FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Design and Implementation of Pure Operation-based CRDTs

Filipe Recharte

Mestrado em Engenharia Informática e Computação

Supervisor: Hugo Pacheco

Second Supervisor: Carlos Baquero

July 28, 2023

Design and Implementation of Pure Operation-based CRDTs

Filipe Recharte

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Pedro Souto
External Examiner: Prof. Carla Ferreira
Supervisor: Prof. Hugo Pacheco

July 28, 2023

Resumo

À medida que a tecnologia evolui, as aplicações necessitam de mais disponibilidade e capacidade de resposta para fornecer um sistema robusto a milhões de utilizadores. Isto é obtido através do desenvolvimento de aplicações por cima de sistemas geo-replicados que compartilham os dados da aplicação entre várias réplicas.

No entanto, a construção de sistemas distribuídos induz problemas com a usabilidade das aplicações uma vez que haverá sempre um trade-off entre consistência e disponibilidade, juntamente com falhas na rede que tornam as réplicas incapazes de entrar em contacto umas com as outras.

Em sistemas distribuídos, a consistência forte é frequentemente usada para mitigar a inconsistência de dados entre réplicas e garantir uma ordem total de operações para evitar conflitos em atualizações simultâneas. No entanto, como afeta a disponibilidade, os modelos de consistência fraca geralmente tornam-se a solução para aplicações que valorizam a capacidade de resposta em detrimento do nível de consistência. Conflict-free Replicated Data Types (CRDTs) são adequados para resolver este problema, pois as operações são executadas diretamente em cada réplica e propagadas por trás, aumentando a disponibilidade.

Em sistemas não distribuídos, as transações são a abordagem padrão para obter garantias ACID, oferecendo consistência em pedidos concorrentes. Em sistemas distribuídos, a consistência entre réplicas pode ser alcançada de várias maneiras, sendo, por exemplo, baseada em modelos como os CRDTs. No entanto, combinar múltiplos CRDTs aumenta a complexidade de preservar invariantes.

Algumas abordagens foram desenvolvidas para mitigar esse problema, como reparar invariantes, misturar consistências ou construir CRDTs consistentes por design. Neste trabalho, exploramos a última abordagem usando a framework de CRDTs pure op-based, que torna o design dos CRDTs "quase" genérico, mas deixa alguma lógica dependente do tipo de dados para ser feita manualmente. Implementamos uma framework pure op-based e exploramos como duas abordagens recentes para o design de CRDTs podem ser usadas para realizar CRDTs pure op-based "mais próximos" do genérico, onde os utilizadores apenas raciocinam sequencialmente, desenvolvendo um tipo de dados, as suas operações e propriedades semânticas, e um CRDT consistente com a execução sequencial é automaticamente construído. Este trabalho é também um exercício para entender e explicar o design distribuído de CRDTs complexos a partir da semântica de tipos de dados sequenciais.

Abstract

As technology evolves, applications need more availability and responsiveness to provide millions of users with a robust system. This is achieved by developing applications on top of geo-replicated systems that share the application's data among multiple replicas.

However, building distributed systems induces problems with the usability of applications since there will always exist a trade-off between consistency and availability, along with network faults that make replicas unable to contact each other.

In distributed systems, strong consistency is often used to mitigate data inconsistency within replicas and ensure a total order of operations to avoid conflicts on concurrent updates. However, as it affects availability, weak consistency models often become the solution for applications that value responsiveness over consistency. Conflict-free replicated data types (CRDTs) are suitable to address this problem as operations are executed directly on each replica and propagated in the background, boosting availability.

In non-distributed systems, transactions are the standard approach to obtain ACID guarantees, offering consistency on concurrent requests. In distributed systems, consistency within replicas can be achieved in many ways, such as being based on models like CRDTs. However, it is challenging to build applications on top of CRDTs.

Some works have been developed to mitigate this problem, such as repairing invariants, mixing consistencies, or building CRDTs consistent by design. In this work, we explore the latter approach using the pure op-based framework, which makes the design of CRDTs "almost" generic but leaves some datatype-dependent reasoning to be manually crafted. We implement a pure op-based framework and explore how two recent approaches for the design of CRDTs can be put to use to accomplish "closer to" generic pure op-based CRDTs, where users only reason sequentially, developing a datatype, its operations, and semantic properties, and a CRDT consistent with sequential execution is automatically constructed. This work is also an exercise in understanding and explaining the distributed design of complex CRDTs from the semantics of sequential datatypes.

Acknowledgments

This dissertation is the result of a five-year-long marathon. It's been a wild ride filled with many challenges but also rewarding experiences. I've been incredibly lucky to have the support of my close friends and family, who've stood by me under all circumstances. I couldn't have done it without them.

I am profoundly grateful to my supervisor, Hugo Pacheco. His guidance, expertise, and readiness to help were invaluable throughout this journey. I'm also thankful for my co-supervisor, Carlos Baquero, whose insights significantly enriched our research. This dissertation is the result of our collective effort, and I couldn't be happier with what we've achieved together.

Filipe Recharte

Contents

List of Figures

Abreviaturas e Símbolos

SEC	Strong Eventual Consistency
ADT	Abstract Data Type
CRDT	Conflict-free Replicated Data Type
ACID	Atomicity, Consistency, Isolation, Durability
ECRO	Explicitly Consistent Replicated Objects
PO-Log	Partially Ordered Log
TCSB	Tagged Causal Stable Broadcast
OT	Operational Transformation
RGA	Replicated-Growable Array
MVR	Multi-value Register

Chapter 1

Introduction

1.1 Context

Throughout the years, technology has evolved to satisfy users' daily needs, forcing applications to handle an increasing amount of data and requests. Regarding this substantially increasing demand, some applications are expected to be highly available without response delays, to provide users with the best experience possible. This responsiveness is achieved by building applications on top of geo-replicated systems, that share data among numerous replicas using a variety of methods to synchronize the data between them.

However, building applications on top of partitioned systems focused on availability brings problems with data consistency. To conserve data integrity within replicas, synchronization is required, resulting in increased communication between them and, consequently, decreased responsiveness. Programmers can pursue several strategies regarding consistency, by choosing from strong to weak consistency models. This choice depends on an application's specific requirements and trade-offs since some users tolerate seeing stale data in exchange for a system that always comes up with a response.

Strong consistency models are often preferred in systems where it is important to ensure that all readers are accessing the most up-to-date version of the data. It is often used in transactional systems, where the integrity of the data is critical. Thus, it requires synchronization mechanisms (e.g. locks, transactions) to ensure that all reads and writes are properly ordered. In contrast, eventual consistency models relax the constraints on when updates become visible to readers, allowing for increased availability and scalability at the cost of potentially weaker consistency guarantees.

Strong eventual consistency offers a balance between the two. It guarantees that if two replicas receive the same set of updates of some shared data, their view of that data is the same, meaning that they have equivalent states, even after independently solving conflicting updates. This can be achieved through various techniques, such as using Conflict-free replicated data types (CRDTs) that use a variety of methods to ensure that conflicting updates can be safely merged, such as tracking the total order of updates to a record, defining arbitration rules between updates and using certain data types that support merging. In this context, causal consistency can also be ensured, as these methods allow operations that depend on each other to be seen and applied by all processes in the same order.

CRDTs are particularly useful in distributed systems where it is not suitable to use strong consistency models due to the overhead of coordination and communication. They can provide a way to ensure the integrity of the data while still allowing increased availability and scalability.

When building a distributed application that needs a distributed data store, it is possible to combine multiple CRDT objects to build more complex data structures, and express the high-level application operations as combinations of low-level operations on the underlying CRDT objects. For example, a distributed store might use a CRDT to represent a set of items, another CRDT to represent a map of key-value pairs, and another CRDT to represent a list of items. Combining these CRDTs in a single store makes it possible to create a data structure that can be updated concurrently by multiple processes without the need for locking or other coordination. However, preserving high-level application invariants in such architecture is far from trivial and poses significant challenges. An emerging approach involves constructing custom CRDTs from sequential code, which entails expanding sequential data types with a distributed specification. This method offers more flexibility than combining pre-existing CRDTs, as it allows specifying the behavior of complex data types and their operations, together with custom conflict resolution strategies that suit the application's needs.

Certain distributed storage systems provide the capability to group application operations into atomic blocks. However, this does not ensure strong consistency, as these blocks are executed locally, and the operations that are synchronized are the inherent ones, potentially leading to state divergences. Alternatively, some systems adopt a hybrid model where it is possible to enforce global transactions to have stronger consistency.

1.2 Motivation

It is difficult to ensure that a collection of CRDTs satisfies high-level application requirements.

It is often hard to directly instantiate an application with an existing CRDT, as there is often a limited set of CRDTs to choose from, with limited operations and invariants, and supporting novel features often requires significant extensions and redesigns. A recent example is the bounded counter CRDT [5]. The solution is, therefore, to translate the application's state into that of an existing CRDT. However, the CRDT will have its own vision of consistency and its own conflict-resolution policy that sometimes does not match the assumptions of the high-level applications written on top of such CRDT.

Having an application structured on top of a composition of replicated data types only makes the challenge even harder, as the CRDTs will ensure convergence, but each CRDT will operate independently from each other, and still not necessarily preserve cross-object invariants assumed by the high-level application.

Even with careful design and implementation, identifying how an invariant might be violated can turn out to be as challenging as defining it. Thus, the difficulty lies in how to devise a strategy to maintain invariants. This is particularly true in systems that operate at a large scale, where there may be many replicas of the same data and where the invariants may be subject to a high volume of concurrent updates.

Some frameworks have been proposed that allow for the formal verification of the behavior of applications composed of basic CRDTs, in order to reason about their behavior and verify that the distributed system adheres to the desired application invariants. Still, they typically require the creation of formal abstract specifications of the application and can involve a significant amount of manual effort to prove that a distributed implementation respects an abstract specification. Additionally, these techniques have not yet been widely adopted in mainstream CRDT development practices. This can make it challenging for developers to use these techniques in practice, as they may not have access to the necessary tools or the expertise to effectively use them.

Considering these challenges, the creation of generic CRDTs from sequential code becomes a good alternative. In this approach, users intuitively reason sequentially, developing a datatype, its operations, and semantic properties, and a CRDT consistent with sequential execution is automatically constructed. This simplifies the process of defining and maintaining application invariants and ensures that the resulting CRDTs are inherently consistent with a conflict resolution strategy that preserves application invariants. As a result, this approach makes the advantages of CRDTs more readily achievable for developers seeking to meet specific application requirements.

1.3 Proposed Work

This work will implement a framework and explore how two recent approaches for the design of CRDTs can be put to use to accomplish “closer to“ generic CRDTs, where users only reason sequentially, developing a datatype, its operations, and semantic properties, and a CRDT consistent with sequential execution is automatically constructed. This framework will be also used to understand and explain the distributed design of complex CRDTs from the semantics of sequential datatypes.

In order to assess the framework and its practicality, some use cases and complex data types will be collected and scrutinized. This analysis aims to comprehend how a sequential specification of classic CRDT examples relates to their distributed behavior. Furthermore, these implementations will be tested to verify the accuracy of both the generic CRDTs implementation and datatype specifications, ensuring that the resulting CRDTs not only meet the desired requirements but also maintain the expected level of correctness and consistency.

1.3.1 Objectives

For a clearer comprehension of the goals of this study, we clarify them into the following three key objectives:

1. Explore the challenges of using CRDTs to build complex, large-scale applications and how these challenges can affect the ability to preserve application invariants.
2. Implement generic CRDT constructions that define a CRDT from a sequential data type.
3. Evaluate how some use cases and complex data types can be implemented using generic CRDT constructions.

1.4 Document Structure

Besides the introduction presented in chapter 1, this document contains six more chapters.

Chapter 2 describes the basic theoretical knowledge needed to understand the problem of designing CRDTs that preserve application invariants.

Chapter 3 enunciates related work previously developed to address the problem, along with the respective trade-offs.

Chapter 4 presents the implementation of a Pure Op-based framework used to transmit and handle messages among replicas.

Chapter 5 demonstrates our implementation of generic CRDT constructions that are used to define a CRDT from a sequential data type.

Chapter 6 illustrates examples of classical CRDTs for specific sequential data types, and demonstrates how they can be obtained by using our generic CRDT constructions.

Chapter 7 discusses the performance of an implemented data type on top of all the generic CRDT constructions.

Finally, Chapter 8 provides a reflection of what was explored and what could still be done to improve this research.

Chapter 2

Background

This chapter presents the fundamental theoretical knowledge essential for understanding the challenges of distributed systems and CRDTs. It explains their key properties and how they work.

2.1 CAP Theorem

Real-world services are expected to possess three desirable properties [11]: Consistency, Availability, and Partition-tolerance. However, it has been proved that only two of the properties can be fully ensured. It is impossible to have a partitioned system that simultaneously provides atomic operations and consistent data while being totally available.

A system has strong consistency if the data is the same for all clients at any point in time as if there was one single node in the system. This means that operations should be totally ordered in all replicas, and when a write operation happens, it is instantly shared with all nodes.

To provide availability, a system should always come up with a response for every request it receives, independently of the number of replicas that are online.

Finally, to have partition tolerance, a partitioned system should be able to tolerate message losses or node failures, without compromising the correctness of the system responses.

Normally, partition tolerance is prioritized in distributed systems, which leads developers to reach a suitable balance between availability and consistency, thus, building systems based on different consistency models.

2.2 Consistency Models

Consistency models describe the consistency level preserved in a distributed system, where multiple nodes may concurrently access and modify the same data. Different consistency models have different trade-offs regarding availability, performance, and the guarantees they provide regarding data consistency, as mentioned in the previous section.

2.2.1 Strong Consistency Models

Strong consistency is the strongest model of consistency that ensures that all read and write operations to shared data are assembled in the order in which they were issued, as if executed on a single centralized replica. Thus, all readers will see a consistent view of the data, even in the presence of concurrent writes. This is normally achieved using a leader-based approach, where a chosen node coordinates updates and ensures that all nodes have the same data. Strong consistency is often selected in applications where data integrity and consistent results are required, such as database systems that handle transactional data like MySQL and PostgreSQL. Transactions are another usual way to achieve strong consistency as they provide a method to ensure that a set of operations are executed in a consistent and atomic way. This topic is explored with more detail in Section 2.3. Nevertheless, these methods can negatively impact performance as they require more time to process operations because of the additional communication and coordination between replicas to guarantee that only one consistent state is always observed.

Operations that involve commutative operations, such as adding or multiplying numbers, do not necessarily require strong consistency in order to maintain the correctness of the system. These types of operations can be reordered between different replicas, which can improve performance. Besides that, applications that can tolerate some level of staleness in the data they access, such as caching systems or social media feeds, do not need strong consistency for the correct operation of the system. In these cases, weak consistency models may be used.

2.2.2 Weak Consistency Models

Eventual consistency is the weakest model of consistency. It is possible for conflicting updates to be applied to the same data simultaneously, as the system will solve those conflicts and converge to a consistent state. As this model allows conflicting updates to occur, it requires a mechanism to solve those conflicts, such as adopting policies like ”last-writer-wins” or even manual resolution. This type of consistency is often used in systems where high availability is more important than strong consistency, including distributed storage systems like Cassandra.

RedBlue consistency is a consistency method introduced by Li et al. [20] for achieving a balance between the speed and consistency of replicated systems. It allows for systems to be as fast as possible while still ensuring consistency when necessary. This consistency is a new approach that addresses this trade-off by dividing operations into two categories: blue operations and red operations. Operations that are classified as blue can have their order of execution change between locations. These operations can be performed efficiently and locally without requiring coordination between sites. On the other hand, red operations must be executed in a consistent order across all sites.

Strong Eventual Consistency (SEC) is a newer model of consistency proposed by Shapiro et al. [22] that lies in the middle of strong and eventual consistency. In addition to ensuring that the system will eventually become consistent, SEC also ensures that nothing will ever compromise a single object of the system during the execution. Therefore, the need for reverting changes to resolve conflicts is eliminated, as it not only consumes resources excessively but also needs a consensus among replicas to handle conflicts in a consistent manner. In other words, it ensures that all replicas of a single object in a replicated system will eventually converge to the same state, regardless of the order in which they receive the same update operations. This condition is typically implemented using Operational Transformation (OT) or Conflict-free Replicated Data Types (CRDTs), specialized data structures designed to follow this model.

Finally, causal consistency is another weak model of consistency when compared to strong consistency. Instead of ensuring convergence to reach the same state, it ensures that the order of operations is maintained within a causal relationship. A causal relationship refers to the association between two operations that are related in time. Causal consistency ensures that any operation that is causally related to a previous operation will be seen by all nodes in the same order.

Causal+ consistency (CC+) was defined [21] and designates a balance between availability and consistency. It follows the basic Causal Consistency model but besides ensuring a causal order of operations, it also ensures that replicas converge to a common state when conflicts happen due to concurrent updates. To achieve this, many systems implementing CC+ use Conflict-free Replicated Data Types (CRDTs) or a "last-writer-wins" rule, where the last update is applied, while any previous updates are overwritten.

However, none of these approaches guarantee a total order of operations, which can result in integrity violations.

2.3 Transactions

A transaction is a sequence of one or more operations (such as reading or writing data) executed together on a database. It ensures that databases remain consistent and can recover from failures, allowing for reliable execution of operations. Transactions follow a set of properties that ensure data consistency and integrity.

2.3.1 ACID Properties

ACID properties are a set of guidelines for designing and implementing database transactions. The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties are important as they help ensure that database transactions are reliable and accurate, even in the face of failures or concurrent accesses:

- Atomicity: When a transaction is executed, all its operations must be successful for the changes to persist otherwise none will. In the event that any operation within a transaction fails, the entire transaction is undone, and the database is restored to its previous state.
- Consistency: A transaction must leave the database in a consistent state. The data in the database must follow all the rules and constraints set by the database schema, such as data types and foreign key relationships.

- Isolation: Transactions should not interfere with each other. If one transaction reads data from the database, it should not be affected by any other transaction written to the database simultaneously.
- Durability: Once a transaction has been committed, it cannot be lost or undone. If the database experiences a failure, the transaction will still be recorded and preserved.

However, the level of consistency provided by Serializable transactions, which is the highest level in traditional ACID databases, cannot be reached while maintaining high availability in scenarios where network connectivity is compromised. Weak isolation and consistency guarantees are often used as a trade-off to improve performance, concurrency and to achieve high availability.

2.3.2 Isolation Levels

Transactional systems usually support different levels of isolation that represent how transactions are isolated from each other.

