NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SOFIA FREDERICO DE SOUSA BRAZ

Bachelor in Computer Science and Engineering

TRANSACTION PROCESSING OVER GEO-PARTITIONED DATA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

January, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TRANSACTION PROCESSING OVER GEO-PARTITIONED DATA

SOFIA FREDERICO DE SOUSA BRAZ

Bachelor in Computer Science and Engineering

Adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Pedro Abílio Duarte de Medeiros
Associate Professor, FCT-NOVA

Rapporteur: João Coelho Garcia
Auxiliar Professor, Instituto Superior Técnico

Adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, FCT-NOVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
January, 2022

Transaction Processing over Geo-Partitioned Data

To my grandparents.

Acknowledgements

First, I would like to thank my advisor, Professor Doctor Nuno Preguiça, for his guidance and the opportunity to work on this project. His support, availability and knowledge were fundamental throughout the development of this dissertation.

I would like to thank the PhD student André Rijo, who, as a subject matter expert in PotionDB, helped me throughout the project by explaining how PotionDb works and giving recommendations on how our solution should be implemented.

To my closest friends and colleagues, who I met during my journey at FCT-NOVA and who were always available to help me. Over the years, I have always been able to count on their friendship, support and motivation, and for that, I am incredibly grateful. A special thanks to my friend and colleague Joana Barbosa, who was always available to motivate and put up with me in times of stress.

Finally, I would like to thank my family. I am thankful to my parents for inspiring me to pursue this path, for their support and for providing me with all the necessary tools to complete this journey. To my brothers for encouraging me every day to be better. And to my boyfriend for his patience, his ongoing support and always believing in me.

This work was partially supported by FCT / MCTES grants PTDC/CCI-INF/32662/2017, Lisboa - 01 - 0145 - FEDER - 032662 (SAMOA) and UIDB/04516/2020 (NOVA LINCS).

"Originality implies a return to the origins,
original is returning to the simplicity of the first solutions."
(Antoni Gaudí)

Abstract

Databases are a fundamental component of any web service, storing and managing all the service data. In large-scale web services, it is essential that the data storage systems used consider techniques such as partial replication, geo-replication, and weaker consistency models so that the expectations of these systems regarding availability and latency can be met as best as possible.

In this dissertation, we address the problem of executing transactions on data that is partially replicated. In this sense, we adopt the transactional causal consistency semantics, the consistency model where a transaction accesses a causally consistent snapshot of the database. However, implementing this consistency model in a partially replicated setting raises several challenges regarding handling transactions that access data items replicated in different nodes.

Our work aims to design and implement a novel algorithm for executing transactions over geo-partitioned data with transactional causal consistency semantics. We discuss the problems and design choices for executing transactions over partially replicated data and present a design to implement the proposed algorithm by extending a weakly consistent geo-replicated key-value store with partial replication, adding support for executing transactions involving geo-partitioned data items. In this context, we also addressed the problem of deciding the best strategy for searching data in replicas that hold only a part of the total data of a service and where the state of each replica might diverge.

We evaluate our solution using microbenchmarks based on the TPC-H database. Our results show that the overhead of the system is low for the expected scenario of a low ratio of remote transactions.

Keywords: Geo-replication; partial replication; causal consistency; transactions.

Resumo

As bases de dados representam um componente fundamental de qualquer serviço web, armazenando e gerindo todos os dados do serviço. Em serviços web de grande escala, é essencial que os sistemas de armazenamento de dados utilizados considerem técnicas como a replicação parcial, geo-replicação e modelos de consistência mais fracos, de forma a que as expectativas dos utilizadores desses sistemas em relação à disponibilidade e latência possam ser atendidas da melhor forma possível.

Nesta dissertação, abordamos o problema de executar transações sobre dados que estão parcialmente replicados. Nesse sentido, adotamos uma semântica de consistência transacional causal, o modelo de consistência em que uma transação acede a um snapshot causalmente consistente da base de dados. No entanto, implementar este modelo de consistência numa configuração parcialmente replicada levanta vários desafios relativamente à execução de transações que acedem a dados replicados em nós diferentes.

O objetivo do nosso trabalho é projetar e implementar um novo algoritmo para a execução de transações sobre dados geo-particionados com semântica de consistência causal transacional. Discutimos os problemas e as opções de design para a execução de transações em dados parcialmente replicados e apresentamos um design para implementar o algoritmo proposto, estendendo um sistema de armazenamento chave-valor geo-replicado de consistência fraca com replicação parcial, adicionando suporte para executar transações envolvendo dados geo-particionados. Nesse contexto, também abordamos o problema de decidir a melhor estratégia para procurar dados em réplicas que guardam apenas uma parte total dos dados de um serviço e onde o estado de cada réplica pode divergir.

Avaliamos a nossa solução utilizando microbenchmarks baseados na base de dados TPC-H. Os nossos resultados mostram que a carga adicional do sistema é baixa para o cenário esperado de uma baixa percentagem de transações remotas.

Palavras-chave: Geo-replicação; replicação parcial; consistência causal; transações.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

1.1 Context and Motivation

Databases represent a vital component for any web service, as they are responsible for storing and managing all service data. Given any large distributed database system, it is essential to consider features such as data replication and consistency to guarantee the database's good functioning.

When developing global services, it is necessary to rely on geo-replicated databases to ensure that services running in multiple locations can access replicas of data in the data centers where the service is running. Geo-replication is an essential technique for any large distributed system that wants to achieve low latency and high availability, allowing clients to access the nearest data center and improve their response times [35].

In terms of consistency, databases that offer strong consistency, giving the illusion that a single replica exists, lead to high latency and can compromise availability since it is necessary to perform coordination between the various replicas in order to perform write operations. Databases that offer weak consistency lead to lower latency and high availability since any replica can process a client request without constant synchronization [35]. In this context, causal consistency [26, 4] is a particularly interesting model, as it provides stronger guarantees than simple eventual consistency while still retaining the high availability property under the presence of faults. However, in this case, the state of different database replicas can diverge temporarily, in the presence of concurrent updates, which makes system development a challenge [14].

When adopting geo-replication, the database is replicated in multiple data centers across the world. In general, geo-replicated databases rely on a full replication model, where each data center replicates the complete database. This model's downside appears as the amount of data and the number of data centers increases. In addition to the overhead that this brings in terms of storage (storing some data in all data centers may be unnecessary as some data is only needed in some locations), increasing the number of data centers makes the replication process more complex and expensive (each update needs to be propagated to all other data centers). These problems suggest that in this

case, the system could benefit from partial replication, where each data item is replicated only in a subset of the data centers.

1.2 Problem

The fact that there is a vast number of global distributed web services with low latency, high availability and fault tolerance requirements has led some services to choose to distribute their data centers worldwide so that clients can access the closest data center. Considering the requirements of these systems, it is beneficial to rely on databases that provide weak consistency, thus ensuring low latency and high availability. A form of weak consistency that is often used, as it provides some consistency guarantees to applications while remaining highly available in the presence of faults, is causal consistency. As mentioned in the previous section, partial replication is also considered an attractive feature for this type of system. In this case, each data center only replicates a part of the data, thus avoiding the problems that arise from databases that rely on full replication models [35]. However, with partial replication, several challenges arise, for example, regarding the management of partitioned data and how data is replicated in each replica.

In partial replication, system data is partitioned and distributed, considering that usually, clients will execute transactions on the data that replicated in the closest data center. However, it is necessary to keep in mind that there may be circumstances in which clients intend to carry out transactions that involve objects replicated by another data center. Therefore, it is important to support the execution of transactions over geo-partitioned data [8, 16]. In our work we focus on supporting this execution, while providing causal consistency.

In order to execute transactions over partially replicated data, it is fundamental to address some challenges. First, it is necessary to know what data is kept in each node to access data items involved in a transaction when they are not replicated locally. Second, when adopting a transactional causal consistency semantics [3, 39], it is necessary to efficiently support the semantics properties, namely, supporting reading from a causally consistent snapshot and to guarantee that transaction updates are applied atomically.

1.3 Proposed Solution

This dissertation aimed to design and implement a system capable of executing transactions on partially replicated data. More specifically, we intended to extend a weakly consistent geo-replicated key-value store that supports partial replication to support the execution of transactions that might involve remotely replicated data items. To achieve this goal, we had to create a mechanism that allowed us to know the list of replicas that replicated each data item and design an algorithm that guarantees causal consistency when dealing with transactions involving data replicated in different replicas.

Our work was built on previous work that has developed a weakly consistent geo-replicated key-value store with partial replication (PotionDB). PotionDB allows answering some common queries (those usually performed by the application) regarding global data efficiently by using views, even though the data is partially replicated. Our work extends PotionDB to be able to execute generic transactions that access both local and remote data. In this section, we start by presenting an overview of PotionDB.

1.3.1 PotionDB

PotionDB [35] is a weakly consistent geo-replicated key-value store with partial replication that allows answering queries regarding global data efficiently by using views, even though the data is partially replicated (e.g. the top 10 most sold products in the world of an online store). By using pre-existing partial and non-uniform replication algorithms, PotionDB provides materialized views of data that can be divided across multiple servers. In this way, a client can obtain global information by accessing only one server.

PotionDB is composed of multiple servers. Database objects are replicated in a subset of the servers. New updates are periodically transmitted to replicas asynchronously. Clients connect to the closest server. Updates and queries are performed on a single server and return as soon as their execution completes. Clients can group various operations into a transaction. When executing transactions, PotionDB respects causality and consequently provides causal consistency.

In PotionDB, only objects replicated on the server can be accessed, that is, to access objects not present on a particular server, the client must connect to a replica that holds those objects.

1.3.2 Objectives

PotionDB has limitations for executing remote transactions since it only allows access to objects that are replicated on the server being accessed by the client. Thus, if a client wanted to perform a transaction it would have to connect to a server that replicates all objects that need to be accessed in the transaction. If no such replica existed, it would be impossible to execute the transaction. To overcome this limitation, we have extended PotionDB with an algorithm that allows performing transactions on objects replicated on any server.

To support this feature, we identified the need to address the following two main challenges.

Accessing Remote Objects. For objects not replicated locally, it was necessary to know how we could access this data. In order to send a request to a replica holding the object, it was necessary to know which objects were replicated in which replicas. In this way, we created an object (shared by all servers) that kept the mapping between objects and

replicas. Additionally, we keep information about the state of each replica, which is used when selecting which replicas to contact to fetch the remote objects.

Ensuring Transaction Semantics. The second objective involved ensuring that transaction semantics was respected, in this case transaction causal consistency. This is related to how replicas are updated, and transaction updates are propagated to replicas. To ensure transaction atomicity, we created an algorithm where all transaction updates were saved and propagated as a unit to the necessary replicas. When executing a transaction, the replica first checks that all dependencies are satisfied before executing it. Then, when executing a transaction operation, it is necessary to fetch from a remote replica the objects that are not locally replicated. When a replica requests objects from another replica, the replica sends the transaction snapshot in its request. In this way, the replica obtains the appropriate version of the objects to ensure that the transactions reads are executed in a consistent database snapshot. Updates are stored in the local replica to be applied at commit time. If there is a read in the transaction, the updates are applied temporarily to observe the effects of the updates already performed in that transaction. At commit time, all updates (local and remote) are propagated as a unit to ensure transaction atomicity.

1.3.3 Contributions

The main contributions presented in this dissertation can be summarized as follows:

- Design of a mechanism that provides access to remote objects;
- Design of an algorithm that guarantees the execution of transaction over geo-partitioned adopting transactional causal consitency semantics;
- Design and implementation of the proposed algorithms in a geo-replicated database systems that supports partial replications;
- An evaluation of the proposed algorithms in a simulated geo-replicated setting.

1.3.4 Publications

The work described in this document generated the following communication presented in INForum'21 [23]:

- *Transactional Causal Consistency over Geo-Partitioned Data;*
  Sofia Braz, André Rijo and Nuno Preguiça

1.4 Document Organization

The rest of this document is organized as follows:

Chapter 2 analyses the work related to this dissertation, focusing on consistency models, replication techniques, distributed transaction protocols and ultimately on geo-replicated storages.

Chapter 3 analyses the system design of the implemented solution, starting by describing the system architecture. Then transactional causal consistency is defined, and the challenges of implementing this consistency model are discussed. Finally, the proposed solution is presented together with the developed algorithms to deal with the identified challenges.

Chapter 4 focuses on the implementation details of the proposed solution, namely on details related to the management of updates applied to remote objects and communication and replication techniques.

Chapter 5 presents the experimental evaluation carried out for the developed solution. First, the experimental setup and the configurations used in the evaluation are described, focusing on the benchmark. Afterwards, the results obtained in the experiments are presented and discussed.

Chapter 6 concludes the dissertation and presents possible directions for future work.

2

Related Work

This chapter analyses relevant related work, considering the objectives of the work carried throughout this dissertation. The section 2.1 begins by presenting and discussing the properties of different consistency models that storage systems can adopt. Section 2.2 presents replication techniques, namely techniques used in conflict resolution that can arise between replicas. In the section 2.3, protocols that were developed to execute transactions in geo-replicated systems are studied. Finally, section 2.4 addresses multiple Geo-replicated Storages.

2.1 Consistency Models

Consistency models have a decisive influence on the design of distributed storage systems, defining a set of rules for the execution of operations and for how restrictively we want these operations to occur under concurrency. We refer to more restrictive consistency models as "stronger" and more permissive consistency models as "weaker" [10]. Therefore, there are different consistency models with different characteristics and with different choices for consistency implementations. Depending on the service requirements, we must choose which consistency model is most suitable for the storage system, considering that each consistency model has its own set of trade-offs between performance and consistency. Typically, if we have stronger consistency guarantees, we are sacrificing performance to achieve that consistency level.

Next, we will discuss different consistency models that the storage system can adopt.

Serializability. Serializability is related to a set of objects, and their operations are performed in the context of transactions. A transaction is executed atomically at a certain time, and all other transactions are entirely executed before or after. Serializability ensures that transactions that perform a set of operations on a group of objects appear to be executed sequentially, creating a total order among all transactions in a system. Thus, in a system that provides serializability, each client must see the operations issued to the system in the same order, even if that order does not correspond to the global real-time order in which the operations were issued. In this way, it is possible to keep each replica

state consistent since all replicas appear to be running simultaneously. If this guarantee did not exist, a client could read two different values from different replicas. Serializability is a consistency model designed for databases, and it gives guarantees that prevent any concurrent anomalies in the database [30].

Snapshot Isolation. In many of the most popular databases, the isolation level provided is Snapshot Isolation (SI) [3]. When a transaction reads from a snapshot, the updates are applied to respect a total order. In this isolation level, competing transactions come into conflict if updates are made to the same data items. For two transactions *t1* and *t2*, that are concurrent, *t1* and *t2* are not serializable when *t1* reads a data item written by *t2* and *t2* reads a data item written by *t1*. As in this case, the state of the database within a transaction is a fork from the database state. This phenomenon is called write skew or **short fork**.

Causal consistency. Causal consistency is one of the strongest weak consistency models, providing high availability while achieving the strongest possible consistency. This model requires the system to monitor causal dependencies between operations to ensure that clients see operations in an order that respects causal relationships even though operations diverge between replicas. Operations that are unrelated and therefore have no influence on each other are independent and called concurrent operations. These operations can be presented to users in any order.

Causal+ consistency (CC+). Causal+ consistency [3] has the same guarantees as Causal Consistency, but it requires processes in the system to converge to a single state in the presence of concurrent operations. Causal+ consistency (CC+) ensures that if one update happens before the other, they will be observed in the same order and that the replicas converge to the same state under simultaneous conflicting updates. Many CC+ systems adopt the last-writer-wins rule to ensure state convergence, where the update that occurs "last" overwrites the previous ones. CC+ is the strongest model for individual operations compatible with availability, as it ensures that the causal order of operations is respected.

Parallel SI (PSI) and Non-Monotonic SI (NMSI). Both Parallel Snapshot Isolation (PSI) and Non-Monotonic Snapshot Isolation (NMSI) [3] are weaker isolation levels than SI that allow the **long fork** anomaly, so that for two concurrent transactions *t1* and *t2*, it is possible to commit, in order to write in two different data items. In this way, when two other transactions occur afterwards, one sees the effects of *t1* but not *t2*, and the other sees the effects of *t2* but not *t1*. However, when two transactions attempt to update the same data items simultaneously, both in PSI and NMSI, these transactions are aborted to avoid conflicts created by concurrent writes.

Transactional Causal Consistency (TCC). The Transactional Causal Consistency (TCC) model [3] represents the strongest semantics that a system can achieve under high availability and low latency. Cure's TCC (Section 2.4.2.1) extends CC+ functionality by combining interactive transactions and CRDT support to guarantee that replicas converge. Interactive transactions allow to flexibly combine read and write operations in the same transaction, ensuring that transactions are read from a causally consistent snapshot (view of the data store that includes the effects of all transactions that causally precedes it). Furthermore, when updating multiple objects, a transaction respects atomicity (all updates occur and are made visible simultaneously, or none does) [3]. The TCC, in addition to allowing short and long forks, has a new feature that allows concurrent transactions to update the same data items. This feature is called convergent forks, and concurrent updates are merged using CRDTS.

2.2 Replication Techniques

Replication under Strong Consistency models involves coordinating a replication quorum whenever an operation is performed, leading to very high latency values. Additionally, in the event of a failure, such as network partitions, it may become impossible for some nodes to contact the quorum required to perform operations. For these reasons, many systems rely on weaker consistency models, such as Eventual Consistency or Causal Consistency models. Replication under Eventual Consistency allows any replica to accept updates, propagated asynchronously to other replicas, without remote synchronization. This approach guarantees performance and scalability in large scale distributed systems. By adopting a weak consistency model, these systems allow the states of the replicas to diverge temporarily, requiring a mechanism for merging concurrent updates in a single common state.

Possibly the most well-known reconciliation mechanism is the *last-writer-wins* [13] technique. This technique guarantees reconciliation through timestamps, where the update with the largest physical timestamp value is chosen as the correct version. As a result of this reconciliation mechanism, some of the concurrent updates may be lost since the update with the largest timestamp will "win" over the others.

Conflict-free Replicated Data Types (CRDT) present a principled approach focused on divergence resolution [32]. CRDTs are abstract data types designed to be replicated across multiple nodes. In a CRDT, any replica can be modified without coordination with any other replicas. When any two replicas receive the same set of updates, they will reach the same state, deterministically guaranteeing states' convergence in the replicas. More specifically, the CRDTs' synchronization policies ensure that all replicas will converge to the same state after all updates are propagated, allowing an operation to be executed immediately on any replica, with the replicas being synchronized asynchronously. Thus, CRDTs can guarantee a system with low latency and high availability, even in the presence of network failures.

Regarding the synchronization process of the various replicas, there are two main models:

- State-based synchronization;
- Operation-based synchronization.

2.2.1 State-based synchronization

State-based CRDTs are also called convergent replicated data types (CvRDTs).

