FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

State-based CRDT Infrastructure

Miguel Rodrigues

Mestrado em Engenharia Informática e Computação

Supervisor: Carlos Baquero

July 23, 2024

State-based CRDT Infrastructure

Miguel Rodrigues

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the comitee:

President: Prof. Pedro Souto
Referee: Prof. Filipe Araújo
Referee: Prof. Carlos Baquero

July 23, 2024

Abstract

Conflict-free Replicated Data Types (CRDTs) establish mathematical rules to reach consistency deterministically by exploring monotonicity. CRDTs feature eventual consistency and appear in two formats: op-based and state-based. Despite their ability to operate over unreliable networks, state-based CRDTs lack practicability. The state-based synchronization procedure requires full-state bidirectional exchange, which may be intractable as the message's size grows indefinitely alongside the state.

Recent efforts, such as δ-CRDTs and irredundant join-decompositions worked towards smaller messages expressed as δ-groups. Despite these efforts, at least one replica must transfer its entire state in a synchronization round between two replicas. This aspect can be enhanced, especially if replicas share a significant portion of the state.

This dissertation aims to comprehend the impact of state similarity on state-based synchronization and how leveraging it can improve existing methods. The main contributions of this work comprise two new synchronization techniques for state-based CRDTs: bucketing and bloom-based, akin to rsync's weak and strong checksums.

Our techniques exchange digests between replicas to detect similarities and leverage irredundant join-decompositions to derive *differences*. We also detail our implementation, intending to ease the adoption of these techniques in new products. We analyze the transmission volume and the time to synchronize to verify the impact of similarity on the synchronization process and to validate our contributions. Our results demonstrate an improvement over the existing methods as similarity increases.

Keywords: CRDTs, Join-decomposition, Similarity, Synchronization, Anti-entropy

Resumo

Conflict-free Replicated Data Types (CRDTs) establecem regras matemáticas de modo a atingir consistência de forma determinística explorando monotonicidade. Os CRDTs oferecem consistência eventual e surgem em dois formatos: *op-based* e *state-based*. Apesar da sua capacidade para operar em redes instáveis, os *state-based* CRDTs apresentam lacunas no que toca à sua praticabilidade. O processo de sincronização baseado em estado troca a totalidade do estado das réplicas envolvidas bidirecionalmente, o que pode ser inviável, uma vez que, o tamanho das mensagens cresce ao mesmo ritmo que o estado.

Esforços recentes, como δ-CRDTs ou *irredundant join-decompositions* trabalharam no sentido de obter messagens mais pequenas expressas sob a forma de δ-*groups*. Apesar destes esforços, num processo de sincronização baseado em estado entre duas réplicas, pelo menos uma delas necessita de transferir a totalidade do seu estado. Este é um aspeto que pode ser melhorado, especialmente se estas réplicas partilharem entre si uma porção significativa dos seus estados.

Esta dissertação procura compreender o impacto que a similaridade de estado tem sobre a sincronização baseada em estado e como tirar partido dela para melhorar os métodos existentes. As principais contribuições deste trabalho incluem duas novas técnicas de sincronização dirigidas aos *state-based* CRDTs: *bucketing* e *bloom-based*, análogas às verificações fraca e forte feitas pelo *rsync*.

As técnicas que apresentamos trocam mensagens curtas entre réplicas de modo a detetar similaridades e tiram partido de *irredundant join-decompositions* de maneira a derivar as *diferenças*. Também detalhamos a nossa implementação com o objetivo de facilitar a adoção destas técnicas em novos produtos. Analisamos o volume de transmissão e o tempo de sincronização, com vista a verificar o impacto da similaridade no processo de sincronização e a validar as nossas contribuições. Os resultados apresentados demonstram uma melhoria sobre os métodos existentes à medida que a similaridade aumenta.

Acknowledgements

To my supervisor, Professor Carlos Baquero, for his valuable insights, helpful advice and guidance, and exciting discussions during this journey. To my family for their support and for always encouraging me to pursue my interests. To Margarida for her company and patience and for diagnosing me with *daisies madness*. And finally, to my friends with whom I had the privilege to grow and gather remarkable memories.

Miguel

"I guess, this is my dissertation"

Kanye West

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Symbols and Abbreviations

BF	Bloom Filter
CRDT	Conflict-free Replicated Data Type
δ-CRDT	Delta-based Conflict-free Replicated Data Type
EC	Eventual Consistency
MST	Merkle Search Tree

Chapter 1

Introduction

In this chapter, we briefly outline the context, the problem, and the goals of this dissertation.

Section 1.1 starts by supplying the context for this work. Section 1.2 debriefs the motivation for this work by exposing some of the drawbacks present in the art. Section 1.3 enumerates the objectives of this thesis. Section 1.4 concludes the chapter by mapping the document structure.

1.1 Context

Sharing a consistent state across multiple computers, or replicas, is ubiquitous in the modern software era. It is so relevant that computer scientists have been devising models for it for decades now.

Theoretically, the answer to this problem is surprisingly simple, and it is the foundation to *strong consistency* [38]. This model's premise is to ensure that all replicas execute updates in the same order. Nonetheless, computer networks are unreliable, and systems must implement synchronization mechanisms to guarantee that their peers effectively receive such updates. Naturally, this hurts the system's performance, and consequently, the capacity to respond to incoming requests, i.e., availability decreases.

To address this concern, [31] introduced *eventual consistency* (EC). Under this model, replicas may be inconsistent for a short period and converge into an equivalent, thus consistent, state if no new updates arrive. Furthermore, reasoning about the system's state when divergences exist is challenging.

Conflict-free Replicated Data Types (CRDTs) [33] are an instrument that delivers EC. Although prior ad-hoc solutions existed, CRDTs were the first to introduce a model for deterministic state reconciliation based on mathematical properties. CRDTs come in two flavors: (i) op-based and (ii) state-based, depending on how updates propagate among replicas. Moreover, CRDTs have a strong presence in the industry's products within the realm of geo-replicated systems [4].

One noteworthy benefit of state-based CRDTs is that updates can be disseminated through an unreliable channel. Even though updates can be lost, duplicated, or reordered, they only need to arrive at their intended destinations to ensure consistency. There is, however, a price for such

flexibility. In state-based CRDTs, synchronization requires a full-state bidirectional exchange between two peers. As operations arrive at the replicas, their states grow indefinitely. Thus, synchronization eventually reaches a point where it becomes prohibitively costly.

Moreover, δ-CRDTs [2] target this exact issue of state's indefinite growth. It introduces the concept of a δ-mutator – a function that conveys a mutation whose output is δ-mutation, i.e., a small size state. This way, δ-mutations become the subject of dissemination. Besides, δ-mutations decouple the effects of an incoming operation from a replica's state. The join between a δ-mutation and a target state results in a δ-group. Hence, one can view a state-based CRDT as a composition of δ-groups.

1.2 Motivation

Although δ-CRDTs are a clever mechanism to decompose larger states, the fact is that they are inapt to avoid the synchronization problem mentioned above by themselves.

The solutions to this problem attach metadata to these data types. For instance, Δ-CRDTs [36] use logical clocks to compute a δ-group to send to a peer. However, these approaches only work when such metadata is correct. Otherwise, it must fall back to the classic state-based synchronization. Therefore, they do not function in scenarios of high churn, e.g., dynamic membership or high likelihood of network partitions.

Another problem of δ-CRDTs is the high ratio of state redundancy transmitted among replicas. Enes et al. [13] identified two kinds of inefficiencies: (i) when two replicas exchange the same state more than once, and (ii) when a replica joins state that it already contains, thus propagating it more than once. Given this, the authors introduced irredundant join-decompositions to tackle these inefficiencies. Essentially, they express atomic portions of the state, from which one can derive optimal *differences* between two given states. Such *differences* take the format of δ-groups.

With join-decompositions, synchronization takes the following shape: (i) a replica ships its entire state, while (ii) its peer computes and sends back the *difference*. Again, we land on a scenario of a full-state transmission, which can be expensive or unattainable. Besides, replicas may share a significant portion of the state, given the replication context. Thus, it is vital to determine the degree of similarity between replicas to achieve efficient synchronization.

1.3 Objectives

This work seeks to enhance the existing synchronization methods in the state-based CRDTs domain. We study the influence of replicas' similarity on synchronization and develop new strategies that take it into account in a scenario where the network recently healed from a partition.

Furthermore, we construct an environment that implements and simulates synchronization between replicas. The resultant product is open to anyone willing to contribute. We also look forward to new state-based CRDT implementations that benefit from the work portrayed in this document.

1.4 Document structure

Beyond this introductory chapter, this dissertation contains seven more chapters.

Chapter 2 revisits the fundamental concepts of CRDTs. Chapter 3 explores prior art in the context of synchronization and probabilistic data structures. Chapter 4 documents the research questions guiding our investigation and the methodology used. Chapter 5 introduces new techniques for synchronizing state-based CRDTs. Chapter 6 describes the implementation details of the developed synchronization simulator. Chapter 7 showcases the experiments made with the developed simulator. Chapter 8 closes this dissertation and supplies directions about future work.

Chapter 2

Background

In this chapter, we will discuss the fundamental concepts behind CRDTs. These are the necessary concepts to understand the remaining chapters of this thesis.

Section 2.1 outlines the relevant consistency models. Section 2.2 introduces the CAP theorem and details its importance regarding this work. In section 2.3, join-semilattices are presented and defined. To close the chapter, section 2.4 provides the background on op-based and state-based CRDTs, with particular attention on the latter.

2.1 An overview on consistency

Data replication provides better availability and reliability for distributed systems. When a network failure occurs, and a particular node becomes unreachable, another node may fulfill incoming requests and provide data recovery. However, this is possible only if a replica is up to date with the faulty replica. Otherwise, the system lands in an inconsistent state. In this context, replicas are nodes that store replicated data.

Therefore, consistency happens when all replicas are in an identical state. In practice, what this means is that when a client requests data from the system, the information obtained must be the same regardless of which replica processed the request.

Even though the contract for consistency looks very simple, it has various models. The most relevant model for this thesis is eventual consistency.

2.1.1 Strong consistency

A distributed system follows a strong consistency model if it is permanently consistent. More formally, this implies that upon an update completion, the most recent value must be returned [38].

This model usually relies on leader-based protocols and mimics a centralized system. However, such protocols exchange a considerable volume of messages and use locks to ensure consistency under concurrency, adding latency to each operation and decreasing the system's availability.

Strong consistency found its practicability in applications requiring referential integrity, such as RDBMSs. On the other hand, this model falls short for applications that require strict time constraints.

2.1.2 Eventual consistency

Saito and Saphiro [31] introduced eventual consistency (EC) as optimistic replication. It is a particular form of weak consistency, thus admitting a *inconsistency window* [38]. It represents the period when an observer cannot see an updated value. If there are no failures, its maximum length depends on communication delays, system load, and number of replicas involved. However, in practice, failures happen, and the length of the inconsistency window may be unpredictable, making it particularly hard to reason about the system's correctness.

Under this model, storage guarantees that if no new updates occur, all the replicas will eventually read the last updated value, i.e., an equivalent state.

This consistency model's most notable application is the domain name system (DNS) [38]. Generally, it fits use cases that can momentarily cope with inconsistency.

2.1.3 Causal consistency

Succinctly, this consistency model guarantees that writes on each replica are visible in order, for any reads which follow writes [5].

Therefore, on a replica i, if $a \rightarrow b$, according to *happens-before* [25], the remaining replicas can only observe b after a. Consequently, it preserves the order of operations and does not abdicate correctness.

This model is fundamental towards δ-CRDTs understanding, explored in depth in section 2.5. It ensures that all causal dependencies are fulfilled upon coordination between replicas, leveraging data integrity among systems using them.

2.2 CAP theorem

Shared-data systems have three desirable properties: consistency, availability, and partition tolerance. The CAP theorem [19] states that at any given moment, a system can only possess two of them.

A corollary from the theorem is that some trade-offs exist when designing such systems. These trade-offs have different formulations, considering the possible combinations of the abovementioned properties.

In real-world systems, sacrificing partition tolerance is pointless, as it defeats the point of distributed computing. That opens the question of prioritizing consistency + partition (CP) or availability + partition (AP). Given the context of this thesis, and as in many modern geo-replicated systems [38], the latter option was chosen.

2.3 Join-semilattice

The concept of a join-semilattice [12] is essential to understand how CRDTs operate.

A join-semilattice is a partial ordered set that admits a lower upper bound (LUB) for any nonempty finite subset, i.e., all pairs of elements in the set.

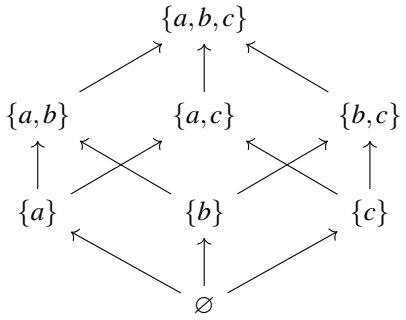
Definition 1 (Join-semilattice)**.** *The poset $(S, \leq)$ is a join-semilattice if any nonempty finite subset has a least upper bound:*

$$\{\forall x, y \in S \mid LUB(x, y) \neq \varnothing\}$$

When working with CRDTs, the lower upper bound results from the join operation $\sqcup$. Moreover, the join operation is an associative, commutative, and idempotent binary operation. Typically, these lattices are bounded by a bottom $\bot$, an element such that $\forall x \in S, \bot \leq x$.

