David Pereira Alves Neves Lopes

Licenciatura em Engenharia Informática

Eventually Consistent Database Replication based on Conflict-Free Replicated Data Types

Relatório intermédio para obtenção do Grau de
Mestre em Engenharia Informática

Orientador: Rodrigo Rodrigues, Prof. Associado,
Universidade Nova de Lisboa
Co-orientador: João Leitão, Investigador Doutorado,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2015

Abstract

Today's internet services handle massive amounts of requests and data from users all over the world. To achieve the performance and responsiveness required by these services, providers rely on replication to allow the processing of concurrent updates, and execute under weak consistency models to avoid the costs of replica coordination.

Many of these services are supported by NoSQL databases, mostly because they provide better performance and scalability than relational databases. However, NoSQL databases are limited by the poor data model and query capabilities they provide, as well as the lack of a transactional model and integrity constraints. Hence, applications where these features are required must resort to relational databases, giving up the performance and scalability of NoSQL databases.

We propose a solution that tries to achieve a performance and scalability close to NoSQL databases, while supporting the compelling characteristics of relational databases, such as integrity constraints, transaction semantic, a rich data model and a powerful query language.

The aim of this dissertation is to develop a weakly consistent relational database. To increase both performance and scalability, we will explore Conflict-Free Replicated Data Types (CRDTs), a recent family of data types that enable replica convergence without the costs of coordination. However, avoiding coordination is at odds with preserving global invariants, because concurrent updates may violate these invariants. For this reason, operations that might break invariants must be coordinated. To this end, we will also develop a coordination service that will ensure invariants preservation.

To evaluate our proposed solution, we will deploy it in a *cluster* and run TPC-C and Rubis benchmarks.

Keywords: Replication; Weak consistency; Relational Database; CRDTs;

RESUMO

Os serviços de Internet disponíveis na atualidade lidam com enormes quantidades de pedidos e dados, provenientes de utilizadores espalhados por todo o mundo. Para atingir a *performance* e agilidade necessárias destes serviços, os provedores dos mesmos replicam-nos, permitindo o tratamento de pedidos concurrentes, e recorrem a modelos de consistência fraca, evitando assim os custos envolvidos na coordenação de réplicas.

Muitos destes serviços são suportados por bases de dados NoSQL, principalmente pela melhor performance e escalabilidade que oferecem quando comparadas com as bases de dados relacionais. No entanto, as bases de dados NoSQL estão limitadas pelo seu modelo de dados pouco expressivo e pobre linguagem de pesquisa sobre esses mesmos dados. Além disso, grande parte destas bases de dados não oferece nem o modelo transacional nem as restrições de integridade encontradas nas bases de dados relacional. Por estas razões, aplicações que necessitem de alguma destas características têm que optar por bases de dados relacionais, perdendo a performance e escalabilidade oferecida pelas bases de dados NoSQL.

Propomos uma solução que tenta conciliar a performance e escalabilidade das bases de dados NoSQL e, ao mesmo tempo, providenciar as características mais atrativas das bases de dados relacionais, como por exemplo o seu modelo de dados expressivo, restrições de integridade e uma poderosa linguagem de pesquisa.

O objectivo desta dissertação é desenvolver uma base de dados relacional com consistência fraca. Para aumentar a performance e escalabilidade, vamos explorer CRDTs (*Conflict-Free Replicated Data Type*), um tipo de dados que garante a convergência do estado de cada réplica sem ser necessário coordenação. No entanto, preservar invariantes sem coordenação é desafiante, uma vez que operações concorrentes podem quebrar estas invariantes. Por esta razão, operações que potencialmente quebrem alguma invariante terão de ser coordenadas. Para este efeito, iremos desenvolver um serviço de coordenação que será responsável por preservar as invariantes do sistema.

Para avalar a nossa solução, iremos testá-la num *cluster*, recorrendo aos *benchmarks* TPC-C e Rubis.

Palavras-chave: Base de dados relacionais; Replicação; Consistência Fraca; CRDTs

Contents

Contents vii

List of Figures ix

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Proposed Solution . . . 3
- 1.4 Document Organization . . . 3

2 Related Work 5
- 2.1 Overview . . . 5
- 2.2 Database Replication . . . 6
  - 2.2.1 System Model . . . 6
  - 2.2.2 Distributed Databases . . . 7
  - 2.2.3 Replication Protocols . . . 7
  - 2.2.4 Case Studies . . . 11
- 2.3 Weakly Consistent Systems . . . 12
  - 2.3.1 Eventual Consistency . . . 14
  - 2.3.2 Causal Consistency . . . 14
  - 2.3.3 Case Studies . . . 15
- 2.4 Conflict-Free Replicated Data Types . . . 16
  - 2.4.1 State-based CRDT . . . 17
  - 2.4.2 Operation-based CRDT . . . 18
  - 2.4.3 CRDTs Catalogue . . . 18

3 Proposed Work 23
- 3.1 Solution . . . 23
  - 3.1.1 Previous Work . . . 24
  - 3.1.2 Architecture . . . 24
  - 3.1.3 Evaluation . . . 24
- 3.2 Work Plan . . . 24

Bibliography 27

List of Figures

2.1 Eager Replication Protocol 8
2.2 Lazy Replication Protocol 9
2.3 Master-Slave Replication Protocol 10
2.4 Multi-Master Replication Protocol 10
2.5 Partition in a Distributed System 13

3.1 Proposed solution high level overview 25
3.2 Gantt chart 26

List of Algorithms

1 LWW Register (state-based) 19
2 LWW Register (operation-based) 19
3 G-Set (state-based) 20
4 2P-Set (state-based) 20
5 OR-Set (operation-based) 21

Acronyms

1SI One-Copy Snapshot Isolation.

2PC 2-Phase Commit.

CC Causal Consistency.

CmRDT Commutative Replicated Data Type.

CRDT Conflict-Free Replicated Data Type.

CvRDT Convergent Replicated Data Type.

EC Eventual Consistency.

LWW Last-Writer-Wins.

RC Read Committed.

RR Repeatable Read.

SEC Strong Eventual Consistency.

SI Snapshot Isolation.

SPOF Single Point of Failure.

VC Vector Clock.

CHAPTER 1

INTRODUCTION

1.1 Context

The widespread access to the Internet around the world has enabled millions of people to interconnect, store personal data, and disseminate information online. These services are accessed by users from all over the world that demand responsiveness and an always-on experience. Examples of such services are social networks like Facebook and Twitter and distributed storage such as Google Drive and Dropbox [Dro15; Inc15].

This class of applications manages massive amounts of data. For instance, Facebook reported over 4 billion items shared and 4.5 billion "likes" per day in 2013 [Fac13]. Applications that manage data at this scale require massive infrastructures to deal with the high-load of requests. Hence, these applications are mostly deployed in huge clusters hosted in data centers scattered around the world.

Replication is a widely used technique to achieve performance, fault-tolerance, availability, and low latencies in cloud services. Distributing data across different nodes (i.e. replicas) improves performance (in terms of throughput) because replicas can process user's requests concurrently. Moreover, if a replica fails (for instance due to a crash), the system remains accessible, thus providing availability and fault-tolerance. If replicas are geographically distributed, clients can contact the closest available replica to also reduce user-perceived latencies.

Ideally, replicated services would ensure strong consistency and availability all the time. However, the CAP theorem [GL02] has famously demonstrated that impossibility, in a distributed system prone to partitions. Indeed, the implications of the CAP theorem force applications to make hard decisions regarding the guarantees they provide to costumers. Providing strong consistency requires coordination among replicas to ensure that all replicas synchronously transit to the next (correct) state. Applications that deal with sensible information (e.g. online banking services) or applications that cannot afford to let

users read stale data tend to opt for strong consistency. However, this approach greatly limits the scalability and performance of the whole system [Bai+14]. For this reason, online services typically execute under more relaxed consistency models such as Eventual Consistency (EC) and Causal Consistency (CC). This allows replicas to process requests locally, thus boosting performance (which is translated into lower responses times to users). To ensure state-convergence, updates are lazely disseminated in the background. Systems that employ these consistency levels are known as weakly consistent systems.

