

Authenticated Conflict-free Replicated Data Types

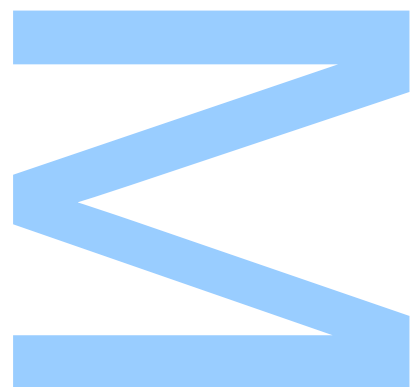
João Azevedo

Master in Computer Science

Department of Computer Science

Faculty of Sciences of the University of Porto

2025



Authenticated Conflict-free Replicated Data Types

[João Azevedo](#)

Dissertation carried out as part of the Master in Computer Science

[Department of Computer Science](#)

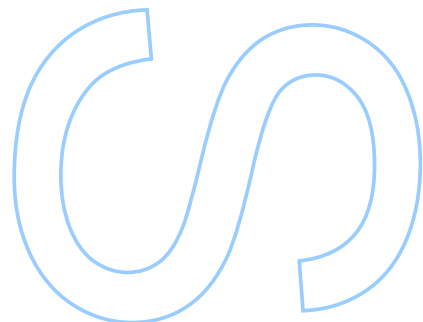
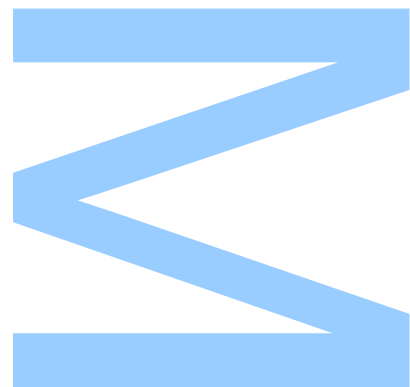
2025

Supervisor

[Prof. Bernardo Portela](#), Faculty of Sciences, University of Porto

Co-supervisor

[Prof. Hugo Pacheco](#), Faculty of Sciences, University of Porto



Acknowledgements

I would like to express my special appreciation and gratitude to my advisors, Professor Bernardo Portela and Professor Hugo Pacheco, who introduced me to the research and provided me with countless opportunities for personal and professional development. I would also like to thank them for all their dedication throughout the development of this thesis.

I would also like to thank my entire family, especially my parents and grandmother, who were there for me whenever I needed them, and not forgetting my sisters, with whom I shared wonderful moments. Thank you all for your love.

Finally, I would like to thank my friends with whom I was able to exchange ideas and who provided me with distractions to clear my mind.

Resumo

Os sistemas distribuídos requerem estruturas de dados que sejam consistentes sob concorrência e verificáveis quanto à autenticidade. As estruturas autenticadas tradicionais, embora ofereçam fortes garantias de integridade, sofrem frequentemente de crescimento linear de estado e elevada sobrecarga computacional, limitando o seu uso prático. Neste trabalho, apresentamos um Authenticated Conflict-free Replicated Data type (Auth-CRDT), uma composição de um Conflict-free Replicated Data Type (CRDT) com um Authenticated Merkle Directed Acyclic Graph (Auth-MDAG), que mantém a convergência e a verificabilidade enquanto controla o crescimento de estado. O nosso projeto transfere as garantias de consistência para o CRDT subjacente, permitindo que a estrutura autenticada apenas acompanhe as atualizações pendentes, reduzindo as sobrecargas computacionais e de rede.

Para avaliar esta abordagem, desenvolvemos uma série de implementações incrementais, cada uma refletindo diferentes compensações entre autenticidade, gestão de estado e desempenho. Através de experiências práticas, mostramos que a introdução da autenticação incorre em sobrecarga dentro do esperado para técnicas criptográficas e que as operações de consulta podem ser otimizadas. Para além disso, delegar a lógica das réplicas para um sistema existente não introduz custos de eficiência mensuráveis, sugerindo que o projeto proposto pode ser integrado na prática.

