence properties of CRDTs still hold. In this case, stale values can be read. The system replication factor and read/write *Quorum* can also be globally configured to hide that complexity from the programmer. On the other hand, a finer control on these properties allow better performance, by defining the most adequate read/write *Quorum* for each operation.

5.1.1 CausalityClocks

CausalityClock is a clock abstraction that allows to summarize a causal history, i.e. a set of events, and trace causal dependencies in our system.

We have implemented the following CausalityClocks: VersionVector and VersionVectorWithHoles. VersionVector implementation uses a Map that maps sources of operations to the maximum counter of Timestamps recorded. VersionVectorWithHoles extends the VersionVector implementation allowing to reflect a Causal History which has missing events. The implementation simply keeps missing events in a set.

The interface of CausalityClock is shown in listing 5.1. To represent events in CausalityClocks, we use the EventClock interface. The *record* operation records a new event in the causal history represented by the CausalityClock. The *includes* operation verifies if the given EventClock belongs to the CausalityClock. The *merge* joins the identifiers of both clocks keeping the maximum counter for each source. The *compareTo* operation compares two version vectors and determines if they are concurrent, equal or if one dominates the other.

We have implemented the following EventClocks: Timestamps and TripleTimestamps. A Timestamps is composed by a siteID and a counter of executed operations. A Triple Timestamps contains the same information of a Timestamp plus an extra counter. Both Timestamps and Triple Timestamps can be used with CausalityClocks.

We compare Triple Timestamps as follows: $(ca_1, cb_1, s_1) < (ca_2, cb_2, s_2)$, *iff* $ca_1 < ca_2 \vee ca_1 = ca_2 \wedge s_1 < s_2$ or $cb_2 = cb_2 \wedge s_1 = s_2 \wedge cb_1 < cb_2$. Comparing a TripleTismestamp with a Timestamp is a special case, where we can see the Timestamp as a TripleTimestamp with the cb counter having the maximum possible value.

5.1.2 Utility classes

5.1.2.1 SwiftConfigurations

The programmer can edit the configuration file to specify the default serializer class, Riak replication parameters and node access points. The transactional SwiftCloud configuration object also has auxiliary parameter to specify the address of the transaction server. SwiftConfigurations class handles every configuration on the system and provides methods to read any of these parameters. It also manages the connections to the Riak cluster, providing means to get connections to a set of specified Riak nodes.

Listing 5.1: CausalityClock Interface

```
1  public interface CausalityClock extends Serializable {
2      /**
3       * Records the next event.
4       * @param ec Event clock.
5       */
6      void record(EventClock ec) throws IncompatibleTypeException,
7              InvalidParameterException;
8      /**
9       * Checks if a given event clock is reflected in this clock
10      * @param c Event clock.
11      * @return Returns true if the given event clock is included in this
            causality clock.
12      */
13     boolean includes(EventClock c) throws IncompatibleTypeException;
14     /**
15      * Compares two causality clock.
16      * @param c Clock to comapre to
17      * @return Returns one of the following:
18      *         CMP_EQUALS : if clocks are equal;
19      *         CMP_DOMINATES : if this clock dominates the given c clock;
20      *         CMP_ISDOMINATED : if this clock is dominated by the given c
            clock;
21      *         CMP_CONCUREENT : if this clock and the given c clock are
            concurrent;
22      */
23     int compareTo(CausalityClock c) throws IncompatibleTypeException;
24     /**
25      * Merge this clock with the given c clock.
26      * @param c Clock to merge to
27      */
28     int merge(CausalityClock c) throws IncompatibleTypeException;
```

5.1.2.2 Runtime

CausalityClocks register events that are generated with the unique identifier of some site (e.g. clients, servers, replicas, etc.) which are uniquely identified. For this reason, it is necessary to guarantee that correct unique identifiers are stored in the CRDTs. To this end, the Runtime class encapsulates the Causality clock for a given site and must be used to generate new unique identifiers. CRDT implementations use the Runtime in the non-transactional SwiftCloud implementation. In the transactional version, operation identifiers are generated within the TransactionHandler.

5.2 SwiftCloud Transactional

In this section we will explain the transactional version of SwiftCloud. This version supports transactions as explained in 4.6. The system is composed by a set of components that interact to deliver the transactional support. First, we will start by explaining how we archive isolation using Versioned CRDTs and the information that the CausalityClocks encode. Then we explain the interaction protocol between clients and the transactions servers. Finally we explain the different components implementations.

5.2.1 Transactions Server

As explained in section 4.6, TxServer maintains the version of the database and the state of ongoing transactions. The Server can rollback transactions when a client fails.

When a client starts a transaction, the TxServer generates a new event, which is the transaction identifier and records the status of the transaction.

It is necessary to keep track of ongoing transactions, otherwise we could not detect client failures. To detect those situations, TxServers set a timeout for a transaction to execute and an auxiliary thread aborts transactions whose timeout has expired. However, a transaction duration is uncertain, and the client can always renew the granted time to complete the transaction.

There can be multiple TxServer instances running on a cluster in order to provide better availability and response time for handling transactions. TxServers communicate between each other to synchronize their CaulsaityClocks.

It would be possible to implement a crash-recovery solution for TxServer. Our implementation only uses soft-state, and assumes that a TxServer will not recover after failing.

5.2.2 SwifClientTransactional and TransactionHandler

SwifClientTransactional is a different version of the client that enforces the programmer to request and modify objects within transactions. As showed in listing 4.5, the interface has *begin* and *commit* operations and the remaining are similar to the original client with minor adjustments. The *begin* method returns a TransactionHandler; *fetch*, *store* and *createObject* operations require

a TransactionHandler to execute. The *getKeys* method does not guarantee isolation, i.e., the operation may return keys from objects that were created after the database state that the transaction is accessing.

In our implementation, the SwiftClientWithMetaTransactional class implements the SwiftClientTransactional interface. This implementation relies on the TxServer to handle transactions, as explained in section 4.6.

The TransactionHandler maintains the state associated with a transaction. This object contains the database version to access the EventClock that identifies the transaction and the timeout of that transaction. While the transaction is undergoing, TransactionHandler sends messages to the TxServer to renew the timeout for that transaction and verify that the TxServer has not failed. If during this, or any other step of the transaction, the TransactionHandler receives an answer informing that the transaction was aborted, it will undo any work done and inform the application that the transaction has aborted.

VersionedCRDT operations must explicitly call the TransactionHandler to generate new identifiers. Whenever an identifier is generated at the TransactionHandler, the object that issued that operation is logged for update. This means that, when the application tries to commit a transaction, the log of operations is processed to store the objects in the database.

The TransactionHandler also includes the code to commit transactions. When the programmer issues a *commit* operation, the TransactionHandler executes the protocol described in 4.6.3.1

5.3 CRDT Serialization

So far, we have seen how to fetch and store objects in the Key-Value store and the interface of different CRDT types. However, since Riak only stores arrays of bytes, it is necessary to define how to serialize CRDTs to store them. In SwiftCloud we serialize/deserialize objects to/from byte representation when interacting with the Riak Key-Value store.

We start by introducing the serialization interface. Then, in the following sub-sections, we present two different implementations of that interface: One based on Java and the other on JSON [Cro06].

5.3.1 Serialization Interface

The design of the interface is simple and very generalist. The interface has operations to serialize and deserialize CRDTs and clocks. It is necessary to have different methods to serialize CRDTs and clocks because, in Riak, clocks are stored in the Meta-data of the objects and these must be an alphanumeric array of characters. In listing 5.2, we show the concrete serialization interface of SwiftCloud prototype.

Serialization libraries are very distinct and some may require the user to specify the types of objects, while others can infer the types automatically. To deliver an uniform interface, the programmer must always supply the type of the objects.

