Sapienza
Università di Roma

Design and Implementation of a Replicated File Storage System Using Conflict-free Replicated Data Types

Facoltà di Ingegneria dell’Informazione, Informatica e Statistica
Master’s Degree in Engineering in Computer Science

Giorgio Francescone
ID number 1974461

Thesis Advisor
Prof. Andrea Vitaletti

Academic Year 2024/2025

Design and Implementation of a Replicated File Storage System Using Conflict-free Replicated Data Types
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: francescone.1974461@studenti.uniroma1.it

Here's to the crazy ones.

The troublemakers.

The ones who see things differently.

Because they change things.

Steve Jobs, 1997

Abstract

The landscape of modern computing has increasingly shifted torwards distributed architectures to meet the demand of high availability and real-time collaboration. Traditional approaches to data synchronization often rely on centralized strategies, such as Operational Transformation (OT). This thesis provides a more sensible approach by exploring Conflict-free Replicated Data Types (CRDTs), which are data structures backed by sound mathematical frameworks, and guarantee Strong Eventual Consistency (SEC) in decentralized environments, without the need for a central authority.

To demonstrate the efficacy of this approach, this work presents the design and implementation of a Proof-of-Concept replicated file storage system. The system simulates a peer-to-peer network, where file contents and metadata are synchronized across multiple replicas connected to the same network. A distinguishing feature of the PoC is the integration of a cryptographic digital scheme mechanism, which enables users to assert ownership of files and allows peers to perform validity checks on shared data.

Contents

CHAPTER 1

Introduction

The modern computing landscape has moved towards building distributed systems as a means to fulfill the necessity of delivering high volumes of data to a continuously growing number of users over the Internet. It has become commonplace to have systems span over a network (cluster) of machines – or *nodes* – which communicate with each other in a coordinated fashion. Compared to monolithic systems that assign the entire workload to a single machine, this approach efficiently addresses three fundamental properties that the majority of modern systems seeks to achieve.

- *Scalability*: the system's ability to adapt to the scale of the workload by either allocating a larger quantity of resources on one node – *vertical scaling* – or dividing it among different nodes – *horizontal scaling*;
- *Resiliency*: the system's ability to tolerate eventual faults that may affect the correct operation of one or more nodes;
- *Efficiency*: the system's responsiveness expressed in terms of time and space [1].

Advancements in this field sought to optimally achieve these pivotal demands have led to the emergence of more advanced paradigms, such as *cloud computing*, which poses itself as a model that enables ubiquitous, on-demand access to a shared pool of configurable computing resources – networks, servers, storage, services – provisioned with minimal management

[1]An ideal system operates under low latency and high throughput

effort [1]. The advent of cloud computing has fueled an increasing interest in applications that are built to empower collaborative tasks. Platforms like Google Drive[2], Microsoft OneDrive[3], and Dropbox[4] enable groups of two or more users to engage in common tasks by sharing access to a set of resources – documents, spreadsheets, presentations, images, ... – from anywhere in the world, in real time [2].

Within collaborative applications lies the challenge of making sure that concurrent updates are applied seamlessly and without conflicts, especially in situations where users may be working across different geographical locations with varying network latencies. Despite the risk of network connectivity being intermittent, users expect their collaborative experience to be responsive, their changes to be preserved, and conflicts to be resolved seamlessly without loss of data. This challenge can be broadly addressed either through *consensus* – leveraging algorithms such as Paxos or Raft – or through *convergence* [3]. Traditional approaches to achieve convergence, like the one adopted by Google Drive, heavily rely on centralized conflict resolution strategies such as *Operational Transformation* (OT), where a single node acts as the authoritative source of truth and mediates all concurrent changes [4]. While this mechanism has proven to be reliable, a compelling alternative could be achieved by employing a truly decentralized, peer-to-peer (P2P) network, over which data is replicated across multiple nodes and concurrent changes are handled without the need of a centralized authority.

Conflict-free Replicated Data Types (CRDTs) offer a powerful solution suitable to support highly available file sharing and collaborative systems. CRDTs are data structures that allow the system to update any replica independently and concurrently, without coordinating with other replicas [5]. The core benefit is that the data type itself contains an algorithm that automatically resolves any inconsistencies, guaranteeing that replicas will eventually converge to an identical state, even in the case in which a node may receive

[2]https://drive.google.com
[3]https://onedrive.live.com
[4]https://dropbox.com

updates out of order.

The formalized problem addressed by this thesis is the design and implementation of a Proof-of-Concept (PoC) replicated file storage solution that leverages CRDTs to synchronize file contents and metadata. The system is conceptualized as a P2P network where each participating node maintains a local replica of the storage system. All changes made on any node are propagated to other nodes and merged automatically using CRDT algorithms. For the PoC, the complexities of a true P2P network are abstracted away by using a relay server. This server acts as a message broker, broadcasting updates from one node to all other connected nodes, thereby simulating the propagation of changes in a P2P environment. The core of the system is built using the *PyCRDT library*[5], a Python binding for the popular *Yjs*[6] CRDT framework. This allows for the use of mature and well-tested CRDT implementations to model the file system's directory structure, file contents, and metadata. The system is complemented by a simple web-based client that provides a user interface for seamless file management, where users can upload their files and enforce a robust data integrity guarantee by attaching a synthetic digital signature to each file. Other users can validate such signatures through an offline-first validation mechanism, using the data that was received through CRDTs.

The rest of this thesis is structured as follows:

- In **Chapter 2**, we provide a comprehensive overview of the foundational concepts of CRDTs, along with their taxonomy, core data structures, and conflict resolution strategies.
- In **Chapter 3**, we analyze existing CRDT implementations and frameworks, with a particular focus on the Yjs ecosystem, which forms the basis of the PoC.
- In **Chapter 4**, we delve into the design and implementation process a

[5]https://github.com/y-crdt/pycrdt
[6]https://yjs.dev/

functioning PoC that demonstrates the feasibility of using CRDTs for file synchronization in a simulated P2P environment.

- In **Chapter 5**, we evaluate the PoC's performance, identify its strengths and limitations, and propose directions for future work.

CHAPTER 2

Fundamentals of Conflict-free Replicated Data Types

2.1 Consistency in Distributed Systems

Since the early days of networked computing systems, the challenge of maintaining strong consistency (or *linearizability* [6]) has been a critical research topic. As distributed systems started spanning across an ever increasing amount of devices spread over vast geographic locations, ensuring that all the nodes of a system operate on a coherent shared state at all times becomes a nontrivial task. The complexity also lies in making sure that these systems are always available and robust to any fault or partition that the network could be affected to. While traditional approaches to replication, such as Lamport's state machine [7], achieve consistency by relying heavily on central coordination mechanisms, the adoption of such approaches often comes at the cost of higher latency and disruption of availability under network partitions. This trade-off has been formalized in literature by the CAP Theorem [8, 9], which identifies three key guarantees sought by distributed systems:

- *Consistency* (C): all nodes of the systems are up to date with the updates issued to the internal state. In the context of an atomic read-/write register, every read operation receives the most recent write – or an error.

- *Availability* (A): every request received by a non-failing node must result in a response.
- *Partition tolerance* (P): the system continues to operate despite faulty conditions, such as network outages, that lead to arbitrary message loss between nodes.

Moreover, the CAP Theorem states that it is impossible for a real-world system to simultaneously guarantee more than two of these three properties.

Since network partitions may always occur, system designers were forced to make explicit choices about the system's behavior during partitions, by either sacrificing consistency or availability. As a consequence, two dominant design philosophies emerged, being *AP* (Availability-Partition tolerance) and *CP* (Consistency-Partition tolerance) systems. In AP systems, such as Amazon's Dynamo [10], eventual inconsistencies between replicas that may arise during network outages are tolerated to ensure that the system is kept operational at all times. Conversely, CP systems – such as Google's Bigtable [11] – prioritize strong consistency guarantees, and particular measures are adopted to ensure that inconsistent data is never served, such as affecting the system's availability during partitions. While replicated systems are geared towards the latter approach, high availability needs remain interesting from a business standpoint, as "even the slightest outage has significant financial consequences and impacts customer trust" [10]. For this reason, research was driven towards alternative consistency models, namely those where the consistency guarantees are relaxed, so that all three properties denoted by the CAP Theorem could simultaneously co-exist.

One such consistency model, known as *optimistic replication*, addresses concurrency control among replicas by letting the data be accessed "optimistically" (without a priori synchronization), propagating the resulting updates in the background, and fixing conflicts after they happen [12]. Another prominent example of weak consistency model, named *eventual consistency* (EC), allows for replicas to temporarily diverge, but guarantees that, when updates to the replicas are no longer issued, they will converge to an identical state [13]. This model significantly enhances availability and responsiveness, as operations can proceed without waiting for acknowledgement from other peers. However, if concurrent updates are not reconciled in a principled manner, replicas may end up in an inconsistent state, leading to erroneous behavior within the system. An alternative approach to conflict resolution has led to the emergence of CRDTs, which provide a mathematically sound framework for achieving state convergence without the need for synchronization among replicas [14].

2.2 Foundational Concepts on Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) are abstract data types specifically engineered for replication across multiple processes in a distributed system. A defining characteristic of CRDTs is that any replica can be modified without requiring immediate coordination with other replicas, and replicas that have received the same updates reach the same state deterministically [15]. This property is fundamental to achieving high availability and low latency in distributed applications.

The advantage of CRDTs lies in their mathematical foundation. Rather than relying on application-specific conflict resolution logic, they ensure convergence through carefully designed data structures whose operations satisfy specific algebraic properties.

2.2.1 CRDT Operation Semantics

A CRDT can be implemented as a standard data type, which provides a set of operations, which usually can be distinguished into two categories [14]:

- *query* operations, which perform a lookup on the internal state – without altering it – and return a result based on what is observed;

- *update* operations, which perform a set of actions that explicitly mutate the internal state.

As we will see when discussing CRDT implementations, performing an update on a CRDT which is replicated across n nodes firstly involves updating the state of a single replica – the one which is directly accessed by the application – and then propagating the update to all other nodes of the network. When nodes receive an update, they apply it to their replica, and autonomously resolve any conflicts that may arise – in other words, they *merge* the update with their internal state. Broadly speaking, designing an effective update operation can be regarded as a complex endeavor, as failure to properly address the correctness criteria involving the set of operations that form the merged update may result in compromising the system's convergence guarantees. More precisely, the implementation of update operations must guarantee convergence by satisfying one or more of the following key properties:

- *commutativity*: the order in which delivered updates are applied to the replica does not affect the final outcome;
- *associativity*: the order in which merges are grouped does not affect the outcome;
- *idempotency*: repeated applications of an update yields the same outcome as a single application.

In most cases, it is desirable to deal with commutative operations; due to the reliability of the network, a replica may receive updates out of order, and still be required to converge to a state that is consistent with all other replicas, once all the updates are applied.

2.2.2 Strong Eventual Consistency

Convergence in the case of CRDTs is backed by a stronger version of the EC model, known as *Strong Eventual Consistency* (SEC)[15], which guarantees that, once all nodes have received the same set of updates, they will (eventually) reach the same final state. Formally, we can define the following.

Eventual Consistency (EC) An object is said to be *Eventually Consistent* if it satisfies the following properties:

- **Eventual delivery:** An update delivered to a replica is eventually delivered to all replicas;
- **Convergence:** Replicas that have delivered the same updates eventually converge to an equivalent state;
- **Termination:** All method executions terminate.

Strong Eventual Consistency (SEC) An object is said to be *Strongly Eventually Consistent* if it is Eventually Consistent if it satisfies the following additional property:

- **Strong Convergence:** Replicas that have delivered the same updates converge to an equivalent state.

SEC achieves state convergence without requiring complex, ad-hoc conflict resolution logic to be implemented at the application level. Instead, the "conflict-free" nature is embedded within the design of the data type itself; concurrent operations are either inherently commutative or are resolved by a deterministic merge procedure built into the CRDT.

2.3 Mathematical foundations of convergence

We will now describe some notions from order theory that are useful to understand the mathematical properties behind CRDTs' guarantee of convergence.

2.3.1 Partial Order and Semilattices

A fundamental mathematical structure underlying CRDTs is the join semilattice. Before we describe the definition of join semilattice, we will provide the definition for what a partially ordered set and what a least upper bound is [16].

Partial Order A *partially ordered* set – or *poset* – is a set S and a binary relation $\leq$ such that, for all $a, b, c \in S$, the following conditions hold:

1. $a \leq a$ (*reflexivity*);
2. If $a \leq b$ and $b \leq a$, then $a = b$ (*antisymmetry*);
3. If $a \leq b$ and $b \leq c$, then $a \leq c$ (*transitivity*).

A partially ordered set is denoted as $(S, \leq)$.

Least upper bound A *least upper bound* (LUB) of two elements a and b in a partially ordered set is an element that satisfies the following conditions.

1. The LUB is greater than or equal to both a and b. That is, LUB $\geq a$ and LUB $\geq b$.
2. There is no other element c, with $a \geq c$ and $b \geq c$, such that c is less or equal than the LUB.

