Ema Rodrigues Vieira

Bachelor in Computer Science and Engineering

An edge enabled replication kernel for CRDTs

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser:	João Carlos Antunes Leitão, Assistant Professor, NOVA University of Lisbon
Co-adviser:	Nuno Manuel Ribeiro Preguiça, Associate Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

Abstract

Distributed storage systems are essential in supporting large-scale on-line services. The design of these systems has evolved in order to improve latency and availability, so that they can provide users with the best possible experience. Several new storage systems have surfaced that make the most of sophisticated replication techniques and weaker consistency models, with the purpose of fulfilling users' expectations.

Even though these solutions effectively improve latency and availability, they tend to resort to simplistic conflict resolution policies (e.g., last-writer-wins), making them inadequate for some applications. Conflict-free Replicated Data Types (CRDTs) were created to allow replicas to apply operations concurrently without any coordination, offering elaborate concurrency semantics for many data types that can be used to configure conflict resolution policies adapted to the needs of different applications. While some recent systems use CRDTs for conflict resolution, they either fail to take advantage of genuine partial replication, or they do so on the client-side, thus overwhelming resource-limited client devices by making them store partial replicas of the database and perform an excessive amount of computations.

In this work, we propose a novel CRDT replication kernel that simplifies the task of leveraging on genuine partial replication, offers causal+ consistency guarantees, and provides efficient synchronization mechanisms built on top of CRDTs to offer configurable conflict resolution policies. Furthermore, we aim to develop this replication kernel with the edge computing paradigm in mind, so as to improve client perceived latency while exploiting data locality. Finally, we plan to experimentally evaluate the replication kernel by implementing a simple storage system prototype and demonstrating its applicability under specific circumstances.

Keywords: Distributed Storage Systems, Causal+ Consistency, Geo-Replication, Genuine Partial Replication, CRDTs.

Resumo

Os sistemas de armazenamento distribuídos são essenciais no apoio a serviços online de grande escala. O desenho destes sistemas tem evoluído no sentido de melhorar a latência e a disponibilidade, para que possam proporcionar aos utilizadores a melhor experiência possível. Têm surgido vários novos sistemas de armazenamento que tiram máximo partido de técnicas de replicação sofisticadas e de modelos de consistência mais fracos, com o objetivo de satisfazer as expectativas dos utilizadores.

Apesar destas soluções efetivamente melhorarem a latência e a disponibilidade, elas tendem a recorrer a políticas de resolução de conflitos simples (e.g., last-writer-wins), o que as torna inadequadas para algumas aplicações. Os Conflict-free Replicated Data Types (CRDTs) foram criados para permitir que réplicas apliquem operações concorrentemente sem coordenação, oferecendo semânticas de concorrência elaboradas para muitos tipos de dados que podem ser utilizadas para configurar políticas de resolução de conflitos adaptadas às necessidades de diferentes aplicações. Embora alguns sistemas recentes utilizem CRDTs para resolução de conflitos, estes não costumam tirar partido de replicação parcial genuína ou, quando tiram, fazem-no do lado do cliente. Isto pode levar à sobrecarga dos dispositivos de clientes que têm recursos limitados, pois fá-los armazenar réplicas parciais da base de dados e efectuar uma quantidade excessiva de computações.

Neste trabalho, propomos um novo núcleo de replicação de CRDTs que simplifica a utilização de replicação parcial genuína, oferece garantias de consistência causal+ e fornece mecanismos de sincronização eficientes construídos sobre CRDTs, para oferecer políticas de resolução de conflitos configuráveis. Além disso, pretendemos desenvolver este núcleo de replicação tendo em mente o paradigma de computação na berma, de modo a melhorar a latência percebida pelo cliente e, simultaneamente, explorar a localidade dos dados. Por fim, planeamos avaliar experimentalmente o núcleo de replicação implementando um simples protótipo de sistema de armazenamento e demonstrando a sua aplicabilidade sob circunstâncias específicas.

Palavras-chave: Sistemas de Armazenamento Distribuídos, Consistência Causal+, Geo-Replicação, Replicação Parcial Genuína, CRDTs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed storage systems are essential in supporting large-scale on-line services. Today's web applications are becoming increasingly demanding when it comes to offering their users the best possible experience [12]. As such, they need scalable databases that provide high availability and low user perceived latency. Traditional ACID databases are not able to meet these requirements because they choose to offer strong consistency over availability [3]. Recently, numerous new storage systems have emerged that combine weaker consistency models with replication techniques as a way to remain responsive and available [2, 5, 9, 13, 16, 25, 27, 28, 34].

Many large-scale on-line services have users spread around the globe. As such, they tend to use storage systems that are geo-replicated, i.e., systems that have replicas of the database in strategic geographical locations in order to be close to a majority of their users, thus improving the response times of applications while making the system fault tolerant. To take full advantage of data locality, some systems combine geo-replication with partial replication [9, 16, 25, 34]. This combination allows replicas to store only data objects that are accessed by the users of their region, thus requiring less resourceful devices to store and manage the data. Some of these systems opt for server-side approaches [9, 16] and store replicas in the cloud, while others bring the storage and computation to the client devices [25, 34].

Relying solely on cloud infrastructures tends to increase client perceived latency. This happens because the network may become a bottleneck when all client requests need to be forwarded to distant servers in the cloud. On the other end of the spectrum, moving storage and computation to the client-side is often too heavy for resource poor client devices. As of lately, attention has been brought to the edge computing paradigm, where storage and computation are moved to devices in the periphery of the network that are closer to clients (e.g., ISP servers and 5G towers), thus allowing applications to have lower response times and to take advantage of data locality.

Several recently proposed distributed storage systems have adopted weak (available) consistency models, choosing to offer availability over strong data consistency in order to meet the standards of responsive web applications. While some systems implement eventual consistency [12], many have been proposed that implement causal+ consistency [5, 13, 27, 28], as it provides availability while enforcing a set of guarantees that respect the cause and effect relationship. This allows developers to easily reason about the state of the system and provides sensible semantics for applications.

Most of the solutions that provide causal consistency while trying to minimize the amount of used metadata resort to simplistic conflict resolution policies (e.g., last-writer-wins) which are not adequate for some applications. Conflict-free Replicated Data Types (CRDTs) [29] were created to solve this problem, allowing replicas to apply operations concurrently without any coordination, while still guaranteeing that their states will eventually converge. CRDTs offer sophisticated concurrency semantics for many data types, which can be used to configure conflict resolution policies adapted to the needs of different applications.

Motivation

While some recent systems have opted to use CRDTs for conflict resolution [2, 25, 34], they either fail to take advantage of partial replication, or they do so exclusively on the client-side, thus overwhelming resource-limited client devices by making them store partial replicas of the database and perform excessive amounts of computations. Furthermore, some solutions that support partial replication do so inefficiently, by making partial replicas handle metadata regarding objects that they do not replicate, thus creating additional overhead and introducing false dependencies (i.e., these systems do not offer genuine partial replication).

By leveraging on existing edge nodes, partial replicas of the data can be stored closer to smaller groups of clients. Other than having the advantage of being replicated at a larger set of sites (when compared to the cloud), this decreases end-to-end latency, which is critical for responsive applications [31], and ensures that the durability of data is not compromised while exploiting data locality. However, developing dynamic and scalable replication schemes that leverage on partial replication and are capable of extending towards the edge is a complex task [23]. Firstly, the data items stored at each replica change over time according to the dynamic access patterns of the users of the region. This requires the replication protocols to know where each data item is stored at all times, so that the updates are propagated only to the interested parties. Furthermore, in order to ensure convergence and availability, the protocol needs to adapt and remain correct when replicas fail, or when more replicas are added to the system.

Adding to the dynamic nature of the edge, come the challenges of implementing causally consistent systems. Choosing the right way to track causality is not an easy task due to the trade-off between data freshness and throughput [18]. By tracking causality with more fine grained approaches we incur in storage overhead by increasing the amount of metadata, which can degrade the throughput of the system. However, because the causality information is so detailed, the system is able to make remote updates visible more rapidly. On the other hand, when choosing to track causality with less granular approaches, the compression of metadata causes operations to have false dependencies, increasing the update visibility times [16].

Finally, in order to give support to demanding large-scale web applications, new storage systems need to be scalable. The challenge is to find ways to keep metadata constant and bounded as the number of operations, clients, and replicas of the data store increases.

Problem Statement

To overcome the challenges mentioned above, we plan to explore how to provide more sophisticated concurrency semantics by leveraging on CRDTs, while retaining efficiency and scalability by exploring genuine partial replication at the edge and using minimal metadata to track causality. To that end, we plan to design a CRDT replication kernel that simplifies the task of leveraging on partial replication, offers causal+ consistency guarantees, and provides efficient synchronization mechanisms built on top of CRDTs to offer configurable conflict resolution policies.

Expected Contributions

The contributions we expect to present with this work are the following:

- A new and efficient CRDT synchronization mechanism that offers causal+ consistency guarantees, adapts to membership changes, and supports genuine partial replication by leveraging on highly dynamic and adaptative communication patterns between replicas;
- A replication kernel that leverages on the aforementioned synchronization mechanism;
- A simple data store prototype designed to demonstrate the applicability of our replication kernel under concrete and predefined circumstances;
- An extensive experimental evaluation of the proposed solution compared against current state-of-the-art solutions.

Research Context

The work to be conducted in the thesis is contextualized in the New Generation of Data Storage And Management Systems (NG-STORAGE) national funded project.