In state-based synchronization, when an update is performed on a replica (locally), the state of that replica is changed. Occasionally, replicas are synchronized by establishing synchronization sessions (bidirectional or unidirectional). A replica sends its state to another replica, which will merge the state it received with its state, ensuring that all updates reach all replicas, directly or indirectly. This merge function must be idempotent, commutative and associative. Merge and update functions must increase each replica's internal state according to the partial order of the possible states [32][38].

A simple example of a CDRT is a Counter, whose state-based version is presented in Algorithm 2.1. In this case, since it is a state-based implementation, it is necessary to save the state of the CRDT in two associative arrays, saving in one of the arrays the sum of positive values and in the other the sum of negative values. These arrays maintain a partial result for each replica that reflects the updates submitted by it. Maintaining these arrays is fundamental because it is necessary to keep the most recent value for each entry in the array when merging. By maintaining two arrays, it is possible to infer that the most recent value of an entry is the one that has the highest absolute value since the absolute value of an entry does not decrease. When using a single array, it would not be possible to draw this conclusion since an entry's value could increase and decrease due to an inc execution with a positive or negative value [32].

Algorithm 2.1 State-based Counter CRDT (taken from [32]).

```
payload int[] valP, int[] valN                    ▷ For any id, the initial value is 0
query value () : int
    return Σ_r valP[r] + Σ_i valN[r]
update inc (int n)
    let id := repId()                             ▷ repId : generates the local replica id
    if n > 0 then
        valP[id] := valP[id] + n
    else
        valN[id] := valN[id] + n
merge(X, Y) : payload Z
    for r ∈ X.valP.keys ∪ X.valN.keys ∪ Y.valP.keys ∪ Y.valN.keys do
        Z.valP[r] := max(X.valP[r], Y.valP[r])
        Z.valN[r] := min(X.valN[r], Y.valN[r])
```

2.2.2 Operation-based synchronization

Operation-based CRDTs are also called commutative replicated data types (CmRDTs).

In operation-based synchronization, when an update is received on a replica, it is applied to the replica state. Then replicas converge by propagating the updates to the remaining replicas. It is necessary to ensure that each update is delivered reliably to all replicas and some types of CRDTs require that updates be delivered in a specific order, typically in the causal order. In CmRDTs, all concurrent operations must be commutative. Besides, it is necessary that updates are received not only in order but only once [32][38].

The operation-based version of a Counter, presented in Algorithm 2.2, is much simpler than the one presented in the previous section since the state is just an integer with the counter's current value. In this implementation, a value that can take positive and negative values is received as a parameter in the increment update. This value is added to the counter value in the effector operation [32].

Algorithm 2.2 Operation-based Counter CRDT (taken from [32]).

```
payload int val                                  ▷ Initial value: 0
query value () : int
    return val
update inc
    generator (int n)
        return (inc, [n])                        ▷ Operation name and parameters for effector
    effector (int n)
        val := val + n
```

2.2.3 Non-uniform Replication

Partial replication is the typical approach to deal with the increase of information since it dictates that each replica keeps only a part of the system's full information. In this way, a remote replica may have to be contacted to calculate the response to a query since each replica can only locally process a subset of the database queries, leading to high latency costs, especially when talking about geo-replicated systems. The non-uniform replication model [9] represents an alternative to the partial replication model from the perspective that despite the similarity of the partial replication model keeping only parts of the system data, it can execute all queries. Therefore, in this model, the central intuition is that not all data is necessary to return the result obtained from the execution of reading operations.

When used to solve realistic problems, it is possible to verify that this type of replication is more efficient than traditional, using less storage space and network bandwidth. The non-uniform replication model is based on eventual consistency and CRDTs, being formalized for an operation-based replication approach [9].

There are two typical examples of data that benefit from non-uniform replication, the first being a **top-K** elements object, where each replica needs to keep those top-K elements. A top-K object can be used, for example, to keep a leaderboard in an online game. In this case, whenever updates are made to a replica that does not affect the leaderboard, there is no need to propagate that update to the other replicas in a non-uniform replicated system since the query executed in any replica continues to return the same correct leaderboard. Another example of a non-uniform CRDT, the **Top Sum**, maintains the top-K elements, mentioned above, that have been added to an object. However, in this case, each element represents the sum of the values added to that element.

Examples of using this type of data are: an online game in which whenever a player completes a level, points are awarded, and therefore the leaderboard can be changed; an online store where it is necessary to maintain a list of the best-selling products; etc. In this case, it is more challenging to understand which operations impact the observable state and, therefore, must be propagated to other replicas. One approach would be to propagate all operations, which is what non-uniform replication aims to avoid. Another approach is minimizing the number of propagated operations by calculating, for each id that does not belong on top, how much should be added to the id to make it to the top. Then, only if the sum of local adds exceeds this value, the operations will be propagated to the other replicas because only then the top is affected [9].

2.3 Distributed Transaction Protocols

A transaction is a unit of work that groups several operations and executes them in a database, providing certain guarantees. Relational databases typically implement transactions that provide ACID properties [12, 21]:

- **Atomicity:** states that all operations in the transaction are performed, or none are, treating each transaction as an atomic unit;
- **Consistency:** if the database was in a consistent state before the execution of a transaction, after its execution, it remains in a consistent state;
- **Isolation:** a transaction must be executed without affecting the other transactions that are executed in parallel so that all transactions are executed as if they were the only transaction in the system;
- **Durability:** even in the presence of failures, the effects of operations must persist in the database

By offering ACID properties, the databases consequently provide strong consistency, which results in decreased fault tolerance and may limit performance. Furthermore, in distributed databases, transactions have the particularity of performing operations on

data located in several places. This implies that providing ACID properties in a geo-distributed scenario implies that there is also an increase in latency due to the synchronization that has to exist between data centers [35, 21].

Since our work aims to design and implement the best solution for a system capable of executing transactions on partially replicated data, it is relevant to address, throughout this section, previous works that explore the problem of executing transactions in distributed systems.

2.3.1 Clock-SI

Clock-SI [15] is a protocol that implements snapshot isolation (SI) for partitioned data stores.

Before Clock-SI, existing SI implementations for partitioned data stores implied a centralized authority. In these cases, to assign snapshot and commit timestamps to transactions, it was necessary to request them from the centralized authority. The disadvantage of this approach is that it had a single point of failure and a potential bottleneck, resulting in low availability and high latency.

In Clock-SI, to avoid resorting to a centralized authority, snapshot and commit timestamps are assigned to transactions through loosely synchronized clocks. More precisely, the local clock value is assigned to the snapshot's timestamp when the transaction starts. The same happens with a local update transaction's commit timestamp, obtained by reading the local clock.

Like conventional SI, each partition does its certification for update transactions. To commit transactions that update multiple partitions is used a two-phase commit (2PC) protocol.

Using loosely synchronized clocks can lead to clock skews and pending commits. These factors may cause a transaction to receive a timestamp snapshot for which the corresponding snapshot is not yet completely available. Clock-SI's approach to solving this problem is to delay operations that access the unavailable part until it becomes available.

Clock-SI's read and commit protocol are essential for its performance improvement. The **read protocol** consists of two major aspects:

- Assign snapshot timestamps;
- Delay read operations until the requested snapshot becomes available in the partition being accessing, ensuring that transactions access consistent snapshots. A read in a partition gets delayed if the partition current clock is smaller than the transaction snapshot or there is a pending transaction for which the commit timestamp is now known.

For both cases where read operations are delayed, an operation waits only until a commit operation completes or a clock catches up. Under the same condition as the read

request, a remote partition update request is also delayed. This way, it is ensured that the commit timestamp of an update is always higher than the timestamp snapshot.

In Clock-SI, a read-only transaction reads from its snapshot and commits without requiring additional checks, even if it reads from multiple partitions.

For read-write transactions, the **commit protocol** has two distinct behaviours regarding update transactions, depending on whether the transaction modifies only one or not:

- **Single-partition update transaction:** In this case, the transaction commits at that partition. Clock-SI starts by certifying the transaction being committed, verifying its updates with concurrent committed transactions. Next, the transaction state changes from *active* to *committing*, and the partition reads its clock to assign the commit timestamp to the transaction. Finally, after writing the commit record to stable storage, the transaction changes its state from *committing* to *committed*, making its effects visible.

- **Distributed update transaction:** This case uses a 2PC protocol and a transaction coordinator to commit or abort the transaction at the updated partitions. This coordinator runs on the source partition. First, it is necessary to certify, locally, all partitions that performed updates for the transaction. Afterwards, each participant changes its state from *active* to *prepared* and sends a prepare message with prepare timestamp to the coordinator and waits for the response. The 2PC coordinator chooses the commit timestamp: the highest prepare timestamp of all participants. Finally, the update transaction effects will be visible for transactions with snapshot timestamps greater than its commit timestamp.

Clock-SI improves system availability and performance since there is no longer a single point of failure and a potential bottleneck.

2.3.2 Walter

Walter [40] is a key-value store that supports transactions for geo-replicated systems. Walter adopts Parallel Snapshot Isolation (PSI), which provides low latency when replicating data asynchronously and offers strong guarantees within each site.

In web applications, clients typically access the closest sites, where it should be possible to observe a consistent data state. Therefore, the system must provide strong consistency between hosts on the same site. Across sites, weaker consistency is used since it is acceptable to have a slight delay for a client's updates to be seen by others and because replication between sites should be asynchronous to ensure low latency. Within a site, PSI ensures a common order of transactions and that transactions access a consistent snapshot. Across different sites, PSI supports a causal ordering of transactions, allowing transactions to be replicated asynchronously between sites.

Walter uses *preferred sites* and *counting sets* to avoid conflicts that arise from concurrent writes when implementing PSI.

Preferred Sites. Considering that in web applications, objects are usually updated by the user who owns the object on the site where he logs in, each object is assigned to a *preferred site*. On *preferred sites*, objects are updated more efficiently because updates can be committed without checking other sites for concurrent writes. Objects can be updated at any site, although Walter executes and commits a transaction faster if executed at the *preferred site* of all objects modified by the transaction. For example, in a web application, the *preferred site* for a user's post wall is closest to the user. However, using *preferred sites* is not always enough. For example, a user's friends list can be updated by users on multiple sites. In order to avoid conflicts, in this case, the system uses a data type called *counting set (cset)*.

Conflict-free counting set objects. If an object is updated frequently by multiple sites, it is natural that it is not evident to which *preferred site* the object is assigned. So, to solve this problem, Walter resorts to *counting set (cset)* objects. A *cset* is similar to a set, but in this case, each element has a count, and unlike sets, the operations are commutative not to create conflicts. In this way, through *csets*, transactions can be committed without verifying if transactions between sites produce conflicts. Even if several transactions access a *cset* object, write conflicts are not generated. *Cset* supports the *add(x)* operation to add element x, incrementing the counter of x and the *rem(x)* operation to remove element x and decrement its counter. This way, *cset* will map the element ids to the counts. This mapping indicates how many times an object appears in the *cset*, which, if the counter is used directly, can be helpful, for example, to find out the number of items in a shopping cart. If the user count is hidden, *cset* can be used, for example, to keep a list of friends. In this case, although the count has no meaning for the user, it is used to keep the list of elements to zero or one - the system should not add an element to the *cset* if the element is already there (counter to 1) nor remove if the element if it is not there (counter to 0).

Walter supports transactions for geo-replicated web applications, providing the following properties:

- **Asynchronous replication between sites:** transactions are replicated asynchronously to reduce latency;
- **Efficient update for particular objects:** objects can be modified efficiently on their preferred site, and counting sets can be modified efficiently on any site;
- **Freedom from conflict-resolution logic:** the use of preferred sites and counting sets prevents conflicts between sites from arising;
- **Strong isolation within each site:** the system's adoption of the PSI consistency model provides this property.

2.3.3 P-Store

P-Store [37] is a partially replicated key-value store for wide area networks that supports transparent transactional access. In P-Store, the multiple sites are clustered in *groups*. As P-Store adopts a partial replication model, each site stores a subset of the application data. Sites that belong to the same *group* replicate the same dataset, and multiple sites from different groups can replicate a specific data item. P-Store tries to minimize the cost of communication across *groups*, which is assumed to be slow.

P-Store executes each transaction on one or more sites, and each transaction needs to be certified to ensure execution serialization. The certification process involves only the sites that replicate the data items involved in the transactions to be certified. P-Store uses an atomic multicast service to guarantee correctness. Only one message is multicasted to certify a transaction, and there is no multicast of messages during its execution. If the execution of a transaction is distributed, it is necessary to resort to an additional voting phase.

Figure 2.1 illustrates the execution of a global transaction, which reads data items from groups *g1* and *g2*. After the requests are executed, the transaction is submitted to the certification protocol.

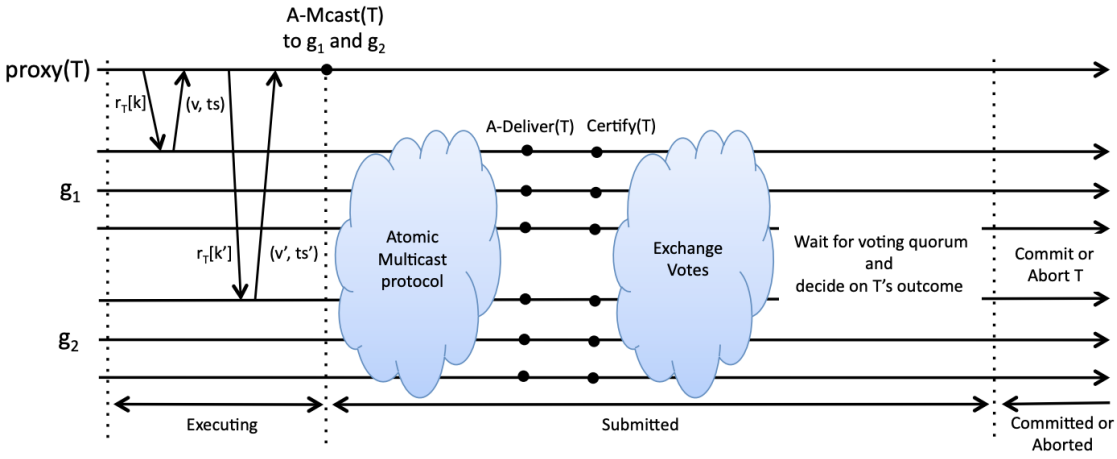


Figure 2.1: The execution and certification of a global transaction (taken from [37]).

When a transaction is submitted for certification, the transaction is multicasted to all groups that replicate at least one data item read or written by the transaction. Each of the sites checks whether the values read are up-to-date. If the version stored on the website is the same as the one read in the transaction, the transaction passes the certification. However, as this is a partially replicated context, a site can only store a subset of the data items read by the transaction and therefore does not have enough information to decide the transaction's outcome. Thus, a voting phase was added. Each site that replicates at least one data item read by the transaction sends its certification result to each site that replicates at least one data item written by the transaction. When this site receives

a voting quorum for the transaction, it can decide its outcome. A transaction can be committed when all of its voting quorum sites vote in favour. Otherwise, an old value has been read, and the transaction must be aborted. In this way, it is possible to guarantee the serializability of the execution. In order to optimize the transaction certification process, the system seeks to certify non-conflicting transactions in parallel.

2.3.4 SCORe

SCORe [31] is a scalable one-copy serializable partial replication protocol that guarantees that:

- Only the replicas that store the data accessed by a transaction are involved in its execution, being, therefore, a genuine protocol;
- Read operations consistently access consistent snapshots. More specifically, the protocol adopts a one-copy serializable multi-version scheme that never aborts read-only transactions and allows them to bypass any distributed validation phase, allowing the execution of massive workloads of reads efficiently.

SCORe adopts an asynchronous distributed system model composed of several nodes that represent transactional processes. The data maintained by the nodes assume a key-value model, where each data item stored by a node is represented as a sequence of versions. Each data item is represented as follows:

(key, value, version)

where *key* represents the data item, *value* is its value, and *version* is the timestamp scalar used to identify and order the versions of the data item. The data set is split across multiple partitions, and a set of nodes replicates each partition.

SCORe provides strong consistency guarantees by adopting One-Copy Serializability [7]. SCORe adopts a *genuine* partial replication scheme that ensures read-only transactions get a consistent snapshot of the data without resorting to expensive remote validation protocols. To do this, SCORe resorts to a local multi-version concurrency control algorithm and a highly scalable distributed logical-clock synchronization scheme. This synchronization scheme only requires exchanging a clock value between the nodes involved in executing the transaction.

SCORe features a distributed timestamp management scheme that seeks to solve the following issues:

- Determine the snapshot visible by transactions, choosing the version that a transaction should see after a read operation;
- Define the global serialization order for update transactions through a distributed agreement protocol during the transactions' commit phase.

To achieve this, each node in the SCORe keeps two variables:

- ***commitId***: timestamp assigned to the last update transaction committed on the node;
- ***nextId:*** timestamp the node will propose the next time it receives a commit request.

To determine the snapshot visibility for transactions, SCORe associates a timestamp scalar named *snapshot identifier* to each transaction. The *snapshot identifier* is initialized after its first read operation, adopting the *commitId* value. If the read is local, the chosen value of *commitId* is the local node. Otherwise, it determines the highest value between the *commitId* of the local node and the *commitId* of the remote node read by the transaction. That way, any succeeding read will observe the most recent committed version of the requested data with a timestamp less than or equal to the snapshot identifier.

SCORe employs a genuine atomic commit protocol that uses the Two-Phase Commit (2PC) algorithm to confirm update transactions and ensure application atomicity. In connection with the 2PC, it implements a distributed agreement protocol that allows:

- Total ordering, among all nodes that replicate a partition, of the commits of transactions that update data items for that partition;
- Track the serialization order of update transactions that manifest data dependency, setting the total order from the *commit timestamp*.

To serialize transactions and track write-after-read dependencies, SCORe updates a node's *nextId* after performing a read. Consequently, SCORe guarantees that when a transaction issues a commit on a node, it is assigned a commit timestamp greater than the timestamp of any transaction that has read values from that node before.

Lastly, a transaction uses the *snapshot identifier* assigned to it after the first read, which ensures that the snapshot read by a transaction is always consistent. This way, read-only transactions are never aborted and avoid any distributed validation.

2.3.5 RAMP Transactions

Bailis et. al [5] proposed algorithms for Read Atomic Multi-Partition (RAMP) transactions. Read Atomic is an isolation level that guarantees atomic visibility, ensuring that other transactions see all or none of a transaction's updates. RAMP transactions guarantee:

- ***synchronization independence:*** a client's transaction cannot cause another client's transactions to block or fail. This property is vital to ensure that each client can progress even in the presence of partial failures. In the absence of failures, this property it is essential to maximize concurrency;
- ***partition independence:*** clients do not need to contact partitions that are not accessed by transactions. This property implies that the load on servers not involved

in executing the transaction will be reduced. Additionally, when a partition fails, only transactions that access data items belonging to that partition are affected.

Through these properties, RAMP transactions ensure guaranteed completion, limited coordination between partitions, and horizontal scalability when accessing multiple partitions.