The interface uses the TypeToken interface to store the type of an object. Besides allowing

Listing 5.2: CvRDTSerializer Interface

```
1  public interface CvRDTSerializer {
2    /*
3    * Creates a CRDT from its byte representation.
4    * @param bytes - The serialized CRDT
5    * @param type - The type of the stored object
6    */
7    <T extends CvRDT> T deserializeCRDT(byte[] bytes, TypeToken<T> type) throws
         CvRDTSerializationException;
8    /*
9    * Serializes a byte array into its byte representation.
10   * @param obj - the CRDT to serialize
11   * @param type - The type of the object
12   */
13   <T extends CvRDT> byte[] serializeCRDT(T obj, TypeToken<T> type) throws
         CvRDTSerializationException;
14   /*
15   * Transforms a CausalityClock to a String representation
16   * @param clock - The clock
17   */
18   String clockToString(CausalityClock clock) throws CvRDTSerializationException
         ;
19   /*
20   * Retrieves a CausalityClock from a String
21   * @param string - The string representing the CausalityClock
22   * @param type - The type of the CausalityClock
23   */
24   CausalityClock clockFromString(String string, Type type) throws
         CvRDTSerializationException;
25 }
```

the retrieval of an object type, TypeToken enforces the serialized/deserialized objects to conform to a specific CvRDT Type statically. TypeTokens do not make deserialization type safe: if on deserialization, the user specifies a type different from the object's type, an exception is thrown. Listing 5.2 shows an example of serializing a Shopping Cart using the TypeToken.

5.3.2 SwiftSerializerJAVA

The first, and more simple serializer is based on the *Java Serialization API*. To make CRDTs work with this API, the CRDT interface extends *Serializable*. As Serializable is natively supported in Java, no further change is required on CRDT implementations, besides requiring that all internal objects are also serializable.

The Java Serialization support is specific to the Java platform and many standard Java Library structures already implement it, which make it very comfortable for programmers to use. Objects serialized with this method cannot be used by non Java clients that may access the Riak key-value store, reducing the inter-operability of the system.

Listing 5.3: CRDT Serialization example

```
1
2     try {
3       TypeToken<ShoppingCart> shoppingCartToken = new TypeToken<ShoppingCart>()
            {};
4       CvRDTSerializer serializer = ...;
5       byte[] shoppingCartByte = ...;
6       ShoppingCart cart = serializer.deserializeCRDT(shoppingCartByte,
7           shoppingCartToken);
8     } catch (CvRDTSerializationException e) {
9       ...
10    }
```

5.3.3 SwiftSerializerJSON

JSON [Cro06] is a lightweight format to represent objects used extensively in the web, where it is becoming more popular as a solution to transfer data. JSON is text-based and it requires parsers to serialize/deserialize objects in different languages. It can be used between different platforms as long as every platform has a parser to process stored objects. This solution promotes interoperability, but the text representation can possibly incur in extra overhead.

5.3.3.1 GSon Serializer

Among the several Java JSON libraries[1], we have selected the GSON library. We chose this library because it required no modification in the existing CRDTs implementation. Additionally, it had support to override the serialization/deserialization process of specific classes, which was useful to fix some limitations and extend the functionalities of the library. If we want to override serializion/deserializion we must install the handlers before starting to use the parser, as the library does not allow to register them on the fly.

The library transforms the object's fields into JSON data types (primitives, objects, arrays, etc.). To serialize maps, the serializer creates JSON associative arrays, which uses strings to index positions.

The serialized objects do not contain any information regarding the data types they encode. Thus, the user must specify the type on the deserialization process. However, during deserialization, if we are using classes with generic types, the GSON library cannot infer the specific type of the object due to Java type erasure. To tackle this limitation, the library provides the TypeToken class to keep track of the generic types used.

5.3.3.2 Storing Object types

When objects are stored in SwiftCloud, they are stored with meta-data that indicates the class of the stored object. This can be used to deserialize objects without requiring the application to

[1] http://jackson.codehaus.org/; http://code.google.com/p/google-gson/; https://github.com/douglascrockford/JSON-java

specify their type. However this method do not work for all situations and we had to add some alternatives to serialize/deserialize objects.

For example, the library cannot retrieve objects by their interface. We have to specify the concrete class of the object in the TypeToken. For example, if we want to retrieve a *List*, we cannot retrieve a *java.util.List* object, because the parser do not know which implementation is actually stored. Instead, we must indicate the actual class, for instance, a *java.util.LinkedList*.

We extended the JSON parser to include the class of the object when it is stored. To do it, we implemented a custom serializer and deserializer that is used when the programmer stores a CRDT specifying its interface. In this case, the custom serializer, instead of serializing the object, creates a JSON array *(object, type)*, where *type* indicates the class of the stored object and *object* represents the actual value. If the stored object contains any generic field, it will also be stored as an *(object, type)* array. During deserialization, the same interfaces have a custom deserialization procedure: the deserializer expects an object (with a specific type), but receives a JSON Array containing *(object, type)*. The deserializer simple deserializes the object using the *type* class and delivers it. This way, an object that implements the expected type is created and no exception is thrown.

5.3.3.3 CRDT Composition

For supporting CRDT composition, we added two new methods to the JSON serializer to intersect the serialization/deserialization of ReferenceCRDTs. When serializing a CRDT with reference values, the JSON serializer overloads the default serialization of ReferenceCRDT type and stores a JSON array containing the location information (bucket and key) instead of the value, the referenced CRDT is also stored independently in the given location. For deserialization, we have implemented two strategies.

We provide an interface to implement composite CRDTs, the CvRDTReferenceable interface that is shown in listing 5.4. This interface is responsible for specifying the key of each referenced value. If the field of an object is a ReferenceCRDT, it is possible to simply infer the key from the annotation. However, if the composite CRDT has a field whose values are References to CRDTs, it must provide the rules to determine the key for each object. In listing 5.5, we show the implementation of the Referenceable interface for the OR-Map. The implementation checks if the values of the OR-Map are ReferenceCRDTs, then it sets the key of each object by using the key of the value in the OR-Map. Furthermore, if the values of the OR-Map are not ReferenceCRDTs, but they are Referenceable, the execution continues recursively.

To provide the eager and lazy reference fetching strategies, we have two different implementations of the ReferenceCRDT interface, the EagerReferenceCRDT and the LazyReferenceCRDT. The programmer can specify which one is used in the configurations file. Stored references can be fetched using any strategy, regardless of the method that was used to store them.

Listing 5.4: CRDT Referenceable interface

```
public interface CvRDTReferenceable {
    CvRDTReferenceOverBucket processReferenceValues();
}

public interface CvRDTReferenceOverBucket {
  void storeReferences(final String bucket, final String key);
}
```

Listing 5.5: ORMap CRDT Referenceable interface implementation

```
    public CvRDTReferenceOverBucket processReferenceValues() {
        return new CvRDTReferenceOverBucket() {

            public void storeReferences(final String bucket, final String key)
                {
                for (Entry<K, Set<Pair<V, EventClock>>> e : elems.entrySet()) {
                    for (Pair<V, EventClock> pair : e.getValue()) {
                        if (pair.getFirst() instanceof CvRDTReferenceable) {
                            CvRDTReferenceable ref = ((CvRDTReferenceable) pair
                                    .getFirst());
                            CvRDTReferenceOverBucket handler = ref.
                                processReferenceValues();
                            if (handler != null) {
                               handler.storeReferences(bucket,
                                        key + e.getKey() );
                            }
                        } else {
                            break;
                        }}}}};
    }
```

6

Evaluation

In this section we present the evaluation of CRDTs base performance and SwiftCloud system. We have evaluated specific parts using Micro-Benchmarks. The overall system performance was evaluated relying on a standard Macro-Benchmark, TPC-w. We have also used this benchmark to do a qualitative evaluation of the difficulty of adapting applications to use our system.

6.1 Micro-Benchmarks

The purpose of the Micro-Benchmark evaluation is to analyse specific parts of the developed system. In our work we want to evaluate the performance of CRDTs, as well as the overhead of handling CRDTs in our client. To analyse CRDTs performance, we compare a CRDT implementation of the Set data type with a Java implementation. Then we compare the performance of our client, using the JSON serializer with and without CRDT composition, to the default Java Serializer.

6.1.1 Micro-Benchmark description

The Micro-Benchmark simply executes operations over a CRDT, which can be maintained only in memory or stored in SwiftCloud after each operation. The benchmark executes a variable number of threads for a certain amount of time. Each thread executes a sequence of operations in the same object.

The benchmark allows to use Set or Map data types and, for each type, test different implementations. The user can specify which operations of the CRDT are executed and their frequency. For the OR-Map, the available operations are *put*, *get*, *contains* and *remove*. The ratio of *put*/*remove* operations is always equally distributed, to avoid making the map grow or become too small. The

map keys are the *hash* of the values. For the OR-Set the available operations are *add*, *remove* and *contains*. We also set equal *add* and *remove* ratios to maintain a similar size of the Set.