Through the LUB, we can also understand the definition of a join. For a partially ordered set $(S, \leq)$ and two elements $a, b \in S$, the *join* of a and b – denoted by $a \sqcup b$, is a LUB of S. Fundamentally, the join obeys satisfies three key properties that are relevant to CRDTs:

1. $a \sqcup b = b \sqcup a$ (*commutativity*);
2. $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ (*associativity*);
3. $a \sqcup a = a$ (*idempotency*).

Join semilattice A *join semilattice* is a partially ordered set $(S, \leq)$ where there exists a join $x \sqcup y$ for any $x, y \in S$.

It is also useful to observe that a join semilattice is *monotonic*. That is, for any $a, b \in S$, $a < a \sqcup b$.

The properties of a join semilattice are crucial in ensuring that, regardless of the order in which updates are merged, the result is always the same and represents a consistent "join" of the information from all replicas. Moreover, the guarantee of monotonicity ensures that replicas always "grow forward" through merge operations, with respect to state mutation.

2.4 Taxonomy of CRDTs

Before the introduction of a few relevant CRDT implementations in the following chapter, it is useful to provide a comprehensive taxonomy of CRDT designs, each with its distinct characteristics with respect to how replicas – more precisely, updates among replicas – are synchronized. Fundamentally, these branch into one of two categories, *state-based* or *operation-based* CRDTs[17], with a hybrid approach – *delta state-based* CRDTs – also being viable.

2.4.1 State-based CRDTs (CvRDTs)

State-based CRDTs, also called convergent replicated data types (CvRDTs), operate on the principle of achieving convergence through propagation of entire states. The source replica receives an update and applies it locally, and the resulting state is then propagated to other replicas through a payload. Upon receiving a state, if a replica's local state and the received state conflict, a deterministic merge operation is applied, which leads to a new state that reflects the conflict resolution. The overall process can be described as follows.

1. A client invokes an update on a source replica.

2. The update immediately modifies the local state.
3. The modified state is asynchronously propagated to other replicas.
4. Replicas receive the modified state and merge it with their own local state.

As states can be propagated in many ways – for example, employing an epidemic propagation algorithm to an unknown set of participants [18] – and be delivered to replicas in any order, the merge operation must be compliant with the constraints already discussed above – it must be commutative, associative, idempotent, and monotonic. Thus, for this approach to work, the replica state must be a join semilattice.

2.4.2 Operation-based CRDTs (CmRDTs)

Operation-based CRDTs, also called commutative replicated data types (CmRDTs), take a different approach with respect to state-based replication. as the name suggests, op-based CRDTs rely on achieving convergence by propagating operations to replicas, rather than propagating entire states.

The process of op-based replication is typically subdivided into two phases.

1. A *"prepare"* phase, where the update is firstly applied to the source replica.
2. An *"effect"* phase, where the operations which form the update are asynchronously propagated and executed on other replicas.

In order for convergence to occur, a reliable messaging channel must be adopted, which complies with the constraints dictated by the CmRDT. For instance, operations are, in general, not idempotent, which implies the adoption of an exactly-once messaging mechanism. Furthermore, op-based replication typically occurs through causal broadcast mechanisms [19], which leads to two or more operations which are causally dependent to each other to be viewed in the correct order. While CmRDTs are more complex in design, they provide greater expressive power [17].

2.4.3 Delta State CRDTs (δ-CRDTs)

While state-based and operation-based CRDTs are both successfull in achieving SEC guarantees, a third approach has also emerged in literature which addresses the drawbacks of the previous two approaches. On one hand, state-based CRDTs may be inefficient when the state object is large in size, affecting the bandwidth of the network. On the other hand, operation-based CRDTs heavily rely on the design of complex propagation channels that require the guarantees of reliable and causal communication. *Delta State CRDTs* – also commonly notated as δ-CRDTs – evolve from both state-based and operation-based CRDTs. They rely on adopting a simple, unreliable propagation channel (state-based replication) for the dissemination of messages that are small in size and of incremental nature (operation-based replication) [20].

While traditional state-based CRDTs rely on issuing state changes via mutator functions that return the entire modified state, δ-CRDTs rely on *δ-mutators* which return a value – or *delta* – reflecting the incremental changes. The delta is then propagated and merged with remote replicas to achieve convergence.

CHAPTER 3

Survey of Common CRDT Implementation and Frameworks

This chapter provides a systematic survey of Conflict-free Replicated Data Type (CRDT) implementations documented in academic literature, examining how data structures of varying complexity – registers, counters, sets, maps, lists – are designed to achieve state convergence. Each data structure is supported by a set of semantics that model the system's behavior in the face of concurrent operations.

We then conclude with an examination of a few production-grade CRDT frameworks tailored for general use, such as *Automerge* and *Yjs*, and illustrate their underlying algorithmic foundations, bridging theoretical formulations with practical distributed systems engineering. The latter will also constitute the backbone of the Proof-of-Concept we will discuss in the following chapter.

3.1 Registers

A *register* is an object that holds an opaque value, that supports a *write* operation to update its value, and a *read* operation to query it. Concurrent operations writing to the same register require safeguard measures to be adopted, which comes in the form of two rules, each supporting their own CRDT implementation, the *Last-Writer-Wins Register* (LWW-Register) and the *Multi-Value Register* (MV-Register) [17].

3.1.1 Last-Writer-Wins Register (LWW-Register)

The LWW-Register represents the simplest form CRDT implementation. It is formed by a single value, with a timestamp associated to it; the timestamp's presence is due to how the *last-writer-wins* rule is applied. Given two conflicting states, s_1 and s_2, which are affected by concurrent updates, the CRDT's *merge* operation is applied as follows:

$$\text{merge}(s_1, s_2) = \max(s_1.\text{timestamp}, s_2.\text{timestamp})$$

The merge operation compares the timestamp from each update and retains

Specification 1 State-based Last-Writer-Wins Register (LWW-Register)

```
let state $s$ := (value $x$, timestamp $t$)                ▷ initially $(\perp, 0)$
procedure UPDATE(value $x'$)
    $s.x = x'$
    $s.t = now()$
end procedure
procedure QUERY( )
    return $s.x$
end procedure
procedure COMPARE(states $s$, $s''$)
    if $s.t \leq s'.t$ then
        return true
    else
        return false
    end if
end procedure
procedure MERGE(states $s$, $s'$)
    let state $s''$ := $(\perp, 0)$
    if $s \leq s'$ then                                    ▷ COMPARE$(s, s')$
        $s''.x, s''.t = s'.x, s'.t$
    else
        $s''.x, s''.t = s.x, s.t$
    end if
    return s”
end procedure
```

the replica with the highest timestamp. This forms a semilattice where the partial order is defined by timestamp comparison, ensuring deterministic convergence regardless of message delivery order. A high-level state-based LWW-Register [17] implementation can be found in specification 1.

3.1.2 Multi-Value Register (MV-Register)

Made popular by Amazon's Dynamo [10] key-value store, Multi-Value Register CRDTs take a different approach to concurrency with respect to the LWW-Register. Rather than discarding values based on their timestamp, MV-Registers maintain all concurrently written values. In this case, the payload is formed by a set S of values, each with their associated causal context (e.g., vector clock or version vector).

As shown in specification 2, the *query* operation returns a copy of the payload, while the *update* operation overwrites it by associating the value to a "fresh" version vector that dominates all previous ones. The *merge* operation returns the union. Compared to its LWW-Register counterpart, the *merge* operation acts by preserving all concurrent values in the set [17] – in other words, a merge between two conflicting sets returns a union of every element in each input set whose version vector is not dominated by the one associated to an element in the other input set.

3.2 Counters

A *counter* is an integer object supporting both *increment* and *decrement* operations to update its value, and a *read* operation to query the current value. Trivially, we can construct an operation-based counter CRDT in its simplest form by propagating update operations as-is. Given that both addition and subtraction operations commute, the delivery order of concurrent operations is irrelevant for achieving convergence [17] – as long as no overflow is caused by such operation.

Specification 2 State-based Multi-Value Register (MV-Register)

```
1: let state $s :=$ set $X$        ▷ $X$: set of $(x, V)$ pairs of value $x$ and version
   vector $V$ (initially, $\{(\bot, [0, .., 0])\}$)
2: procedure UPDATE(set $R$)
3:     let $V' :=$ INCREASEVERSIONVECTOR( )
4:     $s.X = R \times \{V'\}$
5: end procedure
6: procedure QUERY( )
7:     return $r.X$
8: end procedure
9: procedure COMPARE(sets $A$, $B$)
10:     if $(\forall (x, V) \in A, (x', V') \in B : V \leq V')$ then
11:         return true
12:     else
13:         return false
14:     end if
15: end procedure
16: procedure MERGE(sets $A$, $B$)
17:     let $A' := \{(x, V) \in A | \forall (y, W) \in B : V || W \vee V \geq W\}$
18:     let $B' := \{(y, W) \in B | \forall (x, V) \in A : W || V \vee W \geq V\}$
19:     let $C := A \cup B$
20:     return $C$
21: end procedure
```

3.2.1 Grow-only Counter (G-Counter)

The simplest form of state-based counter CRDT is the *G-Counter*, which only supports *increment* operations. For this purpose, each replica i maintains a vector $V[1..n]$, where $V[i]$ represents the local increment count and $V[j]$ (for $j \neq i$) represents the last known count from replica j [17]. This implementation is shown in specification 3.

3.2.2 Positive-Negative Counter (PN-Counter)

In effort to support decrement operations, the PN-Counter is designed by combining two G-Counters. This CRDT's implementation is therefore formed by a compound payload of two integer vectors, one to keep track of incre-

Specification 3 State-based Grow-only Counter (G-Counter)

```
1: let state $s$ := vector $V[i_1, .., i_n]$ (where $n$: number of replicas) ▷ initally, $[0, .., 0]$
2: procedure INCREMENT( )
3:     let $g := localID()$                                   ▷ $g$: source replica
4:     $V[g] = V[g] + 1$
5: end procedure
6: procedure QUERY( )
7:     return $\sum_i V[i]$
8: end procedure
9: procedure COMPARE(vectors $X$, $Y$)
10:     if $\forall i \in [0, n-1] : X[i] \leq Y[i]$ then
11:         return true
12:     else
13:         return false
14:     end if
15: end procedure
16: procedure MERGE(vectors $X$, $Y$)
17:     let vector $Z := [0, .., 0]$
18:     $\forall i \in [0, n-1] : Z[i] = \max(X[i], Y[i])$
19:     return Z
20: end procedure
```

ments (P), while the other to keep track of decrements (N) [17].

Specification 4 illustrates this state-based implementation, with *increment* and *decrement* operations affecting separate vectors of the state's payload. The *query* operation returns the counter's value by computing the difference between the P and N vectors, while the *merge* operation independently merges the two vectors through component-wise maximum.

3.3 Sets

A *set* is a container for an arbitrary number of elements. It provides updates in the form of *add* and *remove* operations, which respectively add or remove a given element to or from the set, plus a *lookup* operation that checks whether a given element e belongs to the set. Several concurrency semantics

Specification 4 State-based Positive-Negative Counter (PN-Counter)

```
1: let state $s$ := Payload $(P[i_1, .., i_n], N[i_1, .., i_n])$ (where $n$: number of replicas)    ▷ initally, $([0, .., 0], [0, .., 0])$
2: procedure INCREMENT( )
3:     let $g := localID()$    ▷ $g$: source replica
4:     $P[g] = P[g] + 1$
5: end procedure
6: procedure DECREMENT( )
7:     let $g := localID()$    ▷ $g$: source replica
8:     $N[g] = N[g] + 1$
9: end procedure
10: procedure QUERY( )
11:     return $\sum_i P[i] - \sum_i N[i]$
12: end procedure
13: procedure COMPARE(payloads $X$, $Y$)
14:     if $\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \wedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i]$ then
15:         return true
16:     else
17:         return false
18:     end if
19: end procedure
20: procedure MERGE(payloads $X$, $Y$)
21:     let payload $Z := ([0, .., 0], [0, .., 0])$
22:     $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$
23:     $\forall i \in [0, n-1] : Z.N[i] = \max(X.N[i], Y.N[i])$
24:     return Z
25: end procedure
```

are viable in the presence of concurrent insertion and removal operations on the same element.

3.3.1 Grow-only Set (G-Set)

The simplest implementation of a set CRDT is to only permit *add* operations, which is made possible by employing an union of all added elements. Since the union operation is commutative, the operation-based implemen-

Specification 5 State-based Grow-only Set (G-Set)

```
let state $s :=$ set $A$                                ▷ initally, $\varnothing$
procedure ADD(element $e$)
    $A = A \cup \{e\}$
end procedure
procedure LOOKUP(element $e$)
    if $(e \in A)$ then
        return true
    else
        return false
    end if
end procedure
procedure COMPARE(states $s$, $s'$)
    if $s.A \subseteq s'.A$ then
        return true
    else
        return false
    end if
end procedure
procedure MERGE(states $s$, $s'$)
    let state $s'' := \varnothing$
    $s''.A = s.A \cup s'.A$
    return $s''$
end procedure
```

tation naturally converges, regardless of delivery order [17].