Serializability is the strongest isolation level that can be used in transactions. It ensures that the execution of concurrent transactions will produce the same results as if they were executed in sequential order by a single machine, which is not ideal in distributed systems.

Weaker isolation models are used to increase the availability of systems such as Read Uncommitted, Read Committed, Repeatable Read, and Snapshot Isolations, the strongest isolation level after Serializability.

2.3.2.1 Snapshot Isolation

Snapshot isolation (SI) is a level of isolation that provides a consistent view of the database at a specific point in time and prevents conflicting transactions from committing simultaneously, thus, avoiding write-write conflicts. Snapshot Isolation enforces a total order of committed transactions when they are committed. This isolation level is widely adopted by databases because it provides improved performance compared to serializability. It avoids the need for blocking during read operations and eliminates the majority of anomalous behaviors, except for the brief fork anomaly that may occur when one transaction reads an object that has been updated by another transaction, and the second transaction reads an object updated by the first transaction.

As it ensures a total order for write operations, it may not be the optimal choice for distributed transactions as it needs increased coordination between nodes.

2.3.2.2 Parallel Snapshot Isolation

Parallel Snapshot Isolation (PSI) is a technique that was implemented in Walter by Sovran et al. [23] and addresses the need for ensuring consistency in a geo-replicated data store that supports transactions. It builds upon the guarantees provided by Snapshot Isolation (SI). It allows for greater flexibility regarding the order in which transactions are committed across different sites. PSI allows distinct nodes to have different orders of committed transactions. This is achieved through the use of asynchronous replication, which guarantees a causal order of transactions among nodes.

It is possible for the long fork anomaly to occur. This scenario can happen when two simultaneous transactions, t_1 and t_2, finish successfully and make updates to distinct pieces of data, followed by two additional transactions that occur subsequently, where one of them observes the changes made by t_1 but not t_2, and the other one notices the changes made by t_2 but not t_1. Nevertheless, PSI has mechanisms to avoid write-write conflicts by aborting any transactions that try to modify the same items concurrently, ensuring that multiple transactions do not write to the same data simultaneously.

2.3.3 Highly Available Transactions

Highly Available Transactions (HAT) [3] are database transactions characterized by providing a high level of availability by avoiding strong isolation properties such as rolling back in case of failures. HATs can continue processing and returning valid results on server failures or network disruptions. This is achieved by using weak isolation and consistency models that do not require all servers to be in perfect synchrony and instead rely on weaker consistency guarantees, such as eventual consistency. These transactions are commonly used in distributed key-value stores, with the degree of isolation and consistency depending on the particular implementation.

However, these ACID isolation levels and distributed data consistencies were evaluated, and Bailis et al. [3] states that guarantees such as causal consistency and read-your-writes can be provided. Still, other desirable semantics like Snapshot Isolation and Strong Serializability cannot be met owing to the lack of means to avoid issues like Lost Update and Write Skew/Short Fork.

2.3.3.1 Transactional Causal Consistency

Transactional Causal Consistency (TCC) is a type of consistency used by Akkoorath et al. [2] in Cure system, that aims to support interactive transactions which enable both read and write operations to occur within one transaction. TCC ensures that all transactions read from a causally consistent snapshot of the data store. This means that the snapshot captured contains updates from previously committed transactions that are causally consistent, in line with the causal+ consistency model (mentioned in 2.2.2), with convergence being achieved using CRDTs. Besides that, atomicity is ensured when multiple objects are updated within a single transaction.

2.4 Operational Transformation

As presented in [17], the challenge of consistency maintenance in systems where multiple users can view and edit the same data simultaneously can be achieved using Operational Transformation (OT) algorithms.

In Operational Transformation (OT), the basic operations supported are `insert` and `delete`, which operate on a linear data structure. When a user performs an operation on their local replica, the operation is immediately applied to the local replica and subsequently transmitted to the other replicas to be consequently applied.

Transformation functions ensure the commutativity of the effect of operations in a state. The effect of new incoming operations is adapted based on the operations that have already been applied to repair inconsistencies. This ensures that the intended changes made by users to the shared data are accurately reflected, aligning with the objective of creating collaborative systems as flexible and adaptable collaboration platforms. With OT, users have the freedom to edit any part of the shared data at any given time.

2.5 Conflict-free Replicated Data Types

Conflict-free replicated data types (CRDTs) are abstract data types specifically designed to be replicated across multiple nodes. As previously mentioned, CRDTs are associated with weak consistency models that aim to guarantee convergence within replicas to increase consistency without losing availability. They are useful in distributed systems where data may be replicated across multiple nodes, and where it is important to have consistency and availability even in the presence of network partitions and other failures. This is achieved with minimal coordination required as they have these two properties: (i) each replica can be independently and concurrently modified without any coordination, and (ii) two replicas that receive the same set of updates will reach the same state in a deterministic way, by ensuring and relying on the commutativity of updates.

CRDTs are typically classified into two different synchronization models - Operation-based and State-based - that have different trade-offs and are suited for different use cases.

2.5.1 Concurrency Semantics

Concurrency semantics is essentially a function that, given a Directed Acyclic Graph (DAG) of updates constructed using the happens-before relation, returns the state of the CRDT after applying these updates. This function plays a crucial role in determining the behavior of the CRDT in the presence of concurrent updates, thereby enabling the system to maintain consistency across replicas. Having a distributed system with shared data structures implies handling concurrent updates. Some updates can be applied in any order and produce the same result. These updates are called commutative. However, some updates may not commute, meaning that the order in which they are applied will affect the final result. In such scenarios, the developer must choose the appropriate semantics for their use case.

When defining concurrency semantics, some relations need to be taken into account, such as:

1. Happens-before relation: if one event happens before another, the first event is guaranteed to have been completed before the second event starts.
2. Partial order of updates: certain updates are concurrent and are not ordered with respect to each other. This is the result of having a set of updates only with a happens-before relation.

3. Total order of updates: all updates in a set are comparable according to some order. Achieving a total order requires additional rules or coordination to ensure that replicas apply updates consistently ordered.

CRDT designs for types such as registers, sets, counters, graphs, and, more recently, bounded counters, have concurrency semantics that ensures convergence, in the sense that the CRDT state is independent from the order according to which concurrent operations are applied. CRDTs can employ a variety of concurrency strategies, including ”add-wins”, where additions have priority over deletions, and ”remove-wins”, where deletions have priority over additions.

2.5.2 Synchronization Models

CRDTs ensure that all replicas will converge and obtain the same data, regardless of the order in which updates are made. This means that the primary focus for developers is to ensure that updates are distributed to all replicas, which is achieved with synchronization models.

2.5.2.1 State-Based Synchronization

State-based CRDTs replicate data by sending their state to a peer replica. They focus on replicating the entire state of the data structure at each replica, and when replicas need to merge their states, they do so by using a merge function that is defined to integrate the state of remote replicas.

There are three requirements to ensure convergence:

- States on replicas are partially ordered, forming a join semi-lattice: for each pair in the set of states of each replica, it is possible to produce a join. The states are partially ordered based on a "happens-before" relation.
- Merge function produces a least upper bound: Updates change the state of a replica by increasing it. For any update `u`, the updated state `u(s)` is greater than or equal to the original state `s`.
- Replicas form a connected graph.

With these properties, merges of state-based CRDTs tend to converge to one common true value.

The main advantage of state-based CRDTs is that they are relatively simple to implement and maintain, but they can result in larger messages or states being transferred during replication. This trade-off makes them appropriate for situations where the data structure is relatively simple and the cost of transmitting the entire state can be tolerated.

2.5.2.2 Operation-Based Synchronization

In Operation-based CRDTs, data replication is achieved by transmitting operations to all replicas. When a modification occurs on a replica, it invokes a `prepare` method, which produces an `effect` function to be executed on other replicas. The `effect` is thus a closure that modifies the state in other replicas. Once applied, the replica executes the `effect` on its local state and disseminates the updates to other replicas. To ensure convergence, the following requirements must be met:

- A protocol that guarantees reliable transmissions.
- When the `effect` function is transmitted according to the causal order, any concurrent `effect`s must have a commutative relationship. If the causal order is disregarded during delivery, all `effect`s must exhibit commutativity.
- The `effect` function must be idempotent if multiple deliveries are possible.

Operation-based synchronization leads to more efficient replication by transferring fewer data during the replication process but may require more complex implementation.

2.5.2.3 Delta-Based Synchronization

Considering State- and Operation-based synchronization, it becomes clear that transmitting the entire state of an object is not always necessary when only a portion of it has been altered. On the other hand, if there are multiple updates to the same state (such as in a counter), it is more efficient to transmit the state once.

Delta-state CRDTs address this issue by combining both synchronization models and propagating delta-mutators, which update the state based on the latest synchronization date. During the initial communication between replicas, a full state transfer is necessary.

To further optimize, in the presence of delays, an operation-based log compaction technique may be beneficial.

2.5.2.4 Pure operation-based synchronization

In operation-based synchronization, the designers have a lot of freedom in defining the `prepare` function, which can lead to complex state structures and large messages. Regular operation-based CRDTs distribute the logic between both the prepare and effect phases. The prepare phase not only collects the required information but also contains some of the decision-making and coordination logic.

On the other hand, in pure operation-based CRDTs, the prepare phase is designed to be generic, and the logic of updating a data type happens during the effect phase. Pure Operation-based CRDTs, as proposed in [7], were introduced to remark the simplicity and efficiency of operation-based CRDTs. Here, the transmission of information is exclusively done with operations through a reliable causal delivery protocol, ensuring the preservation of causality in delivered messages, which are then incorporated into a partially ordered log (PO-log). The entire logic of executing the operation in each replica is delegated to the `effect` function, while the `prepare`, which is made generic (i.e., not data type dependent), serves as a preliminary step to transform the request into a message before applying updates using `effect`, and broadcasting the message to other replicas.

The framework of Pure op-based CRDTs also proposes the concept of stability, where messages become stable if it is guaranteed that no concurrent updates with that message will arrive in the future. An extended API called Tagged Causal Stable Broadcast (TCSB) provides extra causality information upon delivery and is used to inform later when delivered messages become causally stable, allowing log compaction and consequently reducing the storage overhead.

Chapter 3

Related Work

This chapter provides an overview of the existing approaches for addressing the challenge of designing high-level applications on top of CRDTs, and in particular how to preserve application invariants. It explores the trade-offs associated with each approach and specifies which approaches will be adopted in our research, along with the motivation behind the selection.

3.1 Composing Replicated Data Types

Creating an application that uses currently available Replicated Data Types can present challenges in real-world scenarios. One limitation of such RDTs is that developers must store and handle all application data in a general format, like JSON, making it hard to include application-specific concepts or utilize programming constructs like classes and types which need relations to be maintained among them, to successfully represent the application's state.

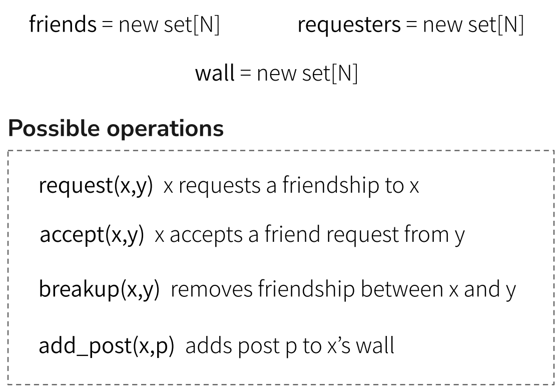
When Conflict-free Replicated Data Types (CRDTs) solve conflicts among concurrent operations, they ensure consistency of operations defined on the specific type (e.g. list properties in a list CRDT, invariants like ≥ 0 in a bounded counter), but it may not align with the expectations of developers relatively to high-level invariants of the application. For example, using a standard list CRDT, if one user moves an element of the list (as a sequence of deletes and inserts) while another edits it concurrently, then the concurrent edits will be lost. As moving items is common in applications, Kleppmann [16] describes a "list-with-move" CRDT as a solution.

Developers could build applications based on CRDTs specifically designed for each application's necessities like Kleppmann did. However, designing these types of CRDTs can be a challenging task that does not scale well for increasingly complex applications.

As current RDTs are insufficient for preserving application invariants, complex applications can use several data types to represent the desired application state. For example, a social network application may need to maintain information about *users*, their *friends*, and their *posts*. Some of these different types of data may need to be combined to represent the overall state of the application. Composing different standard CRDTs makes it possible to provide convergence, ensuring strong eventual consistency guarantees for each data type, consequently enabling the application to handle concurrent updates consistently. Some libraries have been developed in this direction, such as Yjs [13], Automerge [1], and more recently, Collabs [25], which provides modularity and composition for Replicated Data Types.

However, combining different CRDTs with convergence guarantees is not enough for ensuring application integrity, since updates sent by replicas do not have a global order and, when applied on each replica, they may not follow the order in which they were issued. As a result, if a replica emits updates assuming a specific relative order and preserves invariants on its state while applying operations locally, their distributed application of the same operations can potentially break the assumed invariants.

Considering our social network example, we can represent friendship relations and user's `feeds` with arrays of sets, giving the composition presented in Figure 3.1.



```
friends = new set[N]        requesters = new set[N]
                wall = new set[N]
```

Possible operations

```
request(x,y)  x requests a friendship to x
accept(x,y)  x accepts a friend request from y
breakup(x,y)  removes friendship between x and y
add_post(x,p)  adds post p to x's wall
```

Figure 3.1: Example of a social media representation (Adapted from [12]).

For both cases, there are high-level application invariants that must be maintained. In the representation of figure 3.1, it is assumed that friends relationships are symmetrical, and if two users are already friends, their requests should not be in `requests` array anymore. It is also assumed that a user must only see posts on their `feed` from people they are friends with.

To ensure convergence, this example can be implemented using different combinations of set CRDTs, each with a specific concurrency semantics. However, as previously said, the convergence guaranteed by CRDTs is not enough to prevent invariant violations, as they have no intrinsic knowledge of the high-level application invariants.

3.2 Supporting Application Invariants

The problem of preserving invariants is mitigated with existing approaches such as:

1. Formally model the semantics of the datatype and the CRDT, and formally check that the CRDT concurrency semantics does not compromise the application's semantics and its specified invariants.

2. Letting violations occur and propose repairs when it is detected that the system is inconsistent.
3. Using different consistencies depending on the operation performed. Operations that might violate invariants will have more synchronization than others.
4. Synthesize correct CRDT implementations from an abstract specification that expresses the necessary invariants.

All of these methods come with trade-offs and require identifying the critical operations and their effect on the application's state.

3.2.1 Formal Verification

One way of building applications on top of eventually consistent stores is to use a composition of replicated data types and transactions over these objects to update them. However, these semantics are subtle, which makes it difficult to reason about the behavior of the application on concurrent updates. A new programming concept of a composite replicated data type was introduced by Gotsman and Yang [12] that formalizes this method of organizing applications.

A composite replicated data type made up of multiple components of replicated data types and includes composite operations for accessing them through transactions.

Going back to our previous example of a social network, stated in representation 3.1, following the composite implementation, friend relationships would be represented with sets defined with conflict resolution policies, such as "remove-wins" or "add-wins" and have operations built with transactions to guarantee atomicity and causal order of updates, as shown in Figure 3.2.

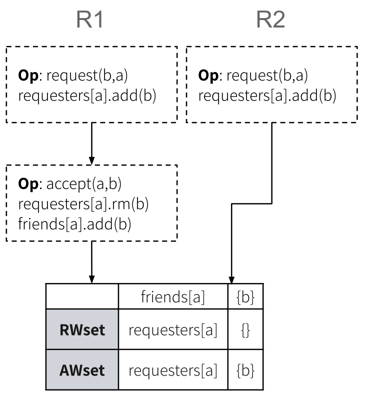


Figure 3.2: Representation of conflict results for different conflict resolution policies (adapted from [12]).

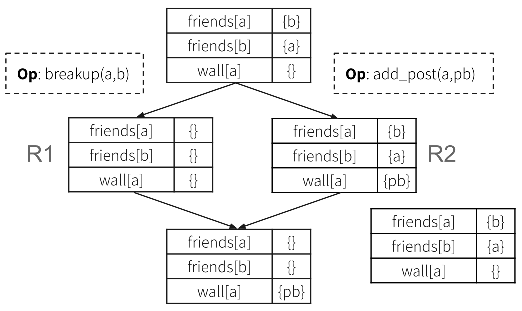
The paper puts forth the idea that assigning the ”remove-wins” behavior to requesters is a suitable solution to preserve the invariant, which holds that if person `a` and `b` are friends, then `a` cannot appear in the list of requesters for `b`. In figure 3.2, two users are managing the same account, b, and both send friend requests to a concurrently. With the "add-wins" approach, if a accepts one of the requests, it would only affect the request they see, while the other request would later be transmitted to all replicas in the system. This would result in get calls in the implementation, returning b as both a friend and requester of a, violating the integrity invariant. The ”remove-wins” policy for requesters guarantees that when a user approves or rejects a request, it also deletes any other identical requests made at the same time.

As this behavior is very subtle, this method abstracts from the internal design of the data type, allowing developers to define the behavior of the composed CRDT by only referring to the composed operations and not the operations of each constituent CRDT. This helps to reason about applications built using this method. However, these formalization techniques still require reasoning about the distributed behavior of the composed CRDT, while using advanced techniques/tools and manual verification to be considered useful for a common developer.

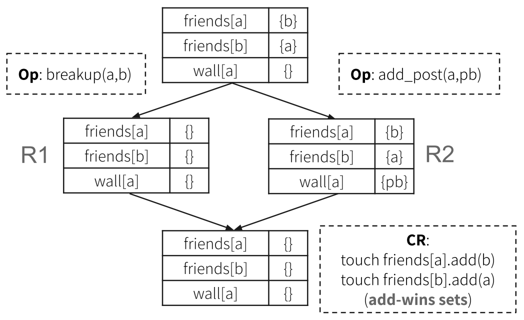
3.2.2 Invariant Repair

Avoiding formalizations, some works provide more practical solutions to preserve invariants, such as IPA introduced by Balegas et al. [6]. It is a system that preserves invariants on weakly consistent replicated databases. This software detects conflicting operations and suggests the required modifications to them.

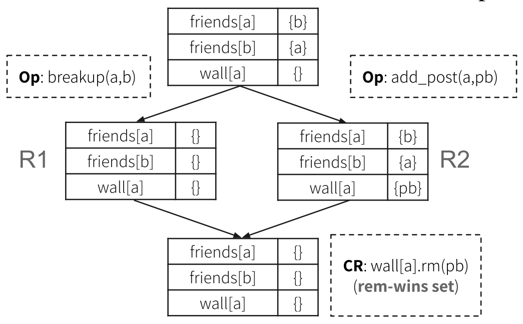
The core concept is to improve operations by incorporating updates that preventively ensure the maintenance of invariants on concurrent updates. These updates execute when there is a need to correct undesirable states.