The data structures store character strings. The domain of elements, i.e. the set of different string that can be used in operations, can be parametrized in size and length. Parametrizing the number of different elements that can be added is necessary to control the amount of hits on query/update operations. Increasing the size of objects potentially increases the communication latency to retrieve an object. These values are computed before the benchmark begin.

We can configure SwiftCloud to use any serializer that implements the CvRDTSerializer interface. This allows us to compare the performance of Serializers. Our evaluation compares the two implementations in the prototype: *SwiftSerializerJAVA* and *SwiftSerializerJSON*. We also evaluated the CRDT composition with lazy and eager evaluation, in the SwiftSerializerJSON.

6.1.2 CRDT serialization

The retrieval of an object from the Riak Key-Value store is divided in three phases: fetch the object from Riak, process the retrieved bytes and merge the CRDT versions, if needed, returning the result to the application. The way bytes are stored depends on the serializer we use. The Java serializer is a good approach if we are interested in implementing Java applications, as it allows to use the Java library and all its functionalities seamlessly. The JSON serializer will be more adequate if we are using the Key-Value store in an heterogeneous environment, since the values are stored as strings and can be processed by any JSON library.

We ran the Micro-Benchmark executing one thread that accesses an OR-Map in the SwiftCloud storage and evaluate the operations latency with the two serializers that we have implemented, the SwiftSerializerJava and SwiftSerializerJSON. We test our system with domain sizes of 10, 100 and 1000 different elements. The initial size of the map is 5, 50 and 500 elements, respectively for each domain size, which remains roughly constant over the time of the experiment. The update ratio of the benchmark is 0.2 and we used elements of 128, 1024 and 4096 bytes.

To test the CRDT composition, we store each value of the map in a reference, using LWW-Registers. We evaluated both eager and lazy composition methods, we refer both serialization methods as JREF (Eager) and JLAZYREF (Lazy) respectively.

We ran our tests in a Intel Core Duo Machine @ 2.53 Ghz with 4Gb of RAM and 3M cache. The Machine was running Ubuntu 10-11, Riak enterprise 1.0 and OpenJDK 1.6.0_23. The tests ran in the same machine as Riak to avoid latency time. The Riak ring was composed by only one node, hence the read/write *Quorum* has the size one. Each test runs during 20 seconds and the results reported are the average of five runs.

We start by presenting the results with CRDT size small. In this case, the fetch and deserialization time of objects is expected to be relatively low. Figure 6.1 shows that retrieving an object using the Java Serializable is generally faster when compared with the simple JSON serializer. Also, we see that the fetch/storage time varies with the size of the object.

The figure also shows that storing small objects in different keys has a large overhead if we retrieve all of them during object deserialization due to the need of additional calls to the system.

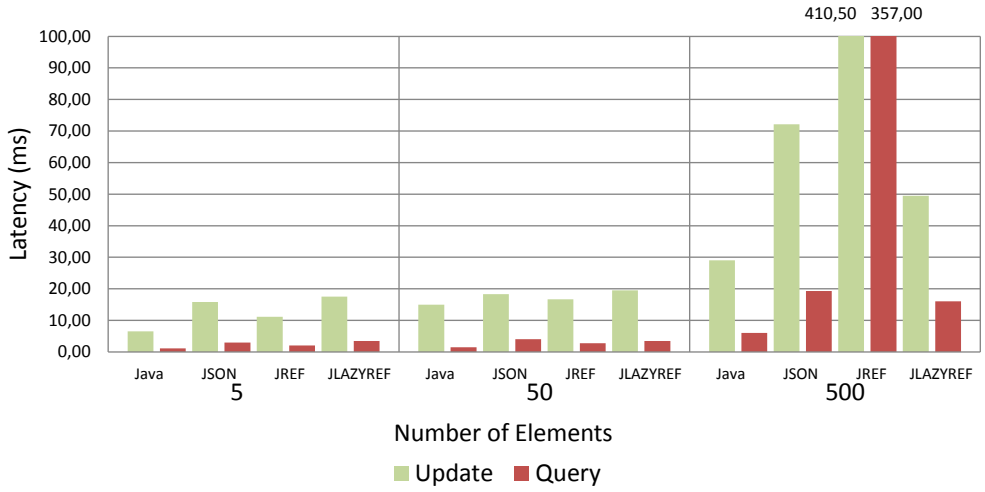


Figure 6.1: Micro-Benchmark latency time for 128 bytes objects.

However, the lazy JSON approach, which retrieves objects only when they are accessed, shown a slight performance improvement when comparing to the simple JSON serializer.

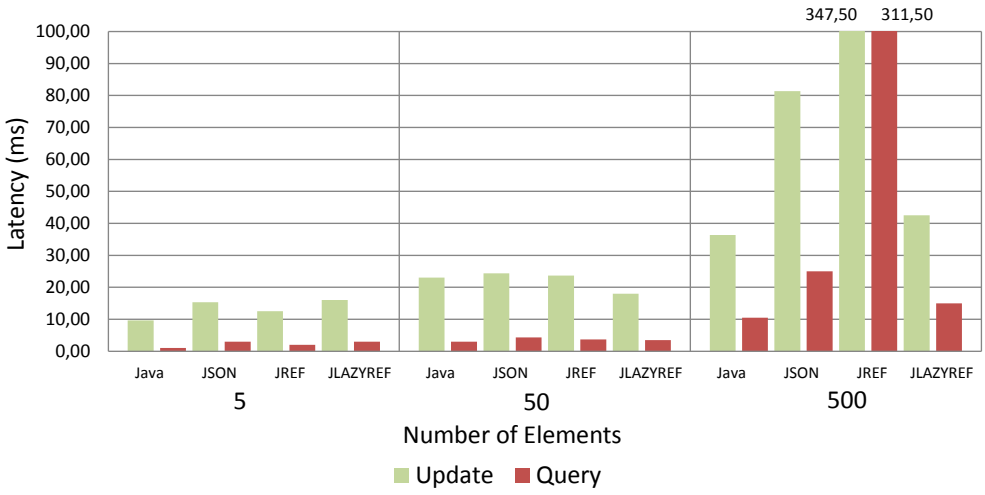


Figure 6.2: Micro-Benchmark latency time for 1024 bytes objects.

Figure 6.2 presents the result for objects of 1024 bytes. The results show a performance improvement of about 35% when updating the CRDT using the lazy strategy for CRDTs with 50 elements, compared to the simple JSON serializer. The improvement increases with the size of the CRDT, being twice as fast when the object had 500 elements.

Furthermore, comparing the *update* and *query* operations, we see that the serialization time of objects is minimal comparing to the time it takes to store an object in the key-value store, because the update operations have a significant higher latency.

The trend observed continues, when the size of objects increases. Figure 6.3 shows the results for objects of size 4096 bytes. In this case, using references with lazy load becomes the best

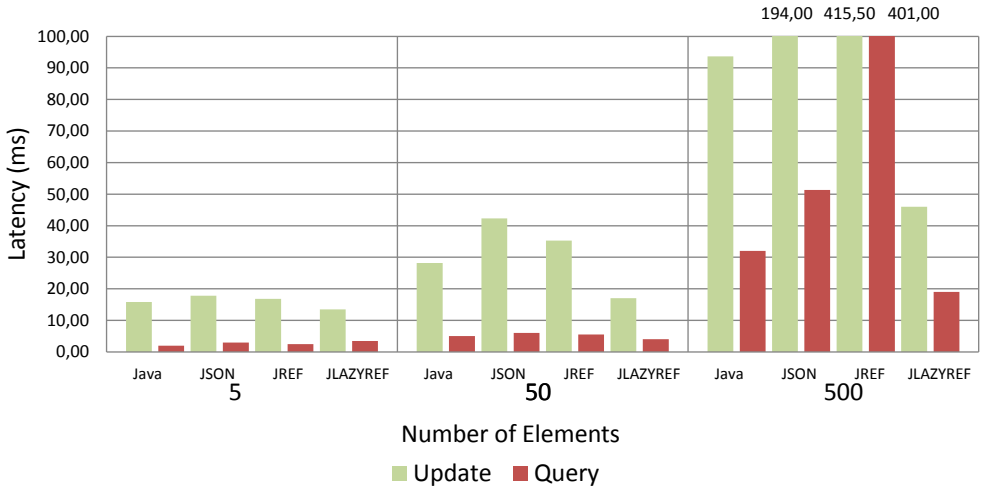


Figure 6.3: Micro-Benchmark latency time for 4096 bytes objects.

approach as less information is read/write and serialized/deserialized. Overall, the experiments show that the JSON serializer is slower when compared to the Java Serializer, which is expected as JSON is stored in less efficient text format, which must be parsed to retrieve the original values. Also, the results show good performance for the lazy CRDT composition, when the number and size of stored elements grows. The trade-offs were not addressed in this work. For a more efficient solution we could build it based on work performed in object-oriented databases - e.g. [LAC+96, CDF+94, Tsa92].