To retrieve and store data, applications primarily resort to databases. In the literature, one can find two major classes of databases: relational databases and NoSQL databases. Relational databases promote well defined data relationships and expressive query capabilities. On the other hand, NoSQL databases emerged as a more practical solution to deal with big data, providing a simpler interface to programmers. Scaling databases to suit the needs of these applications has been a topic of intense research among both academia and industry communities in the last decades.

1.2 Motivation

Relational databases diverge from NoSQL databases in several fundamental ways.

The transactional model of relational databases is a powerful abstraction that reliefs developers from the complex reasoning about concurrency details. To exploit this feature in a distributed database, protocols like 2-Phase Commit (2PC) must be employed to coordinate transactions. However, in 2PC the chances of transactions *aborts* increases as the number of replicas grows [Gra+96]. This severely affects throughput and limits scalability. To overcome this challenge, services generally direct all updates to a single replica that serializes operations and pushes changes to secondary replicas [DS06; PA04; Pla+08; Sal+11]. Unfortunately, this approach does not scale well with *write intesive* workloads, as the primary replica becomes a *bottleneck*. On the other hand, relational databases offers a rich, well-structured data model and a compelling query language (the `SQL` language) that eases the development of applications.

In contrast, NoSQL databases tend to favor a more loose behavior, dropping the transactional semantics of relational databases. NoSQL databases are generally used in the context of weakly consistent systems, allowing every replica to process *write* requests [DeC+07; LM10; Tec15]. While this technique promotes conflicts (due to concurrent updates of the same object by different replicas), it is also very scalable and available, two increasingly important requirements for large scale applications. NoSQL systems accept this *tradeoff*; to resolve conflicts, these systems either use straightforward strategies such as *last-writer-wins* as in Cassandra [LM10], or expose conflicts to the application, which has the responsability to resolve them to ensure state convergence as in [DeC+07] and [Ter+95]. However, the simple data model provided by NoSQL databases may be inadequate for applications that require more complex data relationships and integrity constraints (e.g. *foreign-keys* and *check* contraints as in `SQL`).

Currently, there is a lack of pratical solutions that strive to combine the performance and scalability of NoSQL databases while offering a rich data model and transactional semantics as found in relational databases, which is essential for many applications.

CRDTs [Sha+11a; Sha+11b] are a family of data types very suitable to build weakly consistent systems because they provide three key characteristics: (1) *coorditation-free* operations, (2) state convergence, and (3) automatic reconciliation. Thus, CRDTs can be leveraged to build weakly consistent systems with high performance and availability.

1.3 Proposed Solution

We propose a solution that will try to consolidate the performance and scalability of NoSQL databases with the transtional model, integrity constraints and expressive data model provided by relational databases. Combining these features will allow developers to build applications with strict data relationships and expressive query capabilities while achieving a performance (in terms of throughput) close to NoSQL databases.

Our system will execute under a weak consistency level, in particular Eventual Consistency or Causal Consistency. This allows replicas to process requests locally and avoid coordination, thus boosting performance. We also want to preserve `SQL` invariants such as *check contraints*, *primary-keys*, and *unique* attributes. While some of these invariants can be preserved without any kind of coordination, others provably cannot [Bai+14]. For the latter, we will resort to a coordination service that will assure invariant's preservation.

We will deploy our system in a *cluster* and evaluate its performance using the TPC-C and Rubis benchmark [Con15; Cou15].

Chapter 3 discusses in more detail our proposed solution.

1.4 Document Organization

The remainder of this document is organized as follows. Chapter 2 presents fundamental topics related to the work proposed in this document and discusses proposed solutions in the literature. In particular, it starts by addressing replication in the context of databases. We present the most prominent database replication protocols and their characteristics, followed by a presentation of systems that illustrate the use of these protocols. We continue by discussing replication in the context of weakly consistent systems. We examine the requirements of today's Internet services and how they impact the design of distributed systems. We close this chapter by introducing and discussing CRDTs. Chapter 3 provides a high level overview of the proposed solution. We present the envisioned architecture of our system and show how CRDTs can be leveraged to achieve our goals.

CHAPTER 2

RELATED WORK

2.1 Overview

Relational databases have been a building block for many large scale internet applications. Much of the popularity of relational databases comes from the consistency model they expose to programmers. The ACID properties hide concurrency anomalies to programmers and ease the development of concurrent applications. These applications provide services to millions of users that demand responsiveness, high availability, and fault-tolerence. To tackle these challenges, current systems replicate their data across multiple nodes (i.e. replicas).

Replication can be broadly defined as the act of maintain multiple instances of the same object. Having multiple instances of an object improves performance, availability, and fault-tolerence. In the first case, the requests to access an object may be distributed over all the replicas that hold a copy of that object. Processing requests concurrently boosts performance because more requests can be processed per time unit, thus improving throughput. Moreover, it allows workloads to be distributed over the available resources, which enables efficient resources utilization. Additionally, if one of the replicas becomes unavailable (for instance due to a failure) the object is still accessible by clients, thus providing availability and fault-tolerence.

Given that the state of an object may vary over time, replication must also guarantee the consistency among different copies of the object. In a replicated environment, at some point different replicas may hold different values for a given object. Depending on the application semantic, such factor may or may not be a problem. For instance, a flight reservation system should not allow inconsistencies between replicas because that could lead to overselling tickets to clients. In contrast, a simple information feed, such as the one offered by Facebook, tolerate inconsistencies without major negative consequences. The amount of time replicas are allowed to diverge depends on three main aspects: (1) the

replication protocol used to disseminate information between replicas, (2) the deployment environment, and (3) the conditions of the network. Database replication protocols are discussed in the next section. Deployment environment refers to the underlying infrastructure in which replicas are executing (either in a *cluster* or in a geo-replicated setting). Network conditions are greatly influenced by the deployment environment. For instance, inside a *cluster*, network conditions deteriorate if links are congested. Links congestion arise because the oversubscription of channels communication (i.e. the ratio between the servers' bandwidth to the total uplink bandwidth at the access layer [HM14]).

As a consequence of allowing replicas to individually change a global (and shared) state, conflicts and divergent states naturally emerge. Typically, these challenges are solved by employing some deterministic technique such as Last-Writer-Wins (LWW), or left for the application to resolve [DeC+07; Ter+95]. CRDTs [Sha+11a; Sha+11b] go a step further and hide conflicts inside the data type itself while still guaranteeing state-convergence, making such data types very usefull for developers to use in weakly consistent replicated scenarios.

The remainder of this chapter covers topics that are directly related to our work. Section 2.2 discusses replication in the context of relational databases. Section 2.3 comprises weakly consistent systems and the *tradeoffs* associated. Finally, section 2.4 introduces and discusses CRDTs.

2.2 Database Replication

Before exploring databases in the context of a replicated system, for better understanding, we start by descriping the system model within a single replica.

2.2.1 System Model

Transactions are a building block for databases. A transaction is a logical unit of work that executes atomically (i.e either all operations are successfull or none is). Each transaction executes under an isolation level (one of the four ACID properties), which determines how other concurrent transactions affect it. Stricter isolation levels protect transactions to see the changes induced by concurrent transactions, while more relaxed isolation levels do not. Serializability, a particular strong isolation level, is the traditional correctness criteria for objects in relational databases. We now provide a formal definition of Serializability:

Definition 2.1. *Serializability: the execution of a set of concurrent transactions is equivalent to some serial execution of the transactions [Bai15; Ber+86; Bor+11]*

Executing transactions under Serializability isolation level requires major efforts by the concurrency control protocol to provide an isolated environment where transactions execute. This greatly reduces performance (in this case transcations per unit time). To overcome this challenge, more relaxed isolation levels have been proposed. Snapshot

Isolation (SI), Repeatable Read (RR), and Read Committed (RC) are popular isolation levels provided by most commercial database systems [MyS15; Ora15; Pos15]. Applications can leverage these isolation levels to achieve better throughput, in exchange of relaxing the isolation between concurrent transactions.

To increase performance, databases can also be distributed and replicated. Such approach is explored in the following sections.

2.2.2 Distributed Databases

Databases are replicated to improve performance and provide fault-tolerance. Distributed databases enable database replicas to process client's requests concurrently, thus boosting throughput. Relational databases foundations were not devised to be explored in a distributed system. This created major challenges when applying the same concepts from single replica databases to replicated databases. For instance, the traditional implementation of SI applied to a distributed environment allows some erratic behaviours that could not happen in a single replica scenario, such as transaction inversion [DS06]. Recent solutions have been proposed to generalize the concept of SI to a distributed database [Cha+14; DS06; Eln+05; Lin+09].