2.3.1 Hasse diagram

Hasse diagrams are mathematical diagrams used to represent posets intuitively. Figure 2.1 shows a Hasse diagram and some examples of the LUBs. From the examples, it is visible that the least upper bound owns a geometric intuition – the first intersection point between a pair of elements while going up on the join-semilattice.



(a) Hasse diagram.

$$LUB(\{a,b,c\}, \varnothing) = \{a,b,c\}$$
$$LUB(\varnothing, \{a,b,c\}) = \{a,b,c\}$$
$$LUB(\{a\}, \{b,c\}) = \{a,b,c\}$$
$$LUB(\{a,b\}, \{c\}) = \{a,b,c\}$$
$$LUB(\{a\}, \{a\}) = \{a\}$$

(b) Examples of LUBs.

Figure 2.1: Join-semilattice of $\{a,b,c\}$ under union.

2.4 CRDTs

Conflict-free Replicated Data Types (CRDTs) were formally defined in 2011 by Shapiro et al. [33]. It pioneered a non-ad-hoc solution for state convergence over a network, using mathematical principles, such as monotonicity [21].

According to Preguiça et al. [30], a CRDT is a data type with a well-defined interface. CRDTs are designed to be replicated across multiple machines and may be modified concurrently. Moreover, they operate under strong eventual consistency (SEC). Replicas can diverge momentarily before reconciling, meaning that, under the CAP theorem, these objects favor availability and low latency over consistency.

Furthermore, CRDTs guarantee that when a pair of replicas see the same set of updates, they are in the same state. In this environment, convergence is obtained deterministically by adopting mathematical rules.

Simultaneously, CRDTs do not contact any other replica upon receiving an update. Instead, modifications are propagated to other replicas at a later stage, usually in periodic intervals.

The way these messages get constructed is known as synchronization models. Considering this, two main categories of CRDTs emerge: op-based and state-based, described in section 2.4.1 and section 2.4.2, respectively. A proof on the equivalence between these two models can be found in [33].

2.4.1 Op-based CRDTs

Operation-based CRDTs, as the name suggests, encode each modification as an operation. After a modification in one of the replicas, its operation disseminates over the network to the remaining replicas.

Two components are vital to make this possible: an *effector* and a *generator*. The former is a function whose input is an operation and modifies the state of a replica locally, accordingly. The latter is a function whose task is to encode an update into an operation ready to be sent over the network and consumed by some *effector*. It is a function free from side effects since it does not alter the object's internal state.

Nonetheless, the op-based approach is restrictive concerning the dissemination layer. It must ensure that operations get delivered to all replicas exactly once, albeit consensus is not required. On the other hand, the propagated messages are simple and do not grow in proportion to the CRDT state's size.

In this approach, updates must be commutative. Therefore, concurrent updates lead to the same outcome regardless of the execution order.

2.4.2 State-based CRDTs

Conversely, to the previous approach, state-based CRDTs achieve synchronization by transferring the state between replicas. Generally, this transfer occurs periodically.

A state-based CRDT is defined by the triple $(\mathscr{L}, \sqsubseteq, \sqcup)$. $\mathscr{L}$ is a join-semilattice, $\sqsubseteq$ is a partial order, and $\sqcup$ a binary join operation which computes the least upper bound for any two elements of $\mathscr{L}$.

State-based CRDTs use a set of mutators to describe updates over them. A *mutator* m is a function that accepts as input a state and produces a new state (eq. 2.1). Moreover, mutators are inflationary, i.e., the causal context is monotonically non-decreasing over $\mathscr{L}$ (eq. 2.2).

$$x' = m(x) \tag{2.1}$$

$$x \sqsubseteq x' \tag{2.2}$$

Recall, from section 2.3, that $\sqcup$ is a binary operation that must exhibit the following properties: associativity, commutativity, and idempotency. Compared with the op-based model, such properties impose fewer guarantees on the dissemination layer to reach convergence. In this scenario, the network can be completely unreliable. Hence, messages may be duplicated, dropped, and reordered.

However, a flaw regarding the messages' size becomes evident as they grow proportionally to the size of causal context, i.e., the state's size. Naturally, this is a compromise on the scalability and practicability of this solution.

2.5 Delta-CRDTs

To overcome the issue of sending entire state over the network and reduce message sizes, δ-CRDTs were introduced by Almeida et al. [2]. Essentially, instead of propagating the entire state between any two replicas, these objects ship the state differences since the previous exchange round.

In this model, δ-mutators are the algebraic object that describes updates. A δ-mutator m^δ is a function whose input is a state $x \in \mathscr{L}$, and the output is a δ-mutation $m^\delta(x)$ that represents a smaller state $\in \mathscr{L}$.

Furthermore, in δ-CRDTs, each standard mutator m, as in section 2.4.2, has a corresponding δ-mutator m^δ (eq. 2.3). Given this, δ-mutators are decoupled from the target state x'.

$$m(x) = x \sqcup m^\delta(x) \tag{2.3}$$

Another relevant definition introduced in [2] is the one below:

Definition 2 (δ-group)**.** *A δ-group is either a δ-mutation or the union of several δ-groups.*

The exchange of δ-groups between replicas describes the dissemination of updates. A buffered δ-group exists at each replica, which sends it periodically to the neighbor replicas. Upon receiving a δ-group, a replica alters its buffered δ-group by joining it with the received one, achieving *transitive propagation*.

2.5.1 Specification

A CRDT specification comprises its initial value, the methods provided by its interface, and the conflict resolution mechanism.

The specification of a δ-based Grow-only set (GSet) and a δ-based Add-wins set (AWSet) are depicted in Figure 2.2 and Figure 2.3, respectively. Notice that each method that alters the state, i.e., a standard mutator, has a corresponding δ-mutator method.

As the name suggests, a GSet only supports adding elements but does not support removal. Furthermore, we can see that the join of two sets is their union. Thus, the join-semilattice that

$$
\begin{aligned}
\mathsf{GSet}\langle E\rangle &= \mathscr{P}(E) \\
\bot &= \varnothing \\
\mathtt{add}_i(s,e) &= s \cup \{e\} \\
\mathtt{add}_i^\delta(s,e) &= \begin{cases} \{e\} & e \notin s \\ \bot & e \in s \end{cases} \\
\mathtt{elems}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}
$$

Figure 2.2: $\mathsf{GSet}\langle E\rangle$ specification for a replica $i \in \mathbb{I}$, where $\mathbb{I}$ is the domain of replicas [2].

models the state of this data type is the generalization, w.r.t. cardinality, of the one shown in Figure 2.1.

On the other hand, an AWSet extends GSet's functionality and allows the deletion of elements. It assigns a unique identifier to each insertion. This mechanism allows us to distinguish between insertions of the same element. Without them, a removed element would remain indefinitely removed without the possibility of being reinserted. Therefore, an element removal marks its associated identifier as removed.

One of the main challenges here is generating unique identifiers across multiple replicas.

$$
\begin{aligned}
\mathsf{AWSet}\langle E\rangle &= \mathscr{P}(E \times U) \times \mathscr{P}(U) \\
\bot &= (\varnothing, \varnothing) \\
\mathtt{add}_i((s,t),e) &= (s \cup \{(e, \mathtt{uid}())\}, t) \\
\mathtt{add}_i^\delta((s,t),e) &= \begin{cases} (\{(e,u)\}, \varnothing) & (e,u) \notin s \\ \bot & (e,u) \in s \end{cases} \\
\mathtt{rm}_i((s,t),e) &= (s, t \cup \{u \mid (e,u) \in s\}) \\
\mathtt{rm}_i^\delta((s,t),e) &= \begin{cases} \bot & (e,u) \notin s \\ (\varnothing, \{u\}) & (e,u) \in s \end{cases} \\
\mathtt{elems}(s) &= \{e \mid (e,u) \in s \wedge u \notin t\} \\
(s,t) \sqcup (s',t') &= (s \cup s', t \cup t')
\end{aligned}
$$

Figure 2.3: $\mathsf{AWSet}\langle E\rangle$ specification for a replica $i \in \mathbb{I}$, where $\mathbb{I}$ is the domain of replicas [2].

2.5.2 Anti-entropy

Similarly to the state-based model, the δ-based execution model works under an unreliable communication layer. However, while the state-based model ensures causal consistency by exchanging the entire state [11], the δ-based approach provides increased flexibility in this regard.

In this execution model, the anti-entropy algorithm can solely ensure state convergence, without necessarily providing causal consistency guarantees [2]. Furthermore, an update reaching all replicas is necessary for state convergence. Therefore, adopting a specific algorithm represents a trade-off between consistency and performance. In this context, the anti-entropy algorithm refers to the algorithm executed when replicas synchronize.

In [2], the authors present two different anti-entropy algorithms: (i) a basic version that only ensures eventual convergence, and (ii) a version that ensures causal consistency. The second version relies on the concept of δ-intervals. It also maps each replica to the last acknowledged update, thus being capable of constructing the needed δ-interval.

Definition 3 (δ-interval)**.** *For a replica i, a δ-interval $\Delta_i^{a,b}$ is a δ-group formed by joining the range of deltas δ_i^a, $\delta_i^{a+1}, \ldots, \delta_i^{b-1}$, computed from consecutive state transitions between states $x_i^a, x_i^{a+1}, \ldots, x_i^{b-1}$:*

$$\Delta_i^{a,b} = \bigsqcup \{\delta_i^k \mid a \leq k < b\}$$

Given this definition, it is necessary that any state merges between any two replicas use δ-intervals in order to attain causal consistency.

2.5.3 The issues with delta-based anti-entropy

Surprisingly, the synchronization algorithms presented in section 2.5.2, did not represent an improvement over the state-based synchronization. They have shown to be less performant than the state-based model by experiments [13], albeit not necessarily ensuring causal consistency. The leading cause of underperformance is the dissemination of redundant states over the network.

The first technique to tackle this issue was Δ-CRDTs [36]. Essentially, replicas exchange metadata about their versions, which causes δ-groups production to occur on the fly. Δ-CRDTs represent an improvement when such metadata is available. Otherwise, they must resort back to the classical state-based approach.

Enes et al. [13] identified two sources of inefficiencies in the aforementioned anti-entropy algorithms: (i) **back propagation of δ-groups** (BP), and (ii) **redundant state in received δ-groups** (RR).

The first, BP, occurs when a replica i receives a δ-group from another replica j. However, the incoming δ-group contains a δ-group originated in i. Naturally, this constitutes an inefficiency since it is known that i already updated its local state with that exact δ-group. The key to circumventing this problem lies in tracking the origin of each incoming δ-group, and not sending them back to their origin.

The other one, RR, happens when δ-groups are sent more than once to a neighbor replica. Thus, to avoid this, replicas must (i) keep track of the shipped δ-groups and their corresponding destinations and (ii) filter out propagated δ-groups upon synchronization. The application of this optimization depends on state decomposition, a topic explored in section 2.5.4.

2.5.4 Join decompositions and optimal deltas

In this subsection, we will explore the concept of state decomposition in state-based CRDTs. The target here is to obtain optimal δ-groups for size. This way, no redundancies are propagated during synchronization rounds between replicas. As a result, the messages transmitted over the network will be more concise, translating into smaller latencies.

We must introduce several mathematical definitions [12, 8] before deriving these optimal δ-groups. The intent here is to provide the background on the concept of *irredundant join decompositions* in join-semilattices.

Definition 4 (Join-irreducible state)**.** *$x \in \mathscr{L}$ is join-irreducible if x cannot result from the join of a finite set of states $F \subseteq \mathscr{L} \setminus \{x\}$:*

$$x = \bigsqcup F \implies x \in F$$

Remark. *The bottom $\bot$ is **never** a join-irreducible state. It is the join over an empty set $\bigsqcup \varnothing$.*

Hasse diagrams, e.g., shown in Figure 2.1a, provide a clear visual intuition on join-irreducible states, which contain precisely one link below.

Moreover, let $\mathscr{J}(\mathscr{L})$ be the set of all join-irreducible states over $\mathscr{L}$.

Definition 5 (Join decomposition)**.** *Given $x \in \mathscr{L}$, a set of join-irreducibles D is a join decomposition of x if its join produces x:*

$$D \subseteq \mathscr{J}(\mathscr{L}) \wedge \bigsqcup D = x$$

Definition 6 (Irreducible join decomposition)**.** *A join decomposition D is irreducible if no element in D is redundant:*

$$D' \subset D \implies \bigsqcup D' \sqsubset \bigsqcup D$$

For illustration, let $s = \{a,b,c\}$. Then the unique irreducible join-decomposition of s is $\{\{a\},\{b\},\{c\}\}$. It is a set of sets with a single, join-irreducible element that is a maximal below x. For given state $x \in \mathscr{L}$, $\Downarrow x$ (eq. 2.4) represents its decomposition.

$$\Downarrow x = \max\{r \in \mathscr{J}(\mathscr{L}) \mid r \sqsubseteq x\} \tag{2.4}$$

Furthermore, obtaining such decompositions depends on the data type. To the interested reader, we refer to the work of Enes et al. [13] (Appendix A and B), which demonstrates how to obtain those decompositions for a vast collection of CRDTs.

With the irredundant decomposition defined, we can derive optimal δ-groups. The function Δ (eq. 2.5) receives as input two states $a,b \in \mathscr{L}$, and produces the *difference* between a and b as a δ-group.