The implementation of a custom Java serializer, to store composite CRDTs, could deliver the performance of the Java Serializer and the benefits of accessing objects' values on demand.

6.1.3 CRDT performance

CRDTs deliver automatic conflict resolution at the cost of more complex implementations when compared with simple data structures. The exact overhead depends on the specific data type. To give an idea of the overhead we can incur when using CRDTs, we compare the performance of Set CRDT implementations with the plain Java HashSet implementation.

We use the Micro-Benchmark, executing operation over a Set object, using the HashSet, C-Set [AMSMW11] and OR-Set without tombstones (ORSet-NOTOMBS). Objects are stored in memory only, incurring in no overhead for communicating with the underlying storage.

The tests execute one thread during 20 seconds, 1000 elements of 128 bytes each. The tests were performed in a Sun Fire X4600 M2 x64 server running Linux, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 Ghz, 1M of cache in each processor and a total of 32 GByte of RAM. The Java version used was OpenJDK_1.6.0_18.

Figure 6.4 presents the throughput with variable ratio of query/update operations. Throughput is measured as the total operations (add, remove and contains) were executed per time unit. As expected the results show that the more complex CRDT designs perform worse than Java HashSet. For our OR-Set without tombstones, performance degrade up to less than 20%, increasing with

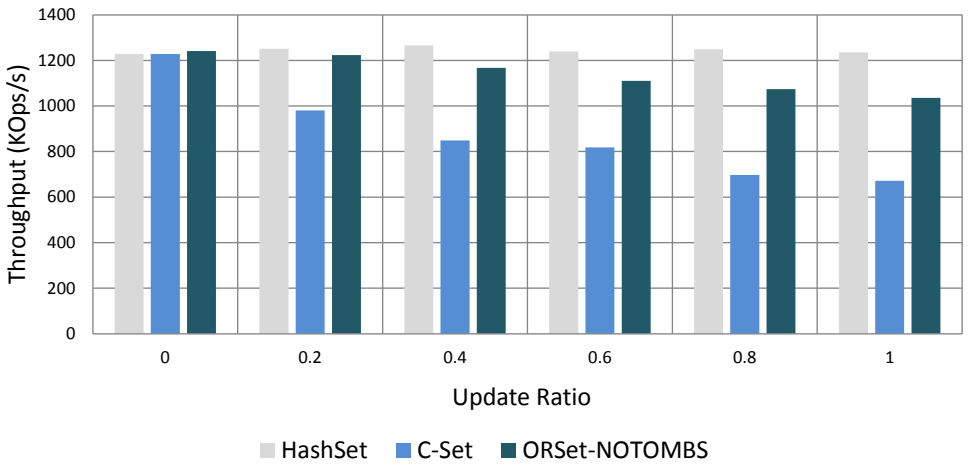


Figure 6.4: Java HashSet and CRDT Set throughput comparison with different update ratios.

the ratio of updates. The reason for the increase degradation is due to the additional complexity in update operations. For C-Set, the performance degrades further. The main reason for this is that our implementation does not remove tuples when *add* and *remove* counters are equal, as suggested by ASlan et. al. [AMSMW11]. This could improve the performance on this situation.

6.2 Macro-Benchmark

To evaluate the system with a real world application, we used TPC-W Benchmark [GG03]. The TPC-W Benchmark simulates an e-commerce platform that allows registering users, browsing items, adding elements to a shopping cart and placing orders. This benchmark is frequently used to evaluate the performance of relational DBMSs [PA04, MMA06, Cam07] and there are some public domain implementations.

In our work, we ported a TPC-W open-source Cassandra implementation[1] to Riak, using the Riak-Java-Client presented in section 4.1.1. Afterwards, we implemented the SwiftCloud version. This allowed us not only to evaluate the performance of the system, but also the complexity of using SwiftCloud and porting an application designed for a key-value store to a key-CRDT store.

In this section, we will do a brief overview of the application and briefly explain the Riak implementation and how it was ported to SwiftCloud, focusing on the adaptations in the code and used/implemented CRDTs.

6.2.1 TPC-W

TPC-W simulates an e-commerce platform. The original TPC-W includes a set of operations that simulate the user's interactions through a web application with a graphical interface. In this

[1] https://github.com/PedroGomes/TPCw-benchmark. We have contributed with our Riak implementation, which is now available as part of the project.

Operation	**Parameters**	Description
PRODUCT DETAIL	item_id	retrieves information about an item with item_id
HOME	item_id, custumer_id	retrieves information about an item with item_id and customer with customer_id
SHOPPING CART	item_id, cart_id, CREATE	adds a new item, with item_id, to an existing shopping cart with cart_id, or a new one if CREATE is set to true.
SHOPPING CART	item_id, qty	adds quantity qty of item_id items to the shopping cart
BUY REQUEST	cart_id	computes the total cost of a shopping cart and the billing information
BUY CONFIRM	customer_id, cart_id	Creates a new order and a new payment for a shopping cart that was previously processed in BUY REQUEST
ORDER INQUIRY	order_id	checks the status of an order
BEST SELLER		Computes the Best Seller information for each category of items
ADMIN ACTION	item_id	Adds the item with item_id to its subject index, adds the five most sold items to the related
CUSTOMER REGISTRATION	customer_id	Registers a new customer

Table 6.1: Description of TPC-W operations

implementation, we only simulate the data access executed from the application. The implemented operations are described in table 6.1.

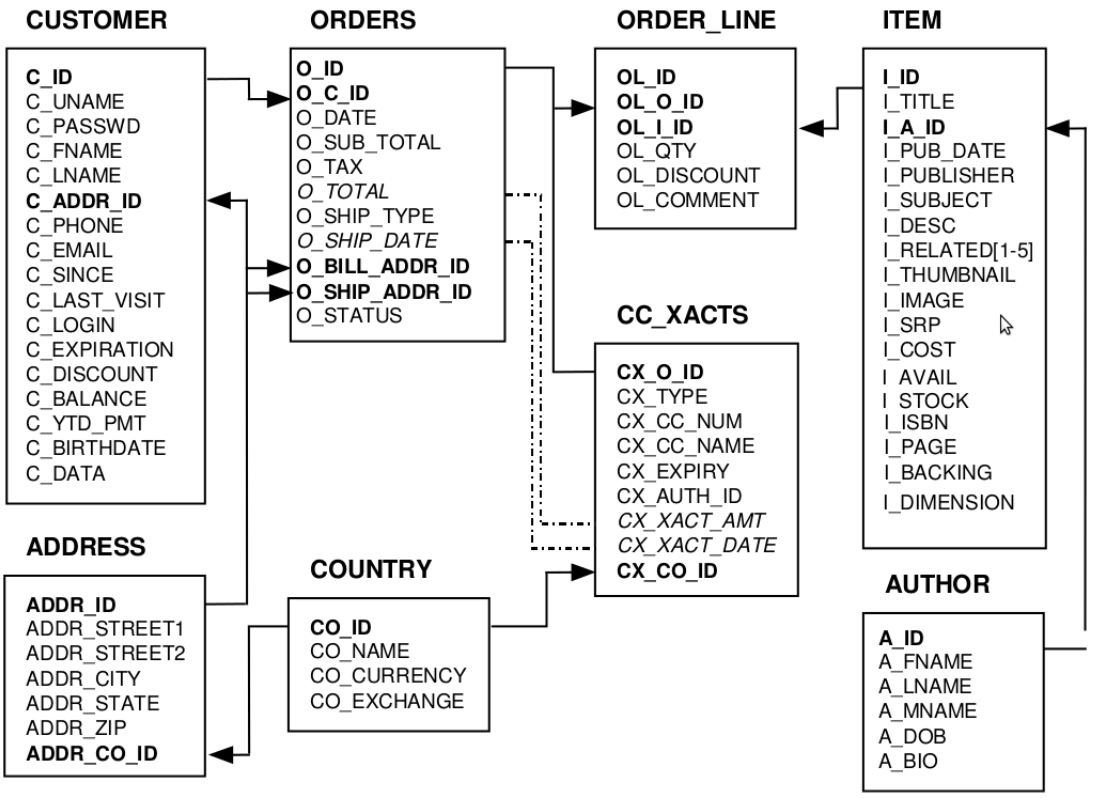
Most of these operations only need access to the primary key of objects and we can do that with good performance on a key-value store, as we can store data indexed by key. However, some operations require more complex queries that would benefit from the use of secondary indexes and other mechanisms traditionally present in DBMSs.