In a replicated setting, ideally, the system's throughput would grow linearly with the number of replicas. However, in order to ensure state convergence, replicas cannot act as totally independent components. Replication protocols are a fundamental feature to assure this.

2.2.3 Replication Protocols

Database replication protocols can be discussed in two dimensions. The first dimension comprise **how** replicas transit state, and includes active and passive replication. The second dimension refers to **where** and **when** updates occur (Master-Slave or Multi-Master, and Eager or Lazy respectively).

The remainder of this section explores these protocols in more detail. We will use *update*, *write*, and *operation* interchangeably. Note that *read-only* operations do not modify the state of the database, hence, they are not relevant for ensuring replica convergence.

Active Replication

Active replication (also known as State Machine replication) [Sch90; Gor; Wie+00] is a non-centralized replication technique. The intuition at the core of this approach is that if all replicas execute the same sequence of operations they reach the same state. In active replication, operations are replicated to all replicas and executed independently in each one. To ensure state convergence, this approach requires two properties: (1) operations are executed in the same order in every replica, and (2) operations are *deterministic*. It is easy to understand why both properties are necessary. First, if operations are executed in different orders, replicas may end up in an inconsistent state. For instance, if the effects of two *write*

operations to the same object do not commute. Secondly, if operations are not deterministic they produce different *outputs*. A classical example of this in the `SQL` language are *time-related* macros such as `now()`. Typical implementations of active replication resort to the properties that Atomic Broadcast provides [Gor; Wie+00] to ensure a total order of operations. For completeness we now provide a formal definition of Atomic Broadcast:

Definition 2.2. *Atomic Broadcast [Gor; Wie+00]: if a process delivers a message m before it delivers a message m', no process delivers m after delivering m'.*

Ensuring determinism of the operations is usually left to the developer to guarantee.

Passive Replication

In contrast to active replication, in passive replication [Gor; Wie+00] operations are executed once in a single replica. The modifications induced by the execution of the operations (also called the operation's *write-set*) are propagated to other replicas, which in turn install these modifications. Compared to active replication, this approach is more resource efficient because the operation need not to be executed, only its *side-effects*.

Eager Replication

In a Eager Replication scheme (also known as synchronous replication) [KA00; Wie+00], after receiving an update, the replica immediately propagates it to others replicas and waits for them to acknowledge the correct execution of the operation. Upon receiving acknowledgment from all replicas (or a subset of them, for instance a majority [KA00]), the operation is said to be sucessfully completed and the client is notified. Eager replication assures serializable execution of requests because replicas are (synchronously) contacted, within the boundaries of the operation. Hence, Eager replication provides strong consistency (see section 2.3). Figure 2.1 illustrates an eager protocol. Before replying to the client, the request is disseminated to all replicas, which increases the perceived latency.

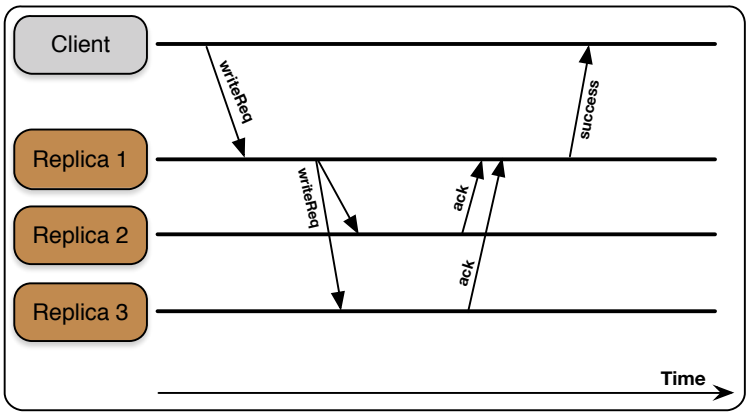

Figure 2.1: Eager Replication Protocol

Lazy Replication

By contrast to Eager Replication, in a Lazy Replication approach (also known as asynchronous replication, or optimistic replication) [DS06; Lad+92; SS05] operations are executed in a single replica and asynchronously disseminated to other replicas. Avoiding coordination in each operation greatly improves response times perceived by the user. This come at a cost of weakening consistency guarantees because replicas are allowed to diverge. Lazy Replication does not ensure that clients read the latest version of an object, hence providing weak consistency guarantees. Figure 2.2 presents how requests are processed under this protocol. Dashed arrows represents *background* communication.

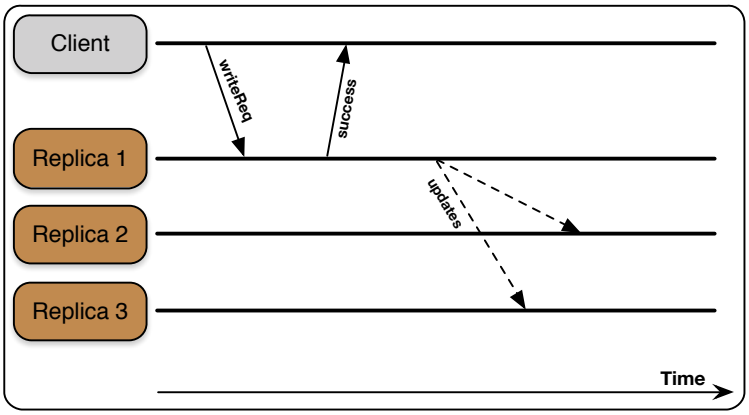

Figure 2.2: Lazy Replication Protocol

Master-Slave

In a Master-Slave setup (also known as Primary Copy) [Gor; Wie+00], one replica is designated the master replica while the others are called slaves. Only the master replica can process *write* requests. After receiving an update, the master replica propagates it accordingly to one of the schemes described above. Given that all *write* requests are directed to a single replica, ensuring a total order of updates is trivial and corresponds to the order by which the master processes the updates. A Master-Slave approach has two main drawbacks: (1) the master replica is a potential bottleneck, particularly in a workload composed mostly of *write* operations, and (2) the master replica is a Single Point of Failure (SPOF). Master-Slave schemes are mostly used to improve read performance, given that *read-only* operations can be issued to any replica and coordenation is not required for those operations. Figure 2.3 depicts a Master-Slave architecture. *Write* requests are directed to the master replica, while *read* requests are issued to a slave replica.