Specification 5 shows the state-based approach, where the *add* operation modifies the local state through union. A partial order on some state s and s' can be defined as $s \leq s' \Leftrightarrow s \subseteq s'$. Conversely, the *merge* operation between two states returns the union of their respective sets. The G-Set constitutes a basic building block of more complex implementations, such as the *2P-Set*

3.3.2 Two-Phase Set (2P-Set)

The 2P-Set is an extension of the G-Set with removal capability. It is formed by combining two G-Sets, which are used to keep track of added and removed

Specification 6 State-based Two-Phase Set (2P-Set)

```
let state $s :=$ sets $A, R$                    ▷ initally, $\{\varnothing, \varnothing\}$
procedure LOOKUP(element $e$)
    if $(e \in A)$ then
        return true
    else
        return false
    end if
end procedure
procedure ADD(element $e$)
    $A = A \cup \{e\}$
end procedure
procedure REMOVE(element $e$)
    precondition LOOKUP($e$)
    $R = R \cup \{e\}$
end procedure
procedure COMPARE(states $s$, $s'$)
    if $s.A \subseteq s'.A \vee s.R \subseteq s'.R$ then
        return true
    else
        return false
    end if
end procedure
procedure MERGE(states $s$, $s'$)
    let state $s'' := \{\varnothing, \varnothing\}$
    $s''.A = s.A \cup s'.A$
    $s''.R = s.R \cup s'.R$
    return $s''$
end procedure
```

elements.

Specification 6 shows an example of a state-based 2P-Set implementation. Similar to what occurs with PN-Counters, the two G-Sets of the 2P-Set keep track separately of *add* and *remove* operations, and the *merge* operation returns a new state resulting from the union of each set individually. It is worth noting that a critical issue may arise when a *remove* operation is attempted for an element e that does not currently exist in the set. As a consequence, the *remove* operation requires – as a precondition – that e already belongs to the set, before removing it.

3.3.3 Last-Writer-Wins element Set (LWW-element-Set)

An alternative approach to the 2P-Set can be obtained by applying the *last-writer-wins* semantic. Aside from having separate sets for keeping track of added and removed elements, each element has a timestamp associated to it. As a consequence, the *lookup* operation ensures that a given element e is in the set if it is in A and it is not in R with a higher timestamp [17]:

$$\mathit{lookup}(e) = \exists t, \forall t' > t : (e,t) \in A \land (e,t') \notin R$$

3.3.4 Observed-Remove Set (OR-Set)

A more intuitive construction of set CRDT can be obtained by considering *add-wins* semantics. Simply put, a situation in which an *add* and a *remove* operation on the same element e occur concurrently results in the outcome of e being in the set [15] – *add(e)* "wins" over *remove(e)*.

For the *OR-Set*, this means that concurrency strictly depends on causal delivery. An operation that cancels another will cancel the operations in its causal past, but not the ones that are being concurrently issued [5].

Specification 7 illustrates an op-based implementation of OR-Set CRDT, where the *add* operation always ensures to add an element to the set associating it with a fresh unique identifier U. This ensures that the *remove* operation is capable of retrieving all unique identifiers associated to the element e which is target of removal – more precisely, all the unique identifiers for the element e that are causally observed at the source replica – and issue removal for those specific entries, ensuring preservation of causality [17]. The unique tagging ensures that concurrent adds create distinct entries.

Specification 7 Op-based Observed-Remove Set (OR-Set)

```
1: let state $s$ := set $S$ of pairs (element $e$, unique ID $U$)    ▷ initially, $\varnothing$
2: procedure LOOKUP(element $e$)
3:     if $\exists U : (e, U) \in S$ then
4:         return true
5:     else
6:         return false
7:     end if
8: end procedure
9: procedure ADD(element $e$)
10:     prepare $(e)$
11:         let $\alpha = unique()$    ▷ $unique()$ returns an unique ID
12:     effect $(e, \alpha)$
13:         $S = S \cup \{(e, \alpha)\}$
14: end procedure
15: procedure REMOVE(element $e$)
16:     prepare $(e)$
17:         precondition LOOKUP$(e)$
18:         let $R = \{(e, U) | \exists U : (e, U) \in S\}$
19:     effect$(R)$
20:         precondition $\forall (e, U) \in R : add(e, U)$ has been delivered
21:         $S = S \setminus R$    ▷ remove pairs observed at source
22: end procedure
```

3.4 Maps

A *map* is a data type that maps keys to objects. It typically provides a $put(k, o)$ operation to associate key k to object o, and a $remove(k)$ operation to that removes the mapping for key k. Defining suitable concurrency semantics for maps is crucial, as decisions must be made regarding the outcome of concurrent *put* operations for the same key, or concurrent *remove* and *put* operations on the same key [14].

3.4.1 Observed-Remove Map (OR-Map)

We can define a map CRDT with semantics similar to the ones already observed for OR-Sets. In this case, the payload is formed by a set of triplets (key k, value v, unique ID U). The $put(k, v)$ operation associates key k to value v, tagging it with a fresh identifier. $remove(k)$ deletes all existing mappings associated to key k. Specification 8 shows an example of a naive

Specification 8 Op-based Observed-Remove Map (OR-Map)

```
1: let state $s$ := set $S$ of triplets (key $k$, value $v$, unique ID $U$)  ▷ initially, $\varnothing$
2: procedure PUT(key $k$, value $v$)
3:     prepare $(k, v)$
4:         let $\alpha = unique()$                    ▷ $unique()$ returns an unique ID
5:         let $R = \{(k', v', U') \in S | k' = k\}$
6:     effect $(k, v, \alpha, R)$
7:         precondition $\forall (k, v, U) \in R : put(k, v, U)$ has been delivered
8:         $S = (S \setminus R) \cup \{(k, v, \alpha)\}$  ▷ Replace elements observed at source by new one
9: end procedure
10: procedure REMOVE(key $k$)
11:     prepare $(k)$
12:         let $R = \{(k', v', U') \in S | k' = k\}$
13:     effect$(R)$
14:         precondition $\forall (k, v, U) \in R : put(k, v, U)$ has been delivered
15:         $S = S \setminus R$                        ▷ remove elements observed at source
16: end procedure
```

op-based implementation [17], where the set of unique identifiers that are un-mapped in the $remove(k)$ operation are retrieved at the source replica.

3.5 Lists (Sequences)

We will now discuss one of the most complex data types, which is the *list* – or *sequence* – which is essentially a set of elements which are placed in a precise order. For this purpose, each element in the sequence is associated to a positional index, and the sequence is sorted according to that same index. Implementing a suitable CRDT for this type of structure poses a particular challenge, as insertion and deletion operations for an element affect the positioinal indices of other elements, and it is crucial to preserve ordering in the face of concurrent modifications.

We will pay close attention to a few list CRDT implementations, as these are well versed to closely model structured documents in collaborative editing use cases – and, as such, serve as a solid foundation to the CRDT frameworks that will be described near the end of the chapter.

3.5.1 Replicated Growable Array (RGA)

The *Replicated Growable Array* implements sequence objects in the form of a linked list. Each node of the list holds an element and an identifier associated to it, plus a reference to its parent node [21] – i.e., the node after which it was inserted.

Each element's identifier is assumed to be unique and ordered in line with causality [17]. For this purpose, a typical identifier is modeled as a pair formed by a timestamp (e.g., vector clock) and the source replica's identifier. Namely, given two identifiers i_1 and i_2, respectively formed by tuples (timestamp t_1, unique ID U_1) and (timestamp t_2, unique ID U_2), i_1 is said to "precede" i_2 ($i_1 < i_2$) if $t_1 < t_2$ or if $t_1 = t_2 \wedge U_1 < U_2$ [22].

This definition of precedence is crucial for understanding concurrency semantics for *insert* operations. In case of two concurrent insertions referencing the same parent node – more precisely, two insertions both targeting the position immediately after the same node – the logic applied at the remote replica is to place the element with the highest timestamp after the parent node, and the element with the lowest timestamp immediately after that.

As a practical example of this behavior, consider a system formed by two replicas, with IDs 0 and 1 respectively. Suppose that an element identifier i is denoted in the form $X.Y$, where X is the operation's timestamp, and Y is the source replica's ID. For the sake of simplicity, we will consider a set of characters as the payload for the CRDT. As shown in Figure 3.1, the initial state for both replicas is given by the sequence of characters $\{s, o, d\}$, with identifiers 1.0, 2.0, and 3.0 respectively. Replica 0 issues operation $ins(2.0, n, 4.0)$, while replica 1 issues operation $ins(2.0, u, 4.1)$. Since both replicas are concurrently attempting an insertion after character o (identifier 2.0), they are compared according to the positional identifier's precedence rules described before. Since $4.1 > 4.0$, this results in character u being inserted before character n. As a consequence, the final state for both replicas is given by the sequence $\{s, o, u, n, d\}$.

Delete operations also require particular attention, since permanently deleting the same element is considered unsafe – a concurrent insertion referencing

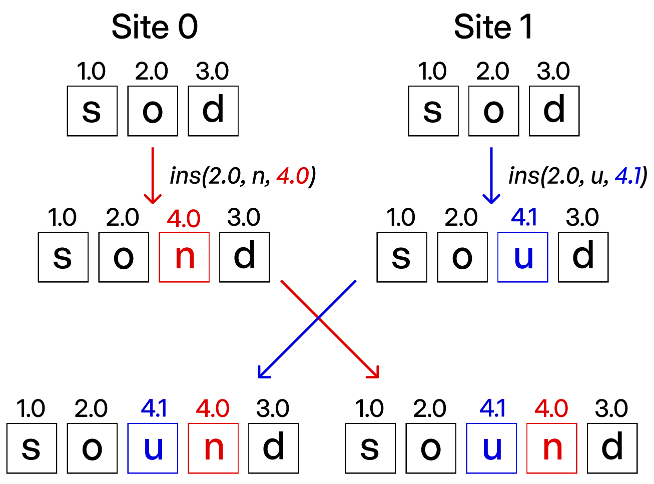


Figure 3.1. Example of concurrent insertion in RGA

a deleted node would be unable to correctly locate the insertion position. This can be solved by marking the target node with a *tombstone* ($\perp$), rather than physically removing it. When materializing the CRDT's content at a user-facing site, nodes marked with a tombstone are skipped.

3.5.2 WOOT (WithOut Operational Transformation)

$WOOT$ takes a similar approach with respect to RGA. It aims to achieve consistency by enforcing correctness through *causality*, *convergence*, and *intention preservation* [1] [24].

In this case, operations are denoted by the target position's adjacent neighbors – rather than by an absolute index or reference to a predecessor node, as it occurs in RGA. For instance, operation $ins(o_{x},_{y}, o_{z})$ inserts object o_{y} after object o_{x} and before object o_{x}.

While it is trivial to apply an insertion at the source replica, carefully designing the insertion integration algorithm at remote replicas represents the heart of WOOT. For example, suppose we have an ordered sequence of characters, and we would want to insert character c between characters c_p and c_n. The most trivial outcome is that no other concurrent insertions occur between c_p and c_n, resulting in c being inserted as-is. On the contrary, if two or more characters are inserted between c_p and c_n, the insertion integration requires particular attention and evaluaiton of the subsequence of characters between c_p and c_n, in order to compute a consistent placement for each character of the subsequence [24].

As it already occurs in RGA, $delete$ operations simply mark the target element with a tombstone, which requires no additional effort for integration at remote replicas.

[1]Formally, *intention preservation* means that any operation that has been applied at all replicas will have the same effects as if it were applied at the originating replica [23].

3.5.3 Logoot

We now briefly describe another prominent sequence CRDT implementation, provided by the *Logoot* algorithm. The basic idea is to use a lexicographic order to totally order the sequence elements [25]. Namely, Logoot employs a unique identifier allocation strategy, where each element in a sequence is assigned a dense, totally-ordered position identifier from an unbounded space. These identifiers are typically constructed in the form of variable-length lists of (integer index, replica ID) pairs, which enables the insertion of new characters between any two existing positions without reassigning existing identifiers.

3.6 Application Frameworks for general use

From an application development standpoint, while the CRDTs presented above are useful for a wide variety of situations – with the main use case being collaborative editing tasks – their implementation in source code, along with the quirks of the data structures themselves and/or of the update communication protocol, can be particularly challenging. In recent years, a few frameworks have emerged and achieved notable success for providing a "toolbox" of CRDTs ready to use and to integrate in many applications. Two of the most prominent frameworks are *Yjs* and *Automerge* [5], each of which uses a particular CRDT implementation from modern literature as a building block to construct more complex data structures. We will now introduce both of these frameworks, and illustrate the algorithmic foundations backing each of them.

