FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

ROSES: Renaming Operations for Scalable Eventually-Consistent Sets

Juliane Marubayashi

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Carlos Baquero

September 26, 2023

ROSES: Renaming Operations for Scalable Eventually-Consistent Sets

Juliane Marubayashi

Mestrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Prof. Pedro Souto
Arguente: Prof. João Leitão

September 26, 2023

Resumo

Criar sistemas altamente disponíveis, como o e-mail, num sistema distribuído, requer replicação de dados, já que um único ponto de falha pode comprometer a sua disponibilidade. Para isto, dispositivos necessitam replicar os seus dados na rede para que utilizadores possam ter acesso ao serviço mesmo em cenários de interrupção de conexão e outros tipos de falha.

Com o tempo, coerência eventual (CE) tornou-se cada vez mais popular entre empresas. Tratando-se de uma replicação otimista, CE possibilita a divergência de estados entre réplicas. No entanto, se nenhuma atualização for efetuada, e estado continuar a propagar-se, todos os acessos a um registo eventualmente irão convergir e retornar o mesmo valor. Em outras palavras, a rede irá chegar a um estado consistente.

Não menos importante, utilizar uma estratégia de replicação otimista arbitrária não necessariamente implica na convergência dos estados da rede. Nesse contexto, *Conflict-Free Replicated Data Type* (CRDT) foi formalmente definido em 2011 sendo utilizado em abundância em sistemas de jogos, apostas 'online', salas de conversação 'online' e outros tipos de sistemas. CRDT é uma estrutura de dados partilhada entre réplicas de um sistema distribuído, assegurando a resolução de conflitos por meio da junção de dois estados: o estado local e o que foi recebido.

CRDTs asseguram coerência eventual e possibilitam a resolução automática de conflitos entre estados. Porém, uma réplica necessita disseminar o seu estado por meio da rede para tornar isto possível. Assim como outras réplicas fazem o mesmo, o estado de uma réplica irá se tornar significativamente grande, já que uma entrada irá ser criada no estado local para cada estado recebido de uma réplica desconhecida.

Apesar deste problema, Almeida e Baquero propuseram mudanças na topologia da rede para que apenas um pequeno número de nós contenham informação das réplicas. Esta abordagem, no entanto, foi apenas formalizada para CRDTs baseados em contadores. Neste sentido, esta dissertação possui como intuito generalizar a solução apresentada por eles para CRDTs baseados em causalidade.

Abstract

Providing highly available systems, such as e-mail, in a distributed system requires data replication since a single point of failure might compromise availability. To this end, replicas must synchronize their data across the network to deliver the promised service despite network outages and general failures. There are many ways to provide consistency, and one is ensuring that each replica receives information in the same order, which is an expensive approach.

Throughout history, eventual consistency has made its way to popularization among enterprise systems. As a lazy replication strategy, EC allows replicas to diverge. Still, if no new updates are made, and the state keeps propagating, all accesses to a specific resource will eventually return the same value. In other words, the network will cost-effectively reach a shared state.

Nevertheless, an arbitrary lazy replication strategy does not entail a common state in the network: convergence might never happen. In this context, the Conflict-Free Replicated Data Type (CRDT) was defined in 2011 and is widely used in games, online gambling, online chats, and other systems. CRDTs are a data structure shared among replicas in a distributed system, assuring the resolution of conflicts by "merging" two states: the local and the received one.

The CRDTs ensure eventual consistency and automatically resolves possible conflicts between states. However, a replica must disseminate its state through the network to make it possible. As other replicas do the same, the state of a peer will become significantly large since a new entry in the local state will be created for every state received from an unknown peer.

Regarding this problem, Almeida and Baquero proposed changing the network topology so that only a small number of nodes contain the information of the replicas for CRDT counters. This dissertation contributes to their previous work by applying the developed protocol for other causal based CRDTs and implementing it in Rust language.

Acknowledgements

I want to express my sincere gratitude to my supervisor, Professor Baquero, for his invaluable guidance, support, and stimulating discussions throughout my academic journey. Professor Baquero not only provided me with insightful ideas but also motivated me to understand more about distributed systems. I am truly fortunate to have had such an exceptional supervisor, and I sincerely appreciate his contributions to my success.

To my boyfriend, Franciso Andrade, who patiently heard and supported me. Thank you for being by my side and for making me a better person. I also would like to thank my friends, especially Diana Freitas and Diogo Samuel Fernandes, for the laughs, random insights and for making academic life more colorful.

I also would like to thank my brother, who always stood up for me and supported me. And also my parents, who always gave their best to give me a better education. A special thanks to my mother, who advised me to study abroad and took care of me in my worst moments. Being a mother is hard, and I'll never be able to express how grateful I am.

Juliane Marubayashi

"Father laughed, which upset Bruno even more; there was nothing that made him more angry than when a grown-up laughed at him for not knowing something, especially when he was trying to find out the answer by asking questions"

John Boyne, The Boy in the Striped Pajamas

Contents

List of Figures

List of Algorithms

Abreviaturas e Símbolos

CRDT	Conflict Free Replicated Data Types
SEC	Strong Eventual Consistency
δ-CRDT	Delta-Based Conflict Free Replicated Data Types
AWOR-SET	Add-Wins OR-Set
GCounter	Grow-only Counter

Chapter 1

Introduction

This chapter introduces the problem at hand and outlines its motivation, context, and general goals. Section 1 provides the context of the work, Section 1.1 details the motivation behind the study, and Section 1.3 elaborates on the problem and enumerates the goals. The document structure is described in Section 1.4

1.1 Context

An essential feature of modern distributed systems is consistency across the network. A naive way of ensuring this condition would be forcing all the players in a network to receive the information in the same order. Unfortunately, this strong consistency model is unsuitable for a large-scale distributed system as it sacrifices performance and availability in favor of consistency [39].

Eventual consistency [40] promises to be a better solution for high availability in large-scale distributed systems against strong consistency. It allows the presence of inconsistencies and affirms that if no new updates are performed in a network, all the nodes will eventually return the same value. However, the difficulty of resolving conflicts is the main drawback of this approach since the system must anticipate all kinds of conflicts between divergent states and fix the possible output. In this context, the Conflict-Free Replicated Data Types (CRDTs) [32] were formally defined in 2011 to provide a non-ad-hoc solution for ensuring the convergence of propagated states in a network.

The idea relies upon a node abdicating consistency to allow users to apply operations (e.g., add, delete, update) in a scenario of network partitioning or delay. CRDT-based networks also

tolerate message loss, reordering and duplication as the procedures are idempotent. The state will eventually converge to a correct value as messages are propagated through the network.

Nowadays, CRDTs play a significant role in the industry, such as online game systems [3], online gambling platforms [4], and text editors [2]. It provides a systematic way to ensure that if no more updates are made in the network, the whole system will eventually contain the same information (i.e., eventual consistency).

1.2 Motivation

Low latency and available systems are prerequisites for successful digital-driven business [16]. Datacenter systems such as Facebook support billions of users, and the "always connected and anywhere" trend became a rule with the use of mobiles.

Regarding performance, businesses have strong investments to ensure latency within milliseconds. As users prefer fast systems [5], for an enterprise to keep growing, it needs to ensure that the system is scalable. Otherwise, it might compromise the quality of the product or lose to the competition.

The adoption of distributed architectures has risen in both technical and socio-technical systems, such as Open Source Software Development, due to its benefits of scalability, flexibility, and improved responsiveness. However, there is still room for improvement in this field, as these benefits do not come without challenges [30].

As the CAP theorem [24] explains, it's not possible to assure consistency, availability, and partition tolerance at the same time. As explained in the previous section, CRDTs approach this paradigm by promoting an abstraction for eventual consistency and conflict resolution. Although powerful, CRDTs aren't a *silver bullet*: a problem still to be solved is scalability.

1.3 Problem

Consider a well-known social network with more than 1 billion users. If the system were to be implemented using CRDTs, each device would have to store in a map the *id* of every device in the network. In other words, a mobile phone would have to keep billions of entries in its state, to support a simple "like" feature. For this reason, many systems do not use client-side CRDTs, but server-side only, as it has fewer nodes and is prepared to support a huge amount of data.

With the absence of CRDTs, on the client-side, the problem of unreliable network communication comes into discussion again. Even with more reliable network protocols (e.g., TCP), the exactly-once constraint remains. As a message of increment acknowledgment fails to be delivered or the connection times out, there will never be certainty whether an increment message was delivered. This implies a permanent inconsistent state across the network. There are solutions to solve the inconsistency, but as the CAP theorem [24] describes, this would sacrifice availability.

1.4 General Goals

This research aims to enhance scalability in causal CRDTs (sets, registers) by generalizing an existing solution limited to specific CRDT types. We discuss the design and the theoretical implementation behind the proposed mechanism by first walking through the background and state-of-the-art and then exposing and discussing the solution.

This work outputs a reference code library. Using this library, we created a simulation that analyzes the memory consumption of our protocol against a specific CRDT: Add-Wins Observed Remove Set (AWORSet).

1.5 Document Structure

The dissertation is organized as follows:

- Chapter 2, **Background**, prepares the stage for the rest of this work and provides the reader with the necessary information to understand the research being conducted;
- Chapter 3, **Problem Statement**, exposes the open problems of the current state of the art and explains the dissertation's hypothesis. We also go through some research questions to guide the investigation process;
- Chapter 4, **State Of the Art**, reviews the current knowledge related to the research topic being studied, in particular, Handoff Counters;
- Chapter 5, **ROSES Protocol**, we briefly explain the protocol and then discuss the results obtained by the tests;
- Chapter 6, **Evaluation**, discusses tests created to improve correctness confidence on the protocol, and analysis made to compare its effectiveness against a typical CRDT;
- Chapter 7, **Conclusion**, highlights the conclusions of this work and the future work;

Chapter 2

Background

In the previous chapter, we introduced the thesis and the problem under study. This chapter reviews key concepts to understand this work. Section 2.1 overviews consistency and discusses the differences between strong and eventual consistency. Section 2.2 explains the CAP theorem and its relation with the research topic. Section 2.3 explains what are lattices and why they are important. Section 2.4 introduces State-Based CRDTs and their problems, whereas Section 2.5 introduces the Delta-Based CRDTs. At the end of this chapter, in Section 2.6, we make a brief summary of what was discussed.

2.1 An overview about consistency

Storing data on distinct nodes in the network allows for better reliability and availability in case of any network failure. When a node fails, for instance, data can still be recovered by requesting information from another node. This replica, however, must be up to date with the faulty one so that the information is consistent. By replicas, we mean nodes that store replicated information.

Consistency is reached when all the replicas have the same state. Therefore, when a client requests information from a node, the returned value should be the same regardless of which replica is processing the request.

Although this is a simple contract, consistency has many models. The most relevant models to understand this thesis are strong, eventual, and causal consistency.

2.1.1 Strong Consistency

Strong consistency [40], [1], enunciates that after an update, all the replicas in the network will report the same value if the client requests. In other words, the network is always consistent. It behaves as if it had a centralized server handling all the operations.

Although consistent, it sacrifices performance as it requires synchronization among replicas and adds a layer of latency. This results in higher response times, making it less suitable for applications with strict time constraints.

2.1.2 Eventual Consistency

Eventual consistency, also known as optimistic replication [34], allows temporary inconsistencies in a system, and if no new updates are made, all the nodes will eventually converge to the same state. There is no way to predict how long it will take to achieve a consistent state.

A distributed system, for instance, may allow a user to update their profile in one place, but it may take some time for that update to be replicated across all the nodes in the system. This makes it difficult to ensure correctness in the system, as it is hard to guarantee that the data is consistent across all nodes. As a result, eventual consistency is unsuitable for applications where correctness is paramount.

2.1.3 Causal Consistency

Causal consistency ensures that if an operation A happened before another operation B in another node, then B can only be observed after A [11]. This concept is crucial in delta-based CRDTs, as *join* operations can only occur if there are no missing causal dependencies when joining an incoming state.

It guarantees that the order of operations is preserved, allowing the system to be eventually consistent without sacrificing correctness. As a result, CRDTs can provide strong guarantees of data integrity and correctness, even when the system is distributed across several nodes.

2.2 CAP Theorem and SEC

2.2.1 CAP theorem

According to the CAP theorem [24], "It is impossible for a replicated web service to provide the following three guarantees: Consistency, Availability, and Partition-Tolerance. All three properties are desirable - and expected - from a real-world web service". However, different combinations of these three provisions can be achieved depending on the system's requirements. For example, a system that prioritizes availability will have different guarantees than one that prioritizes consistency over availability.

Many interpretations have been published in the academic society explaining what each of these guarantees precisely means [28]. In this dissertation, we are going to stick with the following definitions:

- **Availability** means that *"a system always provides a response to every request."* [25];
- **Consistency** means that *"all copies of data in the system appear the same to the outside observer at all times."* [22].
- **Partition Tolerance** means that *"a given system continues to operate even with data loss or system failure. A single node failure should not cause the entire system to collapse."* [17]

Partition tolerance in distributed systems is a must, which leaves developers to decide and analyze the trade-offs between Availability + Partition Tolerance (AP) and Consistency + Partition Tolerance (CP). However, as explained in Section 1.1, many modern systems (e.g., Cassandra [29] and Dynamo [20]) implement AP design prioritizing availability over consistency.

2.2.2 SEC

However, as Shapiro et al. [35] explain, one does not necessarily need to give up on all the consistency guarantees. It can opt for lower levels of consistency. It presented Strong Eventual Consistency (SEC) as a solution for this issue, as it provides a series of rules that guarantees convergency to a predictable state.

The most significant difference between SEC and EC (Eventual Consistency) is that states generated by EC are challenging to predict and reason about, as they may lead to an arbitrary state [6]. However, by defining a set of rules, predictability increases.

2.3 Join-semilattice

This section provides a brief overview about join-semilattices [10], a partially ordered set. Understanding join-semilattices is crucial for proper CRDT understandment.

If every two elements in a set have a non-empty least upper bound (LUB), then the set is a join-semilattice. In the context of CRDTs, the least upper bound results from a join operation $\sqcup$.

Definition 1 (join-semilattice) *If a poset* $(S, \leq)$ *is a join-semilattice then* $\{\forall x, y \in S \mid LUB(x, y) \neq \emptyset\}$

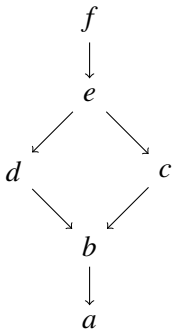


Figure 2.1: A join-semilattice example

In Figure 2.1 we show a Hasse Diagram [12] as an example of a join-semilattice. Every pair of elements have a least upper bound, some examples are:

- LUB(b,a) = b
- LUB(d,c) = e
- LUB(f,e) = f

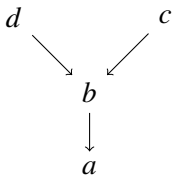


Figure 2.2: A Hasse Diagram that is not a join-semilattice

Figure 2.2 depicts a Hasse Diagram that is **not** a join-semilattice due to the absence of a least upper bound (LUB) between elements d and c, LUB(d,c) = $\emptyset$.

2.4 State-Based CRDTs

Section 1 introduces Conflict-Free Replicated Data Types (CRDTs) as a solution for achieving strong eventual consistency (SEC) in distributed systems. With CRDTs, nodes that receive the same set of operations will reach the same state deterministically [35].

This section focuses on the State-Based CRDT, a specific type of CRDT. We first present the basic concepts of this variant, then we dig into details, exploring the example of the GCounter CRDT.

According to Kleppmann et. al [27], State-Based CRDT is:

> *"A CRDT in which replicas synchronize by sending each other their entire state over the network; when one replica receives such a state from another replica, it uses a merge function to combine the two states. This merge function is defined in such a*

> *way that it is commutative (i.e., merge(a,b) = merge(b, a)), associative (i.e., merge(a, merge(b,c)) = merge(merge(a,b),c)), and idempotent (i.e., merge(a, a)=a).*"

The size of a CRDT causal context never diminishes. It keeps storing metadata to decide how to solve conflicts when merging states (Eq. 2.1, join operation). This allows the CRDT to establish a partial order $\sqsubseteq$ between states and identify the most recent modification on an element.

$$\sigma_i' = \sigma_i \sqcup \sigma_j \tag{2.1}$$

To perform updates, CRDTs use operations called **mutators**. Mutators (m) accept a state as input and produce a new state, as defined by equation (Eq. 2.2). To ensure the partial order relationship, the CRDT establishes rules to tag states. In this way, it knows σ precedes σ' as depicted in equation (Eq. 2.3).

$$\sigma' = m(\sigma) \tag{2.2}$$

$$\sigma \sqsubseteq \sigma' \tag{2.3}$$

2.4.1 Grow-Only Counter

Grow-Only Counter (GCounter) is a CRDT that implements a distributed counter. Single-threaded counters have two trivial operations: *increment* and *read*. Similarly, a GCounter provides the same services, which allows a user to increment a value and retrieve it for consulting.

Multi-threaded applications, however, must handle multiple updates simultaneously and share these updates with other replicas in the network to reach consistency. But how can we solve possible conflicts when a node shares a state with other replicas? And what is a conflict?

Consider two nodes, M and N. Node M incremented its local counter in 10, and N incremented in 3. When M shares its state with N, what should be the final state of N? One approach would sum the values 10 and 3, then N's state becomes 13. But if we follow this strategy and M shares its state once again, N's state will become 23 wrongly, and in the given conditions, N can't verify a duplicated message. We could also consider the final state as the maximum value between two nodes. However, this ignores inputs from one of the nodes, which might not be the best approach in some situations. Imagine that the nodes stored the number of times users clicked a button. If we select 10 as the final result, we will ignore the other 3 times the user incremented the counter. This is the type of conflict a GCounter solves.