RAMP transactions control the visibility of updates without decreasing concurrency. In the case of concurrent reads and writes, RAMP transactions can detect non-atomic (partial) reads and restore them, when necessary, by communicating with the servers again. To do this, RAMP transactions append metadata on each write and use multiple versions to avoid blocking read operations. The algorithms proposed offer a trade-off between the size of the attached metadata and the system's performance.

RAMP-Fast. RAMP transactions support the concurrent execution of reads and writes, which provides good performance. However, it can cause a transaction to read only a subset of the writes of another transaction, resulting in a race condition. For writes, RAMP-Fast (RAMP-F) requires two round trip time delays (RTTs), one for `PREPARE` and the other for `COMMIT`. RAMP-F only needs one (RTT) for reads if there are no concurrent writes and two otherwise. RAMP-F stores metadata in the form of write sets, which implies that transactions need a metadata size that is linear in the transaction's write size.

In RAMP-F, metadata is used by writers as a record of the writes they intend to perform. Thus, a reader can detect if its operation and an ongoing commit give rise to a race condition and, in this case, use the writer's metadata to get the missing data. Therefore, a reader only performs the second round of reads if it reads from a partially committed transaction. The two RTTs required in the case of writes ensure that readers never need to wait for a write that has not yet reached a partition. If a transaction commits a write, all corresponding writes are present in their respective partitions. Besides, even if they are written, they are possibly not locally committed. In RAMP-F, a timestamp is associated with each write, identifying the transaction and the set of items written by it. Assuming that a reader identifies the corresponding version through the timestamp, the reader can get the version from the corresponding set of pending writes without pausing the read.

RAMP-Small. In RAMP-Small (RAMP-S), metadata size is constant, and for reads, it requires two RTTs. Writes are the same as RAMP-F but with the particularity that writers only store the transaction timestamp. In RAMP-S, readers perform the first round of reads to get the highest timestamp of each item in their partition. In the second round, the readers send the set of timestamps obtained to the partitions. For each item in the request, the servers return the item version with the highest timestamp. Consequently, readers can only return items after the second round.

In RAMP-S, if a transaction is partially committed and a read transaction concurrently accesses the committed items, the transaction timestamp is returned in the first round.

In the second round, all uncommitted partitions find the committed timestamp in the set provided by the reader and return the correct version. Therefore, in RAMP-S, readers use the provided set of timestamps to decide which value to return to the partition.

RAMP-Hybrid. RAMP-Hybrid (RAMP-H) is an intermediate solution of RAMP-F and RAMP-S, but in this case, the writers maintain a Bloom filter [6] with the transaction's write set. Like RAMP-F, in RAMP-H, the readers perform a first communication round to get the last committed version of every item from its partition. Thus, RAMP-H obtains a list of potentials writes with higher timestamps. Then, in the second round, it will look for the likely missing versions.

RAMP-H behaves as an intermediate solution of the other two algorithms because if the Bloom filter does not present false positives, its reads are performed as in RAMP-F. Otherwise, they are performed as in RAMP-S. Thus, the number of second rounds is equivalent to the number of false positives in the Bloom filter.

2.3.6 PaRiS

PaRiS [41] is a Transactional Causal Consistency (TCC) system that implements non-blocking parallel reads and supports partial replication. Executing parallel non-blocking reads proves to be especially relevant to ensure good performance in intensive reads applications. However, it should be borne in mind that in a partially replicated environment, within a transaction, multiple reads can be performed in parallel by different data centers, which are unaware of the transactions executed in other data centers, which can lead to consistency violations.

PaRiS uses Universal Stable Time (UST) to track dependencies and hence deal with this problem. Through a lightweight gossip process, UTS identifies the snapshot that all data centers in the system have established. Additionally, PaRiS offers each client a private cache where they store their updates that are not yet reflected in the snapshot identified by the UST. This cache allows the system to achieve TCC even if clients have access to a snapshot of data that is slightly delayed. Consequently, transactions can consistently read from the snapshot on any server without having to block. Furthermore, PaRiS obtains resource efficiency and scalability by using only one timestamp to trace dependencies and define transactional snapshots.

Transactions. Key versions and snapshots are identified through a scalar timestamp. This timestamp scalar, together with the contents of the client cache, defines the snapshot observed by the transaction. In addition, each update transaction is assigned a commit timestamp that reflects causality. This timestamp is determined using a 2PC protocol.

Non-blocking reads. To identify the snapshot that each data center took, PaRiS defines a stable snapshot. A stable snapshot with timestamp *ts* stores all versions with a timestamp

$\leq ts$ and indicates that all transactions that have a commit timestamp $\leq ts$ have been applied across all data centers that replicate the keys written by those transactions. This allows transactions to read from a stable snapshot without blocking, regardless of the data center where reads are performed.

When a transaction starts, the coordinator partition assigns a stable snapshot to the transaction. The coordinator of a transaction can be any node, and the coordinator ensures that snapshots assigned to transactions from the same client progress monotonically.

Universal Stable Time protocol. Each partition maintains a version vector with the timestamps of the last transactions applied globally. Through a gossip protocol, the multiple partitions share the smallest timestamp of their versions vectors. Aggregating the shared timestamps creates the stable snapshot that will be assigned to the transactions when they start.

Through a single timestamp, the UST protocol defines stable causally consistent snapshots, increasing the scalability and efficiency of the system by reducing the overhead of communication involving partitions.

Cache. The fact that the commit timestamp of a transaction is higher than its stable snapshot can lead to the commit timestamp being higher than the snapshot assigned by the next transaction of the same client. It implies that this snapshot would not include the updates made by the client in the previous transaction, which violates the Read-Your-Writes property required in causal consistency. To deal with this issue, PaRiS maintains the versions updated by the client in its cache until it receives a snapshot timestamp that allows it to remove the versions with less than or equal timestamps. So when a client wants to read a key, it first checks if there is a version in cache. If not, the client issues a read request to the replica. Either alternative allows to perform reads without blocking.

In conclusion, to ensure good system performance, PaRiS presents transactions with a view of data that may be slightly delayed. As a result, PaRiS provides low latency and high storage capacity while adopting transactional causal consistency semantics.

In our solution we decided to use as the transaction snapshot the latest snapshot installed in the local replica. This decision came after our initial experiments that have shown that when accessing remote replicas, the reads were getting blocked due to the late delivery of transactions from the replica making the request. This lead us to develop a solution to overcome this situation. Additionally, the solution proposed in PaRiS has the disadvantage that if a client closes a session and restarts it again, it might miss updates performed in its old session.

2.4 Geo-replicated Storages

Designers of distributed storage systems always try to balance the system they are designing to guarantee a strong consistency model, offering the best data consistency without

compromising other factors such as latency, availability, scalability, and performance. However, although addressing most of these factors is a priority, it may be impossible to provide all of them simultaneously [18].

Often, it is necessary to make decisions regarding the type of storage to be used, which are based on the CAP theorem. It is usually possible to choose between "CP" databases that provide strong guarantees, but that turns out to be slow, expensive, and unavailable under partition and "AP" databases that end up compromising data consistency despite being fast and available [3]. Thus, according to the CAP theorem, there cannot be a system that offers perfect availability and strong consistency in the presence of partitions, which are inevitable in large-scale systems such as geo-replicated systems. Therefore, generally, distributed applications have to choose between consistency and availability.

In this section, we take a closer look at different geo-replicated storages, exploring their properties and concepts.

2.4.1 Amazon Dynamo

Dynamo is a system developed by Amazon to address the need for a highly available and highly scalable key-value storage system for some of Amazon's main services that only need primary-key access to a data store. The Amazon platform provides services for websites worldwide, which implies that it is implemented on thousands of machines present in various data centers around the world. Therefore, one of Amazon's main goals with geo-replication was to have high availability and performance. To ensure high availability and low latency, Dynamo sacrifices consistency in certain failure scenarios. This means that Dynamo was designed to be an eventually consistent data store, thus allowing concurrent writes. [13]

To fulfil latency requirements, during the development of Dynamo, it was necessary to avoid resorting to the design adopted by several distributed hash table systems, which involve routing operations through several nodes (such as Chord [42], or Pastry [36]). Therefore, Dynamo can be called a zero-hop DHT, considering that each node contains enough routing information locally for an operation to be forwarded directly to the correct node. As a result, latency with this approach decreases compared to when using multi-hop routing systems.

2.4.1.1 System architecture

Dynamo data model can be seen as a map, where a key has an associated value object. The system provides only two operations, the *get(key)* operation and the *put(key, context, value)* operation. The *get(key)* operation returns a single value or several values together with a context, in case versions are the result from conflicts. The *put(key, context, value)* operation, sets the new value for key, with context being used to detect concurrent updates.

The Dynamo's partitioning scheme is based on consistent hashing to dynamically distribute partitioned data across multiple storage hosts, making it possible to scale the system quickly. On consistent hashing, all nodes in the system have a position in a ring represented by the output range of a hash function. Each ring node is assigned to a random value within the ring range that represents its position. Each data item is associated with a node by hashing the data item's key. Through the hash operation of the data item's key, the data item's position in the ring is produced, and the first node with a larger hash than the calculated is searched (in the clockwise direction). This way, each node is only responsible for the region between it and its predecessor in the ring, as shown in Figure 2.2.

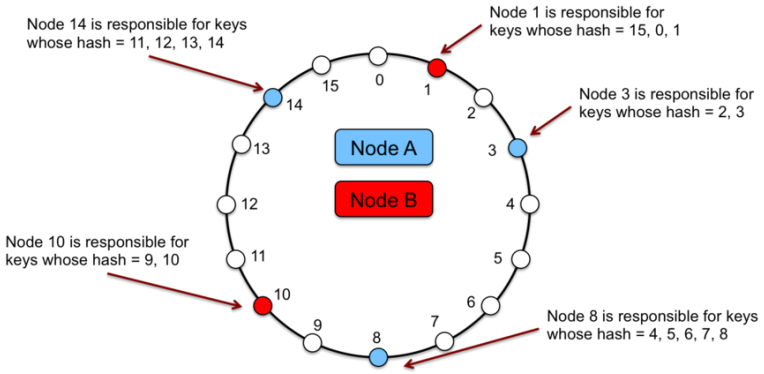


Figure 2.2: Virtual nodes in Dynamo (taken from [14]).

Dynamo adopted the concept of virtual nodes, where each node can be responsible for several virtual nodes.

2.4.1.2 Replication

Dynamo replicates each data item in *N* (parameter defined by instance) hosts, achieving high availability and durability. Each coordinator has a set of keys assigned, corresponding to the data items that fall within the coordinator range and for which the coordinator is responsible for replication. The coordinator replicates the keys on the *N-1* successor nodes in a clockwise direction on the ring, resulting in a system in which each node is in charge of the region between it and its *Nth* predecessor. In Figure 2.3, it is possible to observe the behaviour of the system: node B, in addition to storing it locally, replicates key K in C and D; node D will store the keys corresponding to the intervals (A, B], (B, C] and (C, D].

2.4.1.3 Eventual consistency and vector clocks

The eventual consistency model provided by Dynamo causes the update operations to be propagated to all replicas asynchronously. Under this model, a *put()* call can return a value to the caller before the update is propagated to all replicas, resulting in scenarios

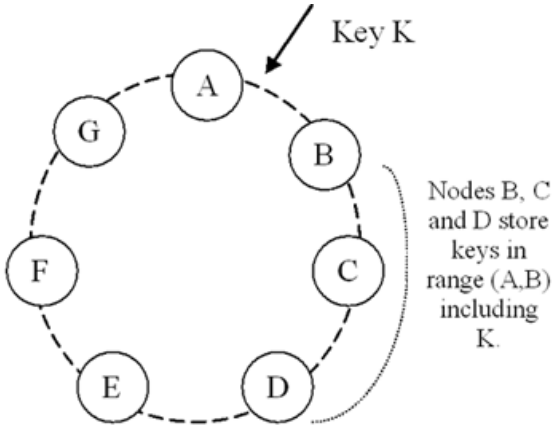


Figure 2.3: Partitioning and replication of keys in Dynamo ring (taken from [13]).

in which *get()* operations performed later can return values from replicas that do not yet have the updates performed recently. There may be updates that do not reach the replicas for an extended period when there are failures. In typical cases, there is a limit on the propagation times of the updates.

Amazon has a category of applications that can tolerate the mentioned inconsistencies and be built to run under these conditions. For each update that is performed, Dynamo treats the result as a new and immutable version of the data, allowing multiple versions of a value to exist in the system simultaneously.

New versions usually include the previous versions and, the system automatically determines the official version. This operation is called syntactic reconciliation. However, simultaneous updates in the presence of combined failures can result in conflicting versions of an object. Reconciling the multiple branches into a single one must be performed by the client through semantic reconciliation. Some failure models, such as updates when network partitions and nodes fail, lead to the system having sub-histories of different versions of the same data, which will have to be reconciled. Therefore, to obtain causality between the various versions of an object, Dynamo uses vector clocks.

Through a vector clock, it is possible to determine if two versions of the same object are in parallel branches or if they have some causal order since the vector clock is associated with each version of each object through a list of pairs *(node, counter)*. Meaning that if the counters on the first object's clock are smaller or equal to all the nodes on the second clock, the first is a previous version of the second one and can be forgotten. If not, the two changes are considered to be in conflict and require reconciliation [13].

The version of an object is specified by the client whenever it is intended to update it, through the context, which is one of the arguments of the *put()* operation that contains the vector clock information through *get()* operations made previously. The use of vector clocks can be demonstrated in Figure 2.4.

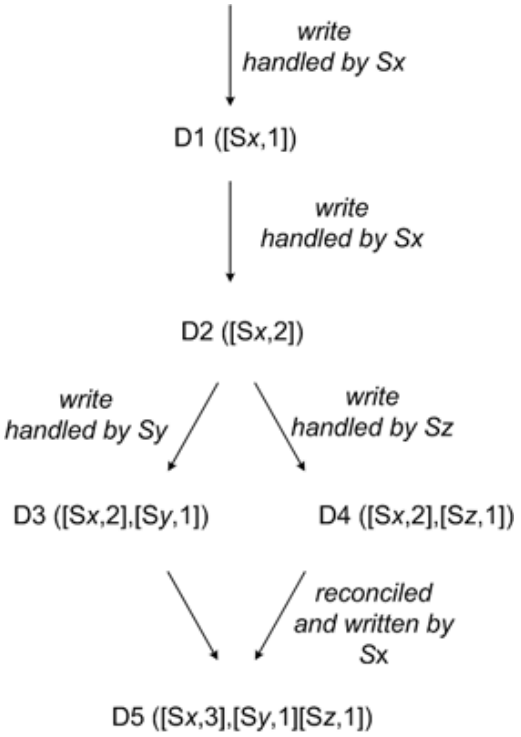


Figure 2.4: Version evolution of an object over time (taken from [13]).

2.4.2 AntidoteDB

AntidoteDB is a highly available geo-replicated key-value database where operations are based on the synchronization-free execution principle using Conflict-free replicated data types (CRDTs) [20].

In AntidoteDB, data is partitioned between servers in a cluster using consistent hashing, thus being organized in a ring. Read and write requests are answered by the server that keeps a copy of the data. When performing a write or read operation on several objects, a transaction contacts only the servers with the objects accessed by the transaction. In this way, it is allowed to execute requests even when some servers fail [29].

In Figure 2.5, it is possible to observe the four components present in each AntidoteDB partition:

- **Log:** In this component, a log-based persistent layer is implemented, thus allowing the updates to be saved in a log, which is persisted to disk. This component also has a cache layer to allow faster access to the log. The log is used to load the state of objects in the Materializer (on restart), resend lost updates to external data centers, and read an older version of an object available in memory [27].

- **Materializer:** This component generates and caches the various versions of objects requested by clients. Materializer is connected to Log and Transaction Manager. Materializer caches materialized versions of keys in memory, having two types of

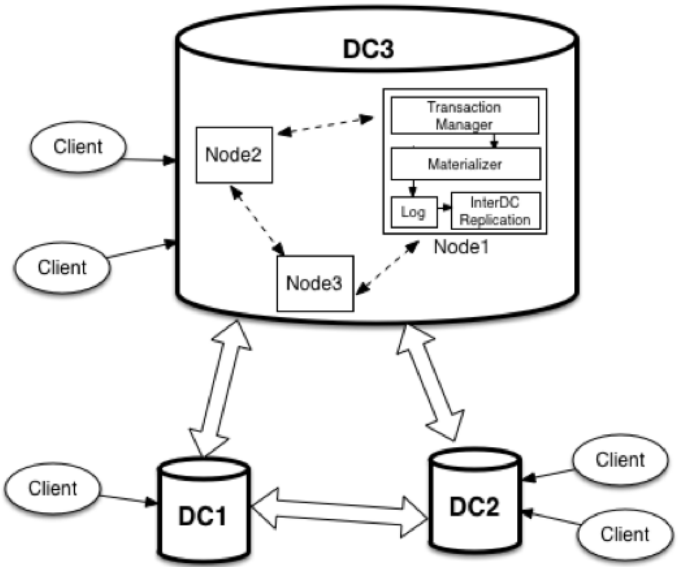


Figure 2.5: AntidoteDB architecture (taken from [29]).

cache for each key: a cache operation in which the most recent operations are stored and a snapshot cache where the last materialized views of a CRDT are kept [28].

- **Transaction Manager:** This component is responsible for implementing the transaction protocol. The Transaction Manager receives clients requests, executes and coordinates transactions and then responds to clients [29].
- **InterDC Replication:** InterDC Replication is the component used to propagate updates fetched from the log to other data centers, through partition-wise communication [29].

In order to replicate updates from one cluster to another, AntidoteDB uses **Cure**. This highly scalable protocol provides the strongest consistency model compatible with high availability, causal consistency.

2.4.2.1 Cure

Cure is a protocol that guarantees causal consistency, atomicity and support for high-level data types with a safe resolution of concurrent updates. Cure can achieve scalability similar to that in eventually consistent NoSQL databases while offering stronger guarantees simultaneously [3]. Cure supports:

- **Causal+ consistency (CC+)**;
- **Support for high-level replicated data types (CRDTs)** as counters, sets and tables with intuitive semantics and assured convergence even in the presence of conflicting updates and partial failures [3];

- **Transactions** ensuring that multiple keys are read and written consistently and interactively. Executing several operations in a transaction allows the application to maintain relations between several objects or keys. Cure introduced the Transactional Causal Consistency (TCC), where all transactions provide readings of a snapshot and atomicity of the updates [3].

In a geo-replicated key-value store that handles a large number of objects, the complete set of objects is replicated in different data centers (DC) in order to guarantee high availability and low latency. Each of these DCs has N partitions, and each of these partitions stores a non-overlapping subset of the keyspace. So that in the design of the Cure it was possible to include Transactional Causal Consistency without compromising availability while achieving high scalability, the developed protocol adopts three major design decisions [3]:

- It depends on the timestamps of the events in order to be able to encode the causal dependencies that allow partitions to make decisions locally. In this way, it is possible to avoid dependency verification messages, which generally penalize performance.
- Does not depend on centrally assigned timestamps.
- Separates the problem of propagation of updates between replicas from making updates visible, thus allowing partitions to propagate the updates without coordinating these updates with other partitions. A lightweight protocol involving all partitions is executed asynchronously to establish the set of updated transactions. That is, all causal dependencies are known in a given data center.