Estas descobertas demonstram que o Auth-CRDT fornece uma solução para cenários em que a integridade e a consistência são necessárias.

Palavras-chave: Tipos de dados replicados sem conflito; Consistência eventual; Tipos de dados replicados; Tipos de dados autenticados; Computação verificável

Abstract

Distributed systems require data structures that are both consistent under concurrency and verifiable for authenticity. Traditional authenticated structures, while providing strong integrity guarantees, often suffer from linear growth in state and high computational overhead, limiting their practical use. In this work, we introduce an Authenticated Conflict-free Replicated Data type (Auth-CRDT), a composition of a Conflict-free Replicated Data Type (CRDT) front end with an Authenticated Merkle Directed Acyclic Graph (Auth-MDAG), which maintains convergence and verifiability while controlling state growth. Our design shifts consistency guarantees to the underlying CRDT, allowing the authenticated structure to track only pending updates, reducing computation and network overheads.

To evaluate this approach, we developed a series of incremental implementations, each reflecting different trade-offs between authenticity, state management, and performance. Through benchmarks, we show that introducing authentication incurs overhead within the expected range for cryptographic techniques, and that query operations can be optimized. Furthermore, delegating replica logic to an existing system does not introduce measurable efficiency costs, suggesting that the proposed design can be integrated in practice.

These findings demonstrate that Auth-CRDT provides a solution for scenarios where both integrity and consistency are required. **Keywords:** Conflict-free Replicated Data Types; Eventual Consistency; Replicated Data Types; Authenticated Data Types; Verifiable Computation

Table of Contents

List of Figures	vi
List of Abbreviations.....	vii
1. Introduction.....	1
2. Preliminaries.....	5
2.1. Cryptographic Authentication Techniques	5
2.1.1. Hash Functions and Cryptographic Hash Functions	5
2.1.2. Message Authenticated Code	7
2.2. Replication and Distributed Systems	8
2.2.1. Message Ordering	8
2.2.2. Directed Acyclic Graph.....	9
2.3. Authenticated Data Type	11
2.3.1. Merkle Tree	12
2.3.2. Merkle Directed Acyclic Graph.....	14
3. State Of the Art	16
3.1. Conflict-free Replicated Data Types.....	16
3.1.1. State-based Convergent Replicated Data Type	18
3.1.2. Operation-based Commutative Replicated Data Type.....	19
3.1.3. Delta-Mutation Convergent Replicated Data Type.....	20
3.2. Proof-carrying Data	22
3.3. Stateful Multi-client Verifiable Computation	24
4. Replicated Authenticated Data Type.....	26
4.1. Replicated Authenticated Data Type	26
4.2. Authenticated Merkle Directed Acyclic Graphs	28
4.3. Authenticated Conflict-free Replicated Data type	30
5. Implementation.....	33
5.1. Implementation Interface	34
5.2. Base implementation.....	36
5.2.1. Cursor	36
5.2.2. Node	36
5.2.3. MerkleDag	37
5.2.3.1. Insertion Operations.....	37
5.2.3.2. Query Operations	39
5.2.3.3. Merge and Messages	39

5.3. Authenticated implementation.....	40
5.3.1. Node.....	40
5.3.2. MerkleDag	41
5.4. Caching Optimization	42
5.5. Matrix	47
6. Experimental Setup	49
6.1. Benchmarks Architecture.....	49
6.2. System Configuration and Reproducibility.....	49
6.3. Micro benchmarking	50
6.3.1. <code>baseline</code>	51
6.3.2. <code>authenticated</code>	52
6.3.3. <code>cached</code>	53
6.3.4. Micro Benchmark Results Discussion	53
6.4. Macro benchmarking.....	54
6.4.1. Workload Generation	55
6.4.2. <code>baseline</code>	56
6.4.3. <code>authenticated</code>	56
6.4.4. <code>cached</code>	57
6.4.5. <code>matrix</code>	58
6.4.6. Discussion.....	59
7. Conclusion and Future Work.....	61
Bibliography	62
A. Full Graphs.....	71
A.1. Micro Benchmarks	71
A.2. Macro Benchmarks.....	72