(a) Invariant Violation



(b) Invariant repair by friendship recreation



(c) Invariant repair by post removal

Figure 3.3: Representation of conflicts and resolutions of two operations.

Looking into our social network example, if on different replicas, the operation `breakup(a,b)` runs concurrently with `add_post(a,` p_b`)`, the system will converge to a state where `a` has a post of `b` without a friendship, violating an invariant as shown in Figure 3.3a.

To prevent this violation, the system can (i) remove the post that was added to `a`'s wall or (ii) restore the friendship to its previous state.

To solve the invariant violation by recreating the friendship relation, the system extends the effects of adding a post to touch friendship `f`. The touch operation has no discernible impact as it only modifies metadata to ensure that concurrent execution is identified and handled according to the specified conflict resolution strategy. Therefore, there is no coordination needed to solve this, as shown in Figure 3.3b.

Since it is a social network, it makes more sense to solve this specific case with approach (i). Thus, to prevent the invariant violation by deleting posts in removed friendships, the system extends the effects of the remove friendship `f` to preventively remove any concurrent addition of posts related to `f`. This does not produce any observable effect because the removal operation should leave no elements on the wall. Choosing a "remove-wins" strategy for the wall set ensures that concurrent additions of posts will not have any impact, as Figure 3.3c states.

Despite being a good approach to avoid coordination on maintaining invariants, creating the specifications needed to find the proper updates that preventively ensure the preservation of invariants can require as much effort as writing the code itself.

Easing specification effort, De Porre et al. [9] introduced Explicitly Consistent Replicated Objects (ECROs) that are specified through a distributed definition that outlines the application's semantics and ensures the replicated state's consistency via invariants. ECROs have the capability to previously identify potential conflicting operations (via program analysis of the operation definitions), which is used during runtime to reorganize conflicting operations in order to maintain the invariants.

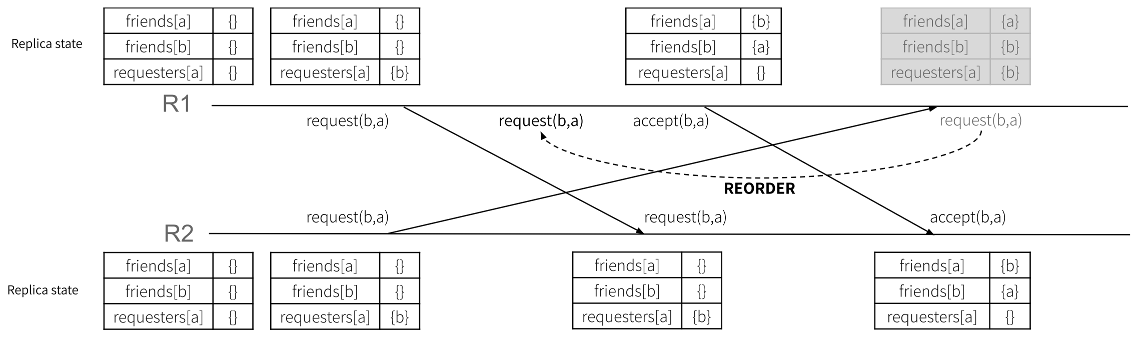


Figure 3.4: Representation of reordering of operations to preserve invariant (adapted from [9]).

Considering the same situation presented in subsection 3.2.1, which is represented in Figure 3.4 in a chronological form. Here, `R2` requests friendship to `a`, and `R1` makes the same request, followed by an acceptance of that request, which removes `b` from `requesters` of `a` and adds it to its friends. After this concurrent situation, the request made by `R2` is sent to `R1`, and the acceptance made by `R1` is sent to `R2`. This behavior will lead to an invariant violation where `R1` ends up with a and b being friends and b still belonging to a requesters. Thus, as shown in Figure 3.4, the ECRO will reorder the operations in R1 and lead to a consistent state that does not violate the invariant.

However, this solution has a trade-off since it involves the presence of rollbacks, which can result in a client observing operations being executed in a certain order initially, but later perceiving them as being applied in a different order. Besides that, it is not always feasible, as ordering operations may not be enough to preserve invariants such as bounded counters. In that case, ECROs would use more coordination between these operations, leading to stronger system consistency when needed.

3.2.3 Mixed Consistencies

Anomalies are avoided in systems that enforce strict serializability, meaning that all operations are executed in the order they occur. However, weaker consistency models like serializability or snapshot isolation can also lead to increased latency and decreased availability due to the need for frequent coordination among replicas. An alternative approach is to take advantage of both weak and strong consistency models, depending on the operation being performed. Redblue consistency already does this but does not consider invariants, sometimes leading to unnecessary coordination.

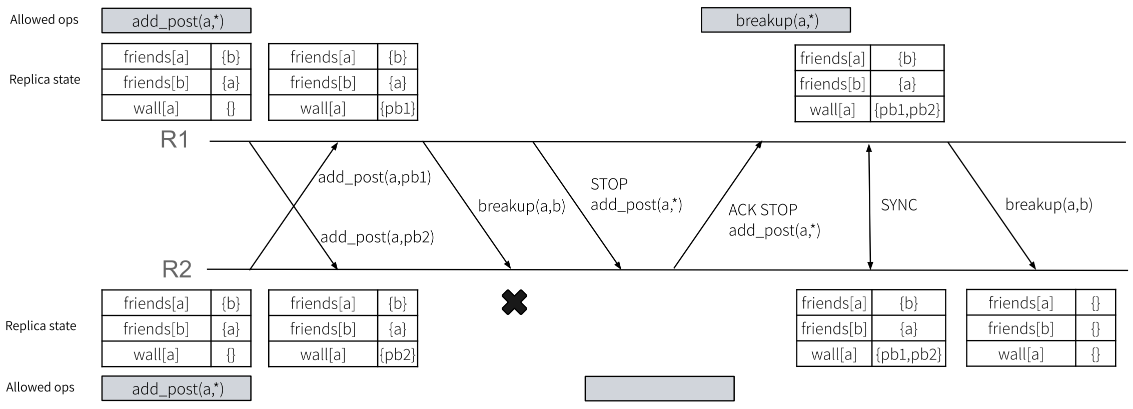


Figure 3.5: Preserving invariant with mixed consistency method.

Concerning that, Indigo provides Explicit Consistency proposed by Balegas et al. [4]. Explicit Consistency aims to extract more information about applications to support properties that enable the system to avoid coordination. Invariant violations are avoided using reservations, which assign rights to operations on a multi-level lock basis. This alternative consistency model identifies which operations are unsafe if executed concurrently and allows developers to select "violation-avoidance" or "invariant-repair techniques."

In our example, add_post(a, p_b) and breakup(a,b) represent an I-offender set. An I-offender set is a set of operations that, if executed concurrently, can result in a violation of invariant.

With the "invariant-repair" approach, operations can execute concurrently, but conflict resolution rules should include code to fix the invariant violation. Here, the merged state might ignore the newly added post, similar to IPA in the previous example (3.2.2).

In the "violation-avoidance" approach, the system restricts concurrency enough to avoid invariant violation. Considering our example, any replica is allowed to execute `add_post(a,`p_b`)` as long as all replicas are forbidden to run `breakup(a,b)`. To execute `breakup(a,b)`, it is necessary to obtain permission on the `reservation` for `breakup(a,b)`. This guarantees that deleting a friendship from another user will not execute concurrently with adding an element to `a`'s wall.

Another approach is Carol [19], a programming language designed to enable modular and sequential programming and verification of replicated store operations without the need for understanding the concurrent execution model. The key idea is a two-state predicate that establishes a relationship between the locally-viewed store and the hypothetical remote store where updates from an operation may be eventually applied. In Carol, this predicate allows programmers to declare specific consistency requirements, enabling precise specification of invariants. Programmers specify *guards*, which are straightforward data pre-conditions. The algorithm statically translates those *guards*, adding coordination when needed to enforce the preservation of invariants.

Although solving the problem of preserving invariants, these approaches limit availability when too many operations that can lead to invariant violations are executed.

3.2.4 Consistency by Design

Most of the previous approaches propose constructions that yield consistent CRDTs. For instance, ECROs take a sequential data type and devise a distributed algorithm that guarantees convergence by reordering and reapplying operations as needed. The approaches catalogued in this section take a sequential data type, and output a "standard" CRDT.

Katara, an open-source system that uses program synthesis techniques to simplify the creation of CRDTs, was introduced by Laddad et al. [18]. This system transforms annotated sequential data type implementations in languages like C/C++ into complete CRDT implementations with equivalent functionality. Users only need to add a basic function to their sequential data type annotations that outline the order of conflicting operations. Katara then verifies the synthesized CRDT candidates against both the sequential semantics and the conflict resolution policy specified by the user, making the CRDT creation process easier and more attainable.

Program analysis and SMT encoding is used to perform such verification and provide CRDTs convergent by construction. However, this synthesis technique is usually limited to small CRDTs and would require expansion to address invariants effectively.

Another similar approach was introduced by Kaki et al. [15], where replicated data types, called Mergeable Replicated Data Types, are also created from annotated sequential specifications. Here, the replicated state is represented by versioned data structures that are constantly changing and originated from a shared original version. The synchronization is made by pairs of replicas merging their concurrent versions into a single convergent version that retains the essential characteristics of their parents. This merging process is assisted by additional information from the lowest common ancestor (LCA) of the merging versions. This provides correct merge functions automatically for data type definitions that can be combined at any level of complexity. Yet, this approach does not take into account the invariants of an application.

3.2.5 Semidirect Product

A new CRDT construction technique, the *semidirect product of op-based CRDTs* is a restricted kind of Operation Transformation (OT) presented in [24]. It is an advanced composition pattern that combines two op-based CRDTs integrating their respective operations to create a new CRDT, handling conflicts between their concurrent operations in a uniform way.

This approach involves using a transformation function which is employed to appropriately adjust operations from the first CRDT, considering concurrent operations from the second CRDT. However, it does not take invariants into account.

3.3 Discussion

The main challenge explored in this section is how developers can write familiar sequential code and translate it to a distributed implementation. One solution to this problem is to introduce coordination, which allows all replicas to agree on the order of execution. However, as previously discussed, this approach is expensive for many applications, particularly on a global scale, as it weakens the benefits of replication by requiring high-latency communication between nodes.

All of the previously discussed approaches have trade-offs, such as manual proof effort or unavoidable coordination. We will follow the general approach (adopted by ECROs and other synthesis techniques) of describing a type, its operations, and their properties and consequently generate a CRDT.

In particular, we will start from a pure op-based framework, identified in [7] as follows:

> "The Pure CRDT framework makes the design of op-based CRDTs "almost" generic. In our experience, making them fully generic is impossible due to the native semantic discrepancy of the designed data types — making them more generic will be impractical, as we've seen in the PO-Log based CRDTs before compaction".

We will then follow closely the approach laid out by Explicit Conflict-free Replicated Data Types [9] towards the design of "closer to" generic pure op-based CRDTs. As in ECROs, our approach considers that users define a data type, its possible operations and some associated properties, and a generic pure op-based construction subsumes a CRDT that is consistent with the semantics of the data type. In particular, this means that the distributed behavior of the CRDT will always be explainable as a sequence of operations on the data type. Along the way, and unlike ECROs, we will also draw connections to the Semidirect Product [24] approach as a way to make our construction more efficient.

Chapter 4

Implementation of a Pure Op-based CRDT Framework

Our framework was developed from scratch using the Golang programming language, leveraging a range of beneficial open-source tools and standard-library packages to construct both the middleware and client processes.

The client is used by the application/data type that leverages the tagged causal delivery service to establish communication with the middleware process. All requests arrive from the application at the client process. The middleware process is responsible for executing the necessary tasks to broadcast and receive messages while guaranteeing tagged causal delivery and stability.

There were alternative options for middleware implementations, such as the one presented by Bauwens and Boix [8] in Flec, a versatile programming framework designed for the development and use of CRDTs. Flec's key strength lies in its portability, with the ability to operate across many platforms, making it highly adaptable to various environments. However, since the efficiency and support of the middleware were not the primary focus of this project, and Flec's code was not readily accessible, we decided to develop our own implementation.

To implement the tagged causal stable broadcast, the middleware, and client processes adhered to the model outlined in Younes [26], allowing only minimal modifications. Such adherence enabled a reliable delivery of messages with causality and stability information, essential for the development of pure operation-based CRDTs [7]. This provides directly enables to reconstruct a happens-before relation from the PO-Log and provides support for easy PO-log compaction over time.

We intentionally omitted a network layer because it was not crucial for the scope of our research. Consequently, nodes function as threads, and the message transmission is done using Golang channels between these threads.

4.1 Tagged Causal Stable Broadcast

As noted earlier, pure operation-based CRDTs need a reliable broadcast protocol. Given this requirement, besides considering a static group of processes, [26] defines a set of simplifying assumptions concerning this protocol:

- *"No message is delivered more than once"*
- *"No message is delivered unless it was broadcast"*
- *"If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j"*
- *"If one correct process delivers a message m, every correct process eventually delivers m"*

In this context, the term 'receive' refers to a message's arrival at a certain process via the communication layer. On the other hand, 'delivering a message' denotes the process of transferring an already received message to the client layer for further use.

4.1.1 Causal Broadcast

Causal broadcast represents a delivery mechanism that establishes guidelines for employing the previously mentioned timing primitives, predominantly logical clocks, to accurately order events following the 'happens-before' relation. Existing implementations internalize this information without making it accessible to the client. However, as specified in the pure op-based CRDT framework [7], our implementation includes a 'tag' on each message, transmitting causality information among processes, leading to simplified state structures and smaller messages. To accurately depict the causality between operations, we employ vector clocks.

The algorithm of our causal broadcast implementation has the following assumptions, also defined in [26]:

- *"Each process step takes a finite time to occur"*
- *"Message transfer delays are unpredictable but finite"*
- *"Communication channels are reliable"*
- *"Computation is failure-free"*

4.1.2 Causal Stability

Causal Stability is another extension present in the pure op-based framework [7]. Causal stability is crucial for an efficient implementation of CRDTs, as it ensures that stabilized messages will not have any concurrent messages in the future. This requirement is essential for compressing the log of operations over time, which allows replicas to handle messages efficiently without needing to maintain all operations from the beginning.

A protocol with this property means that the delivery of a message with a timestamp t' concurrent with t at node i is strictly forbidden once t achieves causal stability at i. To achieve causal stability at a specific node, it is not enough for a message to have been received by all nodes, all nodes must have also delivered it, and no additional concurrent messages should be delivered at that node.

Each node i maintains a map M_i (mapping node identifiers to vector clocks), which records the most recent vector clock received locally from each respective node. This enables establishing a function that determines the greatest lower bound on messages originating from node j and delivered to all nodes.

4.2 Architecture

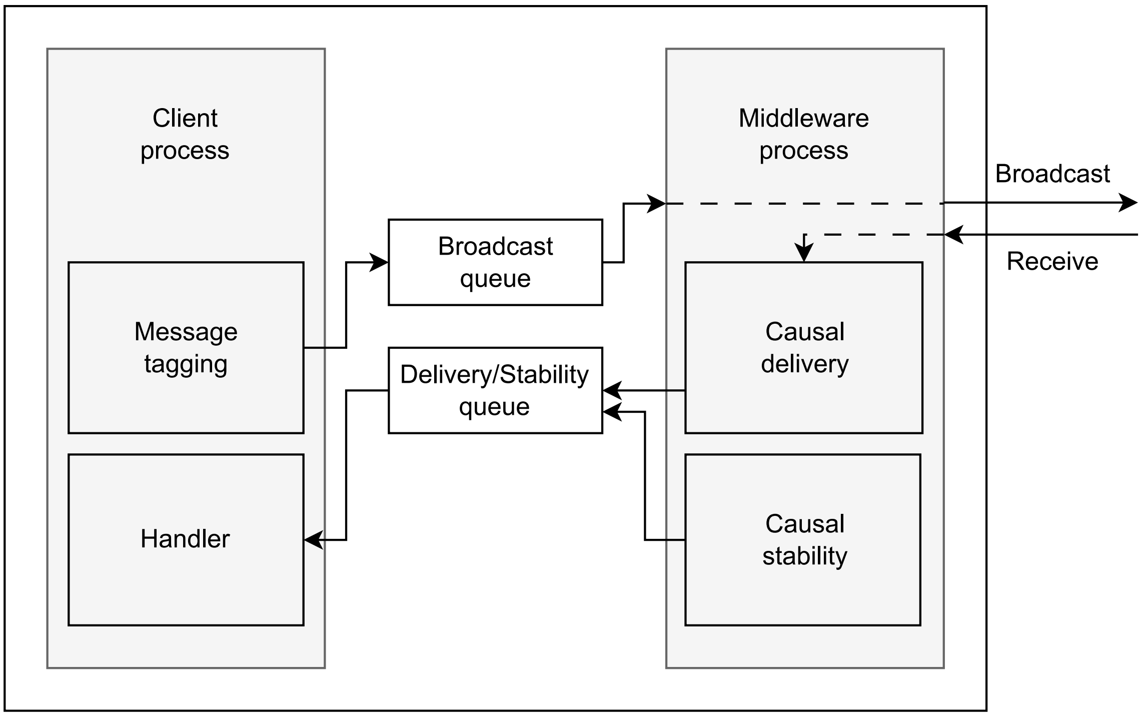


Figure 4.1: Framework architecture (adapted from [26]).

Figure 4.1 depicts the framework's structure designed to ensure both causality and stability within the system. It illustrates the essential processes involved and showcases the intra- and inter-communication flows that occur among nodes. The subsequent sections will delve more comprehensively into the workings of both client and middleware processes.

The client process serves as a basis for our generic pure op-based CRDTs, which we will delve deeper into in the forthcoming Chapter 5. This process plays an intermediary role, enabling communication between the CRDTs and the middleware and, consequently, with other nodes within the network. The middleware is responsible for ensuring causality and stability information to the pure op-based CRDTs.

In the system, the client process serves as the entry point for all requests originating from the application layer. It directs these requests to the broadcast queue, where they are subsequently broadcasted to the other middlewares within the network. The middleware not only receives messages from the local client but also receives messages from all other nodes across the network via their respective middlewares. To uphold causality and stability, the middleware assesses each incoming message and uses the delivery/stability queue to send them to the client while preserving their order and applying appropriate tags.