The traditional TPC-W data-model is for a relational database and it specifies the minimum tables the system should implement. We show the database schema in figure 6.5. The table *Order* registers the clients orders, *order_line* the items from a particular order and *cc_xacts* represents the payment of an order. The other tables store information for the customers, addresses, countries, items and authors. The MySQL and Cassandra implementations, that we have used, have additional tables for storing the shopping cart and the associated items in separate tables.

The simplified data-model of a Key-Value Store creates some challenges to develop efficient queries in the database. We address this issue in section 6.3.

6.2.1.1 Workloads

There are different benchmark workloads that intend to simulate different usage patterns of the system. The workloads vary on the amount of read and write operations, which is one of the



Legend:

- Dotted lines represent one-to-one relationships between non-key fields related through a business rule. These fields are shown in italic.
- The arrows point in the direction of one-to-many relationships between tables.
- Bold types identify primary and foreign keys.

Figure 6.5: TPC-W database schema

primary parameters that influence performance. The shopping workload has 95% of read-only interactions, the browsing workload has 80% and the ordering workload has only 50% of read-only operations. In our tests we use the browsing and ordering workloads, as these are the workloads available in the benchmark implementation.

In the following sections we present the deployments and testing configuration of our system. Finally we present and discuss the results gathered in the different configurations and workloads.

6.2.2 Benchmarks Configurations

We have used two setups for evaluating the performance of the system. In the first set-up, clients and servers run in a single data-center, emulating a web environment where the clients of the storage system run in the same data-center as the storage nodes. We measure the throughput and latency of operations with an increasing number of clients. We compare three solutions: Riak, non-transactional SwiftCloud and transactional SwiftCloud with the browsing and the ordering workloads. We use the JSON Serializer in all tests.

Riak is our baseline for comparison as it presents the best solution (SwiftCloud also requires Riak access). The others allow to measure the overhead of using CRDTs instead of simple Key-Value stores and the cost of transactions.

In the second set-up, we compare Riak and non-transactional SwiftCloud in an environment with clients running in two different data-centers. This allows to measure the performance improvement of placing an always accessible replica close to the client. In the Riak version, half of the clients must do an inter-data center access to contact the database. In the SwiftCloud version, each data-center has a SwiftCloud database and clients contact the database in the local data-center.

We rely on the Riak replication mechanism to synchronize the clusters. Since CRDTs guarantee that eventually all replicas converge to the same state in the future, we can execute operations in object stored in the different clusters regardless their version. In the Riak deployment, we do not use replicas in each data-center as it would generate conflicts whenever two clients concurrently access the same database elements. We cannot run the transactional version in a multi-data center environment, since, as we discussed in 4.6.2, our prototype does not implement transactions propagation across data-centers.

We used the Amazon Web Services EC2[2] commodity cluster to run the benchmarks. We deployed two Riak clusters composed by two nodes each, one in Europe and other in United States East Coast data-centers. The average latency between machines in the same data-center is 0.5ms and between two machines in different data-centers is 100ms. The machines we used were all running a custom 64bits Linux distribution with 7.5 GB memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB for storage and High I/O Performance. We have used Riak Enterprise edition 1.0.

Clients run in a number of dedicated machines, each with 20 clients. When using two data-centers, clients are equally distributed across data-centers. The machines we used were all running

[2]http://aws.amazon.com/ec2

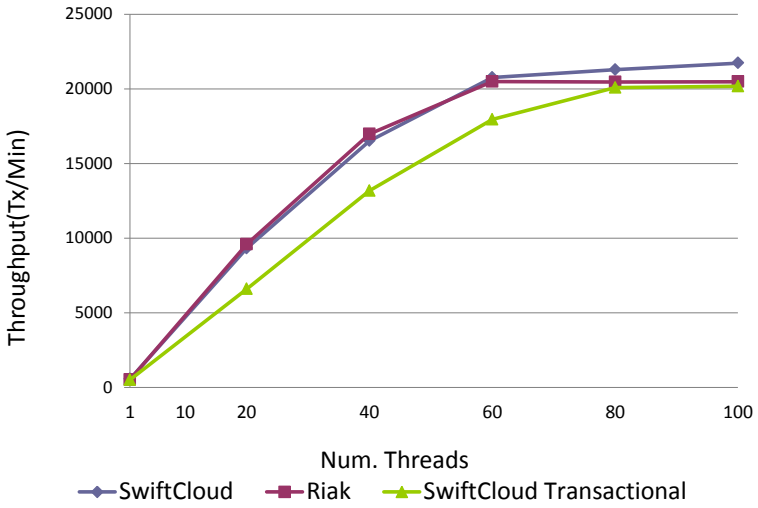


Figure 6.6: TPC-W throughput results with ordering workload.

a custom 64 bits Linux distribution with 3.75 GB memory, 2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit), 410 GB for storage and the I/O performance is classified as moderate.

The TPC-w database is initially populated with 10000 items, 28800 costumers and 25000 initial processed orders. Each experiment has a warm-up phase before starting the measurements. The presented values are the average of 2 runs. For the multi-data center benchmark, we do a full synchronization of the clusters before starting.

The replication factor is set to two, the read *Quorum* is one and the write *Quorum* is two to assure durability. We use this configuration because we have a cluster composed just by two nodes.

6.2.3 Single data-center

In this section we present the results for a single data-center. We start by presenting the results for the ordering workload. Figure 6.6 present the results for the throughput. The results show that the performance scales linearly until 60 clients and 20000 transactions per minute, for the non-transactional SwiftCloud and Riak versions. The transactional version reached the throughput limit later. The reason for a lower throughput is related to the extra communication step in the processing of transactions and the overhead of Versioned CRDTs.

Figure 6.7 shows the latency of operations. The latency increases with the load of the servers, as usual. We see that transactions impose some overhead and that it becomes less important as the load increases.

The throughput and latency for the browsing workload are shown in figure 6.8 and 6.9. In this workload the database becomes saturated more quickly. This limitation occurs because this workload has operations, such as the *admin-change* and the *BestSellers* that require iterating the bucket keys, which is an expensive operation in Riak. The bottleneck of the system seems to be related with the access to the storage. This leads to a lower throughput and higher latency.

In this workload, the transactional SwiftCloud performance is similar to the other two implementations due to the bottleneck on the database. Riak performs slightly worse but the difference

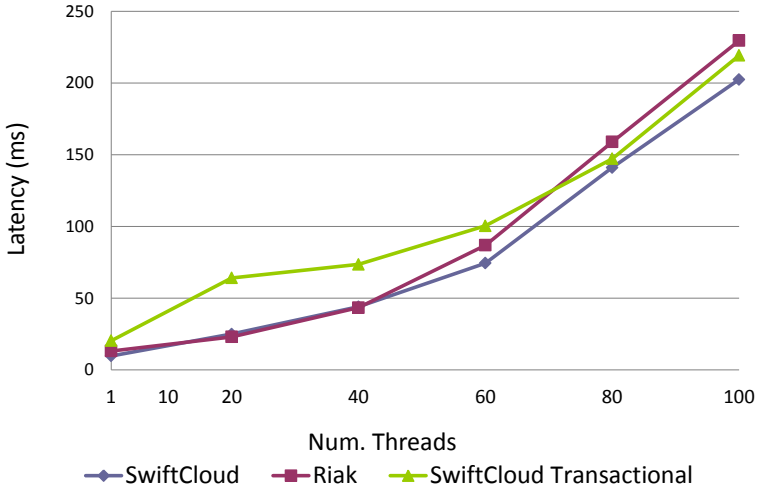


Figure 6.7: TPC-W operations latency with ordering workload.