3.6.1 Yjs

Yjs was developed in an effort to bridge the gap in collaborative applications deployed in peer-to-peer settings, since existing solution at the time – mostly based on Operational Transformation – did not scale well in decentralized environments [26]. It is a modular framework that provides several common CRDTs, such as map and array CRDTs, and several tools that enable integration with many communication protocols and databases. For instance, Yjs includes connection providers for protocols such as *Websocket*, *WebRTC*, and *Webxdc*, which can be leveraged for state update propagation, and database providers – e.g., *Redis* – which can be used to employ e persistent store for CRDTs and updates.

Under the hood, Yjs is essentially an open-source library implementation of the *YATA* (Yet Another Transformation Approach) protocol, written in the *JavaScript* language. YATA presents itself as a novel approach to peer-to-peer shared editing in near real-time, collaborative applications. It aims to ensure convergence and to preserve user intention for several arbitrary data types [27], and can easily be integrated with Web and mobile applications. In order to guarantee cutting-edge P2P concurrency control, YATA utilizes an internal linked list structure – taking inspiration from WOOT – and a set of predefined rules to enforce a total order on the shared data types. More precisely, linear data structures – such as text – is modeled as a doubly-linked list, where each element e references its *left* (e_{left}) and *right* (e_{right}) neighbors.

In terms of operation semantics, YATA provides both *insert* and *delete* operations. The *insert* operation can be denoted as:

$$insert(ID_{op}, e_{orig}, e_{left}, e_{right}, isDeleted, content(e))$$

Moreover:

- ID is the operation's unique identifier;
- e_{orig} references the inserted element's direct predecessor at creation time – i.e., the node in the linked list after which it was originally inserted;
- e_{left} and e_{right} reference the previous and the next node in the list, respectively;

- $isDeleted$ is a flag that marks the element as visible – or as a tombstone;
- $content(e)$ holds the inserted element's payload – e.g., a character or a string.

Through this model, YATA aims to guarantee that the inserted element will be placed somewhere between nodes e_{left} and e_{right}. If conflicts arise due to one or more elements being concurrently placed between the two nodes, a set of ad-hoc resolution strategies is applied, making sure that all replicas converge to the same state, and that the user's intention behind the operation itself is preserved.

Being based on the tombstone approach, *delete* operations are straightforward, and do not have an impact on concurrent insertions or deletions.

Yjs exposes its YATA implementation in the form of a *document*, which is essentially a container of shared data types – maps, arrays, sets, and so on. When a shared data type is affected by one or more changes, it triggers a *transaction*, which generates an update message – i.e., a delta-state – that can be shared over the broadcast channel and seamlessly be merged by remote replicas [28]. In compliance with the decentralized nature of CRDTs, a replica can autonomously apply a received update to its internal state, and does not require coordination with any external entity – e.g., a centralized coordination server.

3.6.2 Automerge

Automerge presents itself as a data synchronization and conflict resolution library, which enables mobile devices to make changes to shared data types [29], regardless of network topology – this includes changes that a device would have made while offline. As the name suggests, changes made on data are automatically merged to other devices – this also includes changes made while the local device was offline.

At its core, Automerge provides a document structure implemented as a CRDT, namely a JSON (JavaScript Object Notation) data model. Automerge documents are composed of nested maps and lists or simple values or text sequences [30]. They employ simple merge rules, which are based on RGA, and depend on the specific data type. In general, conflict resolution is handled arbitrarily, but in such a way that all nodes agree on the chosen value – e.g., *last-writer-wins* semantics are applied. Moreover, Automerge provides the application developer a complete view surrounding the logic which was applied to resolve a conflict. This means that, although only one of the concurrently written values shows up in the object's final state, the other values are not lost, but merely relegated to a list of conflicts appended to the object itself. Through this approach, Automerge employs both *last-writer-wins* and *multi-value* semantics.

CHAPTER 4

Design and Implementation of the Proof-of-Concept Application

Whereas the previous chapters were dedicated in providing a comprehensive literature review on the topic, we will now focus on the practical implementations of integrating solutions based on CRDTs in distributed software. In order to achieve this, we will analyze a use case in which the implementation of CRDT-based data structures proves to be indispensable, and document the development of a Proof-of-Concept application built around that use case.

4.1 Defining the Use Case for the Proof-of-Concept

The traditional infrastructure surrounding the broad family of collaborative applications – e.g., Google Docs, OneDrive, and many more – have constantly relied on centralized coordination paradigms, such as OT. While the approach of designating a central coordinator for conflict resolution offers distinct advantages – primarily revolving around enabling near real-time, non-blocking collaboration and preserving the user's intention – there are some contexts in which centralization may present significant drawbacks, mostly in terms of partition tolerance. For instance, the central coordinator may be put in a position to be the single point of failure; if it is unreachable, there is no other node that can approve or order changes, and the operativity of the entire system collapses[1].

This precisely illustrates the fundamental advantage of decentralized, peer-to-peer (P2P) architectures based on CRDTs; through replication across multiple nodes, information is maintained with high availability and consistency – i.e., compliant with SEC constraints. Furthermore, if one or more nodes of the system fail, the system's operativity is still guaranteed, as updates travel to all active nodes – whereas the faulty nodes will reconcile with the system's consistent state, once they come back online. This approach eliminates the threat of a single point of failure that would compromise system-wide reliability.

To illustrate this principle, we present a Proof-of-Concept implementation in the form of a replicated file storage system that leverages CRDTs to achieve eventual consistency without the need of centralized conflict resolution mechanisms. This system enables multiple users to execute fundamental file operations – including file sharing and deletion – on a shared storage infrastructure.

Furthermore, the solution provides a *file signature* mechanism, through which an unique user provides a secure method of certifying its ownership over the same file, and on-demand validity check over the signature can be requested by any other user.

In the following sections, we will refer to the Proof-of-Concept by the name of *CRDTSign*.

4.1.1 Proof-of-Concept Scope and Boundaries

The primary objective that CRDTSign seeks to carry out in its current form is to serve as a tangible illustration of the capabilities of CRDTs in contemporary distributed peer-to-peer (P2P) systems. As such, several high-

[1]Some countermeasures can be adopted, such as implementing a leader election mechanism for designating a new central coordinator node among the nodes of the network. Nevertheless, this approach would introduce a further layer of complexity in the distributed system, and would require appropriate handling to enforce convergence.

level requirements must be satisfied:

- consistent with the principles of decentralized applications, the back-end logic is not centralized on a single node, but is served independently at each participating node;
- because CRDTSign aims to provide an example of modern collaborative software, convergence among nodes must be achieved in near real-time – to satisfy end-user expectations for system responsiveness.

Conversely, given that the digital signature mechanism is integral to the solution – serving as a certificate of ownership for files shared by users – the system must adopt a cryptographic signature scheme that ensures:

- *Integrity*: signatures cannot be altered without detection;
- *Authenticity*: the signer's identity can be verified;
- *Non-repudiation*: the signer cannot deny having signed the data.

As an additional requirement, the system adopts a data retention policy, which ensures that files whose creation timestamp exceeds a configurable interval – e.g., 90 days – are automatically removed from the shared storage. This is made possible to prevent cases in which the storage's memory becomes saturated over time by files which are unused or obsolete.

It is worth underlining that the solution developed through CRDTSign poses itself as a "toy" implementation. While it satisfies the core requirements illustrated above, certain architectural constraints hinder its expectations of performance in production-grade environments. For instance, the P2P architecture was not constructed from the ground up; instead, it was abstracted through the use of a relay server. Rather than enabling direct node-to-node communication, all messages are routed through the relay server, which in turn broadcasts them to all peers – thereby simulating a P2P environment. This design choice does not fundamentally compromise the decentralized nature of the system, as concurrency management logic remains managed across individual nodes, rather than being centralized within the relay server.

Additionally, the CRDT data structure was sourced from an open-source library, and was not custom-made for this application. As we will discuss in the next chapter, a more robust and scalable solution could be achieved by designing a domain-specific CRDT structure and message exchange protocol tailored to the characteristics of the shared data.

4.1.2 User Stories

We will now discuss several use cases of CRDTSign by describing the functionalities of the system that are available to the end-user.

Uploading a file to storage Suppose that *User A* has the need to share a file with a group of users – e.g., *User B* and *User C*. To do so, it possesses a dedicated terminal (*Node A*), through which it can access CRDTSign's *Web interface*. Once connected and authenticated in the Web interface, User A uploads the file that it wants to share to the storage; at this point, the system reads the file, and computes a digital signature that uniquely identifies User A as the author of the uploaded file. Internally, the system first adds the file to its local storage, saving its metadata accordingly, and encapsulates the corresponding change to its state in an update message that is then broadcast to all other active nodes. Conversely, both Nodes B and C receive the update, and autonomously apply it to their respective local storages. After the update is successfully applied, both User B and User C can visualize the uploaded file and its metadata in their respective Web interfaces, and can inspect it for further details.

Verifying the validity of a file signature Suppose now that, once User B receives the file uploaded from User A, it wants to confirm that the file was indeed uploaded by User A itself. To do so, it can request a *signature validity check* for the aforementioned file. Consequently, Node B retrieves the file metadata, combines it with details concerning User A's identity – i.e., User A's unique identifier – and uses them to cryptographically validate the signature to verify. Node B can either assert that the signature is valid,

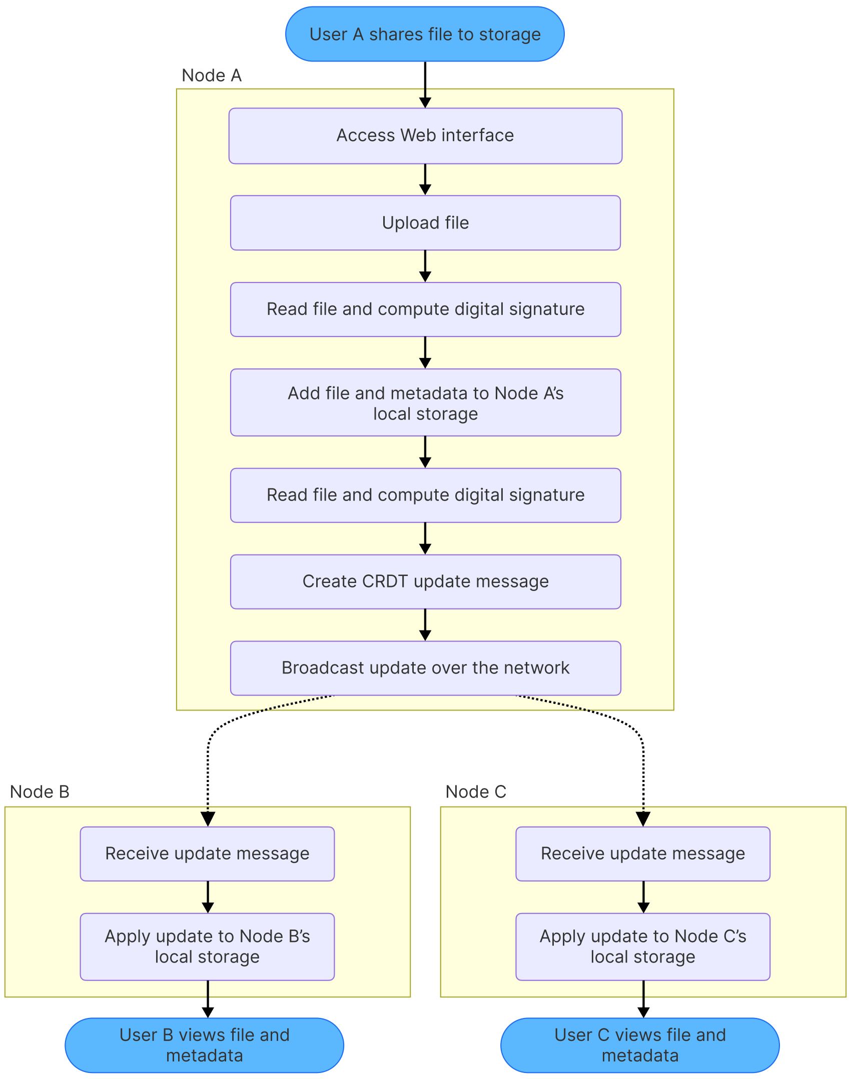

User A shares file to storage
Node A
Access Web interface
Upload file
Read file and compute digital signature
Add file and metadata to Node A's local storage
Read file and compute digital signature
Create CRDT update message
Broadcast update over the network
Node B
Receive update message
Apply update to Node B's local storage
User B views file and metadata
Node C
Receive update message
Apply update to Node C's local storage
User C views file and metadata

Figure 4.1. High-level flow diagram for the file upload procedure

if the check is successfull, or it is not valid, if the cryptographic validity check fails with the given parameters. Finally, Node B informs the user – through a notification in the Web interface – of the result of the signature validity check.