Joining the output of Δ with one of the input states is equivalent to joining its input states. As a consequence, each mutator m has a corresponding smallest δ-mutator m^δ (eq. 2.6), computed using Δ.

$$\Delta(a,b) = \bigsqcup\{y \in \Downarrow x \mid y \not\sqsubseteq b\} \tag{2.5}$$

$$m^{\delta}(x) = \Delta(m(x), x) \tag{2.6}$$

2.5.5 Revisiting anti-entropy

The Algorithm 2.1 presents the proposed synchronization from [13] as an improvement over the basic version in [2]. Although there are some transformations to incorporate the optimizations for BP and RR, both versions are similar.

Algorithm 2.1 Improved δ-based anti-entropy algorithm.

```
1:  input:
2:      n_i ∈ 𝒫(𝕀)                                   ▷ set of neighbors
3:  state:
4:      x_i ∈ 𝒟, x_i^0 = ⊥                           ▷ durable state
5:      B_i ∈ 𝒫(𝒟 × 𝕀), B_i^0 = ∅                    ▷ volatile state
6:  on operation_i(m^δ)
7:      δ = m^δ(x_i)                                 ▷ stage durable state
8:      store(δ, i)
9:  periodically synchronize
10:     for j ∈ n_i
11:         d = ⨆{s | ⟨s, o⟩ ∈ B_i ∧ o ≠ j}          ▷ filter out the δ-groups already sent to j
12:         send_{i,j}(delta, d)
13:     B_i' = ∅                                     ▷ volatile state cleanup
14: on receive_{i,j}(delta, d)
15:     d = Δ(d, x_i)                                ▷ optimal δ-group between received and durable states
16:     if d ≠ ⊥
17:         store(d, j)                              ▷ update durable state if d is inflatious
18: fn store(s, o)
19:     x_i' = x_i ⊔ s                               ▷ commit durable state
20:     B_i' = B_i ∪ {⟨s, o⟩}                        ▷ append and tag an update with its origin to volatile state
```

Every replica i has a set of neighbor replicas, durable and volatile states. The durable state x_i also called the local state, contains the replica's state per se. The algorithm assumes that it is persistent under crashes. On the contrary, the volatile state B_i does not have to be persistent. It accumulates incoming δ-groups issued on other replicas, mapping them to their origins.

When an update is issued at replica i, a δ-group δ is generated from a δ-mutator m^{δ} representing a modification. The next step is to call the `store` function. In `store`, two relevant things happen: (i) the local state x_i is modified, and (ii) some state s gets appended to volatile state B_i alongside its origin. In this circumstance, the origin is replica i.

The synchronization between replicas occurs periodically. The procedure iterates over the set of neighbor replicas. For each neighbor replica j, the δ-groups whose origin differs from j are joined together and shipped to j. Retaining only δ-groups that did not come from j avoids BP, described in section 2.5.2. The last step resets the volatile state B_i to the initial value $\varnothing$.

Upon reception of an update, the first step is to compute the optimal delta between what has been received d and the local state x_i. As specified in section 2.5.4, function Δ carries off this. The

following step invokes `store` if the output of Δ is not $\perp$. As a result, the incoming update strictly inflates the current state. Otherwise, the corresponding δ-group would be in one of both scenarios: (i) already propagated or (ii) present in the volatile state B_i not yet propagated. In either scenario, such an update is part of the local state. This way, the algorithm avoids RR, see section 2.5.2, which can occur in topologies containing cycles due to distinct paths between replicas.

Chapter 3

Related work

This chapter delves into some related work concerning state-based CRDTs. It explores several systems that are useful for the development of this thesis.

Section 3.1 provides a deep dive into rsync. In section 3.2, we look into the details of bloom filters, a space-efficient probabilistic set. To close the chapter, in section 3.3, we explore Merkle Search Trees, a novel data structure that encodes state-based CRDTs.

3.1 rsync

Rsync [35] is a protocol introduced by Andrew Tridgell and Paul Mackerras in 1996. They also released a Unix tool of the same name in the same year.

It is a protocol intended to synchronize files between two distinct computers. It assumes that both machines communicate over a slow connection, essentially "a low-bandwidth high-latency bi-directional communication link." Furthermore, it uses similarities between files to reduce the data transmitted over the network.

3.1.1 Algorithm

In order to execute rsync, some pre-conditions must hold. The algorithm assumes that there are two computers: α, the sender, and β, the receiver. These computers can access *similar* files A and B, respectively. As described above, it also assumes a slow connection between the sender and receiver.

The rsync algorithm comprises the following steps:

1. β splits the file B in a series of non-overlapping fixed blocks of size S bytes. The last block may be shorter than S bytes.
2. For each block, β computes a weak *rolling* checksum and a strong checksum.
3. β sends these checksums to α.

4. α searches through A to find all the blocks of size S bytes (at any offset) that have the same weak and strong checksum as one of the blocks of B. It only takes a single pass over A due to the property of the rolling checksum, described in subsection 3.1.2.

5. α sends β the sequence of instructions to reconstruct A. Each instruction is either a reference to a block of B or literal data when a section in A has no correspondence in B.

The algorithm only requires one communication round-trip minimising the impact of latency.

3.1.2 Rolling checksum

Understanding the concept of a rolling checksum is essential to understand the algorithm. It is the primary mechanism to detect similarities between the file versions on each machine efficiently. The adler-32 checksum inspired the algorithm conceived by the authors.

The following recurrence relations describe the vital property of this checksum:

$$a(k+1,l+1) = (a(k,l) - X_k + X_{l+1}) \bmod M$$

$$b(k+1,l+1) = (b(k,l) - (l-k+1)X_k + a(k+1,l+1)) \bmod M$$

From these relations, it is visible that computing the checksum of the bytes $X_{k+1} \ldots X_{l+1}$ is cheap if the checksum of $X_k \ldots X_l$ and the values of X_k and X_{l+1} are known. Meaning that there is very little computation to do at each point while scanning the files for differences, hence the *rolling* fashion, which resembles the behavior of a FIFO (first in, first out). At each step, it pops (or subtracts) X_k and pushes (or adds) X_{l+1} to the checksum value.

However, this checksum admits collisions. Therefore, it is only used as a first instance check for a match of two file chunks. In the case of a match, more computationally demanding checksum is computed to detect the presence of an actual match. This checksum is called *strong* checksum.

3.1.3 Discussion

Despite almost 30 years since its conception, rsync has proved that it is a reliable and efficient piece of software. It is used across a diverse range of applications [34], mainly synchronization of software repositories.

Concerning this thesis, the synergy between the rolling and strong checksums mechanism is an inspiration for efficiently detecting differences in the state replicated among distinct nodes.

3.2 Bloom filters

A Bloom filter [9] is a space-efficient data structure representing a probabilistic set. They were introduced in 1970 and are the most well-known approximate membership query (AMQ) filter.

When querying an element for membership on a bloom filter, two outcomes are possible: (i) the element is *possibly* in the set, or (ii) the element is not in the set. It means a bloom filter accepts

false positives but does not admit false negatives. Besides, it is possible to balance the probability of false positives with the filter's memory footprint.

3.2.1 Analysis

Bloom filters consist of a bit array of length m and k independent hash functions. Its interface is restricted relative to the interface of a set: it only supports insertion and membership querying. The time complexity of both operations is $O(k)$, which does not depend on the number of inserted elements.

Inserting an element into the filter signifies computing each hash function and obtaining a value that sets a particular bit; such a combination of bits functions as an element's *shadow*.

On the other side, querying for the presence of an element is very similar. Again, querying requires computing each hash function and obtaining a value that conveys a particular bit. Given the element's *shadow*, if any of the bits is not set, the filter does not contain the element. Otherwise, it is only possible to say that the filter *possibly* contains the element. A not inserted element may have a *shadow* whose all the bits are set, meaning the filter contains it, thus being a false positive.

The probability of false positives ε can be determined according to the expected number of distinct elements n contained in the filter. It assumes that hash functions select bits uniformly and independently for each element. Therefore, an estimation of the false positive rate is given by:

$$\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{3.1}$$

To interpret this equation, we will start from the inside to the outside. The value $1 - \frac{1}{m}$ represents the probability that a bit is not set. Raising this value to the power kn denotes the probability of a bit not being set after n insertions. Recall that each insertion depends on k distinct hash functions. From this point, inverting the probability and raising it to the power k represents the probability that all the bits on the subsequent insertion will be set, constituting a false positive.

Rearranging the equation 3.1 and using the limit definition of e, we obtain:

$$\varepsilon \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k} \tag{3.2}$$

Notice the $\frac{n}{m}$, representing the number of bits per element. Also, observe that both m and k must be positive integers, therefore, their optimal values w.r.t. to ε are given by:

$$m = \left\lceil -\frac{n \ln \varepsilon}{\ln^2 2} \right\rceil \tag{3.3}$$

$$k = \left\lceil -\frac{\ln \varepsilon}{\ln 2} \right\rceil = \lceil -\log_2 \varepsilon \rceil \tag{3.4}$$

With the number k defined, any operation will require k independent hash functions. There are several ways to obtain them. The most straightforward solution is to append k distinct suffixes to an element and hash each concatenation.

Another, more efficient approach, introduced by Kirsch and Mitzenmacher [22], only requires two independent uniform hash functions h_1 and h_2 to obtain k independent ones. The relation that determines which bits compose an element's *shadow* is the following:

$$g_i(x) = h_1(x) + ih_2(x) \bmod m, \forall i \in \{0, 1, \ldots, k-1\}$$

3.2.2 Discussion

Despite its reduced functionality compared to a set, bloom filters provide immense benefits with reduced memory usage by giving up on accuracy. Naturally, numerous variants emerged based on Bloom's work.

Quotient [29], and cuckoo [15] filters extended the functionality of bloom filters by providing the ability to delete elements. Both variants maintain a counter that keeps track of insertions and deletions. Xor filters [20] were developed to reduce the number of bits per element. Scalable bloom filters [1] allow for dynamic resizing of the filter while keeping a fixed bound on the probability of false positives.

In the context of this work, we use bloom filters as a primitive to synchronize pairs of replicas. Moreover, the inevitable presence of false positives does not guarantee full synchronization. Despite the work of Kiss et al., [23] to overcome this drawback – a filter without false positives over a finite universe of elements, results indicate that memory expenditure is considerably higher than bloom filters [26].

3.3 MSTs

A Merkle Search Tree (MST) [3] is a data structure presented by Aulovat and Taïani. Its goal is to provide an efficient anti-entropy mechanism for large-scale open networks, as in the number of network round-trips. The structure of an MST is the combination of a Merkle tree [28] and a B-tree [6], thus leveraging the best of both by inheriting their properties.

3.3.1 Motivation

In large-scale open networks, devices may enter and leave the network as they wish in distinct geographic locations. In this setting, network partitions are unavoidable. Such partitions are a source of churn, where tracking membership and causality are particularly challenging tasks. A straightforward solution would be using some form of logical clock, such as vector clocks [16, 27]. Nonetheless, there are relevant problems to consider.

The most concerning problem relates to the size of these clocks since this type of network is expected to be constituted by millions of devices. Such clocks are part of the protocol's metadata; thus, sometimes, metadata could be larger than the actual data. Naturally, this phenomenon would lead to a negative impact regarding bandwidth consumption. Also, depending on the rate at which

each device performs a new operation, some clock entries would get propagated often to other peers yet remain unchanged most of the time.

3.3.2 Properties

Due to the above construction, MSTs are a set over a totally ordered space $\mathbb{K}$. This idea resembles a database index, which provides fast lookup and maintains key ordering over a table.

Moreover, MSTs are robust to data corruption, employing a collision-resistant hash function. Such a function is essential in keeping the MST (probabilistically) balanced, which ensures a better worst-case time complexity for lookup. The number of leading zeros on its hash value determines which tree level an item is assigned. Thus, the hash function must project the inputs uniformly over a space whose size is a power of the branching factor B. An example of a hash function suitable for this context is SHA-512.

However, the real advantage revolves around the combination of these properties. The construction of an MST is deterministic and has a unique representation. The hash value at the tree's root, which is easily verifiable, determines the unique representation of the MST.

3.3.3 CRDTs and anti-entropy

Transforming an MST into a map where $\mathbb{K} \mapsto \mathbb{V}$ is attained by simply tagging each of its items with a value from a set of values $\mathbb{V}$. Building the appropriate $\mathbb{V}$ permits one to encode a CRDT's states, creating an MST as a state-based CRDT.

In this context, recalling the benefit of state-based CRDT not imposing constraints on the network layer an anti-entropy algorithm that handles the structure of the MST is all that is needed to attain the goal described above.

In [3], the authors introduce a causal consistent anti-entropy algorithm. The algorithm builds upon the causal consistent version discussed in section 2.5.2. Its main novelty lies in the transversal of the tree according to the missing block hashes between peers when the dissemination of updates occurs. The number of network round-trips required to synchronize states between replicas depends on the branching factor B. Hence, higher values of B lead to shallower trees, fewer round-trips, and larger messages.

3.3.4 Discussion

The algorithm mentioned in the previous section tries to reduce the number of required round-trips for state reconciliation when compared with other methods, namely the ones based on vector clocks, such as *Scuttlebutt* [37]. The results in [3] show that MSTs perform better in networks with a large number of devices, albeit only when the rate of events is moderate to low.