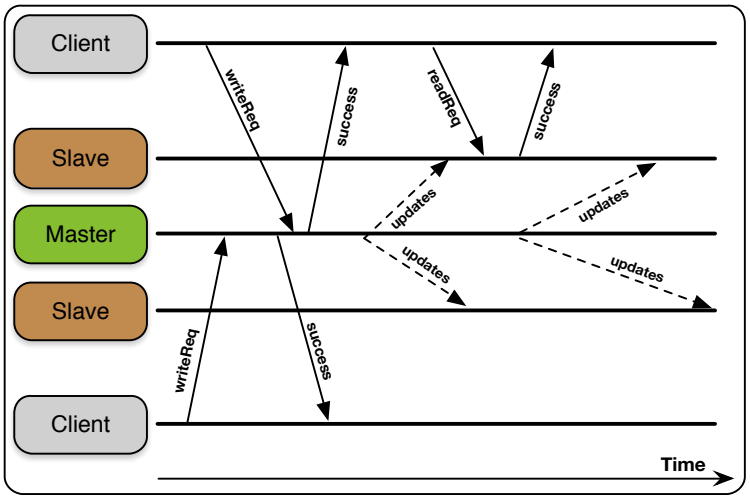

Figure 2.3: Master-Slave Replication Protocol

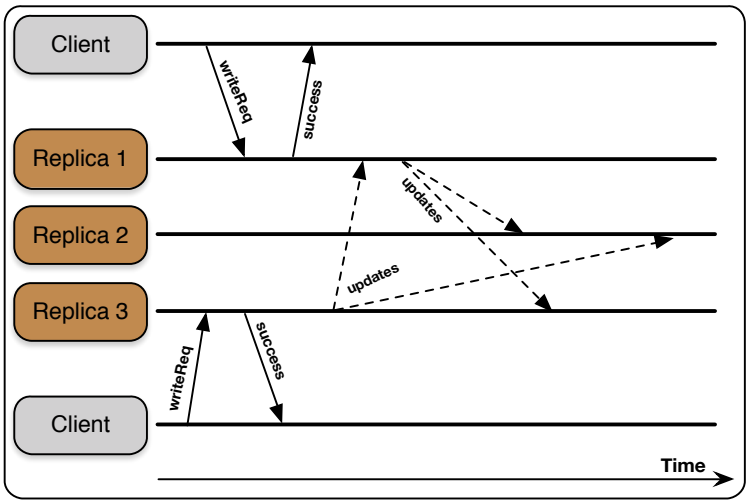

Figure 2.4: Multi-Master Replication Protocol

Multi-Master

In a Multi-Master architecture (also known as update anywhere replication) all replicas have the same capabilities. *Writes* can be performed by any replica, thus boosting *write* throughput. This also enables users to contact any replica for *write* operations, potentially reducing the latency for those operations. Coordenation mechanisms however are much more complex than in a Master-Slave architecture since concurrent updates may result in conflicts. The chances of deadlocks dramatically increase with the number of nodes. Gray et al. observed a thousand fold increase in deadlocks for a ten-fold increase in nodes and traffic [Gra+96]. Moreover, if updates are lazely replicated conflicts may arise only after some replicas have committed the operation. Thus, reconciliation techniques

must be employed in order to reconcile conflicting operations [DeC+07; LM10; Ren+08; Sha+11b]. One of these techniques is discussed in detail in section 2.4. Figure 2.4 shows a multi-master architecture together with a lazy replication protocol.

2.2.4 Case Studies

This section discusses existing solutions that resort to database replication techniques.

Plattner and Alonso introduced the concept of *satellites* databases [PA04; Pla+08]. In their system Ganymed, *satellites* databases serves *read-only* transactions while *writes* are tunneled to the master replica. This is achieved by a *scheduler* that separates and routes requests to the appropriate database replica. The same concept is applied by Salomie et al. in Multimed [Sal+11]. Multimed treats a *multi* core machine as a distributed system, where a subset of the cores work as *satellites* and another subset as the master replica. In both cases, *writes* are centralized in a single node, which could become a *bottleneck*. Moreover, in Ganymed, the presence of a network partition can prevent clients from communicating with the master replica leading to the unavailability of the system.

Amza et al. try to minimize the number of conflicts in transactions [Amz+03]. To this end, a central *scheduler* virtualizes the cluster to the application server as if it was a single database. All requests are tunneled to the *scheduler*, who is in charge of minimizing conflicts and redirect requests to appropiate replicas. In order to minimize conflicts the *scheduler* needs to know up front which tables will be accessed by each transaction. This solution has some drawbacks: (1) since all requests are tunneled to a *scheduler* the latency of the operations can be potentially high (2) the application must be adapted to provide the information needed by the *scheduler*.

Lin et al. proposed a *middleware* solution that implements One-Copy Snapshot Isolation (1SI) [Lin+05]. 1SI guarantees that SI properties hold even in a distributed context thus avoiding anomalies such as transaction inversion (see section 2.2). However, this solution executes under an eager replication protocol, which has poor scalability and performance.

ConfluxDB, an extension of PostgreSQL, [Cha+14] is a *multi* master lazy replicated system. ConfluxDB is able to determine a total SI compatible order of update transactions while avoiding synchronization. Each replica can either be master or slave: master replicas process updates and slaves only process *read-only* transactions. The database is partitioned over the set of primaries replicas. Each master produces an update's log that is later merged into a single stream by a separate component called Log Merger. Secondaries replicas lazily consume this unified log to be kept in sync. A major disadvantage of ConfluxDB is that data must be partitioned over primaries replicas. Since data is partitioned over different replicas, when two or more primaries are present, there is no guarantee that the user transaction can be processed exclusively in a single replica.

2.3 Weakly Consistent Systems

The CAP theorem proposed by Eric Brewer and later demonstrated by Gilbert and Lynch [GL02] had a decisive impact on the design of distributed systems. The CAP theorem states that it is impossible to achieve strong consistency (i.e. linearizability [GL02]), availability [BG13] while tolerating network partitions. We consider a system available if it continues to operate as expected, in the presence of partitons. For instance, we do not consider available a system that in the presence of partitons always respond with *no-op* to client's requests.

In distributed systems comprised by hundreds of interconnected nodes, partitions will eventually arise [Hal15]. Therefore, for this class of systems, the choice has always been between consistency and availability. The *tradeoff* between consistency and availability has been a *hot topic* in the research community for a long time.

Before exploring this *tradeoff*, it is important to clarify that consistency, from the point of view of the CAP theorem, has a different meaning when compared to the "Consistency" provided by the ACID properties of relational databases. While in a relational database Consistency refers to the guarantees that after each transaction the database remains in a consistent state, in distributed systems, Consistency refers to how *updates* are propagated between replicas and what guarantees the application provides when clients access its data [Vog08].

Lets assume two processes p_A and p_B. At some point p_A *writes* the value 1 in the system. In a strongly consistent system, it is guaranteed that any subsequent requests (either from p_A or p_B) observe this value. To achieve strong consistency, replicas must coordinate and synchronously transit to the next state. After a client issues an *update* operation, a subset of replicas (typically a *quorum*) must successfully acknowledge the execution of the operation before notifying the client. When an update is installed in a *quorum* it is guaranteed that subsequent operations (that also require a *quorum*) observe the update, because at least one replica participated in both *quoruns*. Strong consistency greatly reduces performance because more replicas must be contacted per operation, which has a severe impact on the throughput of the whole system. Moreover, strong consistency is at odds with availability. Figure 2.5 depicts a situation of a system with three replicas under a partition. Client $C1$ contacts replica $R1$ to issue an *update* operation. This operation is successfull because replica $R1$ is able to form a *quorum* with replica $R2$. Client $C2$ is connected to $R3$, which is on the other side of the partition. Because $R3$ cannot contact other replicas, operations in this replica will not suceed until the partitions heals. Thus, this system is not available for client $C2$.

In weakly consistent systems operations do not require upfront coordination among replicas. Operations are processed in the replica where the request was issued, and updates are lazely disseminated in the *background* to other replicas. Because replicas do not coordinate in each operation, weakly consistent systems have much better performance (in terms of throughput) and availability than strongly consistent systems. In exchange,

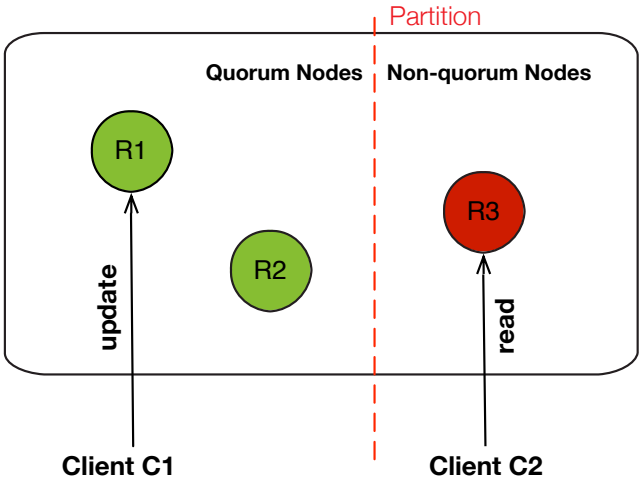

Figure 2.5: Partition in a Distributed System

these systems cannot provide strong consistency as defined before. We now return to the example in figure 2.5. Client $C1$ *writes* updates replica $R1$. At this point, replicas are not consistent because this update was not yet propagated. Now suppose that client $C2$ issues a *read* operation to $R3$. Because coordination is not required, $R3$ will successfully respond to this requests, in contrast to the previous example (even though it returns a stale value). Weak consistency also promotes conflicts, which arise when two (or more) concurrent updates modifies the same object. Automatic conflict resolution techniques such as Last-Writer-Wins resolves conflicts and ensures state convergence. Another approach is discussed in section 2.4.

A key idea that empowers weakly consistent systems is that partitions are temporary and eventually - some unknown time in the future - heal. In the absence of partitions, replicas can exchange updates and eventually reach the same state.

Whether an application should opt for strong or weak consistency greatly depends on its requirements. For instance, a flight reservation application should opt for strong consistency to avoid overselling tickets. On the other hand, social networks like Twitter or Facebook tend to opt for weak consistency because it provides a better performance and user experience, without introducing major negative consequences.