This project aims to tackle new emergent challenges in data storage and management systems related with three complementary aspects: i) flexible and large-scale partial replication; ii) self-management of replicas life-cycles, and iii) providing different consistency guarantees when replicating data objects.

By designing a new synchronization mechanism (and prototype storage system) that ensures causal consistency, leverages on CRDTs, manages membership changes, and supports genuine partial replication (at the edge), the results this work is expected to produce will bring relevant advances in the project's context.

Document Structure

The rest of this document is organized as follows:

Chapter 2 discusses related work. It covers topics such as different replication strategies; strong and weak consistency models, focusing primarily on causal (and causal+) consistency and causality tracking mechanisms; CRDTs as a conflict resolution technique that ensures replica convergence; and some relevant existing systems that provide causal+ consistency guarantees.

Chapter 3 further details the proposed approach and presents the planning of future work.

Chapter 2

Related Work

In this Chapter we present and discuss previous related work that is relevant for the context and objectives of this thesis. To help understand the contents of this document, we focus on the following topics:

In Section 2.1 we discuss replication, its benefits and downsides, and some design choices that may affect the performance of replicated storage systems.

In Section 2.2 we discuss several strong and weak consistency models, giving some emphasis to causal and causal+ consistency. Additionally, we cover some of the most common causality tracking mechanisms and conflict resolution techniques.

In Section 2.3 we cover the subject of CRDTs, discussing concurrency semantics and synchronization methods. Furthermore, we present some examples of existing CRDT implementations.

In Section 2.4 we present an in depth description and analysis of some relevant existing replicated data storage systems, focusing mainly on those that use CRDTs to offer configurable conflict resolution policies.

2.1 Replication

Distributed storage systems that support large-scale on-line applications strive to provide their users with optimal experiences. Nowadays, users are becoming more and more demanding, so these systems resort to replicating data across multiple replicas in order to provide some desirable properties:

Availability and fault-tolerance: A replica can become unavailable due to hardware failure, software fault, network partitions, or even due to external events like floods or power outages. By maintaining copies of the data in several replicas, we ensure that the system remains functional in the presence of a large number of replica failures, by guaranteeing that there is always a replica that is available to serve client requests.

High scalability: When several replicas maintain a copy of the data the system can scale to large numbers of clients by using load balancing to distribute the requests across those replicas, particularly read operations (i.e., operations that do not modify the data).

Low latency: By spreading replicas of the system throughout distinct geographical locations (geo-replication), we can ensure that the users in their proximity experience lower user perceived latency, because the time to access the data decreases.

There is, however, a downside to using replication protocols: having multiple copies of the data may lead to consistency problems. When a copy of the data is modified at a given replica it becomes inconsistent with all other replicas. This creates the need for a protocol that propagates the modifications to all the remaining replicas so that they converge to a common state.

2.1.1 Geo-Replication

When replicating a storage system, we could choose to place all the replicas in close physical proximity to each other (e.g., in the same data center, or in neighboring data centers). However, this would not make the system fault-tolerant or available in the face of multiple replica failures caused, for example, by a natural disaster. If such an external event affected the geographical area of the data center(s), it could cause all the replicas to fail simultaneously, rendering the system unusable [1].

To solve this problem we can geo-replicate the system by distributing the replicas across several distant geographical locations, making the system fault-tolerant by reducing the probability of all replicas becoming unavailable at the same time. These replicas are usually placed in strategic locations, so as to be closer to the users of the system, reducing user perceived latency and giving them an overall improved experience.

Because replicas are far apart from each other, the time it takes for them to propagate updates to each other increases. However, these systems tend to prioritize user satisfaction, so they usually opt to give lower response times to users over making the replicas' synchronization process faster.

2.1.2 Replication Schemes

Replication protocols often distribute the data across replicas in a way that trades higher levels of redundancy for less communication and storage requirements.

Full replication: Refers to the approach where every replica of the system maintains a complete copy of the data. This type of replication is very costly when it comes to communication between replicas to propagate updates, because they must be sent to (and executed in) all sites. Additionally, as the data grows so do the storage needs for the system. Oftentimes, it is not reasonable to assume that the replicas have enough resources to store the full dataset.

Partial replication: Refers to an approach in which replicas may not have full copies of the dataset, but only of a subset of it. This means that some replicas may store the same subset, while others store different parts of the full dataset. This scheme greatly reduces the cost of synchronizing replicas, because the updates only need to be sent to the replicas that maintain a copy of the modified data item. However, this adds the additional complexity of keeping track of which replicas store each data item, so that the updates are only propagated to them. Furthermore, this approach reduces the (total) storage requirements of the replicas and, when combined with geo-replication, increases spacial locality, by storing data items only in the places where they are accessed and modified. These properties are essential in edge computing because edge nodes tend to have less resources, and thus less storage space, and because users in a given geographical area are more likely to access the same data items repeatedly.

2.1.3 Replication Models

Replication protocols can use different models when synchronizing replicas (i.e., propagating operations from one replica to the others). There are two important models that choose between low user perceived latency and showing users consistent data.

Synchronous replication: When a system uses synchronous replication it ensures that the operation is propagated to (and executed in) all replicas before sending a response to the client. This type of synchronization is normally employed when trying to provide strong consistency guarantees, because it forces all replicas to have consistent states. Due to the extra steps performed before sending a response to the client, this type of synchronization makes for non-scalable systems (propagating updates takes longer if the number of replicas grows, and if they are geo-replicated), increases user perceived latency, but makes it easier to ensure durability.

Asynchronous replication: On the other hand, asynchronous replication sends a response to the client before propagating the operation to the other replicas. This allows for lower user perceived latency and improves user experience. Because operations are propagated in the background after replying to the client, replicas can present divergent states for certain periods of time, thus allowing clients to have inconsistent views of the system. As such, this type of synchronization is normally used when providing weaker consistency guarantees. These systems tend to scale better than systems with synchronous replication, because the synchronization costs are mitigated by the fast responses to users. Note that if a replica that has not propagated an update fails, the system might not be able to guarantee durability.

2.1.4 Horizontal Partitioning / Sharding

A distributed storage system usually has multiple machines in a single data center. If the same copy of data is stored in all of them, each node will have to execute all updates to data items stored in that data center, hindering the system's scalability. Sharding is a technique that consists in splitting the dataset into several fractions (often called partitions) and storing them on different nodes of the data center.

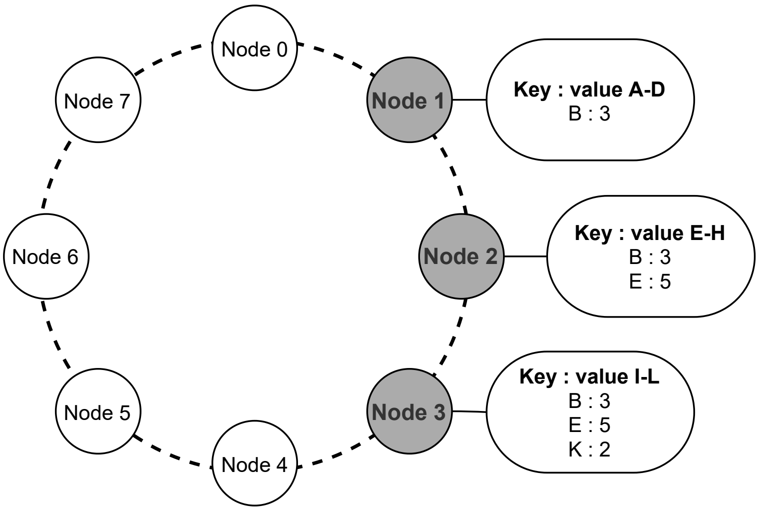

Figure 2.1: Example of sharding in data center organized in a ring.

Consider the data center illustrated in Figure 2.1 where the nodes are organized in a ring structure. This structure can be used to determine the nodes in which a partition is stored. In this example, *Node 1* stores all keys from A to D, *Node 2* stores all keys from E to H, and *Node 3* stores all keys from I to L. For increased tolerance to failures, each partition tends to be replicated in more than one node. In Figure 2.1, all keys that are stored in *Node 1* (i.e., key *B*) were also stored in *Node 2* and *3*, as these are the two closest clockwise neighbors of *Node 1*.

By using this mechanism, the data store is able to process several concurrent requests by distributing the load across the nodes of each data center. Systems that use sharding tend to have good horizontal scalability (scaling out), because they can add more machines if the number of client requests requires it.

2.1.5 Multi-versioning

Some distributed storage systems that offer weak (or available) consistency models use multi-versioning. This technique consists in maintaining multiple older versions of the data items in addition to the most recent one. In systems that provide, for instance, causal consistency, these versions are used to guarantee that the safety properties of the model are never violated.

In a situation where the newest version of a given data item hasn't been propagated to all the relevant replicas, the system could return an older, but stable, version, making it impossible for a client to see a non-monotonic sequence of versions of that item.

Systems that use multi-versioning usually implement some form of garbage-collection mechanism to ensure that older versions are not stored forever or keep growing indefinitely, wasting valuable storage space.

2.2 Consistency Models

The CAP theorem [17] states that it is impossible for a distributed system to provide, simultaneously, the following properties:

1. **Strong consistency:** The clients of the system always see the most recently written value for all data items;

2. **Availability:** All operations issued to non-faulty replicas eventually complete;

3. **Partition tolerance:** The system must remain functional even in the presence of network partitions that may affect the communication between replicas.