2.4.3 Spanner

Spanner [11] is a scalable, globally-distributed and synchronously-replicated database developed by Google. Spanner was the first system to distribute data globally and to support externally-consistent distributed transactions. This system supports general-purpose transactions and provides an SQL-based query language. Spanner uses the Paxos [24] algorithm to shard data across many state machines (replicas) in data centers worldwide. In Spanner, read-only and read-write replicas keep a complete copy of the data, implying that the execution of queries is simple [34]. When there is a complete copy of the data in each replica (full replication), it means that any replica has an overview of the system. Therefore, partial replication problems do not arise (e.g. execution of a query that involves data not present in the replica).

Replication is used to ensure high global availability and low latency, ensuring that users can access another replica within a relatively short distance when a replica fails. As the amount of data or the number of servers changes, Spanner will automatically reshard and migrate data between machines to balance the load and respond to failures. Spanner

was designed to scale up to millions of machines in hundreds of data centers, allowing applications to use it for high availability when replicating their data in data centers within or across continents.

Spanner allows applications to dynamically control data replication settings, specifying restrictions that establish which data centers contain which data and how far the data is from its users, thus controlling the latency of reads. Applications can also specify the distance between replicas to control the writes' latency and how many replicas are maintained to control durability, availability and reads performance. For a balanced distribution of resources between data centers, data can be moved dynamically and transparently between data centers.

Spanner provides external consistency [17] guarantees (read and write) and globally consistent reads across the database at a given timestamp. The fact that Spanner supports these features also enables at global scale support for consistent backups, consistent MapReduce executions and atomic schema updates, even in the presence of ongoing transactions. Spanner achieves these features by assigning globally-meaningful commit timestamps to transactions even when distributed to reflect the transaction's serialization order. The TrueTime API [11] and its implementation are responsible for providing these properties. The API directly exposes Spanner characteristics that depend on the implementation's limits, such as clock uncertainty and timestamp guarantees. If the uncertainty is large, Spanner will have to slow down to wait for that uncertainty to pass.

2.4.4 Azure Cosmos DB

Azure Cosmos DB [1] is a globally distributed, horizontally partitioned, multi-model database service developed by Microsoft. This service was designed to offer clients easy to scale throughput and storage across any number of regions worldwide. Azure Cosmos DB offers comprehensive Service Level Agreements (SLAs) for throughput, latency at the 99th percentile, availability, and consistency, these being the four dimensions most appreciated by clients.

Concerning the database API, Azure Cosmos DB supports multiple data models. Among others, relation, graph and document data models are provided. It is possible to support all these APIs in the same database since the core type system of Azure Cosmos DB's database engine is atom-record-sequence (ARS) based. Atoms consist of a small set of primitive types such as strings, booleans and numbers. Records are structs (pairs key, value), and sequences are arrays of atoms, records or sequences. The database engine translates and projects the data models on the ARS based data model.

All data in an Azure Cosmos DB container, such as collections, tables, and graphs, is horizontally partitioned and managed transparently by resource partitions. A resource partition is a consistent and highly available container of data partitioned by a client-specified partition-key, which provides a fundamental scalability and distribution unit. In Azure Cosmos DB, the client can scale elasticly throughput based on application traffic

patterns in different geographic regions to support workloads that vary by geography and time. Partitions are managed without compromising the availability, consistency, latency or throughput of a container. Azure Cosmos DB provides a single system image of the globally-distributed resources, regardless of scale, distribution or failures. Developed by Microsoft, this service transparently performs partition management operations across all the regions when clients elastically scale throughput or storage.

Like Spanner, analyzed in the previous section, in CosmosDB, each data center maintains a complete database replica. In this way, performing queries will be simple without the challenges of partial replication being raised.

In Azure Cosmos DB, developers can choose from five consistency models defined along the consistency spectrum, selecting the most suitable consistency model for each database/operation. The consistency models that can be chosen are strong, bounded staleness, session, consistent prefix and eventual. A default consistency level can be configured in the database account and override on a specific read request when another consistency level is desired.

3

System Design

This work presents a solution to execute transactions over geo-partitioned data with transactional causal consistency semantics. In this chapter, we present the system design of the implemented solution, starting by revealing the system model followed by the system architecture specification. Then, we discuss the challenges posed and alternative directions. Finally, we present our proposed solution with the algorithms developed to address the identified challenges.

3.1 System Model

We consider a system composed of a small set of data centers - for simplicity, we refer to each data center as a replica. The system maintains a database composed of a set of objects, where each object is replicated only in a subset of the replicas.

Clients, running in each replica, access and modify the database by issuing transactions, with a transaction consisting in a sequence of *read* and *write* operations that ends in a commit or rollback operation. On rollback, all side-effects of the transaction's operation are discarded. On commit, if the transaction executes correctly according to the supported consistency model, transactional causal consistency, the side effects are made durable. Operations are submitted in the local data center, with the system responsible for executing the necessary steps for executing transactions. We expect that clients will access mostly data items replicated in the local replica, but they can occasionally access data items not locally replicated - we assume this to be an infrequent operation.

For defining transactional causal consistency (TCC) semantics [3], we start by introducing some definitions.

We denote by $t(S)$ the state of the database after applying the write operations of committed transaction t to some state S. We define a database snapshot, S_n, as the state of the database after a sequence of committed transactions $t_1, \ldots, t_n$ from the initial database state, S_{init}, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$.

The transaction set $T(S)$ of a database snapshot S is the set of transactions included in S, e.g., $T(S_n) = \{t_1, \ldots, t_n\}$. We say that a transaction t_a executing in a database snapshot

S_a happened-before t_b executing in S_b, $t_a \prec t_b$, if $t_a \in T(S_b)$. Two transactions t_a and t_b are concurrent, $t_a \parallel t_b$, if $t_a \not\prec t_b \wedge t_b \not\prec t_a$ [25].

For a given set of transactions T, the happens-before relation defines a partial order among them, $O = (T, \prec)$. We say $O' = (T, <)$ is a valid serialization of $O = (T, \prec)$ if O' is a linear extension of O, i.e., $<$ is a total order compatible with $\prec$.

We say that a system implements the TCC semantics iff transactions executes in a snapshot obtained by a valid serialization of O: a write modifies a private copy of the snapshot, while a read reads from this private copy. Upon commit, the updates of a transaction t, will be added to the set of committed transactions of the system, T.

We note that transactions can execute concurrently, with clients accessing to different valid serializations. We assume the system guarantees state convergence, i.e., all valid serializations of $(T, \prec)$ lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [38].

As the system adopts partial replication, there will be no replica that holds a full database snapshot.

3.2 Architecture

In this section, we present the PotionDB architecture. Our work can be seen as an extension or a component of PotionDB, and as such, shares the same architecture of PotionDB.

PotionDB is a weakly consistent geo-replicated key-value store that adopts a partial replication model. One of the key features of PotionDB is that it maintains data structures with the results of pre-defined queries over data that is geo-replicated. This is similar to views in database systems.

PotionDB architecture is composed by multiple servers. Database objects are replicated on at least one server. New updates executed in a replica are propagated to other replicas asynchronously. Clients connect to one server to execute operations. Updates and queries are performed on a single server and return as soon as their execution completes. Clients can group a sequence of operations into a transaction.

In Figure 3.1, it is possible to that each PotionDB replica is composed by the following components:

- **Materializer**: The materializer generates and caches versions of objects requested by clients. It is used by the Transaction Manager to apply reads and writes to objects and determine when it can perform an operation. This component is also connected to the Replicator.

- **Transaction Manager**: This component is responsible for ensuring transaction causal consistency, as it is the component that coordinates transactions execution, implementing the Cure protocol [3]. The Transaction Manager receives requests

from clients to execute transactions. It accesses the Materializer to execute read and write operations and to commit transactions' results.

- **Replicator**: This is the component used that propagates updates to other data centers. This component uses RabbitMQ to handle communication between replicas.

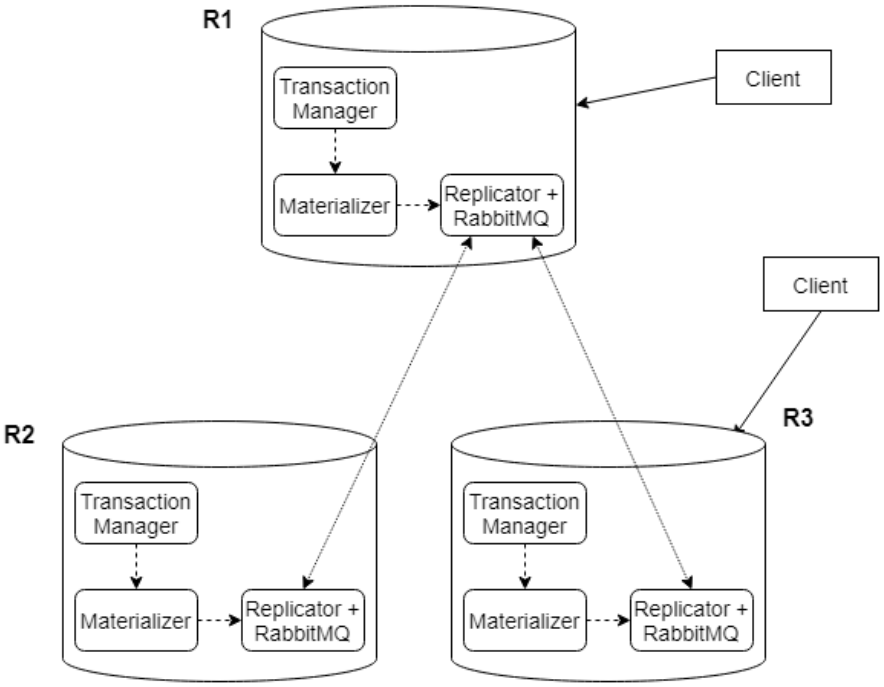


Figure 3.1: PotionDB architecture (taken from [35]).

3.2.1 System API

PotionDB implements a key-value store interface, and all its objects are indexed by a key and support get and update operations. Furthermore, in PotionDB, the objects are CRDTs, which ensures that their states will eventually converge even if objects are modified concurrently in different replicas. This allows operations to be performed locally and asynchronously propagated to other replicas.

In PotionDB, objects are uniquely identified by the following triple:

$$(key, bucket, crdtType)$$

where the value for *key* is chosen by the user to identify the object; *bucket* is the container where the object is stored, which is also used for partial replication purposes, allowing to control which replicas will replicate the object; *crdtType* identifies the object type. Throughout the document, and more clearly in the algorithms below, the terminology adopted to express the access to this triple is *".KeyParams"*.

Although PotionDB relies on executing *get* and *update* operations to query or update the state of an object, these operations are not performed independently. Instead, when the client wants to perform one or more operations, the operations are encapsulated in a single transaction. The operations supported by the PotionDB API are presented in Table 3.1.

StartTxn()	Starts the transaction and returns the transaction identifier txn_{id} and a copy of the replica vector clock, which identifies the snapshot in which the transaction will execute, i.e., the state of the database in which reads are performed. The clock vector entry concerning the local replica ensures that each transaction commits with a different clock vector, as an increment is employed to that entry for each transaction.
Get (txn_{id}, uid, opName(optional args))	Reads a single object identified by uid [1]. *optional args* are only used when the intention is to read a part of the state.
Get (txn_{id}, <uid_1, op_1>, ..., <uid_n,op_n>)	Reads a set of objects for the transaction identified by txn_{id}. Reads are performed on the object version corresponding to the transaction's vector clock, taking into account the updates performed in the same transaction.
Update (txn_{id}, uid, opName(args))	An update operation is similar to a get operation, but the arguments are required in this case. *opName* is the operation to be applied on the object, and arguments represent the operation arguments if any.
Update (txn_{id}, <uid_1, op_1>, ..., <uid_n,op_n>)	Updates a set of objects for the transaction with txn_{id}. Although updates are visible for subsequent reads during a transaction, updates are only applied to the latest version of the object at commit time.
CommitTxn (txn_{id})	Commits the transaction. The transaction is considered committed after all operations are applied. Commit updates are applied in the order defined by the vector clocks.

Table 3.1: PotionDB transaction interface

PotionDB supports transactions that ensure causal consistency. By implementing this consistency model, PotionDB ensures that operations performed by clients follow a

[1] uid represents the triple *(key, bucket, crdtType)* that identifies an object.

causal order between them. More specifically, the system ensures that clients always see operations following an order determined by the causal dependencies between operations. Since causal consistency is one of the weak consistency models, when running operations on different replicas, concurrent updates can occur in a given object. PotionDB uses CRDTS to deal with concurrency issues, ensuring that all replicas eventually reach the same converged state.

3.2.2 Partitioning

In PotionDB, objects are partitioned internally (within a server) and externally (across multiple servers). This implies different partitioning mechanisms that work differently:

- **Internal partitioning:** divide objects into several partitions. Each object is assigned to a partition by calculating a hash based on the object's *key*, *bucket*, and *crdtType*. When partitioning objects across multiple partitions, a replica can process requests concurrently without coordination if different clients access objects on different partitions.

- **External partitioning**: defines on which servers each object will be replicated. As it has been described, each object is identified by the triple *(key*, *bucket*, *crdtType)*. Buckets define groups of objects replicated by a replica group. Therefore, all objects within the same *bucket* are replicated in the same replica set.

In PotionDB, transactions can span any number of internal partitions, requiring that the transactions are local and cannot be applied to buckets that are not replicated locally. Within each partition, updates are performed sequentially and are applied only when committed. The updates of other transactions can be executed before or after the current transaction, before the first operation or after the last one, which means that each transaction's effects are ordered sequentially in each replica, although different replicas can apply transactions from a different replica in different orders. In this work, we extend the original transaction execution model to allow PotionDB to execute transactions that span multiple external partitions. Thus, even though PotionDB transactions are atomic, they only offer transactional causal consistency.

3.2.3 Replication

In PotionDB, operations are performed locally, and, periodically, transaction updates are propagated asynchronously to other replicas. Since all objects in PotionDB are operation-based CRDTs, the information needed to propagate an update to another replica consists of the operation type and its arguments.

To handle the communication between replicas, PotionDB uses RabbitMQ. RabbitMQ allows consumers to register the message topics they are interested in. As such, we leverage the topics to ensure that each replica only fetches the messages, including updates for objects in buckets they are replicating.

For each bucket, there is a topic and updates to object in the bucket are propagated through the bucket's topic.

A transaction is propagated splitting the transaction's updates by the appropriate topics. Additionally, each replica periodically sends in all topics a special message with the current clock of the replica - this allows a replica receiving this message to know that it has not missed any operation from the remote replica up to that clock. This information is used in the process to advance the entry of the remote replica in the version vector that represents the current database version, making those updates visible by future transactions (as in Cure [3]).

3.3 Challenges

This section discusses the challenges of implementing transactional causal consistency in a partial replication setting.

3.3.1 Challenge 1: Accessing Remote Objects

The first challenge to implement TCC in a partially replicated setting is accessing data objects that are not locally replicated. To this end, several problems must be addressed.

First, it is necessary to send the request to a replica holding the object. When the mapping between the objects and the replicas is stable, it is easy for each replica to maintain this mapping - typically at the granularity of a partition, containing multiple objects. If the replicas replicating an object (or partition) may change, this mapping needs to be updated. An alternative approach would be to search the replicas that hold the object when necessary. In our solution, we assume that the mapping between replicas and objects may change - to this end, we will have an object that maintains information about which replicas replicate each partition.

Second, sending the request to a replica is not enough, as we need to find a replica of the object in the same snapshot as the transaction being executed. In general, we can assume that as replicas evolve concurrently, the probability of finding a replica in the exact same version is zero. This seems to call for the need to have a system that maintains multiple versions of objects (or that can generate the multiple versions of an object). In our solution, a replica can produce different versions of the objects to be able to service requests from remote replicas.

Third, it is necessary to decide if the read or write operation should be executed in the remote replica or if a copy of the object should be (temporarily) created in the client's replica. If the object is a register, where the only two operations are read and write,

transferring the object's state or returning the result of the read should lead to a similar overhead. This is not the case if we have objects, such as sets, and support operations such as contains, where the result of a contains is much smaller than the contents of the set. A second aspect that needs to be considered is that if the object is transferred, the following operations for that object in the same transaction can be executed locally. In our solution, we decided to transfer the state of the object to the client's replica.

3.3.2 Challenge 2: Ensuring Transaction Atomicity

The second challenge is how to guarantee that transaction atomicity is respected. This is related to how replicas are updated, and transaction updates are propagated to replicas.

A possible approach to guarantee transaction atomicity is to collect all transaction updates and propagate them as a unit to all replicas of the objects modified in the transaction. This guarantees that a replica can apply all updates atomically. This process can be optimized by propagating to a replica only the updates for the objects replicated in that replica, at the cost of requiring to execute a more complex recovery process if a replica fails. This is the approach we have adopted in our solution.

An alternative approach would be to allow transaction updates to be partitioned, with a fragment for each replica accessed. In this case, to guarantee that a replica that holds more than one object updated in more than one fragment is updated atomically, we could set a circular dependency between the fragments - specifying that each fragment depends on all the others. This would require the process of executing transaction updates to consider these dependency circles.

3.4 Design

This section presents our solution to perform transactions over partially replicated data, addressing some of the alternatives we have explored throughout our work. We implemented our solution in the context of PotionDB, a weak consistency geo-replicated key-value store with partial replication that is currently being developed in the NOVA LINCS research group.

To recall, PotionDB is a distributed key-value store composed of multiple servers, where each database object is replicated on at least one server. Clients connect to one server to execute operations. Updates and queries are performed on a single server and return as soon as their execution completes. New updates are propagated to other replicas asynchronously. Clients can group a sequence of operations into one transaction.

3.4.1 Replication Mapping

PotionDB allows responding to queries related to global data through materialized views that can be split across multiple servers. In this way, a client can obtain global data by accessing only one server. Besides, replicas do not need to know about the replicated

buckets in each of the replicas to update the materialized views. This implies that, up to this point, PotionDB did not kept an explicit register of the mapping between objects (buckets) and replicas - a replica would receive updates regarding the objects it replicates simply by subscribing to the associated topics in RabbitMQ. However, to support transactions over geo-partitioned data, it was essential to implement support for explicitly keeping this mapping. The alternative, disseminating requests for objects through RabbitMQ, would require implementing some way to coordinate who would reply to the request. Furthermore, as RabbitMQ is designed for asynchronous communication and is not optimized for handing interactive requests, it would lead to high latency.

Considering the example in Fig. 3.2, we can see a client trying to execute an operation on an object that is not replicated locally. In that situation, it must be possible to identify that object "C" is not replicated by "R1" and is replicated by "R2" and "R3" In order to do that, there should be a mapping between objects and replicas, where it is easy to identify which replicas hold the object the client is trying to access. Therefore, the first step of our work was to implement this mapping. This way, when a transaction accesses an object that is not replicated locally, it is possible to know which replicas replicate the object.