Listing 4.1 specifies the interface and the implemented structure of a client object. It contains a generic CRDT which implements a CrdtI interface containing the methods `Effect`, `Stabilize`, and `Query` used to handle delivered messages, handle stabilized messages, and retrieve the state of the CRDT, respectively. The specification of these procedures on each generic CRDT will be further explained in the subsequent Chapter 5.

```
type CrdtI interface {
    // Effect callback called when a message is ready to be delivered.
    Effect(msg communication.Operation)
    // Stabilize callback function is called when a message is set to stable.
    Stabilize(msg communication.Operation)
    // Query made to a client which returns the current state of the CRDT
    Query() any
}

type Client struct {
   Crdt          CrdtI
   id            string
   clients       map[string]chan any
   middleware    *middleware.Middleware
   VersionVector communication.VClock
}
```

Listing 4.1: Implemented client structure.

Listing 4.2 shows the implementation structure of the middleware, instantiated upon client creation. The communication between the two processes is made using two Go channels, `causalQ` and `bcastQ`, which function as FIFO queues. The `causalQ` channel is used to send causally ordered messages to the client, while the `bcastQ` channel is employed for transmitting messages to the middleware, enabling it to broadcast them to the other nodes' middleware. All the necessary fields for the communication flow and message handling are described in Figure 4.1.

```
type Middleware struct {
   client           string                         // client id
   channels          map[string]chan any           // all channels of the network
   DeliveredVersion communication.VClock           // last delivered vector clock
   ReceivedVersion  communication.VClock           // last received vector clock
   bcastQ            chan communication.Message    // channel to receive messages
   causalQ     chan communication.Message    // channel to causally deliver messages
   DQ          []communication.Message       // messages awaiting causal predecessors
   Observed    VClocks                       // vector clocks of observed network
```

```
  StableVersion     communication.VClock  // stable vector clock
  SMap              SMap                  // unstable messages already received by
   the client.
}
```

Listing 4.2: Implemented middleware structure.

4.3 Client Process

The process of preparing and receiving messages is shown in Algorithm 1. The state is represented by a vector clock V_i, with a length equal to the number of nodes in the network. Every position starts at 0 and is incremented when receiving messages.

Algorithm 1 TCSB algorithm at the client process for node *i* (adapted from [26]).

```
Require:
 1: $V_i : \mathbb{I} \rightarrow \mathbb{N}$                         # delivered vector clock
Ensure:
 2: $V_i = j \mapsto 0 | j \in \mathbb{I}$

 3: procedure prepare($op, value$)
 4:     $V_i = V_i[i] + 1$
 5:     Effect($i, V_i, op, value$)
 6:     $(i, V_i, op, value) \rightarrow bcastQ$     # enqueues message to bcast queue
 7: end procedure

 8: on $causalQ \rightarrow (type, j, V_m, op, value)$   # dequeues message from causal queue
 9:     if $type = dlv$ then
10:         $V_i[j] = V_m[j]$
11:         Effect($i, V_i, op, value$)           # crdt handles delivered message
12:     else if $type = stb$ then
13:         Stabilize($j, V_m, op, value$)        # crdt handles stabilized message
14:     end if
15: end on

16: function QUERY
17:     return Query()                           # queries crdt state
18: end function
```

Queries are made through the `query` method of the Client and requests are made via the `prepare` method. This method is responsible for converting the user request (operation type and value) into a message that the middleware process broadcasts to each node within the network, and applying the operation locally.

This message is then placed into the `bcastQ` FIFO queue, as depicted in Figure 4.1, and the `Effect` method is invoked.

The local context operates as a vector clock, denoting the last messages received at the client process. It establishes the causal dependencies of the message being prepared. The responsibility of maintaining a record of these dependencies lies with the client process, ensuring the causal tracking of messages.

Messages are delivered to the client through the `causalQ` FIFO queue, the sender's entry in V_i is updated and `Effect` is invoked, or `Stabilize` is invoked, depending on the type of the message assigned by the middleware.

4.4 Middleware Process

The Middleware process is described in Algorithms 2 and 3, and ensures causality and stability. Each node has a vector clock R_i used for keeping track of the received messages, another vector clock V_i for delivered messages, and a queue DQ_i with messages that do not respect causality, waiting to be delivered. Vector clocks R_i and V_i have the vector clock values of the last received and delivered messages, respectively, and have their entries initialized at 0, while DQ_i is set to empty.

Algorithm 2 TCSB algorithm at the middleware process for node i (adapted from [26]) - Part 1.

```
Require:
 1: $V_i : \mathbb{I} \to \mathbb{N}$                                  # delivered vector clock
 2: $R_i : \mathbb{I} \to \mathbb{N}$                                  # received vector clock
 3: $DQ_i$ :                                                            # delivery queue
Ensure:
 4: $V_i = j \mapsto 0 | j \in \mathbb{I}$
 5: $R_i = j \mapsto 0 | j \in \mathbb{I}$
 6: $DQ_i : \emptyset$

 7: on $bcastQ \to (i, V_i, op, value)$                    # dequeues message from bcast queue
 8:     $V_i[i] = V_i[i] + 1$
 9:     updatestability($i, V_i, op, value$)
10:     broadcast($i, V_i, op, value$)                     # broadcasts message to all other nodes
11: end on

12: on receive($j, V_m, op, value$)                          # receives message from network
13:     $R_i[j] = R_i[j] + 1$
14:     if $V_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k], \forall k \neq j$ then          # checks causality
15:         $(\text{dlv}, j, V_m, op, value) \to causalQ$        # enqueues message to causal queue
16:         updatestability($j, V_m, op, value$)
17:         deliver($j, V_m, op, value$)           # checks if waiting messages can be delivered
18:     else
19:         $DQ = DQ + (j, V_m, op, value)$              # waits on DQ until causality is satisfied
20:     end if
21: end on

22: procedure deliver
23:     while $(j, V_m, op, value) =$ getDQmsg() do                     # gets waiting message
24:         if $V_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k], \forall k \neq j$ then      # checks causality
25:             $V_i[j] = V_i[j] + 1$
26:             $(\text{dlv}, j, V_m, op, value) \to causalQ$    # enqueues message to causal queue
27:             updatestability($j, V_m, op, value$)
28:             updateDQ()                         # removes delivered message and resets get
29:         end if
30:     end while
31: end procedure
```

Messages arrive at the middleware from the network and from the client process through the `bcastQ` queue. When a message is received from the client, V_i is updated, and the message is broadcasted to all other nodes.

Upon receiving a message from the network, the entry $R_i[j]$ is incremented. The system then verifies if all causal predecessors of the message have already been delivered. If not, the message is added to DQ_i for future dispatch. However, if all predecessors have been delivered, the message is forwarded to the client via the `causalQ` queue. The `updatestability` method is invoked to refresh the stability status of messages. The `deliver` method then traverses DQ_i. This traversal is necessary because the delivery of a message could potentially trigger the readiness of other queued messages if one of their causal predecessors has been received and delivered.

The algorithm for causal stability is described in Algorithm 3. It employs a stability matrix, denoted as M_i, a stable vector clock, SV_i, and a map of stable dots, referred to as $SMap_i$.

Algorithm 3 TCSB algorithm at the middleware process for node i (adapted from [26]) - Part 2.

```
Require:
32: $M_i : \mathbb{I} \rightarrow V_i$                                   # stability matrix
33: $SV_i : \mathbb{I} \rightarrow \mathbb{N}$                            # stable vector clock
34: $SMap_i : \mathbb{D} \rightarrow \mathbb{M}$                          # stable dots map
Ensure:
35: $M_i = j \mapsto V_i | j \in \mathbb{I}$
36: $SV_i = j \mapsto 0 | j \in \mathbb{I}$
37: $SMap_i = \emptyset$

38: procedure updatestability($j, V_m, op, value$)
39:     $M_i[i] = V_i$
40:     if $i \neq j$ then
41:         $M_i[j] = V_m$
42:     end if
43:     $SMap_i[(j, V_m[j])] = (j, V_m, op, value)$           # stores message as stable dot
44:     $NewSV =$ calculateSV($j$)          # calculates the greatest lower bound vector clock
45:     if $NewSV \neq SV_i$ then
46:         $StableDots = NewSV\ SV_i$                       # obtains new stable messages
47:         stabilize($StableDots$)                    # sends stable dots that became stable
48:         $SV_i = NewSV$
49:     end if
50: end procedure

51: procedure stabilize($SD$)
52:     $S =$sort($SMap_i[d] d \in SD$)                          # sort stable dots by causality
53:     for $(j, V_m, op, value) \in S$ do
54:         $(\text{stb}, j, V_m, op, value) \rightarrow causalQ$       # enqueue stable message to causalQ queue
55:     end for
56:     $SMap = SMap - SD$                  # removes stable dots that became stable from map
57: end procedure
```

The matrix M_i has a space value of N x N when the network has N nodes, where row j contains the last delivered message from node j. SV_i is a vector clock with N entries where each position is the minimum of the same column in M_i, which represents the greatest lower bound vector clock of the matrix. $SMap_i$ saves the messages that are delivered but have not yet achieved stability.

As shown in Algorithm 2, `updatestability` is called every time middleware receives a message. Here, the row $M_i[i]$ is updated with the received message vector clock, and the message is added to the $SMap_i$. A new stable vector is computed by identifying the greatest lower bound vector clock within the matrix M_i. The vector clocks of the stable dot messages that have become stable are obtained by excluding vector clocks until the current stable vector from the new stable vector. Subsequently, these messages are sorted in accordance with their causality and dispatched to the client via the *causalQ* queue designated as a stable message, consequently, the stable dots map and the current stable vector are updated.

Some functions and optimizations of the algorithm have been omitted, given that their specification is not relevant to our objective. For a comprehensive view of the entire algorithm, please refer to [26].

Chapter 5

Generic CRDT Constructions

Given the pure operation-based framework outlined in the previous Chapter 4, we will now define four generic CRDT constructions that instantiate the `Effect`, `Stabilize`, and `Query` methods of the CRDT interface. These constructions are generic in the sense outlined at the end of Chapter 3: they receive a sequential data type and implement a replicated type that follows the CRDTI interface, as depicted in Figure 5.1.

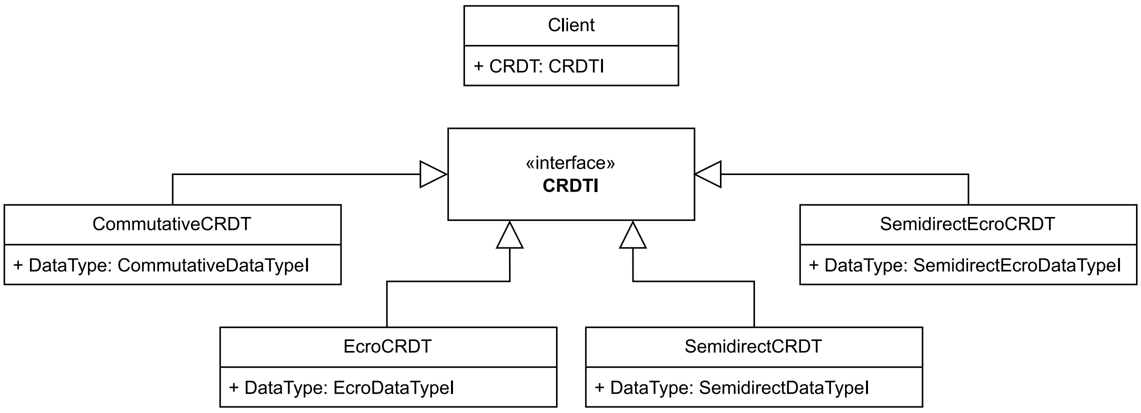

Client
+ CRDT: CRDTI
«interface»
CRDTI
CommutativeCRDT
+ DataType: CommutativeDataTypeI
SemidirectEcroCRDT
+ DataType: SemidirectEcroDataTypeI
EcroCRDT
+ DataType: EcroDataTypeI
SemidirectCRDT
+ DataType: SemidirectDataTypeI

Figure 5.1: Application layer architecture.

As outlined in Chapter 4, the `Effect` method is triggered by the Client, (through the CRDTI instantiation as depicted in Figure 5.1) to allow the generic CRDTs to handle local or delivered operations made to the data type that it instantiates. Meanwhile, when a stabilization message is delivered to the client process informing that a previously delivered operation became stable, the `Stabilize` method comes into play to allow the operation to be handled by the generic CRDT. The `Query` method is also invoked by the client when a user wants to obtain the CRDT state.

The following sections delve further into the specific purpose and implementation details of each of our four generic CRDT constructions.

5.1 Commutative CRDT

If all concurrent operations for a given data type commute, the design of a pure operation-based CRDT becomes straightforward in the sense that the distributed effect of each operation corresponds to simply applying the sequential operation. This is because the particular order in which concurrent operations are applied won't affect the eventual state of the data type due to the commutative property of concurrent operations; non-concurrent operations are guaranteed to be applied in causal order by the middleware.

Commutative data types only need to implement the `Apply` method of the CommutativeDataTypeI interface, specifying how an operation is applied to the state.

5.1.1 Algorithm

The CRDT `Effect` method only invokes the `Apply` method from the data type to update the state, and the CRDT `Query` method returns the CRDT state as shown in Algorithm 4.

Here, the CRDT `Stabilize` method is empty, because commutative operations do not require logging and, therefore, log compaction is unnecessary.

Algorithm 4 Commutative CRDT algorithm.

```
Require:
 1: $State : any$                                          # crdt state

 2: procedure Effect($op$)
 3:     $State = $ Apply($State, op$)    # data type applies operation to stable state
 4: end procedure

 5: procedure Stabilize($op$)
 6:                                                 # ignores stable operations
 7: end procedure

 8: function Query
 9:     return $State$                                       # returns state
10: end function
```

The fundamental insight behind CRDTs lies in the commutativity of concurrent operations. This means that the main challenge in creating a CRDT is figuring out how to reinterpret a data type's non-commutative sequential operations as commutative effects within a distributed environment. Any existing CRDT can then be implemented via our commutative interface. Two details are worth noting:

1. Translating a sequential data type into a commutative CRDT often requires changing the abstract type interface and its operations. Classic examples are the Multi-value Register (MVR), which models a distributed register that can hold multiple concurrent values and

whose abstract type can be more naturally seen as a set, or the Replicated-Growable Array (RGA), which models distributed lists by assuming unique stable position identifiers instead of indices, and list operations on identifiers instead of indices. The latter will be explained in more detail in Chapter 6.

2. Given that the middleware already supplies causal information, numerous traditional CRDT examples such as MVR, RGA, and others can be simplified by removing some of the causality-related meta-information suggested in their original designs. This aspect was already explored for various CRDTs with the introduction of Pure op-based CRDTs [7].

5.2 ECRO-like CRDT

The previous construction allows to turn a commutative data type into a CRDT, but does not give any additional insight into how a data type with non-commutative operations can be turned into a CRDT.

This generic CRDT emulates the principles adopted by ECROs, as outlined in [9]. Here, a user-defined arbitration order is established among operations. In an effort to maintain this order for concurrent updates, operations are rearranged as needed. This results in the execution of operations in identical order across all replicas and guarantees operation stabilization, essential for log compaction.

The ECRO approach is fully generic and can be used to turn any data type into a CRDT, without considering the semantics of its operations. The ECRO will ensure convergence by deterministically ordering operations in each replica and rolling back updates when inconsistencies are found. Regardless of the particular actions or changes an operation might embody, all replicas will ultimately process the same sequence of operations. The semantic information about operations is only used as an optimization. In particular, the original ECRO work [9] uses program analysis on the defined data type to infer which operations commute. This knowledge helps minimize the number of rollbacks and avoid unnecessary relations between commutative operations, thus optimizing the processing time for operations.

5.2.1 Data Type Interface

The data type interface of the ECRO CRDT, specified in Listing 5.1 is an extension of the commutative data type interface, designed to give additional information about updates. In addition to the `Apply` method, it also incorporates the `ArbitrationOrder` method to establish a desired arbitration order for concurrent updates. This method takes two operations and returns a boolean indicating whether the first is smaller or equal to the second. The `Commutes` method accepts two operations and returns a boolean to indicate whether these operations are commutative. This method allows algorithm optimization, as (even causally dependent) commutative operations can be applied in any order. These two methods together provide the generic CRDT with the capacity to calculate a desired order of updates.

Concrete examples of the instantiation of these methods are presented in the following Chapter 6.

```go
type EcroDataTypeI interface {
   // Apply operations to a given state.
   Apply(state any, operations []communication.Operation) any

   // Check if two operations are ordered
   ArbitrationOrder(op1 communication.Operation, op2 communication.Operation) bool

   // Check if two operations commute
   Commutes(op1 communication.Operation, op2 communication.Operation) bool
}
```

Listing 5.1: Data type interface of ECRO CRDT.

5.2.2 Algorithm

As shown in Algorithm 5, this generic CRDT features an operation log, `Unstable_ops`, represented by a directed graph that maintains causal and arbitration relations between operations. The `Stable_st` represents a stable state of the data type, where all operations applied to this state causally precede any future operations. Meanwhile, the `Unstable_st` represents the state resulting from the application of all log operations to the last stable state.

The need for the two states sheds light on the main compromise behind the ECRO approach and helps to explain the choice of nomenclature: the stable state is the most recent state in which all replicas agree, while unstable operations are tentatively applied to the unstable state by each replica, but may need to be rolled back and recalculated as new operations arrive.

Integrating remote operations When a new operation arrives, it is handled by `Effect`, which adds the operation to the directed graph as a vertex. This new operation is then compared, in the `addEdges` method, with every other operation in the graph, and commutativity is checked. If the operations are not commutative, two scenarios may arise: either the new operation is causally after the current operation, or it is concurrent with it. In the case of the former, a 'happens-before' edge is added. For the latter an 'arbitration' edge is added to the graph, following the arbitration order specified by the data type.

After all the edges and vertices are added to the graph, the operation can be immediately applied to the unstable state `Unstable_st` if it commutes with all operations or does not commute but is causally after them. If these conditions are not met, a rollback is needed, and for that, an adapted topological sort is performed to order the log operations taking into account the newly inserted operation, and the result is applied to the last stable state `Stable_st`, updating the unstable state `Unstable_st`.

Stabilizing operations When an operation becomes stable, the `Stabilize` method is invoked. It starts by verifying whether the ordered operations that precede the stabilized operation have also already stabilized. If so, all operations up to and including the received one can be deleted from the log and applied to the stable state because it is guaranteed that no upcoming operations will affect their order.