List of Figures

2.1. An example of a DAG where the heads are highlighted.	10
2.2. A system whose events are ordered through causal ordering, and the DAG that results from the ordering.....	10
2.3. Example of how the root hash of a Merkle tree is calculated.	13
2.4. Example of a Merkle-DAG with five nodes, showing data D_i , hashes H_i , and parent-child dependencies.	15
3.1. Grow-only Counter (G-Counter) Specification.....	19
3.2. Op-based CRDT Observed-Remove Set (OR-Set) Specification.....	20
3.3. OR-Set delta-based CRDT example.	21
3.4. Example of a PCD system.	23
4.1. RADT environment: client-replica interaction(top), replica-replicas interaction(bottom). 28	
4.2. Auth-MDAG-based algorithms implementing the RADT API: client interface (top) and replica interface (bottom).....	30
5.1. Architecture of our implementations.....	33
5.2. An example system DAG with seven nodes labelled 1 through 7.....	43
5.3. Minimal example of how layer property works, the labels of each node are their layer value.....	44
6.1. Architecture of benchmarks.....	49
6.2. Micro benchmarks results of the baseline implementation.....	51
6.3. Micro benchmark results for authenticated implementation.	52
6.4. Micro benchmarks results of the cached implementation.....	53
6.5. Comparison of the micro benchmark results of the different implementations.	54
6.6. Macro benchmark results for baseline	56
6.7. Macro benchmark results for Authless.....	57
6.8. Macro benchmark results for cached	57
6.9. Macro benchmark results for matrix	58
6.10. Comparison of the macro benchmark results of the different implementation.	59
A.1. Micro benchmarks results of the baseline implementation.....	71
A.2. Micro benchmark results for authenticated implementation.	71
A.3. Micro benchmarks results of the cached implementation.....	71
A.4. Macro benchmark results of the baseline implementation.	72
A.5. Macro benchmark results of the authenticated implementation.....	72
A.6. Macro benchmark results of the cached implementation.	72
A.7. Macro benchmark results of the matrix implementation.	72

List of Abbreviations

- ADT** Authenticated Data Type. iv, 1, 3, 5, 11, 12, 24
- API** Application Programming Interface. vi, 26, 27, 30, 33, 36, 37, 48
- Auth-CRDT** Authenticated Conflict-free Replicated Data type. ii–iv, 3, 26, 30, 31, 61, 62
- Auth-MDAG** Authenticated Merkle Directed Acyclic Graph. ii–iv, vi, 2, 3, 26–34, 39–41, 48, 61, 62
- CHF** Cryptographic Hash function. 6–8
- CMAC** Cipher-based MAC. 8
- CRDT** Conflict-free Replicated Data Type. ii–iv, 1–3, 16–19, 21, 22, 26, 30–34, 36, 48, 50, 61, 62
- DAG** Directed Acyclic Graph. vi, 2, 3, 5, 8–11, 14, 15, 28–30, 36, 48, 51, 62
- delta-based CRDT** Delta-Mutation Convergent Replicated Data Type. iv, vi, 18, 20–22
- FIFO** First-In-First-Out. 8, 9
- G-Counter** Grow-only Counter. 19, 31
- HF** Hash function. 5
- HMAC** Hash-Based MAC. 8
- IPFS** InterPlanetary File System. 2, 15
- MAC** Message Authenticated Code. iv, 2, 3, 5, 7, 8, 26, 28, 29, 31, 34, 36, 40, 41, 44, 54, 57, 59
- MEG** Matrix Event Graph. 3, 48, 61
- Merkle-DAG** Merkle Directed Acyclic Graph. vi, 2, 12, 14, 15, 26, 28, 33, 37