Deleting a file from storage Having received the file, User C seeks to remove it from storage[2]. It can do so by accessing the Web interface, selecting

[2]For the sake of demonstration, whether User C decides to delete the file in agreement with other peers, or of its own accord – potentially exposing a malicious intent – is irrelevant. In a production-grade application, this can be easily addressed by implementing a

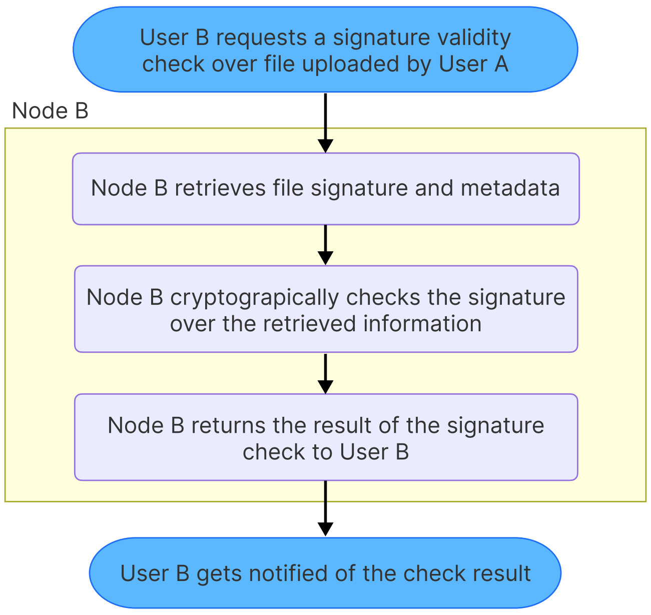

User B requests a signature validity check over file uploaded by User A
Node B
Node B retrieves file signature and metadata
Node B cryptograpically checks the signature over the retrieved information
Node B returns the result of the signature check to User B
User B gets notified of the check result

Figure 4.2. High-level flow diagram for the signature check procedure

the file in question, and request its removal from storage. The system responds by first removing the file from local storage, and then computing an update message that reflects the node's local state after the file removal. This update is then broadcast to all other active nodes, which individually apply it to their respective local states – leading to a consistent state where the file is removed from storage. The procedure carries out similarly to what occurs in Figure 4.1.

4.2 System Architecture

CRDTSign is made up of various components, each carrying out a specific task to achieve the functionalities that were previously discussed. In the overall distribute network, we can distinguish between two kinds of node:

- *Regular node*: is in charge of accepting and handling the user requests, employing both frontend and backend logic. It is implemented through an application written in the *Python*[3] programming language, which

permission-based mechanism, which limits the set of actions a certain user can perform on files uploaded by other users.

[3]https://www.python.org/

exposes a Web interface through which all of CRDTSign's core functionalities are provided to the end-user. When triggered, each functionality invokes a custom-made Python library which implements and executes the backend logic locally, interacting with suitable CRDT data structures when required. These structures are implemented through *pycrdt*[4], a library that provides bindings to an open-source port of Yjs in the *Rust*[5] programming language.

- *Relay node*: the server in charge of broadcasting messages to regular nodes. When a regular node sends an update to the relay node, the latter sends the corresponding message to all other nodes that need to receive it. Communication between regular nodes and the relay node occurs through the *WebSocket* [31] protocol.

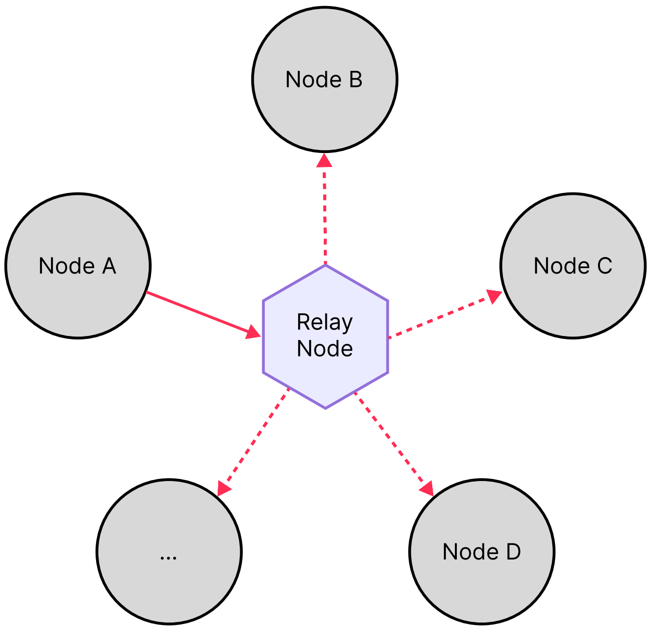


Figure 4.3. Example of peer-to-peer communication through the relay node – Node A delivers an update to the relay node, and the relay node delivers it to all other nodes

[4]https://y-crdt.github.io/pycrdt/
[5]https://rust-lang.org/

4.3 Software Components

This section describes the technical implementation of the software components that form CRDTSign. The overall solution is organized in a modular fashion, and can be clustered into three core elements, which are the *backend* engine, the *frontend* interface, and the *relay server* logic. While the first two are equally deployed at each individual node of the distributed network, the third is hosted by a dedicated node belonging to the same network. Furthermore, the specifications of the chosen digital signature scheme are illustrated, along with the design choices that were taken to integrate such scheme in the solution.

4.3.1 Backend

The backend serves as the main computational engine for each node in the network. As discussed above, the backend logic is encapsulated through a custom-made library that supports several core functionalities of the solution. We can distinguish among the following operations.

- User identity management (module `user.py`)
- Cryptographic signature generation and validation (module `sign.py`).
- CRDT state management and persistence (module `storage.py`).
- Additional utility operations, such as file serialization and deserialization (module `file_utils.py`), and operations concerning the enforcement of ad-hoc data retention policies (module `data_retention.py`).

`user.py` – User identity management

The `user.py` module curates the identity management and persistence on local storage. The operations provided by this module ensure that all the necessary information revolving the user connecting to the solution is correctly generated. This information is then referenced when the user uploads a file through the solution's frontend interface.

For instance, the following objects are associated to a user entity:

- the user's *unique identifier* (`user_id`), which is generated by randomly selecting 12 alphanumeric charaters from the ASCII set – which comprises both upper-case and lower-case variants of the 26 letters of the latin alphabet, and 10 digits from 0 to 9;
- the user's *displayed name* (`username`), which is inputted by the user upon first login to the web interface.

Moreover, the module provides operations for handling the saving and loading operations of the generated information into a dedicated file in local storage. Saved files are placed in a location which is accessible to the `storage.py` module – and as we will see in subsection 4.3.2, from the API interface.

All of the above operations are contained in a dedicated abstract class, named `User`.

`sign.py` – Cryptographic signature generation and validation

The `sign.py` module provides a collection of Python methods that enable the system to generate suitable file signatures – and consequently validate them on-demand.

Under the hood, the module leverages the *Ed25519* scheme [32] for signature management, which is the primary implementation of the *Edwards-curve Digital Signature Algorithm* (EdDSA) – based on the *SHA-512* hash function and the *Curve25519* elliptic curve [33]. The Python implementation of Ed25519 is provided by the open-source *cryptography* library[6]. Namely, the core payload of the Ed25519 implementation provided by this library is a 64 byte-sized cryptographic keypair – made up of a private and a public key, both having a size of 32 bytes – that can be associated to the specific user. The system then uses the user's private key to *sign* a given payload – in this case, the target file's *SHA-256* hash digest was chosen to represent such payload. The output of this process is a 64 byte signature, that can

[6]https://github.com/pyca/cryptography

be then passed to the `storage.py` module for attachment to the target file's metadata.

The module also provides a custom signature *validation* method, which requires three inputs:

- the target file's *hash digest*;
- the target file's *signature*;
- the *public key* associated to the target file's signing user.

The custom method uses the public key to verify the given signature over the file hash that was used to generate the same signature, and returns a boolean output reflecting the outcome of the verification – `True` if the signature is deemed *valid* with the provided parameters, `False` otherwise.

`storage.py` – CRDT state management and persistence

The *storage.py* serves as the computational engine responsible for operations executed on the available CRDT objects. On each node, two core objects are managed, each within a dedicated abstract class: the `FileSignatureStorage` class and the `UserStorage` class. These classes implement state management performed both on the CRDT object they instantiate, and on the persisted data on local storage. For instance, each performed CRDT operation – *add* or *remove* – triggers a dump of the CRDT's current state to local storage. The system also checks at boot time whether a state artifact is present in local storage, and loads it accordingly.

`FileSignatureStorage` The `FileSignatureStorage` class is employed to manage the files shared among nodes in dedicated CRDTs modeled as key-value stores – implemented using the *pycrdt* library's `Map` object. The key for each item in the CRDT is the uploaded file's unique ID, while its corresponding value is a Python *dictionary* object which contains the following information:

- the uploaded file's *name*;

- the uploaded file's *hash digest*;
- the uploaded file's *signature* and its *timestamp*;
- information revolving the *signing user* associated to the uploaded file – *display name* and *unique ID.*

For convenience, the uploaded file is shared within the CRDT by preliminarily embedding it in the metadata as a (serialized) *file blob.* Details on the file blob serialization and deserialization operations are illustrated in the next sub-section. Upon retrieval of CRDT update containing the new file entry, if the node's backend identifies a file blob embedded in the metadata, it deserializes it and saves it to local storage, and deletes the file blob from the metadata dictionary.

The metadata dictionary may also contain a timestamp indicating the expiration of the file signature's validity. This element is optional, as the user can choose to input it from the Web interface during the file upload process – or leave it blank, which indefinitely preserves the validity of the file signature itself.

An additional method is provided within the `FileSignatureStorage` class, which regulates the custom *data retention policy enforcement* routine, which triggers each time the class is instantiated – generally, this only occurs when the backend module is initially booted. The routine retrieves a configuration parameter that indicates the data retention interval for newly uploaded files – e.g., 90 days. For each file in the CRDT, the routine constructs a synthetic timestamp ($t_{retention}$) by using the file signature's timestamp, and offsetting it forwards by the retention interval identified previously. This synthetic timestamp is then compared with the current timestamp (t_{now}), which leads to one of two possible scenarios:

- if $t_{now} \geq t_{retention}$, this means that the file's retention has expired. Consequently, the file is removed from CRDT, and an update operation issued to all remote nodes.

- if $t_{now} < t_{retention}$, this means that the file's retention has not yet expired. No action is taken by the system in this case.

`UserStorage` Similar to what occurs for the `FileSignatureStorage` class, a dedicated CRDT key-value store is reserved within the `UserStorage` class, to collect information concerning the identities of registered users. For each user, an entry in the key-value store is present – with key equal to the given user's unique ID – with a Python dictionary attached to it containing the following user data.

- the user's *display name*;
- the user's *public key*;
- the *timestamp* associated to the registration of the user.

Fundamentally, the CRDT provided by this class is made available for quick access by any signature validity check operation, as it may occur that a user requires verification for a file shared by another user. By retrieving the signing user's unique ID, it can then access the CRDT to obtain the correct public key to use in the validity check operation.

Interoperability with the relay server Both the `FileSignatureStorage` and the `UserStorage` classes require connection with the relay server before any operation can be issued on their respective CRDT structures. This is possible thanks to a dedicated library, *pycrdt-websocket*[7], which extends the CRDT objects provided by the *pycrdt* library with the possibility of broadcasting CRDT updates to an available WebSocket server. Thus, during their respective initialization phases, both the classes mentioned above start with a blank CRDT key-value store, load the current state snapshot from local storage, if available, and initiate a connection to a given IP address and port, which points to the network's available relay server. If the connection is successful, the connected CRDT object is saved within the newly initialized class object, ready to be used for further operations. If the

[7]https://github.com/y-crdt/pycrdt-websocket

node fails to connect to the relay server over WebSocket, the node continues to function in an "offline-only" fashion – any operation issued during this period will be broadcast on the next successful connection to the relay server.

`file_utils.py` – File blob serialization and deserialization

File blobs that are placed in the CRDT key-value store employed by the `FileSignatureStorage` class are subject to custom serialization and deserialization procedures. For instance, the *serialization* operation takes the target file's binary data as input, and applies the open-source *LZ4* compression algorithm[8] to the retrieved data. The output of the compression algorithm is the payload that is embedded as the target file's blob in the metadata saved inside the CRDT. Conversely, the *deserialization* operation takes the file's blob as input, and decompresses it via the same LZ4 algorithm, returning the original file's content as output.

4.3.2 Frontend

CRDTSign's `api.py` module serves as the user's entry point to the application. Its main objective is to initialize an asynchronous RESTful API through the *Quart*[9] framework. Namely, the API instantiates several endpoints that can be accessed by interacting with the components of the deployed Web application.