Querying the state When a query is made to the state of the replica, the method `Query` is invoked, returning the current unstable state, which is the most recent state available at a local replica but not necessarily consistent with the other replicas.

Algorithm 5 ECRO-like CRDT algorithm (adapted from [9]).

```
Require:
1: $Stable\_st : any$                                         # crdt stable state
2: $Unstable\_ops : G\langle O,E\rangle$                      # graph of unstable operations
3: $Unstable\_st : any$                                       # crdt unstable state
4: $Sorted\_ops : []$             # sorted ops calculated by the last topological sort

5: procedure Effect($op$)
6:     $O = O \cup op$
7:     $is\_safe$ = addEdges
8:     if $is\_safe$ then
9:         $Sorted\_ops = Sorted\_ops \cup op$
10:        $Unstable\_st$ = Apply($Unstable\_st, op$)
11:    else
12:        $Sorted\_ops$ = incTopologicalSort ($Sorted\_ops, op$)
13:        $Unstable\_st$ = Apply($Stable\_st, Sorted\_ops$)
14:    end if
15: end procedure

16: function addEdges
17:     $is\_safe$ = true
18:     for $v \in O \wedge v \neq op \wedge \neg$ Commutes($v$,$op$) do
19:         if isDescendant($v$,$op$) then            # checks if op is causally after op
20:             $E = E \cup E\langle v, hb, op\rangle$
21:         else if isConcurrent($v$,$op$) then       # checks if op is concurrent with op
22:             $is\_safe$ = false
23:             if ArbitrationOrder ($v$,$op$) then   # checks if v is ordered before op
24:                 $E = E \cup E\langle v, ao, op\rangle$
25:             else if ArbitrationOrder($op$,$v$) then  # checks if op is ordered before v
26:                 $E = E \cup E\langle op, ao, v\rangle$
27:             end if
28:         end if
29:         return $is\_safe$
30:     end for
31: end function

32: procedure Stabilize($op$)
33:     if $prefixStable(Stable\_ops, op)$ then  # checks if previously sorted operations are stable
34:         $E = E \setminus$ Outgoing($op$)
35:         $E = E \setminus$ Incoming($op$)
36:         $O = O \setminus op$
37:         $Sorted\_ops = Sorted\_ops \setminus op$
38:         $Stable\_st$ = Apply($Stable\_st, Sorted\_ops$[:index($op$)])   # apply new stable ops
39:     end if
40: end procedure

41: function Query
42:     return $Unstable\_st$                                 # returns unstable state
43: end function
```

5.2.3 Topological Sort

Algorithm 6 shows the implementation of our adapted topological sort which is based on Kahn's Algorithm [14] used to order log operations of the ECRO CRDT.

Incremental topological sort The `incTopologicalSort` function is recursively designed to perform an incremental topological sort, which improves efficiency by attempting to execute the actual `topologicalSort` function fewer times and on a subgraph of the unstable log, over a reduced number of vertices.

The function `incTopologicalSort` takes two arguments: `sorted_ops`, which is a list of sorted operations, and `op`, the operation to be incorporated into the sorted list.

The function operates by evaluating the leading element of the incoming list with the operation being added. If the new operation is causally after the leading operation, then the leading operation can be prepended to the result of a recursive call of the function without the leading element and with the new operation. The same happens if the new operation does not commute and respects the arbitration order with the leading element.

Otherwise, if the new operation does not commute and is ordered before all operations it is prepended to the list of sorted operations.

If none of these conditions are satisfied, the adapted topological sort is carried out with the remaining sorted operations.

Adapted topological sort Since our approach only maintains a directed graph, it requires a sorting process that is a modified version of a standard topological sort designed to work with cyclic graphs. Given that adding edges to the graph may create cycles, the process is adapted to manage such occurrences.

The process begins by checking for the existence of a minimum vertex. If no such vertex exists, it implies the presence of cycles within the graph. To resolve these cycles, the algorithm employs a deterministic approach by identifying and discarding an 'arbitration edge' with the minimum ID.

Once this edge is discarded, if none of the vertices emerges as the minimum, the procedure is repeated until all cycles are resolved. When a minimum vertex is found, it is added to the list of sorted ops, to be further returned when the number of ordered operations is the same as received ones.

It is important to note that this algorithm does not minimize the number of edges removed.

Efficiency The complexity of our algorithm can be defined as `O(|V|`^{2} `x |E|)`, where `V` are the vertices and `E` are the edges. One viable approach could have been to implement an approximation of the Feedback Arc Set algorithm, as studied in [10], where the complexity of such a solution is `O(|V| x |E|)`, and the edges are minimized. Given our primary focus beyond such optimizations, we decided to maintain our original implementation, as the potential minimization and efficiency gains are not crucial for our purposes, although having a significant impact on our benchmarks, discussed in Chapter 7.

While less efficient than randomly removing an 'arbitration order' edge, our approach provides greater predictability regarding the edges it removes. Other cycle elimination trade-offs could be explored, as discussed in [9].

Algorithm 6 Adapted Topological sort for ECRO-like algorithm.

```
function incTopologicalSort($sorted\_ops, op$)
    if length($sorted\_ops$) = 0 then
        return op
    end if
    $x = sorted\_ops[0]$
    if isDescendant($x, op$) $\wedge \neg$ Commutes($x, u$) then
        return $x$ : incTopologicalSort($sorted\_ops[1:], op$)
    else if ArbitrationOrder($x, op$) $\wedge \neg$ Commutes($x, u$) then
        return $x$ : incTopologicalSort($sorted\_ops[1:], op$)
    else if ArbitrationOrder($op, y$) $\wedge \neg$ Commutes($y, u$) forall $y \in sorted\_ops$ then
        return $op : sorted\_ops$
    end if
    return topologicalSort ($u \cup sorted\_ops$)
end function

function topologicalSort($ops$)
    $order = []$
    $removedVertices = []$
    $removedEdges = []$
    $inDegree = map[]$
    $edges = edges \cup e \wedge inDegree[e.Target] = inDegree[e.Target] + 1$ forall $e.Source \in ops \wedge$
$e.Target \in ops$
    while length($order$) $\neq$ length($ops$)  do
        for $vertex, degree \in inDegree$ do
            if $degree = 0 \wedge vertex \notin removedVertices \wedge vertex < minVertex$ then
                $minVertex = vertex$
            end if
        end for
        if $vertex = \perp$ then
            for $edge \in edges$ do
                if $edge \notin removedEdges \wedge edge.Type = \text{ao} \wedge edge.Id < minEdge.Id$ then
                    $minEdge = edge$
                end if
            end for
            $removedEdges = removedEdges \cup minEdge$
            $Removed\_edges = Removed\_edges \cup minEdge$
            continue
        end if
        $order = order : minVertex$
        $removedVertices = removedVertices \cup minVertex$
    end while
    return $order$
end function
```

5.2.4 Divergences from the original approach

The original approach of ECROs always upholds a Directed Acyclic Graph (DAG). In their approach, an addition of an edge always leads to a check of whether it leads to a cycle in the graph. If it does, an 'arbitration order' edge is discarded to break the cycle, and that choice is propagated across all replicas to guarantee that all replicas remove the same edges, to guarantee convergence. While this simple decision privileges the algorithm's efficiency, it does not directly fit the standard pure op-based framework, as it requires additional coordination and a tailored adaptation of the framework to propagate removed edges across replicas. In order to stay within the classical pure op-based CRDT framework, we have opted for a slightly different design.

Many other approaches are discussed and left open-ended in the original paper [9]. Their choice is arbitrary and, as in our implementation, it does not ensure the minimization of edge removals (e.g., two replicas may remove different edges to solve the same cycle). With this in mind, it becomes clear that the ECRO approach only guarantees causal dependencies. Preserving the arbitration order is a "best effort" approach.

Another difference of our approach is that, for simpler implementation, we assume that the data type comes annotated with commutativity information, while the original ECRO paper uses program analysis to automatically infer such information. Moreover, we consider that updates are total functions (that can be applied to any state), while they assume partial functions, thus supporting other kinds of edges to attempt to find sequences of non-failing operations. They also consider and identify cases where synchronization among replicas is necessary, introducing locks.

5.3 Semidirect CRDT

A particularly different approach to constructing CRDTs, greatly inspired by OT, is put forward by the Semidirect Product proposed in [24].

The Semidirect Product introduces a distinction in the way operations are handled. Its key insight is to systematize the distributed effect of operations by "repairing" them with previously received concurrent operations. Unlike OT, which considers algorithms for repairing general operations, the Semidirect Product can be seen as a greatly simplified form of OT that only considers two classes of operations, which we will name `A` and `B`, such that all concurrent operations in each class commute. Updates from class `B` are higher than updates from class `A` in the arbitration order. Consequently, updates from `A` are repaired with updates from `B`, and not vice versa.

The main practical difference of the Semidirect towards ECROs is that it leverages the data type semantics to repair operations in a way that ensures convergence without rollback. Naturally, it does so by imposing a more rigid structure on the underlying data type.

5.3.1 Data Type Interface

The data type interface of the Semidirect CRDT, specified in Listing 5.2 is also an extension of the commutative data type interface, designed to give additional information about updates. In addition to the `Apply` method, it also incorporates the `IsB` method to establish what operations belong to class `B`. The `RepairRight` method accepts two operations and returns the second operation repaired accordingly to the first.

Concrete examples of the instantiation of these methods are presented in the following Chapter 6.

```go
type SemidirectDataTypeI interface {

   // Apply operations to a given state.
   Apply(state any, operations []communication.Operation) any

   // Check if is a class B operation
   IsB(op communication.Operation) bool

   // Repair op2 to the right op1
   RepairRight(op1 communication.Operation, op2 communication.Operation)
    communication.Operation
}
```

Listing 5.2: Data type interface of ECRO CRDT.

5.3.2 Repair right operation

The core principle of the Semidirect Product involves defining an arbitration order between the two classes, such as "add-wins" or "rem-wins". The data type always needs to be equipped with a new “repair right” operation.

For two classes of updates A and B, and $a < b$ with $a \in A \wedge b \in B$, and where ";" means the composition of operations, the operation, read as ”repairs right” and denoted by $\triangleright$, must satisfy the following property:

$$a\,;\,b = b\,;\,b \triangleright a$$

For instance, `add` x $\triangleright$ `rem` x implies that the `rem` x operation is repaired with the knowledge that `add` x was previously applied. Thus the `rem` x is repaired to the right of the `add` x operation. If two class `B` operations `add` x t and `add` x t_1, were previously applied, the `rem` x would be repaired in right-associative fashion: `add` x t_1 $\triangleright$ `(add` x t $\triangleright$ `rem` x`)`. These scenarios are typical for an "Add-wins" Set, where `add` operations repair `rem` operations.

It is also assumed that the effect of two concurrent class `B` operations is independent of their order (they are commutative). Moreover, given two concurrent operations $(b_1 \,||\, b_2)$ the ”repair right” operation must satisfy a second property:

$$b_1 \triangleright b_2 \triangleright a = b_2 \triangleright b_1 \triangleright a$$

Finally, if two class `B` operations can be compressed into just one $(b_1;b_2 = b_{12})$, meaning that it would have the same effect as applying both operations, the ”repair right” operation satisfies an optional third property, that can be used for log compaction:

$$b_{12} \triangleright a = b_2 \triangleright b_1 \triangleright a$$

We chose not to implement this property as it is not straightforward to incorporate compaction properties into a generic algorithm, and it's usually simpler to implement this on a case-by-case basis for specific CRDTs.

5.3.3 Algorithm

As shown in Algorithm 7, this generic CRDT features an operation log, `B_ops`, with class `B` operations, represented by a set of operations because of the existence of the second property mentioned before. The state `State` represents the state resulting from applying all repaired operations to the state.

Integrating remote operations When a new operation is handled by `Effect`, the repair process is conducted by invoking the generic CRDT's `repairRight` method, which iterates over the class `B` operations that have been received to date and applies the necessary repairs in line with the `RepairRight` method as defined by the data type. Once the operation of class `A` has been repaired, it is applied to the state. If the operation is classified as a class `B` operation, as determined by the data type using the `isB()` method, the operation is then added to the log of `B` operations. This log serves as a reference for repairing future `A` operations.

Stabilizing operations Here, when an update of the class `B` of operations is stabilized, the `Stabilize` method simply removes the operation from the log.

Querying the state The state is retrieved by invoking the `Query` method, which simply returns the current state.

Algorithm 7 Semidirect CRDT algorithm (adapted from [24]).

```
Require:
 1: $B\_ops : \{\}$                                     # set of class B operations
 2: $State : any$                                       # crdt state

 3: procedure Effect($op$)
 4:     $repOp$ = repairRight($op$)
 5:     $State$ = Apply($State, repOp$)
 6:     if isB($op$) then                               # checks if op is from class B
 7:         $B\_ops = B\_ops \cup op$
 8:     end if
 9: end procedure

10: procedure Stabilize($op$)
11:     if isB($op$) then
12:         $B\_ops = B\_ops \setminus op$
13:     end if
14: end procedure

15: function repairRight($op$)
16:     for $o \in B\_ops$ do
17:         if isConcurrent($o, op$) then
18:             $op$ = RepairRight($o, op$)             # data type repairs op to the right of o
19:         end if
20:     end for
21: end function

22: function Query
23:     return $State$                                  # returns state
24: end function
```

5.3.4 Scalability beyond two classes

The Semidirect CRDT does not naturally scale for more than two classes of updates. Extending this to a third set of operations requires a more complex set of repairs among all concurrent operations.

The original semidirect CRDT works as a combinator that takes two CRDTs, say containing operations of class A and B, and produces a new CRDT. The operations of this composite CRDT are not a mere union of A and B operations but rather B $\rhd$ A operations union with A and B operations.

Incorporating a third CRDT with class C operations into the composite CRDT, it is not sufficient to just have repairs where A < C and B < C. It would demand that all operations in class A (B $\rhd$ A and A operations) and operations in class B be repairable by class C operations, which leads to more complex requirements.

5.3.5 Divergences from the original approach

Our implementation of the Semidirect CRDT introduces some differences from the original concept.

The original proposal essentially accepts two CRDTs (whose concurrent operations are commutative), both sharing the same state, and produces a composite CRDT supporting the operations of both. We have adapted this concept to the scenario where we have a sequential data type and two classes of operations.

Their approach has to maintain some causal information itself, to be independent of the underlying CRDTs it is combining, and besides that, it assumes partial operations. In contrast, we rely on the causal information offered by the middleware and consider total operations.

A third key distinction is seen in the application of stabilization. Stabilization is introduced by them as an extension or optimization of the construction. In our implementation, stabilization is directly inherited from the middleware.

5.4 Continuous Semidirect

The Semidirect approach embodies a binary arbitration criterion, for example, prioritizing operations in a class `B` over operations in a class `A` by applying the previously shown "repair right" operation ($\triangleright$). This results in a discrete system with two distinct classes of operations, such that `A < B`. In particular, this imposes a very rigid partitioning of the order of updates. In this section, we explore and propose a more general semidirect construction that scales beyond two classes of updates.

5.4.1 Generalization Problem

Consider a naive attempt to generalize the Semidirect approach for three classes of updates A, B, and C, and updates a, b and c with $a \in A$, $b \in B$ and $c \in C$, in a scenario with two replicas rep_0 and rep_1.

Assuming that the expected order between the updates is $a;b;c$, and that we have the natural generalization of the Semidirect, both replicas converge when the updates are broadcasted, as represented in Figure 5.2a. However, in the case of Figure 5.2b, both replicas diverge.

$rep_0 : b\ ;\ c$

$rep_1 : a$

$rep_0 : b\ ;\ c\ ;\ c \triangleright b \triangleright a = b\ ;\ b \triangleright a\ ;\ c = a\ ;\ b\ ;\ c$

$rep_1 : a\ ;\ b\ ;\ c$

(a) Example 1

$rep_0 : c\ ;\ a$

$rep_1 : b$

$rep_0 : c\ ;\ a\ ;\ c \triangleright b$

$rep_1 : b\ ;\ c\ ;\ b \triangleright a$

(b) Example 2

Figure 5.2: Distributed behaviors of a generalization.

We argue that the second example illustrates the difficulty of composing semidirect CRDTs. Due to the repair process, the behavior of the CRDT is not an ordering of the updates issued by the users (as was the case, for example, in ECROs). However, this behavior also occurs in the original semidirect model, with only two classes, resulting in less intuitive distributed behavior.

Divergence insights The divergence occurs precisely when we have causal updates that violate the arbitration order (as demonstrated in 5.2b). When this does not occur (as seen in 5.2a), the CRDT converges and corresponds to an ordering of the issued updates.

Even if we assume that all operations in a class `C` can repair all operations in a class `B`, and all operations in a class `B` can repair all operations in class `A` (ensuring that "`A`s repaired by `B`s are repairable by `C`s" as in 5.2a), we do not have guarantees of convergence (as shown in 5.2b). Again, this happens when we have causal dependencies contradicting the arbitration order.

5.4.2 Generalization Approach

Our proposal in this section is, therefore, to derive a general CRDT construction that follows the arbitration order. On one side, we consider that causally dependent updates always respect the arbitration order. Some CRDTs, such as the RGA, whose study motivated this construction, already satisfy this restriction. We believe that this restriction is not mandatory, and later conjecture that it can be relaxed in future work (Chapter 8.2). On the other side, this restriction enables a simple semidirect construction that extends beyond two classes of updates. In fact, it is no longer necessary to focus on disjoint classes of updates, but we can simply rely on the notion that updates are repaired across the continuum offered by a total arbitration order.

When such an arbitration order is chosen, applying operations following the causal order does not disrupt the arbitration order and vice versa. This allows maintaining the log of operations always ordered, which guarantees that operations arriving at a replica are repaired by an ordered sequence of operations. Therefore, given two concurrent updates ($u_1 \mid\mid u_2$) where $u_1 \leq u_2$, the following properties need to be met:

$$u_1 \; ; \; u_2 = u_2 \; ; \; u_2 \rhd u_1$$

$$u_2 \rhd u_1 \leq u_2$$

This means that operations only repair lower operations regarding the defined arbitration order. Consequently, the operation resulting from this repair process will always be less than or equal to the operation with the higher standing in the arbitration order.

Our refined semidirect construction can be perceived as a specialized variant of Operational Transformation (OT), where updates are exclusively repaired in alignment with the arbitration order. Hence, this could be interpreted as a unidirectional repair model within the broader OT paradigm, which typically encapsulates the concept of bidirectional repair for operations.