- op-based CRDT** Operation-based Commutative Replicated Data Type. iv, vi, 18–22, 32
- OR-Set** Observed-Remove Set. vi, 20, 21
- PCD** Proof-carrying Data. iv, vi, 16, 22–24
- RADT** Replicated Authenticated Data Type. iv, vi, 2, 3, 26–28, 30, 61
- RDD** Resilient Distributed Dataset. 11
- RDT** Replicated Data Type. 17
- SEC** Strong Eventual Consistency. 1–3, 22, 30
- state-based CRDT** State-based Convergent Replicated Data Type. iv, 18–22, 31, 32
- TCP** Transmission Control Protocol. 9, 20
- UDP** User Datagram Protocol. 8
- UMAC** Universal MAC. 8

1. Introduction

Cloud services are widely used for outsourcing data and computation due to their competitive pricing and immediate availability. They enable online collaboration between multiple clients in applications such as file storage, office suites, and domain-specific software solutions. As a result, many users rely on remote service providers instead of local storage. To improve performance and availability, data is often replicated across multiple cloud servers, which introduces additional challenges.

By delegating data and computation to the cloud, clients relinquish direct control, making the cloud the effective authority over their data. In other words, users must trust the storage provider, which can access and subtly modify data without consent or notice. Various approaches have been proposed to address specific security concerns. With a particular focus on authenticity and integrity, this involves ensuring that users can easily verify their data.

In cryptography, outsourcing data to an untrusted server while guaranteeing integrity is studied under verifiable computation [1]. A key efficiency metric is the succinctness of proofs, ensuring that the cryptographic material required for verification remains independent of the dataset size. One approach utilizing verifiable computation is Authenticated Data Type (ADT) [2], where an untrusted server processes queries and updates on a data structure and provides cryptographic proofs of correctness. While most ADT research focuses on single-client scenarios, [3] extends this model to multiple mutually trusting clients interacting with a server using shared digital signatures. However, existing ADT solutions do not generalize to scenarios where data is replicated across multiple servers. The ADT model of [3] relies on a lock-step protocol, where the server processes client updates sequentially, which entails strong consistency in the absence of malicious server behaviour. In theory, a concurrent variant of the protocol could allow clients to concurrently operate at their own pace, though readers must always wait for conflicting writes [4]. However, when multiple servers are involved, achieving global synchronization without central coordination is infeasible.

Conflict-free Replicated Data Types (CRDTs) [5] are a popular approach designed for replicated state in peer-to-peer settings, providing weaker guarantees known as Strong Eventual Consistency (SEC). Internally, CRDT replicas maintain event metadata akin to

Directed Acyclic Graph (DAG). By integrating Merkle Directed Acyclic Graph (Merkle-DAG), CRDTs can tolerate Byzantine faults [6]. Still, CRDT approaches seldom adopt a clear client-server model with distinct trust assumptions, and when they consider Byzantine behaviour, they are typically concerned with replicas compromising SEC and normal system operation. Digital signatures are often used for authentication, ensuring that only trusted entities can issue updates. However, in CRDTs, updates are typically generated by replicas rather than clients, leaving clients vulnerable when interacting with malicious replicas.

In distributed systems, decentralized data storage and group communication models often rely on replicated event histories. To ensure integrity and tolerate Byzantine faults, Merkle trees [7] link events via recursive hash commitments. These structures underpin blockchains [8, 9] (with consensus mechanisms enforcing a valid head) and are generalized in source control systems like Git [10] (where merging divergent histories requires user intervention). Systems such as Matrix [11, 12] and InterPlanetary File System [13] employ hash-linked DAGs (Merkle-DAGs), which remain available under network partitions while maintaining a partially ordered event history. Keeping the whole causal history in the DAG allows newcomer parties to synchronize from scratch out-of-the-box. However, a major challenge of Merkle-DAGs is their unbounded growth, since each event extends the causal history.

Approach

In this thesis, we introduce Replicated Authenticated Data Types (RADTs) that extend the guarantees of traditional authenticated data types to distributed environments with both multiple clients and multiple servers. Our construction generalizes the multi-client authenticated data type model of Cachin et al. [3] by supporting replication across servers. Rather than targeting strong consistency, the model prioritizes scalability and availability under network partitioning, with servers maintaining a DAG of causal event histories.