GCounter implements its state as a map $id \mapsto value$, and a node is limited to increment a particular entry: the one with a key equals its ID. The current value of the counter is the sum of all values in the map. To solve the earlier question, GCounter defines a *join* operation to solve eventual conflicts between replicas. The *join* performs the union between the states, but the node selects the maximum value when both states share the same key.

$$
\begin{aligned}
&\textstyle\sum = \mathbb{I} \hookrightarrow \mathbb{N} \\
&\sigma_i^0 = \{\} \\
&\mathbf{inc}_i(m) = m\{i \mapsto m(i) + 1\} \\
&\mathbf{read}_i(m) = \sum_{i \in \mathbb{I}} m(i) \\
&m \sqcup m' = \{k \mapsto \max(m(k), m'(k) \mid k \in l\} \\
&\qquad\qquad \textbf{where } l = \mathrm{dom}(m) \cup \mathrm{dom}(m')
\end{aligned}
$$

Figure 2.3: GCounter definition [7]

To better visualize how the *join* operation works, consider Figure 2.4. A and B are two nodes in a network, and both start with an empty initial state. Initially, node A increments its local value in 5 and B in 6. Consequently, both replicas created a single entry in their state where the keys are equal to their ID (i.e., $A \mapsto 5$ and $B \mapsto 6$).

After that, A shares its state with B, which performs a *join*, equivalent to a union, as there are no conflicts between the states. Then, A increments its value in more 3 units creating $\sigma A'$, and B shares its state with A. Upon receiving $\sigma B'$, A notices that both $\sigma B'$ and $\sigma A'$ have a key in common A. This is a conflict. As a solution, node A assigns the maximum value between the two given (i.e., $m_A(A) \mapsto max(8,5)$). Thus, the A's final state becomes $\sigma A''$. Finally, when A shares its state back to B, the same strategy is applied, and the network reaches convergence.

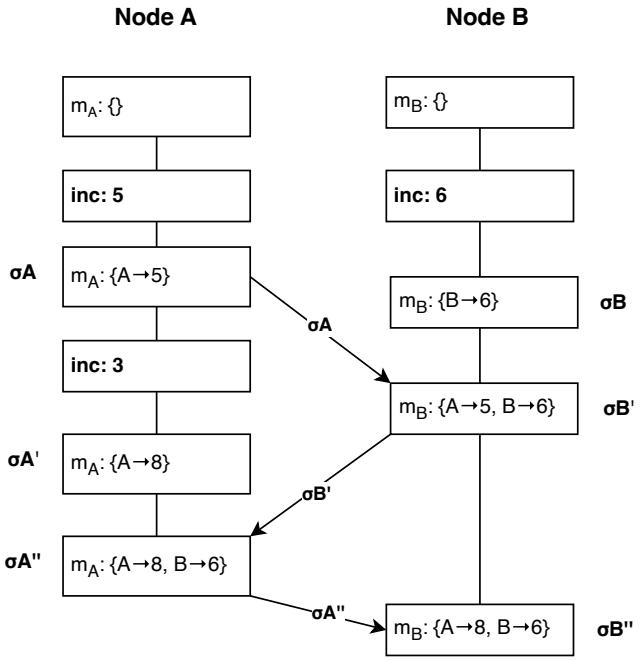


Figure 2.4: GCounter flow example

2.4.2 Discussion

State-Based Conflict-Free Replicated Data Types (CRDTs) provide robust data structures for distributed systems. However, performance issues may arise due to the internal state growth, which increases network overhead. In particular, the GCounter may experience this issue when the number of nodes in the network increases because the state's map size grows, making message transfer more costly. In other words, if a network has one million nodes, the internal state of each node would eventually reach one million entries. Therefore, sending a huge state through the network will increase the traffic flow and cause an overhead.

2.5 Delta-State CRDTs

Delta-State CRDT (Δ-CRDT) [7] is another kind of CRDT, where a node does not need to ship its entire state to propagate modifications. Like operation-based CRDTs [13], a Delta-State CRDT transmits a set of applied operations since the last exchange. This method reduces the size of the message the node shares.

To achieve this, Δ-CRDT have **delta-mutators** (m^δ). It takes a state σ as an argument and returns a smaller state, a **delta mutation** $m^\delta(\sigma)$. Like traditional mutators, defined in section 2.4, the merge between a delta mutation and a state creates a new state (Eq. 2.4) that is subsequent to the current one (Eq. 2.5).

$$\sigma' = \sigma \sqcup m^\delta(\sigma) \tag{2.4}$$

$$\sigma \sqsubseteq \sigma' \tag{2.5}$$

As well as in State-Based CRDTs, Δ-CRDTs do not require reliable network communication. The incremental updates and the *join* operation hold the same properties discussed in section 2.4: commutativity, associativity, and idempotency.

The next subsections will approach AWORSets as an example of Δ-CRDTs. We will first examine the concept of dots. Then we will explore AWORSets as a State-Based CRDT, followed by the modifications necessary to convert this CRDT into a Delta-Based CRDT.

2.5.1 Dots

CRDTs can distinguish the order of operations issued in a node. As using a global sequence number is not feasible (the replicas would have to coordinate at every write) CRDTs use a local sequence counter as a timestamp: (replica_id, sequence_value).

Dots are essential to solving conflicts between nodes due to some properties [38]:

- They uniquely identify operations in a replica;

- As the sequence number increases by one after every operation and dots are never deleted, it becomes easier to track missing operations.;

2.5.2 State-Based Add-Wins OR-Set

An Add-Win Observed-Removed Set (AWORSet) is a CRDT that solves conflicts between concurrent operations of add and remove over a set [9]. In other words, AWORSet implements a distributed set where a client can add and remove the elements that were previously added.

An AWORSet's state comprises a **causal context** (c) and a **set** (s) containing tagged elements. The causal context keeps track of the order of events and comprises dots. Initially, c is empty, and when adding an element, the node creates a new dot (replica_id, sequence_value + 1), where the sequence_value starts with 0. Yet, a replica can never delete its dots. Imagine that a client applied the following set of operations:

$$M = \{\text{add("x")}, \text{add("y")}, \text{rm("y")}\}$$

For each operation, the node will create a new dot, resulting in the following sequence of causal contexts:

$$\begin{aligned}
&\text{add("x")} \rightarrow c_A = \{(A,1)\} \\
&\text{add("y")} \rightarrow c_A = \{(A,1),(A,2)\} \\
&\text{rm("y")} \rightarrow c_A = \{(A,1),(A,2)\}
\end{aligned}$$

To add a new entry "y", the node retrieved the biggest sequence value associated with A (i.e., 1) and added the new entry with this value incremented by one: $(A,2)$. Notice that the causal context remained the same when the client removed "y", as a causal context never diminishes its size.

Adding elements modifies not only the causal context but also the set of elements. Each entry of s is a triple (replica_id, sequence_value, element) similar to a dot, but entries can be removed when deleting an element.

To better visualize it, consider the following sequence of actions over node A:

$$N = \{\text{add("a")}, \text{add("k")}, \text{add("k")}, \text{rm("k")}, \text{add("j")}\}$$

Node A obtains the following sequence of states when applying the operations one at a time:

$$\mathsf{add}(\text{"a"}): A = \{s = \{(A,\text{"a"},1)\}, c = \{(A,1\}\}$$
$$\mathsf{add}(\text{"k"}): A = \{s = \{(A,\text{"a"},1),(A,\text{"k"},2)\}, c = \{(A,1),(A,2)\}\}$$
$$\mathsf{add}(\text{"k"}): A = \{s = \{(A,\text{"a"},1),(A,\text{"k"},2),(A,\text{"k"},3)\}, c = \{(A,1),(A,2),(A,3)\}\}$$
$$\mathsf{rm}(\text{"k"}): A = \{s = \{(A,\text{"a"},1)\}, c = \{(A,1),(A,2),(A,3)\}\}$$
$$\mathsf{rm}(\text{"k"}): A = \{s = \{(A,\text{"a"},1),(A,\text{"j"},4)\}, c = \{(A,1),(A,2),(A,3),(A,4)\}\}$$

Until now, it was explained what happens when an element is added and removed from a node locally. Nevertheless, a merge/join function is necessary as distributed systems contain many nodes, and they must share states.

Consider Figure 2.5. It formally defines the state of an AWORSet, a tuple containing: the set of elements (s) and causal context (c). The node starts with these entries empty and formally defines the operations we discussed: add and rm.

The **merge** operations joins both sets of the local node to the received one (s' and c'). A node can merge two causal contexts by performing the union between the current causal context c and the one receiving c'.

$$\begin{aligned}
&\mathsf{AWORSet} = \mathscr{P}(\mathbb{I}\times\mathbb{N}\times\mathbb{E})\times\mathscr{P}(\mathbb{I}\times\mathbb{N})\\
&\bot = (s,c) = (\{\},\{\})\\
&\mathsf{add}_i(e,(s,c)) = (s\cup\{(i,n+1,e)\}, c\cup\{(i,n+1)\})\\
&\qquad\qquad \text{with } n = \mathsf{max}(\{k \mid (i,k)\in c\})\\
&\mathsf{rm}_i(e,(s,c)) = (s\setminus\{(j,n,e) \mid (j,n,e)\in s\}, c)\\
&\mathsf{elements}_i((s,c)) = \{e \mid (j,n,e)\in s\}\\
&(s,c)\sqcup(s',c') = ((s\cap s')\cup\{(i,n,e)\in s \mid (i,\ n)\notin c'\}\\
&\qquad\qquad \cup\{(i,n,e)\in s' \mid (i,n)\notin c\}, c\cup c')
\end{aligned}$$

Figure 2.5: State-Based AWORSet definition

Merging sets of elements is not so trivial since some elements can be removed. Consider Figure 2.6 as an example. The network in the example contains two nodes, A and B.

Initially, both nodes A and B have empty states. Then "y" and "z" are added to A, and "y" to B. After that, node A shares its state with B. When B receives A's state, it makes a union of its state and A's:

$$\begin{aligned}
B = \{&s = \{(B,1\text{"y"}),(A,1,\text{"y"}),(A,2,\text{"z"})\},\\
&c = \{(B,1),(A,1),(A,2)\}\}
\end{aligned}$$

Then B removes "y". The operation removes all the y elements from s, but the causal context remains untouched.

$$B'' = \{s = \{(A,2,\text{"z"})\}, c = \{(B,1),(A,1),(A,2)\}\}$$

When B sends its state to A, the destination node needs to solve conflicts. In this case, a simple union between states won't be enough to merge the states as elements that also belongs to A were modified by B.

Notice that when a tagged element is deleted, it can never be re-added to a node. The instance $(A,1,\text{"y"})$, for example, once belonged to A, but as the sequence number is continuously increasing, the node's sequence number will never be 1 again. Looking at the causal context, A knows an element tagged with $(A,1)$ was once part of B's state. But since it was no longer a member of s_B, A knows this element was deleted.

Following this logic, the final state of A are the common elements between A and $B(s_A \cap s_B)$, plus the elements not known by each other. A not-known element is at s_A, but not at c_B or s_B, but not at c_A.

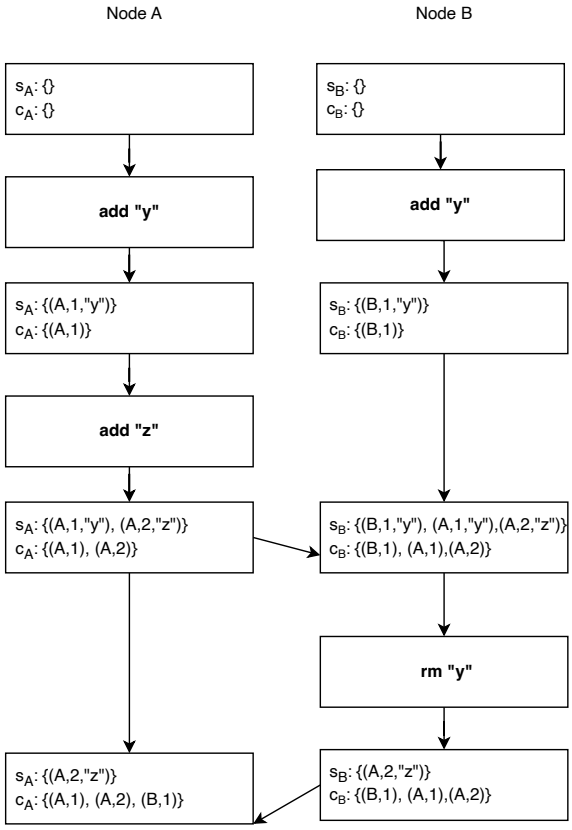


Figure 2.6: AWORSet State-Based example on merging operation

2.5.3 Delta-Based Add-Wins OR-Set

In this subsection, we present and discuss the differences between the AWORSet Delta-Based and State-Based. In the Delta-Based CRDTs, the operations generate a delta (δ), which contains the modifications applied to a CRDT. The add operation, for instance, produces a δ (Eq. 2.6), which, when joined to X, generates a new state X' (Eq. 2.7).

$$\delta_X \leftarrow add(X, n) \tag{2.6}$$

$$X' \leftarrow X \sqcup \delta_X \tag{2.7}$$

Suppose that node X has the following state:

$$X = \{s = \{(X,1,\text{"a"}),(X,2,\text{"b"}),(X,4,\text{"d"})\},$$
$$c = \{(X,1),(X,2),(X,3),(X,4)\}\}$$

When X adds a new element "e", add operation generates a $\delta_{X'}$ which contains its mutations.

$$\delta_{X'} = \{s = (X,5,\text{"e"}),\ c = \{(X,5)\}\}$$
$$X' = \delta_{X'} \sqcup X \ \Rightarrow$$
$$X' = \{s = \{(X,1,\text{"a"}),(X,2,\text{"b"}),(X,4,\text{"d"}),(X,5,\text{"e"})\},$$
$$c = \{(X,1),(X,2),(X,3),(X,4),(X,5)\}\}$$

When we merge $\delta_{X'}$ to X, we obtain X'. The logic behind generating a δ for a remove operation is similar. Suppose that we remove "b" from X'. The delta generated would be:

$$\delta_{X''} = \{s = \emptyset,\ c = \{(X,2)\}\}$$
$$X'' = \delta_{X''} \sqcup X' \ \Rightarrow$$
$$X' = \{s = \{(X,1,\text{"a"}),(X,4,\text{"d"}),(X,5,\text{"e"})\},$$
$$c = \{(X,1),(X,2),(X,3),(X,4),(X,5)\}\}$$

Likewise, when joining $\delta_{X''}$ to X' we would get X''.

In the formal definition, the functions add and rm now are adapted to return deltas, as Figure 2.7 shows.

$$
\begin{aligned}
&\mathsf{AWORSet} = \mathscr{P}(\mathbb{I} \times \mathbb{N} \times \mathbb{E}) \times \mathscr{P}(\mathbb{I} \times \mathbb{N}) \\
&\bot = (s,c) = (\{\},\{\}) \\
&\mathsf{add}_i^{\delta}(e,(s,c)) = (\{(i,n+1,e)\},\{(i,n+1)\}) \\
&\qquad\qquad \text{with } n = \mathsf{max}(\{k \mid (i,k) \in c\}) \\
&\mathsf{rm}_i^{\delta}(e,(s,c)) = (\{\},\{(j,n) \mid (j,n,e) \in s\}) \\
&\mathsf{elements}_i((s,c)) = \{e \mid (j,n,e) \in s\} \\
&(s,c) \sqcup (s',c') = ((s \cap s') \cup \{(i,n,e) \in s \mid (i,\ n) \notin c'\} \\
&\qquad\qquad \cup \{(i,n,e) \in s' \mid (i,n) \notin c\}, c \cup c')
\end{aligned}
$$

Figure 2.7: Delta-Based Aworset definition [9]

Deltas provide a solution for the network overhead of State-Based CRDTs. Instead of sending its entire state to its correspondent, a node can transmit only the modifications the replica has applied since the last transmission. When an operation generates a delta, the node merges it to a buffer and keeps merging deltas until it is delivered to another node. When the transmission is done, the buffer can be again set to null. This buffer is called Delta-Interval and will be discussed in the next section.

2.5.4 Causal Consistent Anti-Entropy Algorithm

While State-Based CRDTs guarantee causal consistency by disseminating their complete state across the network [15], Δ-CRDTs rely on an algorithm to provide this assurance. Before going through the algorithm, consider the definitions below:

Definition 2 (Delta-Mutator) *A single delta-mutation or a join of multiple delta-groups.*

Definition 3 (Delta-Interval ($\Delta_i^{a,b}$) [7]) *Is a delta-group produced by joining all the deltas in the range d_i^a, d_i^{a+1}, ..., d_i^b of a replica i.*

Algorithm 1 presents a version of the algorithm presented in [7] with modified syntax.