Picking strong or weak consistency is achieved effectively by the consistency model employed. In the literature, many consistency models have already been proposed: Linearizability [HW90]; Serializability [BG81]; Causal Consistency [Alm+13; Lam78; Llo+11]; Eventual Consistency [Vog08]. Moreover, recent proposals offers a two-fold behavior and execute under a multi-level consistency model [Kra+09; Lad+92; Li+12; Li+14; Ter+13]. While there are many systems that opt for strong consistency [Ard+13; Cor+12; Mah+13; Zha+13], we will focus on weakly consistent systems because they are closely related to the proposed work described in this document (see chapter 3).

The following sections discusses two particular weak consistent models: Eventual

Consistency and Causal Consistency.

2.3.1 Eventual Consistency

For completeness, we start by presenting a formal definition of Eventual Consistency.

Definition 2.3. *Eventual Consistency [EL14]:* ***Eventual delivery:*** *An update executed at one node eventually executes at all nodes.* ***Termination:*** *All update executions terminate.* ***Convergence:*** *Nodes that have executed the same updates eventually reach an equivalent state.*

Eventual Consistency is a particular weak consistency model. The only guarantee that EC provides is that if all update activity stops, then eventually all replicas reach the same logical state (i.e. replicas are indistinguishable).

To be available under network partitions, EC sacrifices consistency. Given that operations are not coordinated, EC cannot guarantee serializability [BG13] as originally defined by Haerder and Reuter [HR83]. However, most databases systems have been prioritizing weaker isolation levels such as Repeatable Read and Read Committed for quite some time, which suggest that these isolation levels are usefull for programmers to develop their applications [BG13; MyS15; Ora15; Pos15].

Implementing an EC system is rather straightforward. Replicas execute requests locally to ensure responsiveness, and asynchronously (in the *background*) disseminate information about the operations they have executed. The process of exchanging information between replicas is often called *anti-entropy* because it allows replicas to diminish the divergence between them. The time required for an update be visible in all replicas is called the *inconsistency window* [Vog08]. The *inconsistency window* can be shortened by employing *anti-entropy* protocols more frequently.

Section 2.3.3 discusses relevant systems that resort to Eventual Consistency.

2.3.2 Causal Consistency

Causal Consistency is another form of weak consistency. CC enables the tracking of relationships between operations, which in some applications might be necessary to provide an intuitive behavior. For instance, imagine three users exchanging messages among them; user `A` sends $m1$ to all other participants. User `B` reads $m1$ and replies with $m2$. In this scenario, it does not make sense for user `C` to receive $m2$ before $m1$. Clearly, there is an implicit relationship between $m1$ and $m2$, because $m2$ only "happened" in reply of $m1$.

Lamport in [Lam78] characterizes this relationship as "happened-before" relationship, which is doneted by $\rightarrow$. This relationship defines a partial order $\prec$ of events (in the above example, messages) in a distributed system [Lam78]. Formally, "happened-before" relationship is defined as follows [Lam78]:

Definition 2.4. ***(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 $a \rightarrow c$*

Causal Consistency [Alm+13; Lam78; Llo+11] guarantees that operations become visible to users in a order that respects their causal order. Hence, when the above scenario is executed under Causal Consistency, user $\subset$ is guaranteed to observe $m2$ only after observing $m1$ because there is a partial order $\prec$ such as $m1 \prec m2$ (see definition 2.4).

Tracking "happened-before" relationships can be achieved by using Vector Clocks (VCs). VCs consist in an vector of N entries (usually counters), where N is the number of nodes in the system [SS05]. Each replica manages an entry of the vector. When replica i processes an operation o, it increments $VC[i]$ and associates the new vector clock with o. It is said that a vector clock VC_1 dominates VC_2 if $VC_1 \neq VC_2$ and $\forall i \in 1...N, VC_1[i] \geq VC_2[i]$. Operation a "happened before" b if VC_a dominates VC_b. If neither vector clock dominates the other, the operations are concurrent [SS05].

Causal Consistency can be achieved with a modest performance penalty when compared to EC [BG13] and have been employed in recent geo-replicated systems [Alm+13; Llo+11; Llo+13]. CC it is also known to be the strongest consistency model that is available in the presence of network partitions [Mah+11].

2.3.3 Case Studies

In this section, we review relevant distributed systems that advocate weak consistency models.

In the last decade, the appearance of NoSQL databases has led many weakly consistent systems to be developed, both by academia and industry [Cha+08; Coo+08; DeC+07; Fit04; LM10; Llo+11; Llo+13; Tec15]. In Dynamo [DeC+07], each object is identified by a *key* and is replicated across N replicas (hence tolerating up to $N-1$ failures). Replicas are placed in a ring and each one is responsible for a subset of the *key-space*. *Read* and *write* operations specify a parameter - R and W respectively - that correspond to the number of replicas that must acknowledge the request before notifying the client. Maximum availability and minimum response time are achieved by setting $W = R = 1$. Dynamo is used by Amazon to store and retrieve shopping carts data. Google's BigTable [Cha+08] and Yahoo's PNUTS [Coo+08] are another examples of non-relational databases that achieve high scalability and performance.

Li et al. proposed Gemini [Li+12], a coordination layer that supports a two-fold consistency model called RedBlue Consistency. In Gemini operations are labeled either blue or red. Blue operations execute locally and are lazily replicated to remote sites, thus providing EC. Red operations are totally ordered among them and require coordination between replicas, thus providing strong consistency. For an operation to be labeled blue, it must fullfill two conditions: (1) the operation must be commutative in respect to all others, and (2) the operation must not break application's invariants. To boost the number of

blue operations, each operation is decoupled into two parts: (1) a *side-effect free* generator operation determine the changes induced by the original operation, and (2) a shadow operation that effectively modifies the state of the system . While Gemini offers a flexible behavior, the generation of shadow operations is not automated and demands manual intervention. Sieve [Li+14], presents an extension to this work, which was designed to tackle this problems. Sieve automatically labels and produces commutatives shadow operations, minimizing the developer interventation.

Clements et al. [Cle+13] also highlights the importance of commutativity in the scalability of a system. They introduced a general rule that states the following: "whenever interface operations commute, they can be implemented in a way that scales". To exploit this property, they developed a tool that analyzes a set of operations based on the interface's specification, and identify under which conditions operations commutes. They used this tool to modify POSIX's call in a linux-based *kernel*, and reported that most POSIX's are possible to implement in a scalable way.

Bayou is a weakly consistent replicated storage system [Ter+95]. It was design to serve clients with unstable network conditions, as found in early mobile connections. To overcome this limitation, operations only need to contact a single replica. Updates are lazely propagated in the background, thus providing EC. In contrast to Dynamo, Bayou resolves conflicts automatically without exposing them to the application level. However, the merge procedures must the provided by the application developer.

ChainReaction [Alm+13] is a distributed *key-value* store that provides CC. Replicas placement follow the same approach as in Dynamo. The subset of replicas responsible for managing a given object is called a chain. A client-side library forwards requests to the apropriate chain. Writes are managed by the head of the chain, which has the responsability to propagate the updates throughout the rest of the chain; read operations are directed to the tail of the chain. When a write request reaches the tail, it is guaranteed that all the chain have already seen the update. At this point, the tail acknowledge back to the client (intuitively this works as a serialization point). To minimize latency, a parameter k defines the number of replicas that must receive the update before returning to the client. While achieving CC in read operations with low latency, write operations must wait for the update to be replicated in k replicas, augmenting the perceived latency.

2.4 Conflict-Free Replicated Data Types

Section 2.3 presented EC, a weak consistency model suitable for providing performance and availability. Replicate data under Eventual Consistency is complex because any replica can accept updates without coordinating with other replicas, which may result in conflicts. Resolving conflicts is usually left for the application to resolve [DeC+07; Ter+95]. This approach does not resolve the problem in a general way because the merge *procedures* are application-dependent. Moreover, this solution requires significant effort from developers and can introduce *bugs* in the application.