This impossibility result implies that a distributed system is only able to fully provide two of the three properties mentioned above. However, in large-scale distributed systems, network partitions are unavoidable and when they happen the system may stop functioning altogether (violating the availability property), or replicas' states can diverge (violating strong consistency guarantees) [10]. As such, most systems opt to be partition tolerant and, in the presence of partitions, they choose between providing availability (AP) or strong consistency (CP).

As mentioned before, nowadays most systems are concerned with providing users with the best experience possible and, consequently, tend to prioritize availability over strong consistency, adopting weaker consistency models.

Consistency model: Specifies how the replicas' states diverge due to write operations. It also refers to a set of safety restrictions that define what the clients can see, given the system's and the client's histories. Note that these restrictions only apply to read operations, so when a storage system does not receive read requests the consistency model is irrelevant.

There are two main categories of consistency models: strong and weak. In the following sections we define them and present some of their advantages and disadvantages.

2.2.1 Strong Consistency

When using strong consistency models, the system gives clients the illusion that a single replica exists despite the fact that it is distributed and that operations are executed in several replicas. These models guarantee that the clients always see the most recent values written in the system. In order to provide the illusion of a single replica, the system must guarantee the following key property:

Atomicity: Given a system trace, there must exist a serialization point for each operation (between the point of its reception and the point in time when the reply is sent to the client) so that the story of the system as a whole, considering the relative order between operations, makes sense. Note that an operation that does not finish (and never returns a reply to the client) may or may not have a serialization point defined. In other words, there must exist a total order for all operations, so that they are observed in the same order by all clients, maintaining the consistency of the data.

By ensuring this, it becomes easy to reason about the possible states of the system, and non-commutative operations can be executed without causing replicas to diverge.

Although it might appear simple to ensure that replicas execute all operations in the same order, it is actually a very complex problem to solve. This requires all replicas to agree on a sequence of operations to execute, effectively achieving consensus. Note that the consensus problem has been proved impossible to solve in the asynchronous model where replicas can fail [15]. Seen as this model is a good approximation of the real world, it is impossible to fully solve the consensus problem, but it is possible to implement algorithms that solve approximations of consensus with relaxed properties, such as Paxos [20].

The downside to these algorithms is that they require several high latency communications steps, for example: a replica must contact a majority of all replicas before confirming the operation to the client, so as to guarantee that the operation is executed in a quorum of replicas, enforcing data consistency. By itself, this hinders the availability of the system in the presence of multiple replica failures (i.e., the system stops if a majority of replicas becomes unavailable). Furthermore, because these systems tend to be geo-replicated, the latency's lower bound depends on the highest Round Trip Time (RTT) between two replicas.

Next, we discuss two of the most relevant strong consistency models:

Linearizability: A system that provides linearizability ensures that replicas execute all operations in the same order. Additionally, that total order must respect the real-time ordering of events, meaning that operations must be executed in the order by which clients issued them.

Serializability: On the other hand, serializability only ensures that there is a total order of operation executions across all replicas, even if that order does not respect the real-time ordering of events. This allows the rearranging of operations, as long as all replicas execute the same sequence in the end.

2.2.2 Weak Consistency

In contrast to strong consistency models, when providing weak consistency the operations do not need to be executed in the same order across all replicas. As such, replicas may diverge and clients may read stale values, or even inconsistent ones.

These systems tend to have lower user perceived latency because replicas have no need to contact a majority quorum before confirming the operation to the client. This implies that the availability of these systems is not compromised even in the presence of multiple replica failures.

Next, we discuss three of the most relevant weak consistency models:

Eventual consistency: Systems that provide eventual consistency are highly available but have no safety guarantees. Because of this, it is debatable if eventual consistency should be considered a consistency model. The only promise these systems make is that eventually, when no more operation requests are issued to the system, replicas will converge to the same state. This implies that, before this happens, the system might allow users to see stale or even inconsistent values. To ensure convergence, conflict resolution techniques (like the ones discussed in Section 2.2.4) must be employed.

Causal consistency: Causal consistency is one of the strongest forms of weak consistency a system can provide while still remaining available. This model guarantees that all replicas see states that respect the happens-before relationship between operations, thus respecting the notion of potential causality [19]. If we consider two operations o_1 and o_2, and a partial order $\prec$ that represents the happens-before relationship, if $o_1 \prec o_2$ then causal consistency guarantees that all replicas can only observe the effects of o_2 after observing the effects of o_1. More formally, we can say that o_1 happened before o_2, $o_1 \prec o_2$, iff one of the following rules applies:

- **Execution thread:** o_1 and o_2 are operations in a single thread of execution, and o_1 happens before o_2;
- **Reads-from:** o_1 is a write operation and o_2 is a read operation that reads the value written by o_1;
- **Transitivity:** Considering another operation o_3, if $o_1 \prec o_3$ and $o_3 \prec o_2$, then $o_1 \prec o_2$.

Note that the happens-before relationship is a partial order, meaning that not all operations are ordered by it. When two operation have no causal relation between them we say that they are concurrent. The effects of two concurrent operations can be observed in any order because they do not depend on each other.

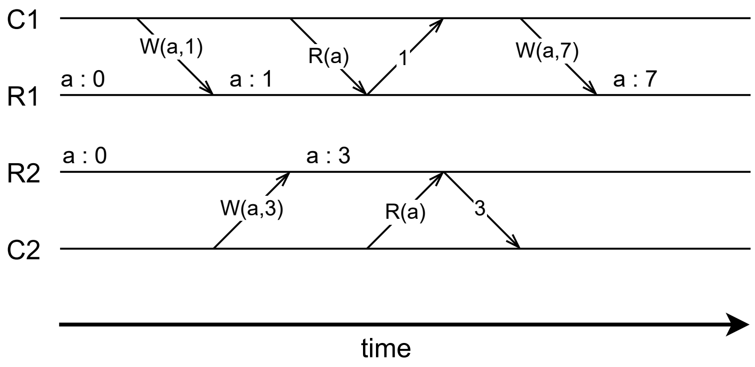

Figure 2.2: Execution with replica divergence in causal consistency.

One important aspect of causal consistency is that it does not guarantee replica convergence. Figure 2.2 presents an execution where the states of two replicas, *R1* and *R2*, diverge under causal consistency. *a* is the key of an object stored in both replicas. As can be seen in this execution, when two clients, *C1* and *C2*, write values for the same key in two different replicas without ever seeing the effects of the other ones' write, their operations do not have causal dependencies between each other (i.e., they are concurrent). As such, they will be executed and the replicas will diverge. If the clients always interact with their corresponding replica (i.e., if they have a sticky session), the replicas will never communicate and their states will diverge forever without ever violating causal consistency.

Causal+ consistency: Causal+ consistency, first coined in [27], combines the properties of causal consistency with those of eventual consistency, providing causal consistency with convergent conflict handling. Causal+ consistency does not order concurrent operations, with the exception of conflicting ones. Two unrelated operations are in conflict if they are writes for the same key happening at two different replicas. If we analyze the example from Figure 2.2 again, we can say that *W(a,1)*, in replica *R1*, and *W(a,3)*, in replica *R2*, are conflicting operations. Contrary to causal consistency, under causal+ consistency this conflict would have to be resolved in order for the replicas' states to converge.

2.2.3 Causality Tracking Mechanisms

There are several ways for a system to enforce causal consistency. As described in [26], some systems rely on the topology of the network to disseminate operations in causal fashion. Whether the nodes are arranged in a star or tree topology, if the channels between them are FIFO (First In, First Out) it is possible to guarantee that the dependencies of an operation have already been propagated at the time of its reception.

Other systems, such as C^3 [16] and Saturn [9], separate the metadata management from the data store itself, and deliver the metadata to each data center while respecting causal order. The data centers have to wait for the reception of the metadata before applying the corresponding operation and making its effects visible to (local) clients.

In peer-to-peer (P2P) networks where all nodes can communicate among themselves the system needs to keep track of causal dependencies between operations. The simplest way to track these dependencies is by using causal histories, explained in detail in [7].

Causal history: The causal history of an event can be represented by the set of all events that happened before it (i.e., all events it causally depends on).

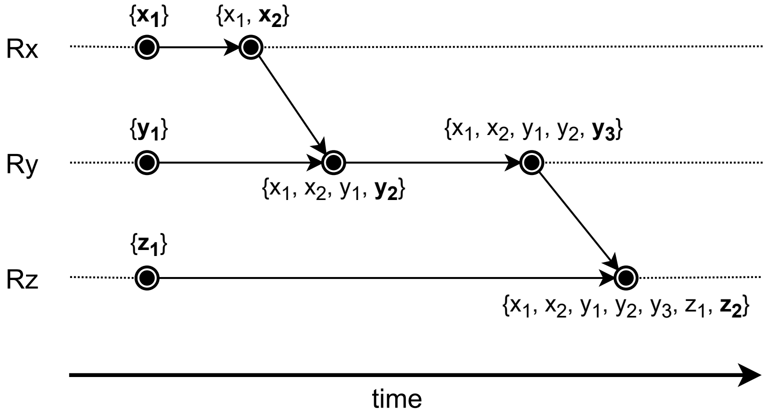

Figure 2.3: Example of causal histories.

Consider the system with three replicas, *Rx*, *Ry* and *Rz*, represented by Figure 2.3. In this system each event is assigned an unique identifier that consists of the replica's name and the value of a local monotonically increasing counter (logical clock). When a new event occurs, its causal history is the union of its unique identifier and the causal history of the previous event in that replica. For example, the causal history of the second event in replica *Rx* is $H_{x_2} = \{x_1, x_2\}$, because it is the union of its unique identifier, x_2, with the causal history of x_1. Furthermore, when a node sends a message to another one, the causal history of the send event is also transmitted along with the message so that the causal history of the remote node can be merged with the local history. This is exemplified in Figure 2.3 when *Rz* receives a message from *Ry* and creates a new event, z_2, with the corresponding merged causal history.