5.4.3 Algorithm

As Algorithm 8 shows, the state is composed of an array of operations, `Ops`, and a state `State`, as in the previous generic Semidirect CRDT 5.3.

Integrating remote operations When an operation is received, the method `RepairRight` defined by the data type is used to repair the operation concerning all concurrent operations currently in the log `Ops`. The final operation, obtained from the repair, is then applied to the state, and the received operation is added to the operations' log `Ops` in an order that respects the arbitration defined by the data type in the `ArbitrationOrder` method. This insertion is performed by iterating from the end of the array towards the beginning. As the likelihood of locating the position for the new operation is higher at the end of the array, this method enhances efficiency.

Stabilizing operations When a message is stabilized, the `Stabilize` method is invoked and proceeds to remove that operation from the operations' log `Ops`, given that the prefix (all operations preceding it respecting the arbitration order) of the operation in question is also stable.

Querying the state The state is retrieved by invoking the `Query` method, which simply returns the current state.

Algorithm 8 Continuous Semidirect algorithm.

```
Require:
 1: $Ops : [\,]$                                              # log of operations
 2: $State : any$                                             # crdt state

 3: procedure Effect($op$)
 4:     $repOp =$ repairRight($op$)
 5:     $State =$ Apply($State, repOp$)
 6:     for $i, o \in Ops$ do                  # iterates starting at the end of the array
 7:         if ArbitrationOrder($o, op$) then
 8:             insert($Ops, i, op$)                          # insert op at index i
 9:         end if
10:     end for
11: end procedure

12: procedure Stabilize($op$)
13:     if prefixStable($op$) then
14:         $Ops = Ops \setminus op$
15:     end if
16: end procedure

17: function repairRight($op$)
18:     for $o \in Ops$ do
19:         if isConcurrent($o, op$) then
20:             $op =$ RepairRight($o, op$)     # data type repairs op to the right of o
21:         end if
22:     end for
23: end function

24: function Query
25:     return $State$                                        # returns state
26: end function
```

5.5 Continuous Semidirect and ECRO

The construction from the previous section will only help explain part of the RGA, namely concerning `inserts`. In this section, we explore how to extend an existing semidirect CRDT with new operations that cannot necessarily be repaired. This construction was again motivated by the RGA, and arose from the study on how it handles `deletes`. Our rationale will be to show that we can construct a CRDT that combines the Continuous Semidirect logic (Section 5.4) for base repairable updates with the ECRO logic (Section 5.2) for additional general updates. This integration allows an elegant tradeoff between both approaches, and design semidirect CRDTs that are extensible without having to limit expressiveness. It is also worth noting that a probably more intuitive approach would be to extend ECROs to support a “repairable“ property in addition to commutativity, and repair operations when possible instead of rolling back. We argue that such approach, however, would lead inevitably to the same design, as the semidirect approach only works if all updates are repairable; a more dynamic setting where only certain concurrent operations were repairable would not ensure convergence, since the order in which concurrent operations were executed in different replicas would affect if they were repaired or rolled back.

The general intuition for this construction is that the data type updates are separated into two classes of `repairable/A` or `non-repairable/B`, where `A` updates always come before in the arbitration order, which is what happens, for instance, on RGA, considering `inserts` as operations of the class `A` and `deletes` as operations of class `B`. This entails that the construction shall guarantee that class `A` operations will always be applied before concurrent class `B` operations. The most intriguing detail is what shall happen when a `B` operation is causally before an `A` operation: we cannot exceptionally treat such `A` operation as part of the ECRO, since it could repair or be repaired by other `A` operations.

5.5.1 Repair Left operation

In order to guarantee consistency and preserve causality, we introduce a new “repair left” operation between class `A` and class `B` operations, denoted by $\lhd$. Given an $a \in A$ and $b \in B$, where b happens-before a, the operation, which can be read as ”repairs left” is defined by:

$$b \,;\, a = a \lhd b \,;\, b$$

For instance, `add` x $\lhd$ `rem` x implies that the `add` x operation is repaired with the knowledge that `rem` x will be applied right after. Thus, the `add` x is repaired to the left of the `rem` x operation. If two operations `rem` x t and `rem` x t_1, were previously applied, the `add` x would be repaired in left-associative fashion: `(add` x $\lhd$ `rem` x t`)` $\lhd$ `rem` x t_1.

We will notice later that, for examples like the RGA, the two styles of repair are not necessarily associative, meaning that we assume that:

$$(a_1 \rhd a_2) \lhd b \neq a_1 \rhd (a_2 \lhd b)$$

This means that the repair order matters, and consequently, in our construction, a ”repair left” operation ($\lhd$) will always need to precede a ”repair right” operation ($\rhd$).

5.5.2 Data Type Interface

Listing 5.3 shows the methods that a data type must implement to use this generic CRDT. It includes the `Apply`, `isB` and `RepairRight` methods to enable semidirect to handle the operations specified in `isB`. All of these methods have the same function as in the previous semidirect CRDT 5.3.

Besides that, the `ArbitrationOrder` method is also needed to provide the semidirect information on how to maintain its operations ordered in the log. This method takes two operations and returns a boolean indicating whether the first is smaller or equal to the second. The `Commutes` method from ECROs is also needed with the same function and should take into account operations from both classes `A` and `B`.

Finally, the behavior of the ”repair left” operation should be specified in the `RepairLeft` method.

```
type SemidirectECRODataI interface {

   // Apply operations to a given state.
   Apply(state any, operations []communication.Operation) any

   // Check if two operations are ordered
   ArbitrationOrder(op1 communication.Operation, op2 communication.Operation)  bool

        // Check if two operations commute
   Commutes(op1 communication.Operation, op2 communication.Operation) bool

        // Check if is a class B operation
   isB() bool

   // Class A operations repair class A operations
   RepairRight(op1 communication.Operation, op2 communication.Operation, state any)
     communication.Operation

   // Class B operations repair Class A operations
   RepairLeft(op1 communication.Operation, op2 communication.Operation)
    communication.Operation
}
```

Listing 5.3: Data type interface of Semidirect ECRO CRDT.

5.5.3 Algorithm

This approach has two states, a stable state `Stable_st` maintained by the Continuous Semidirect's logic and an unstable state `Unstable_st` maintained by the ECRO's logic.

The ECRO logic has the same behavior as in the generic ECRO CRDT approach when a class `B` operation arrives: it applies the operation instantly or does a rollback, if the operation does not commute or does not have only causal relations with the previous operations in the log.

The behavior of the Continuous Semidirect CRDT is also the same. The operation it handles is repaired to the right regarding the `A` operations in the log. The key part is in the ”repair left” operation, which repairs class `A` operations regarding class `B` operations in the log, before sending them to the semidirect's logic.

The Algorithm 9 shows the complete generic CRDT, which includes the Continuous Semidirect and ECRO logics.

Integrating remote operations When a received operation is handled by `Effect`, if it is a class `B` operation, the ECRO's `Effect` behavior is performed: the operation is added to the graph of `B` operations `B_ops`, and edges are added accordingly to the causal and order relations with the `B` operations previously added to the log. The operation is applied to the unstable state `Unstable_st` if it is considered safe. Otherwise, the unstable state is rolled back by ordering the log `B_ops` and applying the log operations into the stable state, resulting in a new updated unstable state `Unstable_st`.

In the event that a class `A` operation is received, the method `RepairLeft` defined by the data type is used to perform a ”repair left” on the operation concerning all causally precedent `B` operations currently in the log `B_ops`, which will later be applied to the state. The result operation is then handled by the Continuous Semidirect's `Effect` behavior: the operation is repaired to the right by the `RepairRight` method, which repairs the operation concerning all concurrent `A` operations currently in the log `A_ops`, and is subsequently added to the log respecting the arbitration order defined by the data type on `ArbitrationOrder`.

Finally, the received operation is repaired to the right by the `RepairRight` method to ensure consistency between the stable and unstable states. This operation is then directly applied to the unstable state if it is concurrent and commutative with all of the `B` operations in the log `B_ops`. Otherwise, a rollback is needed: our incremental topological sort is conducted on the `B` operations, and they are subsequently applied to the stable state, updating the unstable state.

This rollback verification is needed because the unstable state has to be consistent with the following rules:

1. Concurrent and non-commutative operations enforce a rollback, since when `B` and `A` operations are concurrent, operation `A` has to be applied before `B` (which was previously applied to the unstable state. Thus, a ”repair left” of `A` in the unstable state is not possible).

2. Non-concurrent operations enforce a rollback because, as previously mentioned, we assume that the associative property between repair operations does not exist. Thus, when `B` is causally before `A`, the ”repair left” operation must be performed before the ”repair right” operation.

Stabilizing operations When the `Stabilize` method receives a stabilized operation, the first course of action is to check the received operation is a `B` operation, if so, it is applied to the stable state, updating it. Otherwise, it is an `A` operation, and the continuous semidirect's `Stabilize` behavior is performed: the operation is removed from the log `A_ops` if its prefix of `A` operations is stable.

Querying the state The state is retrieved by invoking the `Query` method, which simply returns the current unstable state. As we adopt the compromise inherent in ECROs where doing a rollback when conflicting operations arrive, doing a query has no intrinsic complexity.

Algorithm 9 Continuous Semidirect + ECRO CRDT algorithm.

```
Require:
1: $B\_ops : [\,]$                                      # log of B operations
2: $A\_ops : G\langle O, E\rangle$                      # log of A operations
3: $Unstable\_st : any$                                 # unstable state updated by ecro
4: $Stable\_st : any$                                   # stable state updated by semidirect

5: procedure Effect($op$)
6:     if isB($op$) then                                # checks if op is a class B operation
7:         ecro.Effect(($op, \emptyset$))               # performs ecro's Effect behavior
8:         return
9:     end if
10:    $repLeftOp =$ repairLeft($op$)
11:    contSemidirect.Effect($repLeftOp$)   # performs continuous semidirect's Effect behavior
12:    $repRightOp =$ repairRight($op$)
13:    if isConcurrentAndCommutative($repRightOp, B\_ops$) then
14:        $Unstable\_st =$ Apply($Unstable\_st, repRightOp$)
15:    else
16:        $Unstable\_st =$ Apply($Stable\_st$, incTopologicalSort($B\_ops$))
17:    end if
18: end procedure

19: procedure Stabilize($op$)
20:    if isB($op$) $\wedge$ prefixStable($op$) then
21:        $Stable\_st =$ Apply($Stable\_st, op$)
22:        $B\_ops = B\_ops \setminus op$
23:        return
24:    end if
25:    contSemidirect.Stabilize($op$)       # performs continuous semidirect's Stabilize behavior
26: end procedure

27: function repairRight($op$)
28:    for $o \in A\_ops$ do
29:        if isConcurrent($o, op$) then
30:            $op =$ RepairRight($o, op$)              # data type repairs op
31:        end if
32:    end for
33: end function

34: function repairLeft($op$)
35:    for $o \in B\_ops$ do
36:        if isDescendant($o, op$) then
37:            $op =$ RepairLeft($o, op$)               # data type repairs causality of op
38:        end if
39:    end for
40: end function

41: function Query
42:    return $Unstable\_st$                            # returns unstable state
43: end function
```

Chapter 6

Concrete CRDT Instantiations

This Chapter shows how some data types that we have implemented can be instantiated on top of the Commutative, ECRO-like, Semidirect, and SemidirectECRO generic CRDT constructions introduced in the previous Chapter 5. This involves specifying the functions of the interfaces of the generic CRDT that a data type uses.

Implementing diverse data types provides practical insights into the complexities involved in building upon each generic CRDT, while also understanding the key distinctions and trade-offs among the various methodologies.

6.1 Notations

To specify how data types are implemented, we use certain notations to aid readability and comprehension.

Methods are typically designated with their names followed by arguments enclosed in square brackets. For instance, `inc` and `[add, v]` represent a function without arguments and with arguments, respectively. When denoting elements in a list, the notation `m[j]` is employed, where `j` represents the index of the required element.

Slicing of lists is expressed using the `o[x:]` or `o[:x]` notations, denoting a cut from the list starting at index `x` until the end, or from the beginning of the list until index `x`, exclusive.

Causality precedence is denoted using $\prec$. Thus $\mathtt{a} \prec \mathtt{b}$ means that `a` precedes/happened-before `b` in causal order. Lastly, the `nil` value is denoted by $\perp$.

6.2 Commutative Data Types

This section presents the implemented commutative data types using the commutative generic CRDT defined in 5.1. The implemented examples of commutative CRDTs include both sequential types, whose operations are naturally commutative, and classic CRDT designs specifically engineered to ensure operation commutativity.

6.2.1 PNCounter

The implementation of this data type is trivial, as its operations are commutative. This data type only implements the method `Apply`, defined in Figure 6.1, of the generic CRDT interface.

$$
\begin{aligned}
& st : 0 \\
\text{Apply}(\text{inc}, st) &= st + 1 \\
\text{Apply}(\text{dec}, st) &= st - 1 \\
\text{Query}(elem, st) &= st
\end{aligned}
$$

Figure 6.1: Specification of PNCounter data type.

This method simply increments or decrements the state, which is an integer, when an `inc` or `dec` operation is received, respectively. Figure 6.1 also shows how the `Query` is implemented, which only returns the state of the data type.

6.2.2 ”Add-wins” Set

The ”Add-wins” Set is a classic CRDT. By default, the addition and removal of elements are not commutative operations on sets, leading to several possible distributed designs; the ”add-wins” semantics is one such design that guarantees commutative effects. Therefore, to distinguish from data types with sequentially commutative operations, we directly instantiate the general CRDT interface.

Data type modifications One of the modifications in the ”Add-wins” Set data type involves maintaining the state `st` as a set, with each element `e` forming a pair of a value and a timestamp. Figure 6.2 illustrates the representation of the state and its elements. It also outlines the `Effect` method: the `add` operation simply adds an element to the set, whereas a `rem` operation removes an element from the set, given that the timestamp `t'` of the element to be removed is causally prior to the timestamp `t` of the removal operation. This mechanism ensures that values added and removed concurrently are not discarded from the state.

$$
\begin{aligned}
& st : \{\} \\
& e : (v,t) \\
\text{Effect}([\text{add}, v, t], st) &= st \cup (v,t) \\
\text{Effect}([\text{rem}, v, t], st) &= st \setminus (v',t') \textbf{ with } v' = v \land t' \prec t \\
\text{Stabilize}([\text{add}, v, t], st) &= st \setminus (v',t') \land st \cup (v', \perp) \textbf{ with } (v',t') \in st \land t' = t \\
\text{Query}(elems, st) &= \{v \textbf{ with } (v,t) \in st\ \}
\end{aligned}
$$

Figure 6.2: Specification of "Add-wins" Set data type.

Keeping a set of values along with their timestamps may gradually escalate memory usage. As illustrated in Figure 6.2, once an `add` operation has stabilized, the timestamp linked to that operation's value can be pruned from the set. The `Query` method simply returns a set of values present in the state without its timestamps.

6.2.3 Replicated Growable Array (RGA)

The RGA is another classic CRDT that models lists. Since `insert` and `delete` operations on lists are not directly commutative, the RGA also follows a specific design to ensure commutative effects. As before, we implement it directly on top of the general CRDT interface. In a sequential list, operations typically specify the positions where the insertion or deletion takes place. However, the index of a list element may change if other elements are inserted or deleted concurrently, leading to inconsistencies.

Data type modifications To increase the independence among list operations, RGA assigns a unique identifier to each element of the list, which is assumed to be unique and ordered in line with causality. In our implementation, this id is represented by a tuple that uses the timestamp (vector clock) and an origin id, both arriving as metadata of the operation from the middleware. To compare ids, an `ID()` function combines the sum of all values of the vector clock and the origin id of the message. Consequently, if the sum is equal between updates from different replicas, they are ordered by their replicas' ids.

Besides that, operations use the timestamp (vector clock) as reference t_{ref} to insert the new element with a given timestamp t after an existing element of the list or at the head of the list if no timestamp is specified.

Insert operations The behavior of `Effect` is shown in Figure 6.3. When it receives an `insert` operation, it uses the `find` method to locate the element with t_{ref} in the state, and the `shift` method handle concurrent updates by shifting the index where the value will be inserted depending on the values concurrently inserted referencing the same element. This method ensures that if two concurrent insertions occur, the latter `insert` will occur on the left of the former if it has a higher timestamp `t`.

Delete operations A `delete` operation also uses t_{ref} to refer to the timestamp of the element in the list to be deleted. However, it is not safe to permanently delete elements, as concurrent insertions may be using that element as a reference and consequently will not be able to locate the insertion position. To solve this, RGA uses tombstones, which instead of permanently deleting the elements, tombstone them by putting their value to $\perp$. This approach solves the occurrence of concurrent inserts referencing this element. The state of this data type is an array of elements. Each element `e` is a tuple consisting of the timestamp of the element referenced t_{ref}, the element's value `v`, and its timestamp id.

Stabilization An element designated as a tombstone is only omitted in response to a query and is permanently deleted only when the delete operation has been stabilized. It is worth noting that in the classic RGA, the timestamps and necessary stabilization information is maintained within itself. However, in our approach, we delegate this responsibility to the middleware, retrieving the necessary information from it.

$$\begin{aligned}
& st : [\,] \\
& e : (t_{ref}, v, t) \\
\text{Effect}([\text{insert}, t_{ref}, v, t], st) &= \textbf{if } t_{ref} = \perp \textbf{ then} (t_{ref}, v, t) : st \\
&\quad \textbf{else if } \text{find}(t_{ref}, st) = -1 \textbf{ then } st \\
&\quad \textbf{else } st[:i] : (t_{ref}, v, t) : st[i:] \\
&\quad \textbf{with } i = \text{shift}(\text{find}(t_{ref}, st), t_{ref}, st) \\
\text{Effect}([\text{delete}, t_{ref}, t], st) &= st[j] = (t'_{ref}, \perp, t_{ref}) \textbf{ with } j = \text{find}(t_{ref}, st) \ \wedge j \neq -1 \\
\text{shift}(index, (i, t, st)) &= \textbf{if } i = |st| \vee \text{ID}(t') < \text{ID}(t) \textbf{ with } (t'_{ref}, v', t') = st[i] \textbf{ then } i \\
&\quad \textbf{else } \text{shift}(i+1, t) \\
\text{find}(index, (t, st)) &= \textbf{if } t' = t \textbf{ with } (t'_{ref}, v', t') \in st \textbf{ then } i \\
&\quad \textbf{else } -1 \\
\text{Stabilize}([\text{delete}, t], st) &= st \setminus st[j] \textbf{ with } j = \text{find}(t, st) \wedge j \neq -1 \\
\text{Query}(elems, st) &= \{v \textbf{ with } (t_{ref}, v, t) \in st \ \wedge \ v \neq \perp\}
\end{aligned}$$

Figure 6.3: Specification of commutative RGA data type.