CRDTs [Sha+11b] are a collection of data types where conflicts are resolved inside the data type itself, hence masking conflicts to programmers, relieving them from worrying about concurrency details. Eschewing complex merging *procedures*, CRDTs provably converge and remain available under any number of *failures* [Sha+11b]. Hence, they are very useful in weakly consistent systems. CRDTs provide Strong Eventual Consistency (SEC). Formally, SEC can be defined as:

Definition 2.5. *Strong Eventual Consistency [Sha+11b]: An object is Strongly Eventually Consistent if it is Eventually Consistent and* ***Strong Convergence:*** *Correct replicas that have delivered the same updates have equivalent state:* $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$

The additional property of SEC (when compared to Eventual Consistency) implies that CRDTs are in a consistent state after delivering the same set of operations. Note that Eventual Consistency does not require this property; it only states that after delivering the same set of updates, **eventually** replicas converge. The key insight here is that in CRDTs, reconciliation is done as part of the operation itself.

We now give a general overview on how CRDTs operate.

CRDTs expose two types of operations to developers: *query* and *update* operations. *Query* operations read the local state of the object and *update* operations modify its local state. *Update* operations are decoupled in two phases: (1) the first phase is executed locally and may perform some preliminar computations, (2) the second phase, also known as `downstream` phase, disseminates the update asynchronously to all replicas [Sha+11a]. The replica where the client issues the operation is called the `source` replica for that operation.

CRDTs can either be *operation-based* or *state-based*. In a *operation-based* CRDT, the operation is replicated by the `source` replica and executed at all replicas (i.e. active replication). In a *state-based* CRDT, the modified state is replicated instead of the operation (i.e. passive replication). Both approaches are discussed in the following sections.

2.4.1 State-based CRDT

In a state-based CRDT (or Convergent Replicated Data Type (CvRDT)), an *update* executes exclusively in the `source` replica. Asynchronously, the modified objects are sent to other replicas. Hence, it is a classical example of passive replication. When a replica receives a remote state it executes a *merge* operation that merge the received and the local state. After *merge* operations, both replicas are in a consistent state.

We now present a more formal definition. A CvRDT is a tuple (S, s^0, q, u, m). S represents the state domain of the object and s^0 represents the initial state; q is the set of query operations and u the set of update operations. When an update operation is executed, the object transits from s^i to s^{i+1}

2.4.2 Operation-based CRDT

An operation-based CRDT (or Commutative Replicated Data Type (CmRDT)), is a tuple (S, s^0, q, t, u, P) where S, S^0 and q preserve the same meaning as before. In CmRDTs, an *update* operation is decoupled in two operations: (1) t is a *side-effect free* prepare-update operation, and (2) u is an effect-update operation, that actually modifies the state. The *prepare-update* operation corresponds to the first phase described before. It executes only at the *source* replica and it does not modify the internal state of the CRDT. This phase is used to generate unique *tags* or calculate *timestamps* needed to allow automatic reconciliation. Let us call this the operation's *metadata*.

The *effect-update* operation is executed at *source* replica right after phase 1. In the *background*, the operation is shipped to other replicas along with the metadata. The *effect-update* operation corresponds to phase 2. Given that all replicas effectively execute the operation, a CmRDT is a typical example of active replication.

Finally, P is the precondition that must be met before executing phase 2 (in the next section we will give a pratical example of how *preconditions* are exploited in this scenario).

2.4.3 CRDTs Catalogue

While there are more CRDTs defined in the literature (such as graphs, counters and replicated arrays [Sha+11a]), this section will describe the data types that are most relevant to our work: registers and sets. The following specifications were presented by Shapiro et al. in [Sha+11a].

LWW Register

A register is a memory cell that supports both *assign* (*update*) and *value* (*query*) operations. Each *assign* operation is associated with a *timestamp*. Non-concurrent *assigns* preserve sequential semantics (i.e. the later one overwrites the earlier one), which means that *assign* operations do not commute. In a state-based `LWW Register`, the *assign* operation always suceeds, since there is no "greater" *timestamp* then the one just calculated. The new state is eventually replicated to all replicas. The `merge` operation checks both remote and local *timestamp* to decide the final value of the object. Specification 1 presents the *pseudocode* for this CRDT.

In an operation-based `LWW Register` (see specification 2), *assign* operation is splitted into two operations as discussed before. During phase 1, a new *timestamp* is calculated; phase 2 (`downStream`), modifies the value of the register, if and only if the new *timestamp* is greater then the current *timestamp*.

Sets

Sets are a building block for implemeting many data structures such as Maps, Containers and Graphs. A traditional implementation of a Set includes three basic operations: *add*,

Specification 1 LWW Register (state-based)

```
1:  payload X x, timestamp t                          ▷ X : some type
2:    initial ⊥, 0
3:  update assign(X w)
4:    x, t = w, now()                  ▷ Timestamp, consistent with causality
5:  query value() : X w
6:    let w = x
7:  compare (R, R') : boolean b
8:    let b = (R.t ≤ R'.t)
9:  merge (R, R') : payload R''
10:   if R.t ≤ R'.t
11:     R''.x, R''.t = R'.x, R'.t
12:   else
13:     R''.x, R''.t = R.x, R.t
```

Specification 2 LWW Register (operation-based)

```
1:  payload X x, timestamp t                          ▷ X : some type
2:    initial ⊥, 0
3:  query value() : X w
4:    let w = x
5:  update assign(X x')
6:    atSource() t'
7:      let t' = now()                                      ▷ Timestamp
8:    downstream(x', t')          ▷ no precondition: delivery order is empty
9:      if t < t'
10:       x, t := x', t'
```

remove and *lookup*.

G-Set

A G-Set (i.e Grow-Only Set) supports *add* (*update*) and *lookup* (*query*) operations. Since the only operation that modifies its state is the *add* operation (which is commutative by design), an operation-based G-Set naturally commutes.

In a state-based approach, the *add* operation locally adds an element to the set, and the merge operation computes the union between local and remote state.

The specification 3 defines the state-based G-Set CRDT.

2P-Set

In the 2P-Set (see specification 4), elements may be added and removed, but never added again. The implementation is based on two G-Set, one for adds (S_a) and the other for removes (S_r). An element e is said to be in the Set if $e \in S_a \wedge e \notin S_r$. Remove operation is allowed if and only if the *source* replica contains the element. Hence, a *remove* operation must first query the set to check if element e is present. This behavior can be captured by requiring a precondition $P : lookup(e)$.

Specification 3 G-Set (state-based)

```
1:  payload set A
2:    initial ∅
3:  update add (element e)
4:    A := A ∪ {e}
5:  query lookup (element e) : boolean b
6:    let b = (e ∈ A)
7:  compare (S, T) : boolean b
8:    let b = (S.A ⊆ T.A)
9:  merge (S, T) : payload U
10:   let U.A = S.A ∪ T.A
```

Specification 4 2P-Set (state-based)

```
1:  payload set A, set R                          ▷ A: added; R: removed
2:    initial ∅, ∅
3:  query lookup (element e) : boolean b
4:    let b = (e ∈ A ∧ e ∉ R)
5:  update add (element e)
6:    A := A ∪ {e}
7:  update remove (element e)
8:    pre lookup(e)                               ▷ precondition
9:    R := R ∪ {e}
10: compare (S, T) : boolean b
11:   let b = (S.A ⊆ T.A ∨ S.R ⊆ T.R)
12: merge (S, T) : payload U
13:   let U.A = S.A ∪ T.A
14:   let U.R = S.R ∪ T.R
```

OR-Set

While usefull in some cases, previous specifications do not reflect the natural behavior of a Set. The G-Set does not allows removes, while the 2P-Set does not allows elements to be added after being removed. The OR-Set closely resembles a more general implementation of a Set. Specification 5 defines an operation-based OR-Set.

In this variant, each element has an unique *tag* associated. *Add* operation generates an unique *tag* at *source* replica, inserts the pair *(element, tag)* in the set and then propagates this pair to other replicas. Since two concurrent *adds* generate unique *tags*, duplicates may be present. Hence, *lookup* operation must mask duplicates.

When removing an element, the set of unique *tags* present at *source* replica are recorded. After *remove* operation, the element is no longer in the set and the collection of *tags* presented during the operation is propagated to other replicas, which in their turn will remove all elements that match with the *tags* received. In this CRDT, *add* takes precedence over *remove* because the *tags* observed by *remove* do not include the *tag* generated by the concurrent *add* operation.