Such an environment aligns with several use cases, like a social media platform. One concrete example in the industry is Bluesky[1], the company that develops the AT protocol [24]. The AT protocol is a federated protocol built with social platforms in mind that uses MSTs.

[1] https://blueskyweb.xyz/

According to the protocol specification, a *repository*, or *repo*, provides *"self-authenticated storage for public contents."* It is an MST that stores all the public active content produced by a single user. The content is all publicly visible posts or interactions produced by a user. Essentially, the MST maps *record keys* to an IPFS CID [7], which can either represent the actual contents or a link to another *repository*. Pointing to another *repository* is the key to making a distributed storage system.

A similar solution to this problem has been developed [32]. It draws some inspiration from the operational model of blockchains and relies on existing infrastructure. However, they use logical clocks, which is unsuitable for large-scale open networks.

Regarding this thesis, we adapt MST's operational behaviors to build concise representations of δ-groups. Accordingly, given a δ-group, we retain properties such as deterministic representation and efficient verification.

Chapter 4

Problem statement

In this chapter, we examine the problem under investigation.

Section 4.1 initiates the chapter by exposing the problems concerning CRDTs, particularly state-based CRDTs. Section 4.2 and section 4.3 settle the hypothesis and the research questions regarding this work, respectively. Closing the chapter, section 4.4 addresses the methodology employed.

4.1 Open problems

Despite gaining traction in recent years in industry and academia, CRDTs still stand behind centralized approaches when the matter is performance, i.e., number of operations per second. Consequently, a central system may not be as fault-tolerant as a distributed one that employs CRDTs.

Moreover, in CRDTs *one size does not fit all*. A CRDT better suits a specific use case whose network has particular characteristics. For example, op-based approaches work better with unstructured data in which the state constantly changes, i.e., a high rate of updates. However, it demands some conditions from the network, as in section 2.4.1.

Since the formalization of CRDTs, op-based approaches have dominated the number of implementations. There is an evident similarity between an op-based message exchange and a remote procedure call (RPC) system. These latter systems have been around for decades, meaning they are well-studied and optimized. Traditionally, op-based outperforms state-based CRDTs, even in unfavorable conditions [13].

CRDTs are a hot research topic, and new state-based techniques, as the ones discussed in section 2.5, worked on narrowing the gap to op-based approaches.

4.2 Hypothesis

Hypothesis. *There is a state-based synchronization algorithm that (1) benefits from the similarity between states and (2) replicas do not have any a priori knowledge of any peer's state.*

The hypothesis starts by claiming that it is possible to achieve state-based synchronization without executing a costly full-state bidirectional exchange. For instance, Δ-CRDTs attain (1) by storing metadata associated with a data type. Such metadata gives knowledge about peers' states. However, it must be evicted after a network partition since it may be no longer valid. Thus, this thesis seeks to validate the hypothesis by further introducing the constraint enunciated by (2).

4.3 Research questions

We identified the following research questions with the intent of guiding our investigation.

RQ1. *How does CRDT similarity impact performance regarding state-based synchronization?*

CRDTs express a framework of mathematical rules to attain consitent replication. This way, one can expect that distinct replicas, holding the same data type, share some part of their state. Therefore, answering this research question helps prove part (1) of the hypothesis. Furthermore, it is necessary to specify how to determine a similarity value between two states.

RQ2. *What are, in the literature, the strategies employed for data synchronization?*

Data synchronization became a ubiquitous task with the advent of cloud technologies. Systems users became acquainted with accessing data on any device instead of carrying around physical drives. This way, studying which strategies exist in the domain of data synchronization is pivotal to understanding what functions best when operating with similar and distributed data. Such a study should help determine what is feasible regarding part (2) of the hypothesis.

RQ3. *How can we improve CRDT state-based synchronization by adapting the strategies found?*

This research question draws upon the analyses of RQ1 and RQ2 and works as the connector between both. Besides, answering it is decisive in accepting or rejecting the hypothesis. In this setting, if results indicate that state-based synchronization can be improved, we must demonstrate under which circumstances we gathered such results.

4.4 Methodology

The algorithms introduced in Chapter 5 and the simulator's implementation described in Chapter 6 comprise the main contributions of this thesis. The development of these contributions followed an iterative approach.

The first iteration included the development of (i) the baseline algorithm and (ii) a component responsible for extracting relevant metrics. In the second iteration, we developed the bucketing strategy following the divide and conquer paradigm. In the third iteration, we introduced the bloom algorithm as a complementary solution to the bucketing strategy. Despite its inaptness to the state-based CRDT synchronization problem, this method revealed encouraging results under circumstances unfavorable to the bucketing strategy. In the fourth and final iteration, we formulated bloom + bucketing. As the name suggests, it combines the previously developed methods.

Chapter 5

Efficient synchronization

In this chapter, we dive into several techniques and algorithms towards efficient state-based synchronization. We also discuss the implications and trade-offs of each algorithm proposed.

Section 5.1 starts by outlining the standard system model among all the algorithms. In section 5.2, we detail some inefficiencies regarding the art of state-based anti-entropy. To finish the chapter, in section 5.3, we present two novel techniques in the context of efficient state-based synchronization: (i) bucketing and (ii) bloom-based.

5.1 System Model

We assume that two replicas, α and β, hold a similar data type that intends to be synchronized and admits irredundant join-decompositions. Both α and β do not have any *a priori* knowledge of the other replica's state. α denotes the replica that takes the initiative to execute the algorithm.

A replica may undergo a failure but eventually recover. It also must not accept any operations that inflate its state while synchronizing. Still, operations can be buffered and applied later to keep availability.

We assume that a transmission layer exists and it is responsible for the network's topology. It also guarantees that messages eventually arrive at their intended destinations despite message loss, reordering, or duplication. Besides, messages are guaranteed to arrive with the correct content.

This way, α and β can establish a point-to-point bidirectional communication channel that permits asynchronous communication. This channel is the main bottleneck in the system due to low bandwidth and high latency; hence, the cost of local computation on a replica is negligible.

All the algorithms in this chapter assume the system model described in this section.

5.2 State-driven

As previously seen in section 2.5.4, the work of Enes et al. on join-decompositions permits the construction of optimal deltas, i.e., δ-groups that strictly inflate state using the function Δ. Assuming a pair of replicas α and β, Algorithm 5.1 details the state-driven synchronization procedure.

Algorithm 5.1 State-driven synchronization algorithm.

1: **input**:
2: $x_\alpha, x_\beta \in \mathscr{L}$ ▷ states of α and β
3: **on** `synchronize`
4: $\texttt{send}_{\alpha,\beta}(x_\alpha)$ ▷ α transmits its entire state to β
5: $d = \Delta_\beta(x_\alpha, x_\beta)$ ▷ compute the optimal δ-group d between x_α and x_β at β
6: $\texttt{send}_{\beta,\alpha}(d)$
7: $x'_\alpha = x_\alpha \sqcup_\alpha d$ ▷ join x_α with the received d at α
8: $x'_\beta = x_\beta \sqcup_\beta x_\alpha$ ▷ join x_β with x_α at β

Figure 5.1 provides a visual intuition of how the algorithm works. This method only takes a single round-trip over the network. It also guarantees that both replicas are synchronized, given that, during the whole process, no operations that inflate state occur.

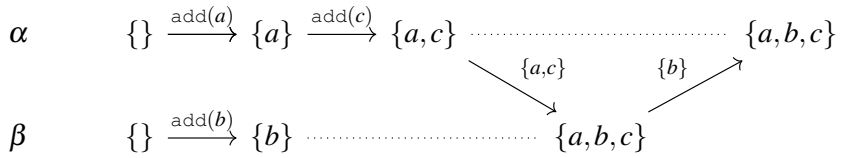


Figure 5.1: $\mathsf{GSet}\langle E\rangle$ synchronization with a state-driven approach.

This strategy constitutes an improvement over the initial state-based anti-entropy algorithm, as described in section 2.5.2. However, it is not ideal for the common scenario where replicas are similar, i.e., partially share state. Again, one of the peers must send its state entirely, only to a receiver to find that its state is similar; hence, the optimal delta is much smaller than what has been received. As noted in section 2.5.3, this approach may be intractable when dealing with large data volumes.

Given this scenario, this strategy defines the baseline for the experiments conducted throughout this work.

5.3 Digest-driven

The idea of a digest-driven algorithm to attain state synchronization is suggested in [14] by Enes et al. They also propose a (different) function Δ capable of deriving optimal δ-groups, although not generalized for all data types, that receives a state a and a digest of a state b as input. Moreover, such a digest is supposed to characterize a state in a much more concise format.

However, this shorter representation leads to a loss of information that requires at least one additional network operation beyond a bidirectional exchange. Therefore, a trade-off exists between the number of network operations and the total volume of data sent over the network.

Fortunately, it is possible to mitigate the impact caused by such a trade-off regarding the number of network operations using the pipelining technique. It is a widespread technique that allows one party to issue several requests without waiting for an acknowledgment from the other.

Several application layer protocols use it, such as HTTP [17] and SMTP [18]. In the context of digest-driven synchronization, pipelining allows for an algorithm that uses a minimum of three network operations.

Furthermore, observe that producing a digest for a given state depends on the data type. It is a problem that resembles the design of irredundant join-decompositions.

5.3.1 Bucketing

The general idea of bucketing is to aggregate join-decompositions into buckets and compute differences at a more granular level. A bucket is a δ-group, therefore, it is much smaller in size when compared with the entirety of a replica's state. Algorithm 5.2 displays how bucketing functions.

The algorithm starts (line 5) by assigning each join-decomposition to a specific bucket based on a shared property. In this context, we verify if irredundant join-decompositions share a common suffix on their hash values, hence a digest-driven approach. This suffix determines which bucket a join-decomposition is assigned (lines 20 and 21).

Replicas must also agree on a policy to compute hash values deterministically, under the penalty of computing incorrect deltas. Furthermore, we assume that the hash function used produces uniformly distributed values. This way, it is essential to select an appropriate hash function carefully. Modern collision-resistant hash functions, such as SHA-256, can satisfy these guarantees.

Furthermore, for bucketing to be practical, buckets should be expressed in a compressed format. Otherwise, this approach would behave exactly like the state-driven approach if buckets get transmitted as δ-groups.

A solution to this problem lies in mapping the bucket's contents to a unique digest. To achieve it, we use the Merkle "trick" (line 25), a name driven by its similarities with the operational behavior of internal nodes of a Merkle tree [28]. First, the bucket's join-decomposition hash values are sorted and concatenated. The next step is hashing the result of the concatenation, which produces a digest that uniquely identifies a bucket according to its contents.

The algorithm, depicted in Figure 5.2, functions in the following manner for synchronizing two replicas α and β:

This algorithm fails to synchronize when two distinct buckets generated in different replicas with the same index have colliding bucket hashes. In this scenario, it is impossible to determine non-matching buckets correctly. However, this is an improbable scenario.

This algorithm benefits mainly in scenarios where both replicas are similar. In such scenarios, the number of matching buckets, which only require the transmission of a small digest, is high. On the other hand, in the worst-case scenario, where all the buckets do not match, replicas exchange their entire states. Therefore, it is essential to choose the number of buckets adequately to avoid sending redundant states in excess. Another benefit is that all metadata can be generated on the fly without storing it, e.g., attached to the data type.

Algorithm 5.2 Bucketing synchronization algorithm.

input:
  $x_\alpha, x_\beta \in \mathscr{L}$ ▷ states of α and β
  $n \in \mathbb{N}$, with $n > 0$ ▷ number of buckets (determined by α)
on `synchronize`
  $B^{x_\alpha} = \texttt{buckets}_\alpha(x_\alpha)$ ▷ assign each irredundant join-decompositions to a bucket
  $d^{x_\alpha} = \{\texttt{digest}_\alpha(b) \mid b \in B^{x_\alpha}\}$ ▷ compute buckets' digests
  $\texttt{send}_{\alpha,\beta}(d^{x_\alpha})$
  $B^{x_\beta} = \texttt{buckets}_\beta(x_\beta)$
  $d^{x_\beta} = \{\texttt{digest}_\beta(b) \mid b \in B^{x_\beta}\}$
  $n = \{\langle i, B_i^{x_\beta}\rangle \mid i \in \{0,\ldots,n-1\} \wedge d_i^{x_\alpha} \neq d_i^{x_\beta}\}$ ▷ non-matching buckets and their indices
  $\texttt{send}_{\beta,\alpha}(n)$
  $g = \bigsqcup_\alpha \{\Delta_\alpha(B_i^{x_\beta}, B_i^{x_\alpha}) \mid \langle i, B_i^{x_\beta}\rangle \in n\}$ ▷ bucket-wise differences to be joined at α
  $g' = \bigsqcup_\alpha \{\Delta_\alpha(B_i^{x_\alpha}, B_i^{x_\beta}) \mid \langle i, B_i^{x_\beta}\rangle \in n\}$ ▷ bucket-wise differences to be joined at β
  $\texttt{send}_{\alpha,\beta}(g')$
  $x'_\alpha = x_\alpha \sqcup_\alpha g$
  $x'_\beta = x_\beta \sqcup_\beta g'$

fn `buckets`(x)
  $B^x = \{\bot \mid i \in \{0,\ldots,n-1\}\}$ ▷ n empty buckets, i.e., δ-groups
  for $d \in \Downarrow x$ ▷ loop through x's irredundant join-decompositions
    $i = h(d) \bmod n$
    $B_i^x = B_i^x \sqcup d$ ▷ join d to the i^{th} bucket
  return B^x
fn `digest`(b)
  $z = \{h(d) \mid d \in \Downarrow b\}$ ▷ hash each irredundant join-decompostion in bucket b
  $w = \texttt{concatenate}(\texttt{sort}(z))$ ▷ Merkle "trick"
  return $h(w)$ ▷ bucket's hash

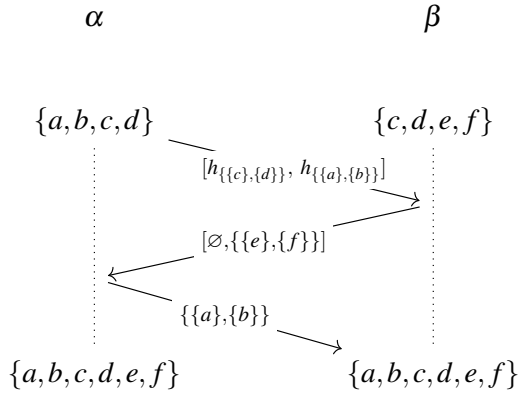


Figure 5.2: $\mathsf{GSet}\langle E\rangle$ synchronization with a bucketing approach.