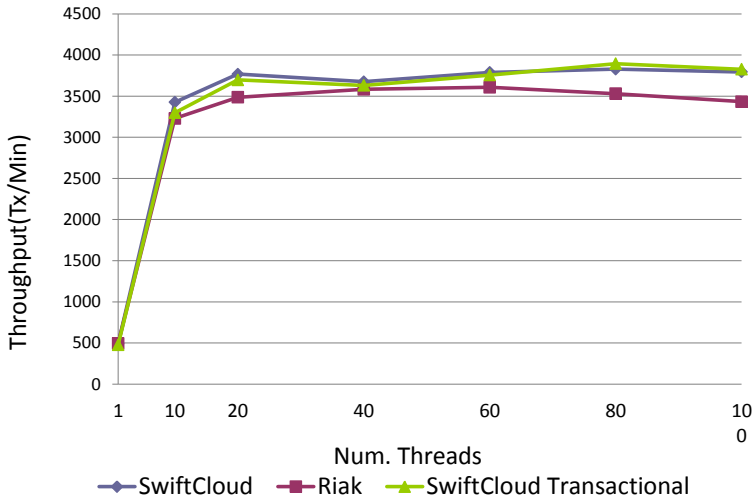


Figure 6.8: TPC-W throughput results with browsing workload.

is negligible.

The overall results show that SwiftCloud imposes a modest overhead with the benefit of providing strong eventual consistency with no lost of updates.

6.2.4 Multiple data-centers

The multi-data center experiments intend to analyse the effects of communication latency and verify the performance gains of having data replicated in a data-center close to the clients.

Figure 6.10 shows the throughput of clients in each data-center when using two data-centers for the ordering workload. We observe that SwiftCloud clients are able to produce almost twice as much operations than the Riak implementation. This is because half of Riak clients experience large latency to access data, contributing very little to the overall throughput. The SwiftCloud

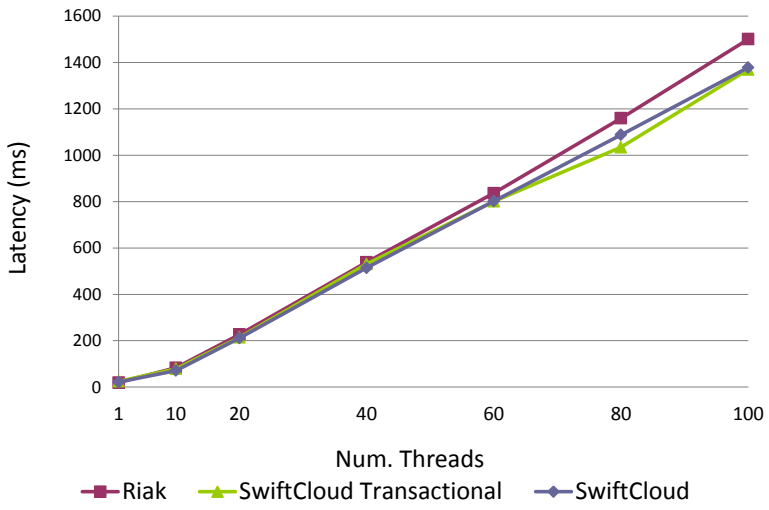


Figure 6.9: TPC-W operations latency with browsing workload.

European data-center performs less 11% operations when compared to the results shown in the previous section. The decrease in throughput is related to the cost of propagating updates among data-centers.

Figure 6.11 presents the latency of operations. We see that the latency of operations in Riak clients, when executing in a different data-center, is more than 10 times the latency of executing in the local data-center. This result is due to the communication latency.

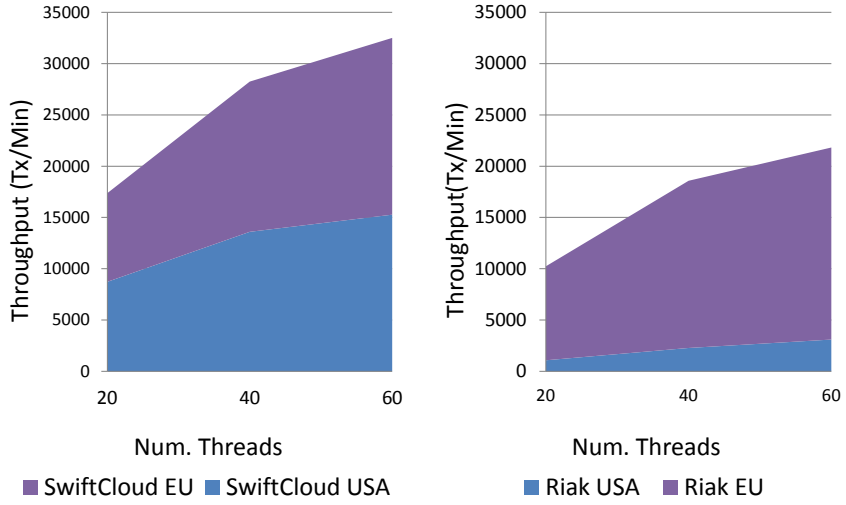
6.3 Implementation

In this section we describe the different implementations of the TPC-W benchmark. First, we describe the base Riak implementation, then we explain how we ported it to SwiftCloud and finally we explain the adaptations to use the transactional SwiftCloud.

6.3.1 Riak implementation

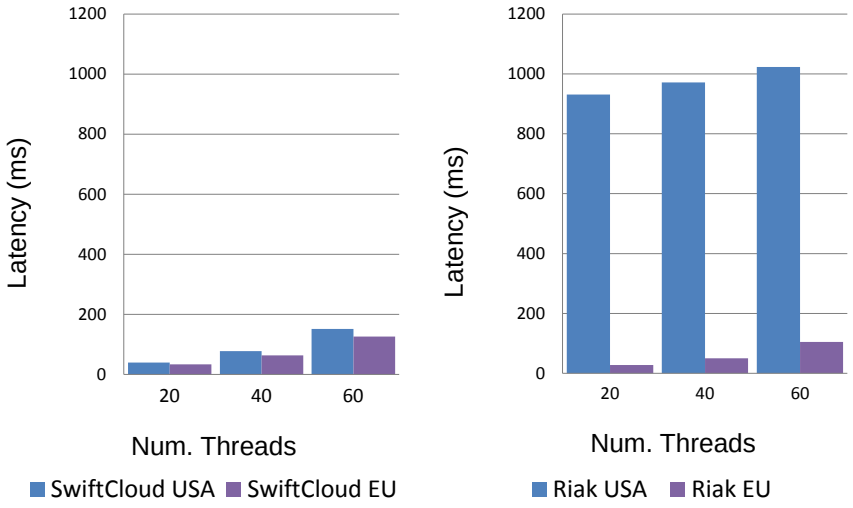
The Riak implementation is the base version of the benchmark for our comparisons. We implemented it by porting an existing Cassandra implementation. The Cassandra data-model is richer than the one provided by Riak, which complicated the process of porting the code. Cassandra has a special type of columns, called super-columns, that provide multiple attributes inside a column, while Riak has a flat data-model. In this section, we will overview how we implemented Riak TPC-W.

We stored the entities shown in figure 6.5 indexed by identifier in separate buckets. For each table, we have created a class to represent a a table row, adding fields for each attribute and storing the primary keys whenever an attribute is an external key. The Shopping Cart and the Order classes also have a map to store the items in the shopping cart and order, respectively. Most of that entities are accessed by primary key, leading to good performance. Comparing to the MySQL implementation, we store the shopping cart items in the value of the shopping cart itself, because



(a) SwiftCloud (b) Riak

Figure 6.10: TPC-W throughput with ordering workload on a multi-cluster deployment.



(a) SwiftCloud (b) Riak

Figure 6.11: TPC-W operations latency with ordering workload on a multi-cluster deployment.

Listing 6.1: Riak Java Client

```
1  public void addToCart(String cart_id, String item, int qty_to_add) throws
       Exception {
2    IRiakObject obj= rawClient.fetch("shopping_cart", cart_id).getRiakObject();
3    ShoppingCart sc= GSON.fromJson(obj.getValueAsString(),  ShoppingCart.class)
       ;
4    SCLine sc_line = sc.getSCLine(itemInt);
5    sc_line.setSCL_QTY(sc_line.getSCL_QTY() + qty_to_add);
6    obj.setValue(GSON.toJson(sc).toString());
7    rawClient.store(obj, writeQuorum);
8  }
```

the shopping cart and the items are often accessed together. Furthermore, if we store each item of the cart in a different key, we would have to fetch multiple keys to get the shopping cart, which would be very inefficient. The order table and the order lines use a similar approach.