To realize this approach, we define the Authenticated Merkle Directed Acyclic Graph (Auth-MDAG), which replaces digital signatures with Message Authenticated Code (MAC) applied over DAG nodes. This design provides client-side integrity and authentication while ensuring that only clients are able to issue updates. Compared to digital signatures, the use of MACs yields a more efficient and lightweight mechanism for securing replicated data.

Building on this foundation, we further integrate CRDT semantics to define Authenticated Conflict-free Replicated Data types (Auth-CRDTs), and we provide several practical instantiations of these constructions. Finally, we evaluate the implementations experimentally, analysing their relative performance, overheads, and trade-offs. These results demonstrate the feasibility of our approach and highlight both its strengths and current limitations.

Our theoretical framework is general: clients are lightweight, and application-specific logic resides solely in event semantics rather than server state. However, servers must relay the entire DAG, and clients must replay it to reconstruct the latest application state. Inspired by *local-first* applications [14], we propose a stateful realization of our framework where clients store a local copy of their application state. The result is perhaps the closest attempt to realizing the local-first manifesto: clients can operate fully offline, while servers enable fast incremental querying and maintain long-term, scalable histories. To validate our approach, we implement a prototype comprising two main components:

- **Client-side:** We employ Automerge [15], a CRDT designed for JSON document stores, which offers native support for complex, nested data structures and provides a mature, widely-used implementation that facilitates practical integration into real systems. An additional advantage of using a CRDT is that it guarantees SEC of the client state with respect to the DAG.
- **Server-side:** We leverage Matrix as a widely adopted transport and persistence layer [12]. We encode Auth-MDAG on top of the Matrix Event Graph (MEG) and model CRDT update payloads as Matrix events, ensuring scalable and efficient replication.

Summary of contributions

The contributions of this thesis are as follows. First, we introduce a general definition of RADT, extending the model of ADT to distributed environments with multiple clients and multiple servers. Second, we define the Auth-MDAG, a novel construction that underpins RADT by combining causal histories with MACs to ensure authenticity and integrity of client operations. Third, we introduce Auth-CRDTs by integrating CRDT semantics on top of Auth-MDAG, and we design and implement several practical instantiations, each exploring different trade-offs and culminating in a prototype that leverages MEG

as a transport layer. Finally, we conduct a systematic experimental evaluation of all implementations, including both micro and macro benchmarks, to measure computational and communication costs as well as scalability.

Thesis structure

The remainder of this thesis is structured as follows. Chapter 2 introduces the fundamental concepts and definitions required to understand the subsequent chapters. Chapter 3 reviews the state of the art, presenting recent work that is directly relevant to and forms the foundation for our study. Chapter 4 , we present our approach by introducing an authenticated replicated data type, which constitutes the core contribution of this thesis. Chapter 5 focuses on practical aspects, describing the implementation of our proposed approach. Chapter 6 evaluates the performance and correctness of the implementation through a series of experiments. Finally, Chapter 7 concludes the thesis by summarizing the findings, discussing limitations, and outlining possible directions for future work.

2. Preliminaries

This chapter introduces the basic concepts that will be used throughout the thesis. We provide formal definitions of fundamental structures and operations, as well as illustrative examples, to ensure the presentation is self-contained and accessible.

In particular, we begin by defining cryptographic authentication techniques such as hash functions and Message Authenticated Codes (MACs). We then turn to systems for data replication, focusing on Directed Acyclic Graph (DAG). Finally, we present techniques that merge both concepts, including Authenticated Data Type (ADT) and Merkle trees.

2.1. Cryptographic Authentication Techniques

We now turn to cryptographic authentication techniques, which are essential for ensuring the integrity and authenticity of data in modern systems [16].