This technique draws similarities with the rsync algorithm, described in section 3.1.1. In rsync, the input file is split into non-overlapping blocks of size S bytes. Likewise, in this context, the state gets divided into non-overlapping buckets of irredundant join-decompositions.

5.3.2 Bloom-based

Given the system model detailed above, any anti-entropy algorithm can be reduced to the set symmetric difference over replicas' irredundant join-decompositions. Ideally, a replica would only transfer a δ-group that strictly inflate its remote peer state, avoiding any redundancies. Nevertheless, a replica must start by transmitting some information about its current state, since replicas do not have any *a priori* knowledge of another's state.

The solution we propose to this problem is to use an Approximate Membership Query (AMQ) filter. In the context of this work, we will use bloom filters, described in section 3.2. As previously mentioned, this data structure relies on hash functions, hence being a digest-driven approach.

Bloom filters support insertion and membership querying, which are the only operations required by our algorithm. Concerning its implementation, such filters must operate deterministically across replicas, i.e., a given element must produce the same *shadow* despite the replica. Algorithm 5.3 illustrate how this method works.

Algorithm 5.3 Bloom-based synchronization algorithm.

1: **input**:
2: $x_\alpha, x_\beta \in \mathscr{L}$ ▷ states of α and β
3: $\varepsilon \in (0,1)$ ▷ probability of false positives
4: **on** `synchronize`
5: $f^{x_\alpha} = \texttt{buildfilter}_\alpha(x_\alpha)$
6: $\texttt{send}_{\alpha,\beta}(f^{x_\alpha})$
7: $u = \bigsqcup_\beta \{y \mid y \in \Downarrow x_\beta \wedge y \in f^{x_\alpha}\}$ ▷ *possibly* common irredundant join-decompositions
8: $v = \bigsqcup_\beta \{y \mid y \in \Downarrow x_\beta \wedge y \notin f^{x_\alpha}\}$ ▷ v strictly inflates x_α
9: $f^u = \texttt{buildfilter}_\beta(u)$
10: $\texttt{send}_{\beta,\alpha}(f^u, v)$ ▷ pipelining
11: $v' = \bigsqcup_\alpha \{y \mid y \in \Downarrow x_\alpha \wedge y \notin f^u\}$ ▷ v' strictly inflates x_β
12: $\texttt{send}_{\alpha,\beta}(v')$
13: $x'_\alpha = x_\alpha \sqcup_\alpha v$ ▷ x'_α and x'_β are not guaranteed to be equivalent
14: $x'_\beta = x_\beta \sqcup_\beta v'$

15: **fn** `buildfilter`(x)
16: $f = \texttt{bloom}(|\Downarrow x|, \varepsilon)$ ▷ empty bloom filter
17: **for** $d \in \Downarrow x$
18: $f = f \cup d$ ▷ insert d into f
19: **return** f ▷ bloom filter of irredundant join-decompositions

Due to its space-efficiency properties, bloom filters are a data structure that works well in this context, where the links between may be thin and slow. It efficiently detects true negative join-decompositions, i.e., δ-groups that will inflate the remote peer's state, akin to rsync's rolling

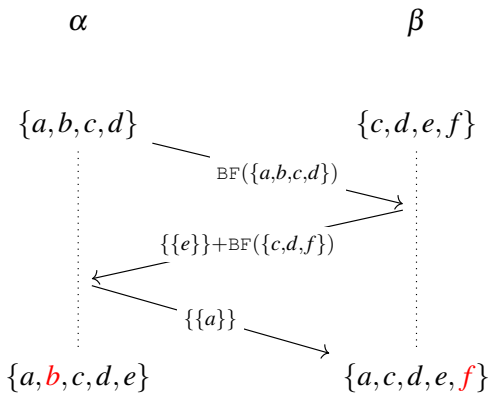


Figure 5.3: GSet⟨E⟩ synchronization with a bloom-based approach.

checksum (cf. section 3.1.2). Nonetheless, the presence of false positives, marked in red in Figure 5.3, prevents it from guaranteeing that both replicas will be synchronized.

5.3.3 Bloom-based + Bucketing

The algorithm described in this section results from combining the techniques detailed in the sections above. Therefore, it can be considered a digest-driven approach as well. Algorithm 5.4 details how this method unfolds.

Naturally, this algorithm is more complex. Even though it leverages pipelining, debriefed in section 5.3, it performs four network operations, illustrated in Figure 5.4, one more than the other algorithms discussed in this chapter.

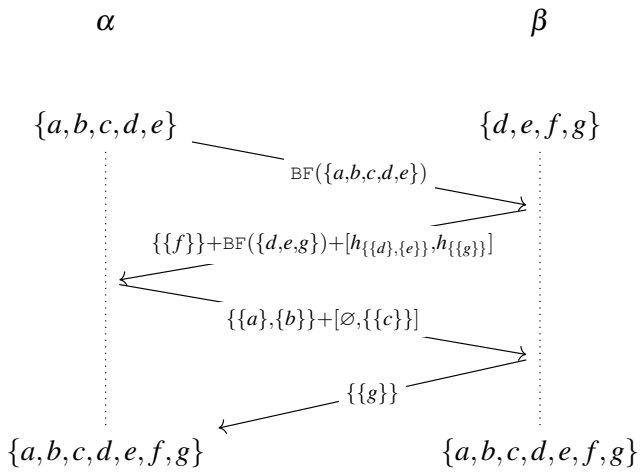


Figure 5.4: GSet⟨E⟩ synchronization with bloom-based + bucketing approach.

Nonetheless, this algorithm promotes the best of both approaches that compose it. It starts by exchanging bloom filters to efficiently detect which irredundant join-decompositions inflate the peer's state. After that, operating over a diminished fraction of state, the bucketing technique filters out any false positives and guarantees that both replicas synchronize. Observe that these

Algorithm 5.4 Bloom + bucketing synchronization algorithm.

1: **input**:
2: $x_\alpha, x_\beta \in \mathscr{L}$ ▷ states of α and β
3: $n \in \mathbb{N}$, with $n > 0$ ▷ number of buckets (determined by α)
4: $\varepsilon \in (0,1)$ ▷ probability of false positives
5: **on** `synchronize`
6: $f^{x_\alpha} = \texttt{buildfilter}_\alpha(x_\alpha)$
7: $\texttt{send}_{\alpha,\beta}(f^{x_\alpha})$
8: $u = \bigsqcup_\beta \{y \mid y \in \Downarrow x_\beta \wedge y \in f^{x_\alpha}\}$ ▷ u contains false positives
9: $v = \bigsqcup_\beta \{y \mid y \in \Downarrow x_\beta \wedge y \notin f^{x_\alpha}\}$ ▷ v strictly inflates x_α
10: $f^u = \texttt{buildfilter}_\beta(u)$
11: $B^u = \texttt{buckets}_\beta(u)$
12: $d^u = \{\texttt{digest}_\beta(b) \mid b \in B^u\}$
13: $\texttt{send}_{\beta,\alpha}(f^u, d^u, v)$ ▷ pipelining
14: $u' = \bigsqcup_\alpha \{y \mid y \in \Downarrow x_\alpha \wedge y \in f^u\}$ ▷ u' contains false positives
15: $v' = \bigsqcup_\alpha \{y \mid y \in \Downarrow x_\alpha \wedge y \notin f^u\}$ ▷ v' strictly inflates x_β
16: $B^{u'} = \texttt{buckets}_\alpha(u')$
17: $d^{u'} = \{\texttt{digest}_\alpha(b) \mid b \in B^{u'}\}$
18: $n = \{\langle i, B_i^{u'}\rangle \mid i \in \{0,\ldots,n-1\} \wedge d_i^u \neq d_i^{u'}\}$ ▷ non-matching buckets
19: $\texttt{send}_{\alpha,\beta}(n, v')$ ▷ pipelining
20: $g = \bigsqcup_\beta \{\Delta_\beta(B_i^u, B_i^{u'}) \mid \langle i, B_i^{u'}\rangle \in n\}$ ▷ g includes redundancy, i.e., state also in x_α
21: $g' = \bigsqcup_\beta \{\Delta_\beta(B_i^{u'}, B_i^u) \mid \langle i, B_i^{u'}\rangle \in n\}$ ▷ g' includes redundancy, i.e., state also in x_β
22: $\texttt{send}_{\beta,\alpha}(g)$
23: $x'_\alpha = x_\alpha \sqcup_\alpha v \sqcup_\alpha g$ ▷ join x_α with v (bloom-based) and g (bucketing) at α
24: $x'_\beta = x_\beta \sqcup_\beta v' \sqcup_\beta g'$ ▷ join x_β with v' (bloom-based) and g' (bucketing) at β

ideas have correspondences with the weak and strong checksum mechanisms present in the rsync's algorithm.

Chapter 6

Implementation

This chapter details the main aspects and components of our synchronization simulator, written in Rust. All code artifacts presented are publicly available[1] under the MIT License.

Some of the advancements highlighted in Chapter 2 are still pretty recent, particularly join-decompositions. To our knowledge, [13] presents the only implementation that leverages join-decompositions. However, unlike ours, it does not offer an interface that allows clients to interact with them. Besides, our implementation also supports δ-mutators, typically as a return value upon mutation. Therefore, in this chapter, we will also suggest a set of interfaces (or *traits*) with the intent of supplying a foundation for future implementations of state-based CRDTs.

Section 6.1 starts by introducing the implemented data types. Section 6.2 details how support for join-decompositions is achieved. Before closing the chapter, section 6.3 describes the implementation of the algorithms previously introduced, without depending on a particular data-type.

6.1 Data types

To assess the correctness of the proposed algorithms, we implemented a GSet and an AWSet following the specifications in section 2.5.1. Despite their simplicity, these data types support join-decompositions. Furthermore, they are commonly composed to form more complex state-based CRDTs. Their structure and interfaces are shown in Listing 6.1 and Listing 6.2, respectively.

Listing 6.1 Condensed implementation of `GSet<T>`.

```
1  pub struct GSet<T> {
2    inserted: HashSet<T>,
3  }
4
5  impl<T> GSet<T> {
6    pub fn new() -> Self;
7    pub fn elements(&self) -> Elements<'_, T>;
8    pub fn insert(&self, value: T) -> Self;
9  }
```

[1] https://github.com/mbrdg/xp

A `GSet` is essentially a wrapper around Rust's `HashSet`, from the standard library, which restricts its interface to only support insertion. The method `new` creates an empty set, which represents the bottom $\bot$. Notice the return type `Self` on `insert` that represents a δ-mutator. Furthermore, the method `elements` enables clients to iterate over the set's elements lazily, i.e., upon their request.

Listing 6.2 Condensed implementation of `AWSet<T>`.

```
1  pub struct AWSet<T> {
2    inserted: HashMap<u64, T>,
3    removed: HashSet<u64>,
4  }
5
6  impl<T> AWSet<T> {
7    pub fn new() -> Self;
8    fn uid(&self) -> u64;
9    pub fn elements(&self) -> Elements<'_, T>;
10   pub fn insert(&self, value: T) -> Self;
11   pub fn remove(&self, value: &T) -> Self;
12 }
```

At first sight, an `AWSet` is more complex than a `GSet` due to its extended interface. Upon insertion, a unique identifier tags the received element, forming a key-value pair that gets inserted into an `HashMap`. Conversely, upon deletion, all the unique identifiers associated with the received value get marked as deleted, i.e., inserted into the removed set. Like `GSet`, `AWSet` also has the methods `new` and `elements` which accomplish the same purposes.

Moreover, the generation of unique identifiers is a challenging task to achieve across multiple replicas. In this setting, assuming the usage of collision-resistant hash functions (cf. section 5.3.1), a straightforward solution is assigning an element's hash as its unique identifier.

6.2 Join-decompositions

Data structures provide support for join-decompositions through the `Decompose` trait. A *trait* allows different types to share behavior in Rust. This way, it is possible to decouple a synchronization algorithm from concrete data types and instead rely on any data type capable of ensuring such a contract. Such a mechanism reduces code redundancy while the language's type system ensures that the type provided to the algorithm is always correct.