Specification 5 OR-Set (operation-based)

```
1:  payload set S                                   ▷ set of pairs {(e, unique − tag μ, ...)}
2:    initial ∅
3:  query lookup(element e) : boolean b
4:    let b = (∃_μ : (e, μ) ∈ S)
5:  update add(element e)
6:    atSource (e)
7:      let α = unique()                            ▷ unique(): returns a unique value
8:    downstream (e, α)
9:      S := S ∪ {(e, α)}
10: update remove(element e)
11:   atSource (e)
12:     pre lookup(e)
13:     let R = {(e, μ)|∃_μ : (e, μ) ∈ S}
14:   downstream (R)
15:     pre ∀(e, μ) ∈ R : add(e, μ) has been delivered   ▷ Causal order suffices
16:     S := S \ R                                  ▷ Remove pairs observed at source
```

CHAPTER 3

PROPOSED WORK

This chapter details our proposed work. We start by giving a general overview of our projected system and conclude by presenting our work plan.

3.1 Solution

We plan to develop a weakly consistent replicated relational database with increased performance and scalability than traditional relational databases.

Replication is a typical solution to improve performance and was discussed in section 2.2. We propose a Multi-Master Lazy Replication architecture for two main reasons: (1) it improves *write* throughput by allowing every replica to perform updates, and (2) it avoids the coordination costs associated with eager protocols. This approach has however some challenging problems. First, by employing a lazy protocol we enable replica divergence. Second, by allowing multiple replicas to perform *write* operations, conflicting updates may arise. Moreover, a Multi-Master solution is at odds with preserving global invariants. Consider the following example. Assume that some attribute *stock* must be always positive (i.e. `check stock > 0`) and assume that the current value is `stock = 10`. Now consider two concurrent operations, issued to different replicas, that want to decrease this value by 6 (i.e. `stock -= 6`). Executing each operation locally, would leave both replicas in a consistent state, where `stock = 4`. However, when both operations were disseminated to other replicas, they would reach a state where `stock = -2`, thus breaking the invariant.

To tackle these problems we propose the following. First, we will explore CRDTs properties to reconcile conflicting updates and ensure state convergence, in a *coordination-free* manner. Second, to preserve global invariants we will implement a coordination service that will arbitrate if an operation is safe to proceed. In the example above, both replicas must contact this service before executing the operation to avoid the described

undesired outcome. Note however that in some cases these operations are safe to execute locally. For instance, if the above operations increase the *stock* instead of decreasing it, then they are perfectly safe to execute locally.

3.1.1 Previous Work

Our work will extend Sieve's work [Li+14], developed by Li et al. Sieve already leverages CRDTs in the context of relational databases. However, Sieve was designed for a geo-replicated setting, and executes under RedBlue Consistency model [Li+12]. In contrast, our solution is intended to be deployed in a *cluster* and executes under a weaker consistency level.

Sieve leverages CRDTs as follows. Database tables consist in a list of *tuples*. The relational model requires that each *tuple* is *unique* within each table. Thus, we can look at each table as a collection of *tuples*. By definition, this is equivalent to a Set. Section 2.4 discussed implementations of CRDT-Sets. `SQL` statements `INSERT`, `SELECT`, and `DELETE` can be viewed as CRDT-Set operations (*add*, *lookup*, and *remove* respectively, in the case of the OR-Set). Moreover, because CRDTs are SEC compliant, eventually, the content of each table of the database is consistent across all replicas.

3.1.2 Architecture

Figure 3.1 depicts an high level view of our proposed solution. It consists in a *cluster* of machines; each node is a share-nothing database replica. The *middleware* component is responsible for (1) decide if an operation must contact our coordination service or if it can safely be executed locally, and (2) implement the replication protocol. The *middleware* delivers operations to a custom `JDBC` driver that will transform each operation in a sequence of CRDT commutative operations. Finally, the custom driver *commits* the operations.

This is still a very superficial view of our solution. The architecture design is still an ongoing process.

3.1.3 Evaluation

To evaluate the performance of our solution we will test it against two popular benchmarks: TPC-C [Cou15] and Rubis [Con15]. We will analyze how the size of the *cluster* (i.e. the number of replicas) impacts the throughput of our system. To do so, we will deploy our system in variable sized *clusters* and compare the results.

3.2 Work Plan

Our work plan is divided into three major tasks: (1) implementation, (2) evaluation, and (3) writting. Figure 3.2 presents the gantt chart of the planned work. We now briefly describe each *subtask*.

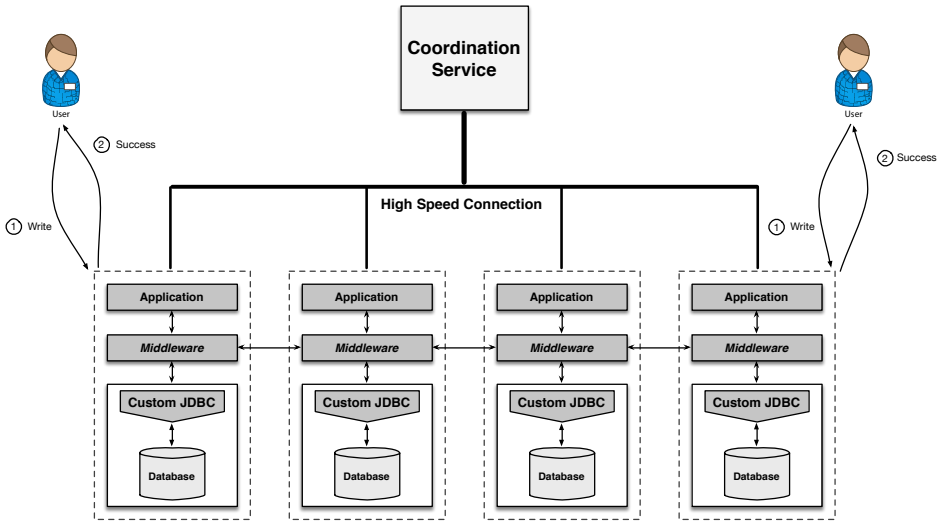

Figure 3.1: Proposed solution high level overview

Design and interfaces specification In this task we will refine the architecture of our solution, by clearly defining the responsibilities of each component and specifying an interface for each one.

JDBC extension During this phase we will extend the `JDBC` driver to provide the required functionalities.

Coordination service implementation In this *task* we will develop the coordination module.

Replication protocol This task comprises the implementation of the replication protocol.

Environment deployment In this phase we will deploy our solution in a *cluster* and make the necessary changes to the TPC-C and Rubis source code.

TPC-C benchmark This task comprises the execution of the TPC-C benchmark.

Rubis benchmark This task comprises the execution of the Rubis benchmark.

Thesis writting This phase encompass the writting of the final document.

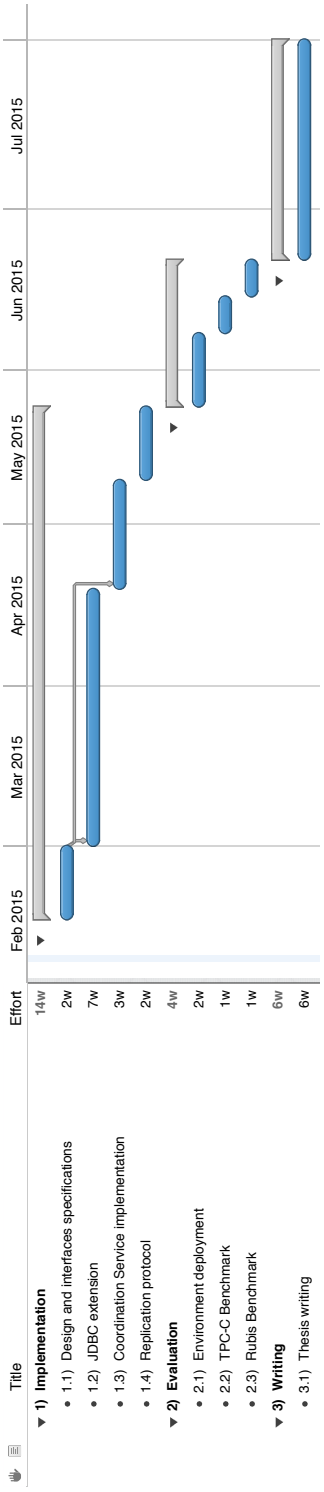

Figure 3.2: Gantt chart

BIBLIOGRAPHY

[Alm+13] S. Almeida, J. Leitao, and L. 1. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2013, pp. 85–98.