In this section we define their fundamental concepts. We begin with hash functions and then we examine their cryptographic versions, which add security properties such as pre-image resistance and collision resistance. Building on these primitives, we finish the section with Message Authenticated Code, which are often constructed from cryptographic hashes and provide strong guarantees of both data integrity and authenticity. Together, these techniques form the foundation of secure communication and many cryptographic protocols.

2.1.1 Hash Functions and Cryptographic Hash Functions

In computer science and cryptography, Hash function (HF) [17] serve as a fundamental building block. They allow arbitrary-length data to be mapped into fixed-size digests, which are easier to store, compare, or use as identifiers. To reason about their properties and applications formally, we first introduce a general definition of hash functions (in Definition 2.1), before specializing to cryptographic hash function.

Definition 2.1 (Hash function (HF)). A hash function f is a function that takes as input data of arbitrary length and outputs a fixed-size string of characters.

To be a hash function f must guarantee:

- determinism: for the same input all outputs of f should be the same,

- fixed-length output: for any input x , the length of $f(x)$ should be consistent,
- efficiency: evaluation of $f(x)$ must be computable in time polynomial in $|x|$, ideally linear, so that the function is practical for real-world use.

While general hash functions are useful as compact representations of data, many applications in security, such as digital signatures, authentication, and integrity verification, require stronger guarantees. For this reason, we distinguish Cryptographic Hash function (CHF) [18], which are hash functions augmented with specific hardness properties that make them resistant to adversarial attacks. We formalize this notion in Definition 2.2.

Definition 2.2 (Cryptographic Hash function (CHF)). A cryptographic hash function h , is a hash function, and therefore, must satisfy the properties of Definition 2.1.

Furthermore an ideal CHF must also satisfy the following properties:

- Uniformity: an ideal n -bit hashing function should output each given digest with probability 2^{-n} ;
- Pre-image resistance: given an hash y it is computationally infeasible to find an input string x such that $h(x) = y$;
- Second pre-image-resistance: given an input x and its hash $h(x) = y$, finding a second input x' with hash $h(x') = y$ is computationally infeasible.
- Collision resistance: it should be computationally infeasible to find two distinct inputs x and x' with the same hash, $h(x) = h(x')$.

The properties defined in Definition 2.2 specify the characteristics of an ideal CHF. In practice, however, CHFs do not always guarantee these properties. MD5 [19] is a CHF that was widely used but has since been proven to be insecure through various attacks [20–22].

In contrast to insecure designs like that of MD5, modern constructions such as the Merkle–Damgård paradigm [23] and the sponge construction [24], are capable of achieving the desired properties.

The Merkle–Damgård paradigm [23] builds a variable-length hash function from a fixed-length compression function. The input message is first padded to a multiple of the block size, then processed sequentially block by block using a compression function, starting

from an initial value (IV). The final output of the last block serves as the hash. This construction preserves collision resistance if the compression function is secure, but is susceptible to length-extension attacks without additional measures. This construction is used in the implementations of MD5 [19], SHA-1[25], and SHA-2 [26].

In contrast, the sponge construction, used in SHA-3/Keccak [24, 27], operates on a large internal state split into a rate and a capacity. During the absorbing phase, input blocks are XORed into the rate portion of the state followed by a permutation. In the squeezing phase, output is extracted from the rate portion, with the permutation applied as needed until the desired hash length is reached. This structure is flexible and provides strong resistance to collisions and pre-image attacks. SHA-3 is currently the NIST-recommended standard for cryptographic hash functions, specified in FIPS 202 [27].

The CHFs have a wide range of applications, including password verification, signature generation and verification, and authentication and integrity verification of messages and files. Many of these applications, such as message authentication, rely not only on hash functions but also on cryptographic constructs known as Message Authenticated Codes.

2.1.2 Message Authenticated Code

Message authentication is an essential requirement in modern communication systems, as it ensures both integrity and authenticity of the exchanged messages.

Message Authenticated Codes (MACs) [16, 18] are fundamental structures for achieving authentication. They allows two parties to verify the authenticity and integrity of messages exchanged between them.