Customers have a shopping cart per session. We cache the last shopping cart key in the customer value to have direct access to that shopping cart object. Another issue to retrieve data from the key-value store is when we have to do range queries. There is no support to fetch multiple values at once in Riak, which requires the values to be read one by one. We could not avoid this, but as range queries are an unusual operation, we have not implemented any workaround.

The Best Seller retrieves the most sold items for each category. The MySQL operation counts the number of sold items, filtering the category, and orders the result to identify the items that were most sold. This would be very expensive to execute in the Key-Value data-model. Our Riak/SwiftCloud implementation, to avoid counting the results for every element, stores the amount sold items whenever an item is sold. When the Best Sellers operation is executed, the list of sold items is processed and updates the index of the most sold items per category, if any of the sold items is more sold then any other item for its category.

We do not provide the search operation as it is not a feasible operation to execute in Key-Value stores due to the lack of secondary indexes or full-text search. Since this operation would have similar implementation on the systems that we are comparing, we considered that not having this operation would not be promoting the performance of a particular system.

As an illustration of the code, listing 6.1 shows the implementation of *addToCart* operation using the Riak Java Client interface. First we fetch the object from the Key-Value store. The IRiakObject contains the value stored in Riak (Using Last-Writer-Wins to store concurrent updates) which is deserialized with the class of the object. The object can be updated and stored again. It is important that the same IRiakObject is used to preserve the context of the read.

6.3.2 SwiftCloud implementation

We ported the Riak implementation to SwiftCloud and developed both transactional and non-transactional versions of the system. The refactoring of the code mostly regards the client calls and changing the object data types to CRDTs.

We used LWW-Registers to store *address*, *country*, *author*, *customers* and *cc_xacts* entities.

Listing 6.2: SwiftCloud code sample

```
1  public void addToCart(String cart, String item, int qty_to_add) throws
       Exception {
2    SwiftObjectWithMeta<ShoppingCart> obj = swiftClient.fetch("shopping_cart",
       cart,  shoppingCartTypeToken).getObject();
3    ShoppingCart sc = obj.getValue();
4    SCLine sc_line = sc.getSCLine(item);
5    sc_line.setSCL_QTY(sc_line.getSCL_QTY() + qty_to_add);
6    sc.addSCLine(sc_line);
7    obj.updated();
8    SwiftClient.store(obj);
9  }
```

Except for the *cc_xacts* and *customer* entities, all the other entities were read-only. The LWW-Register is a CRDT with low overhead and it is very suitable to objects that will not have concurrent updates.

We had to implement two new CRDTs to store the *Shopping_Cart* and the *Order* entities. We want to provide an always available shopping cart. In this context, concurrent updates to different replicas are allowed. We use an OR-Map to map item identifiers to order lines. The result of concurrent puts of different items will lead to a shopping cart with all items. For concurrent puts of the same item, the sum of the number of items is computed. The same applies for the order lines. We store the class fields in LWW-Registers as many of these fields are read-only, and others are updated by the interactions in the Cart/Order.

It is important that the available stock of an item is always consistent in the database, i.e., that we do not lose any update to the stock of an item. We used a CRDT counter to guarantee that property. However, with our operation semantics, it is possible that two concurrent updates lead the stock to a negative value.

The SwiftCloud implementation of *addToCart*, listing 6.2, is very similar to Riak, with the benefit that that operations provide automatic conflict resolution. SwiftObjects must be explicitly marked as updated before being stored.

6.3.3 SwiftCloud transactional implementation

To use SwiftCloud transactional, we had to substitute every CRDT implementation by Versioned CRDTs. This modification required to change the CRDTs that were implemented in the non-transactional SwiftCloud version to receive TransactionHandlers. The TransactionHandler is used to generate the EventClocks that identify the operations, to read a consistent version of the objects and mark the CRDTs that must be updated on commit.

The transactional version of SwiftCloud has differences regarding the interaction model with the client. In this version, programmers must *begin* and *commit* transactions. This interface, requires to change the client calls from the non-transactional version. Furthermore, interaction with CRDTs must be done through the TransactionHandler to *fetch* and *store* objects. With this abstraction, programmers no longer require to explicitly store the objects, as this is automatically managed by the handler.

Listing 6.3: SwiftCloud Transactional code sample

```
1  public void addToCart(String cart, String item, int qty_to_add) throws
       Exception {
2    TransactionHandler handler = SwiftClient.begin();
3    SwiftObjectWithMeta<ShoppingCart> obj = handler.fetch("shopping_cart", cart
         ,  shoppingCartTypeToken).getObject();
4    ShoppingCart sc = obj.getValue(handler);
5    SCLine sc_line = sc.getSCLine(item, handler);
6    sc_line.setSCL_QTY(sc_line.getSCL_QTY() + qty_to_add);
7    sc.addSCLine(sc_line, handler);
8    handler.commit();
9  }
```

The *addToCart* operation with the transactional version of SwiftCloud is shown in listing 6.3. Note that this version has no *store* or *update* operations, as that operations were hidden in the TransactionHandler.

6.3.4 Implementations Comparison

It was easy to modify an application using Riak client to use SwiftCloud. By design, the SwiftCloud interface is very similar to Riak's. SwiftCloud interface even hides some details, like the Read/Write *Quorum*, that can be configured and used by default. The task that requires more effort is to transform the application's objects data types to CRDTs. Thus, the existence of a library of CRDTs is fundamental. For those classes that are read-only or that do not require conflict resolution, we can simply use the LWW-Register CRDT which provides low overhead. Most other data types can be created by specializing existing CRDTs. In the shopping cart example, we just had to change the map implementation to enable automatic conflict resolution on the cart items and added all other fields to a LWW-Register. The shopping cart fields are shown in listing 4.4.

The transactional version of the application requires little changes to the code. However, this version requires the use of new Versioned CRDTs.

Having automatic conflict resolution is a huge benefit when compared with Riak. The main difference is that we do not need to handle conflicting version in the code of the application or risk loosing updates, as CRDTs guarantee that concurrent updates are merged automatically.

7

Conclusion

Cloud computing systems are used to deploy world scale services, often relying on Key-Value stores for storing data. Providing high availability and low latency for cloud applications demands the use of multiple data replicas, spread across the world. In such setting, it is hard to provide strong consistency of data. Thus, systems supporting geo-replication [LFKA11, SPAL11, CRS+08] often trade the consistency of data for high availability and low latency.

Transactions greatly simplify the programming effort by guaranteeing that an application does not see or exposes inconsistent states. Additionally, transactions are a well-known abstraction used pervasively by programmers when accessing data. Thus, supporting transactions is important for simplifying the creation of applications.

In this dissertation, we have presented the design and implementation of SwiftCloud, a key-CRDT store that supports a weak model of transactions. We implemented SwiftCloud on top of an existing Key-Value database, Riak. Developing our system as a middleware allows an easy deployment of the system, since it can run on top of an existing infra-structure.

Our system does not rely on a primary replica to execute update operations, as in many replication systems [CRS+08, PA04]. In SwiftCloud, clients are allowed to execute updates in any replica, which greatly reduces latency and contention for deployments in different geographical locations.

SwiftCloud provides a strong eventual consistency model, by relying on the specification of CRDTs. CRDTs allow multiple replicas to be updated without synchronization, providing an automatic reconciliation model that merges concurrent updates without rolling-back executed updates.

In this work, we have introduced Versioned-CRDTs, new CRDT designs that support multi-versioning with a small overhead. The insights of our solutions are the use of the unique identifiers created during the execution of operations to identify the versions and to add new tags to removed elements. Versioned-CRDTs are crucial to provide transactions in our system.

SwiftCloud provides support for transactions with the following properties. Inside a transaction, the application accesses a snapshot of the database, which includes all CRDTs in the system. All updates of a transactions are executed atomically. Transactions never abort - concurrent updates are merged using CRDT rules. The transactional system of SwiftCloud builds on the convergence properties of CRDTs and the multi-versioning support provided by the Versioned CRDTs.

When relying on Riak inter-cluster synchronization, our system does not provide transactions isolation across data-centers. However, we could achieve such property by coordinating the TxServers of different data-centers and their Riak replication agents.

SwiftCloud includes a simple mechanism for CRDT composition, where a CRDT may be composed of other CRDTs that are stored under different keys in the underlying Riak storage system. This functionality is implemented relying on a JSON serializer,

SwiftCloud provides a generic serialization mechanism that allows different serialization strategies to be used. We have implemented two base strategies: one based on Java serialization and the other in JSON serialization. While the former presents better performance, the second would allow access to CRDTs from application written in other languages.