Algorithm 1 Anti-Entropy algorithm for causal consistency

```
1: input n : N                                   ▷ Set of neighbors' node id
2: durableState
3:     cd : K = {}                               ▷ A CRDT containing a state s
4:     c : u64 = 0                               ▷ Sequence number
5: end durableState
6:
7: volatileState
8:     D: map(u64 : K) = {}                      ▷ Map of deltas
9:     A: map(N : u64) ← {}                      ▷ Ack map
```

```
10: end volatileState
11:
12: function onReceiveDelta_{j,i}(d: K, seqNumber: u64)        ▷ i receives a delta d from j
13:     if d ⋢ cd.s then
14:         cd.s = join(cd.s, d)                                ▷ Update cd's state
15:         D.add(c, d)
16:         c = c + 1                                           ▷ Update curret node's sequence number
17:         send_{i,j}(ack, seqNumber)
18:     end if
19: end function
20:
21: function onReceiveAck_{j,i}(seqNumber: u64)                 ▷ i receives ack message from j
22:     A.add(j, max(A[j], seqNumber))
23: end function
24:
25: function onOperation(m^δ)
26:     d = m^δ(cd.s)                                           ▷ Apply mutation to state
27:     cd.s = join(cd.s, d)                                    ▷ Update crdt's state
28:     D.add(c, d)
29:     c = c + 1
30: end function
31:
32: function shipStateOrInterval                                ▷ Periodically called
33:     Declare minKey = 0
34:     d = { }
35:     minKey = min(D.keys())                                  ▷ Smallest key in D
36:     j = random(n)
37:     if D = { } ∨ D[minKey] > A[j] then
38:         d = cd.s                                            ▷ Send all the state
39:     else
40:         for ∀key ∈ D.keys() do                              ▷ join all the deltas between A[j] and c
41:             if A[j] ≤ key < c then
42:                 d = join(d, D[key])
43:             end if
44:         end for
45:     end if
46:     if A[j] < c then
47:         send_i, j(d, c)
48:     end if
49: end function
```

50: **function** GARBAGECOLLECT ▷ **Periodically called**
51: remove deltas acked by all neighbors
52: **end function**

Every node in the algorithm has a durable and volatile state. The durable state is regularly saved in a durable storage and consists of a CRDT (i.e., *cd*) and a sequence number denoted by *c*. The volatile state includes two maps: *D*, which holds sequence numbers as keys and deltas as values, and *A*, which stores acknowledged messages with node ids as keys and a sequence number as value. This sequence number is the largest index *b* from a delta-interval $\Delta^{a,b}$ that was acknowledged by a node *j*.

When an operation is performed, it generates a delta state by applying a delta-mutation to *cd*'s state. The *cd*'s state is then updated by merging it with the generated delta. The generated delta is then added to the *D* map with the current sequence number, and the sequence number is incremented.

When receiving a delta state with (*d*, *seqNumber*) as a message, node *i* first verifies if the received state is not already present in the state of the CRDT. If not, the delta *d* is joined to *cd*'s state. After that, *d* is computed as change. Thus, it is added to *D* associated with a sequence number so that it can be propagated to other nodes. Finally, *i* increments its counter and sends an acknowledge message to *j*.

Node *i* verifies if the received delta state with a message (d, seqNumber) is absent in the CRDT's state. If it's not, the delta *d* is combined with the CRDT's state. The change *d* is calculated and added to the *D* map with the sequence number for propagation to other nodes. Finally, node *i* increments its counter and sends an acknowledgment message to node *j*.

Upon receiving an acknowledgment message, node *i* records in *A* the sequence number of the latest delta received by node *j*. This information is crucial when delivering states to other nodes. The shipping function, which is periodically executed, selects a random node to receive the modifications recorded in *D*. Node *i* sends its complete state in two cases: when there are no modifications to deliver (i.e., *D* is empty) or when the oldest change (delta) in *D* is older than the last acknowledged sequence number by node *j*. If the two conditions mentioned previously are not met, all deltas with sequence numbers between $A[j]$ and *c* will be combined and sent to node *j*. With this algorithm, node *i* avoids sending information to node *j* that it may already have.

2.5.5 Discussion

Compared to State-Based CRDTs, δ-CRDTs are more efficient in transmitting messages and incorporating modifications to the current state. However, they do not ensure causal consistency like State-Based CRDTs. To address this, an anti-entropy mechanism was introduced.

Despite being introduced as a solution to the lack of causal consistency, the anti-entropy algorithm underperforms and is not an improvement over the state-based approach. This is due to its potential for spreading redundant states among replicas. A proposed improvement is discussed in [21].

2.6 Summary

This chapter makes an overview about consistency, explains the CAP and SEC theorems, and describes State-Based CRDTs and Delta-State CRDTs, which are key concepts to understand this thesis.

Section 2.1 overviews consistency, approaching its concept and describing other types of consistency, namely eventual and strong consistency. In short, eventual consistency allows temporary inconsistencies during the replication process, under the promise that if no updates are made, all copies of the data will eventually be the same. There is no guarantee of how long it will take for consistency to be achieved. This means that when a client updates a replica, it may not be immediately available for all the replicas. Still, it ensures that all copies of the updated data will eventually be consistent. By another hand, strong consistency is a model where all copies of data are always synchronized and consistent. When some data is updated, all the clients have immediate access to its most recent version.

Section 2.2 defines the CAP theorem and presents Strong Eventual Consistency, which allows temporary inconsistencies, which are eventually solved under a series of rules. In constrast to EC, SEC eases development and reasoning, and guarantees that the consistency will be reached.

Section 2.3 explains what are join-semilattices, so that the reader understands the domain of the CRDT's state.

Section 2.4 describes State-Based CRDTs, a structure used in distributed systems that ensures eventual convergency to a consistent state even in network partitions and delays. However, State-Based CRDTs are typically less efficient in terms of latency because they send the entire state to other replicas and have limited scalability, as the size of the data object being replicated may become too large.

Finally, Section 2.5 explains Delta-State CRDTs. In contrast to State-Based CRDTs, the δ-CRDTs do not propagate the entire state of the data over the network, instead they only send the changes (deltas) made to the data, reducing the amount of data that needs to be sent across the network. However, δ-CRDTs relies on an anti-entropy algorithm to ensure causal consistency. On this way, this data structure reduces significantly the latencies issues related to the State-Based CRDTs.

On the following chapter we discuss the current state of the art related to this dissertation.

Chapter 3

Problem Statement

3.1 Open problems

CRDTs face scalability challenges due to state growth proportional to the number of clients. This leads to a phenomenon called "id explosion," where each replica's state is mapped to a replica id.

While CRDTs ensure reliable communication, a lack of CRDTs on the client side demands exact message delivery. To achieve this, a consensus protocol like Paxos can be used, but at the cost of reducing availability, as replicas must wait for consensus to complete an interaction. This may result in financial consequences for enterprises that need to provide continuous services.

3.2 Main Hypothesis

We can summarize the hypothesis of this work as:

> *"There are isolated solutions for solving CRDT's scalability problems (1). There is at least one specific architecture and protocol design that can be generalized for causality-based CRDTs (2)."*

The first part of the hypothesis enunciates that some CRDTs already have a personalized solution for their scalability problem. Handoff Counters, for instance, solve the scalability problems of counter-based CRDTs.

The second part of the hypothesis states that some solutions can be generalized for other types of CRDTs, requiring formalizing this generalized design and its implementation.

3.3 Research Questions

To guide the investigation and inform the study design, the following research questions were identified:

RQ1: What are the isolated solutions for the scalability problems CRDTs, that the literature provides?

This research question is the starting point for the dissertation and validates part *(1)* of the hypothesis. It defines and establishes the current knowledge about the isolated solutions already created for CRDTs.

RQ2: How can the isolated solutions found be adapted to fit causal-based CRDTs?

Not every isolated solution for CRDTs' scalability problem can be generalized to fit causal-based CRDTs. Therefore, **RQ2** tries to prove component *(2)* of the hypothesis by asking "how" the solution can be generalized. If none of the references provided can be generalized, then it will be studied if the solution might fit a smaller set of CRDTs, rather than causal-based ones. In the worst scenario, a study explaining the adaptation problems will be developed.

RQ3: How does the generalization impact the use of CRDTs in terms of latency and memory occupation?

This step evaluates how the scalable mechanism impacts the performance of CRDTs in practice. In essence, it is necessary to measure each node's latency and memory occupation and compare the results with an implementation that does not use the scalable mechanism. Distinct scenarios will be tested and generated by varying the number of clients in each tier. In the end, we expect to verify performance improvements in a network using the developed mechanism.

3.4 Development methodology

The algorithm that will be presented in chapter 5 was developed in steps. Synthesizing all features simultaneously overcomplicates the process. Adapting a poorly developed algorithm is harder them building it in steps.

Suppose a student needs to pack cubic blocks in a backpack for a school project. As the student had little time, he threw all blocks into the sack, leaving some behind. In this situation, the student could try to organize them while still in the bag, but arranging blocks at the bottom is arduous.

An obvious solution would be adding the larger blocks in the bottom, side by side, and then piling up the remaining ones. Although the student might have some mistakes while pilling the cubics, these are easier to fix. Creating an algorithm is no different from the bag case.

Building all features at the same time is complex in terms of logic. We must build and modify an incomplete code until it fits our needs. However, as organizing blocks already in a bag is hard, organizing an incomplete code is also. Therefore, we idealized an order to implement the features,

so we call first build a solid basis and perform small adaptations while having some confidence in what was built before:

1. Addition;
2. Removal, Translations;
3. Caching.

3.5 Summary

The research question focuses on CRDTs' scalability challenges, which arise from growing state proportional to the number of clients, leading to "id explosion." The hypothesis suggests that some CRDTs have personalized solutions (e.g. Handoff Counters) and that some solutions can be generalized to causal-based CRDTs.

The research questions validate part 1 of the hypothesis by defining the current knowledge of isolated solutions and test part 2 by studying how to generalize the solution.

This evaluation assesses the effect of the scalable mechanism on CRDT performance by comparing the developed solution with an unmodified implementation in terms of latency and memory consumption and testing performance enhancements across various scenarios with different numbers of clients.

Chapter 4

State of the Art

4.1 Methodology

To analyze the current state of the art in the problem's domain, a specific methodology was employed. Given that the problem in this domain has not been extensively explored in literature, simple queries were utilized to retrieve a limited number of relevant documents.

4.1.1 Database

The search included ACM Digital Library, CRDT.tech, and IEEE Xplore, and did not have a specific file type restriction. As a result, the investigation covered magazines, journals, conference papers, and other file types. Additionally, web articles were also included in the search.

4.1.2 Queries

The following is a list of the key queries used in the study. Due to the lack of keyword search functionality on the CRDT.tech database, basic filters were applied. Given that the website only covers research related to CRDTs, it was not necessary to include CRDT-related keywords.

- **Q1**: "scalable" eventual consistency (databases: IEEE Xplore, ACM Digital Library)
- **Q2**: scale (database: crdt.tech)
- **Q3**: scalable (database: crdt.tech)

- **Q4**: (crdt OR "eventual consistency") AND (scale OR scalability) (databases: IEEE Xplore, ACM Digital Library)

4.2 KaZaA

KaZaA [26] was a P2P application to share files such as music and movies, created in 2001 and shut down in 2012. It was one of the largest systems deployed on the internet, with over 3 million active users sharing large quantities of data.

4.2.1 Topology

In contrast to overlay networks with random connections, KaZaA improves the system scalability by exploiting the network's heterogeneity. Peers do not have the same characteristics; they have different capacities, bandwidth, memory, and connection time.

Unlike Gnutella, KaZaA has two types of nodes, Ordinary Nodes (ONs) and Super Nodes (SNs). Users own both types, but SNs are more powerful peers and have other responsibilities.

KaZaA built a two-tier hierarchical system where SNs keep track of their children's content. Figure 4.1 shows the topology rules:

- ONs only connect to SNs;
- SNs link to ONs and some SNs;

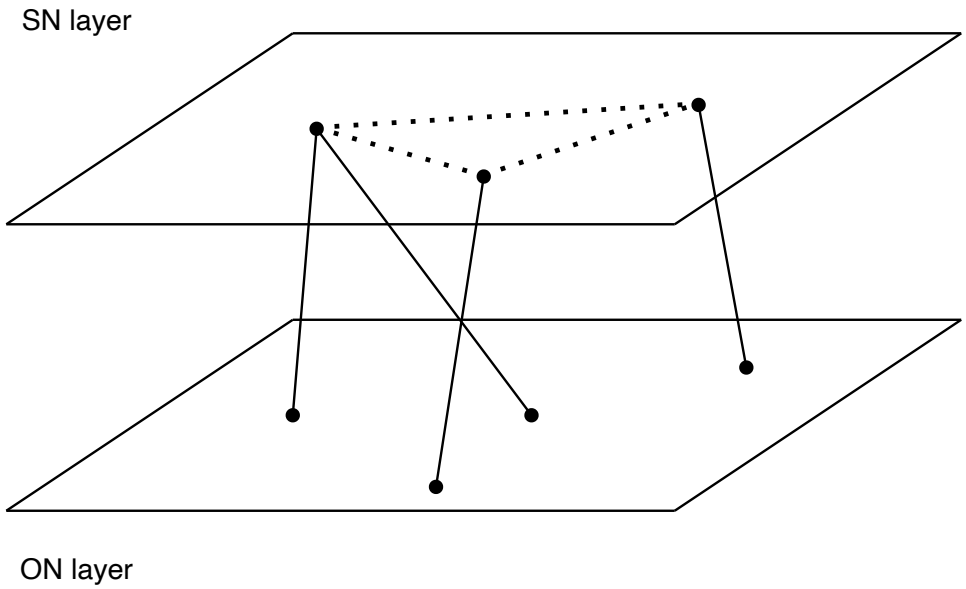


Figure 4.1: KaZaA topology

ONs keep a list of up to 200 SNs, and SNs have lists with thousands of SNs. These lists are exchanged between peers to keep them up-to-date. Based on these updates, the SNs also change their connections occasionally. This topology increases the variety of content to be searched.

4.2.2 Operations

When an ON uploads a file to the network, the SN responsible for the ON receives the file's metadata, which includes, for instance, the file name and size.

Then, when a user searches, the application creates a query, and sends it to the SN via TCP communication. Upon receiving the query, the SN lookup in its internal table for the files that match its parameters. For each match, the SN returns the node's IP storing the file and the corresponding metadata. The SN can also forward the query to other SNs that are connected to it.

Amongst the metadata, there is the **ContentHash**. The file descriptor contains the artist name, album name, and other data. This content is used for keyword matching in the databases and is part of a "recover system." The ContentHash uniquely identifies files. If a download of a file fails, the peer can search for the specific file in the network and continue the download without issuing a new query and request.

4.2.3 Discussion

Unlike Gnutella [18], KaZaA scaled to millions of users thanks to the hierarchical topology, which improves the traffic and memory of Ordinary Nodes.

Without server-like nodes, peers with scarce resources would have to handle high message traffic and store a list of nodes in the network. The scenario is similar to Gnutella, where the biggest part of the messages delivered in the network was PING and PONG messages. Later, the Gnutella protocol was improved to use a two-layer topology [19].

Other solutions could be used to prevent nodes from storing a big list of members in the network. However, it would force the implementation of slower and heavy search mechanisms. To improve scalability, it would be necessary to abdicate performance.

Therefore, although hierarchy makes a system less distributed, it leverages the characteristics of each node to improve performance. Nodes with more memory, for instance, are more suitable for storing more information and coordinating other nodes, for instance.

In this dissertation, we also take advantage of network heterogeneity and split the topology into layers to scale the system, as it will be explained in Section 5.

4.3 Handoff Counters

Almeida and Baquero described Handoff Counters [8] as a **CRDT-based counter mechanism** that meets eventual consistency criteria, works in unreliable networks, and is scalable. It addresses CRDT counters' scalability issue mentioned in Section 1.3 by creating a networking topology and hierarchy. Figure 4.2 shows an example of network topology.

In the network hierarchy, tier 0 nodes store information permanently, while nodes at lower tiers, such as tiers $n+1$, are subject to garbage collection. Data generated by clients at the lowest tier, referred to as leaf nodes, is transmitted up the hierarchy until it reaches tier 0 nodes.

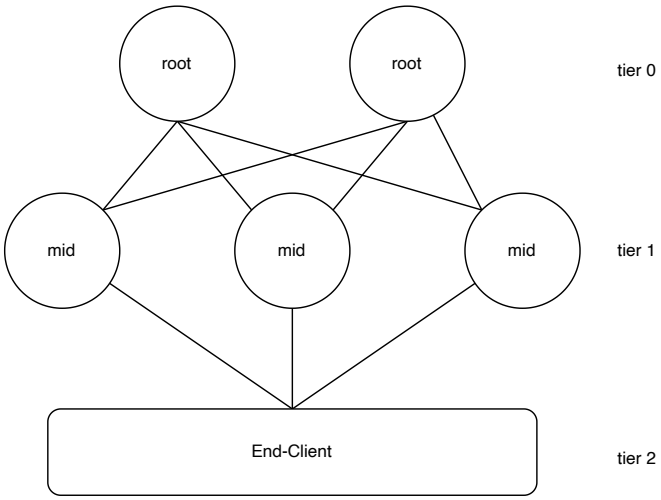


Figure 4.2: Handoff Counters network topology

The network topology must support eventual consistency by allowing node u to switch to another server in case of network partition during a message exchange, ensuring that a local increment is handed off. To achieve this, the topology must follow these guidelines:

- Each link is bidirectional;
- Tier 0 nodes is a sub connected network allowing information to be spread among servers;
- There is always a path from any node to a node at tier 0, which is natural, as the information needs to be properly stored;
- If two nodes have a common parent, then there is a link between them

4.3.1 Distributed Algorithms

When CRDTs are employed, the distributed algorithm is simplified as the complexity is shifted to the CRDT. For instance, a gossip algorithm can distribute information throughout the network, with each node periodically selecting a random neighbor to deliver its state, which the receiver

then merges with its own. The distributed algorithm must therefore have routines to merge states, retrieve the local counter, and increment its value.

4.3.2 Handoff Counter Data Type

This section illustrates, through an example, how Handoff Counter Data Types achieve eventual consistency in a network by employing a dependable 4-way handshake protocol.

The state of a Handoff Counter is composed by the following fields:

- **id**: node identification;
- **tier**: node tier;
- **below**: lower bound of values accounted in lower tiers;
- **vals**: maps node ids to integer counter values;
- **sck**: source clock;
- **dck**: destination clock;
- **slots**: maps node ids to pair (sck, dck);
- **tokens**: maps a pair of node ids (i,j) to a pair ((sck, dck), v), where v is the counter value.

Figure 4.3 illustrates an uninterrupted execution of the Handoff Counter Data Type. In the example, node i has a source clock of 10 and has been incremented 9 times, as shown in the *vals* field. The aim of node i is to transmit its state to node j so that the counter value can be incremented an additional 9 times, reaching a final value of 19.