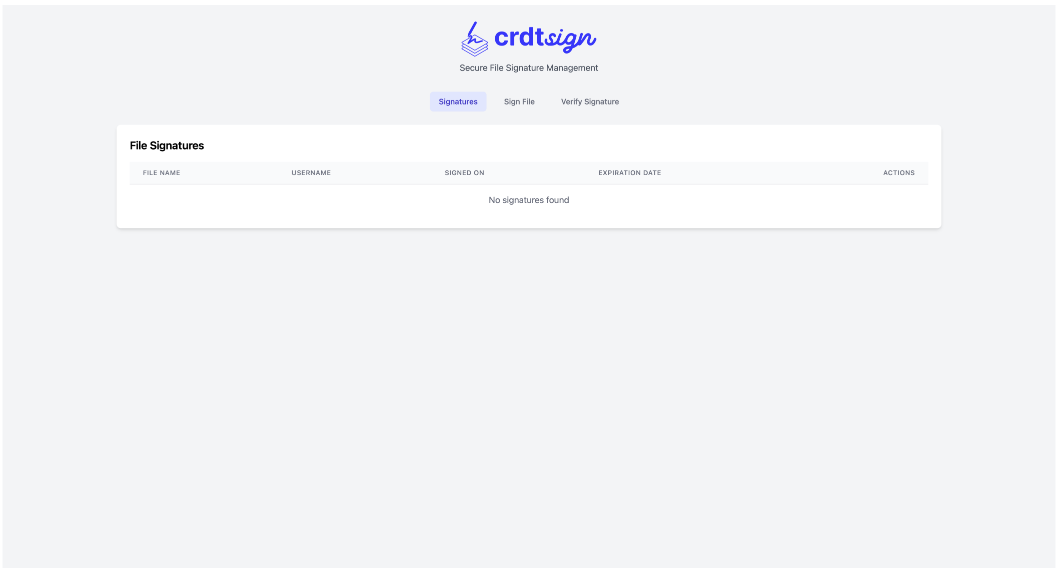
Root (/) The main entry point to the application's user interface (UI). By accessing this endpoint through a traditional Web browser, a dynamic HTML web page containing the UI is rendered on the client's machine[10].

Upon the first access to the root endpoint – performed by an un-registered

[8]https://lz4.org/

[9]https://quart.palletsprojects.com/en/latest/

[10]The dynamic web page is constructed by parsing pre-defined template files written using HTML syntax and special placeholders, which can be read by the *Jinja* templating engine for Python (`https://jinja.palletsprojects.com/en/stable/`)



File Signatures

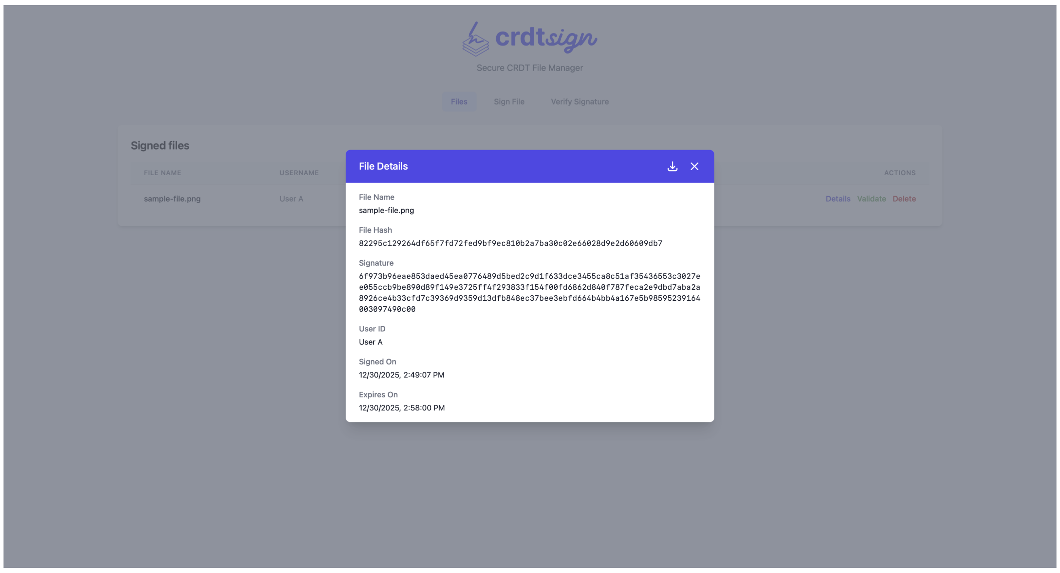
FILE NAME	USERNAME	SIGNED ON	EXPIRATION DATE	ACTIONS
No signatures found				

Figure 4.4. The main view for CRDTSign's frontend.

user – a new User object is created with a unique ID, and a form is displayed on the UI where the user can input the display name it wishes to appear under. After the user submits the inserted display name, the web application's main view is displayed – shown in Figure 4.4. The main view contains a table containing the signed files that are present in the files CRDT storage; for each row of the table, the following information is shown.

- The signed file's *name*.
- The signing user's *display name*.
- The signature's *creation timestamp*.
- The signature's *expiration timestamp* – if present.
- A set of *action buttons*, named *Details*, *Validate*, and *Delete* respectively.

The *Details* button opens a modal which contains further details on the selected file, such as the file's *hash digest*, its *signature*, and all the information mentioned above. An example of the modal being displayed can be found in Figure 4.5



File Details

File Name
sample-file.png

File Hash
82295c129264df65f7fd72fed9bf9ec810b2a7ba30c02e66028d9e2d60609db7

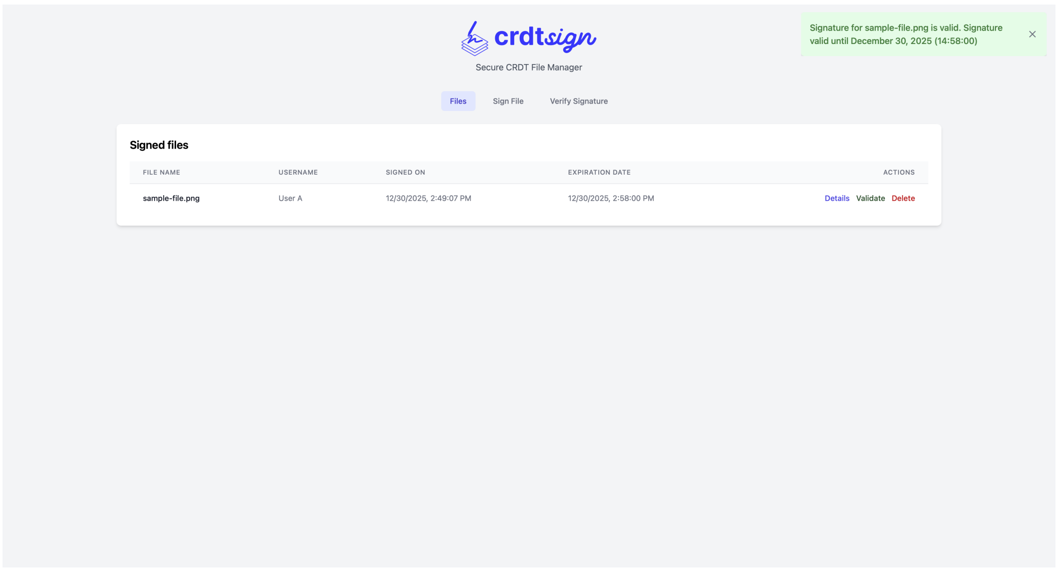
Signature
6f973b96eae853daed45ea0776489d5bed2c9d1f633dce3455ca8c51af35436553c3027e
e055ccb9be890d89f149e3725ff4f293833f154f00fd6862d840f787feca2e9dbd7aba2a
8926ce4b33cfd7c39369d9359d13dfb848ec37bee3ebfd664b4bb4a167e5b98595239164
003097490c00

User ID
User A

Signed On
12/30/2025, 2:49:07 PM

Expires On
12/30/2025, 2:58:00 PM

Figure 4.5. An example of "Details" modal that displays information for a sample file.



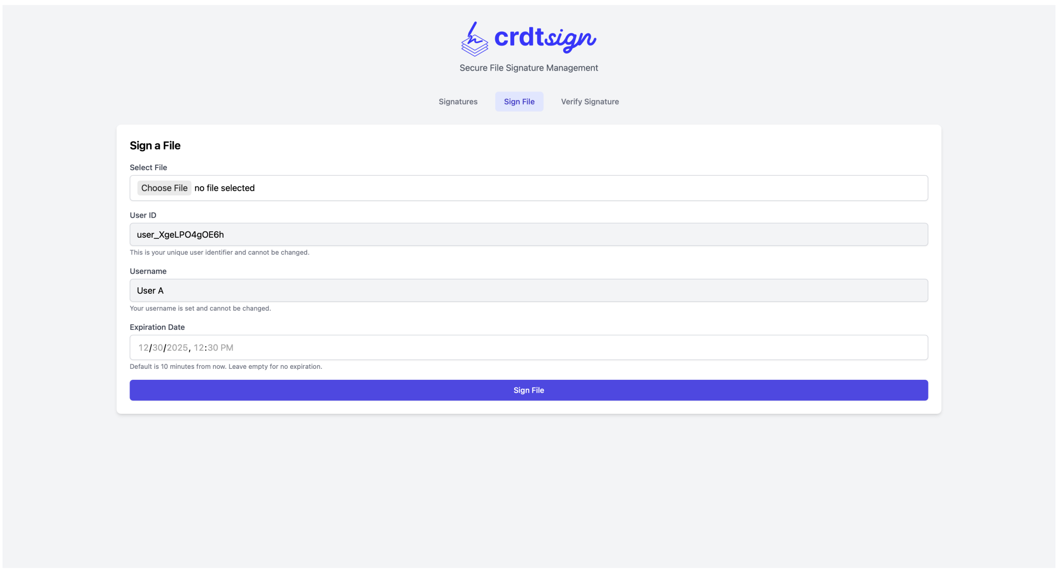
Signature for sample-file.png is valid. Signature valid until December 30, 2025 (14:58:00)

Signed files

FILE NAME	USERNAME	SIGNED ON	EXPIRATION DATE	ACTIONS
sample-file.png	User A	12/30/2025, 2:49:07 PM	12/30/2025, 2:58:00 PM	Details Validate Delete

Figure 4.6. The user is notified that the checked file signature is valid.

The *Validate* button issues a request to check the validity of the selected file's signature. This request is dispatched to the backend using the available methods, and the outcome of the check is then displayed in a pop-up notification appearing in the top-right corner of the web browser – as shown in Figure 4.6. If present, the user is also informed of the signature's validity expiration – date and time.


crdtsign

Secure File Signature Management

Signatures | Sign File | Verify Signature

Sign a File

Select File

Choose File no file selected

User ID

user_XgeLPO4gOE6h

This is your unique user identifier and cannot be changed.

Username

User A

Your username is set and cannot be changed.

Expiration Date

12/30/2025, 12:30 PM

Default is 10 minutes from now. Leave empty for no expiration.

Sign File

Figure 4.7. The form used to upload a new file to storage.

The *Delete* button is used to request the removal of the selected file. After further confirmation to proceeed with the operation is submitted by the user, a *delete* operation is issued on the local CRDT, which then broadcasts it to all of the remote nodes.

Furthermore, the main view presents a set of tab buttons that link to additional views. Namely, the *"Sign File"* view – Figure 4.7 – displays a form that allows the user to upload a new file to be signed and preserved in CRDT storage, and specify additional information, such as the date and the time of the signature's validity expiration.

Retrieve/upload files (`/api/signatures`) The endpoint that is linked to the signed files CRDT storage. Files can be retrieved cumulatively or selectively – if a file ID is specified in the request – or uploaded and signed via a `POST` request to the same endpoint. A request to remove a given file can also be performed on this endpoint via a `DELETE` request, by passing the target file's ID (`<file_id>`).

Validate file signature (`/api/validate/<file_id>`) The endpoint used to request a validity check for the file denoted by the passed ID (`<file_id>`)

and returns the outcome of such check.

4.3.3 Relay Server

The solution's relay server is used to emulate a pure-P2P environment among the nodes of the network. Its sole purpose is to keep track of connections that each node entertains with the server itself over the WebSocket protocol, and broadcast incoming CRDT update messages to the interested nodes. This is done by leveraging the *pycrdt* and *pycrdt-websocket* libraries, and using the latter library's `Store` object to keep a record of incoming update messages. This persistence mechanism is useful for synchronization purposes, as it enables a newly-connected node – or a known node that has suffered a given period of downtime – to be brought up to date with the current state of CRDT objects.

The relay server instantiates a set of *rooms*, each one being a container for a shared object. This means that each backend has to be routed to the correct room – one for the shared files, and one for the registered users – in order to share changes applied to the same objects.

The underlying relay server logic is then made accessible to a known IP address and port the solution's local network by using the open-source *Hypercorn*[11] library, which is used to deploy a standalone web server based on the ASGI[12] (Asynchronous Server Gateway Interface) standard.

4.4 Production Deployment

Although CRDTSign's software architecture is sub-optimal with respect to readiness for production environments, a few considerations have been made to better suit the deployment of the entire solution.

For instance, the Python package that contains the software components de-

[11]https://hypercorn.readthedocs.io/en/latest/

[12]https://github.com/django/asgiref/blob/main/specs/asgi.rst

scribed above was built using the open-source *uv*[13] package manager, which allowed for a seamless and efficient management of the required Python dependencies and of the build environment.

The solution then uses the popular containerization engine *Docker*[14] create lightweight, standalone containers used to deploy software components individually. More specifically, the Docker image that is installed in each deployed container is created using the following script – or *Dockerfile*.