6.3 ECRO-like Data Types

This section presents the data types implemented using the ECRO generic CRDT specified in 5.2. Here, classic CRDTs and real use-case scenarios are implemented.

6.3.1 ”Add-wins” Set

The classic ”Add-wins” Set CRDT was presented in the previous Section as a commutative data type. Using the ECRO generic CRDT, we can simply implement its sequential operations without worrying about their concurrent behavior, and obtain a similar CRDT.

Figure 6.4, specifies the implementation of this generic CRDT interface methods. `Apply` method receives `add` and `rem` operations, aiming to either append or delete an element from the state.

Since these two operations are not commutative, an arbitration order must be established between them in case of concurrency. As Figure 6.4 shows, in the context of an ”add-wins” scenario, the `Order` method defines that `remove` operations precede `add` operations. Meanwhile, the `Commutes` method states that operations of the same kind, as well as operations involving distinct values, always commute.

$$\begin{aligned} & st : \{\} \\ \text{Apply}([\text{add}, v, t], st) &= st \cup (v, t) \\ \text{Apply}([\text{rem}, v, t], st) &= st \setminus (v', t') \textbf{ with } v' = v \\ \text{Order}(res, (op, op')) &= op = \text{rem} \wedge op' = \text{add} \\ \text{Commutes}(res, ((op, v, t), (op', v', t'))) &= op = op' \vee v \neq v' \end{aligned}$$

Figure 6.4: Speficication of ECRO ”Add-wins” Set.

6.3.2 Replicated Growable Array (RGA)

The classic RGA CRDT was also implemented in the last Section as a commutative CRDT. It is also possible to sequentially implement its operations on top of the ECRO approach, leading to a similar CRDT.

As before, the state of this data type is an array of elements. Each element `e` is a tuple consisting of the timestamp of the element referenced t_{ref}, the element’s value `v`, and its timestamp `t`. A unique identifier is assigned to each element of the list, represented by a tuple, vector clock, and origin id. Operations use a reference timestamp t_{ref} to insert the new element with a given timestamp t after an existing element of the list or at the head of the list if no timestamp is specified.

Insert operations The behavior of `Effect` is shown in Figure 6.5. Similarly to the commutative implementation, when it receives an `insert` operation, it uses the `find` method to locate the element with t_{ref} in the state. However, the `shift` method previously used to order concurrent insertions referencing the same element is no longer needed.

Delete operations The behavior of a `delete` operation also uses t_{ref} to refer to the timestamp of the element in the list to be deleted. However, now it is possible to instantly delete the operation instead of tombstoning it like before.

Order of operations An arbitration order is established between all operations in case of concurrency, which does not happen in the classic RGA, where `insert` operations are always applied before `delete` operations. As Figure 6.5 shows, in this context, the `ArbitrationOrder` method specifies that operations with higher unique ids, calculated by combining the sum of the values of the vector clock with the origin id, take precedence over those with lower ids. Therefore, in the case of concurrent operations referencing the same element, those having higher ids are positioned closer to the reference element than the ones with lower ids.

Commutativity of operations The `Commutes` method states the commutative between operations. While `delete` operations remain consistently commutative with each other, similar to the original commutative RGA, there are subtle changes in the interplay of `delete` and `insert` operations. For instance, a `[delete,`t_{ref}`,`t`]` operation commutes with an `[insert,`t'_{ref}`,`v'`,`t'`]` operation only when the position being deleted t_{ref} is distinct from the insertion position `t'` and its reference position t'_{ref}. This is specified by the two last lines of the `Commutes` method in Figure 6.5.

This contrasts with the classic RGA, where an `insert` operation followed by a concurrent delete operation of the same position is allowed. This is because it is implicitly stated within the distributed behavior of `Effect` that `insert` operations always take precedence over the `delete` operations. In this approach, this precedence is enforced explicitly through the use of the arbitration order.

Moreover, the commutativity of two `insert` operations is dependent on all four involved positions being different, defined by the last line of `Commutes` specification. This is another change from the classic RGA, which allows sequential inserts at the same position. This difference exists because the distributed behavior of `Effect` explicitly considers shifts. Here, that behavior is captured by the arbitration order.

$$
\begin{aligned}
& st : [\,] \\
& e : (t_{ref}, v, t) \\
\text{Apply}([\text{insert}, t_{ref}, v, t], st) &= \textbf{if } t_{ref} = \perp \textbf{ then} (t_{ref}, v, t) : st \\
& \textbf{else if } \text{find}(t_{ref}, st) = -1 \textbf{ then } \; st \\
& \textbf{else } st[: i] : (t_{ref}, v, t) : st[i :] \\
& \textbf{with } i = \text{find}(t_{ref}, st) \\
\text{Apply}([\text{delete}, t_{ref}, t], st) &= \; st \setminus st[j] \textbf{ with } j = \text{find}(t_{ref}, st) \; \wedge j \neq -1 \\
\text{find}(index, (t, st)) &= \textbf{if } t' = t \textbf{ with } (t'_{ref}, v', t') \in st \textbf{ then } i \\
& \textbf{else } \; -1 \\
\text{Order}(res, ((op, t_{ref}, v, t), (op', t'_{ref}, v', t'))) &= \text{ID}(t) < \text{ID}(t') \\
\text{Commutes}(res, ((op, t_{ref}, v, t), (op', t'_{ref}, v', t'))) &= (op = delete \wedge op' = delete) \vee \\
(op = delete \wedge op' = insert \; \wedge \; t_{ref} \neq t'_{ref} \; \wedge \; t_{ref} \neq t' \;) \vee \\
(op = insert \wedge op' = delete \; \wedge \; t_{ref} \neq t'_{ref} \; \wedge \; t \neq t'_{ref} \;) \vee \\
(op = insert \wedge op' = insert \; \wedge \; t_{ref} \neq t'_{ref} \; \wedge \; t \neq t'_{ref} \; \wedge \; t_{ref} \neq t')
\end{aligned}
$$

Figure 6.5: Specification of ECRO RGA data type.

6.3.3 Social Network

To explore the practical application of generic CRDTs and how they can solve application invariants, as presented in Chapter 3, we modeled a data type presented in [12] that simulates a social network. In this model, two maps are employed to represent the state `st`, one representing friends and the other representing friend requests. The key of each map represents a user, while the associated value stands for the set of friends or requests linked to that user. Friendship relations are bidirectional, and if two users share a friendship, neither can appear in the other's friend request list.

As Figure 6.6 shows, the `Apply` method handles the `accept`, `breakup`, `request`, or `reject` operations of the data type. `accept` adds `from` user to the set of friends of `to` and vice versa and removes any request that might exist between these two users, `breakup` does the same but for removing a friendship. `request` adds `from` user to the set of requests of `to`, and `reject` removes a request that `to` previously made to `from`.

Order of operations Given that many of these operations are not commutative, an arbitration order is established using the generic CRDT `Order` method to reorder them in case of conflict. This hierarchical arrangement gives priority to `accept` operations over `breakup`, `requests` over `rejects`, and finally, `accepts` over `requests`, ensuring a predictable ordering of updates to the state. However, this ordering may vary depending on the application needs.

The suggested solution for this example discussed in 3.2.1 was to implement a set of requesters as a "remove-wins" set, which would remove a request at any concurrent addition. Here, we opt for a different "add-wins" policy, by prioritizing `accepts` over `requests`, while also prioritizing `requestss` over concurrent `rejects`.

Commutativity of operations To instruct the generic CRDT about which operations are commutative, this data type establishes that commutativity applies when operations are of the same type when the combined argument pair `(from, to)` differs between two operations, and if the pair of operations is `breakup` and `reject`, as they modify different data structures.

The specification of these two methods are detailed in Figure 6.6.

$$
\begin{aligned}
& st : \{friends : I \rightarrow \{\}, requesters : I \rightarrow \{\}\} \\
\text{Apply}([\text{accept}, from, to], st) &= \textbf{if } to \in requesters[from] \textbf{ then} \\
& friends[to] \cup from \wedge friends[from] \cup to \wedge \\
& requesters[to] \setminus from \wedge requesters[from] \setminus to \\
& \textbf{else } st \\
\text{Apply}([\text{breakup}, from, to], st) &= \textbf{if } from \in friends[to] \textbf{ then} \\
& friends[to] \setminus from \wedge friends[from] \setminus to \\
& \textbf{else } st \\
\text{Apply}([\text{request}, from, to], st) &= \textbf{if } from \notin friends[to] \wedge from \notin requesters[to] \textbf{ then} \\
& requesters[to] \cup from \\
& \textbf{else } st \\
\text{Apply}([\text{reject}, from, to], st) &= \textbf{if } from \in requesters[to] \textbf{ then} \\
& requesters[to] \setminus from \wedge requesters[from] \setminus to \\
& \textbf{else } st \\
\text{Order}(res, (op, op')) &= (op = \text{breakup} \wedge op' = \text{accept}) \vee \\
& (op = \text{reject} \wedge op' = \text{request}) \vee \\
& (op = \text{request} \wedge op' = \text{accept}) \vee \\
& (op = \text{reject} \wedge op' = \text{accept}) \\
\text{Commutes}(res, ((op, from, to), (op', from', to'))) &= (op = op') \vee \\
& (from \neq from' \wedge to = to') \vee \\
& (from = from' \wedge to \neq to') \vee \\
& (op = reject \wedge op' = breakup) \vee \\
& (op = breakup \wedge op' = reject) \vee
\end{aligned}
$$

Figure 6.6: Specification of ECRO Social Network.

6.4 Semidirect Data Types

This section presents the data types implemented using the Semidirect generic CRDT specified in 5.3. Here, only data types with classes of updates commutative in isolation can be implemented.

Thus, we analyze different implementations of the ”Add-wins” Set.

6.4.1 ”Add-wins” Set

As shown in [24], the ”Add-wins” Set can also be implemented as a semidirect CRDT. The implementation in this section explains such a design.

Almost ”Add-wins” Set We can start by defining a sequential set data type, and implement the expected element addition and removal operations. Since additions commute with additions, and removals with removals, we can then easily make this set an “almost ”add-wins” CRDT using our semidirect construction. However, the resulting CRDT would not model exactly the original ”add-wins” behavior due to a small detail: adds would potentially win over all concurrent removes, including adds that have been undone by other causally-after removes. This behavior is shown in Figure 6.7 where two replicas concurrently perform $\mathrm{add}(x)$; $\mathrm{rem}(x)$ operations, resulting in a state containing x, consequently breaking causality.

$$rep_0 : \{\}$$
$$rep_1 : \{\}$$

$$rep_0 : \mathrm{add}_0(x);\mathrm{rem}_0(x);\mathrm{add}_1(x);\mathrm{rem}_1(x) = \{x\}$$
$$rep_1 : \mathrm{add}_1(x);\mathrm{rem}_1(x);\mathrm{add}_0(x);\mathrm{rem}_0(x) = \{x\}$$

Figure 6.7: ”Add-wins” Set behavior breaking causality.

”Add-wins” Set To fix the previous behavior, the proposal in [24] is to slightly change the state to be a set of values tagged with a unique timestamp, and the remove operation to include an additional argument that is a set `T` of timestamps of concurrent `adds`. The semantics of the `rem` operation is then to remove the respective element for all pairs whose timestamp is not in `T`.

When a `rem` is first emitted, the set `T` is initially empty, but as `rem` operations are repaired by concurrent `add` operations, they add the timestamp of the add to `T`. The `Query` method will return the set of values `v` that occur at least once in the internal state.

In other words, in an ”add-wins” set CRDT, if there are concurrent `add` and `rem` operations for an element `v`, the `add` operation will "win" because the `rem` operation will only remove `add` messages that happened before it. If any `[add,v,t]` operations are concurrent with a `[rem,v,t′]` operation, they will not be removed, and `v` will continue in the set.

Figure 6.8 presents the behavior of the methods of the generic Semidirect interface. The `Apply` method handles `add` and `rem` operations, where `add` operations simply add elements to the state, and `rem` operations delete elements from the state if its value `v` is the same and its timestamp does not belong to the `T` set of the `rem` operation.

The crucial aspect of this data type's implementation lies in the `RepairRight` method. This method, as shown in the previous Chapter 5, is always called by iterating through the class `B` operations of the data type, defined by `RepairRight`. Thus, it will always be called with an `add` followed by a `rem` operation, meaning that `add` operations repair right ($\triangleright$) `rem` operations. This method, as shown in Figure 6.8, adds the concurrent timestamp `t` of `add` operation to the `rem` set `T`. This adjustment ensures that these `add` operations are preserved when a repaired `rem` operation is applied to the state.

$$
\begin{aligned}
& st : \{\} \\
& e : \{v, \{\}\} \\
\text{Apply}([\text{add}, v, t], st) &= st \cup (v, t) \\
\text{Apply}([\text{rem}, v, T, t], st) &= st \setminus (v', t') \textbf{ with } v' = v \land t' \notin T \\
\text{RepairRight}(res, ([\text{add}, v, t], [\text{rem}, v', T, t'])) &= \textbf{if } v = v' \textbf{ then } [\text{rem}, v',\ T \cup \{t\},\ t'] \\
\text{isB}(res, op) &= \textbf{if } op = \text{add} \textbf{ then } \text{true} \\
&\qquad \textbf{else } \text{false}
\end{aligned}
$$

Figure 6.8: Specification of Semidirect "Add-wins" Set data type.

6.5 Continuous Semidirect + ECRO Data Types

This section presents the RGA data type implemented using the Continuous Semidirect plus ECRO generic CRDT specified in 5.5. Here, we implement the RGA data type that satisfies the restrictions imposed by this generic construction.

6.5.1 Replicated Growable Array (RGA)

We now illustrate that it is possible to faithfully explain the behavior of the classic RGA CRDT as an instance of our combined semidirect plus ECRO construction. Intuitively, we will separate RGA operations into two classes: inserts will be handled by the semidirect logic, and deletes will be handled by the ECRO logic.

Figure 6.9 shows the `Apply` method, which is the same as in the ECRO implementation, where it is possible to sequentially define the operations.

Continuous semidirect inserts Here, the `insert` operations are the class `A` operations, thus, are handled by the continuous semidirect logic previously presented in 5.4.

The most important remark in this construction is that it is possible to define an arbitration order for insert operations such that causally-dependent inserts always respect the arbitration order, therefore allowing the use of our Continuous Semidirect.

Figure 6.9 shows how the arbitration order is defined for these operations. If the element of reference t_{ref} is equal in both operations, they are not commutative and are consequently ordered by their unique ids, which are calculated using `ID` function. This method, as mentioned before calculates the unique id by summing each replica value in the vector clock, and combines it with the origin id of the update which enables ordering updates consistently with causality, a property required by the Continuous Semidirect.

Insertion operations can also be non-commutative if they modify the same references, in that case, the ordering is done respecting causality, which is the same as ordering by their unique ids. If none of the previous conditions are met, it means that the operations are commutative and can be ordered by any rule, we defined `id < id'`, but it could be any other deterministic rule.

Figure 6.9 shows the implementation of the repair methods. The `RepairRight` method, used to repair insert operations, checks if the operations reference the same element. If the operations are out of order, `op'` is repaired to the right of `op`, by putting `op` as its reference. Otherwise, nothing is repaired. This is the same behavior as in the classic RGA, where `insert` operations are shifted accordingly to concurrent insertions for the same reference. Here, a sequential specification of operations is possible due to the responsibility of fixing concurrent insertions being held by the "repair right" operation.

Context-aware repair We introduce a new function called `pos` which checks if the referencing element of the operation being "repaired to the right" exists in the stable state.

This function is necessary since the first property introduced in Subsection 5.3.2 is not always respected with nonexistent positions.

Since referenced positions can be erased by previous `delete` operations, and given that `delete` operations are stabilized, this impacts whether causal insert operations are "repaired to the left" or not. This behavior subsequently influences the "repair to the right" actions of that `insert` operation, potentially leading to inconsistencies.

The classic RGA does not have this problem as it assumes that inserts are always after existent positions.

ECRO deletes In the classic RGA, `insert` operations occur before `delete` operations, and this ordering is the reason why tombstones are used. This is aligned with our approach of placing `insert` operations in the semidirect CRDT and `delete` operations in the ECRO, where `insert` operations are modified to be applied knowing that a `delete` operation will be applied right after. The complexity here is that, unlike pairs of `insert` operations, pairs of `insert` and `delete` operations do not necessarily adhere to the arbitration order (where `insert` < `delete`). This is the motivation for employing the `repairLeft` function in our implementation.

The `RepairLeft` method is used to repair `insert` operations that reference elements that have `delete` operations causally preceding it. Thus, it defines a behavior for insertions on nonexistent positions. Consequently, the reference element of the operation being repaired is changed to $\varnothing$, meaning an empty timestamp, which will result in discarding the insertion on the `Apply` method.

It is worth mentioning that as `delete` operations are commutative, they do not cause rollbacks in the ECRO logic of the Semidirect plus ECRO CRDT.

Finally, Figure 6.9 also shows the implementation of `Commutes` method. This method defines the commutative relations between both operations and is defined as in the ECRO generic CRDT 6.3.2.

$$
\begin{aligned}
& st : [\,] \\
& e : (t_{ref}, v, t) \\
\mathrm{Apply}(op, st) &= \mathrm{ECRO.Apply}(op, st) \\
\mathrm{ArbitrationOrder}(res, (op, t_{ref}, v, t), (op', t'_{ref}, v', t')) &= \textbf{if } op = op' = \mathrm{rem} \textbf{ then } \mathrm{true} \\
& \textbf{else if } t_{ref} = t'_{ref} \textbf{ then } \mathrm{ID}(t) < \mathrm{ID}(t') \\
& \textbf{else if } t = t'_{ref} \lor t_{ref} = t' \textbf{ then } \mathrm{ID}(t) < \mathrm{ID}(t') \\
& \quad \textbf{else } \mathrm{ID}(t) < \mathrm{ID}(t') \\
\mathrm{RepairRight}(res, (op, t_{ref}, v, t), (op', t'_{ref}, v', t'), st) &= \textbf{if } t_{ref} = \mathrm{pos}(t'_{ref}, st) \land t > t' \\
& \textbf{then } (op', t_{ref}, v', t') \\
& \textbf{else } (op', t'_{ref}, v', t') \\
\mathrm{RepairLeft}(res, ((op, t_{ref}, v, t), (op', t'_{ref}, v', t'))) &= \textbf{if } t_{ref} = t'_{ref} \textbf{ then } (op', \varnothing, v', t') \\
& \textbf{else } (op', t'_{ref}, v', t') \\
\mathrm{Commutes}(res, ((op, t_{ref}, v, t), (op', t'_{ref}, v', t'))) &= \mathrm{ECRO.Commutes}() \\
\mathrm{pos}(res, t, st) &= \textbf{if } t \in st \textbf{ then } t \\
& \textbf{else } \text{-}1 \\
\mathrm{isB}(res, op) &= \textbf{if } op = \mathrm{delete} \textbf{ then } \mathrm{true} \\
& \quad \textbf{else } \mathrm{false}
\end{aligned}
$$

Figure 6.9: Specification of Continuous Semidirect + ECRO RGA data type.