The most important *trait*, in the context of this work, is `Decompose`, shown in Listing 6.3. `Decompose` enables state-based CRDTs to produce and consume join-decompositions.

Notice the presence of `type Decomposition` in line 2. This *associated type* represents any join-decomposition, i.e., a δ-group. Usually, implementations set it to `Self` – the type implementing the trait, due to the definition of a δ-group (cf. section 2.5). Albeit not strictly enforced, this aspect has the advantage of being consistent, since clients only have to deal with a single type.

Listing 6.3 `Decompose` *trait*.

```
1 pub trait Decompose {
2   type Decomposition;
3
4   fn split(&self) -> Vec<Self::Decomposition>;
5   fn join(&mut self, deltas: Vec<Self::Decomposition>);
6   fn difference(&self, other: &Self::Decomposition) -> Self::Decomposition;
7 }
```

As the name suggests, the method `split`, in line 4, splits the state into a set of irredundant join-decompositions. Notice, that it does not mutate `self`. Moreover, it returns join-decompositions *owned* by the caller, which requires cloning `self`'s state.

The method `join`, in line 5, consumes a set of join-decompositions and merges each one into `self`. This method is zero-copy, as `self` becomes the *owner* of `deltas`' content due to move semantics.

The last method `difference`, in line 6, is equivalent to $\Delta(a,b)$, detailed in section 2.5.4. It yields an optimal δ-group, *owned* by the caller, from `self` that strictly inflates `other`'s state.

In section 5.3, we presented several digest-driven techniques for state synchronization. However, these techniques are not exclusively dependent on data types supporting join-decompositions. There is the additional condition of an irredundant join-decomposition being able to produce a hash value.

Listing 6.4 `Extract` *trait*.

```
1 pub trait Extract {
2   type Item: Hash;
3
4   fn extract(&self) -> Self::Item;
5 }
```

Therefore, to match such a condition, we propose the *trait* `Extract`, shown in Listing 6.4. `Extract` only contains a single required method of the same name. As its name suggests, it aims to extract an item from an irredundant join-decomposition.

Observe `type Item: Hash`, in line 2, an *associated type* which tells that `Item` has a bound on `Hash`. This *trait*, defined in Rust's standard library, should be implemented by any type that can produce a hash value, i.e., hashable. Hashable types comprise strings, integral types, or even user-defined types – commonly employed by CRDTs.

Moreover, notice that `extract` does not mutate `self` while returning an `Item` *owned* by the caller, implying copying some state. Despite that, the caller will be free to process the item as he wishes.

6.3 Algorithms and telemetry

An additional concern with the development of out simulator is the implementation of different algorithms. Ideally, an algorithms should have an uniform interface across them that accepts any suitable data type. Therefore, we introduced the *trait* `Algorithm`, see Listing 6.5, that materializes a *strategy* pattern. This way, clients can opt for a *strategy*, i.e., an algorithm that better suits their needs.

Listing 6.5 `Algorithm` *trait*.

```
pub trait Algorithm<T> {
  type Tracker: Telemetry;

  fn sync(&self, alpha: &mut T, beta: &mut T, tracker: &mut Self::Tracker);
}
```

`Algorithm` only contains a single required method `sync`. As its name suggest, it synchronizes replicas `alpha` and `beta` whose type is `T`. Even though this is a generic type, implementations can impose constraints on `T`.

Moreover, notice the presence of `type Tracker`. This *associated type* allows algorithms to inject any structure that implements the *trait* `Telemetry`, illustrated in Listing 6.6. We will not discuss the details about the `Telemetry` *trait* since they are not relevant to the scope of this work.

Listing 6.6 `Telemetry` *trait*.

```
pub trait Telemetry {
  type Event;

  fn register(&mut self, event: Self::Event);
  fn events(&self) -> &Vec<Self::Event>;
  // more methods...
}
```

Given this, each algorithm is capable of capturing events. These events are later processed to obtain relevant metrics and other information.

In the context of this thesis, we implemented the algorithms presented in Chapter 5: (i) baseline, the state-driven approach; (ii) bucketing; (iii) bloom, which cannot guarantee full synchronization, and (iv) bloom + bucketing.

Chapter 7

Evaluation

In this chapter, we evaluate the methods presented in Chapter 5 and validate the hypothesis outlined in section 4.2.

Section 7.1 describes the environment in which we conducted the experiments. Section 7.2.1 and section 7.2.2 unveil the results of such experiments. Section 7.3 closes the chapter by digesting our findings and hypothesis validation.

7.1 Environment

To assess the validity of the algorithms introduced, we conducted a suite of controlled analyses. These analyses were performed with the help of our developed simulator.

We have tested all algorithms with several configurations that extract different trends according to each evaluation scenario. Exception made to bloom-based approach, which was left out of any tests since it does not fulfill the requirement of ensuring full-state synchronization, as discussed in section 5.3.2.

Concerning the bucketing method (Bu), detailed in section 5.3.1, our experiments concentrated on varying the number of buckets. The number of buckets B (eq. 7.1) is given by a load factor f_{ld} and the number of irredundant join-decompositions that compose α – the replica that takes the initiative over the synchronization procedure.

$$B = \lfloor |\Downarrow x_\alpha| \cdot f_{ld} \rfloor, \text{with } f_{ld} > 0 \tag{7.1}$$

Additionally, let X be the number of join-decompositions per bucket, then the expected value of X, $\mathbb{E}[X]$, is given by:

$$\mathbb{E}[X] = {f_{ld}}^{-1}$$

For illustration, if $f_{ld} = 5$, then $\mathbb{E}[X] = 0.2$; consequently, buckets will be sparse as most will be empty. This scenario results in more buckets matching, but it demands more metadata. On the

opposite side, if $f_{ld} = 0.2$, then $\mathbb{E}[X] = 5$, thus buckets will be denser, requiring less metadata but at the cost of a decreased likelihood of matches.

Regarding the more complex bloom + bucketing strategy (BlBu), described in section 5.3.3, we shifted the focus of our experiment towards varying the probability of false positives ε. Therefore, we settled the load factor f_{ld} parameter to a group of moderate values.

Additionally, we submitted both GSets and AWSets to our suite of analyses. Each experiment generates a pair of replicas α and β with the same cardinality and a given similarity degree.

In this setting, similarity s (eq. 7.2) is a metric computed from the Jaccard index between the sets of irredundant join-decompositions of α and β. It means that $s \in [0,1]$ and where $s = 0$ conveys disjoint sets.

$$s = J(\Downarrow x_\alpha, \Downarrow x_\beta) \tag{7.2}$$

Moreover, when generating AWSets, the removal of items occurs based on a provided likelihood. We set such a likelihood in our experiments to 0.2, or 20%. The motivation for the choice of this value comes from [10]. To the curious reader, it shows that between 2016 and 2018, roughly 15% to 20% of fact-checked news posts on Reddit got deleted.

In summary, we generate GSets with distinct 100,000 items and AWSets with 20,000 items, from which 20% get removed. In this context, items are strings whose size is uniformly distributed over the interval $[5,80]$.

7.2 Results

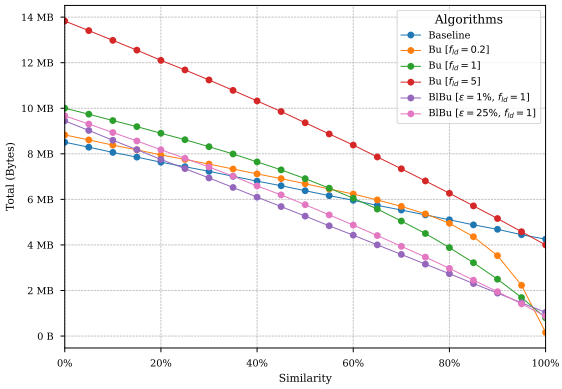
This section displays the results obtained throughout our experiments. For each data type, we performed a transmission and a time analysis.

The former comprises the collection of three metrics: (i) metadata, i.e., data associated with the algorithm itself, such as digests; (ii) redundancy, i.e., join-decompositions whose transmission was not necessary because they were already present at both replicas; and (iii) the total transmitted in bytes. Given these metrics, we also analyzed the ratio that metadata and redundancy represent against the total transmitted.

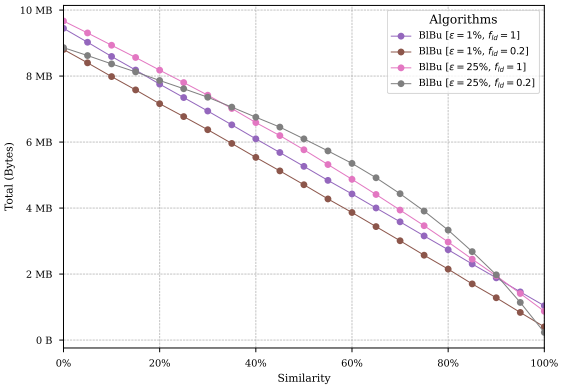
In the latter analysis, we estimate the time required to complete the synchronization procedure by simulating symmetric and asymmetric communication channels. In this context, upload (up) and download (down) express the bandwidth $\alpha \to \beta$ and $\beta \to \alpha$, respectively. Note that, as per our system model, described in section 5.1, the time spent on local computation is negligible. Consequently, the values shown only concern the time spent communicating between the replicas.

7.2.1 GSet

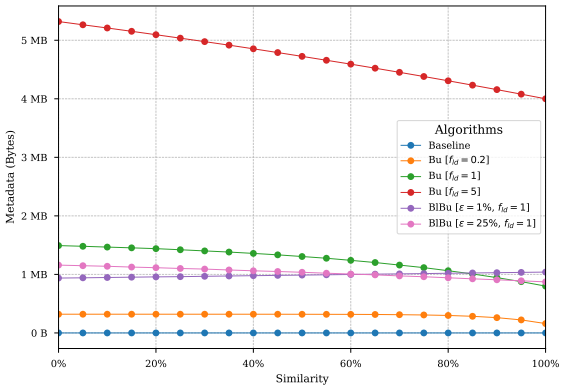
Figure 7.1 shows the transmission analysis results for GSets. In these experiments, we vary the similarity ratio between 0 and 1 with a step of 0.05 (5%). Table 7.1 and Table 7.2 show the rate of metadata and redundancy transmitted against the total transmitted, respectively.



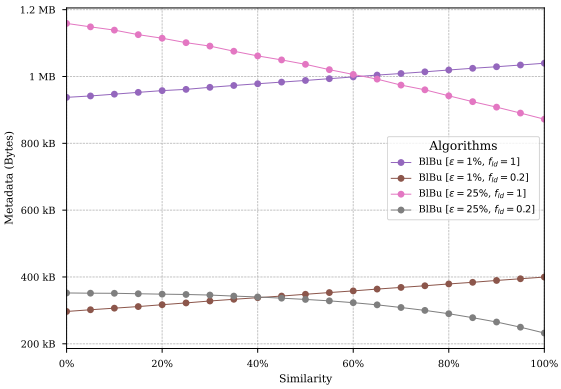
(a) General comparison of transmitted total bytes.



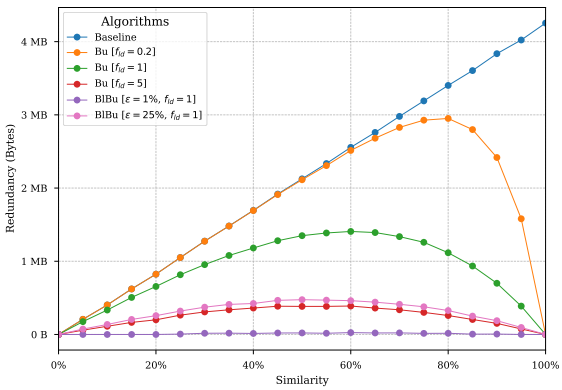
(b) BlBu comparison of transmitted total bytes.



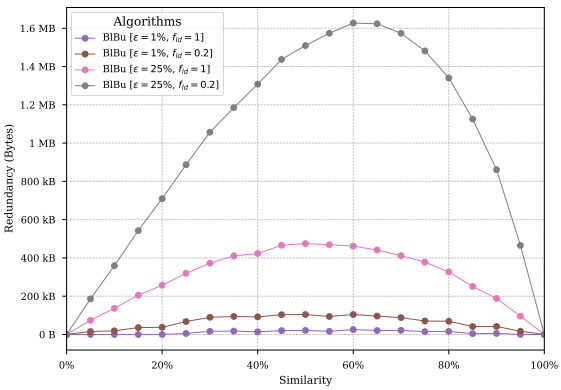
(c) General comparison of transmitted metadata.



(d) BlBu comparison of transmitted metadata.



(e) General comparison of transmitted redundancy.



(f) BlBu comparison of transmitted redundancy.

Figure 7.1: Transmission analysis w.r.t. similarity between a pair of GSets.

Table 7.1: Ratio of metadata sent between GSets for different similarities.