Listing 4.1. Dockerfile

```dockerfile
FROM python:3.13-slim-bookworm
COPY --from=ghcr.io/astral-sh/uv:0.7.14 /uv /uvx /bin/

# Install uv
# The installer requires curl (and certificates) to
   download the release archive
RUN apt-get update \
    && apt-get install -y --no-install-recommends build
       -essential curl ca-certificates \
    && rm -rf /var/lib/apt/lists/* /usr/share/doc /usr/
       share/man \
    && apt-get clean

ADD https://astral.sh/uv/0.7.14/install.sh /uv-
   installer.sh

RUN sh /uv-installer.sh && rm /uv-installer.sh

# Ensure the installed binary is on the `PATH`
ENV PATH="/root/.local/bin/:$PATH"

# Install the project
# Copy the project into the image
ADD crdtsign/core/. /app
```

[13]https://docs.astral.sh/uv/
[14]https://www.docker.com/

```

# Sync the project into a new environment, asserting
    the lockfile is up to date
WORKDIR /app
RUN uv sync --locked

# The web application is deployed on port 5000
EXPOSE 5000

# The server runs on port 8765
EXPOSE 8765

# Run the project [default: server]
CMD [ "uv", "run", "crdtsign", "server", "--host", "
    0.0.0.0", "--port", "8765" ]
```

Once the image corresponding to the above Dockerfile is created, it is used to deploy a set of containers that utilize CRDTSign. Moreover, the solution was tested by invoking the Docker Compose commmand to deploy the cluster of containers specified by the following configuration file – named `docker-compose.yml` – which deploys a simple cluster made up of four containers; one for the relay server, and the others for the "user" nodes.

Listing 4.2. docker-compose.yml

```
services:
  server:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8765:8765
    container_name: crdtsign-server
    networks:
      - crdtsign-network
    command: uv run crdtsign server --host 0.0.0.0 --
```

```yaml
         port 8765
    environment:
      - PYTHONBUFFERED=1
    healthcheck:
      test: ["CMD", "python", "-c", "import socket; s=socket.socket(); s.settimeout(1); s.connect(('localhost', 8765)); s.close()"]
      interval: 5s
      timeout: 3s
      retries: 5
      start_period: 5s

  client-1:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 5001:5000
    container_name: crdtsign-client-1
    networks:
      - crdtsign-network
    depends_on:
      server:
        condition: service_healthy
    command: uv run crdtsign app --host 0.0.0.0
    environment:
      - PYTHONBUFFERED=1
      - IS_CONTAINER=true
    volumes:
      - /etc/localtime:/etc/localtime

  client-2:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 5002:5000
    container_name: crdtsign-client-2
    networks:
      - crdtsign-network
    depends_on:
      server:
        condition: service_healthy
    command: uv run crdtsign app --host 0.0.0.0
    environment:
      - PYTHONBUFFERED=1
      - IS_CONTAINER=true
    volumes:
      - /etc/localtime:/etc/localtime

  client-3:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 5003:5000
    container_name: crdtsign-client-3
    networks:
      - crdtsign-network
    depends_on:
      server:
        condition: service_healthy
    command: uv run crdtsign app --host 0.0.0.0
    environment:
      - PYTHONBUFFERED=1
      - IS_CONTAINER=true
    volumes:
      - /etc/localtime:/etc/localtime

networks:
  crdtsign-network:
    driver: bridge
```

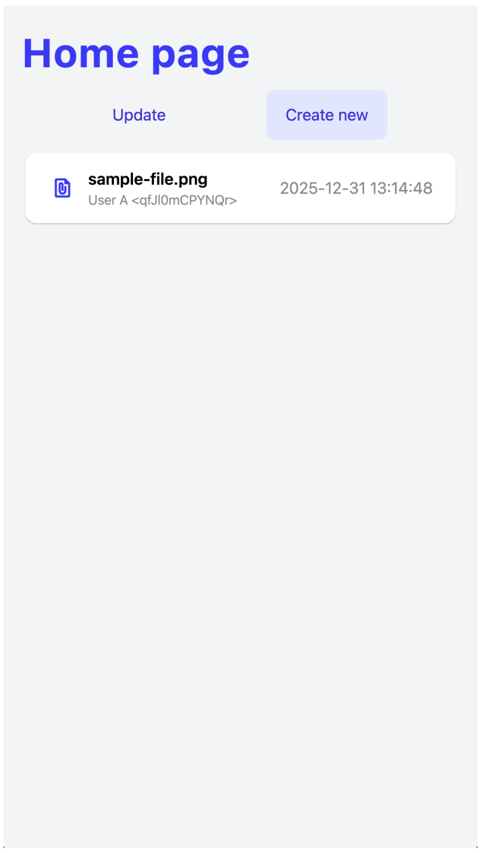
4.5 Development of the Complementary Mobile Application

As an addendum to the provided solution, a prototype mobile frontend application was developed to wrap around the present backend logic. The application was written using the *Flet* framework[15], which offers a simple environment for building mobile and web applications, using Python syntax to construct user interfaces. Software that is developed with Flet can be compiled using Flet's custom build system, which translates it into code that is supported by the open-source UI software development kit *Flutter*[16]. Flutter's capabilities are then leveraged to generate a package that can be executed – or installed – on a wide variety of devices – i.e., desktop operating systems such as Windows, Linux, or macOS, or mobile operating systems such as Android or iOS.

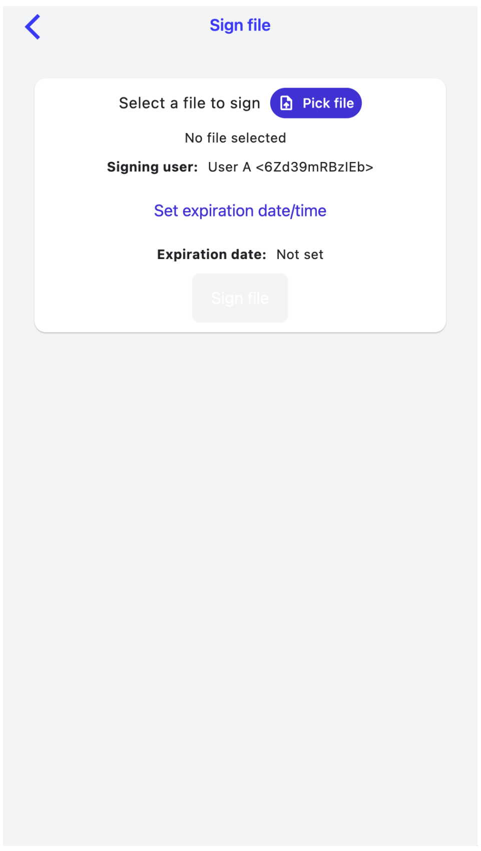
Through Flet, an application was developed – named `crdtsign-mobile`, for distinction with the main application – to mimic the functionalities that are already present in the main frontend application described in subsection 4.3.2. Namely, the mobile application is divided into two principal views that the user can navigate between:

- a *"home"* view – Figure 4.8a – where the user can explore the available signed files from CRDT storage, and for each perform a common set of actions, such as viewing relevant details for a file, requesting a validity check for a file, or remove a file from shared storage;
- a *"create"* view – Figure 4.8b – where the user can upload a file to be signed and shared within the shared storage.

[15]https://flet.dev/
[16]https://flutter.dev/



(a) A screenshot of the *"home" view* for the CRDTSign mobile application.



(b) A screenshot of the *"create" view* for the CRDTSign mobile application.

CHAPTER 5

Evaluation of the Proof-of-Concept Application and Conclusion

5.1 Summary of the Proof-of-Concept Application

The aim of this thesis was to illustrate Conflict-free Replicated Data Types and their utility in modern distributed software. This implied developing a Proof-of-Concept application – CRDTSign – a replicated file storage solution that enables a group of users to collaborate on a shared file storage. Each user has the possibility of uploading a file – and certifying ownership on it through the means of a cryptographic signature – and receiving files from other peers. Thanks to CRDTs, the PoC has the capability to operate in a decentralized manner – each operation can be individually applied from each node, and the system's shared state is kept (eventually) consistent, without the need of a central node that coordinates operations and the order in which they are applied. This is an improvement over similar applications that leverage synchronization technologies such as Operational Transformation (OT), which elects a central node to manage operations for the rest of the network, with the goal of resolving possible conflicts.

In the following sections, we will provide a preliminary evaluation on the PoC's behavior with respect to *scalability*, and then discuss some areas of improvement over the currently developed solution.

5.2 Evaluation of the Proof-of-Concept Application

While the behavior of CRDTSign with respect to functional and tecnhnical requirements has been established, its viability in a production-like environment depends on its scalability profile. Consequently, this section aims to introduce a preliminary evaluation of the solution that was performed with an experimental setup, and the resulting metrics that were measured throughout extensive testing. Through this analysis, we will discuss how the system behaves as demand scales, and assess the throughput and latency requirements achieved by the proposed solution.

5.2.1 Overview of the experimental setup for scalability testing

Two types of operations were the focus of the performed experiments – *add* and *remove* operations. The overall objective of the experiments was to simulate real-world use cases with the maximum possible fidelity; in order to do so, the following mock scenarios were defined.

***Add* operations** A new user registers to CRDTSign, has a newly created cryptographic keypair assigned to it, and *adds* a new file to shared storage – signing it with its own private key. After updating the local CRDT storage, an update message reflecting the change in state is generated and transmitted to all replicas connected to the network. Each peer receiving the update message individually applies the update to its own local CRDT storage, adding the target file and retrieving all the data required to perform a validity check over the new file's signature – the file's hash digest, the file's signature, and the signing user's public key. Once the peer successfully assesses the signature's validity, it is considered "up-to-date" with respect to the CRDT's state. When all replicas finish assessing the signature's validity – and all checks return a positive outcome – the entire system has *converged* to a consistent state.

***Remove* operations** A user requests *removal* of a file already present in the shared storage. After the file is removed in the local CRDT storage, an update message reflecting the change in state is generated and transmitted to all replicas connected to the network. Each peer receiving the update message individually applies the update to its own local CRDT storage, removing the target file from storage. Each replica then checks whether the file still exists in storage. Once the peer successfully assesses that the target file is no longer present in its current storage, it is considered "up-to-date" with respect to the CRDT's state. When all replicas finish assessing the removed file's presence – and all checks return a negative outcome – the entire system has *converged* to a consistent state.

Once these scenarios were defined, they were tested within an architecture made up of two types of node:

- a *writer node*, which is the source node in charge of producing the *test file* that is shared over the network, and which produces the corresponding CRDT update messages that are then received by another type of node;
- a *reader node*, which awaits the update messages produced and transmitted by the writer node, applies the update to its local storage, and assesses the correctness of the resulting change in state.

The architecture for each performed experiment consists of exactly one writer node, and a variable number of reader nodes – this number represents the *scaling* parameter of each experiment. Technically speaking, each node is deployed within its dedicated Docker container, which runs a dedicated sequential Python – depending on the node type – that simulates both scenarios on a pre-determined test file. Each script is also in charge of logging specific timestamps, which are then used to retrieve the necessary test metrics. Namely, we define a measure of an operation's *latency* as the time interval that occurs between the generation of the operation's corresponding update message – writer node – and the successful assessment of correctness of the state that is produced by applying the same update on an individual remote site – reader node.

Once all the measurements for the latency of a given operation are gathered from all deployed nodes, we select the highest latency that was produced throughout the conducted experiment. We denote this latency as a measure of the system's *convergence time* for the given operation – i.e., the time it takes for all nodes to converge to an identical state, once the tested operation is transmitted over the network. Each experiment registered the convergence time for both *add* and *remove* operations applied to the same test file. The experiment was then repeated over multiple rounds, where each rounds consists of multiple iterations over the same experiment – e.g., 10 iterations per round – performed with a specific scaling parameter selected. For example, if the scaling parameter was set to 5, it means that the experiments were conducted over a network of 6 nodes – one writer node and 5 reader nodes. Due to constraints that will be discussed in the following sections, the experiments were bounded to testing all scaling parameter values between 1 and 16.

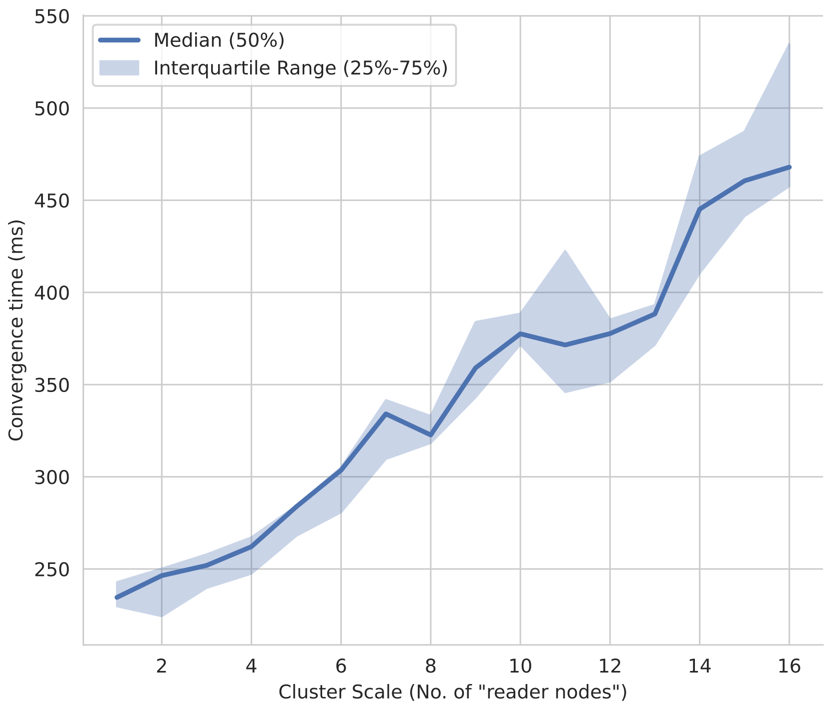
The setup was then deployed through the Docker Compose command, which was used to instantiate one writer node and a desired number of reader nodes – specified through the `scale` flag. The following YML configuration file was used for this purpose.