In the first step, node i will send its state $i1$ to j, where it will be merged:

$$j2 = i1 \sqcup j1$$

The merge will create a slot in $j2$, where the key is the id of node i, and the value is a pair (sck_i, dck_j). Consequently, the j's destination clock (dck) is incremented by one. The slot expresses the capability of node j to receive a value and ensures that a value can not be received more than once.

After creating state $j2$, node j will send its state to i in return, which will evaluate j's slots and create an entry in the token. Consequently, i increments its source clock (sck) in one as it creates a new token by moving values from vals map to the token. The entry created in i's token contains a pair (source_node_id, destination_node_id) as key, and its respective value contains the increment value. Then, node i replies to the j, sending its state again.

In this interaction, node j evaluates the fields in $i2$. As i's state has a token whose key pair contains j as the destination node, and j includes a slot of the source node matching the source

clock and destination clock, j merges $j2$ with $i2$, consequently deleting the respective slot entry and incrementing its local counter in 9.

At last, j sends $j3$ to i, where i merges the received state and deletes its remaining token, creating the state $i3$.

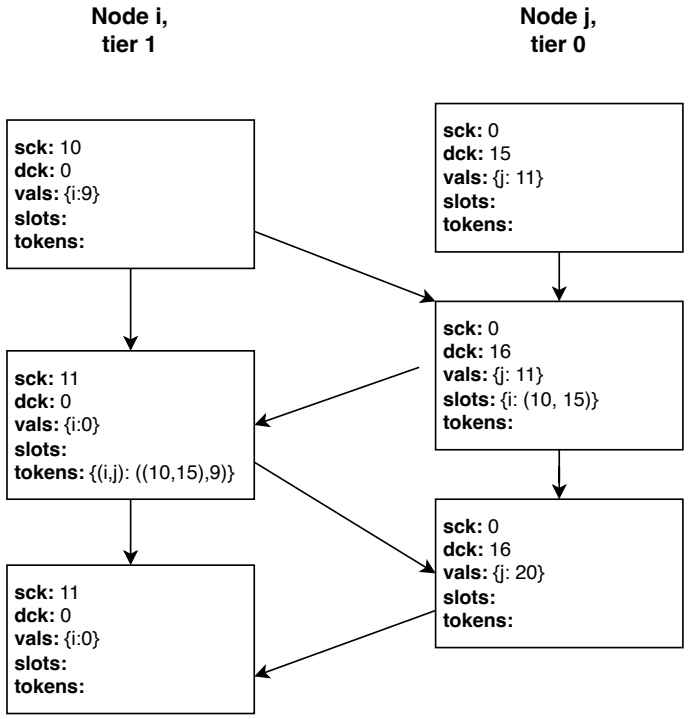


Figure 4.3: Handoff Counters protocol example

4.3.3 Notation

As the next few subsections use extensive mathematical notation, we present here the necessary semantics to understand its content.

A map is a set of tuples with keys and values (k, v), which can also be referenced with the "maps to" sign $k \mapsto v$ when accessing entries. Map updates come as $M\{...\}$. $M\{k \mapsto 1\}$, for instance, assigns 1 to key k and keeps all other keys unchanged.

Domain subtraction uses $\lhd$, where $S \lhd M$ returns the map obtained by removing all key-value pairs from S. The first element of a tuple can be accessed using fst(T), whereas snd(T) for the second. Domains are accessed using dom (i.e., dom(M)) and $\cup^f(m_1, m_2)$ is the join of two maps while applying function f for values with common keys. The function returns the key-value tuple (k, v) resultant from the joining.

Akin to many programming languages, we use **if** and **else** statements to separate different cases.

4.3.4 Operations

Figure 4.4 shows that Handoff Counters initializes its structure with empty sets. Among all operations, fetch is the simplest, as it only returns the counter's current value. The incr operation increments the counter in one, and the merge joins two states. The merge operation is the most complex, and we dedicate the next subsection to explaining each function that composes it.

$$
\begin{aligned}
&\mathsf{init}(i, tier) \doteq \{\mathsf{id} = i,\ \mathsf{tier} = tier,\ \mathsf{val} = 0,\ \mathsf{below} = 0,\ \mathsf{sck} = 0,\ \mathsf{dck} = 0,\\
&\qquad\qquad \mathsf{slots} = \{\},\ \mathsf{tokens} = \{\},\ \mathsf{vals} = \{i \mapsto 0\}\}\\
&\mathsf{fetch}(C_i) \doteq \mathsf{val}_i\\
&\mathsf{incr}(C_i) \doteq C_i\{\mathsf{val} = \mathsf{val}_i + 1, \mathsf{vals} = \mathsf{vals}_i\{i \mapsto \mathsf{vals}_i(i) + 1\}\}\\
&\mathsf{merge}(C_i, C_j) \doteq \mathsf{cachetokens}(\mathsf{createtoken}(\mathsf{discardtokens}(\mathsf{aggregate}(\\
&\qquad\qquad \mathsf{mergevectors}(\mathsf{createslot}(\mathsf{createslot}(\mathsf{discardslot}(\mathsf{fillstlost}(C_i, C_j),\\
&\qquad\qquad C_j), C_j), C_j), C_j), C_j), C_j), C_j), C_j)
\end{aligned}
$$

Figure 4.4: Handoff Counter data type operations

4.3.5 Merge operation

Firstly, the merge operation calls fillslots. If C_j has a token $((src, dst), (ck, n))$ with $(src, dst) = (i, j)$ and C_i a slot $(src, (sck, dck))$ with $src = j$, then fillslots checks if the token's clock is equal the slot's (i.e., $ck = (sck, dck)$). In the affirmative case, the slot is filled since the token matches the slot. By filled, we mean that n is added to the node, $\mathsf{vals}_i(j) = \mathsf{vals}_i(j) + n$, and the slot is deleted.

After that, the node discards invalid slots calling discardslots. The function compares C_j's source clock against $\mathsf{slots}(j)$'s. If C_j's source clock is higher than the one in the slot, the slot will never be filled. Therefore i discards the slot.

The next called function is createslot. If j comes from a lower layer and it contains new information stored under its id, then i creates a slot to receive new tokens.

The mergevectors function merges states from nodes in layer 0 by choosing the maximum between their values in vals, ensuring idempotency. The aggregate function, however, performs a vertical aggregation. It updates the below and val fields according to C_j data. The discardtokens function removes tokens that were already delivered, or that can never be dispatched.

The createtoken function creates a new token when C_j carries a slot with i as key and with source clock equals to sck_i. After creating a slot, sck_i is incremented in one to avoid repeated tags.

The cachetoken is the last function to be called in the merge operation. Node i can keep copies of tokens destined for another node k. When the transmission between two nodes fails, node i inherits the responsibility of delivering the token to k. This increases the availability.

$$\begin{aligned}
\mathsf{fillslots}(C_i, C_j) \doteq\ & C_i\{\mathsf{vals} = \mathsf{vals}_i\{i \mapsto \mathsf{vals}_i(i) + \textstyle\sum[n \mid (_, n) \in S]\},\\
& \qquad \mathsf{slots} = \mathsf{dom}(S) \lhd \mathit{slots}_i\}\\
& \textbf{where } S \doteq \{(\mathit{src}, n) \mid ((\mathit{src}, \mathit{dst}), (\mathit{ck}, n)) \in \mathsf{tokens}_j \mid \mathit{dst} = i \wedge (\mathit{src}, \mathit{dck}) \in \mathsf{slots}_i\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{discardslot}(C_i, C_j) \doteq\ & \textbf{if } j \in \mathsf{dom}(\mathsf{slots}_i) \wedge \mathsf{sck}_j > \mathsf{fst}(\mathsf{slots}_i(j))\\
& \textbf{then } C_i\{\mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\}\\
& \textbf{else } C_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{createslot}(C_i, C_j) \doteq\ & \textbf{if } \mathsf{tier}_i < \mathsf{tier}_j \wedge \mathsf{vals}_j(j) > 0 \wedge j \notin \mathsf{dom}(\mathsf{slots}_i)\\
& \textbf{then } C_i\{\mathsf{slots} = \mathsf{slots}_i\{j \mapsto (\mathsf{sck}_j, \mathsf{dck}_i)\}, \mathsf{dck} = \mathsf{dck}_i + 1\}\\
& \textbf{else } C_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{mergevectors}(C_i, C_j) \doteq\ & \textbf{if } \mathsf{tier}_i = \mathsf{tier}_j = 0\\
& \textbf{then } C_i\{\mathsf{vals} \ = \cup^{\mathtt{max}}(\mathsf{vals}_i, \mathsf{vals}_j)\}\\
& \textbf{else } C_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{aggregate}(C_i, C_j) \doteq\ & C_i\{\mathsf{below} = b, \mathsf{val} = v\}\\
& \textbf{where } b \doteq \ \textbf{if } \mathsf{tier}_i = \mathsf{tier}_j \textbf{ then } \mathtt{max}(\mathsf{below}_i, \mathsf{below}_j)\\
& \qquad\qquad \textbf{else if } \mathsf{tier}_i > \mathsf{tier}_j \textbf{ then } \mathtt{max}(\mathsf{below}_i, \mathsf{val}_j)\\
& \qquad\qquad \textbf{else } \mathsf{below}_i\\
& \qquad v \doteq \textbf{if } \mathsf{tier}_i = 0 \textbf{ then } \textstyle\sum[n \mid (_, n) \in \mathsf{vals}_i]\\
& \qquad\qquad \textbf{else if } \mathsf{tier}_i = \mathsf{tier}_j \textbf{ then } \mathtt{max}(\mathsf{val}_i, \mathsf{val}_j, b + \mathsf{vals}_i(i) + \mathsf{vals}_j(j))\\
& \qquad\qquad \textbf{else } \mathtt{max}(\mathsf{val}_i, b + \mathsf{vals}_i(i))
\end{aligned}$$

$$\begin{aligned}
\mathsf{discardtokens}(C_i, C_j) \doteq\ & \{\mathsf{tokens} = \{(k, v) \in \mathsf{tokens}_i \mid \neg P(k, v)\}\}\\
& \textbf{where } P((\mathit{src}, \mathit{dst}), ((_, \mathit{dck}), _)) \doteq (\mathit{dst} = j) \wedge\\
& \qquad \textbf{if } \mathit{src} \in \mathsf{dom}(\mathsf{slots}_j) \textbf{ then } \mathsf{snd}(\mathsf{slots}_j(\mathit{src})) > \mathit{dck}
\end{aligned}$$

Figure 4.5: Merge operation functions

$$\begin{aligned}
&\mathsf{createtoken}(C_i, C_j) \doteq \textbf{if } i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{fst}(\mathsf{slots}_j(i)) = \mathsf{sck}_i \\
&\qquad\qquad \textbf{then } C_i\{\mathsf{tokens} = \mathsf{tokens}_i\{(i,j) \mapsto (\mathsf{slots}_j(i), \mathsf{vals}_i(i))\}, \\
&\qquad\qquad\qquad \mathsf{vals} = \mathsf{vals}_i\{i \mapsto 0\}, \\
&\qquad\qquad\qquad \mathsf{sck} = \mathsf{sck}_i + 1\} \\
&\qquad\qquad \textbf{else } C_i
\end{aligned}$$

$$\begin{aligned}
&\mathsf{cachetokens}(C_i, C_j) \doteq \textbf{if } \mathsf{tier}_i < \mathsf{tier}_j \\
&\qquad\qquad \textbf{then } C_i\{\mathsf{tokens} = \cup^f(\mathsf{tokens}_i, t)\} \\
&\qquad\qquad\qquad \textbf{where } t \doteq \{((src, dst), v) \in \mathsf{tokens}_j \mid src = j \wedge dst \neq i\}, \\
&\qquad\qquad\qquad\qquad f((ck, n), (ck', n')) \doteq \textbf{if } \mathsf{fst}(ck) \geq \mathsf{fst}(ck') \textbf{ then } (ck, n) \textbf{ else } (ck', n')
\end{aligned}$$

Figure 4.6: Merge operation functions

4.3.6 Discussion

Handoff Counters manage to reduce the causal context space complexity. Instead of increasing space based on the number of replicas in the network, the space in Handoff Counters linearly increases with the number of subset of replicas. Consequently, the authors abdicated from having a totally distributed system and added a layer of hierarchy to scale the structure. Yet, the changes also increased the merge operation complexity.

The protocol presented in this thesis is the successor of Handoff Counters. Some functions declared by Handoff Counters are recycled and reused in the ROSES protocol. Therefore, it's essential for the reader to understand this section.

4.4 Topolotrees

TopoloTree [31] is a data type created by Power et al. that solves the memory consumption problem in counter-like CRDTs. The data type abdicates ID mapping while supporting non-idempotent operations and preserving strong eventual consistency. As a result, the system solves large-memory consumption by reducing the space complexity from O(n) to O(4).

Using a binary tree topology, the author enforces idempotency through hierarchy, where parent nodes deduplicate messages delivered by children.

4.4.1 Algorithm

The algorithm organizes nodes in a tree structure, where a node can only communicate with its parent and its children. The algorithm works as follows:

Algorithm 2 TopoloTree-Replica

1: **input** $state_{local}, state_{left}, state_{right}, state_{parent} \leftarrow \perp$
2: **merge** $merge_{left}$(v)
3: $state_{left} \leftarrow max(state_{left}, v)$
4:
5: **merge** $merge_{right}$(v)
6: $state_{right} \leftarrow max(state_{right}, v)$
7:
8: **merge** $merge_{parent}$(v)
9: $state_{parent} \leftarrow max(state_{parent}, v)$
10:
11: **gossip**
12: **send** left $\leftarrow state_{local} + state_{right} + state_{parent}$
13: **send** right $\leftarrow state_{local} + state_{left} + state_{parent}$
14: **send** parent $\leftarrow state_{local} + state_{left} + state_{right}$
15:
16: **operation**
17: $state_{local} \leftarrow state_{local} \cdot update$
18:
19: **query**
20: **return** $state_{local} + state_{left} + state_{right} + state_{parent}$

Each node contains four *state*: the $state_{local}$ stores the value of the local counter, the $state_{left}$ and $state_{right}$ stores the children subtrees, and $state_{parent}$ stores the value of the system outside the node's subtree, excluding the local node. The sets are non-overlapping. Formally we can say:

$$node_{local} \cap subtree_{right} \cap subtree_{left} \cap subtree_{parent} = \emptyset$$

The protocol works for any monoid. In algebra, a monoid is a set that can use associative binary operations. Here we designate this operation as a dot (i.e., $\cdot$). In algebra, a monoid is a binary operation that holds the *associative* (i.e., $(a \cdot b) \cdot c = a \cdot (b \cdot c)$) and *identity element* (i.e., $e \cdot a = a$ and $a \cdot e = a$) axioms. The system, however, excludes scenarios where the output is a set, multiset, or list. Parents can deduplicate repeated messages by using the *max* operator, ensuring idempotency and convergence.

4.4.2 Discussion

TopoloTree is a much simpler protocol when compared to Handoff Counters. It has a simple design that also leverages the network's topology to reduce the space complexity. No matter how many nodes the network has, any node will be limited to constant space consumption.

Although simple, the protocol might not be adapted to use composed structures such as sets and lists, as messages cannot be deduplicated by the *max* operation, for instance. Additionally, the paper is not clear on how to add new nodes, remove them, and migrate a child pair to another parent, which makes the structure rigid. If a node is removed, the children must migrate to another position in the tree or remain detached from the network. Using standard algorithms to make this migration does not guarantee that the network will remain consistent.

Chapter 5

ROSES Protocol

In this section, we introduce the ROSES (**R**enaming **O**perations for **S**calable **E**ventually-Consistent **S**ets) protocol. As the name proposes, the protocol scales CRDT-based, eventually-consistent sets and registers using renaming techniques. It allows the client to add and remove elements from these structures.

Although a kernel was developed to implement multiple CRDT types, such as multi-value registers, we use AWORSet CRDT to elucidate and provide a clear understanding of the protocol. Yet, we present the protocol without deltas and further improvements in the causal context for simplicity reasons.

5.1 System Model

The protocol uses a distributed system with asynchronous communication and logical time. Nodes in the system can experience failures but will eventually recover. The network may duplicate and lose messages, but lost messages are eventually delivered to their intended destination. However, messages are guaranteed to be error-free and contain the correct content.

The system follows a two-layer topology network. Tier 0 represents a sub-layer consisting exclusively of servers, while tier 1 comprises only clients. In tier 1, clients do not communicate directly with each other, but they can establish connections with multiple servers. Servers in tier 0, on the other hand, need to communicate with each other to enable replication and ensure availability in case of failures.

When a client suspects its corresponding server s crashed, it sends its state to another server s'. This allows s' to finish the protocol when s returns online. To achieve this, we have established that if a client has a link with servers m and n, then m and n are also connected. Ultimately, the network must adhere to the following rules:

- The links are bidirectional. So clients can send information to servers and vice-versa.
- If a client u has a link to two servers s and s', then s and s' are linked;
- All clients have a path to a server;
- Tier 0 is a sub-connected network;

5.2 Notation

A map can be represented as a set of keys and values (k, v), each associated with a unique value. We also represent this relation using an arrow (i.e., $k \mapsto v$). To update a value in a map, we use the $M\{...\}$ notation. $M\{d \mapsto 5\}$, for instance, updates the value of key d to 5.

The sign $\lhd$ is a domain subtraction; $P \lhd Q$ returns a map excluding all key-value pairs of Q from P; $\{(k_1,v_1),(k_2,v_2),(k_3,v_3),...,(k_n,v_n)\} \lhd \{k_1,k_3\} = \{(k_2,v_2),...,(k_n,v_n)\}$. The domain of a relation R is obtained using dom (i.e., $\mathsf{dom}(R)$); $\mathsf{dom}(\{(k_1,v_1),(k_2,v_2)\}) = \{k_1,k_2\}$.