When using causal histories, to verify if an event is causally dependent on another we can use set inclusion:

- If the unique identifier of an event is contained in the causal history of another, then the latter is causally dependent on the former;
- If neither identifier is contained in the causal history of the other event, then the events are not causally related (i.e., they are concurrent).

Note that causal histories are an effective but inefficient way of tracking causality because the size of the metadata grows with the number of operations, becoming unsustainable. Next, we discuss some of the techniques proposed to overcome this problem.

Direct dependencies: One important property that holds in causal histories is transitivity. It can be used to compress causal histories into sets that only include the direct dependencies of an event, because all other dependencies are implicitly present. Consider the causal history of y_2 shown in Figure 2.3: it contains both x_1 and x_2 but, because the causal history of x_2 already contains x_1, it could be omitted without losing the causality information, still allowing us to build the full causal history transitively. This approach is used by systems like COPS [27] and Eiger [28], although the latter uses a superset of the nearest dependencies that includes all dependencies that have a path of length one to the current operation (one-hop dependencies).

Vector clocks: Another compact way of representing causal dependencies are vector clocks. When analyzing a given causal history, we can see that if an event is present then all the preceding events from that node are also present. This implies that it is enough to store the latest event from each node in order to know the full causal history. By taking advantage of this, we can represent causal dependencies in a very compact way by building a vector of Lamport clocks with one entry per node, like so: $H_{y_3} = \{Rx \mapsto 2, Ry \mapsto 3, Rz \mapsto 0\} = [2,\ 3,\ 0]$.

To know if there is a causal dependency between two operations we must check each entry in the respective vector clocks to see if one vector is strictly smaller than the other: $x \prec y$ iff $\forall i : V_x[i] \leq V_y[i] \wedge \exists j : V_x[j] < V_y[j]$. When a new event occurs in a node, we simply increase the corresponding entry in the vector clock instead of generating a new unique identifier for it. Finally, when two nodes communicate we need to merge the two causal histories. We can do this by maintaining the maximum value for each position in the vector clocks: $\forall i : V_z[i] = max(V_x[i], V_y[i])$. For example, if we merged [3,3,0] with [2,4,1] the result would be the vector [3,4,1].

Version vectors: When trying to enforce causal consistency in distributed storage systems, it is usually enough to register the events that effectively alter the replicas. Vector clocks that are only incremented when write operations occur are called version vectors. Systems such as C^3 [16] and ChainReaction [5] use versions vector to track causality relations between operations.

2.2.4 Conflict Resolution Techniques

Systems that provide available forms of consistency, such as eventual or causal+ consistency, need to have conflict resolution mechanisms in place so that they can guarantee that, even when there are concurrent writes for the same key, the replicas of the system will eventually converge to a common state. There are three classical ways of implementing convergent conflict handling:

Last-Writer-Wins (LWW) policy: This approach consists of defining an order among any concurrent write operations and choosing to apply only the effects of the last one. Even though this is one of the most commonly used approaches [5, 27, 28], it is not adequate for all data types because the effects of some client operations are lost.

Consider a set $s = \{a, b, c\}$. When there are two concurrent add operations, $add(d)$ and $add(e)$, only one of the elements will be present in the final set, because the effects of one of the concurrent operations will be discarded. So, the final set will either be $s = \{a, b, c, d\}$ or $s = \{a, b, c, e\}$, and not the intended set, $s = \{a, b, c, d, e\}$.

Application-dependent policy: Some systems opt to expose the divergence to the application, delegating the task of solving the conflict. One such system is Amazon's Dynamo [12], that returns multiple values when executing a read operation that detected concurrent writes. It is the application's responsibility to merge the values and write the new value back to the system.

Merge procedure: The last approach is to give replicas access to a deterministic merge procedure that receives the two divergent states and computes a new merged state. This procedure is normally specific to each application. A particular case of this approach are CRDTs [29], where the logic for merging is encapsulated within data types. CRDTs are discussed in more detail in the following section.

2.3 CRDTs

CRDTs [29] are a family of abstract data types specifically designed to support replication and operation over replicas in large-scale distributed systems. These data types can be concurrently updated with no need for coordination between replicas. CRDTs guarantee that any two replicas that have seen the same set of updates will, eventually and deterministically, converge to a common state. Interaction with CRDTs is made simple by their well defined interfaces that allow to update and query the state of the data structures.

To remain highly responsive and available in the presence of network failure and disconnections, all replicas accept updates at all times and propagate them in and asynchronous manner. Because global coordination is not used, replicas' states may diverge and reads may not reflect some modifications made in other replicas, allowing clients to see inconsistent states. Even though CRDTs intrinsically provide eventual consistency guarantees, when paired with a reliable causal broadcast middleware [8] they are able to provide per-object causal consistency.

Some CRDTs have been formally specified and verified by using the framework proposed in [35], which will not be detailed in this document. Instead, in the following sections we will discuss some key aspects of CRDTs, described in more detail in [29].

2.3.1 Concurrency Semantics

Because CRDTs are specifically designed to allow uncoordinated updates, it is important to define the behavior of the data objects in the presence of concurrent updates.

CRDTs can be used to implement several different data types. For some of them, operations are inherently commutative and executing them in any given order will make the replicas converge to the same common state. However, this is not true for most data types and, in these cases, there are several possible concurrency semantics that can be used. Note that the expected behavior in the presence of concurrency can be different depending on the purposes and requirements of an application. As such, CRDTs strive to give application developers the freedom to choose the most adequate semantics for their application.

We have previously mentioned two concepts that are important to keep in mind when defining concurrency semantics of CRDTs:

1. **Total order among operations:** It is possible to define a total order between all operations so that we can define the **last-writer-wins semantics**, where the state of the replicas will present the effects of the last operation according to that total order;

2. **Happens-before relationship:** When using CRDTs, we can say that $o_1 \prec o_2$ iff the effects of o_1 had been applied in the replica where o_2 was first executed. This partial relationship can be used to define several concurrency semantics.

Going back to the example of the set data type introduced in Section 2.2.4, it is clear that add and remove operations for the same element are not commutative. With this in mind, we can define two concurrency semantics based on the happens-before relationship.

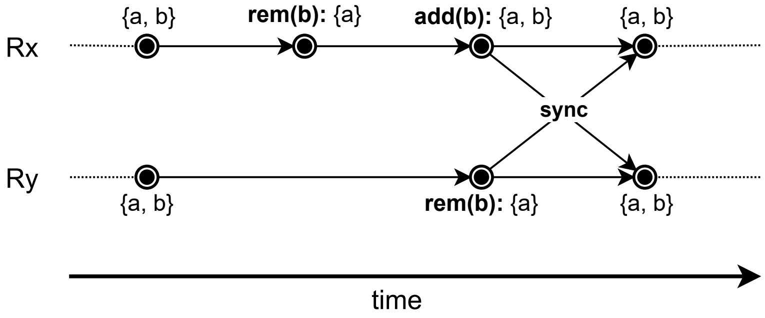

Figure 2.4: Execution with add-wins set.

Add-wins semantics: In the presence of a concurrent add and remove for the the same element, the add always wins over the remove and the element in question will be added to the set. An example can be seen in Figure 2.4, where replica *Rx* adds *b* to the set and, concurrently, replica *Ry* removes it from the set. After the synchronization, *b* belongs to the set (in all replicas).

Remove-wins semantics: In contrast, in the presence of a concurrent add and remove for the the same element, the remove always wins over the add and the element in question will be removed from the set. An example can be seen in Figure 2.5, where replica *Rx* adds *b* to the set and, concurrently, replica *Ry* removes it from the set. After the synchronization, *b* no longer belongs to the set, having been removed in both replicas.

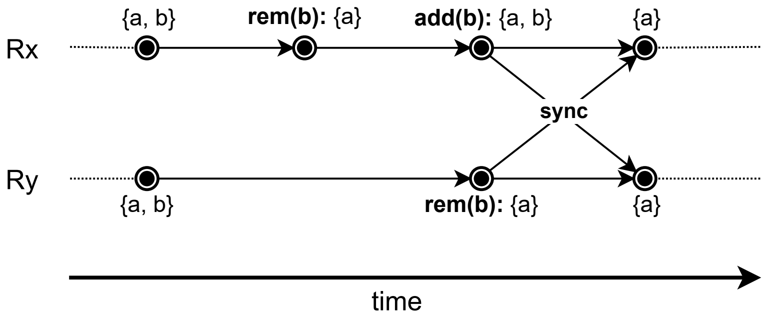

Figure 2.5: Execution with remove-wins set.

2.3.2 Synchronization Methods

When using CRDTs, the system needs to have a synchronization mechanism that allows the replicas to propagate updates to all other replicas so that they can execute them and reach a common state. There are two main types of CRDTs that differ in the synchronization method:

State-based CRDTs: These CRDTs use state-based replication to propagate the effects of update operations. Replicas synchronize by periodically sending a payload with their full state to a neighboring replica. When that replica receives the state, it merges the received state with its local state by using a merge function. If the update propagation graph is connected, then every update will, eventually, reach every replica.