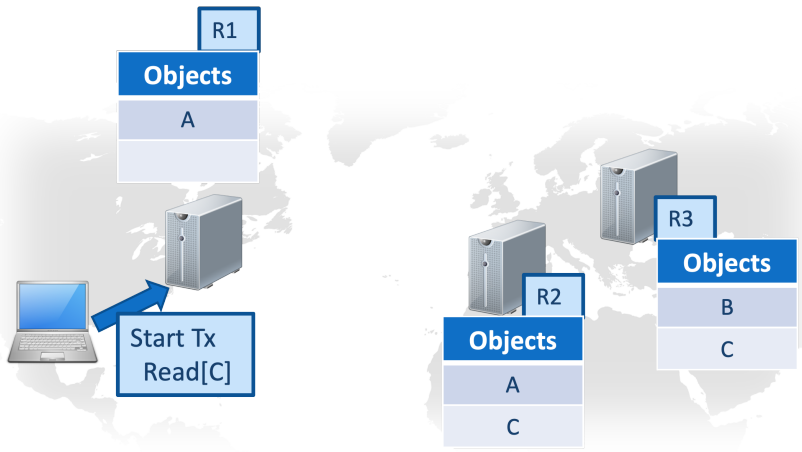


Figure 3.2: A client seeking to access an object that is not replicated locally

In PotionDB, when a replica starts, it has a list of buckets that it replicates. The replica starts by fetching a copy of the bucket from an existing replica, if any, and subscribes to the updates to the bucket in RabbitMQ. We now explain the mechanism developed to share information about which replicas replicate each bucket. This allows for all replicas to know for a given bucket which replica set is replicating it.

To achieve this, the overall idea is to store this information in an object stored in PotionDB. This object is stored in a bucket, *admin*, that is replicated in all replicas. This way, all replicas have access to this information. Algorithm 3.1 shows the operation defined in PotionDB to manage this object, *replicaMap*. The *replicaMap* is a map, where the *key* is the bucket and the *value* is the set of replicas replicating the bucket. Each replica keeps a copy of this object in memory for faster access when needed during the execution

of a transaction. Like all PotionDB objects, this object is a CRDT, but with the particularity of being replicated in all replicas. When a replica initializes, the *replicaMap* of that replica is also initialized (Alg. 3.1 line 3). In other words, at that time, this replica will update the object shared by all replicas, adding the buckets it replicates to the map (Alg. 3.1 line 12-15). Since operations are done on CRDTs, if objects are modified concurrently, their state will eventually converge, and ultimately all replicas will have access to the updated object. The *replicaMap* cached object is updated by reading the object as soon as it knows there was an update (Alg. 3.1 line 5) and also by periodically reading it (Alg. 3.1 line 6).

Algorithm 3.1 Replication mapping algorithm

```
1:  replicaMap ← ⊥
2:
3:  procedure InitializeReplicaMap(replica_id, bucketsList)
4:      adminUpdate(replica_id, bucketsList)
5:      replicaMap ← adminRead()
6:      Setup Periodic Timer ReadreplicaMap(ΔT)          ▷ every ΔT executes
7:
8:  procedure ReadReplicaMap
9:      replicaMap ← RepMapRead()
10:
11: procedure RepMapUpdate(replica_id, buckets)
12:     obj_id ← newObjectId("replicaSet","admin",RRMAP)
13:     for b ∈ buckets do
14:         op ← RWMapUpd(b, SetAdd(replica_id))      ▷ Operations to modify a CRDT
15:         PotionDB.update(obj_id, op)
16:
17: procedure RepMapRead: map
18:     obj_id ← newObjectId("replicaSet","admin",RRMAP)
          return PotionDB.get(obj_id)
19:
```

3.4.2 TCC Algorithm

PotionDB adopts a protocol based on Cure [3] to manage transaction execution in a way that provides TCC. A transaction submitted in replica r_i is identified by a timestamp, (ts, r_i) assigned at commit time in the replica r_i. A transaction executes in the snapshot installed in the local replica when the transaction starts - this snapshot defines the causal dependencies of the transaction.

Each replica maintains a version vector summarizing the transactions executed in the local replica. Replicas propagate transactions peer-to-peer in FIFO order. When receiving a transaction, the local replica verifies if the transaction's dependencies are satisfied before applying the updates. If not, the transaction is logged and executed after dependencies are satisfied.

As a replica only propagates a transaction's updates to the replicas that replicate the objects involved, a replica will not receive information about all transactions. The fact that transactions are propagated in FIFO order allows a replica to know that it has not missed a transaction from a given source. If a replica r_1 does not send transactions for some other replica r_2 for some time, it sends an acknowledge to allow the replica to advance its version vector.

We now present our proposal for executing transactions over data that is geo-partitioned. Algorithm 3.2 presents the algorithm to execute transactions.

Before executing a transaction, it is necessary to start the transaction by assigning it an identifier txn_{id} (Alg. 3.2 line 7) and a value of the replica vector clock that is more recent than the current one (Alg. 3.2 line 8). This value identifies the snapshot in which the transaction will execute. For both reads and writes, the processing of an operation starts by verifying which operations can be executed locally and which ones require accessing objects available only on remote replicas (Alg. 3.2 line 17 and 37). For these objects, the other servers are contacted to bring the data to the local replica and execute the requests locally (Alg. 3.2 line 25 and 43).

As transactions can consist of several operations, the objects we get from other replicas are stored in the local replica. So within a transaction, it is unnecessary to fetch the same object several times. Instead, a previously obtained object can be reused for multiple operations in the same transaction (Alg. 3.2 line 25 and 43). When a transaction ends, the copy is discarded (Alg. 3.2 line 69). A possible optimization would be to keep the object on a cache, and when the object is necessary for another transaction, ask the remote replica for the object only if the object has been changed meanwhile.

When a replica requests objects from another replica, the replica sends the transaction snapshot to get the appropriate version of the object to ensure that the transactions reads are executed in a consistent database snapshot (Alg. 3.2 line 29 and 46). When the remote replica receives a request for a given version of an object, it first checks if it is able to reconstruct the required version. If not, the remote replica waits until it has the necessary updates. If it can, it reconstructs the requested object version and returns it to the requester. All updates performed in the context of a transaction are saved in a list to be applied and propagated to other replicas at commit time - this applies both to objects that are replicated in the local replica and those that have been fetched from remote replicas.

When a commit request is executed, local updates start by sending a `PREPARE` to each of the partitions involved in transaction updates and gathering all proposed timestamps to accept the maximum one, $maxTs$ (Alg. 3.2 line 55 and 57). Then, remote updates are applied directly in the TransactionManager, and each remote update is assigned to a partition (Alg. 3.2 line 60 and 63). Thus, remote updates can be sent to Materializer at `COMMIT` along with their partition's local updates (Alg. 3.2 line 66).

After implementing the first version of our algorithm, we performed an evaluation that showed that the system latency heavily increased when we went from performing 0%

Algorithm 3.2 TCC algorithm

```
 1: txnClocks ← ⊥
 2: txnObjs ← ⊥
 3: localClock ← ⊥
 4: updsToApply ← ⊥
 5:
 6: procedure BEGIN
 7:     txn_id ← generateUniqueId()
 8:     txnTs ← localClock.NextTimestamp(replica_id)
 9:     txnClocks[txn_id] ← txnTs
10:
11: procedure READ(txn_id, ts, reads): set
12:     result ← ⊥
13:     rmtReads ← ⊥
14:     targetBuckets ← ⊥
15:     for read ∈ reads do
16:         if localRep ∈ getReplicas(read) then
17:             result ← result ∪ execLocalRead(read, txn_id, ts)
18:         else
19:             rmtReads ← rmtReads ∪ read
20:             targetBuckets ← targetBuckets ∪ read.Bucket
21:     replicasByBucket ← chooseRmtReplica(targetBuckets)
22:     rmtReq, cachedReq ← splitReadReqs(txn_id, rmtReads, replicasByBucket)
23:     for r ∈ rmtRequest do
24:         result ← result ∪ sendRmtReads(r.rep_id, r.readLst, txn_id)
25:     result ← result ∪ execCachedReads(txnObjs, cachedReq, txn_id)
        return result
26:
27: procedure SENDRMTREADS(rep_id, readLst, txn_id): set
28:     result ← ⊥
29:     sendReqReadObjs(rep_id, txnClocks[txn_id], readLst)
30:     reply ← receiveRepReadObjs(rep_id)
31:     txnObjs.putAll(reply)
32:     checkMissingUpds(txn_id, reply, readLst)
33:     result ← result ∪ execCachedReads(txnObjs, readLst, txn_id)
        return result
34:
35: procedure UPDATE(txn_id, ts, updates)
36:     localUpds, rmtUpds, targetBuckets ← groupWrites(updates)
37:     for localUpdate ∈ localUpds do
38:         execLocalUpd(localUpdate, txn_id, ts)
39:     replicasByBucket ← chooseRmtReplica(targetBuckets)
40:     rmtReqcachedReq ← splitUpdReqs(txn_id, rmtReads, replicasByBucket)
41:     for rep_id, upds ∈ rmtRequests do
42:         sendRmtUpds(rep_id, upd, txn_id)
43:     execCachedUpds(txnObjs, cachedReq, txn_id)
44:
```

```
45: procedure sendRmtUpds(rep_id, upds, txn_id)
46:     sendReqUpdObjs(rep_id, txnClocks[txn_id], upds)
47:     reply ← receiveRepUpdObjs(rep_id)
48:     txnObjs.putAll(reply)
49:     checkMissingUpds(txn_id, reply, upds)
50:     execCachedUpds(txnObjs, upds, txn_id)
51:
52: procedure commit(txn_id, txnPartitions)
53:     proposedTimestamps ←⊥
54:     for part_id ∈ txnPartitions do
55:         proposedTs ← sendPrepareReqToMaterializer(txn_id, part_id)
56:         proposedTimestamp ← proposedTimestamp ∪ proposedTs
57:     maxTs ← acceptMaxTs(proposedTimestamps)
58:     rmtUpdsPartitions ←⊥
59:     for upd ∈ updsToApply do          ▷ upd has object and operation information
60:         updArgs ← applyUpdate(upd)
61:         updToUse ← newUpdateObject(update.KeyParams, updArgs)
62:         currKey ← GetChannelKey(updToUse.KeyParams)
63:         rmtUpdsPartitions[currKey] ← rmtUpdsPartitions[currKey] ∪ updToUse
64:         updByTs ← newUpdateByTxnClock(maxTs, upd)
65:     for part_id ∈ txnPartitions do
66:         sendCommitReqToMaterializer(txn_id, maxTs, rmtUpdsPartitions[part_id])
67:     txnClocks ← txnClocks \ txnClocks[txn_id]
68:     txnObjs ← txnObjs \ txnObjs[txn_id]
69:
```

to 100% of remote operations. The justification for this increase in latency was related to the fact that when replica r_i requests an object from replica r_j, replica r_j needed to wait to get a compatible version of the objects. In particular, replica r_j could only return an object after it receives all operations from r_i up to the transaction snapshot - as transactions are propagated asynchronously, this added latency to the process.

Therefore, we optimized our solution, saving in each replica r_i the last transactions performed locally. Thus, when r_i fetches objects from a replica r_j, r_j can receive a version that still does not include the last transactions from r_i. After receiving the object, r_i will locally apply the missing updates, if any. This optimization implies that r_i does not have to wait for its transactions to propagate to r_j when it wants to fetch a remote object from r_j.

3.4.3 Selecting a replica to fetch objects

The approach of bringing the objects to a replica, updating it locally and registering the transactions with the local replica identifier has implications. If an object o could only be updated in the replicas that replicate o, it would be possible to know if some missing update could impact the state of o or not - only updates from replicas that replicate o

would be relevant. Therefore, only vector entries from these replicas were relevant for those objects. With our approach, this is no longer true.

For estimating the state of other replicas and deciding which replica is more likely to have a version compatible with the local version, each replica maintains, for each other replica, a version vector of the updates that are known to have been received. This version vector is updated using the dependency vector of received transactions. When a remote replica needs to be contacted, this information is used to select the closest replica that is more likely to have all necessary updates.

Algorithm 3.3 presents the algorithm that defines for each bucket which replica will be reached. This algorithm is called through Alg. 3.2 (lines 21 and 39). The *targetBuckets* list was built in this algorithm as the list of buckets that must be accessed remotely. In Algorithm 3.3, we start by checking for each bucket which replicas replicate it (line 10). Knowing the list of replicas, we can see for each replica whether the version vector entries of the replicas that replicate the object are behind the *localClock*. In case any replica is delayed, that replica is excluded.

In Algorithm 3.3, we optimized the accesses to the replicas, combining the highest number of requests in the smallest number of replicas possible. After we exclude the replicas that have the relevant vector entries delayed, we add for each replica the bucket that is being iterated, creating for each replica a list of buckets (*bucketsByReplica*). In this list, we can infer which is the replica with the most buckets. Buckets that were in this replica's list are iterated over and added to the *replicasByBucket* map as a *key*. The *value* assigned to each of these entries is that of the replica with the most buckets (line 22). Buckets that are in this are excluded from the *targetBuckets* list (line 23). This process is repeated until there are no more buckets to be assigned to replicas (line 8).

An alternative approach would be to create a structure that maintains the distance from the local replica to other replicas, using this structure to access the closest replica that replicates the objects to be fetched. In this way, it would also be possible to combine the highest number of requests in the smallest possible number of replicas since the local replica would always access the closest replica that replicated the object. Furthermore, this solution has the advantage of optimizing the round trip time of accessing a remote replica with the requested objects. Still, it increases the probability of accessing an outdated replica. In this case, the replica will only send the requested objects after receiving updates from other replicas that allow the replica to return the requested version of the objects.

3.5 Correctness

In this section, we present correctness arguments for our proposed approach.

According to the definition presented in Section 3.1, a transaction must access a snapshot that includes a serialization of a set of transactions applied according to causal order.

Algorithm 3.3 Remote replica selection algorithm

```
1:  localClock ← ⊥
2:  rmtTs ← ⊥
3:
4:  procedure chooseRmtReplica(targetBuckets) : map
5:      replicasByBucket ← ⊥
6:      bucketsByReplica ← ⊥
7:      maxRep ← ⊥
8:      while len(targetBuckets) > 0 do
9:          for bucket ∈ targetBuckets do
10:             replicas ← getReplicasWithBucket(bucket)
11:             for rep ∈ replicas do
12:                 repVersion ← rmtTs[rep]
13:                 replicas ← removeLateReplicas(localClock.Timestamp, repVersion)
14:             maxCounter ← 0
15:             maxRep ← replicas[0]
16:             for rep ∈ replicas do
17:                 bucketsByReplica[rep] ← bucketsByReplica[rep] ∪ bucket
18:                 if len(bucketsByReplica[rep]) > maxCounter then
19:                     maxCounter ← len(bucketsByReplica[rep])
20:                     maxRep ← rep[0]
21:         for bucket ∈ bucketsByReplica[maxRep] do
22:             replicasByBucket[bucket] ← maxRep
23:             targetBuckets ← targetBuckets \ bucket
        return replicasByBucket
24:
```

Each replica applies transaction according to the causal order, as the updates of a transaction are only applied after the transaction's dependencies are satisfied. Additionally, each replica only makes visible updates from a remote replica j with a timestamp t_j after it receives information that, for all locally replicated buckets, it has received all updates up to t_j - note that as the "admin" bucket is replicated in all replicas, every replica will receive information from every other replica.

This guarantees that the snapshot in which a transaction executes is causally consistent. Likewise, objects fetched from remote replicas, as they are in the same snapshot, will also imply causally.

3.6 Faults

In our work, we have not addressed the problem of replica faults - this is a common assumption, if we consider that a replica can be replicated using some state-machine algorithm, such as Paxos [24], to guarantee that the replica remains alive in the presence of machine faults.

If a replica fails, it would be necessary to guarantee that all replicas can still receive

the updates published by the faulty replica. Either RabbitMQ can guarantee that, or in some corner case, it would be necessary to implement a recovery mechanism where replicas that have already received the updates from the faulty replica would forward it to the other replicas along with information that no additional update is missing from the faulty replica until some know timestamp. This would allow for all replicas to execute the same set of updates published by the faulty replica.

3.7 Summary

In this chapter, we presented the system design of our solution by explaining the algorithms developed to ensure the expected properties of this work.

In section 3.3, we identified the main challenges of this work. In that section, we concluded that one of the challenges (presented in Section 3.3.1) was understanding how to access remote objects. This challenge was divided into two parts: knowing the list of replicas holding an object and finding a replica of the object in the same snapshot as the transaction being executed. First, as described in Section 3.4.1, we created a mapping between objects and replicas to find out the list of replicas that hold an object. All replicas can access and change this mapping, and the others eventually see all changes made by one replica. This way, we guarantee that when a replica wants to access a remote object, it has access to the list of replicas that replicate it. Regarding the second part of the challenge, we ensure that we find a replica of the object in the same snapshot of the transaction being executed by specifying the version of the object we want to use. How transaction clocks are used to fetch an object from another replica is explained throughout Section 3.4.2.

To ensure that atomicity is respected (challenge presented in Section 3.3.2), we collect all transaction updates and propagate them as a unit. More specifically, we know that a transaction executes on a snapshot and that snapshot defines the causal dependencies of the transactions. So for local objects, the replica verifies if the transaction's dependencies are satisfied before executing. Remote objects are fetched using the transaction clock, and the object is returned in the transaction snapshot. All updates made to the remote object are saved and applied at commit time. Thus, all updates (remote and local) are propagated at the commit time to guarantee atomicity.

4

Implementation

This chapter focuses on the implementation details of our solution. In particular, we start by presenting the environment in which we develop our work and the used tools. Then we describe the implementation details related to communication and replication techniques. Finally, we explain how we manage the updates that are applied to remote objects.

4.1 Programming Environment

Our solution was implemented in the context of PotionDB. PotionDB is implemented in Go, and consequently, that was the programming language adopted throughout the development of our work.

4.1.1 Development Tools

To maintain and track changes to the PotionDB code, we use the version control platform GitHub [19].

The PotionDB project is hosted on a private GitHub repository. To fulfil the goals of our work, we needed to extend the PotionDB, so we forked the repository. As a result, we started to have a copy of the repository. From there, we created a branch for our changes and for each feature we wanted to test.

4.2 Communication

In PotionDB, operations were performed locally, without ever having to contact other replicas. Replicas only needed to communicate to replicate transaction updates and share their IDs with other replicas at initialization time. Periodically, transaction updates were propagated asynchronously among replicas through RabbitMQ, which means that PotionDB used RabbitMQ to handle replicas' communication. Therefore, by supporting transactions that access remote objects, we had to create a way to allow replicas to communicate with each other, bypassing RabbitMQ.

We have decided not to use RabbitMQ for this communication, as RabbitMQ, being designed to support asynchronous communication, would lead to non-optimal communication latency. As in our context, a replica needs to contact a remote replica to fetch the necessary data to execute an interactive transaction, this would add latency to the transaction execution.

In PotionDB, communication from clients to servers and between servers was done through protobufs.

4.2.1 Remote Protobuf Messages

In order to have direct communication between the replicas, we decided to use direct TCP connections and protobufs to serialize our requests.