Furthermore, we have implemented a second JSON serializer that allows to scatter composite CRDTs over different keys in the underlying storage system. This way, it is possible to retrieve a CRDT without fetching its complete content of the CRDT - referenced CRDTs are loaded as required. This provides a mechanism for simple composition of CRDTs.

To evaluate our system we implemented micro-benchmarks and ported an implementation of TPC-W to SwiftCloud. The TPC-W implementation allowed to compare our system with the original Riak implementation. Although the developed TPC-W solution could be improved to optimize the querying process on the underlying Key-Value store, the same approach is used for both Riak and SwiftCloud, leading to a fair comparison of results.

The evaluation of our system showed that the non-transactional SwiftCloud was able to provide automatic conflict resolution without significant impact on performance. In addition, SwiftCloud can improve performance of the system, by providing geo-replication while maintaining consistency. The transactional SwiftCloud imposes only a small overhead compared to SwiftCloud, which was a consequence of the extra communication steps in the protocol for executing transactions.

The evaluation of the serialization strategies suggests that our technique to scatter CRDTs over the database can substantially reduce the latency of fetch/store operations for large CRDTs, when a small number of objects are accessed. However, there are cases where this method may reduce performance. For instance, if a CRDT has many small elements that are accessed when a user fetches it, it may compensate to store all the elements alongside with the CRDT instead of retrieving each one of them on access.

We found it very simple to port the Riak TPC-W application to SwiftCloud: the client modifications were very simple, mainly because the SwiftCloud interface is inspired in the original Riak interface. To port an application, the main development effort would lie in developing CRDT data types to use in the application. We already provide a library of data types that can be reused.

Contributions: In summary, this work has contributed with:

- The design and implementation of the first key-CRDT store, which allows replicas to be updated without synchronization while providing strong eventual consistency;
- The introduction of Versioned-CRDT, a new design of CRDTs allowing to access a past version of the data. Versioned-CRDTs are a key element to the design of the transactional support of SwiftCloud and could also be used as the basis to replay the evolution of a CRDT;
- A transactional support in SwiftCloud, which provides atomicity of writes to several CRDTs and an isolation model inspired in snapshot isolation. The transactional support allows a transaction to access a snapshot of the data and, unlike snapshot isolation, transaction never abort as CRDT rules are used to merge updates on concurrent writes;
- A simple CRDT composition model, where a CRDT composed by multiple CRDT is stored in multiple keys in the underlying storage system. This composition solution includes a late-loading approach, that allows to reduce the latency of fetching a large CRDT;
- The implementation of a library of CRDTs, which is crucial to the development of application in SwiftCloud, including OR-Set, LWWRegister, MultiVersionRegister and the new OR-Map and State-based Treedoc.
- The port of an implementation of TPC-W to Riak and to SwiftCloud. The benchmarking tool allowed us to compare the performance of the different implementations, as well as the effort of porting an application form Riak to SwiftCloud.

7.1 Future Work

This work presents a first prototype of a key-CRDT store, allowing us to study features that are relevant toward the development of a more robust Key-Value Store system. The performance results support the desire to keep studying CRDTs and their integration in Cloud-Computing platforms. In the meantime, we already started the development of a key-CRDT store from scratch, which will use Versioned CRDTs in the data-model and provide a transactional support very similar to what we presented here.

Current CRDT designs are not suitable to some applications, as they cannot preserve data invariants, such as not allowing an integer to become negative. Despite that issue, for many large scale systems, CRDTs are a suitable solution with low overhead. The research on CRDT data types can lead to the development of new CRDT types that can address the invariants problem.

State based replication can present a great overhead when copying large objects, as the payload of objects must be propagated for *store* or *fetch* operations. We added the composition of CRDTs which allowed to store the CRDT meta-data independently of the values, which already showed a great reduction in the message size, for large objects. However, if we have to do a complete iteration over the CRDT values, we would introduce a great overhead in communication to fetch every value. There already CRDT designs to address this issue [SPBZ11c]. They provide both state

and operation based convergence. Unfortunately, in our middleware we cannot do server-side data manipulation, hence we are not able to implement these CRDTs. Even with our current CRDT designs, with server-side operations, we could do merges of conflicting values, which would reduce the overhead of sending and merging all the conflicting values in the client.

The Versioned CRDT designs can grow infinitely and, if an object is updated often, the size can considerably hurt the performance of the system. It is necessary to develop a strategy to prune the payload of the object, to remove versions that will no longer be requested. We must devise a strategy to prune the payload of the CRDT, hopefully without a consensus protocol to avoid synchronization. To do so, we can use one of the many consensus protocols available in the literature.

The presented system replicates the database at data-centers and the clients must always access the replicas to interact with, incurring in the negligible overhead of communication. As CRDTs provide convergence despite the updated replica, we can bring replication even closer to the client. Doing partial replication on the client devices can allow the development of system resilient to disconnection or replica faults. We intend to study the caching of data at the clients, study the replication scope, i.e. cache only my data, data that i frequently access, etc., and address the problems of security in this context.

Bibliography

[AMSMW11] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, and Stéphane Weiss. C-Set : a Commutative Replicated Data Type for Semantic Stores. In *RED: Fourth International Workshop on REsource Discovery*, Heraklion, Greece, May 2011.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[Cam07] Lásaro Camargos. Sprint: a middleware for high-performance transaction processing. In *In EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys Eu Conference on Computer Systems 2007*, pages 385–398. ACM, 2007.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 383–394. ACM Press, 1994.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *In proceedings of the 7th conference on usenix symposium on operating systems design and implementation - volume 7*, pages 205–218, 2006.

[CDK05] Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[Cro06] D. Crockford. The application/json media type for javascript object notation (json). Technical Report RFC 4627, 7 2006.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Technical report, In Proc. 34th VLDB, 2008.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[DLS+04] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *NSDI*, pages 85–98, 2004.

[EDP06] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. *SIGOPS Oper. Syst. Rev.*, 40(4):117–130, April 2006.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[EZP05] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[FGC+97a] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, October 1997.

[FGC+97b] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31:78–91, October 1997.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *K. Raymond (Ed.). Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[GG03] Daniel F. García and Javier García. Tpc-w e-commerce benchmark evaluation. *Computer*, 36(2):42–48, February 2003.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[GLR03] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *IN PROCEEDINGS OF THE HOTOS-IX 2003*, 2003.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[JLMS03] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal. In *Proceedings of the 2003 RSA conference on The cryptographers' track*, CT-RSA'03, pages 314–326, Berlin, Heidelberg, 2003. Springer-Verlag.

[KA00] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[Klo10] Rusty Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 210–218, New York, NY, USA, 2001. ACM.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, February 1992.

[LAC+96] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in thor. In H. V. Jagadish and Inderpal Singh

Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 318–329. ACM Press, 1996.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 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, pages 401–416, New York, NY, USA, 2011. ACM.

[Li09] Henry Li. *Introducing Windows Azure*. Apress, Berkely, CA, USA, 2009.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10:360–391, November 1992.

[LM09] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 198–208, New York, NY, USA, 2006. ACM.

[Nah04] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & IT*, 23(3):153–163, 2004.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[Ora99] Oracle. Oracle8i advanced replication: An oracle technical white paper. 1999.

[OUMI05a] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Research Report RR-5795, INRIA, 2005.

[OUMI05b] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Real time group editors without operational transformation. Technical Report RR-5580, INRIA, May 2005.

[PA04] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *In Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, pages 155–174, 2004.

[PMSL09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[Raz93] Yoav Raz. Extended commitment ordering, or guaranteeing global serializability by applying commitment order selectively to global transactions. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '93, pages 83–96, New York, NY, USA, 1993. ACM.

[RNRG96] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[SE98] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *CSCW*, pages 59–68, 1998.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5:63–108, March 1998.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7(3):149–174, March 1994.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.

[SPAL11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, 2011.

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. Technical report, 2011.

[SPBZ11c] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin / Heidelberg, 2011.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. pages 140–149, 1994.

[Tsa92] Manolis M. Tsangaris. *Principles of Static Clustering for Object Oriented Databases*. PhD thesis, 1992.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29:172–182, December 1995.

[WB84] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.

[X3.92] ANSI X3.135-1992. American national standard for information systems - databalse language - sql, November 1992.