We access values of a finite ordered list, such as a tuple, using dot notation followed by the position; $T.0$, for instance, retrieves the first position of a tuple T. We use $\cup^f(m,m')$ to join two maps while applying function f for common keys. The **if** and **else** statements are used to represent conditionals.

5.3 State

In this subsection, we present the state structure of a node using the ROSES protocol. The state is similar to Handoff Counters, as ROSES also uses tiers, clocks, slots, and tokens.

- id; $string$: node identification;
- tier; $i32$: number of the tier (0 or 1);
- sck; $i32$: source clock. Incremented when node creates a slot;
- dck; $i32$: destination clock. Incremented when node creates a token;
- te; $\{i \mapsto \{(sck,n,elem)\}\}$: a map where a node's id maps a set of tagged elements;
- slots; $\{i \mapsto \{(sck_i,dck_j)\}\}$: a map where a node's id maps a set of tuples. A tuple contains two integers;

- cc; $\{(id, sck, n)\}$: the causal context, where n is a counter that starts with 0 and is restarted when the source clock is incremented;
- token; $\{(i, j) \mapsto ((sck_i, dck_j), n_i, \{(sck_i, n', elem)\})\}$: a token that maps a tuple of ids to a triple containing the clocks, a counter and a set of tagged elements;
- transl; $\{((i, sck_i, n_i), (j, sck_j, n_j))\}$. Set of tuples where each tuple contains a triple with three integers.

ROSES protocol ensures idempotency using structures called tokens and slots. As well as in Handoff Counters, a slot means the server is prepared to receive a message from a client. For instance, a slot $i \mapsto (sck_i, dck_j)$ conveys a server j is ready to receive a token from a node i. The corresponding token must have a pair of clocks (sck_i, dck_j), where sck_i is i's current source clock when the slot was created, and dck_j is the destination clock value when the slot was created.

The source and destination clocks uniquely tag slots and tokens. When a node creates a slot, it increases the source clock. Likewise, when a client creates a token, it increases its destination clock. By monotonically increasing *sck* and *dck* the nodes guarantee that they will not be used twice.

5.4 Operations

In Figure 5.1, we show the ROSES node datatype. The init function shows the initial state of its fields. Notice that the clock starts at zero. Operation fetch returns a set with the elements in the node. It gathers all elements from te and tokens and displays them uniquely. The add and rm operations work similarly to AWORSets. The add operations insert elements in te and consequently its tombstone at the causal context cc. Whereas the rm operations not only delete matching elements from te, but also from tokens field.

$$
\begin{aligned}
\mathsf{init}(i, tier) \doteq & \{\mathsf{id} = id, \mathsf{tier} = tier, \mathsf{sck} = 0, \mathsf{dck} = 0, \mathsf{te} = \emptyset, \mathsf{slots} = \emptyset, \mathsf{cc} = \emptyset \\
& \mathsf{tokens} = \emptyset, \mathsf{transl} = \emptyset\} \\
\mathsf{fetch}(C_i) \doteq & \{e \mid ((\_,\_),(\_,\_,x)) \in \mathsf{tokens}_i \wedge (\_,\_,e) \in x\} \cup \{e' \mid (\_,\_,e') \in \mathsf{te}_i\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{add}(C_i, e) \doteq & C_i\{\mathsf{te} = \{i \mapsto \{\{(\mathsf{sck}_i, N+1, e)\} \cup \mathsf{te}_i(i)\}\}\} \\
& \textbf{where } N \doteq \{n \mid \forall (i, sck_i, n') \in cc_i \mid n' \leq n\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{rm}(C_i, e) \doteq & \{\mathsf{te} = \{(id, (sck, n, elem)) \in \mathsf{elems}_i \mid elem \neq e\}, \\
& \mathsf{tokens} = \{(k, v) \in \mathsf{tokens}_i \mid f(v, e)\}\} \\
& \textbf{where } f(v, e) = \{(\_,\_,elem) \in v \mid elem \neq e\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{merge}(C_i, C_j) \doteq & \mathsf{cachetokens}(\mathsf{createtokens}(\mathsf{discardtokens}(\mathsf{mergevectors}(\mathsf{cachetransl}(\mathsf{translate}(\\
& \mathsf{discardtransl}(\mathsf{createslot}(\mathsf{discardslot}(\mathsf{fillslots}(C_i, C_j), \\
& C_j), C_j), C_j), C_j), C_j), C_j), C_j), C_j), C_j)
\end{aligned}
$$

Figure 5.1: Node operations

From all operations, merge is the most complex. It orchestrates a series of functions that handles the ROSES protocol. In Section 5.6, we detail each function.

5.5 Algorithm

In this section, we explain ROSES algorithm without formalizations. The goal is to give the reader a good intuition of the protocol so that the formalization in Section 5.6 becomes easier to read.

The ROSES protocol begins when a client shares its state with a server. This event can happen at every new operation or periodically. Choosing one over another depends on the characteristics of the network. In write-intensive networks, for instance, periodical state exchanges would likely be more suitable.

To explain the algorithm, we will consider a network with periodical writes. In other words, the client will periodically exchange information with servers. Still, a client has an associated server to which it commonly transmits information. However, as we will explain in the following paragraphs, in case the server fails, the client can initiate communication with an alternative server.

The protocol is divided into four main states: *Notify*, *Prepare*, *Incorporate*, and *Translate*.

1. *Notify*: The protocol starts when a client c sends its state to a server s. When s receives the upcoming state, it applies the merge function. If the function verifies that c has new local information (i.e., data stored under c's id in cc), then it generates a slot and sends its state back to the client;

2. *Prepare*: Upon receiving the server's state, the client merges its state with the server's. The merge function will join upcoming elements from s in a similar way to AWORSets. Yet, if the function finds a slot with c's id as a key, the client moves data stored under its key to a new token. After that, the client sends its state to the server;

3. *Incorporate*: The server receives the client's state and merges its state. If a token matches a slot, then the slot is filled. In other words, the information in the token is renamed and added to the server. The server creates a translation entry, matching the previous tags of the elements to the new ones. It then sends its state to the client once again.

4. *Translate*: After receiving the server's state, merging it, and verifying the successful integration of the server's elements, elements in the token are translated and added to the client.

Notice that the merge function drives the algorithm as it is in charge of analyzing the upcoming state and performing changes according to it.

To better understand the steps discussed, we prepared Figure 5.2 as an example. The following paragraphs informally describe the protocol highlighting the steps mentioned previously. The system has two nodes; node x is a client at tier 1, and node y is a server at tier 0. Both are initialized in arbitrary states to make the flow more comprehensive.

Firstly, node x adds element "B" to its state. Hence, x adds a triple to its causal context and an entry to the tagged elements structure te containing "B". At some point, the *Notify* step begins as x sends its state to y, which witnesses an entry in cc_x stored under x's id (i.e., $cc_x(x) \neq \emptyset$). As a result, y creates a slot for this node with x's source clock (i.e., sck_x) and y's destination clock (i.e., dck_y): $x \rightarrow (72,97)$.

The *Prepare* state starts when Node y returns its state to the client. Upon receiving y's state, x checks that there is a slot under its id (i.e., $slots_y(x) \neq \emptyset$). Since the source clock in the slot is equal to sck_x, the client moves its tagged elements and causal context entries to a newly created token. Then the x sends its state back to the server.

The *Incorporate* step begins when Node y receives x's state. In this step, the server verifies that the client has a token matching a slot. By matching, we mean that y is the destination of the token, and y has a slot whose clocks are equal to the tokens' and whose key is equal to the origin node x. Upon this verification, the server incorporates the token elements by merging them as if they were locally added. In the example, the tagged element $(72,1,"B")$ became $(0,91,"B")$. After incorporating the elements, a translation is created. For simplicity, we will explain translations in the next subsection (subsection 5.5.1. After this step, the server sends its state back to the client.

The step *Translate* starts when the client receives the answer from the server. In this step, the client translates its token and deletes it since the server already incorporated it.

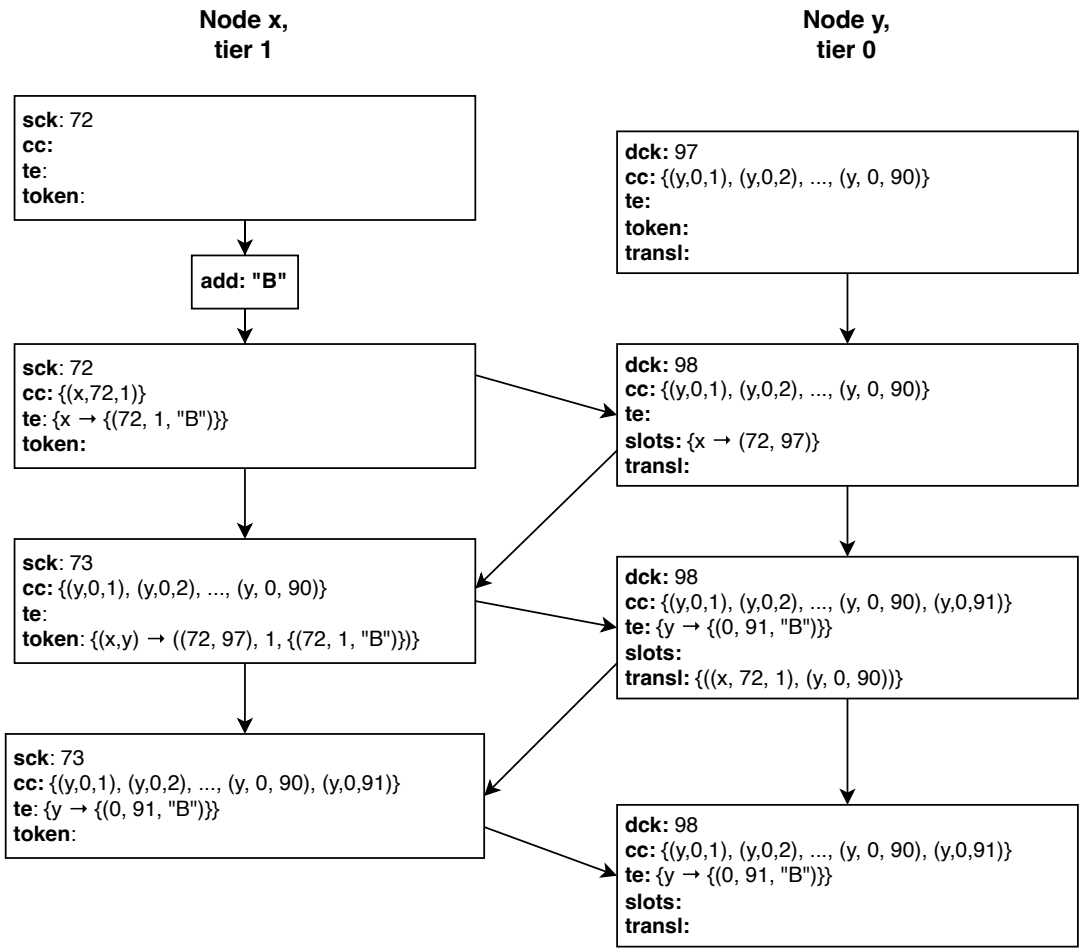


Figure 5.2: Example of ROSES protocol

5.5.1 Remove and Translations

More than just adding, the client can also remove elements from its state. When removing an element, we have three possible scenarios:

1. The removed element was at te. Therefore it wasn't transmitted to the upper layer yet;
2. The element was already moved to the token in the origin node, but it was not sent to the server yet;
3. The element was already moved to the token, and the server already incorporated it.

The **first scenario** occurs as it would in a typical AWORSet CRDT. The replica deletes the element from te while cc remains intact. The server, however, will update its causal context as it receives the token either way, although it will never add the element to te.

In AWORSet, deletions only modify the set of tagged elements. To support the **second case**, we allow the node to remove elements from the token so the user sees its own writes. If "C" was

$$\mathsf{token}_A = \{(A,B) \mapsto ((1,5),3,\{(1,1,\text{"B"}),(1,2,\text{"C"}),(1,3,\text{"K"})\}\}$$
$$\mathsf{rm}(\text{"C"})$$
$$\mathsf{token}_A = \{(A,B) \mapsto ((1,5),3,\{(1,1,\text{"B"}),(1,3,\text{"K"})\}\}$$

Figure 5.3: Remove element from token

present at the token and te, both structures would be modified. Otherwise, without this approach, the user would delete "C" from te, and upon a fetch, the element would still be in the client's state. Figure 5.3 shows an example where element "C" is removed from a token in node A.

To solve the **third case**, we must deal with the question: how can the server apply modifications that happened after it incorporates the token? Tokens are exactly-once structures; therefore, deletions applied to a token must be transmitted to the server in an alternative way, as it's impossible to incorporate the token more than once. The server ignores it, as we will explain in Section 5.6.

In the *Incorporate* step, the server adds the token's elements to its state and creates a translation, which maps new tags to the previous ones. Each entry in transl has the following tuple format $((i,sck_i,n_i),(j,sck_j,n_j))$. The first tuple element describes tags from the origin node, where the first entry is the origin node id, and the second and third are the source clock and the counter attached to the token. The second tuple element has similar information. It holds the id of the destination node, the source clock, and the counter when the server receives the token.

In Figure 5.4, we have a client node M with three elements in the token and a server N. If N receives M's token, the generated translation will be $((M,69,3),(N,0,2))$. This translation represents the range of all tags received in a token.

To better understand what we mean by range, consider the example in Figure 5.5. Node A has three tagged elements. Then, it creates a token upon receiving a slot $s_B = (A \mapsto (31,50))$. The recently created token stores all elements added to A (except the deleted ones) while its source clock was 31. As we can see, a token with source clock k stores all elements ever added to the node when its source clock was k, and only these. Therefore, the first tuple element in a translation reports that the server received all non-deleted elements generated by the client when its source clock was sck_i. It implies that the server added n_i entries to its causal context.

In other words, the server node added new entries in the causal context with tags in the range $[n_j,\ n_i+n_j)$. This assignment is done in sequence: the tag $(sck_i,0)$ becomes (sck_j,n_j), $(sck_i,1)$

$$M = \{\mathsf{sck} = 70, \mathsf{cc} = \emptyset, \mathsf{te} = \emptyset,$$
$$\mathsf{token} = \{(M,N) \mapsto ((31,50),3,\{(69,1,\text{"i"}),(69,2,\text{"j"}),(69,3,\text{"k"})\})\}\}$$
$$N = \{\mathsf{dck} = 7, \mathsf{cc} = \{(N,0,1),(N,0,2)\}, \mathsf{te} = \{N \mapsto \{(0,2,\text{"b"})\}\}\}$$

Figure 5.4: Example of state in a node B

$$
\begin{aligned}
A =& \{\mathsf{sck} = 31, \mathsf{cc} = \{(A,1),(A,2),(A,3)\},\\
& \quad \mathsf{te} = \{(31,1,\text{"m"}),(31,3,\text{"o"})\}, \mathsf{token} = \emptyset\}\\
A' =& \{\mathsf{sck} = 32, \mathsf{cc} = \emptyset, \mathsf{te} = \emptyset,\\
& \quad \mathsf{token} = \{(A,B) \mapsto ((31,50),3,\{(31,1,\text{"m"}),(31,3,\text{"o"})\})\}\}
\end{aligned}
$$

Figure 5.5: Example of state in a node A

becomes $(sck_i, n_j + 1)$ and so on until (sck_i, n_i) becomes $(sck_j, n_j + n_i - 1)$. Then when an origin node receives a translation, it looks for a tuple in transl, which the first entry regards its id. If a matching tuple is found, the origin node translates the elements in its token and joins it with its current state, similarly to AWORSet states. Figure 5.6 shows an execution demonstrating the third case.

The presented solution, however, still needs to be better adapted. Imagine a similar situation: a client x starts a protocol with a server y that crashes. Before crashing, the server has received the token from the client, successfully incorporated it, and propagated its elements to another server z, which merges its current state to y's. Meanwhile, x deletes an element from the token, and y crashes. Before y heals, z propagates its state to x. The problem is that z has x's incorporated elements but lacks translation. Therefore, when z propagates its state to x, the client will merge its state to the received one but will not translate the token. The recently removed element, in this situation, will reappear in the client's state, which is not consistent and not desired.

In this context, when replicating its state, the destination node also needs to share translations that might be in the transl structure. However, this mechanism might cause translations to live forever. For this reason, it was created a mechanism to allow a server to delete the translation when the client has already translated the target elements. If a server s_1 receives a state from a client and realizes that the client already has the incorporated elements, the server deletes the translation. Now consider that another server s_2, stores the same translation. If s_2 receives s_1's state, it will check that the upcoming state contains the incorporated elements, but s_1 does not store the translation. This means that the translation existed once but was deleted. By induction, s_2 knows that the client has already translated the elements and therefore removes it.