Listing 5.1. docker-compose.yml

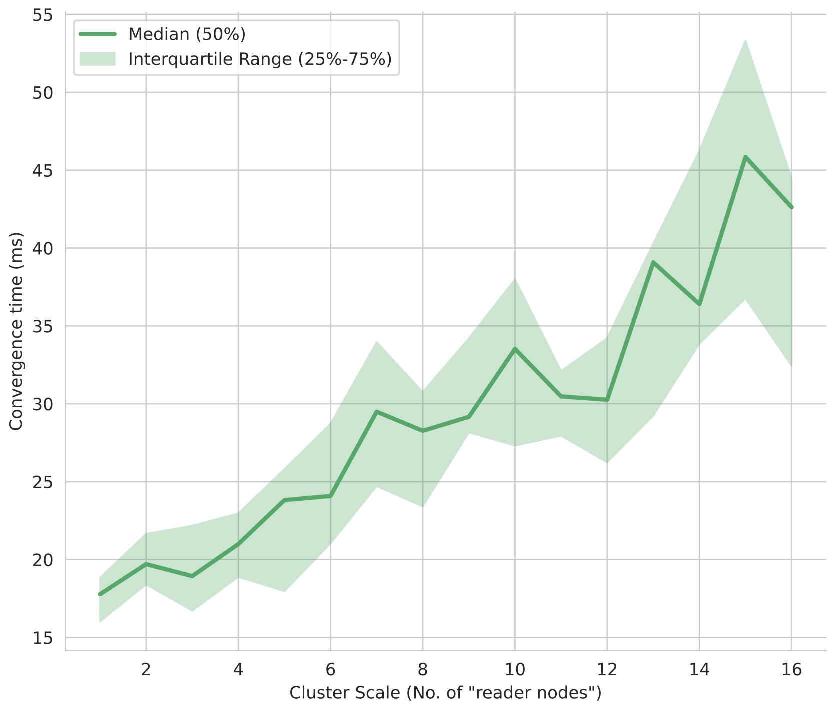
```yaml
services:
  server:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8765:8765
    container_name: crdtsign-server
    networks:
      - crdtsign-network
    command: uv run crdtsign server --host 0.0.0.0 --
       port 8765
    environment:
      - PYTHONBUFFERED=1
    healthcheck:
      test: ["CMD", "python", "-c", "import socket; s=
         socket.socket(); s.settimeout(1); s.connect(('
         localhost', 8765)); s.close()"]
      interval: 5s
      timeout: 3s
      retries: 5
      start_period: 5s


  scale-reader:
    build:
      context: .
      dockerfile: Dockerfile
    # container_name: scale-reader
    networks:
      - crdtsign-network
    depends_on:
      server:
        condition: service_healthy
    command: uv run test-scalereader
    environment:
      - PYTHONBUFFERED=1
      - IS_CONTAINER=true
    volumes:
      - /etc/localtime:/etc/localtime
      - ./crdtsign/core/tests/artifacts:/app/tests/
         artifacts

  scale-writer:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: scale-writer
    networks:
      - crdtsign-network
    depends_on:
      server:
        condition: service_healthy
    command: uv run test-scalewriter
    environment:
      - PYTHONBUFFERED=1
      - IS_CONTAINER=true
    volumes:
      - /etc/localtime:/etc/localtime
      - ./crdtsign/core/tests/artifacts:/app/tests/artifacts

networks:
  crdtsign-network:
    driver: bridge
```

5.2.2 Analysis of the achieved results

We now present the results of the experiments described previously, as shown in Figure 5.1a and Figure 5.1b, which illustrate the (aggregated) convergence times registered on experiments concerning *add* and *remove* operations, respectively. Namely, each figure illustrates a line plot that is centered on the *median* distribution (P_{50}) of the operation's convergence time at increasing values of scale, with a shaded area denoting the *interquartile range* (IQR) for the registered convergence time, in a span between 25% (P_{25}) and 75% (P_{75}). While the median tells us how a certain operation typically takes for a given scale, the IQR serves as a proxy for the system's stability – i.e., a stable system maintains a tight bound, regardless of scale.



(a) Experiment results for *add* operations



(b) Experiment results for *remove* operations

Figure 5.1. Plots for the conducted experiments

Given the samples provided, the results illustrated by both figures exhibit a scaling behavior that follows an approximately linear growth, which is the expected behavior from this type of systems. It is important to note that all experiments presented above were conducted by leveraging consumer-grade hardware, where all the nodes of the cluster run on separate (virtual) containers – but still bounded to the same machine. Employing a distributed cluster of dedicated machines would help in further optimizing convergence times, and in opening the possibility of conducting experiments on a larger scale – e.g., on clusters made up of 100 or more replicas. Despite these factors, the system's response time consistently remains below the 1-second (1000ms) threshold. Given the functionalities of the system, the presented results denote a system that is aligned with the user's expectations [**Doherty2015Keeping**] of interaction with the system itself and its percieved responsiveness.

5.3 Considerations for Future Work

Over the course of this thesis, the main concepts surrounding CRDTs and their applicability in modern software development were explored. Furthermore, the advantages of adopting CRDTs in modern, distributed software were demonstrated through the design and implementation of a Proof-of-Concept application. Through this application, a group of users can share a set of files, which is replicated across multiple nodes for high availability – in case of failure of one node, the same information can be retrieved from other nodes. The operations that the PoC supports are executed in a decenteralized manner, without the need of having a central authority dictate the necessary synchronization strategy and conflict resolution mechanism for the entire architecture.

There exists a greater number of practical applications for CRDTs than the ones that were introduces throughout this thesis. Aside from multiple variations on collaborative editing – where popular applications such as *Zed*[1] and *Figma*[2] employ CRDTs or an approach inspired by them – it is possible to find CRDTs applied to distributed databases, such as *Riak* [**Klophaus2010Riak**], *Redis* [**Redis_crdts**], and *AntidoteDB* [**AntidoteDB_docs**], and in a wider range of decentralized applications, such as social networks [**Farcaster_specs**], and even Blockchains [**Nasirifard2023Orderless**].

As for the use case that was discussed in previous chapters, there is still room for improvement of the solution fleshed out by the development of CRDT-Sign. For instance, the currently developed solution is limited to the tooling that is already available for use within the Python environment – i.e., the *pycrdt* library for constructing CRDT structures, and the *pycrdt-websocket* library for sharing updates over the WebSocket protocol. A more robust solution can be achieved by adopting specialized P2P protocols designed for file synchronization, such as the *Dat* [**Ogden2017Dat**] Protocol. Although a more dedicated protocol would require careful handling of shared files and their metadata, this type of improvement opens up the possibility of extending the functionalities that were defined, and to transform a solution such as CRDTSign into a truly-decenteralized, replicated file system, which is capable of providing more complex functionalities – e.g., organizing files in hierarchical folders, renaming files, version control, and so on.

[1]https://zed.dev/blog/crdts
[2]https://www.figma.com/blog/how-figmas-multiplayer-technology-works/

Bibliography

[1] Mohammad Hamdaqa and Ladan Tahvildari. "Cloud Computing Uncovered: A Research Landscape". In: *Advances in Computers* 86 (Dec. 2012). DOI: 10.1016/B978-0-12-396535-6.00002-8.

[2] C. A. Ellis and S. J. Gibbs. "Concurrency control in groupware systems". In: *SIGMOD Rec.* 18.2 (June 1989), pp. 399–407. ISSN: 0163-5808. DOI: 10.1145/66926.66963.

[3] Martin Kleppmann and Peter Alvaro. "Research for practice: convergence". In: *Commun. ACM* 65.11 (Oct. 2022), pp. 104–106. ISSN: 0001-0782. DOI: 10.1145/3563901. URL: https://doi.org/10.1145/3563901.

[4] Chengzheng Sun and Clarence Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements". In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW '98. Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 59–68. ISBN: 1581130090. DOI: 10.1145/289444.289469.

[5] Paulo Sérgio Almeida. "Approaches to Conflict-free Replicated Data Types". In: *ACM Comput. Surv.* 57.2 (Nov. 2024). ISSN: 0360-0300. DOI: 10.1145/3695249.

[6] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[7] Leslie Lamport. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems". In: *ACM Transactions on Programming Lan-*

guages and Systems (Apr. 1984), pp. 254–280. DOI: 10.1145/2993.2994.

[8] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing.* PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7. ISBN: 1581131836. DOI: 10.1145/343477.343502.

[9] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (June 2008). ISSN: 0734-2071. DOI: 10.1145/1365815.1365816.

[12] Yasushi Saito and Marc Shapiro. "Optimistic replication". In: *ACM Comput. Surv.* 37.1 (Mar. 2005), pp. 42–81. ISSN: 0360-0300. DOI: 10.1145/1057977.1057980.

[13] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. "Session guarantees for weakly consistent replicated data". In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems.* 1994, pp. 140–149. DOI: 10.1109/pdis.1994.331722.

[14] Nuno Preguiça. *Conflict-free Replicated Data Types: An Overview.* 2018. arXiv: 1806.10254.

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free replicated data types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems.* SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 9783642245497. DOI: 10.1007/978-3-642-24550-3_29.

[16] Brian Davey and Hilary Priestley. *Introduction to Lattices and Order.* 2nd ed. Cambridge University Press, 2002. DOI: 10.1017/CBO9780511809088.

[17] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://inria.hal.science/inria-00555588.

[18] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing.* PODC '87. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1987, pp. 1–12. ISBN: 089791239X. DOI: 10.1145/41840.41841.

[19] Kenneth P. Birman and Thomas A. Joseph. "Reliable communication in the presence of failures". In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), pp. 47–76. ISSN: 0734-2071. DOI: 10.1145/7351.7478.

[20] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Efficient State-based CRDTs by Delta-Mutation". In: *CoRR* abs/1410.2803 (2014). DOI: 10.48550/arXiv.1410.2803. arXiv: 1410.2803.

[21] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. "Replicated Abstract Data Types: Building Blocks for Collaborative Applications". In: *Journal of Parallel and Distributed Computing* 71.3 (Mar. 2011), pp. 354–368. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2010.12.006.

[22] Loıck Briot, Pascal Urso, and Marc Shapiro. "High Responsiveness for Group Editing CRDTs". In: *19th International Conference on Supporting Group Work*. GROUP 2016. ACM, Nov. 2016, pp. 51–60. DOI: 10.1145/2957276.2957300.

[23] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems". In: *ACM Trans. Comput.-Hum. Interact.* 5.1 (Mar. 1998), pp. 63–108. ISSN: 1073-0516. DOI: 10.1145/274444.274447.

[24] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. "Data consistency for P2P collaborative editing". In: *20th ACM Conference on Computer Supported Cooperative Work*. CSCW 2006. ACM, Nov. 2006, pp. 259–268. DOI: 10.1145/1180875.1180916.

[25] Stephane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks". In: *29th IEEE International Conference on Distributed Computing Systems*. ICDCS 2009. IEEE, 2009, pp. 404–412. DOI: 10.1109/ICDCS.2009.75.

[26] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types". In: *15th International Conference on Web Engineering*. ICWE 2015. Springer LNCS volume 9114, June 2015, pp. 675–678. DOI: 10.1007/978-3-319-19890-3_55.

[27] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types". In: *19th International Conference on Supporting Group Work*. GROUP 2016. ACM, Nov. 2016, pp. 39–49. DOI: 10.1145/2957276.2957310.

[28] Yjs Team. *Yjs Documentation*. Accessed: 2025-12-10. URL: https://docs.yjs.dev.

[29] Martin Kleppmann and Alastair R Beresford. "Automerge: Real-time data sync between edge devices". In: URL: https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf.

[30] Automerge Team. *Automerge Documentation*. Accessed: 2025-12-10. URL: `https://automerge.org/docs`.

[31] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455.

[32] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*. CHES'11. Nara, Japan: Springer-Verlag, 2011, pp. 124–142. ISBN: 9783642239502. DOI: 10.5555/2044928.2044940.

[33] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. DOI: 10.17487/RFC8032.