Algorithm	**0%**	**25%**	**50%**	**75%**	**90%**	**95%**	**100%**
Baseline	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Bu [$f_{ld} = 0.2$]	3.6%	4.1%	4.8%	5.7%	7.4%	10.0%	100.0%
Bu [$f_{ld} = 1$]	14.9%	16.5%	18.9%	24.8%	37.9%	51.9%	100.0%
Bu [$f_{ld} = 5$]	38.5%	43.1%	50.5%	64.3%	80.6%	89.1%	100.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 1$]	9.9%	13.1%	18.8%	32.1%	54.6%	70.9%	100.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 0.2$]	3.4%	4.8%	7.4%	14.5%	30.4%	47.1%	100.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 1$]	12.0%	14.1%	18.0%	27.7%	46.6%	63.1%	100.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 0.2$]	4.0%	4.6%	5.5%	7.7%	13.4%	21.9%	100.0%

The most notorious observation is that the Baseline approach performs better as the dissimilarity increases. These circumstances are favorable to a state-driven approach because redundancy decreases. Besides, unlike digest-driven approaches, note that the baseline requires no extra metadata.

Conversely, when similarity increases, and replicas are almost identical, it is possible to observe that digest-driven approaches perform better, albeit with two caveats. In these scenarios, digest-driven approaches only exchange metadata, avoiding the expensive transmission of redundant states.

The first and most evident of such caveats concerns the choice of a load factor f_{ld}. Such a choice imposes a trade-off between metadata and redundancy – a phenomenon we have anticipated in section 5.3.1. According to the experiments, a $f_{ld} = 1.0$ exhibits a smooth trend, thereby, providing a nice balance given the trade-off above.

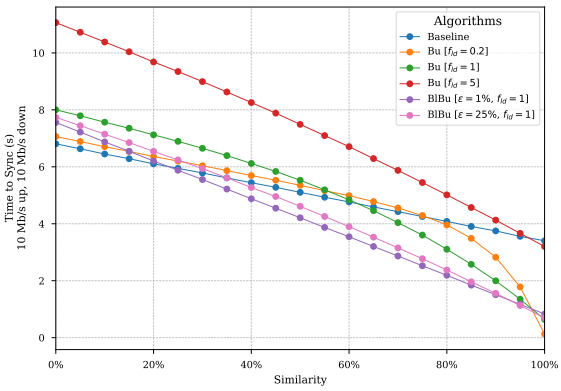
The second caveat lies in choosing a false positive rate ε. Again, this choice imposes a trade-off between metadata and redundancy, but this time, in two formats: (i) concerning the size of the filters themselves, and (ii) the effect it has over the bucketing stage of BlBu (cf. section 5.3.3). A smaller ε introduces more metadata due to the larger bloom filters. Nonetheless, the cost of transmitting more metadata gets amortized by shipping a less redundant state. An $\varepsilon = 1\%$ yields an almost optimal ideal performance regarding the redundancy domain.

Table 7.2: Ratio of redundancy sent for different similarities between GSets.

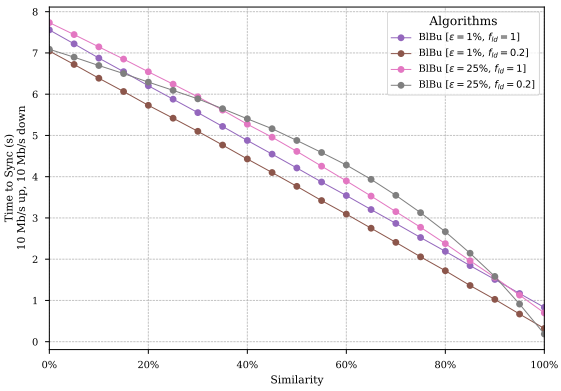
Algorithm	**0%**	**25%**	**50%**	**75%**	**90%**	**95%**	**100%**
Baseline	0.0%	14.1%	33.3%	60.0%	81.8%	90.4%	100.0%
Bu [$f_{ld} = 0.2$]	0.0%	13.5%	31.6%	54.6%	68.5%	70.9%	0.0%
Bu [$f_{ld} = 1$]	0.0%	9.5%	19.5%	28.0%	28.0%	22.9%	0.0%
Bu [$f_{ld} = 5$]	0.0%	2.3%	4.1%	4.4%	2.9%	1.6%	0.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 1$]	0.0%	0.1%	0.4%	0.5%	0.3%	0.0%	0.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 0.2$]	0.0%	1.0%	2.2%	2.7%	3.3%	2.0%	0.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 1$]	0.0%	4.1%	8.2%	10.9%	9.7%	6.8%	0.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 0.2$]	0.0%	11.6%	24.8%	37.9%	43.5%	40.8%	0.0%

Furthermore, lower values of ε positively affect the subsequent bucketing stage. After exchanging bloom filters, replicas operate over δ-groups with a high degree of similarity. In this setting, the similarity between these δ-groups is about $1-\varepsilon$, i.e., almost identical. According to our tests, denser buckets, i.e., $f_{ld} < 1$, obtain the best results in highly similar environments, where $s \geq 0.9$.

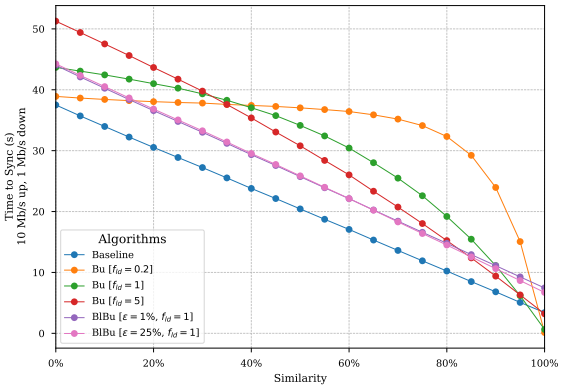
Figure 7.2 shows the time analysis results for GSets. Again, like the transmission analysis above, we vary the similarity ratio between 0 and 1 with a step of 0.05 (5%).



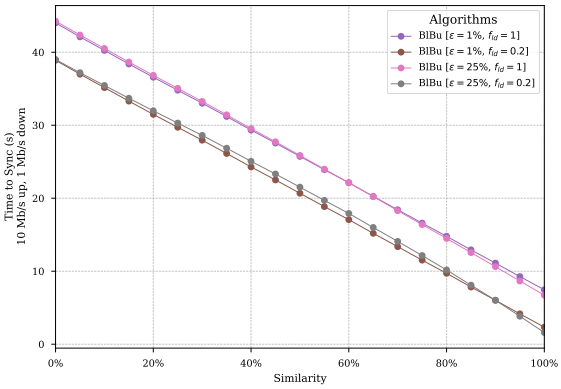
(a) Symmetric channel general comparison.



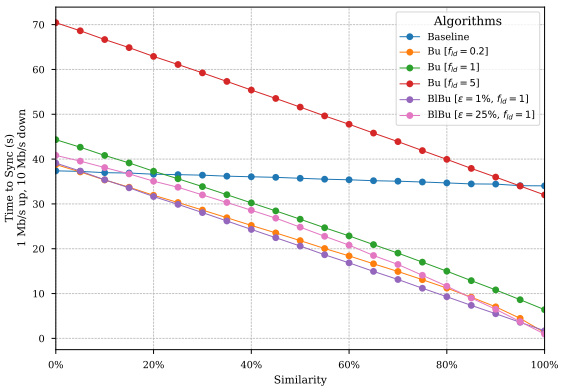
(b) Symmetric channel BlBu comparison.



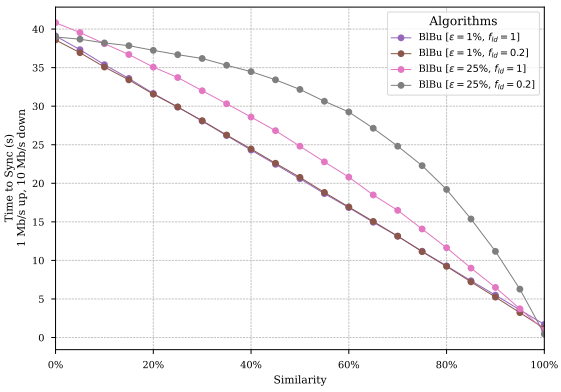
(c) Upload skewed channel general comparison.



(d) Upload skewed channel BlBu comparison.



(e) Download skewed channel general comparison.



(f) Download skewed channel BlBu comparison.

Figure 7.2: Impact of communication channel's bandwidth on the time to synchronize GSets.

The key observations here are rather general and do not revolve around a specific algorithm. As expected, Figure 7.2a is similar to Figure 7.1a. The interest here lies in the asymmetric communication channels. Observe that a relation exists between transmission metrics, i.e., metadata and redundancy, and the communication channel's skewness.

When the communication channel is upload skewed, as in Figure 7.2c, it harms digest-driven approaches, especially algorithms showing a higher metadata transmission ratio. These algorithms require both channel ends to transmit some metadata. Accordingly, part of the transmission takes longer because it happens over a thinner link. Naturally, the baseline, i.e., the state-driven strategy, presents the best result in such circumstances.

On the contrary, when the channel is download skewed, as depicted in Figure 7.2e, digest-driven approaches take the benefit of avoiding transmitting redundant states over a thin channel. Notice that the baseline takes roughly 35 seconds regardless of the similarity between replicas. Compared with an upload skewed channel and for high degrees of similarity, the baseline takes about ten times longer to synchronize. In contrast, most of the digest-driven strategies' times are on par.

Additionally, note that BlBu comparisons over asymmetric channels, shown in Figure 7.2d and Figure 7.2f, confirm this relation described above.

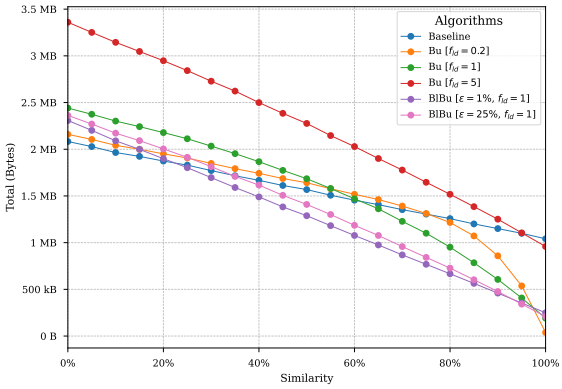
7.2.2 AWSet

Figure 7.3 and Figure 7.4 present the transmission and time analysis, respectively, for AWSets. Like in the experiments made with GSets, we vary the similarity ratio between 0 and 1 with a step of 0.05 (5%). Table 7.3 and Table 7.4 exhibit the rate of metadata and redundancy transmitted against the total transmitted, respectively.

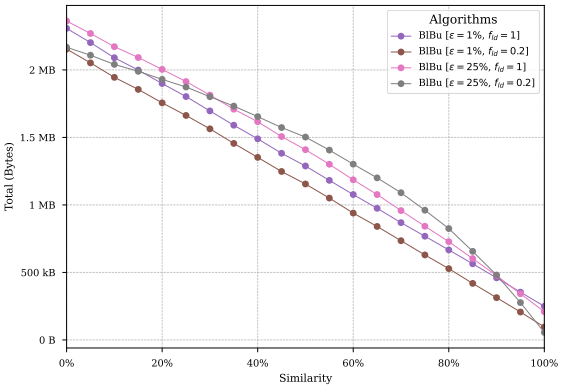
Table 7.3: Ratio of metadata sent for different similarities between AWSets.

Algorithm	**0%**	**25%**	**50%**	**75%**	**90%**	**95%**	**100%**
Baseline	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Bu [$f_{ld} = 0.2$]	3.6%	4.0%	4.7%	5.6%	7.3%	9.9%	100.0%
Bu [$f_{ld} = 1$]	14.7%	16.1%	18.6%	24.4%	37.5%	51.6%	100.0%
Bu [$f_{ld} = 5$]	38.0%	42.5%	49.7%	64.1%	80.0%	88.6%	100.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 1$]	9.7%	12.8%	18.4%	31.8%	54.0%	70.2%	100.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 0.2$]	3.3%	4.6%	7.2%	14.3%	30.0%	45.9%	100.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 1$]	11.8%	13.8%	17.6%	27.4%	46.0%	62.5%	100.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 0.2$]	3.9%	4.4%	5.3%	7.5%	13.3%	21.6%	100.0%

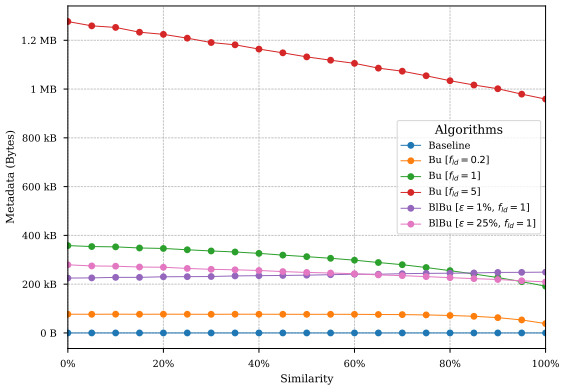
In this scenario, we observe that AWSets draws identical results to GSets, albeit with different values due to the different cardinalities between these. The explanation to such phenomena derives from the similar morphology of these studies data types. Both of them only possess an expandable components, i.e., memory usage increases monotonically.



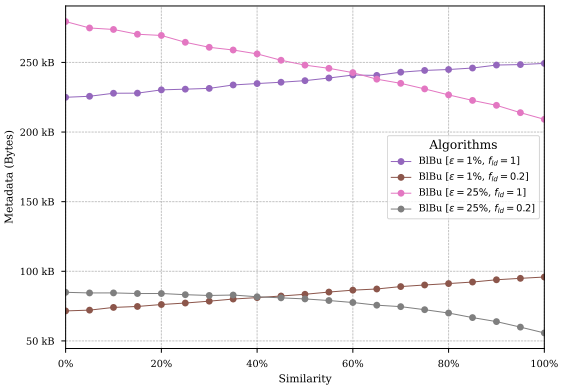
(a) General comparison of transmitted total bytes.