Since in PotionDB there were no requests to fetch objects from other replicas, there were also no protobuf messages for this case. So to request objects from another replica, the first thing we had to do was define the structure of the protobuf messages that we were going to use. Listing 4.1 shows the protobuf messages created.

```
message ReqObjects {
  repeated ApbBoundObject object = 1;
  required bytes clk = 2;
  required int32 replicaID = 3;
}

message ReplyReqObjects {
  repeated ProtoState state = 1;
  required bytes clk = 2;
}
```

Listing 4.1: New protobuf messages

ReqObjects had to be created from scratch to gather and transmit all the needed attributes for the other replica to respond with the requested objects. Therefore, in its attributes *object* is a `repeated` attribute, which means that the message can include multiple objects, and therefore, this attribute represents the list of objects a replica wants to fetch from another replica; *clk* is the clock of the transaction being executed; *replicaID* is the Id of the replica that is sending the request, only the replica that received it knows which clock entry it can ignore when reconstructing the versions of the requested objects.

When the remote replica has all the requested objects ready to be sent to the client replica, it can send them. However, similarly to what happened with remote object requests, it was necessary to define new protobuf messages for object responses. Therefore, we created the *ReplyReqObjects* message, where *state* is the list of serialized objects to be returned, and *clk* is the clock used in the remote replica to obtain the version of those objects.

In order to send and receive the created messages, we used an auxiliary function previously implemented in PotionDB. This function has the following structure:

```
1    SendProto(code, protobuf, conn)
```

where *code* is the code of the protobuf message, transmitting the type of request being sent; *conn* is the TCP connection created between the replicas for which we want to exchange requests; *protobuf* is the message that one replica intends to transmit to another replica.

Therefore, in the case of a read operation, requests for another replica to send an object that is not locally replicated have the following format:

```
1    SendProto(ReqObjects, CreateReadRmtObjs(req, ts), conn)
```

where *ReqObjects* is the code of the protobuf we created, revealing that, in this case, we are requesting remote objects. The *CreateReadRmtObjs* method generates the protobuf that will be sent to request the remote objects. Listing 4.2 shows in detail the implementation of the *CreateReadRmtObjs* function, and it is possible to observe that in this function, the protobuf message is created from the transaction timestamp *ts* and the *reads* to be carried out, which are consequently passed as a parameter to the function. From *reads*, it is possible to obtain the corresponding object to be read (line 5).

```
1 func CreateReadRmtObjs(reads []ReadObjectParams, ts clocksi.Timestamp) (
    protobuf *proto.ReqObjects) {
2   repId := int32(replicaID)
3   protobuf = &proto.ReqObjects{
4     Object:    createBoundObjectRead(reads),
5     Clk:       ts.ToBytes(),
6     ReplicaID: &repId,
7   }
8   return
9 }
```

Listing 4.2: Function to generate a protobuf message

Suppose an update request is addressed, and it is necessary to fetch remote objects from other replicas. In that case, it is only required to change the function that generates the protobuf message to obtain the objects from *[]UpdateObjectParams* and not from *[]ReadObjectParams*.

When a remote replica wants to send the reply message (*ReplyReqObjects*) to the client replica with the requested objects, it follows the same sequence of operations that the client replica followed to send the request initially. However, this message is different from the one used to request the objects. Therefore, another code is employed in the function to send the objects to the replica that requested them (*SendProto(code, protobuf, conn))*, and different functions are used to generate the protobuf message.

4.2.2 Get Object Version

When a replica receives a request to send objects to another replica, it has access to the attributes of the request in the *ReqObjsProto* protobuf object. Among these attributes it

is the list of *objects* to read from the replica. Therefore, as a request of this type arrives, in the TransactionManager, the list of objects is distributed across partitions. Then, every Materializer partition for which there are objects to read is requested to read those objects in the version specified by the clock snapshot received in the request.

In order for the client replica not to have to wait for its updates to be applied by the remote replica (as mentioned in section 3.4.2), we defined that we would ignore the client replica entry in the sent clock. Therefore, the client replica does not have to wait for its transactions to propagate when fetching an object from a remote replica. To do this, we replace in the received clock the value of the client's replica entry with the stable value in the partition of the object. From that moment on, the clock with the changed entry is adopted.

In Materializer, we first need to compare the received clock with the Materializer state. Meaning that the received clock is compared with the Materializer's "stable clock" and "pending clocks" to know if the objects:

- Can be read directly;
- Have to be read in the past;
- Or if the replica has to wait until it receives updates from other replicas - regarding transactions executed in other replicas.

If the object is directly readable, the object is returned without reconstructing any version of the object. However, if we must read the object in the past, the CRDT is asked to reconstruct a passed object state. If it is necessary to wait, the request is put on hold, and when new commits from other replicas are applied, it is verified if the read can already be done.

4.2.3 Restore Version

Each time an operation is performed on a CRDT, a triple is saved with the following format:

$$(timestamp, op, effect)$$

where *timestamp* is the replica clock at the time the operation is performed; *op* is all the information about the operation and *effect* is the specific changes that the operation will make. For example, when removing an element from as ORSet, the *op* is `remove(element, unique)` and the *effect* are the identifiers to be removed. In *op*, `unique` represents a unique identifier generated each time an add is made to the ORSet. This identifier is removed during the remove operation. This allows concurrent adds to override concurrent removes.

When we intend to restore a version in the past, we go through the triple vector from newest to oldest, going as far back as necessary. As we traverse the vector, we apply the

inverse of the effect to the CRDT. Thus, we go back until one of the following events happens:

- We reach the intended version
- We reach a version where all entries are less than or equal to the intended version

The first case is ideal, and it means we get the version we want. However, in the second case, it means that there were operations concurrent with the requested clock. For this reason, it is necessary to undo the effects of these operations, for which it is required to use the triple op. Note that in this case, undoing is a bit more complex if some operation in the vector of locally executed operations is not undone. In this case, it is necessary to replay that operation to guarantee that the effects of this operation are applied. Consider the following example: in a set, two concurrent operations remove an operation in a set. When applying these operations in a third replica, the first operation executed will have the effect of removing the element, and the second would have no side-effect. If we undo only the effects of the first operation, we would restore the element in the set. However, that is not the expected state, as if the second operation would have removed the element if the first was not executed. After dealing with the concurrent operations, the object version is reconstructed and prepared to be sent to the replica that requested it.

4.3 Replication

In Section 3.2.3, we examined the PotionDB replication process, explaining that PotionDB uses RabbitMQ to periodically propagate new updates to other replicas that replicate the objects involved in the transactions performed. In our solution, replication continues to be done by RabbitMQ, but it was also necessary to propagate the updates performed on objects fetched from other replicas.

As we explained in Section 3.2.2, objects are internally divided into partitions. By dividing objects by several partitions, we can execute operations concurrently without using any concurrency control mechanism if performing operations in different partitions. Furthermore, the replication mechanism is also partition-oriented. So to be able to replicate the remote objects, we also had to divide them by partition. Like local objects, each remote object is assigned to a partition by calculating a hash based on the object's identifier *key*, *bucket*, and *crdtType*.

At commit time, we request Materializer to pass the remote objects for replication as if they were local objects. In a transaction where we have both local and remote objects, we want all updates to have the same clock as they all belong to the same transaction. For this reason, we have to ask the Materializer to pass these objects to replication as if they were local objects. Only in this way, the updates of a transaction can be propagated as a unit in the context of the normal replication process.

4.4 Update List Management

To improve the performance of our solution, we decided that each replica could save the last transactions it made for each object. So when a replica requests remote objects from other replicas, the remote replica does not need to wait for the updates from the replica making the request. Therefore, we created a local structure that allows us to determine, for each object, the list of updates that have been executed locally and that the other replicas have not yet observed. Thus, when the local replica receives the list of objects that it requested from a remote replica, it checks which updates executed locally need to be applied to reach the transaction snapshot. This check is referred to as *checkMissingUpds* in lines 26 and 43 of algorithm 3.2.

In Algorithm 4.1 we explain what happens in the method that checks if there are updates to be applied to the objects requested to the remote replica. This function receives the identifier of the transaction, the reply received from the remote replica, which includes a list of the objects fetched (*reply.state*) and the clock (version vector) of the version received (*reply.clk)*, and the list of operations that led to the remote request (*opList*). Note that the object needed for operation in position i of list (*opList*), is in position i of the list of objects received (*reply.state*). For each of the objects read from the remote replica, we need to check the missing updates. The structure that holds for each object the updates performed by transactions already committed is the *localUpdates*. In the function presented in Algorithm 4.1, it is necessary to go through the updates of the requested objects and create a new list with the updates to be carried out in the transaction to obtain the requested version of the objects ($updsToApply[txn_{id}]$).

From the message sent by the remote replica, it is possible to obtain the clock associated with the object's version and extract the clock entry that concerns the local replica by doing *clockReceived.GetPos(replicaID)* (line 4). For each update of an object saved locally, it is necessary to compare the clock entry associated with that update with the clock entry received from the remote replica that concerns the local replica to know which updates to execute (line 8). By doing this, we determine that:

- If the locally stored update clock entry is higher, it means that the update has not yet been applied to the received version, and therefore we need to store that update in the *updsToApply* structure (line 9).
- If the remote replica entry is equal or higher, the received object already has the update applied. For this reason, the update can be removed from the list of pending updates for that object (line 11).

The *updsToApply* structure stores for one transaction all the updates that the CRDT will have to temporarily apply for reading purposes. The updates performed during the transaction are also stored in this structure to be applied to reads. However, only updates concerning requests placed during that transaction are executed and propagated at the

Algorithm 4.1 Check Missing Updates algorithm

```
1:  procedure checkMissingUpds(txn_id, reply, opList)
2:      objects ← reply.state
3:      clockReceived ← reply.clk
4:      clockEntryReceived ← clockReceived.GetPos(rep_id)        ▷ rep_id is the local id
5:      for idx, object ∈ objects do
6:          opKey ← opList[idx].KeyParams
7:          for localUpdate ∈ localUpdates[opKey] do
8:              if localUpdate.clkEntry > clockEntryReceived then
9:                  updsToApply[txn_id] ← updsToApply[txn_id] ∪ localUpdate.upd
10:             else
11:                 localUpdates[opKey] ← localUpdates[opKey] \ localUpdate
12:
```

time of commit. Updates stored in *updsToApply* but referring to past transactions will not be re-applied to the current transaction.

Only after applying all the updates that had been done in the local replica on an object can we guarantee that we have a consistent version of the object when we perform new operations on the object. Therefore, it is important to keep the updates that have not been propagated to the other replicas in the *localUpdates* list. Updates can only be discarded from this list when we have the information that the other replicas have already read it. This means that until we receive an object with the update being checked applied, the update continues in the pending updates list.

4.5 Summary

This chapter specifies some implementation details of the designed solution.

Section 4.1 describes the programming environment in which our work was developed, including its context and the programming language used.

Section 4.2 presents the additions of protobuf messages that were made at the communication level to introduce direct communication between servers. This section also explains how we read a specific version of an object and restore that version of the object.

Section 4.3 defines the replication process of our solution that recognizes the replication of objects that were fetched to a remote replica.

Finally, Section 4.4 explains how we check if the fetched object has missing local replica updates and in what situation these updates are applied. Additionally, we explain how the list of updates made locally is managed and in what situation an update can be removed from this list.

5

Evaluation

In this chapter, we evaluate the performance of our solution in order to understand how our system behaves in different scenarios. For this in the evaluation, we tried to answer the following questions:

- What is the scalability of the system?
- How does the ratio of remote transactions influence performance?
- Does the order in which the operations in a transaction are executed impact performance?
- Is the cost of performing reads and updates on remote objects higher than the cost of performing reads only?
- How does the distribution of objects that are accessed influence performance?
- How does the number of objects accessed in each operation of a transaction influence system performance?
- How does scattering client requests across multiple servers affect performance?

Throughout the chapter, we attempt to answer each of these questions, presenting and discussing the results of our evaluation. In the initial part of the chapter, we describe the experimental setup and the configurations we used in our experimental evaluation.

5.1 Experimental Setup

Our evaluation experiences were performed in the cluster of the Department of Computer Science of the FCT NOVA. To run our experiments, we used 6 of the 24 nodes in the cluster, using five nodes as servers and one for the clients. The servers run on nodes with an AMDEPYC 7281 CPU with 128GB of RAM and 16 cores. On each server runs an instance of PotionDB, each running in a docker container. Each server is responsible for replicating only part of the data, and therefore when it is launched, it knows upfront

which buckets it replicates. For example, each instance replicates data relating to a continent, meaning that one server replicates all data concerning Europe, while another replicates all data from Africa. Data that cannot be associated with a location is replicated across all servers.

In order to run our experiments, we decided to simulate that each of our five servers ran in an Amazon Web Services (AWS) data center. Table 5.1 brings together the information regarding the AWS data centers we chose to simulate and what the latency is between each pair of those data centers. Each data center in the table corresponds to one of the servers used during the experimental evaluation.

The latency between servers was simulated using netem and TC [2]. Netem is a kernel component that provides network emulation to test and simulate the properties of a WAN (Wide Area Network). TC is the command that allows controlling the netem, namely to add network latency to a specific server. Through TC, we added a delay rule to netem for each connection of two servers. The latency chosen to delay traffic between two servers corresponds to the one presented in the table, with four delay rules created for each server. Each rule created is associated with an IP and therefore is assigned to one of the servers.

This simulation allows us to have an approximation of a real context in which there are several data centers worldwide, with different latencies between them. During the different evaluation experiences, the client always makes requests to the same server. The server accessed by the client simulates the EU-NORTH-1 data center, and therefore the latency values adopted between that server and the other servers are presented in the first row in Table 5.1

	Stockholm	**Ireland**	**London**	**Virginia**	**Mumbai**
EU-NORTH-1 Stockholm	—	43.45	35.88	108.35	138.15
EU-WEST-1 Ireland	43	—	36	68.2	122.38
EU-WEST-2 London	34.84	36.09	—	99.83	113.06
US-EAST-1 Virginia	111.47	68.61	101.08	—	187.43
AP-SOUTH-1 Mumbai	137.28	122.5	111.07	187.21	—

Table 5.1: Latencies between AWS data centers (ms) [22]

The client runs on a node with 2x Intel Xeon E5-2620 v2 CPU, with 64GB of RAM and 12 cores. During an experiment, each client always sends requests to the same data center. In this way, we can simulate that clients usually communicate with only the data center that is geographically closest. The client runs an instance of a client based on the TPC-H benchmark in a docker container.

5.1.1 TPC Benchmark^{TM} H

TPC Benchmark^{TM} H (TPC-H) is a decision support benchmark consisting of a set of business-oriented ad-hoc queries and concurrent data modifications over a database simulating an e-commerce site. The TPC-H represents decision support systems that analyze large amounts of data, execute complex queries, and provide answers to crucial business questions [43].

The evaluation was done using a dataset based on the TPC-H Benchmark, and for our tests, we used the data generated by the dbgen tool with SF=0.1. Through the scale factor, we determine the size of the database, and for SF=1, approximately 1GB of data is generated. The relationships between the columns of the generated tables are shown in Figure 5.1.

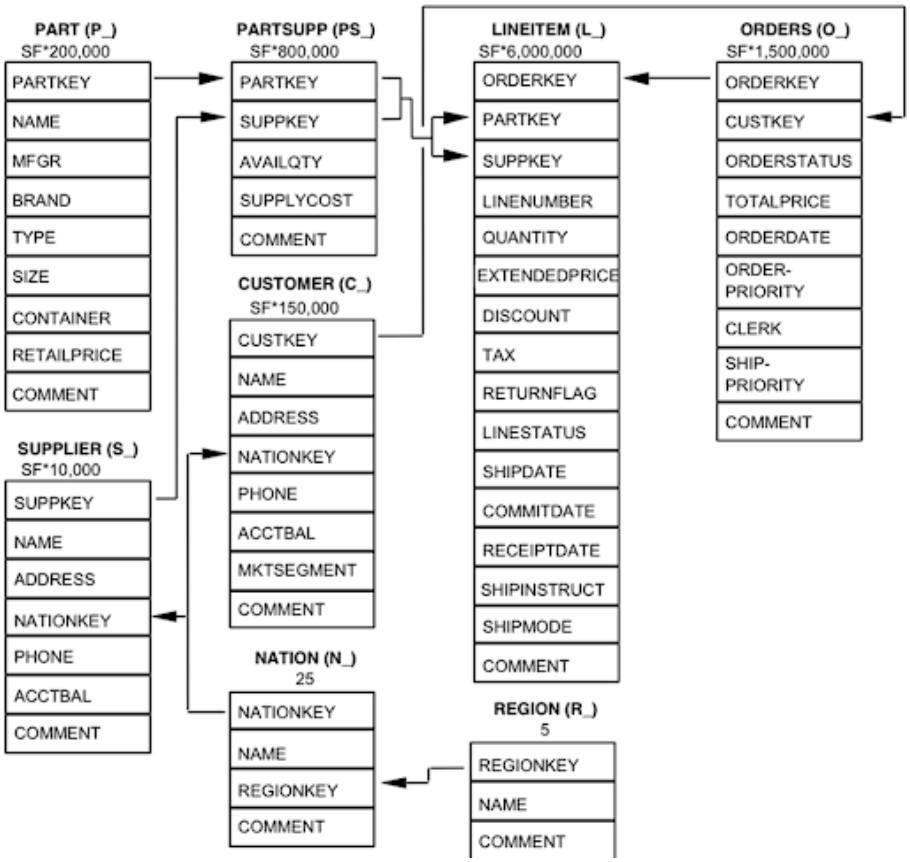


Figure 5.1: TPC-H Schema (taken from [44]).

For all of our experiments, we start by initializing the servers, waiting for them to establish a connection with each other and share the list of buckets they replicate. Consequently, the bucket mapping is created before we start the client to make requests to the servers. Then the client loads the initial database state into the servers, considering that the data is replicated depending on its location. After the data is loaded to the servers, the client starts executing requests to one of the servers. Depending on the test, we vary the number of threads being run on the client. Throughout the test, we increased the ratio

of remote transactions. A transaction is considered remote when its operations involve objects not replicated locally and fetched from other replicas.

Tests are run for various percentages of remote transactions. For each of these percentages of remote transactions, the client executes requests for 2 minutes. By default, each request consists of a transaction with three reads and two writes of 2 objects of the *Order* type.

Since remote objects do not need to be fetched multiple times during a transaction, and the same two objects are used throughout the transaction, the first operation of the transaction is the operation responsible for fetching objects from another replica. Therefore, to test the system's behaviour when it was necessary to fetch remote objects for both reads and writes, it was required to implement two distinct sequences for the execution of operations in a transaction. The two sequences of requests adopted by the majority of the experiments performed were:

1. $read(obj_1, obj_2) \rightarrow upd(obj_1, obj_2) \rightarrow read(obj_1, obj_2) \rightarrow upd(obj_1, obj_2) \rightarrow$
   $\rightarrow read(obj_1, obj_2)$
2. $upd(obj_1, obj_2) \rightarrow read(obj_1, obj_2) \rightarrow read(obj_1, obj_2) \rightarrow upd(obj_1, obj_2) \rightarrow$
   $\rightarrow read(obj_1, obj_2)$

The updates consist of changing the O_ORDERSTATUS variable. When transactions are remote, one or both objects can be located on another server.