Chapter 7

Test and Evaluation

This chapter outlines the tests and evaluations conducted on the implemented data types. For this, we developed a framework that generates random sequences of updates for a given number of replicas, while concurrently profiling sequences of tests to gather benchmark data. This dual approach allows us to assess both the functionality and performance of the implemented data types.

7.1 Testing Framework

To simulate real use-case scenarios, we have built a framework to generate test vectors, that is, sequences of updates, respecting the following conditions:

1. Each replica sends messages with random intervals below two seconds.
2. Each message contains a randomly selected character from a pool of twenty possible options.
3. The operations performed are also random choices accordingly to the number of operations of the data type.

This ensures randomness and unpredictability in the process of generating messages. However, as previously mentioned, our implementation does not have a network layer. Therefore, to properly test the implemented causal delivery of the middleware presented in Chapter 4, we simulate delays in messages by maintaining an array of arrived messages in the middleware.

This system employs a randomness factor based on the number of messages to delay, which is predetermined in the test. Upon reaching this number, each time a new message arrives at the middleware, it is added to the array, and a random message from the array is selected. Once the middleware has received the expected number of messages, also predefined in the test, any remaining messages within the array are randomly retrieved for handling. This approach ensures a randomized acquirement of messages, to simulate the unpredictability of sending messages in a network.

7.1.1 Testing convergence

One of the key characteristics of CRDTs is their ability to ensure eventual consistency. Therefore, once all replicas have received the same messages, they are expected to achieve a consistent state among each other.

To validate this characteristic in the implemented data types, when all replicas have received a number of messages equal to the expected number outlined in the test, all system replicas are queried. Subsequently, we confirm the consistency of their states by comparing them.

7.1.2 Testing consistency

We also assess whether the final state of the data types align with a sequential execution of the updates performed by the replicas. To achieve this, we calculate all topological sorts with updates from the start of execution and verify if the final state corresponds to the result of applying one of the sequences obtained from the topological sort.

7.2 Benchmarking Framework

In order to evaluate the use of resources of our implemented generic CRDTs, we used the built-in Golang profiling tool, which uses statistical sampling to collect performance data about our CRDT programs running in the previous testing framework and provide insights into the overall performance of the program, as well as the resource consumption of individual functions.

It is worth noting that since Golang uses statistical sampling, the results are an approximation. Besides that, the benchmark tests are performed locally, as previously said in Chapter 4, using threads to represent replicas and Go channels to simulate communication among replicas. Consequently, these two factors may have influenced the results obtained.

We now evaluate the performance of different designs of the RGA data type, as it is our most complex example that covers all of the implemented generic CRDTs. In this scenario, the operation type is determined by a random selection between `insert` and `delete`.

7.2.1 Results

Figure 7.1 shows the results obtained for a system with 5 and 50 replicas, respectively. The dashed lines in the graph represent the memory usage per operation, while the solid lines depict the CPU time consumed per operation.

It is worth noting that our implementations are not optimized and may not necessarily reflect the best possible performance of each approach.

ECRO-like As it was earlier explained in Section 5.2 of Chapter 5, our ECRO generic construction was not implemented following the original approach of broadcasting discarded edges of the log, which comes with the tradeoff of reduced efficiency. In order to improve the accuracy of the comparison with the original approach we implemented a new version of this generic CRDT that follows the original idea presented in [9]. While this advancement significantly decreases the topological sort algorithm's complexity, which became `O(V+E)` (following Kahn's algorithm), the procedure for adding edges becomes more complex. Now, every time a new edge is introduced, it requires a cycle check through the execution of a Depth First Search (DFS), which also exhibits a complexity of `O(V+E)`. Despite these changes, this implementation still does not represent the efficiency of the original approach. However, it does exhibit considerable improvements over our initial version, which can be observed in the comparative benchmarks shown in Figure 7.2.

Comparing the second version of ECRO with the other approaches, as shown in Figure 7.1, it is evident that as the number of operations increases, both memory usage and CPU time correspondingly rise, due to the complexity of the topological sort and the cycle checking, since both `topologicalSort` and `addEdges` methods are the ones with more usage rates.

However, a system with five replicas does not experience the same rate of resource usage as a system with fifty replicas. This difference can be explained by the stabilization process, since operations achieve stability more quickly in a system with fewer replicas. This impacts the size of the log, which in turn affects the number of rollbacks as well as the time and memory consumed by functions. The management of vector clocks may also influence system performance, as the size of vector clocks scales with the number of replicas, increasing the size of the operations' metadata, and the time to manage vector clocks' information.

Continuous Semidirect + ECRO Similar to the ECRO approach, the benchmark results demonstrate improved performance in systems with fewer replicas, which can again be explained by the quicker stabilization of operations and small vector clocks in systems with a smaller number of replicas.

Both evaluations show that merging the Continuous Semidirect and ECRO approaches yields better performance than using the ECRO approach alone. It is important to note that our cycle checking and topological sort implementations may be dominating and inflating the benchmarks of the ECRO approach. However, the performance of the combination of Semidirect and ECRO would be better even with an efficient implementation. This is due to maintaining a smaller log, requiring less computation in conflict detection, and the reduced need to perform rollbacks.

A key aspect to consider is the significantly reduced number of rollbacks. This is because operations managed by Semidirect can be immediately applied, while only those handled by ECRO need a rollback.

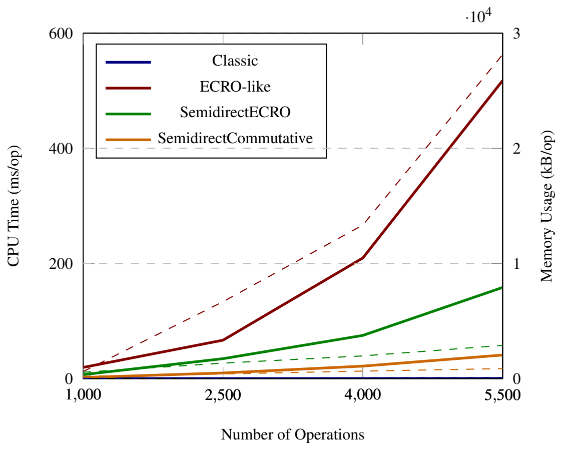
A potential strategy to further enhance the efficiency of this data type could involve implementing the log compaction feature of the Semidirect approach. This would further improve the log compaction carried out by the stabilization process of the Pure operation-based framework.

Continuous Semidirect + Commutative Given that all delete operations are commutative in RGA, we can eliminate the ECRO log and unstable state from our generic Continuous Semidirect and ECRO approach. In this context, the system does not maintain an unstable state and consequently, does not perform rollbacks. Instead, the responsibility of applying `delete` operations is assigned to the `Query` method, when it is requested, as in the classic implementation of RGA.

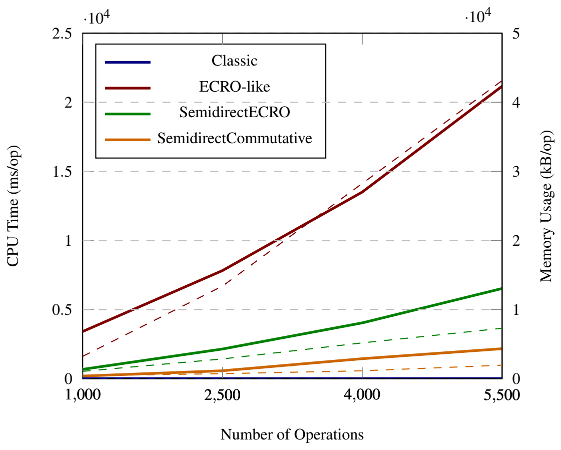
We studied this approach to understand the impact of maintaining an unstable state and having to perform rollbacks every time a conflict occurs. In the case of the RGA, the resource usage would only be made by insertions, removals, and operation repairs. As shown, in Figure 7.1, there is a notable difference between this approach, without rollbacks, and the previous approach, with rollbacks and an unstable state, where the former has better performance regarding resource usage. Consequently, this approach would be more advantageous for systems that do not execute a high volume of queries.

Classic Implementation It is noteworthy that the traditional implementation of RGA outperforms all other approaches. This is largely expected, as its design has been carefully tailored and optimized for attaining a specific behavior. This superior performance is attributed to the fact that it does not utilize an operation log, operations are applied immediately upon arrival, and its state and effect are optimized to store and make use of the extensively studied minimal necessary information to ensure convergence. Naturally, tailoring and optimizing a dedicated CRDT design for a specific data type, such as the RGA requires significant expertise and effort.

Discussion We find it very elegant that our Continuous Semidirect and ECRO variant achieves the same behavior as the classical RGA. In a way, the rules behind its design help explaining the complex distributed behavior of the RGA, and may hopefully guide future explorations of existing and novel CRDTs. It would also be interesting to further explore how close a more optimized version of our generic CRDT construction can get to the original RGA design. In particular, the Semidirect log could be compacted by ignoring ordering among commutative inserts. The challenge here would be to devise general rules on how to perform it automatically for the generic construction, or even more interesting, how to synthesize efficient code – equivalent to the classical design – from the generic construction.



(a) 5 replicas



(b) 50 replicas

Figure 7.1: Benchmarks for RGA data type.

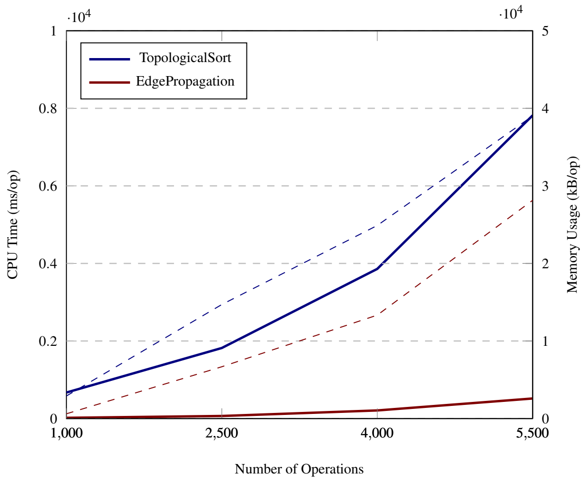


Figure 7.2: Benchmarks for different ECRO implementations in 5 replicas

Chapter 8

Conclusions and Future Work

8.1 Conclusion

The primary goal was to investigate and understand the challenges of building distributed applications with CRDTs while preserving their invariants. Throughout our investigation, we studied the obstacles of composing CRDTs regarding the sequential execution of the application. We explored different solutions to solve the problem, such as variations in the levels of consistency and the ability to repair invariants. A direction that emerged from our exploration was the possibility of building CRDTs that are consistent by design and built from the specification of a sequential data type.

We explored this approach by implementing some generic CRDT constructions capable of building a CRDT from sequential data types: Commutative, ECRO-like, Semidirect, Continuous Semidirect, and Continuous Semidirect plus ECRO.

Throughout the course of this work, we have delved into each one of the implemented CRDT constructions, developing a comprehensive understanding of the underlying principles and trade-offs of the existing approaches, such as the classic commutative CRDTs and the recent approach called ECROs. Besides the latter, we looked into another recent one called Semidirect, which we successfully extended, with some restrictions, to more than two classes of updates. Moreover, we combined it with the already existing approach ECROs. This combination effectively extends both approaches, inheriting the best intents from each: repairing operations and only doing state rollbacks when strictly necessary. Although it does come with some restrictions, it holds a promising concept in the domain of constructing CRDTs from sequential data types that are consistent by design.

8.2 Future Work

Our decision to present the final approach as a combination of Continuous Semidirect and ECRO-like with the restrictions presented in 5.5 is based on our discovery of a Semidirect subcase that can elucidate the behavior of the RGA, thereby enabling us to understand how it works. Our restriction that causality is consistent with arbitration order is motivated by the fact that it is sufficient to produce a CRDT that is consistent with applying the emitted updates in a sequence that respects causal order. However, this design is not a requirement for achieving convergence, which suggests that other more relaxed generalizations of the Semidirect Product may exist. This leaves room for potential extensions and relaxations of our assumptions to capture and explain the behavior of a larger suite of CRDTs.

In addition, this work represents a preliminary analysis where our construction was largely guided by the analysis of RGA. It would be interesting to implement other examples, such as explaining the behavior of the Social Network example presented earlier. In fact, explaining the behavior of the examples that use a composition of CRDTs presented in [12] using the Semidirect approach remains an open challenge. Another apparently simple but albeit challenging task would be to be able to express the behavior of the "add-wins" set without having to adapt the data type.

Lastly, we developed the generic CRDT constructions under the assumption that the distributed properties are provided by the users who implement a data type. As data types become more complex, understanding the properties of operations and determining repairs may not be straightforward and a significant burden. There are existing tools designed to automatically infer properties from sequential code, including commutativity such as [9], [18] and [27]. It would be useful to explore the extent to which we could infer repairs alongside other necessary properties using these tools.

Moving forward with this work, and to share our findings, a poster has been submitted and is planned to be presented at the Inforum conference.

References

[1] contributors . Automerge. URL https://automerge.github.io//.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, Nara, Japan, June 2016. IEEE. ISBN 978-1-5090-1483-5. doi: 10.1109/ICDCS.2016.98. URL http://ieeexplore.ieee.org/document/7536539/.

[3] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, November 2013. ISSN 2150-8097. doi: 10.14778/2732232.2732237. URL https://dl.acm.org/doi/10.14778/2732232.2732237.

[4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, Bordeaux France, April 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741972. URL https://dl.acm.org/doi/10.1145/2741948.2741972.

[5] Valter Balegas, Diogo Serra, Sergio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguica. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36, Montreal, QC, September 2015. IEEE. ISBN 978-1-4673-9302-7. doi: 10.1109/SRDS.2015.32. URL https://ieeexplore.ieee.org/document/7371565/.

[6] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proceedings of the VLDB Endowment*, 12(4):404–418, December 2018. ISSN 2150-8097. doi: 10.14778/3297753.3297760. URL https://dl.acm.org/doi/10.14778/3297753.3297760.

[7] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Pure Operation-Based Replicated Data Types, October 2017. URL http://arxiv.org/abs/1710.04469. arXiv:1710.04469 [cs].

[8] Jim Bauwens and Elisa Gonzalez Boix. Flec: a versatile programming framework for eventually consistent systems. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, Heraklion Greece, April 2020. ACM. ISBN 978-1-4503-7524-5. doi: 10.1145/3380787.3393685. URL https://dl.acm.org/doi/10.1145/3380787.3393685.

[9] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. ECROs: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. ISSN 2475-1421. doi: 10.1145/3485484. URL https://dl.acm.org/doi/10.1145/3485484.

[10] Camil Demetrescu and Irene Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, May 2003. ISSN 00200190. doi: 10.1016/S0020-0190(02)00491-X. URL https://linkinghub.elsevier.com/retrieve/pii/S002001900200491X.

[11] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://dl.acm.org/doi/10.1145/564585.564601.

[12] Alexey Gotsman and Hongseok Yang. Composite Replicated Data Types. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032, pages 585–609. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-46668-1 978-3-662-46669-8. doi: 10.1007/978-3-662-46669-8_24. URL http://link.springer.com/10.1007/978-3-662-46669-8_24. Series Title: Lecture Notes in Computer Science.

[13] Kevin Jahns. Yjs. URL https://docs.yjs.dev/.

[14] A B Kahn. Topological sorting of large networks.

[15] Gowtham Kaki, Swarn Priya, Kc Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, October 2019. ISSN 2475-1421. doi: 10.1145/3360580. URL https://dl.acm.org/doi/10.1145/3360580.

[16] Martin Kleppmann. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–6, Heraklion Greece, April 2020. ACM. ISBN 978-1-4503-7524-5. doi: 10.1145/3380787.3393677. URL https://dl.acm.org/doi/10.1145/3380787.3393677.

[17] Santosh Kumawat and Ajay Khunteta. A Survey on Operational Transformation Algorithms: Challenges, Issues and Achievements. *International Journal of Computer Applications*, 3(12):30–38, July 2010. ISSN 09758887. doi: 10.5120/787-1115. URL http://www.ijcaonline.org/volume3/number12/pxc3871115.pdf.

[18] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: synthesizing CRDTs with verified lifting. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1349–1377, October 2022. ISSN 2475-1421. doi: 10.1145/3563336. URL https://dl.acm.org/doi/10.1145/3563336.

[19] Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–28, July 2019. ISSN 2475-1421. doi: 10.1145/3341710. URL https://dl.acm.org/doi/10.1145/3341710.

[20] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *USENIX Association*, pages 265–278, 2012.

[21] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, Cascais Portugal, October 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043593. URL `https://dl.acm.org/doi/10.1145/2043556.2043593`.

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976, pages 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24549-7 978-3-642-24550-3. doi: 10.1007/978-3-642-24550-3_29. URL `http://link.springer.com/10.1007/978-3-642-24550-3_29`. Series Title: Lecture Notes in Computer Science.

[23] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400, Cascais Portugal, October 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043592. URL `https://dl.acm.org/doi/10.1145/2043556.2043592`.

[24] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based CRDTs with semidirect products. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–27, August 2020. ISSN 2475-1421. doi: 10.1145/3408976. URL `https://dl.acm.org/doi/10.1145/3408976`.

[25] Matthew Weidner, Heather Miller, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, and Christopher Meiklejohn. Collabs: Composable Collaborative Data Structures, December 2022. URL `http://arxiv.org/abs/2212.02618`. arXiv:2212.02618 [cs].

[26] Georges Younes. *Dynamic End-to-End Reliable Causal Delivery Middleware for Geo-Replicated Services*. PhD thesis, UMinho, January 2022.

[27] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. Type-Checking CRDT Convergence. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1365–1388, June 2023. ISSN 2475-1421. doi: 10.1145/3591276. URL `https://dl.acm.org/doi/10.1145/3591276`.