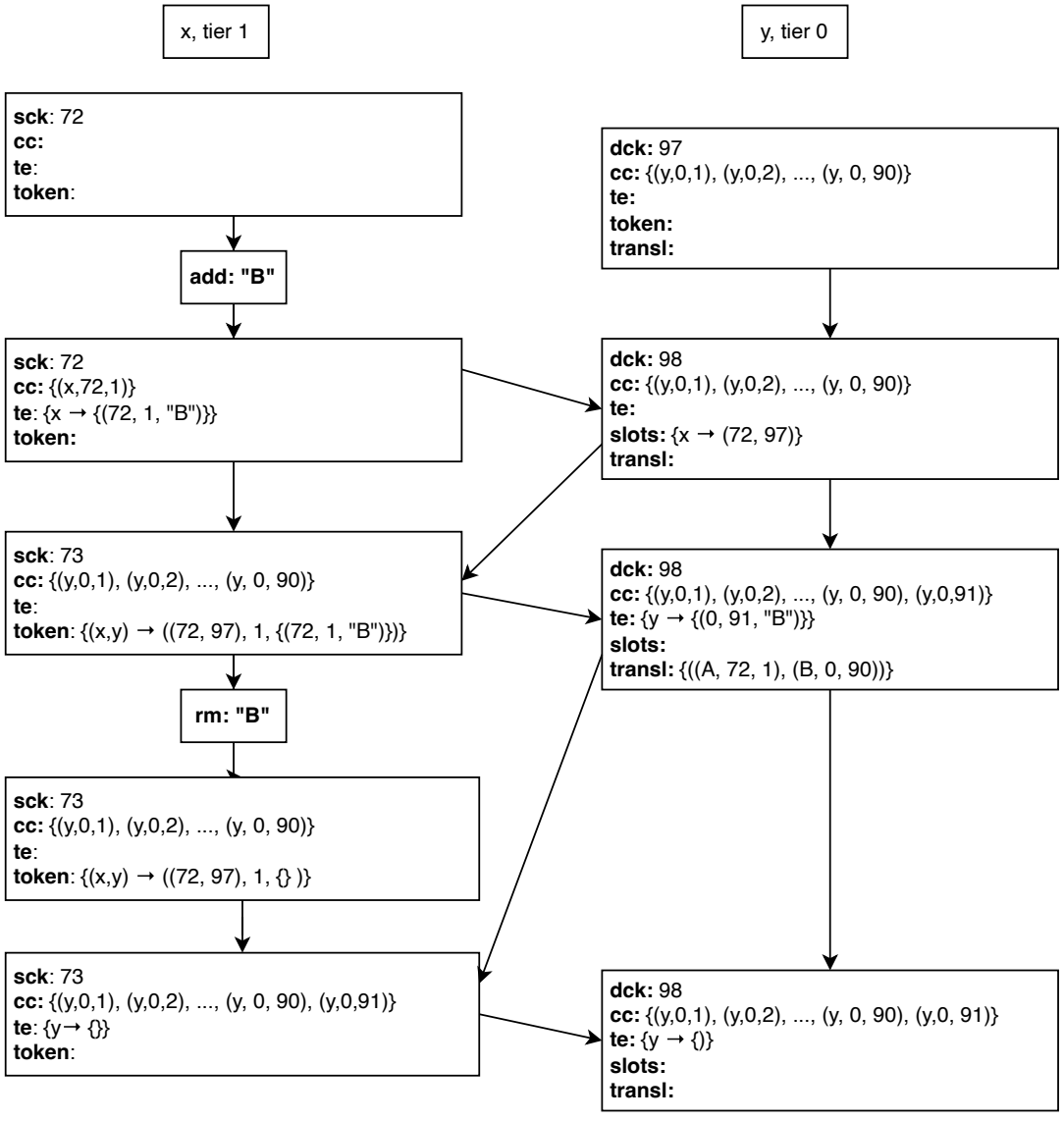


Figure 5.6: Third case when removing elements in ROSES

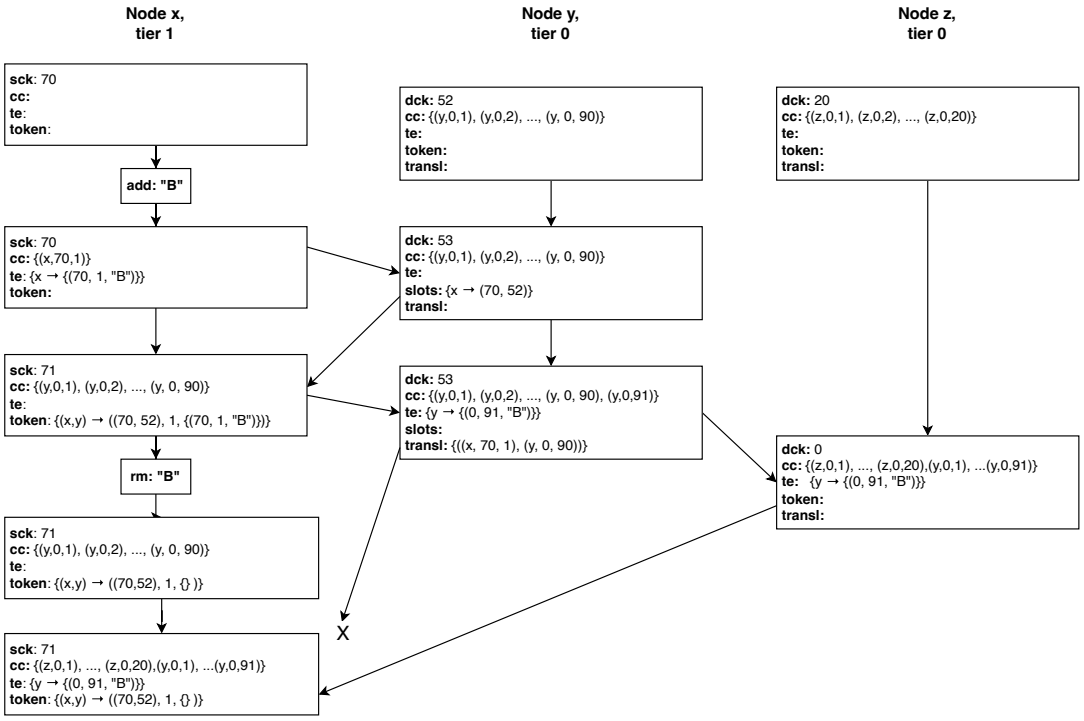


Figure 5.7: Translation in ROSES

5.5.2 Caching

To improve availability, ROSES implements a cache system similar to Handoff Counters. Suppose a node is exchanging messages with a server, but the server stops responding for some reason. In this situation, the client must wait until the server heals. To avoid this, the client starts a new protocol with another server, which inherits the responsibility of delivering the data to the fault server upon its recovery, as Figure 5.8 shows.

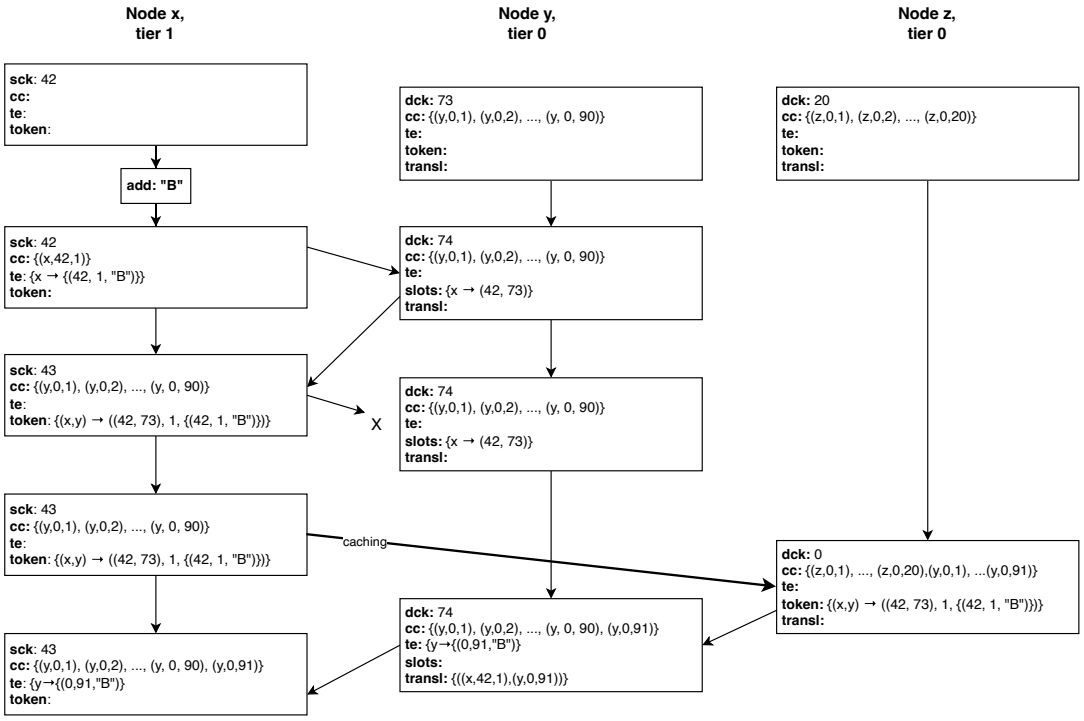


Figure 5.8: Caching in ROSES

Caching was designed to deal with two situations:

- Upon a server crash, caching allows the clients to go offline for a long time with the insurance that the information will be uploaded to the target server as soon as possible;
- In some systems, clients upload data by operation. Imagine a situation where a client added an element, but the upload failed because the target server crashed. Without caching, the client is forced to perform another operation to push the data to the server, which is not ideal. With caching, this situation is avoided;

5.6 Formalization

In this section, we will dive into the merge function. For the sake of clarity, the functions were implemented without using improvements in the causal context.

The fillslots is the first function to be called in the merge operation. If C_j contains a token that matches a slot, then it *incorporates* the elements, adding then to te. The new tags of the elements will be $(\mathsf{sck}_i, N + elem.n)$, where N is the maximum counter associated with i in its causal context. After incorporating the elements, the slot is deleted, and a translation is created in transl.

$$
\begin{aligned}
\mathsf{fillslots}(C_i, C_j) \doteq & C_i\{\mathsf{te} = \{i \mapsto te_i(i) \cup \{(\mathsf{sck}_i, elem.n + N, elem.e) \mid e \in T\}\}, \\
& \mathsf{transl} = \mathsf{transl}_i \cup \{((j, sck, n), (i, sck_i, N+n)) \mid (j, sck, n) \in \mathsf{tokens}_j\} \\
& \mathsf{cc} = \mathsf{cc}_i \cup \{(i, \mathsf{sck}_i, n+N) \in T\}, \\
& \mathsf{slots} = \{\{src \in \mathsf{dom}(T)\} \lhd \mathsf{slots}_i\} \\
& \textbf{where } T \doteq \{((src, dst), ((sck, dck), n, elems)) \in \mathsf{tokens}_j \mid \\
& \qquad \mathsf{dst} = i \wedge src \in \mathsf{dom}(\mathsf{slots}_i) \wedge \mathsf{slots}_i(src) = (sck, dst)\}, \\
& N \doteq \{n \mid \forall (i, sck_i, n') \in \mathsf{cc}_i \mid n' \leq n\}
\end{aligned}
$$

Figure 5.9: fillslot functon

Since the discardslot function is identical to the one implemented in Handoff Counters, we are not going through its details.

$$
\begin{aligned}
\mathsf{discardslot}(C_i, C_j) \doteq & \textbf{ if } \mathsf{j} \in \mathsf{dom}(\mathsf{slots}_i) \wedge \mathsf{sck}_j > \mathsf{slots}_i(j).sck \\
& \textbf{ then } \mathsf{C}_i\{\mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\} \\
& \textbf{ else } \mathsf{C}_i
\end{aligned}
$$

Figure 5.10: discardslot function

The createslot function is akin to the function used in Handoff Counters. If the received state comes from a client and the causal context associated with j's is not empty, then a new slot is created, and the destination clock of i is incremented.

$$
\begin{aligned}
\mathsf{createslot}(C_i, C_j) \doteq & \textbf{ if } \mathsf{tier}_i < \mathsf{tier}_j \wedge (j, \mathsf{sck}_j) \in \mathsf{dom}(cc_j) \wedge \mathsf{j} \notin \mathsf{slots}_i \\
& \textbf{ then } C_i\{\mathsf{slots} = \mathsf{slots}_i\{j \mapsto (\mathsf{sck}_j, \mathsf{dck}_i)\}, \mathsf{dck} = \mathsf{dck}_i + 1\} \\
& \textbf{ else } \mathsf{C}_i
\end{aligned}
$$

Figure 5.11: createslot function

After the slots, the node must handle translations, which should be done before deleting any tokens. If tokens are deleted before handling translations, it might not be possible to translate elements.

Firstly, the node discards translations that node j has already received. This is achieved with a function discardtransl, where i verifies if the translated elements are at cc_j. In case the elements were already incorporated, then the translation is deleted.

$$\begin{aligned}\mathsf{discardtransl}(C_i, C_j) \doteq & \textbf{if } \mathsf{tier}_i < \mathsf{tier}_j \\ & \textbf{then } C_i\{\mathsf{transl} = S \lhd \mathsf{transl}_i\} \\ & \textbf{where } S \doteq \{(ts,tt) \mid (ts,tt) \in \mathsf{transl}_i \mid \mathsf{j} = ts.id \mid tt \notin \mathsf{cc}_j\}\end{aligned}$$

Figure 5.12: discardtransl function

The translate is one the most complex functions. It creates a new detached state C_x, which contains the elements in the corresponding token, but with the newly assigned tags. Then the node i joins the new state and the C_i as in AWORSets.

$$\begin{aligned}\mathsf{translate}(C_i, C_j) \doteq & \textbf{if } \mathsf{tier}_j < \mathsf{tier}_i \\ & \textbf{then } C_i = C_x \sqcup C_i \\ & \textbf{where } C_x = \{\mathsf{te} = \cup^f(\{(ts,tt,token) \mid (ts,tt,token) \in X\}), \\ & \qquad \mathsf{cc} = \cup^c(\{(ts,tt) \mid (ts,tt,\_) \in X\}\}) \\ & \quad X \doteq \{(ts,tt,token) \mid (ts,tt) \in \mathsf{transl}_j \wedge token \in \mathsf{tokens}_i \mid \\ & \qquad token.0 = (ts.id, tt.id) \wedge token.1.0.sck = ts.sck\} \\ & \quad f(ts,tt,token) \doteq \{(tt.id, tt.n - ts.n + e.n, e.elem) \mid e \in token.1.2\} \\ & \quad c(ts,tt) \doteq \{(tt.id, tt.sck, n) \mid n \in [tt.n - ts.n + 1, tt.n]\}\end{aligned}$$

Figure 5.13: translate function

The cachetransl handles the caching of the translations between nodes in tier 0, which is not explicit in the equation; the only condition is tier_i == tier_j. Since client nodes can't communicate with each other, if a node receives a state from a node in the same tier, it means that both nodes are in tier 0.

The function follows the flow explained before. In short:

- Common translations belonging to C_i and C_j (i.e., $\mathsf{transl}_i \cap \mathsf{transl}_j$) are kept;
- If C_i doesn't have a translation t (i.e., $t \notin \mathsf{transl}_i$) that C_j has (e.g., $t \in \mathsf{transl}_j$) and j has already incorporated the elements the translation targets (their tag is at cc_i), it means that i deleted the translation. By induction, the target client already translated the elements. Therefore, i doesn't add t to its set of translations;
- If a translation t belongs to C_j (i.e., $t \in \mathsf{transl}_j$) and not to C_i (i.e., $t \notin \mathsf{transl}_i$), while their tags aren't in cc_i, the translation was never received by i as well as the elements the x targets. This is why this function is called before mergevectors. Translations need to be added before the elements.

$$\begin{aligned}\mathsf{cachetransl}(C_i, C_j) \doteq\ & \textbf{if } \mathsf{tier}_i == \mathsf{tier}_j \\ & \textbf{then } C_i\{\mathsf{transl} = \{(\mathsf{transl}_i \cap \mathsf{transl}_j) \cup \{(id, sck, n) \in \mathsf{transl}_i \mid (id, sck, n) \notin \mathsf{cc}_j\}\} \\ & \qquad \cup \{(id, sck, n) \in \mathsf{transl}_j \mid (id, sck, n) \notin \mathsf{cc}_i\}\end{aligned}$$

Figure 5.14: cachetransl function

Not all elements are added to the node via *incorporation*. Clients when receiving elements from a server do not incorporate it, just join the upcoming *cc* and *te* using the same strategy of AWORSets. The very same occurs when nodes from tier 0 share their state with each other. The only exception is when a client delivers its state to a server. In this case, the server ignores te elements stored under the client's id and joins the other elements with its state. This is natural, since the elements added locally to the client can only be delivered to a server, if these are encapsulated in a token.

$$\begin{aligned}\mathsf{mergevectors}(C_i, C_j) \doteq\ & \textbf{if } \neg(\mathsf{tier}_i = 0 \wedge \mathsf{tier}_j = 0) \wedge \mathsf{tier}_i \leq \mathsf{tier}_j \\ & \textbf{then } (\mathsf{te}_i, \mathsf{cc}_i) \sqcup (S, M) \\ & \qquad \textbf{where } S \doteq \{(id, (sck, n, elem)) \mid (id, (sck, n, elem)) \in \mathsf{te}_j \wedge id \neq \mathsf{j}\} \\ & \qquad\qquad M \doteq \{(id, sck, n) \mid (id, sck, n) \in \mathsf{cc}_j \wedge id \neq \mathsf{j}\} \\ & \textbf{else } (\mathsf{te}_i, \mathsf{cc}_i) \sqcup (\mathsf{te}_j, \mathsf{cc}_j) \\ & \textbf{where } (s, c) \sqcup (s', c') = ((s \cap s') \cup \{(i, sck, n, e) \in s \mid (i, sck, n) \notin c'\}) \\ & \qquad\qquad \{(i, sck, n, e) \in s' \mid (i, sck, n) \notin c\}, c \cup c')\end{aligned}$$

Figure 5.15: mergevectors function

The discardtokens function discards tokens that cannot be delivered anymore. This is achieved by comparing the destination clocks of a matching slot if exists, or the one in the node itself otherwise.

$$\begin{aligned}\mathsf{discardtokens}(C_i, C_j) \doteq\ & C_i\{\mathsf{tokens} = \{(k, v) \in \mathsf{tokens}_i \mid \neg P(k, v)\}\} \\ & \textbf{where } P((src, dst), ((\_, dck), \_)) \doteq (dst = j) \wedge \\ & \textbf{if } src \in \mathsf{dom}(\mathsf{slots}_j) \textbf{ then } \mathsf{slots}_j(src).dck > dck \\ & \qquad\qquad \textbf{else } \mathsf{dck}_j > dck\end{aligned}$$

Figure 5.16: dicardtokens function

The createtoken function as the name suggests, creates a new token to a node when C_j has a slot with i as key and with the same source clock of i (i.e., sck_i). In this case, elements are moved from te_i to the token, as well as the ones in cc_i. Right after that, the node increments its source clock.

$$\begin{aligned}\mathsf{createtoken}(C_i, C_j) \doteq & \textbf{if } i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{slots}_j(i).sck = \mathsf{sck}_i \\ & \textbf{then } C_i\{\mathsf{tokens} = \mathsf{tokens}_i\{(i,j) \mapsto (\mathsf{slots}_j(i), N, \mathsf{te}_j(i))\}, \\ & \qquad \mathsf{te} = \{i\} \lhd \mathsf{te}_i, \\ & \qquad \mathsf{cc} = \{(i, \mathsf{sck}_i, \_)\} \lhd \mathsf{cc}_i, \\ & \qquad \mathsf{sck} = \mathsf{sck}_i + 1\} \\ & \textbf{else } C_i\end{aligned}$$

Figure 5.17: createtoken function

We will not go through details about the cachetoken function as it is identical to the one exposed in Handoff Counters section.

$$\begin{aligned}\mathsf{cachetoken}(C_i, C_j) \doteq & \textbf{if } \mathsf{tier}_i < \mathsf{tier}_j \\ & \textbf{then } C_i\{\mathsf{tokens} = \cup^f(t, \mathsf{tokens}_i)\} \\ & \textbf{else } C_i \\ & \textbf{where } t \doteq \{((src, dst), (ck, n, \_)) \in \mathsf{tokens}_j \mid src = \mathsf{j} \wedge dst \neq i\}, \\ & \qquad f((ck, n, elems), (ck', n', elems')) \doteq \textbf{ if } ck.sck \geq ck'.sck \textbf{ then } (ck, n, elems) \\ & \qquad\qquad \textbf{else } (ck', n', elems')\end{aligned}$$

Figure 5.18: cachetoken function

Chapter 6

Evaluation

The first section of this chapter discusses the executed tests that give correctness confidence on ROSES. We could demonstrate correctness by formally proving the protocol. However, formal proofs are complex and time-demanding, leading us to adopt incremental unit [14] and integration [23] tests.