As we test different percentages of remote transactions in the same run, we must ensure that we provide the same object conditions for all ratios of remote transactions. Therefore, we distribute all the objects by the different tests of the same execution, dividing them by the remote transactions we test. This way, we ensure that all objects are in their original state when testing starts.

5.2 Results

This section presents and discusses the results obtained from the experiments we carried out for different scenarios. The experiments carried out seek to answer the questions we formulated at the beginning of the chapter.

For all of our experiments, we vary the number of clients to determine how our system scales. For example, we use metrics such as the number of operations per second (ops/s) executed by all clients and the average latency of each request to determine if the system behaves as expected when we increase the number of threads and the number of remote transactions.

5.2.1 System Scalability

Figure 5.2 shows that latency increases as the number of threads increases with the ratio of remote transactions, as expected. For a given ratio of remote transactions, as the

number of client threads increase, the throughput increases while the latency remains stable until the point where the servers get saturated and throughput stops increasing. When comparing the different ratios of remote transactions, the system saturates at a lower throughput as the ratio of remote transactions increases.

The throughput evolution according to the number of threads can be seen in more detail in Figure 5.3(a), where each line represents the percentage of remote transactions, and each point on each line represents the number of client threads. The number of client threads necessary to saturate the server increases with the ratio of remote transactions, with the system saturating around 1000 client threads for 50% or more of remote transactions, while for 10% of remote transactions, the system saturates with 500 client threads. This is expected as the processing of a remote transaction includes periods during which the transaction is idle, waiting for replies from the remote replica. Thus, we need more clients to be able to saturate the server.

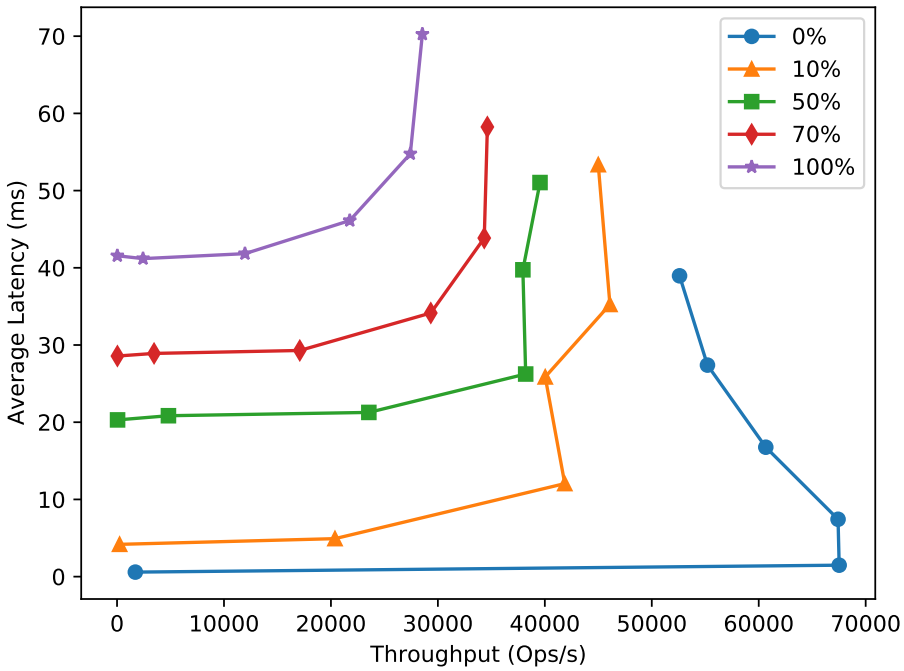
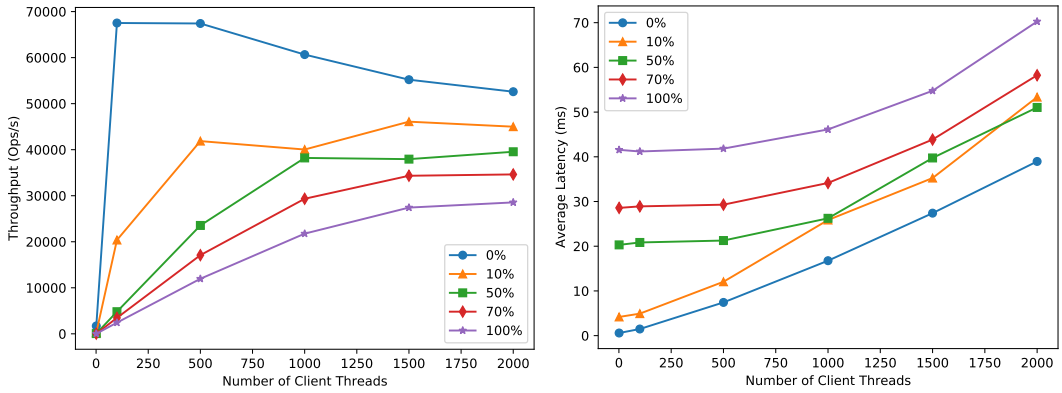


Figure 5.2: System performance for random object distribution.

5.2.2 Remote Operations Ratio Impact

To understand how the ratio of remote transactions influences the system's performance, we can use the same graphs from the section above. From Figure 5.2, we can observe that as the number of remote transactions increases, the latency also increases and the number of operations executed per second for the same number of clients, and the maximum number of operations, decreases. These data can be seen in more detail in Figures 5.3(a) and 5.3(b).



(a) System throughput.

(b) System latency.

Figure 5.3: System performance for a random distribution of objects, with latency and throughput being analysed independently.

This increase in latency and decrease in the number of operations due to the increase in remote transactions was expected. Whenever a request is made on a particular object for the first time in a transaction, it is necessary to fetch the object from another server. On that account, waiting for the object to be returned and applying the missing updates increases the operation's execution time. Much of the increase in execution time is due to the time it takes to transmit a remote object's request and its response, as this time is based on how far the servers are from each other (latency between servers is shown in Table 5.1).

5.2.3 Impact of the Order of Operations

For evaluating how the order of operations could influence the system's performance, we modified the order of requests, as follows:

$$read(obj_1, obj_2) \rightarrow read(obj_1, obj_2) \rightarrow upd(obj_1, obj_2) \rightarrow upd(obj_1, obj_2) \rightarrow upd(obj_1, obj_2)$$

Consequently, instead of reading the objects, updating them and rereading them, we used a sequence of operations that perhaps better represents the normal flow of operations in a real context. Thus, when adopting this sequence of operations, we first read the objects we intend to change and apply the writes without checking the updates we made by rereading the objects. Objects do not need to be fetched multiple times in a transaction. Therefore, since the same two objects are used during the transaction, the first operation ($read(obj_1, obj_2)$) fetches the remote objects used by all transaction operations. So in this experiment, remote objects are only fetched for reading purposes.

Comparing the results obtained in Figure 5.2 with those in Figure 5.4, it is possible to conclude that the results obtained for the two experiments are similar. The tests allow us to infer that considering the number of objects accessed per transaction, the sequence of operations performed does not affect the system's performance. When a read is performed after an update in a transaction, it is necessary to temporarily apply

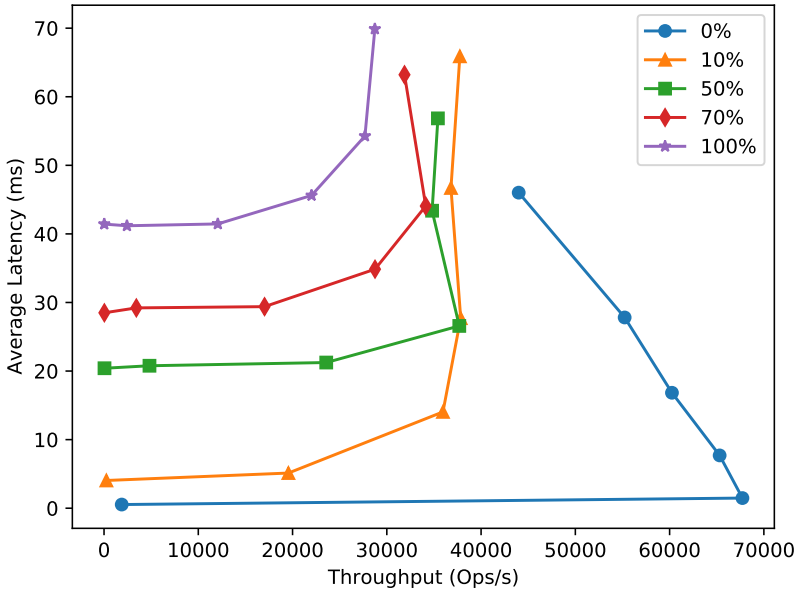


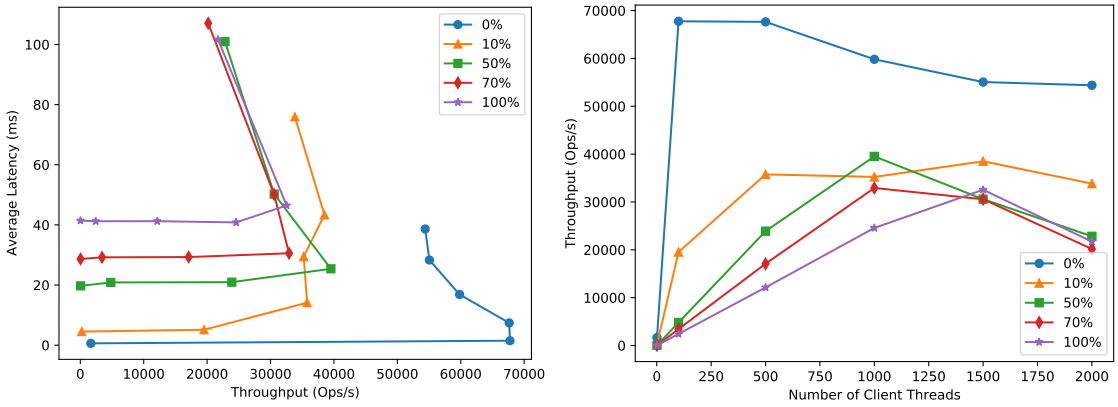
Figure 5.4: System performance for transactions where reads are performed before writes.

the updates to the object's latest version to observe the effects of the updates performed during the transaction. This experiment revealed that applying the missing updates to execute a read operation in the transaction has no significant effect on the system performance, meaning that the cost of performing these updates is low. These results also suggest that the cost of fetching remote objects from another replica for reading purposes is similar to fetching them for update purposes, as one could expect. As the objects used throughout a transaction are the same, remote objects are fetched from another replica in the first operation. Comparing this experience in which objects were only fetched for reading purposes with the experience in which objects were fetched for reading and writing purposes, we see that the results are similar. Therefore this distinction has no impact on system performance.

5.2.4 Impact of Remote Updates

To assess whether system performance improved when operations performed on remote objects are only reads, we had to modify the workload to stop making update requests for remote objects. In this experiment, requests for local objects can still consist of reads and updates, but they consist exclusively of reads for remote objects. This means that when a remote transaction is executed, while the three reads can target remote objects, write are always performed on locally replicated objects.

Figure 5.5 shows the results obtained from updating and reading locally replicated objects and from only reading remote objects. The results obtained in this experiment are similar to those obtained when also writing remote objects (5.2.1), with a slight difference



(a) System throughput-latency.

(b) System throughput.

Figure 5.5: System performance for when remote operations consist exclusively of reads.

in behavior after reaching the maximum throughput - the performance decreases, which can be justified by the fact that in this case all writes need to be executed locally, while when there are remote transactions, some writes are not applied locally. It is not clear what justifies this difference.

The similarity between the results in the two experiments is understandable since, in reality, to perform both writes and reads on remote objects, it is necessary to fetch the object to a replica that replicates the object, representing a significant part of the execution of a request. The only difference between the two types of operations lies in the operations applied to the remote object after being fetched (operations can be reads or updates). This difference is independent of whether the object is local or remote since operations are performed locally. Furthermore, updates applied to objects are propagated asynchronously and therefore do not compromise transaction execution.

5.2.5 Distribution of Objects

For the experiments mentioned above, clients' orders use objects chosen randomly, which means that we randomly select those that the transaction will access from the set of objects. To understand how the distribution we use to access objects influences the system's performance, we changed our experiments to choose the objects following the Zipf Distribution. By using Zipf Distribution[1], we guarantee that some objects are accessed more frequently than others. This distribution allows us to have an approximation of a real context in which there are objects more "popular" than others. The Zipf distribution is based on Zipf's law, which says that when we have a set of objects, the nth object will appear 1/nth times as often as the most frequent object [45].

[1] To implement Zipf Distribution on the client, we used the package rand, which allows us to use the *NewZipf* func [33]: *func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf*

This func returns a Zipf variate generator that generates values $k \in [0, imax]$ such that $P(k)$ is proportional to $(v+k)^{-s}$. For our experiments, we used s=2 and v=1.

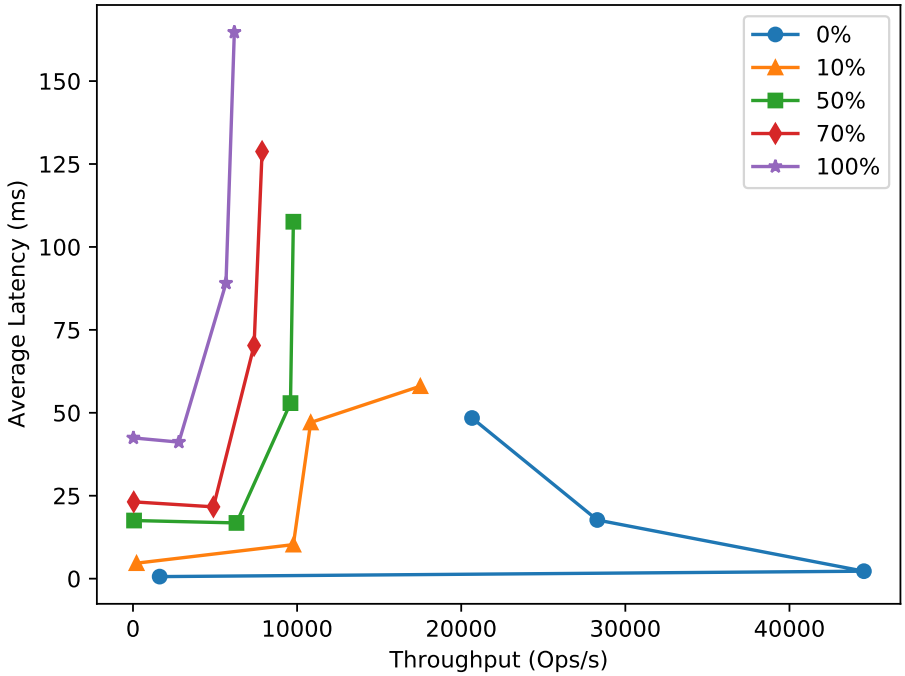


Figure 5.6: System performance for Zipf object distribution.

Figure 5.6 shows the results obtained from experiments with the objects being accessed following the Zipf Distribution. For this experiment, we ran the tests only for 1, 100, 500 and 1000 threads. Comparing this figure with Figure 5.2, we can see that the latency increased and the throughput reduced comparatively to the random distribution. The decrease in performance when some objects are accessed more frequently than others was expected since the access to the same object induces contention on the access to the object - in PotionDB, accesses to the same objects are serialized in the replica.

5.2.6 Impact of the Number of Objects Accessed

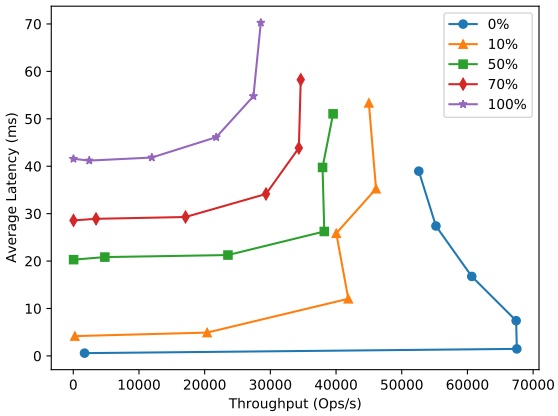
This section evaluates how varying the number of objects accessed influences the system's performance, carrying out experiments for different numbers of objects per operation. As described in Section 5.1.1, the experiments are performed by accessing two objects per transaction by default. Thus, five operations are executed for each transaction, three reads and two writes, and each operation accesses the same two objects. In this section, we experiment with different numbers of objects per transaction and different numbers of objects per operation. The various experiments carried out in this section are shown in Table 5.2, with the configuration defined by default being represented in the first line.

Figure 5.7 shows the results obtained. The results show that, in general, the throughput decreases according to the number of objects accessed. This result is justified because operations access more objects, on average. Another factor that is important to consider is that as the number of objects used in the transaction increases if there are remote objects, it will be necessary to fetch these objects from remote replicas. If no replica replicates all

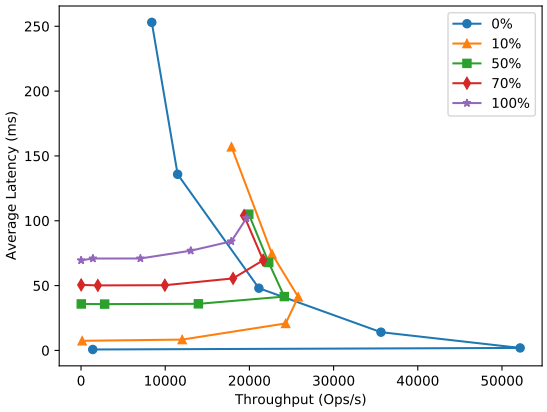
Number of objects	**Read operations** (x3)	**Write operations**
2 objects: 2 for reads and 2 for writes	$read(obj_1, obj_2)$	$upd(obj_1, obj_2)$ (x2)
4 objects: 4 for reads and 2 for writes	$read(obj_1, obj_2, obj_3, obj_4)$	$upd(obj_1, obj_2)$ $upd(obj_3, obj_4)$
6 objects: 6 for reads and 2 for writes	$read(obj_1, obj_2, obj_3, obj_4, obj_5, obj_6)$	$upd(obj_1, obj_2)$ $upd(obj_3, obj_4)$
6 objects: 6 for reads and 4 for writes	$read(obj_1, obj_2, obj_3, obj_4, obj_5, obj_6)$	$upd(obj_1, obj_2, obj_3, obj_4)$ $upd(obj_1, obj_2, obj_5, obj_6)$

Table 5.2: Number of objects used in experiments of Section 5.2.6 and their distribution across the operations of a transaction.