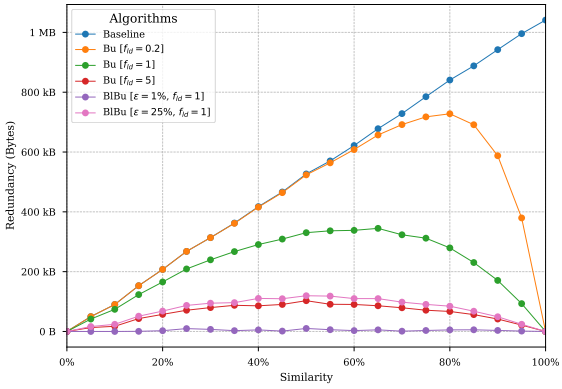
(b) BlBu comparison of transmitted total bytes.



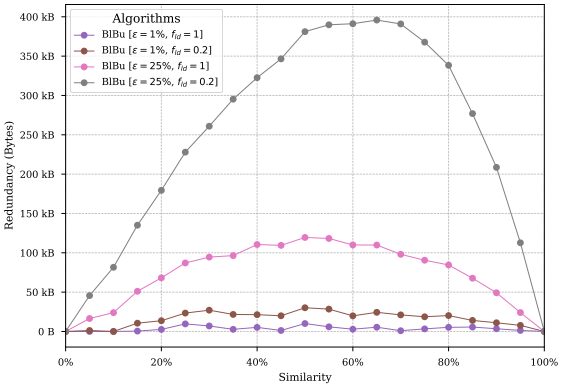
(c) General comparison of transmitted metadata.



(d) BlBu comparison of transmitted metadata.

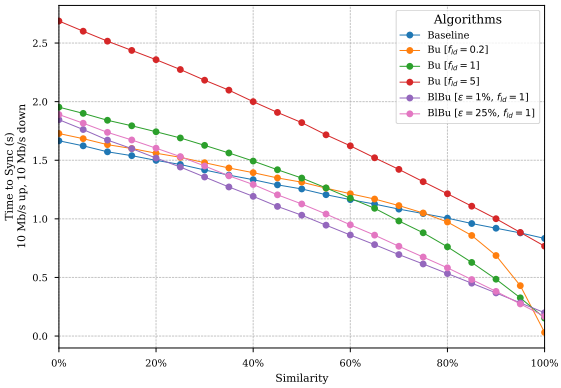


(e) General comparison of transmitted redundancy.

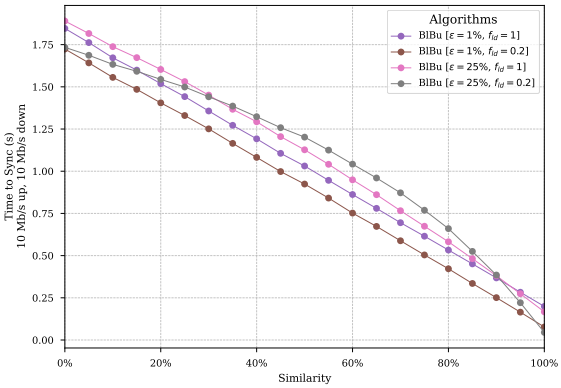


(f) BlBu comparison of transmitted redundancy.

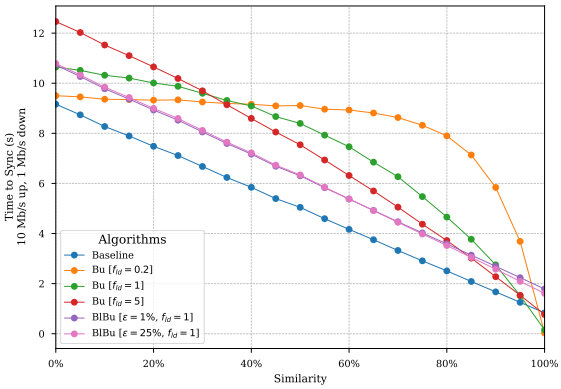
Figure 7.3: Transmission metrics w.r.t. similarity between a pairs of AWSets.



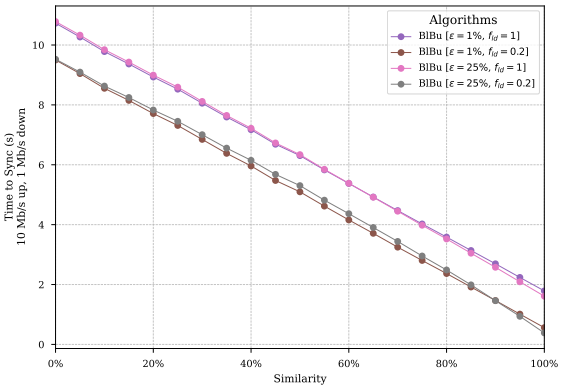
(a) Symmetric channel general comparison.



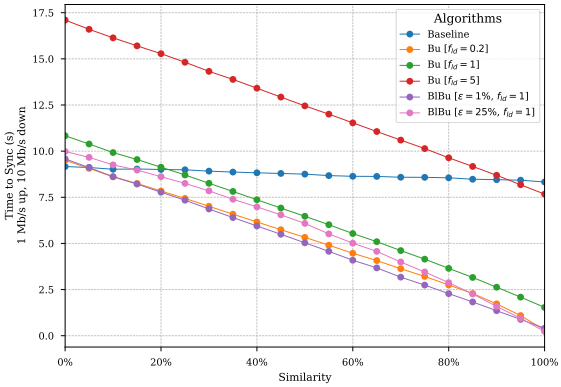
(b) Symmetric channel BlBu comparison.



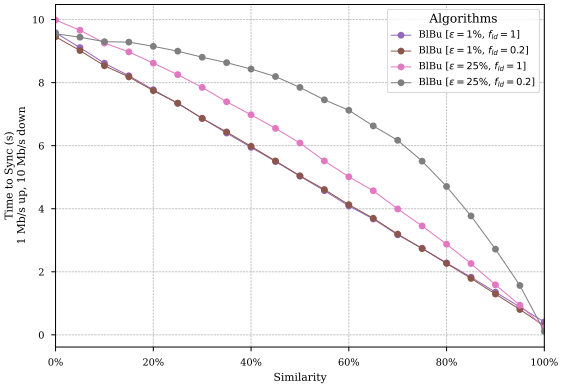
(c) Upload skewed channel general comparison.



(d) Upload skewed channel BlBu comparison.



(e) Download skewed channel general comparison.



(f) Download skewed channel BlBu comparison.

Figure 7.4: Impact of communication channel's bandwidth on the time to synchronize AWSets.

Table 7.4: Ratio of redundancy sent for different similarities between AWSets.

Algorithm	**0%**	**25%**	**50%**	**75%**	**90%**	**95%**	**100%**
Baseline	0.0%	14.6%	33.6%	60.1%	81.9%	90.5%	100.0%
Bu [$f_{ld} = 0.2$]	0.0%	14.0%	31.9%	54.7%	68.4%	70.7%	0.0%
Bu [$f_{ld} = 1$]	0.0%	9.9%	19.6%	28.3%	28.2%	22.8%	0.0%
Bu [$f_{ld} = 5$]	0.0%	2.5%	4.5%	4.3%	3.3%	1.9%	0.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 1$]	0.0%	0.5%	0.8%	0.4%	0.8%	0.4%	0.0%
BlBu [$\varepsilon = 1\%$, $f_{ld} = 0.2$]	0.0%	1.4%	2.6%	3.0%	3.5%	3.7%	0.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 1$]	0.0%	4.6%	8.5%	10.8%	10.3%	7.0%	0.0%
BlBu [$\varepsilon = 25\%$, $f_{ld} = 0.2$]	0.0%	12.2%	25.4%	38.3%	43.4%	40.7%	0.0%

7.3 Summary

In this chapter, we introduced the environment in which our experiments occurred. Moreover, we showed and discussed the results obtained from such experiments. This discussion is essential as it contains the answers to this work's RQs, enumerated in section 4.3.

Transmission and time analyses measure the impact of similarity on the synchronization procedure, thus answering to **RQ1**. In addition, we specified a generic similarity measure (eq. 7.2) for state-based CRDTs. Furthermore, the results indicated an improvement over existing methods, e.g., baseline, with increased similarity between replicas. Thus, these new approaches benefit from similarity. This fact not only answers **RQ3** but also supports part (1) of the hypothesis, delineated in section 4.2.

Given these circumstances, we validate the hypothesis. The two algorithms we tested throughout this chapter satisfy both of the components of the hypothesis. Note that the properties of these algorithms, detailed in Chapter 5, already met part (2) of the hypothesis.

Chapter 8

Conclusion

Geo-replication is fundamental to the operation of virtually all modern cloud infrastructure. These systems often sacrifice consistency in favor of higher availability and lower latency – properties desired by their users. CRDTs represent a significant advancement in this domain as they provide deterministic reconciliation based on a mathematical basis.

Historically, state-based CRDTs attracted less attention than their op-based counterparts. One of the reasons that can help explain such a phenomenon is the lack of practicability. As we examined, messages grow indefinitely alongside the state in the state-based domain. On top of that, it is clear that consecutive synchronization rounds lead to the transmission of a redundant state.

In this thesis, we explore prior work on state-based CRDTs and synchronization algorithms and propose two new techniques that take advantage of some similarities that might exist between synchronization peers: (i) bucketing and (ii) bloom-based.

These strategies leverage join-decompositions to compute optimal *differences* without requiring metadata. As we have seen, this latter aspect is crucial in systems operating over partitionable networks. Moreover, these techniques share similarities with rsync's weak and strong checksums.

Nonetheless, the problem of our thesis has several nuances that differ from the rsync's problem. First, the rsync algorithm is unidirectional, whereas state-based synchronization is bidirectional. Another concern is that rsync admits data loss at the receiver's end if the sender does not contain such data. In state-based CRDTs, data losses cannot happen due to the charateristics of join-semilattices.

In future work, we would like to extend our simulations to more data types, particularly ones that contain fixed, e.g., causal context [2], and expandable components. A more ambitious idea is to leverage or adapt MST's morphology, described in section 3.3, to operate as an index of irredundant join-decompositions. MST's page digests would act as bucket's digests. Upon an update, digests' computation can be delayed to the following query or happen immediately. In the latter case, MST is a caching system for bucket digests. This decision depends on the rate of updates and queries between synchronization rounds. Furthermore, it would be possible to roll over MST's join-decompositions and, at each point, compute a checksum, mimicking rsync's weak checksum. Exploring this idea further can reveal a true rsync for state-based CRDTs.

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

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

[3] Alex Auvolat and François Taïani. Merkle search trees: Efficient state-based crdts in open networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–22109. IEEE, 2019.

[4] Microsoft Azure. Distribute your data globally with Azure Cosmos DB. `https://learn.microsoft.com/en-us/azure/cosmos-db/distribute-data-globally`, 2023. (accessed Feb. 1, 2024).

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

[6] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[7] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[8] Garrett Birkhoff. Rings of sets. *Duke Mathematical Journal*, 3(3):443 – 454, 1937.

[9] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] Robert M Bond and R Kelly Garrett. Engagement with fact-checked posts on reddit. *PNAS nexus*, 2(3):pgad018, 2023.

[11] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. *ACM Sigplan Notices*, 49(1):271–284, 2014.

[12] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[13] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE, 2019.

[14] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Join decompositions for efficient synchronization of crdts after a network partition: Work in progress report. In *First Workshop on Programming Models and Languages for Distributed Computing*, pages 1–3, 2016.

[15] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[16] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10:56–66, 1988.

[17] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014.

[18] Ned Freed. SMTP Service Extension for Command Pipelining. RFC 2920, September 2000.

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

[20] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)*, 25:1–16, 2020.

[21] Joseph M. Hellerstein and Peter Alvaro. Keeping calm: When distributed consistency is easy, 2019.

[22] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[23] Sándor Z Kiss, Éva Hosszu, János Tapolcai, Lajos Rónyai, and Ori Rottenstreich. Bloom filter with a false positive free zone. *IEEE Transactions on Network and Service Management*, 18(2):2334–2349, 2021.

[24] Martin Kleppmann, Paul Frazee, Jake Gold, Jay Graber, Daniel Holmgren, Devin Ivy, Jeromy Johnson, Bryan Newbold, and Jaz Volpert. Bluesky and the at protocol: Usable decentralized social media. *arXiv preprint arXiv:2402.03239*, 2024.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[26] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.

[27] Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[28] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[29] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. An optimal bloom filter replacement. In *Soda*, volume 5, pages 823–829. Citeseer, 2005.

[30] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (CRDTs). In *Encyclopedia of Big Data Technologies*. Springer, May 2018.

[31] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[32] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. Merkle-crdts: Merkle-dags meet crdts. *arXiv preprint arXiv:2004.00107*, 2020.

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[34] Andrew Tridgell et al. Efficient algorithms for sorting and synchronization. 1999.

[35] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[36] Albert van der Linde, João Leitão, and Nuno Preguiça. δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2016.

[37] Robbert Van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–7, 2008.

[38] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.