It has been proven, in [33], that all replicas of a CRDT will converge if three conditions are met:

1. The state, s, of the CRDT is partially ordered by $\leq$, forming a join semilattice. A **join semilattice** is a partially ordered set that has a **least upper bound** (LUB), $\sqcup$, for all pairs of elements. $m = x \sqcup y$ is a LUB of $\{x,y\}$ according to $\leq$ iff $\forall m' : x \leq m' \land y \leq m' \Rightarrow x \leq m \land y \leq m \land m \leq m'$. Note that $\sqcup$ is commutative, associative, and idempotent;

2. The state of a replica is monotonically non-decreasing across updates, i.e., an update operation always modifies the state of a replica by inflation, producing a new state that is larger or equal to the previous state according to $\leq$. Given an operation m, $s \leq m(s)$;
3. Merging the local state, s, with a remote state, s', using the merge function produces a final state that is the LUB of the two states: $s \sqcup s'$.

The main downside to state-based CRDTs is that, if the size of the data objects grows to be significantly large, they tend to become inefficient because they use more bandwidth to propagate their full state during synchronization steps.

Operation-based CRDTs: These CRDTs use operation-based replication. When a replica receives an update that mutates its state, it applies the update locally and then propagates it to all other replicas. For each operation, the CRDTs must define two functions:

- **Generator function:** This is a function with no side-effects that executes in the original replica. It looks at the operation and the current state of the replica and generates an effector (a message) that encodes the side-effects of the operation.
- **Effector function:** This function reliably delivers the effector operations to all replicas so that they can be executed, effectively updating their states.

This type of CRDT uses less bandwidth than state-based CRDTs, because the payloads of the effectors are much smaller than the payload of the full state. However, it requires that all effector operations be delivered to every replica in a reliable manner and, usually, according to an order that respects the happens-before relationship (causal order). Because concurrent updates can be delivered in any order, to guarantee convergence, they must commute. Otherwise, if the underlying broadcast primitive does not ensure causal delivery, then all operations must commute.

In addition to the two main types of CRDTs, several alternative models that try to improve some of their downsides have been proposed.

Pure Operation-based CRDTs [6]: The definition of standard operation-based CRDTs is very relaxed. In order to deal with non-commutative operations, these CRDTs include additional causality information in the state, using it in the generate phase and sending it in messages to be used in the effect phase. This makes them able to send the full state between replicas, blurring the distinction between operation-based and state-based CRDTs.

Pure operation-based CRDTs draw a clear line between the two previous types of CRDTs by only sending operations (and their arguments) to other replicas. Data types with non-commutative operations are implemented by making use of the tagged reliable causal broadcast (TRCB) messaging API, that provides causality information when it delivers operations. It does so by assigning a vector clock to each message so that it can be used to make sure that an operation is only delivered at a replica after all its preceding operations also have been.

To represent the state of CRDTs, this implementation makes use of a partially ordered log (PO-log) of operations. Additionally, the authors introduced a PO-log compaction framework that uses causality and causal stability information in order to reduce the storage overhead. This framework removes obsolete operations from the log, keeping only the minimum number of operation needed (e.g., in the set data type, an add operation makes any previous add or remove operations for the same element obsolete). Finally, the framework uses causal stability information to determine when the causal information regarding an operation can be stripped from the log.

In [8], the authors describe a join model for dynamic CRDT environments, and propose a technique to remove metadata of causally stable operations quicker by relying on acknowledgments.

Just like operation-based CRDTs, pure operation-based CRDTs send smaller messages between replicas, using less bandwidth. However, they always depend on the existence of a reliable causal broadcast middleware. One other issue is that the state of the CRDTs is larger because the PO-log stores operations and their causal metadata (even though some operations and metadata are removed under specific conditions).

(Small) δ-CRDTs [4]: These CRDTs try to combine the advantages of both operation-based CRDTs, by sending only small incremental states between replicas, and state-based CRDTs, by disseminating the messages over unreliable communication channels. When using these CRDTs, operations are encoded as δ-mutators that change the state of the CRDT by generating a delta, d, that represents the effects of the most recent update operation on the state. These delta-states are sent to other replicas, instead of shipping the full CRDT state, so that they can be merged with the local replica states. Delta-states can also be combined to form delta-groups before being shipped.

If the causal order of operations is not important, then the delta-groups can be shipped using an unreliable dissemination layer. However, an anti-entropy algorithm that enforces causal delta-merging was defined, so as to provide causal consistency guarantees.

The downside to δ-CRDTs is that they assume continuous and static synchronization patterns between replicas. This is not ideal for scenarios where clients have unreliable communication channels that manifest in highly dynamic communication patters [24].

Furthermore, delta-propagation algorithms tend to disseminate redundant information between replicas. As such, in [14] the authors find the sources of redundancy (i.e., back-propagation of δ-groups and redundant state in received δ-groups) and introduce a way of computing optimal deltas, as well as an improved synchronization algorithm that reduces the amount of state transmission, memory consumption and processing time for synchronization.

(Big) Δ-CRDTs [24]: These CRDTs were specifically designed to overcome the problem of having highly dynamic communication patterns between replicas. They are an extension of δ-CRDTs that remove the assumption that pairs of replicas continuously communicate, and reduce the storage overhead by not maintaining pairwise communication buffers. To do this, they use internal metadata to compute the minimal delta that needs to be propagated to the other replicas.

In order to compute the delta from a given causal context, Δ-CRDTs need to store metadata regarding deleted elements in the CRDT state (tombstones). However, these CRDTs provide a mechanism to periodically garbage-collect these tombstones so that the space overhead is kept low.

The main downside to using Δ-CRDTs is the increase in latency for replicas to receive operations, incurred from the additional communication step needed for the receiving replica to send their version vector before being sent the corresponding minimal delta.

2.3.3 Examples of CRDTs

In this section, we present a number of CRDT designs for both simple data types, such as counters and registers, and more advanced data types, such as sets and maps. The designs presented below were based on those proposed in [32].

2.3.3.1 Counter

A counter is a replicated integer that supports *increment* and *decrement* operations to update it, and a *value* operation to query it. The integer returned by the *value* operation should be the total number of increments minus the total number of decrements.

An operation-based counter is presented in Algorithm 1. The payload is an integer with initial value of zero. Assuming that there are no overflows or underflows, *increment* and *decrement* operations are intrinsically commutative. The *downstream* phases (where the operations are propagated to all replicas) consist in simply incrementing or decrementing the counter by 1.

Algorithm 1: Operation-based Counter CRDT

```
payload integer $c$
    initial 0
query $value()$: integer $r$
    let $r = c$
update $increment()$
    atSource()                              // No initial processing done at source
    downstream()
        $c := c + 1$
update $decrement()$
    atSource()                              // No initial processing done at source
    downstream()
        $c := c - 1$
```

Note that it is very simple to extend this counter with support for operations that increment or decrement the counter by a specific amount, by simply passing it as an argument in, for example, *incrementValue(integer v)* and *decrementValue(integer v)* operations.

Algorithm 2: State-based PN-Counter CRDT

```
payload integer[$n$] $P$, integer[$n$] $N$             // $n$: the number of replicas
    initial [0,0,...,0], [0,0,...,0]
update $increment()$
    let $g = myID()$                                   // $g$: index of the source replica
    $P[g] := P[g] + 1$
update $decrement()$
    let $g = myID()$
    $N[g] := N[g] + 1$
query $value()$: integer $r$
    let $r = \sum_{i=0}^{n-1} P[i]$ - $\sum_{i=0}^{n-1} N[i]$
compare(X,Y): boolean $b$
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \wedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i])$
merge(X,Y): payload $Z$
    let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$
    let $\forall i \in [0, n-1] : Z.N[i] = max(X.N[i], Y.N[i])$
```

Because the operations of a counter are commutative, it is inherently an operation-based CRDT. However, a state-based CRDT version of a counter is not so trivial to specify. In Algorithm 2, we present a specification of a PN-Counter (Positive-Negative Counter) CRDT that supports the *decrement* operation by combining two Grow-only Counters (G-Counters), i.e., counters that only accept increments.

The payload of the counter consists of two vectors with one entry per replica of the CRDT. Vector P registers the increments, and vector N registers the decrements. The value of the counter is the difference between the sums of all elements in P and N. The *merge* operation chooses, for each position of P and N, the maximum value of the two counters it is merging, *X* and *Y*.

Note that, like the operation-based CRDT counter, the PN-Counter also assumes there are no overflows or underflows, and that the set of replicas is well-known (and static).

2.3.3.2 Register

A register is used to store an object of any type. It supports an *assign* operation to update its value, and *value* operation to query it (i.e., obtain the value currently being stored). Two concurrent *assign* operations are non-commutative. In Algorithm 3, we present the LWW-Register CRDT, that uses the last-writer-wins policy to decide which operation has precedence over the other. There is also another specification, the Multi-Value Register (MV-Register) CRDT, that keeps both values in the presence of concurrent *assign* operations.

To apply the LWW policy, this CRDT creates a total order of *assign* operations. This is done by associating a unique timestamp that is consistent with causal order to each update. The *assign* operation generates the new timestamp at the source replica and then the operation is sent to all replicas. In the *downstream* phase, the update only modifies the state if the new timestamp is greater than the local timestamp of the replica.