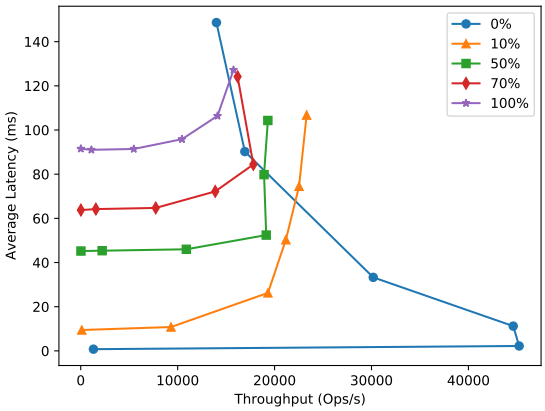
the objects to be fetched, it is necessary to request the objects to different remote replicas, increasing system latency.



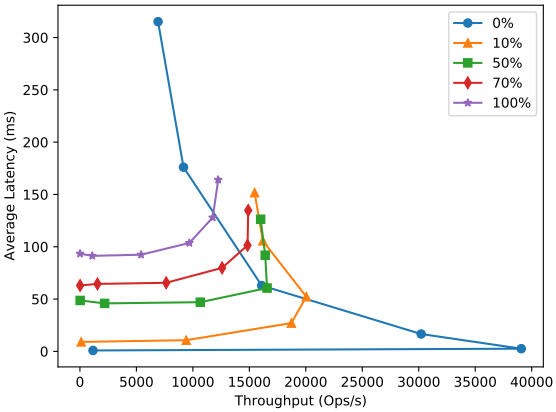
(a) 2 objects for reads and 2 for writes.



(b) 4 objects for reads and 2 for writes.



(c) 6 objects for reads and 2 for writes.



(d) 6 objects for reads and 4 for writes.

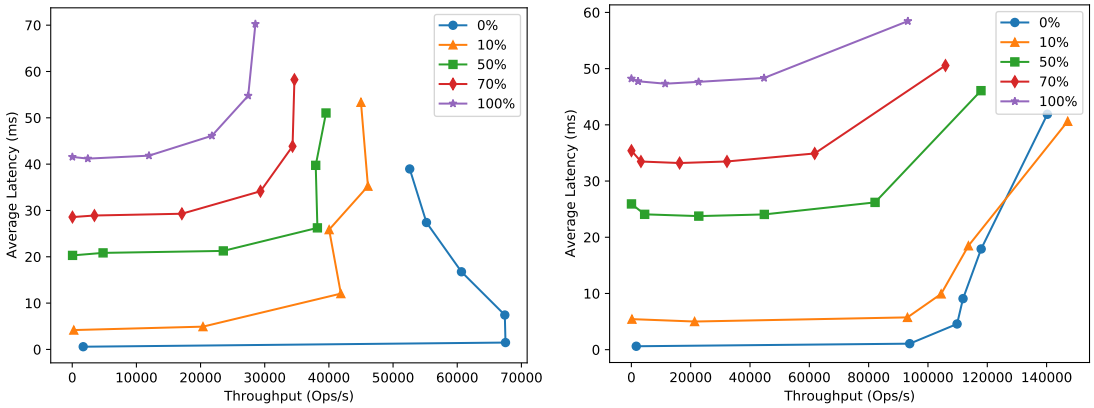
Figure 5.7: Comparison of system performance for different numbers of objects per transaction and per operation.

The results show that the increase of accessed objects resulted in a considerable decrease in performance when all transactions are local. The justification for this result lies

in the fact that more operations are being performed - note that a read over two objects or over six objects is always counted as one operation. Consequently, if throughput were measured concerning the number of objects read/written, multiplying Ops / s by the number of objects involved in the transaction would reveal an increase in throughput for transactions with more objects.

5.2.7 Impact of Concurrency

For all the experiments mentioned above, the threads running on the client execute requests addressed to the same server. In this experiment, each thread was assigned a server to which they address all requests, simulating the scenario where all replicas are being concurrently modified. It is relevant to mention that each thread always performs requests to the same server throughout the test, allowing us to have an approximation of a real context in which clients access the closest data center.



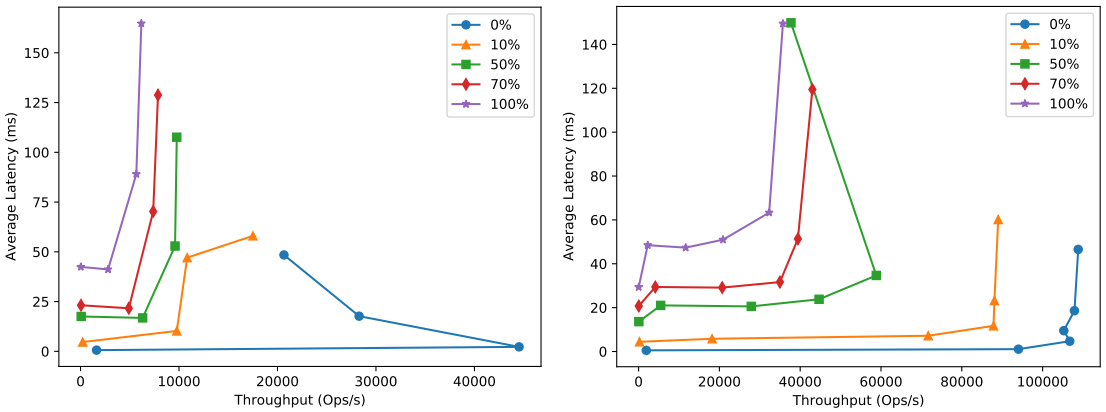
(a) All requests are addressed to the same server. (b) Requests are distributed across multiple servers.

Figure 5.8: Impact of concurrency on system performance for random object distribution.

Figure 5.8 shows the results of this experiment for random object distribution, with Figure 5.8(a) representing the results obtained in Section 5.2.1

In Figure 5.8(a), clients executed requests addressed to the same server. By comparing these results with those obtained throughout this section, the impact of concurrency becomes clearer. Therefore, through Figure 5.8(b), where requests are distributed across several servers, it is possible to verify that it was necessary to increase the number of client threads to saturate the system. In this experiment, a higher number of threads is necessary for the system performance to start to deteriorate because the client requests are distributed over several servers. As all servers have associated threads, the workload is distributed among different servers, unlike what happens in the other experiments.

Similar to what happens for a random object distribution, in Figure 5.9, we witness that if the system adopts a Zipf object distribution, its performance also improves when we distribute client requests across multiple servers. In this case, Figure 5.9(a) represents



(a) All client are addressed to the same server. (b) Requests are distributed across multiple servers.

Figure 5.9: Impact of concurrency on system performance for Zipf object distribution.

the results obtained in Section 5.2.5, where we only had to run tests up to 1000 threads to observe the system saturating. When distributing the requests to several servers, it was necessary to increase the number of threads to saturate the system. Therefore we also ran tests for 2000 and 5000 threads. By comparing the results for the same number of client threads, we verify that system performance is better for requests distributed across multiple servers.

For the other experiments, a single server executed requests from all threads, which implies that this server had to respond to all requests and, therefore, saturated sooner. As for the latency, the impact of the execution of operations in multiple replicas is minimal, with a slight increase. This is expected in our setting, as when a server s_i requests a version of an object to a remote server s_r, it is expected that s_r has already received all updates that had been previously received by server s_i. Thus, it will be unlikely that this causes a significant delay in the execution of the operation. Another small source of delay, in this case, is the fact that, in the remote replica, it might be necessary to recreate the request version. Again, the impact of this seems to be minimal.

5.3 Summary

In this chapter, after describing the experimental setup and the configurations used in our evaluation, we presented the results obtained for our experiments.

The results show that our system scales to a certain extent with the increase in the number of clients. The system performance decreases with the ratio of remote operations since the more operations need to access a remote server, the more transactions are impacted by this additional latency. Systems that adopt a partial replication model and partition data in a way that clients mostly execute transactions on data replicated in the nearest data center. Therefore, although the evaluation considers several remote transaction ratios, this ratio is likely to be low in a real context. The results also showed

a decrease in system performance when some objects are accessed more frequently than others. This is because the number of accesses for these objects increases, and some contention occurs in the access to those objects.

The developed system represents an extension of PotionDB that makes it possible to execute transactions that may involve data not locally replicated. However, after objects are fetched from other replicas, they are treated as replicated locally. The only difference is that instead of operations being executed in Materializer, they are executed directly in TransactionManager using structures specially created for the objects remotes. Therefore, it is understandable that the system overhead is related to the moment objects are fetched from other replicas. Thus, the developed system fulfils the goal of executing transactions over geo-partitioned data without significantly compromising the system performance.

6

Conclusion

Geo-replication is a key technique used to provide low latency and high availability across global services. As the number of data centers worldwide increases and data managed by services also increases, it becomes more challenging to replicate all data across all data centers, making partial replication an interesting approach. However, executing transactions on geo-partitioned data while ensuring transactional causal consistency is a challenge. Therefore, the goal of this work was to design and implement a solution that would allow executing transactions on geo-partitioned data with transactional causal consistency semantics. To this end, our work proposed a new algorithm to execute transactions over geo-partitioned data and a design and implementation of such algorithm in PotionDB, a weakly consistent geo-replicated key-value store that supports partial replication, the PotionDB.

The main functionalities designed and implemented throughout this work can be summarized as follows:

- A mechanism to maintain information on which replicas replicate each object, which is used to decide which replica to contact when the client tries to perform an operation on an object that is not replicated locally;
- A mechanism that allows optimizing accesses to remote replicas, by taking into consideration the operations known to have been received in other replicas, by grouping requests to remote replicas and by maintaining in each replica the list of recently committed transactions, which allows that a remote replica replies before receiving the request from the replica sending the request;
- An algorithm that enforces transaction causal consistency semantics for transactions over geo-partitioned data.

Our evaluation showed that fetching remote objects from another replica is the main source of overhead justified by the fact that whenever a transaction is executed involving objects that are not replicated locally, it is necessary to fetch them the other replies.

Therefore, the system performance decreases as the number of accesses to remote replicas increases since the latency between servers defines the time it takes for a server to send a request and receive the corresponding response. The server can only perform the operations that the client requested for the remote objects after receiving a response from another server with the remote objects. However, in systems that adopt partial replication, it is rare for clients to want to execute transactions on data that the nearest data center does not replicate. Consequently, we can expect that the operation of fetching objects ends up being performed infrequently. Therefore, we consider that our system meets the overall goal of our work by executing transactions over geo-partitioned data with an acceptable impact on the system's overall performance.

6.1 Future Work

Although we had implemented a solution that met the proposed objectives and the evaluation revealed the expected results, our solution could be improved to optimize the system's performance. The optimizations we suggest as future work focus mainly on reducing the times that remote objects are fetched from other replicas. Thus, we consider it interesting to implement a system optimization that would allow storing remote objects in the cache and using them between transactions if the objects in question have not been updated.

First, we contemplate an optimization that reduces the number of unnecessary accesses to other replicas by checking if the clock entry for the replica has changed since the last time it fetched a remote object. The clock represents the version of the database that is accessed during a transaction. Suppose that the only clock entry that has advanced since the last transaction was the entry associated with the local replica. In this case, the only replica that performed updates on the remote objects to be fetched was the local replica. Consequently, since the remote replicas did not perform any new updates, it is not necessary to fetch the objects again because the state of the objects has not changed. As a result, in this case, where the snapshot version is the same as the cache version for all other servers, the local replica could use the cached version.

In order to improve system performance, we also consider executing an optimization based on the fact that the advance of clock entries implies that updates have occurred in those replicas but not necessarily in the objects that the local replica intends to fetch. Thus, similarly to what was implemented in our work, when a replica requests remote objects from another replica, it specifies the version of the object it wants to obtain. However, in this case, the version of the object that the local replica keeps in cache is also sent. Through this information, the remote replica checks if the version of the sent object is out of date. If there was no update in the meantime, a message was sent saying that "the object is up to date", meaning that the local replica could use the version stored locally. In this case, it was necessary to access the remote replica, but it was not necessary to transfer the object again.

Bibliography

[1] *A technical overview of Azure Cosmos DB | Azure Blog and Updates | Microsoft Azure*. https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/. (Accessed on 02/21/2021) (cit. on p. 27).

[2] *Adding simulated network latency to your Linux server - Benjamin Cane*. https://bencane.com/2012/07/16/tc-adding-simulated-network-latency-to-your-linux-server/. (Accessed on 11/27/2021) (cit. on p. 52).

[3] D. D. Akkoorath et al. "Cure: Strong semantics meets high availability and low latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 405–414 (cit. on pp. 2, 7, 8, 21, 25, 26, 29, 30, 34, 37).

[4] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 85–98. ISBN: 9781450319942. DOI: 10.1145/2465351.2465361. URL: https://doi.org/10.1145/2465351.2465361 (cit. on p. 1).

[5] P. Bailis et al. "Scalable Atomic Visibility with RAMP Transactions". In: *ACM Trans. Database Syst.* 41.3 (July 2016). ISSN: 0362-5915. DOI: 10.1145/2909870. URL: https://doi.org/10.1145/2909870 (cit. on p. 17).

[6] B. H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692 (cit. on p. 19).

[7] M. A. Bornea et al. "One-copy serializability with snapshot isolation under the hood". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 625–636. DOI: 10.1109/ICDE.2011.5767897 (cit. on p. 16).

[8] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A Distributed Metadata Service for Causal Consistency". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 111–126. ISBN: 9781450349383. DOI: 10.1145/3064176.3064210. URL: https://doi.org/10.1145/3064176.3064210 (cit. on p. 2).

[9] G. Cabrita and N. Preguiça. "Non-Uniform Replication". In: *arXiv preprint arXiv: 1711.07733* (2017) (cit. on pp. 10, 11).

[10] *Consistency Models*. https://jepsen.io/consistency. (Accessed on 02/19/2021) (cit. on p. 6).

[11] J. C. Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22 (cit. on pp. 26, 27).

[12] *DBMS - Transaction*. https://www.tutorialspoint.com/dbms/dbms_transaction.htm. (Accessed on 11/17/2021) (cit. on p. 11).

[13] G. DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (cit. on pp. 8, 21, 23, 24).

[14] *Distributed Systems : Distributed Lookup Services, Distributed Hash Tables*. 2012. URL: https://www.cs.rutgers.edu/~pxk/417/notes/23-lookup.html (visited on 11/11/2020) (cit. on pp. 1, 22).

[15] J. Du, S. Elnikety, and W. Zwaenepoel. "Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks". In: *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. 2013, pp. 173–184. DOI: 10.1109/SRDS.2013.26 (cit. on p. 12).

[16] P. Fouto, J. Leitão, and N. Preguiça. "Practical and Fast Causal Consistent Partial Geo-Replication". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018, pp. 1–10. DOI: 10.1109/NCA.2018.8548067 (cit. on p. 2).

[17] D. K. Gifford. "Information Storage in a Decentralized Computer System". In: (1981). AAI8124072 (cit. on p. 27).

[18] 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. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on p. 21).

[19] *GitHub*. https://github.com/. (Accessed on 11/01/2021) (cit. on p. 44).

[20] *GitHub - AntidoteDB/antidote: A planet scale, highly available, transactional database built on CRDT technology*. https://github.com/AntidoteDB/antidote. (Visited on 12/30/2020) (cit. on p. 24).

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902 (cit. on pp. 11, 12).

[22] *Home | CloudPing - AWS Latency Monitoring*. https://www.cloudping.co/grid. (Accessed on 10/12/2021) (cit. on p. 52).

[23] *INForum | inforum.org.pt*. https://inforum.org.pt/en/inforum. (Accessed on 11/14/2021) (cit. on p. 4).

[24] L. Lamport et al. "Paxos made simple". In: (2001) (cit. on pp. 26, 42).

[25] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563 (cit. on p. 30).

[26] W. Lloyd et al. "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. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 401–416. ISBN: 9781450309776. DOI: 10.1145/2043556.2043593. URL: https://doi.org/10.1145/2043556.2043593 (cit. on p. 1).

[27] *Log - AntidoteDB*. https://antidotedb.gitbook.io/documentation/architecture/log. (Accessed on 12/31/2020) (cit. on p. 24).

[28] *Materializer - AntidoteDB*. https://antidotedb.gitbook.io/documentation/architecture/materializer. (Accessed on 12/31/2020) (cit. on p. 25).

[29] *Overview - AntidoteDB*. https://antidotedb.gitbook.io/documentation/architecture/overview. (Visited on 12/30/2020) (cit. on pp. 24, 25).

[30] C. H. Papadimitriou. "The Serializability of Concurrent Database Updates". In: *J. ACM* 26.4 (Oct. 1979), pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158. URL: https://doi.org/10.1145/322154.322158 (cit. on p. 7).

[31] S. Peluso, P. Romano, and F. Quaglia. "SCORe: A Scalable One-Copy Serializable Partial Replication Protocol". In: *Middleware*. 2012 (cit. on p. 16).

[32] N. Preguiça. "Conflict-free Replicated Data Types: An Overview". In: *arXiv preprint arXiv:1806.10254* (2018) (cit. on pp. xiv, 8–10).

[33] *rand package - math/rand - pkg.go.dev*. https://pkg.go.dev/math/rand. (Accessed on 11/10/2021) (cit. on p. 58).

[34] *Replication | Cloud Spanner | Google Cloud*. https://cloud.google.com/spanner/docs/replication?hl=en. (Accessed on 02/24/2021) (cit. on p. 26).

[35] A. Rijo, C. Ferreira, and N. Preguiça. "Global Views on Partially Geo-Replicated Data". In: (). Unpublished (cit. on pp. 1–3, 12, 31).

[36] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350 (cit. on p. 21).

[37] N. Schiper, P. Sutra, and F. Pedone. "P-Store: Genuine Partial Replication in Wide Area Networks". In: *2010 29th IEEE Symposium on Reliable Distributed Systems*. 2010, pp. 214–224. DOI: 10.1109/SRDS.2010.32 (cit. on p. 15).

[38] M. Shapiro et al. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400 (cit. on pp. 9, 10, 30).

[39] Y. Sovran et al. "Transactional Storage for Geo-Replicated Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 385–400. ISBN: 9781450309776. DOI: 10.1145/2043556.2043592. URL: https://doi.org/10.1145/2043556.2043592 (cit. on p. 2).

[40] Y. Sovran et al. "Transactional Storage for Geo-Replicated Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 385–400. ISBN: 9781450309776. DOI: 10.1145/2043556.2043592. URL: https://doi.org/10.1145/2043556.2043592 (cit. on p. 13).

[41] K. Spirovska, D. Didona, and W. Zwaenepoel. *PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication*. 2019. arXiv: 1902.09327 [cs.DC] (cit. on p. 19).

[42] I. Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160 (cit. on p. 21).

[43] *TPC-H Homepage*. http://www.tpc.org/tpch/. (Accessed on 02/12/2021) (cit. on p. 53).

[44] *Transaction Processing Performance Council (TPC). TPC BENCHMARKTM H(Decision Support) Standard Specification Revision 3.0.0*. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf. (Accessed on 11/09/2021) (cit. on p. 53).

[45] *What is Zipf's Law? - Definition from WhatIs.com*. https://whatis.techtarget.com/definition/Zipfs-Law. (Accessed on 11/10/2021) (cit. on p. 58).