The second section discusses the results obtained by comparing ROSES with a typical AWORSet implementation. We present the difference between the effectiveness of the two approaches by comparing metrics depicted in graphs. We also show that the ROSES memory consumption depends on the number of servers.

6.1 Tests

We tested the ROSES library using a methodological approach separating tests into two phases:

1. Unit tests;
2. Integration tests (sequential and non-sequential).

We conducted the first phase while building the library, as it is easier to fix bugs while developing than after completion. However, unit tests won't verify if the logic behind the design is correct. They only inspect if functions portray what was idealized. But what if the design was wrong from the beginning? How can we find logic errors that we couldn't catch before? To address these matters, we created integration tests.

Integration tests check if all functions created for the library work correctly as a unit and deliver what we expected. In this context, the following integration tests were created:

- test_rnd_1x1_seq

- test_rnd_1x1_noseq
- test_rnd_1xn_seq
- test_rnd_nx1_seq
- test_rnd_nxm_noseq
- test_std_1x1_noseq
- test_std_cache_token
- test_std_transl

The test modules follows a specific nomenclature: test_<type>_<distribution>_[sequence, feature]:

- type: Each test has a type: random (rnd) or standard (std). Random tests [36] can generate a random number of add and rm operations over a predefined domain of elements. Every execution will output different results. Standard tests, on the other hand, contains a predefined set of operations to be executed. If a test is executed more than once, the result should be the same;
- distribution: Describes the network in a n $\times$ m style, where n is the number of clients and m is the number of servers. For instance, a distribution 1x1 denotes that the tests execute a network with one client and one server, respectively. Whereas a distribution nx1 has a network with n clients and 1 servers.
- sequence: Tests might be sequential (seq) or non-sequential (noseq). As described in the previous chapter, ROSES is a four-handshake protocol. If a replica applies any operation while a protocol is being executed (i.e., add, rm), we call the protocol "non-sequential." In this case, the propagation of operations is periodical to random servers. Otherwise, we consider it sequential; at every new operation, the protocol is initialized, and the user is prohibited from issuing new operations until the current protocol is over. However, the client also communicates with random servers;
- feature: Standard test modules do not necessarily have some sequence type. Inside the same module, there might be sequential and non-sequential tests to test a feature such as token caching. This field describes what feature is being tested independently of the network organization.

6.1.1 Sequential tests

Sequential tests are unrealistic as they isolate the system and inhibit parallel operations and simultaneous protocols. However, they are simple to debug and give us confidence in basic scenarios. Yet, predicting the final output can be troublesome, as each test has a distinct way of verifying if the system's final state matches the expected one.

Sequential tests have the same default configurations. Each test is executed 100 times, and in a test, the network can apply the protocol n times, where n ranges from 10 to 20. Each client cR_i issues 0 to 10 random operations (i.e., add or rm) before starting a protocol. Additionally, a client can propagate its state to any random server, and servers can share states with each other. In the following paragraphs, we explain how we managed to verify if the final system state is the expected one for each test.

As the nomenclature suggests, the **test_rnd_1x1_seq** is a random test module where the network has one client cR and one server sR. For each test, we initialized another two nodes implementing AWORSets (i.e., cA and sA), where cA applies the same operations as cR. Therefore, if cR adds an element and sends it to the sR, then cA adds the same element and sends its state to the sA. At the end of the test, we expect that cR, cA, sR, and sR must have the same set of elements since the operations applied are equivalent.

The **test_rnd_1xn_seq** module contains a variable number of servers, which is 100 by default. After n protocol executions, the client stops issuing operations and exchanges its states with every server until the system converges. Along with this execution, we create a single instance of a node cA implementing the AWORSet data type. All the operations applied to cR are also applied to cA. When convergency is achieved, the set of elements in each node, including servers, should be equal to the set in cA.

The **test_rnd_nx1_seq** modules have a variable number of clients, which is 100 by default. The strategy to validate the system's final state is akin to **test_rnd_1xn_seq**. After n protocol operations, clients cR_i stop issuing operations and exchange their state with the server sR for synchronization. In this module, we created an AWORSet node cA_i for each client cR_i in ROSES protocol and one server sA. The same operations applied to a node cR_i were also applied to its correspondent cA_i, and for every protocol execution between a node cR_i and sR, the node cA_i shares its state with sA. At the end of the test, all the nodes in the server should have the same set of elements of the server implementing AWORSets.

We also implemented **test_rnd_nx1_seq** with a second form of validation. Instead of synchronizing all the elements after n operations, we compared the states between every pair cR_i and cA_i and between the sR and sA. In the end, the corresponding pairs should have the same of elements.

6.1.2 Non-sequential tests

In sequential tests, we would deduce the system's final state by replicating the issued operations in a similar network composed of nodes implementing AWORSets. We follow a similar approach for **test_rnd_1x1_noseq**. But unfortunately, non-sequential tests in a scenario with m clients and n servers do not necessarily have a straightforward translation to AWORSets. Since the ROSES protocol is slower in propagating elements, the final result of the network might differ from its AWORSet equivalent.

Let's consider the example shown in Figure 6.1, which depicts a subgroup of nodes in a network. Here, x acts as a client, while z and y are servers. In this example, x adds an element "A" to its state. However, due to a network partition, x caches its token in z. While y is still waiting

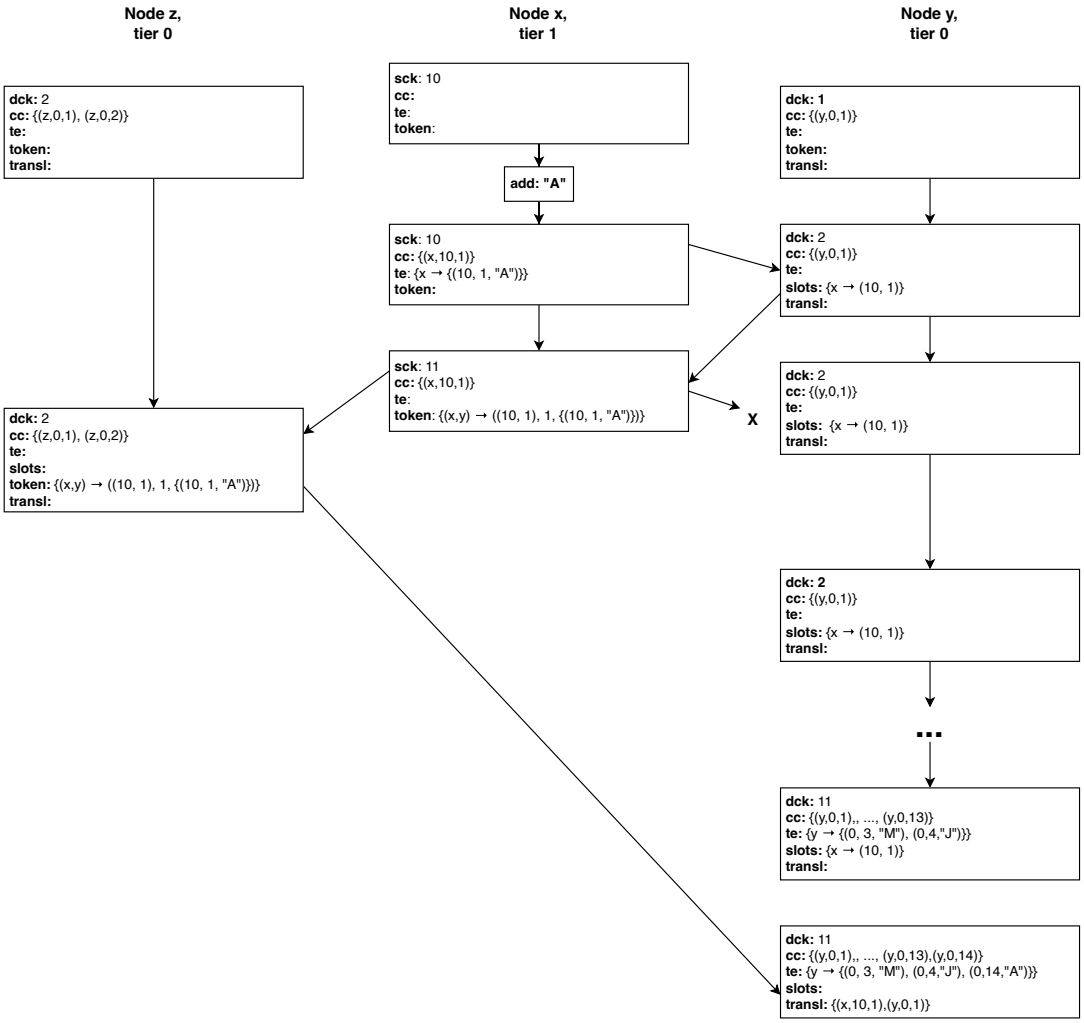


Figure 6.1: Example of unordered operations

for *x*'s token, it receives a series of other elements that were issued after "A". Eventually, *z* finally receives *x*'s token and incorporates "A".

If we were to replicate the operations from the example using AWORSets, the element "A" would have been received by node *y* before the elements issued by other clients (e.g., "M", "J") that came after it. In this example, even though *y* received the elements in a different order, eventually, it will have the same elements of the AWORSet network. However, if a client attempts to remove element "A" before *y* receives it, the result will differ. In the AWORSet network, the element would already have been received and therefore removed, whereas this does not happen in the ROSES network.

We could test the network without caching to avoid this phenomenon, but the translation feature would cause similar behavior. A solution would be keeping track of when the server received a token and share states in AWORSet networks upon this event. However, there are some disadvantages to this approach:

- **Test correctness**: This adaptation would increase the test complexity. At a certain point, it would be hard to tell if the test is correct. It's a good practice to keep it simple;
- **Test forced fitting**: Tests are expected to validate a code output or state. If we modify the structure AWORSet to verify the output, at which point are we just adapting the test so it passes?

For the reasons cited, we devised an alternative test strategy. Consider a network with n clients and m servers with the following rules and variables:

- x is the total number of messages exchanged in the system. Therefore, every time a node sends its message to another, x is incremented in one;
- k is a constant and pre-defined number:
  - If $x < k$, then the client can make operations of add and rm; **(Phase 1)**
  - If $x >= k$, then nodes can only make rm operations. Messages exchanges are not random anymore but are made to achieve convergence as fast as possible; **(Phase 2)**
- A client can add any elements it wants in a range [0, p], where p is also defined before the test starts. Thus, if $p = 2$, the following set of operations is allowed:

$$\{\text{add 0}, \text{add 1}, \text{add 2}, \text{rm 0}, \text{rm 1}, \text{rm 2}\}$$

- Still, a client can have one or no *peculiarity*. A *peculiarity* is when a node doesn't "like" an element i. Every time a node with a *peculiarity* i, receives or adds an element i, the element is removed. When a node x, for instance, with *peculiarity* 1 receives an element whose value is 1, x will apply rm 1. A node cannot apply a rm unless it has a *peculiarity* over the removed element. Therefore, x can apply rm 1, but not rm 2 as it doesn't have *peculiarity* 2;

While the test is in execution, it stores how many times an element was removed and added by each node. By the end of the test, we want to verify that:

1. One element cannot be removed more times than it was added;
2. In **phase 2** all the nodes must reach convergency. Therefore, the final state should be the set of added elements, less the set *peculiar* elements. For example, suppose a system applied the following set of operations {add 1, add 2, add 4, add 10, rm 10}. In that case, we have the set of added elements is: {1,2,4,10}, and if one element was removed, then the set of peculiar elements is {10}. Thus, in the final state, each node should have the following elements: {1,2,4,10} - {10 } = {1,2,4}.

This test builds the module **test_rnd_nxm_noseq**. It helped us fix bugs in the design logic and is simpler than adapting a network using AWORSets. This test passes successfully.

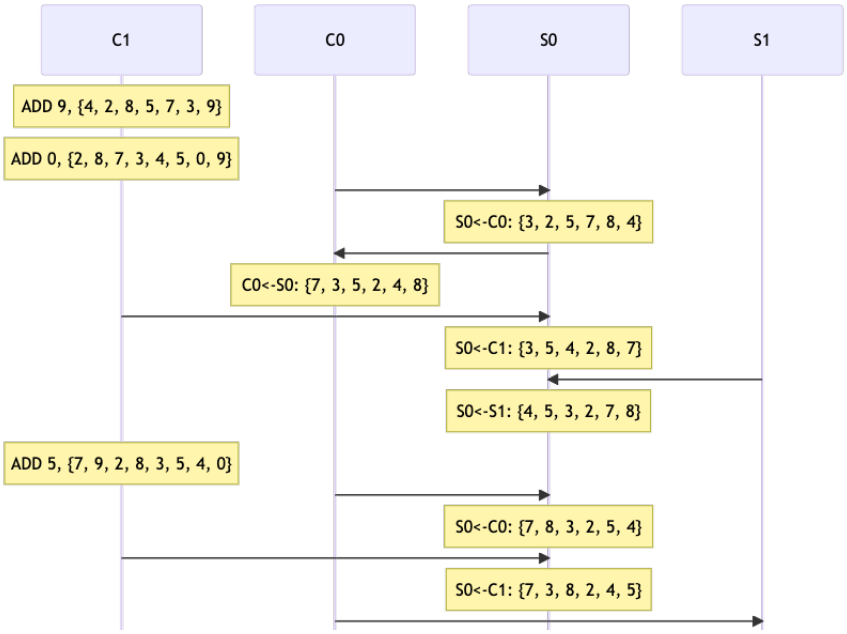


Figure 6.2: Mermaid graph flow

6.1.3 Logs

As tests scale (e.g., high number of clients, tests, servers, messages, etc.), debugging and understanding the source of errors becomes increasingly complex. To address this challenge, we have developed a log syntax to transform it into a visual representation or replay of the last test execution through code. While we won't delve into the implementation details of the log, we aim to provide an overview of how we have applied these text-to-text transformations.

Within the source code, we have introduced a file containing macros. Each macro can optionally accept parameters and outputs a series of instructions. Each instruction is prefixed with either "++ " or "– ". Instructions starting with "++ " are used for generating code that can replay test executions. Following the prefix, these instructions are written in Rust. On the other hand, instructions beginning with "– " are employed to create a visual representation of the most recent test execution. Although we utilize text-to-text transformation, these instructions include diagram syntax suitable for rendering with tools like Mermaid [37].

We can easily generate the log and select instructions prefixed with "– " to produce a graph depicting the test flow by utilizing a Makefile. Conversely, we can choose instructions prefixed with "++ " to generate executable code for reproducing the test execution. Figure 6.2 shows a subsection of a Mermaid Graph Flow using the logs.

6.2 Results

We have conducted a controlled analysis to verify ROSES performance against AWORSet by comparing their memory consumption in bytes. Additionally, we developed a simulator using Rust programming language. The source code with the simulator and with an implementation of

ROSES can be found at https://github.com/Jumaruba/ROSES-protocol. The network simulations have a diversified number of clients $c = \{16, 64, 256\}$ and servers $s = \{4, 16\}$. By taking the Cartesian Product [33] of $s \times c$, we obtained the set of network configurations used in each scenario: $\{4 \times 16,\ 4 \times 64,\ 4 \times 256,\ 16 \times 16,\ 16 \times 64,\ 16 \times 256\}$. In each simulation time unit, nodes can add or remove up to 10 random elements within a predefined range. After transmitting its state to a destination node, the origin node has the possibility of receiving an answer right away.