Algorithm 3: Operation-based LWW-Register CRDT

```
payload X $x$, timestamp $t$                          // X can be any type
    initial $\perp$, 0
query value(): X $r$
    let $r = x$
update assign(X $x'$)
    atSource()
        let $t' = now()$                               // Timestamp
    downstream($x'$, $t'$)
        if $t < t'$ then $x, t := x', t'$
```

2.3.3.3 Set

A set is a collection of distinct elements, i.e., a collection with no duplicates. This collection supports an *add* operation, to add an element to the set, and a *remove* operation, to remove an element from the set. Additionally, it usually has support for a *lookup* operation to check if an element belongs to the set, and an *elements* operation that returns all the elements of the set.

Because *add* and *remove* operations are non-commutative, the main struggle of specifying a set CRDT is to define how to deal with concurrent adds and removes. There are several ways to do this but, in this document, we will only present the specification of

Algorithm 4: Operation-based OR-Set CRDT

```
payload set $S$                                    // S: set of pairs (element, unique-tag)
    initial $\varnothing$
query $lookup$(element $e$): boolean $b$
    let $b = (\exists u : (e,u) \in S)$
update $add$(element $e$)
    atSource($e$)
        let $\alpha = unique()$                                    // $\alpha$: unique-tag
    downstream($e$, $\alpha$)
        $S := S \cup \{(e,\alpha)\}$
update $remove$(element $e$)
    atSource($e$)
        pre $lookup(e)$
        let $R = \{(e,u) | \exists u : (e,u) \in S\}$
    downstream($R$)
        pre $\forall (e,u) \in R : add\,(e,u)$ has been delivered        // Causal order is enough
        $S := S \setminus R$                                // Remove pairs observed at source
```

the OR-Set (Observed-Remove Set) CRDT, that gives precedence to *add* operations over *remove* operations, effectively providing add-wins semantics.

The OR-Set specification can be seen in Algorithm 4. The payload is a set S that includes pairs (element, unique-tag). When an *add* operation occurs, an unique tag is generated in the *atSource* phase (where some pre-processing is done in the source replica). Then, the element and its tag are propagated to the other replicas in the *downstream* phase, so that the replicas can add the element and its unique tag to their respective sets. The *lookup* operation implicitly hides duplicates, so this CRDT supports concurrent adds for the same element. When a *remove* operation occurs, in the *atSource* phase, all pairs to be removed are collected in set R. This set is then propagated to all replicas in the *downstream* phase, so that they can remove the pairs from their local set S.

For this specification to be correct, it is assumed that the underlying dissemination protocol delivers pairs in an order that is consistent with the happens-before relationship, so that a *remove(e)* operation is able to remove all pairs (e, u) added before it.

2.3.3.4 Map

A map is a collection of pairs (key, value), where each key is unique and maps to a value. These values can be of any type. Maps normally have *add*, *remove*, *get*, *contains*, *keys* and *elements* operations. The *add* operation is used to map a value to a key. The *remove* operation removes a key a its corresponding value from the map. The *get* operation returns the value that is mapped to a given key. The *contains* operation returns true if a specific key has a mapping. Finally, the *keys* and *elements* operations return, respectively, the set of all keys or all values in the map.

Map CRDTs can be built upon any set CRDT specification, by making the payload a mapping of keys to sets of values. Map specifications make the *add* operation remove all previous values associated with the key in question. However, if there are two concurrent adds, both values can be kept.

The specification presented in Algorithm 5 was adapted from [30]. It is an OR-Map based on a OR-Set CRDT and, as such, when there are concurrent *add* and *remove* operations, precedence is given to the *add*.

Algorithm 5: Operation-based OR-Map CRDT

```
payload map $M$                 // M: map of keys ↦ set of pairs (element, unique-tag)
    initial $\varnothing$
query contains(key $k$): boolean $b$
    let $b = (k \in M)$                    // If M[k] is an empty set, then k is not in M
query get(key $k$): set $E$
    pre contains($k$)
    let $E = \{e | \exists\, (e,u) : (e,u) \in M[k]\}$
query keys(): set $K$
    let $K = \{k | \exists k : k \in M\}$
query elements(): set $E$
    let $E = \{e | \exists\, (k,e,u) : (e,u) \in M[k]\}$
update add(key $k$, element $e$)
    atSource($k$, $e$)
        let $u = unique()$                                   // $u$: unique-tag
        let $R = A[k]$                                // Pairs to remove from key
    downstream($k$, $e$, $u$, $R$)
        pre $\forall\, (e,u) \in R : add(k,e,u)$ has been delivered      // Causal order is enough
        $A[k] := A[k] \cup \{(e,u)\} \setminus R$
update remove(key $k$, element $e$)
    atSource($e$)
        pre contains($k$)
        let $R = A[k]$
    downstream($k$, $R$)
        pre $\forall\, (e,u) \in R : add\,(k,e,u)$ has been delivered    // Causal order is enough
        $A[k] := A[k] \setminus R$
```

When an *add* operation occurs, a unique tag and the set of values, *R*, associated to the key are computed, in the *atSource* phase. Then, the pair (key, value), the tag and *R* are propagated to every replica in the *downstream* phase, in which the replicas map the new pair to the given key and remove all previous mappings. When a *remove* operation occurs, the set of mapping for the key, *R*, is computed in the *atSource* phase. That set is then passed to every replica, so that they can remove the pairs from the map in the *downstream* phase. The *contains* operation checks if there exists a mapping for the key *k* in the map. Note that if the mapping is an empty set, the *contains* operation should return false.

The *get* operation returns the set of elements that are mapped to the key *k*, if there is mapping different from an empty set. The *keys* operation returns all the keys that have mappings, and the *elements* operation returns all the elements that exist in the map.

2.4 Existing Systems

In this section, we present descriptions of some existing causally consistent data storage systems.

The following systems are discussed in a less detailed manner because they are not as close to our proposed approach, mostly because they do not use CRDTs to guarantee that all replicas converge to the same state despite concurrent updates. All these systems support geo-replication because they are built to serve as base for large-scale services with users all over the world.

COPS [27] (Clusters of Order-Preserving Servers) is the distributed key-value store that first defined the concept of causal+ consistency. It keeps a full copy of the data at each data center (i.e., COPS cluster). However, within the data centers, the key space is partitioned and the keys are stored in different nodes by using consistent hashing. Chain replication is used to replicate each key across a small number of replicas, thus providing linearizability within the cluster. On the other hand, replication between COPS clusters happens asynchronously and provides causal+ consistency. This system uses client-side metadata to track the nearest dependencies on versions of keys and explicitly checks that they have been satisfied locally before making the effects of an operation visible. There is another variant of this system, COPS-GT, that maintains multiple older versions of the values associated with the keys so that it can offer multiple key read-only transactions.

Eiger [28] is a low latency geo-replicated data store for the column-oriented data model. Similarly to COPS, Eiger keeps a full copy of the data in each data center, partitions the data across several machines and uses explicit dependency checking in order to guarantee that an operation can be applied. However, contrary to COPS, Eiger keeps track of one-hop dependencies on operations, not of nearest dependencies on versions of keys. This system supports improved read-only transactions when compared to those of COPS, and write-only transactions that work for keys spread across multiple servers of a data center.

GentleRain [13] is a geo-replicated key-value store. The system is split into several partitions that are replicated across multiple replicas, with consistent hashing of the keys being used to assign data items to a specific partition. Additionally, this data store maintains older versions of the items that are periodically garbage-collected. This system tracks causality with a less granular approach by summarizing dependencies in a single scalar (timestamp), thus reducing the storage and communication overhead. Unlike COPS and Eiger, GentleRain uses a periodic aggregation protocol to determine if an update can be made visible, resulting in improved throughput but also higher update visibility latency.

ChainReaction [5] is a geo-distributed key-value store that uses a new variant of chain replication that offers causal+ consistency. Inside each data center, the replicas are organized in a distributed hash table (DHT) and the data items are replicated only in a chain of nodes of a specified size by using consistent hashing. This allows to distribute the load of read operations across multiple replicas within the same data center. This system deals with causal metadata in an efficient manner by implementing a stabilization procedure that preserves causal guarantees while introducing low metadata overhead, for both the client and the data store.

Saturn [9] is a metadata management service that can be used in combination with existing geo-replicated storage systems. The underlying system can use a full replication scheme, but genuine partial replication is also supported, guaranteeing that each data center only receives and stores information regarding data items that it replicates, and ensuring the scalability of the system. Saturn can be used to provide causal consistency to storage systems by delivering metadata to the data centers in causal order, as long as they apply an update only after receiving the corresponding metadata. To ensure causal delivery, Saturn internally organizes the data centers is a tree connected by FIFO channels, where the data centers are the leaves. By decoupling the metadata management from data dissemination and using clever metadata propagation mechanisms, it ensures that the visibility latency of updates approximates that of weak-consistent systems that do not store metadata.

C^3 [16] is a solution that can be used to extend existing distributed storage systems. It introduces a novel replication scheme that supports partial geo-replication in order to reduce the storage and communication overhead of full replication schemes. The system also works if the underlying storage system partitions data items across multiple nodes inside each data center, even though it is not required that it does. C^3 uses an approach that separates the data store layer from the causality tracking layer. It works in a P2P fashion, where each data center has a causality layer instance associated, and the instances communicate with each other by exchanging labels (i.e., small pieces of data that contain a vector clock and an unique identifier). When executing an operation, its label is propagated to the causality layer of the interested data centers to be added to their log of pending operations until all the operations it depends upon have been completed. This solution allows concurrency in the execution of remote operations, thus lowering their visibility times and enabling the data store to scale through sharding.

The following systems are discussed in a more detailed manner because they are the ones that most closely approximate our proposed approach, namely by using CRDTs to ensure replica convergence.