[Amz+03] C. Amza, A. L. Cox, and W. Zwaenepoel. "Conflict-aware Scheduling for Dynamic Content Applications". In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. Berkeley, CA, USA: USENIX Association, 2003, pp. 6–6.

[Ard+13] M. S. Ardekani, P. Sutra, and M. Shapiro. "Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems". In: *Reliable Distributed Systems, IEEE Symposium on* 0 (2013), pp. 163–172.

[Bai15] P. Bailis. *Linearizability versus Serializability*. Feb. 2015. URL: http://www.bailis.org/blog/linearizability-versus-serializability/.

[BG13] P. Bailis and A. Ghodsi. "Eventual Consistency Today: Limitations, Extensions, and Beyond". In: *Queue* 11.3 (Mar. 2013), 20:20–20:32. ISSN: 1542-7730. DOI: 10.1145/2460276.2462076. URL: http://doi.acm.org/10.1145/2460276.2462076.

[Bai+14] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Coordination-Avoiding Database Systems". In: *CoRR* abs/1402.2237 (2014).

[BG81] P. A. Bernstein and N. Goodman. "Concurrency Control in Distributed Database Systems". In: *ACM Comput. Surv.* 13.2 (June 1981), pp. 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846. URL: http://doi.acm.org/10.1145/356842.356846.

[Ber+86] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10715-5.

[Bor+11] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. "One-copy Serializability with Snapshot Isolation Under the Hood". In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 625–636.

[Cha+14] P. Chairunnanda, K. Daudjee, and M. T. Özsu. "ConfluxDB: Multi-Master Replication for Partitioned Snapshot Isolation Databases". In: *PVLDB* 7.11 (2014), pp. 947–958.

[Cha+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems* 26.2 (June 2008), pp. 4–26.

[Cle+13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2013, pp. 1–17.

[Con15] O. Consortium. *Rubis Benchmark*. Feb. 2015. URL: http://rubis.ow2.org/.

[Coo+08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1277–1288.

[Cor+12] J. C. Corbett et al. "Spanner: Google's Globally-distributed Database". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012, pp. 251–264.

[Cou15] T. P. P. Council. *TPC-C Benchmark*. Feb. 2015. URL: http://www.tpc.org/tpcc/.

[DS06] K. Daudjee and K. Salem. "Lazy Database Replication with Snapshot Isolation". In: *Proceedings of the 32th International Conference on Very Large Data Bases*. VLDB Endowment, 2006, pp. 715–726.

[DeC+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007, pp. 205–220.

[Dro15] Dropbox. *Dropbox*. Jan. 2015. URL: https://www.dropbox.com/developers.

[EL14] M. M. Elbushra and J. Lindström. "Eventual Consistent Databases: State of the Art". In: *Open Journal of Databases (OJDB)* 1.1 (2014), pp. 26–41. ISSN: 2199-3459. URL: http://www.ronpub.com/publications/OJDB-v1i1n03_Elbushra.pdf.

[Eln+05] S. Elnikety, W. Zwaenepoel, and F. Pedone. "Database Replication Using Generalized Snapshot Isolation". In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 73–84.

[Fac13] Q. Facebook Ericsson. *A Focus on Efficiency*. Tech. rep. 2013.

[Fit04] B. Fitzpatrick. "Distributed Caching with Memcached". In: *Linux J.* 2004.124 (Aug. 2004), pp. 5–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1012889.1012894.

[GL02] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59.

[Gra+96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In: *SIGMOD Record* 25.2 (June 1996), pp. 173–182.

[HR83] T. Haerder and A. Reuter. "Principles of Transaction-oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://doi.acm.org/10.1145/289.291.

[Hal15] C. Hale. *You Can't Sacrifice Partition Tolerance*. Feb. 2015. URL: http://codahale.com/you-cant-sacrifice-partition-tolerance/.

[HM14] A. Hammadi and L. Mhamdi. "Review: A Survey on Architectures and Energy Efficiency in Data Center Networks". In: *Comput. Commun.* 40 (Mar. 2014), pp. 1–21. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2013.11.005. URL: http://dx.doi.org/10.1016/j.comcom.2013.11.005.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[Inc15] G. Inc. *Google Drive*. Feb. 2015. URL: https://developers.google.com/drive/.

[KA00] B. Kemme and G. Alonso. "A New Approach to Developing and Implementing Eager Database Replication Protocols". In: *ACM Trans. Database Syst.* 25.3 (Sept. 2000), pp. 333–379.

[Kra+09] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. "Consistency Rationing in the Cloud: Pay Only when It Matters". In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009), pp. 253–264.

[Lad+92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. "Providing High Availability Using Lazy Replication". In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 360–391. ISSN: 0734-2071. DOI: 10.1145/138873.138877. URL: http://doi.acm.org/10.1145/138873.138877.

[LM10] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40.

[Lam78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565.

[Li+12] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. USENIX Association, 2012, pp. 265–278. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

[Li+14] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 281–292. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

[Lin+05] Y. Lin, B. Kemme, M. Pati n o Mart 'ınez, and R. Jiménez-Peris. "Middleware Based Data Replication Providing Snapshot Isolation". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2005, pp. 419–430.

[Lin+09] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Pati n o Mart 'ınez, and J. E. Armend a riz-I n igo. "Snapshot Isolation and Integrity Constraints in Replicated Databases". In: *ACM Trans. Database Syst.* 34.2 (July 2009), 11:1–11:49.

[Llo+11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. 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. New York, NY, USA: ACM, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043593. URL: http://doi.acm.org/10.1145/2043556.2043593.

[Llo+13] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. 2013, pp. 313–328.

[Mah+11] P. Mahajan, L. Alvisi, and M. Dahlin. *Consistency, Availability, Convergence*. Tech. rep. TR-11-22. Computer Science Department, University of Texas at Austin, 2011.

[Mah+13] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. "Low-latency multi-datacenter databases using replicated commit". In: *Proceedings of the VLDB Endowment* 6.9 (July 2013), pp. 661–672.

[MyS15] MySQL. *MySQL 5.6 Documentation*. Jan. 2015. URL: http://dev.mysql.com/doc/refman/5.6/en/index.html.

[Ora15] Oracle. *Oracle Database 11g*. Jan. 2015. URL: https://docs.oracle.com/cd/E11882_01/nav/portal_4.htm.

[PA04] C. Plattner and G. Alonso. "Ganymed: Scalable Replication for Transactional Web Applications". In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 155–174.

[Pla+08] C. Plattner, G. Alonso, and M. T. O zsu. "Extending DBMSs with Satellite Databases". In: *The VLDB Journal* 17.4 (July 2008), pp. 657–682.

[Pos15] PostgreSQL. *PostgreSQL 9.2 Documentation*. Jan. 2015. URL: postgresql.org/docs/9.2/static/.

[Ren+08] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. "Efficient Reconciliation and Flow Control for Anti-entropy Protocols". In: *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2008, 6:1–6:7.

[SS05] Y. Saito and M. Shapiro. "Optimistic Replication". In: *ACM Computing Surveys* 37.1 (Mar. 2005), pp. 42–81.

[Sal+11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. "Database engines on multicores, why parallelize when you can distribute?" In: *Proceedings of the Sixth Conference on Computer Systems*. ACM Press, 2011, pp. 17–30.

[Sch90] F. B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167.

[Sha+11a] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. Tech. rep. RR-7506. INRIA, 2011.

[Sha+11b] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. 2011, pp. 386–400.

[Gor] "State of the Art in Database Replication". In: (Jan. 2007), pp. 1–64.

[Tec15] B. Technologies. *Riak*. Jan. 2015. URL: http://basho.com/riak/.

[Ter+95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1995, pp. 172–182.

[Ter+13] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based service level agreements for cloud storage". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM Press, 2013, pp. 309–324.

[Vog08] W. Vogels. "Eventually Consistent". In: *Queue* 6.6 (Oct. 2008), pp. 14–19. ISSN: 1542-7730. DOI: 10.1145/1466443.1466448. URL: http://doi.acm.org/10.1145/1466443.1466448.

[Wie+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. "Understanding Replication in Databases and Distributed Systems". In: 2000, pp. 264–274.

[Zha+13] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. "Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2013, pp. 276–291.