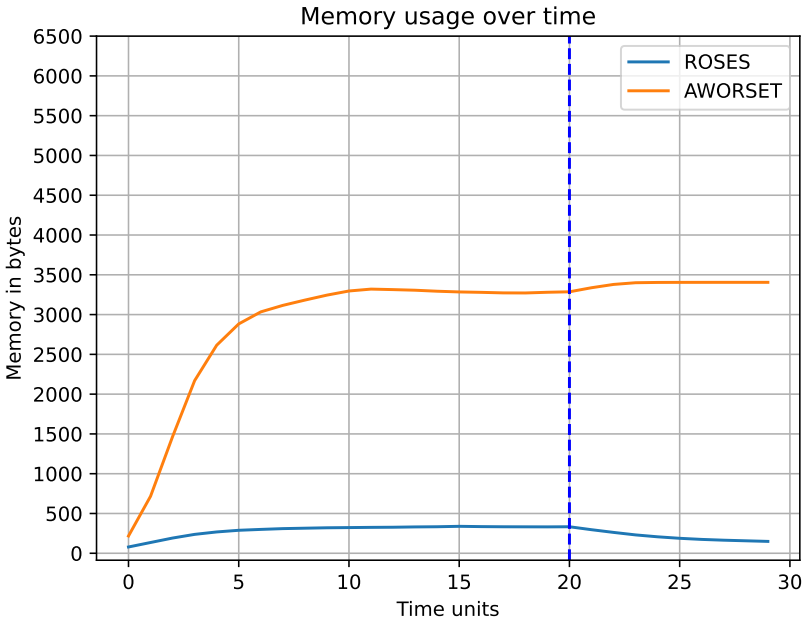
Figures 6.3 and 6.5 depict the graphical analysis for network scenarios with 16 and 64 clients, respectively. We calculated the mean of the space occupied by each node and divided it by the number of simulations executed. After time unit 20, the nodes stop generating new operations and only share their states with other nodes.

All the graphs have similar behavior regarding AWORSets. From time units 0 to 20, the curve initially ascends rapidly and reaches a "constant." We say "constant" as it increases gradually because the causal context never diminishes. But since the probability of removing and adding operations are similar, the curve becomes flat. After time unit 20, the simulation seizes emitting operations, and the network strives to achieve consistency. Consequently, elements added in other clients will be replicated in other nodes without being potentially removed. This causes the curve to ascend for a brief period before stabilizing.

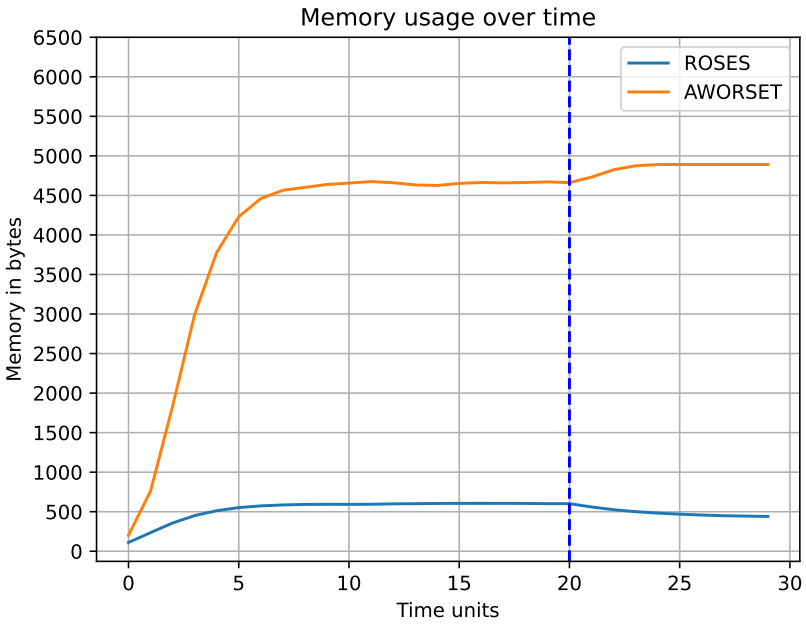
ROSES graph curve presents similar behavior. From time units 0 to 20, its curve increases, although slowly, and then reaches an "equilibrium." But after this mark, memory consumption diminishes. This is expected since structures such as tokens and translations start to be discarded.

Figures 6.3a and 6.3b show two graphs with 16 clients and a variable number of servers, 4 and 16. By comparing the two graphs, we can see that increasing the number of servers affects not only the AWORSets curve but also ROSES'. Figures 6.4a and 6.4a show similar behavior.

However, if we compare Figures 6.3a with 6.4a, and 6.3b with 6.4a, we can see that adding more clients and maintaining the number of servers significantly changes the memory consumption in AWORSets. Still, ROSES' memory consumption remains almost the same. This shows that in ROSES, the memory consumption depends on the number of servers and not clients.

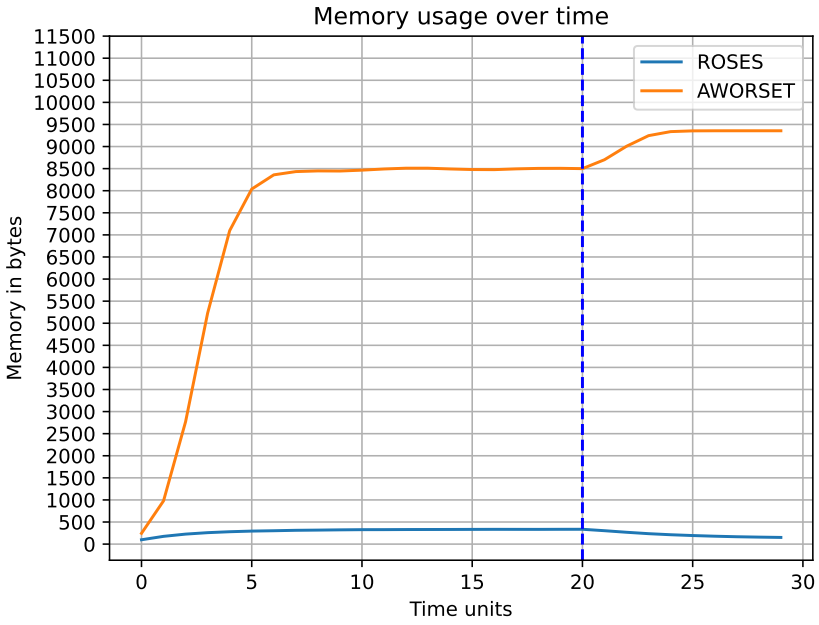


(a) Simulation with 4 servers and 16 clients (4x16)

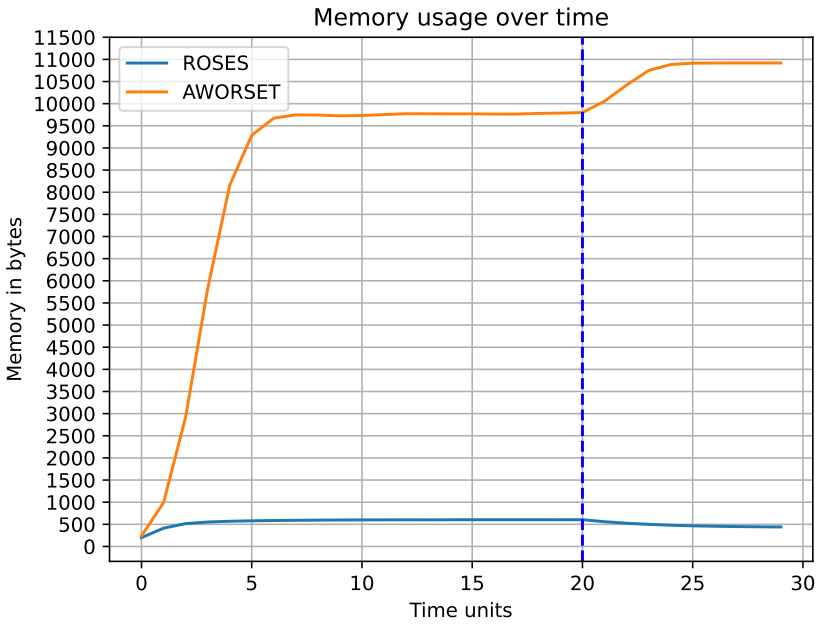


(b) Simulation with 16 servers and 16 clients (16x16)

Figure 6.3: Simulations with 16 clients

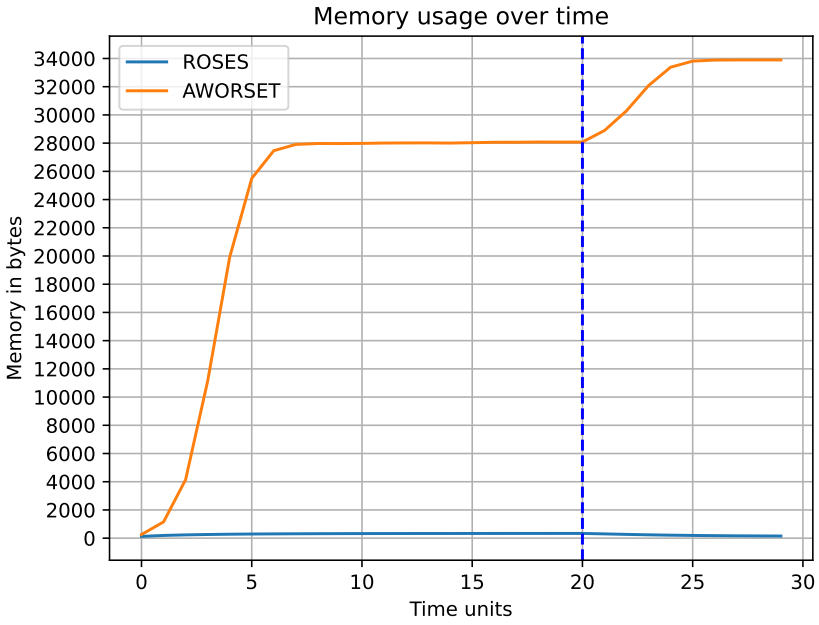


(a) Simulation with 4 servers and 64 clients (4x16)

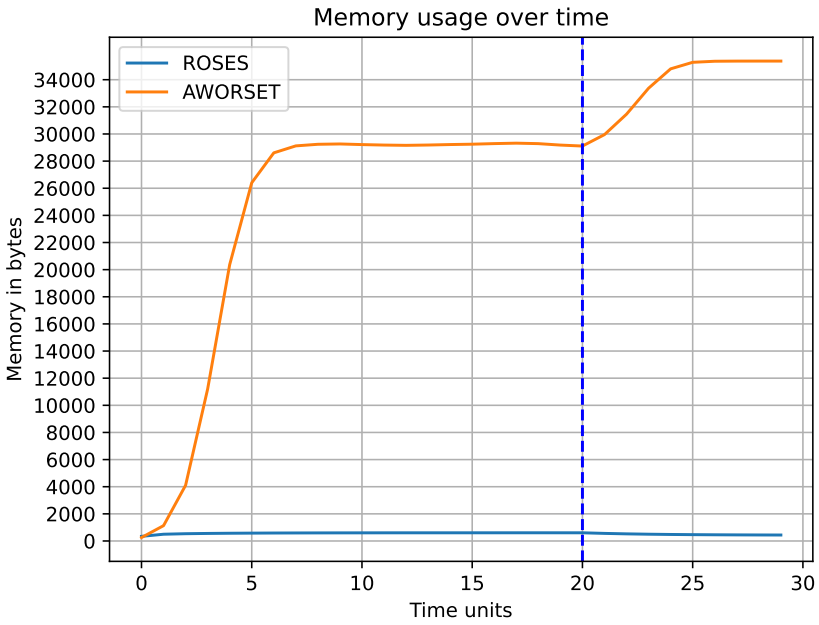


(b) Simulation with 16 servers and 64 clients (16x64)

Figure 6.4: Simulations with 64 clients



(a) Simulation with 4 servers and 256 clients (4x256)



(b) Simulation with 16 servers and 256 clients (16x64)

Figure 6.5: Simulations with 64 clients

Chapter 7

Conclusion

Distributed systems don't have a single server to store all relevant information. Instead, data is spread across the network and can be written to one node, and then replicated to others. However, this design doesn't simultaneously provide high consistency, availability, and partition tolerance (CAP theorem).

CRDTs are distributed data structures suitable for systems that require high availability by momentarily abdicating consistency. It can synchronize without requiring expensive consensus protocols and easily solve conflicts between replicas by storing a copy of each node's state locally. Although highly available, CRDTs don't scale well in large networks.

When replicating other nodes' states, the space complexity grows proportionally with the number of replicas in the network. Then, large networks must count with high storage capacity nodes. However, P2P networks are likely heterogeneous; some replicas have more computer power and storage than others. Therefore, scalability is restricted to the computer power of replicas, which might be out of reach.

As discussed in the State of the Art, some solutions, such as Handoff Counters and TopoloTrees, were created to solve this issue. Most base strategies rely on reshaping the network topology to simplify the state. However, the solutions are limited to counter CRDTs.

In this dissertation, we presented the ROSES protocol. It leverages the structure devised by Handoff Counters and adapts it to suit registers and set-based CRDTs, although we only implemented it for AWORSets. The main contribution of ROSES is implementing a translation mechanism to allow the renaming of deleted elements, which also improves availability. After tests, we verified that the system improves the space complexity of nodes in the system, making this structure eligible for use in machines that lack computational space.

Nevertheless, ROSES have vast potential for improvements. ROSES work with a two-layer topology, encompassing just a small set of modern systems. The most significant refinement would be increasing the number of layers the protocol can work. Other minor improvements could be made, such as diminishing the token's size. Still, ROSES provide a solid start to building more robust protocols supporting set and register-based CRDTs.

References

[1] Balancing Strong and Eventual Consistency with Datastore. https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore/#strong-consistency-on-reading-entity-values-and-indexes. Accessed: 2023-02-03.

[2] Conclave - A private and secure real-time collaborative text editor. https://conclave-team.github.io/conclave-site/. Accessed: 2023-02-02.

[3] How league of legends scaled chat to 70 million players - It Takes Lots Of Minions. http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html. Accessed: 2023-02-02.

[4] Key lessons learned from transition to nosql at an online gambling website. https://www.infoq.com/articles/key-lessons-learned-from-transition-to-nosql/. Accessed: 2023-02-02.

[5] Low Latency Underpins Carrierś Ability to Compete & Succeed in Vertical Markets. https://carrier.huawei.com/en/technical-topics/fixed-network/low-latency-underpins. Accessed: 2023-02-02.

[6] Reading Group. Conflict-free Replicated Data Types | Aleksey Charapko. http://charap.co/reading-group-conflict-free-replicated-data-types/. Accessed: 2022-12-27.

[7] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, jan 2018.

[8] Paulo Sérgio Almeida and Carlos Baquero. Scalable eventually consistent counters over unreliable networks. *Distributed Computing*, 32(1):69–89, February 2019.

[9] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation.

[10] H. A. Priestley B. A. Davey. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2 edition, 2002.

[11] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, May 2013.

[12] K. A. Baker, P. C. Fishburn, and F. S. Roberts. Partial orders of dimension 2. *Networks*, 2(1):11–28, 1972.

[13] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[15] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–284, San Diego California USA, January 2014. ACM.

[16] Eric Burgener. Meeting the High Availability Requirements in Digitally Transformed Enterprises. 2022.

[17] Joe Celko. Chapter 1 - nosql and transaction processing. In Joe Celko, editor, *Joe Celko's Complete Guide to NoSQL*, pages 1–14. Morgan Kaufmann, Boston, 2014.

[18] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '03, pages 407–418, New York, NY, USA, 2003. Association for Computing Machinery.

[19] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 407–418, New York, NY, USA, 2003. Association for Computing Machinery.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.

[21] Vitor Enes, Paulo Sergio Almeida, Carlos Baquero, and Joao Leitao. Efficient Synchronization of State-Based CRDTs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Macao, Macao, April 2019. IEEE.

[22] Dietrich Featherston. Cassandra: Principles and application. 2010.

[23] Martin Fowler and James Shore. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2012.

[24] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[25] Joseph Ingeno. *Software Architect's Handbook*. Packt Publishing, August 2018.

[26] Keith W. Ross Jian Liang, Rakesh Kumar. Measurements and understanding of the kazaa p2p network. In *Current Trends in High Performance Computing and Its Applications*, pages 425–429, Berlin, Heidelberg, 2005.

[27] Martin Kleppmann. CRDT Glossary • Conflict-free Replicated Data Types. https://crdt.tech/glossary. Accessed: 2022-12-28.

[28] Martin Kleppmann. A Critique of the CAP Theorem, September 2015. arXiv:1509.05393 [cs].

[29] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[30] Mohsen Mosleh, Kia Dalili, and Babak Heydari. Distributed or Monolithic? A Computational Architecture Decision Framework. *IEEE Systems Journal*, 12(1):125–136, March 2018. arXiv:1608.00944 [cs].

[31] Conor Power, Shadaj Laddad, Chris Douglas, Joseph Hellerstein, and Dan Suciu. Topolotree: From o(n) to o(1) crdt memory consumption via aggregation tree gossip topologies. Rome, Italy, May 2023.

[32] Nuno Preguiça. Conflict-free replicated data types: An overview. *CoRR*, June 2018.

[33] Catherine M. Ricardo. Relational database systems. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 661–680. Elsevier, New York, 2003.

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

[35] 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*, Lecture Notes in Computer Science, pages 386–400. Springer, 2011.

[36] John Smith. Random tests: A comprehensive analysis. *Journal of Software Engineering*, 15(3):123–145, 2023.

[37] Knut Sveidqvist and Sidharth Vinod. Github: Mermaid. https://github.com/mermaid-js/mermaid. Accessed: 2023-06-23.

[38] Bartosz Sypytkowski. Optimizing state-based CRDTs (part 2), August 2018.

[39] Maarten van Tanenbaum, Andrew S;Steen. *Distributed systems: principles and paradigms*. Always learning;Pearson custom library. Pearson, 2nd ed., new international ed edition, 2013;2014.

[40] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.