Cure [2] is a replication protocol for highly available geo-replicated key-value stores that supports causal+ consistency. It uses operation-based CRDTs to guarantee convergence, and provides an interface that offers interactive transactions that combine read and write operations. Cure assumes that, in each data center, the system stores the full set of objects but creates non-overlapping partitions of the key-space that are the same across data centers.

Cure uses multi-versioning to allow clients to read from causally consistent snapshots of the database. To do this, each version is stored with its causal dependencies in the form of a vector clock. Old versions are garbage collected by making partitions periodically exchange the oldest snapshot vector clock of its active transactions. The minimum of these vectors clocks is computed so that partitions can remove all objects with versions prior to it.

To replicate updates, each partition periodically synchronizes with the partitions that store the same key set in other data centers. If there are no new updates, the partitions send heartbeats. To ensure causal consistency guarantees, an update received from a remote replica is only made visible after the updates it depends on have been received and applied at the local data center. As such, this system relies on a background stabilization protocol that builds a tree over all servers in a data center and makes them exchange their vector clocks in order to compute their the global stable snapshots.

By using vector clocks instead of a single scalar to track causality, Cure's update visibility latency depends on the latency to the data center where the update originated, and not on the latency to the furthest data center, as in GentleRain. However, this improvement in the visibility latency comes at the cost of higher computation and storage overhead related with metadata management, which negatively affects the throughput of Cure.

SwiftCloud [34] is a distributed object database that gives client-side applications local access to a causally consistent cache that contains a partial replica of the database. This system provides immediate responses for reads and writes on local objects, while using the cloud as a back-up for cache misses, thus offering throughput that is equal or better to that of systems that use server-side geo-replication. Because causal consistency does not guarantee the convergence of replicas, this system relies on CRDTs to do so. SwiftCloud is scalable to thousands of clients, providing consistent versioning while using small metadata. It is also tolerant to data center failures, connecting clients to a different data center while maintaining causal consistency guarantees, at the cost of a slight increase in staleness.

The cloud infrastructure of the system connects a small set of geo-replicated data centers to a large set of clients. Each data center maintains a full copy of the database, while clients only keep the objects that interest them (partial replicas). A client usually connects to a single data center but, in case of failure, it can connect to a new one. Because client replicas are partial, the system can only guarantee conditional availability, by which an operation only returns without remote communication if the requested object is in the local cache. If the object is not in the cache, it must be retrieved from a data center. If, for some reason, the data centers are not able to satisfy the request, the operation may block or return an error. Each data center also maintains a best-effort notification session to each of its clients, over FIFO channels, so that it can issue notifications to them regarding updates to objects in their interest sets.

SwiftCloud uses the approach of "reading in the past" to ensure that data centers only expose causally consistent views of the data, even though some of that data may already have newer versions. In order to do this, the system keeps metadata in the form of a version vector. When a data center receives an update operation, it must check if all its dependencies are satisfied before applying it. Otherwise, it must buffer the operation until all dependencies have been satisfied, so as not to cause clients to see inconsistent views of the data.

As is, this solution is not tolerant to data center failures. Because replication between data centers is done asynchronously, when a client switches to a new data center the updates that were delivered in that data center may be stale when compared to the ones of the previous data center. This could cause a causal gap in the client and, as such, the new data center must reject the client. SwiftCloud provides a solution to this problem by maintaining, in each data center, a K-stable version that contains the updates for which the data center has received acknowledgments for at least $K-1$ distinct data centers. The base version (stored in the cache) of all clients must be K-stable, so that clients depend only on external updates that are likely to be found in any data center, or internal ones that the client can transfer to the new data center.

Even though SwiftCloud provides causal consistency under a partial replication scheme, remains fault tolerant, and uses CRDTs to ensure replica convergence, it does so for client-side applications. As such, it does not entirely approximate our proposed approach because it brings the storage and computation to the client devices.

Legion [25] is a framework that allows web applications to use client-side replication of data. Each client maintains a local data store with a partial replica of the shared application objects that can be modified without coordination with any other replicas. Legion offers causal consistency guarantees by propagating updates asynchronously to other replicas.

Contrary to SwiftCloud, that caches data at the client and synchronizes with the servers, Legion allows for both synchronization with servers and among the clients themselves, using P2P interactions. As such, overlay networks are created between clients that share objects so that they can propagate updates to each other. Some of the clients also act as a bridge to the servers (and to clients that are outside the Legion logical networks), allowing the remote updates to eventually reach all clients. This reduces the update visibility latency of clients that are close to each other.

To ensure that all replicas' states converge, Legion uses Δ-CRDTs [24] to resolve conflicts. These CRDTs are efficient when used with decentralized dissemination protocols, such as epidemic protocols, allowing replicas to synchronize by using deltas with the effects of one or more operations (or the full state, if need be). The objects of the data store are encoded as CRDTs and grouped within containers of related objects.

To propagate deltas in causal order, Legion uses a multicast primitive implemented using a push-gossip protocol [22]. Each client maintains a causally ordered list of received deltas for each container, and propagates them (using FIFO channels) to every client it connects to. Upon receiving a delta, a client either discards it (if when checking the version vector of the container it detects that it already received it), or it integrates the delta with its state and adds it to the list of deltas to be propagated to other clients.

Note that, to reduce storage overhead, Legion only stores a suffix of the list of deltas received. Because clients exchange their vector clocks at the start of every synchronization step, this allows them to generate deltas that contain only operations that the other peer has not yet seen. However, when connecting for the first time, it might be impossible or inefficient to compute the appropriate delta between two clients. As such, sometimes clients may have to do a synchronization step where they exchange their full state.

Legion improves the latency for update propagation when compared with server-based systems. Furthermore, it reduces the load to the centralized component by using P2P interactions between clients, allowing disconnected operation. However, because client-side devices are often resource poor, this system imposes a large overhead on them by making clients communicate with each other, store data objects, and perform advanced computation. On the other hand, our approach aims to utilize edge nodes for storage and computation, thus reducing client perceived latency when compared to server-based approaches, while still allowing us to take advantage of data locality.

Summary

In this Chapter we presented and discussed previous related work that is relevant for the context and objectives of this thesis.

In the following Chapter we will detail our proposed approach and the scheduling of the work to achieve the final solution.

Chapter 3

Proposed Work

In this Chapter we present details regarding the work proposed in this document.

In Section 3.1 we explain the intuition behind the proposed solution by discussing the challenges we want to overcome and how we expect to do so.

In Section 3.2 we present the evaluation plan and the metrics we intend to measure in order to adequately evaluate the performance of our solution.

In Section 3.3 we present the scheduling of the work to be developed in the future.

3.1 Proposed Solution

As mentioned in Chapter 1, in this work we plan to design a CRDT replication kernel that provides more sophisticated concurrency semantics by leveraging on CRDTs, while retaining efficiency and scalability by exploring genuine partial replication at the edge and using minimal metadata to track causality. In this section, we will further detail some of the challenges of designing this system and the ways in which we plan to overcome them.

Tracking the location of data objects: In order to take full advantage of genuine partial replication, we must ensure that the replication kernel has a way to know where each data object is replicated (i.e., know which replicas store each CRDT object). This is necessary in order to guarantee that the updates regarding those data objects reach all the interested replicas. Because tracking the location of each individual data object could induce great overhead for the system, we plan to create groups of data objects and track their locations at that granularity level. Furthermore, replicas should cooperate in a decentralized manner to keep this information up to date, and to define adequate communication patterns among themselves to support synchronization steps.

Organization of the replicas: To distribute the cost of synchronization and communication, we plan to organize replicas in some sort of hierarchical structure, namely a tree topology. This layout should be static enough so that each replica mainly communicates with the same neighboring replicas, thus reducing the number of new connections that need to be opened, while still allowing for replicas to leave or join the system dynamically without hindering performance. This requires using mechanisms to ensure that trees are fault-tolerant by considering, for instance, ideas from the Plumtree protocol [21]. By organizing the replicas in a tree, we ensure that the synchronization load is distributed in a uniform manner across most replicas, while the leaf nodes are only connected to one neighbor. This could be used to our advantage if we make the less resourceful nodes of the system communicate with fewer neighbors by making them the leaf nodes of the tree.

Implementation of CRDTs: Because the underlying replication protocol will already ensure causal delivery, we plan to explore a new design of operation-based CRDTs. However, as we are working in a P2P environment, we will base our approach on **Δ**-CRDTs [24] (designed for environments with dynamic communication patters), and we will try to reduce the amount of redundant information that is propagated between replicas by transposing the work done in [14] to the operation-based CRDT side. Additionally, we want to explore the possibility of discarding the propagation of redundant operations altogether for some data types. Consider, for example, a register CRDT, x. An assignment of 3, $x := 3$, followed by an assignment of 5, $x := 5$, renders the operation of assigning 3 to the register irrelevant (i.e., in the long run, there is no need to propagate this update to other replicas of the system).

CRDT Uniform Replication API: Since the process of replicating CRDTs is delegated to an external replication kernel, all CRDTs must expose, through an uniform replication API (i.e., common to all types of CRDTs), all necessary information that has to be shipped to remote replicas. This might include standard representations of the internal state, operations, and associated metadata. Additionally, the uniform replication API must provide mechanisms to incorporate operations (and their associated metadata) received from remote replicas. This cannot be done by using the standard CRDT API used to execute client operations, because doing so could lead to undesirable manipulation of the local CRDT metadata, which could, in turn, lead to the creation of fictitious operations. Finally, all data exposed by the CRDTs should have a language independent format, so as to allow different services to communicate without any need for pre-processing.

By the end of the work proposed in Section 3.3, we plan to have implemented a working replication kernel prototype and a small set of CRDTs (counter, register, set and map), as well as a simple storage system prototype that will serve as the case study for our comparative evaluation. We aim to make the replication kernel extensible, so that specifications of CRDTs besides the implemented ones can be used.

3.2 Experimental Evaluation

This section describes the evaluation plan for the developed work. We propose two different kinds of testing and evaluation phases: one for the intermediate and final CRDT replication kernel prototypes, and another for our distributed storage system prototype.

1. **Evaluation through micro-benchmarks:** To evaluate the performance of the CRDT replication kernel in an isolated environment that approximates real world scenarios, we plan to use a cluster of (containarized) nodes with injected latency and manipulated bandwidth, and measure data freshness, synchronization and communication costs (and their distribution across the replicas of the system). Additionally, we want to study how these metrics are affected when the topology of the system suffers reconfiguration due to replica failures;

2. **Comparative evaluation against existing storage systems:** We plan to perform a comparative analysis of the latency and throughput of our storage system prototype (that uses our CRDT replication kernel), against existing systems that also utilize CRDTs for conflict resolution, such as AntidoteDB [3] (that employs Cure [2] as a replication protocol). To do this, we will use a custom workload generator, such as Yahoo Cloud Serving Benchmark (YCSB) [11].

3.3 Work Plan

In this section we present the scheduling for the remainder of the work to be done in the context of this dissertation. The work plan is divided in five main phases: the first three phases correspond to the design and implementation of both the CRDT replication kernel, and the distributed storage system prototype; the last two phases include the final evaluation of the system, and the writing of a paper and the dissertation.

Table 3.1 presents the overall timespan for each of the five high level tasks. In Figure 3.1, we present a Gantt chart of the schedule that shows a break down of these tasks into sub-tasks.

Replication Kernel Prototype: The first two tasks of the schedule are iterations of the design, implementation and evaluation of the CRDT replication kernel. The purpose of having two iterations is to be able to validate and improve the prototype according to the performance findings of the preliminary evaluation task, so that we can achieve the best possible results.

Storage System Prototype: This task consists on the design, implementation and evaluation of a simple storage system prototype that uses the CRDT replication kernel.

After the evaluation, some optimizations might be performed before moving on to the next task.

Final Evaluation: After finishing the implementation of the kernel and the data store we will perform an extensive and thorough evaluation, that was further detailed in Section 3.2.

Writing: The last task consists of, simultaneously, writing the dissertation and a paper to be submitted to an international conference.

Table 3.1: Work schedule.

Task	Start Date	End Date	Weeks
Replication Kernel: Preliminary Solution Design Implementation Evaluation	1st March	18th April	7
Replication Kernel: Final Solution Design Implementation Evaluation	19th April	20th June	9
Storage System Prototype Design Implementation Evaluation	21st June	1st August	6
Final Evaluation	2nd August	29th August	4
Writing Dissertation Paper	30th August	30th September	4.5

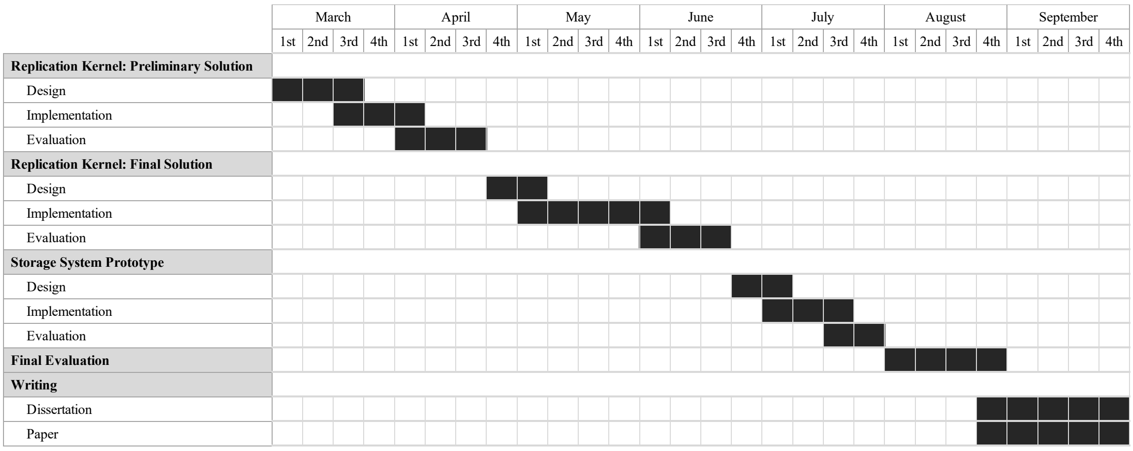

Figure 3.1: Gantt chart of proposed work schedule.

Bibliography

[1] *7 Data Center Disasters You'll Never See Coming*. https://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702. Accessed: Feb. 2021.

[2] 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)*. 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98.

[3] D. D. Akkoorath and A. Bieniusa. "Antidote: the highly-available geo-replicated database with strongest guarantees". In: *SyncFree Technology White Paper* (2016).

[4] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-based CRDTs by Delta-Mutation". In: *CoRR* abs/1410.2803 (2014). arXiv: 1410.2803. URL: http://arxiv.org/abs/1410.2803.

[5] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction". In: (2013), p. 85. DOI: 10.1145/2465351.2465361.

[6] C. Baquero, P. Almeida, and A. Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Proceedings of the 1st Workshop on the Principles and Practice of Eventual Consistency, PaPEC 2014* (Apr. 2014). DOI: 10.1145/2596631.2596632.

[7] C. Baquero and N. Preguiça. "Why Logical Clocks Are Easy". In: *Communications of the ACM* 59.4 (Apr. 2016), pp. 43–47. ISSN: 0001-0782. DOI: 10.1145/2890782.

[8] J. Bauwens and E. Gonzalez Boix. "Memory Efficient CRDTs in Dynamic Environments". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 48–57. ISBN: 9781450369879. DOI: 10.1145/3358504.3361231. URL: https://doi.org/10.1145/3358504.3361231.

[9] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency". In: *Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017* (2017), pp. 111–126. DOI: 10.1145/3064176.3064210.

[10] E. Brewer. "CAP twelve years later: How the "rules"have changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[11] B. F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[12] G. DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281.

[13] J. Du et al. “GentleRain: Cheap and scalable causal consistency with physical clocks”. In: *Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014* (2014). DOI: 10.1145/2670979.2670983.

[14] V. Enes et al. “Efficient synchronization of state-based CRDTs”. English. In: *Proceedings - 2019 IEEE 35th International Conference on Data Engineering, ICDE 2019*. Proceedings - International Conference on Data Engineering. IEEE Computer Society, Apr. 2019, pp. 148–159. DOI: 10.1109/ICDE.2019.00022.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[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.

[17] S. Gilbert and N. Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

[18] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. “Trade-offs in replicated systems”. In: *IEEE Data Engineering Bulletin* 39.ARTICLE (2016), pp. 14–26.

[19] 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.

[20] L. Lamport. “The Part-Time Parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. ISSN: 07342071. DOI: 10.1145/279227.279229.

[21] J. Leitao, J. Pereira, and L. Rodrigues. “Epidemic Broadcast Trees”. In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27.

[22] J. Leitão, J. Pereira, and L. Rodrigues. “HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast”. In: July 2007, pp. 419–429. ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.56.

[23] J. Leitão et al. “Towards Enabling Novel Edge-Enabled Applications”. In: *CoRR* abs/1805.06989 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989 .

[24] A. Linde, J. Leitão, and N. Preguiça. “Δ-CRDTs: making δ-CRDTs delta-based”. In: Apr. 2016, pp. 1–4. DOI: 10.1145/2911151.2911163.

[25] A. van der Linde et al. “Legion: Enriching Internet Services with Peer-to-Peer Interactions”. In: *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292. ISBN: 9781450349130. DOI: 10.1145/3038912.3052673. URL: https://doi.org/10.1145/3038912.3052673.

[26] A. van der Linde et al. “The intrinsic cost of causal consistency”. In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020* (2020). DOI: 10.1145/3380787.3393674.

[27] 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.

[28] W. Lloyd et al. “Stronger Semantics for Low-Latency Geo-Replicated Storage”. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 313–328. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd.

[29] N. M. Preguiça, C. Baquero, and M. Shapiro. “Conflict-free Replicated Data Types (CRDTs)”. In: *CoRR* abs/1805.06358 (2018). arXiv: 1805.06358. URL: http://arxiv.org/abs/1805.06358.

[30] A. Rijo. “Building Tunable CRDTs”. MA thesis. https://run.unl.pt/handle/10362/55171: FCT NOVA, Nov. 2018.

[31] M. Satyanarayanan. “The Emergence of Edge Computing”. In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[32] M. Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[33] M. Shapiro et al. *Conflict-free Replicated Data Types*. Research Report RR-7687. INRIA, July 2011, p. 18. URL: https://hal.inria.fr/inria-00609399.

[34] M. Zawirski et al. “Write fast, read in the past: Causal consistency for client-side applications”. In: *Middleware 2015 - Proceedings of the 16th Annual Middleware Conference* Dc (2015), pp. 75–87. DOI: 10.1145/2814576.2814733.

[35] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. “Formal Specification and Verification of CRDTs”. In: June 2014, pp. 33–48. ISBN: 978-3-662-43612-7. DOI: 10.1007/978-3-662-43613-4_3.