We formally define them in Definition 2.3.

Definition 2.3 (Message Authenticated Code (MAC)). Formally, a MAC scheme consists of three algorithms:

- A key generation algorithm, $\mathcal{G}$, that takes as input 1^λ where λ is the security parameter and generates key k ,
- A signing algorithm, $\mathcal{S}$, that takes as input the key k a binary string x , computing the tag t ,
- A verifying algorithm, $\mathcal{V}$, that given k , x and t outputs *accepted* if and only if t was generated by computing the tag for message x with key k , and *rejected* otherwise.

For correctness, $\mathcal{V}(k, x, \mathcal{S}(k, x)) = \text{accepted}$ must hold for all valid keys k and messages x .

A MAC scheme must also ensure unforgeability. This means that for a key k and an adversary that is given one or more pairs of binary string and their tag, $(x_i, t_i = \mathcal{S}(k, x_i))$, it is computationally infeasible for the adversary to compute any new pair (x, t) for any new input $x \neq x_i$, including $\mathcal{S}(k, x) = \mathcal{S}(k, x_i)$ for some i .

In other words, the signing algorithm computes a tag from the message and secret key, while the verification algorithm checks that the tag matches the sender's computation.

There are various implementations of MAC systems. Hash-Based MAC (HMAC) [28] are based on cryptographic hash functions. Cipher-based MAC (CMAC) [29] are based on block ciphers such as AES in CBC mode. Universal MAC (UMAC) [30] are based on fast polynomial hashing with a one-time key.

In this dissertation, whenever a MAC is utilized, it is instantiated as an HMAC using the CHF SHA-3 with a 256-bit output [24, 27].

While MACs ensure the integrity and authenticity of individual messages, modern distributed systems often require data to be stored and replicated across multiple nodes.

2.2. Replication and Distributed Systems

In this section we detail techniques commonly used for data replication across systems with multiple participants. First we define different message ordering requirements that applications may need, then we introduce Directed Acyclic Graphs which are structures that allow enable efficient and verifiable replication.

2.2.1 Message Ordering

In distributed systems, events occur independently on different processes. To reason about consistency, coordination, or synchronization, we need a mechanism to impose order on these events, even in the absence of a global clock.

There are different ways to order events, in ascending order of strength unordered, First-In-First-Out (FIFO), causal ordering, and total ordering.

The weakest ordering is no ordering (or best-effort ordering). In this scenario events arrive in any order and provide no guarantees. This is typically used in protocols like User Datagram Protocol (UDP) [31], where ordering is not critical.

Stronger than unordered delivery FIFO ordering [32] ensures that events from the same sender are delivered in the order they were initiated. For example, if a sender generates events e_1 and e_2 sequentially, all receivers will see e_1 before e_2 . FIFO ordering can be guaranteed by using the Transmission Control Protocol (TCP) [31, 33] protocol.

Building on FIFO ordering, causal ordering is based on the happens-before relationship [32, 34]. Formally, The happens before relationship is defined as:

Definition 2.4 (Happens-before relationship). For events a and b , a happens-before b ($a \rightarrow b$) if:

- when a and b occur in the same process and a occurs first,
- when a is the sending of a message (a type of event), b is the receipt of that message,
- or when there is an event c such that $a \rightarrow c$ and $c \rightarrow b$.

For instance, In a collaborative document system with two users u_1 and u_2 . u_1 inserts a word into the document, this is the first event, e_1 . Concurrently, u_2 deletes a different word constituting the second event, e_2 . Finally, u_1 deletes the word inserted in e_1 , completing the third event e_3 . Here, $e_1 \rightarrow e_3$, while e_2 is concurrent with e_1 and e_3 . Causal ordering ensures e_3 is applied after e_1 , but e_2 can be applied in any order relative to e_1 and e_3 .

The strongest form of ordering is total ordering [32], where all processes see events in the exact same sequence. This is essential for consensus protocols, such as Paxos [35] or Raft [36], where replicas must agree on a single sequence of events.

Each ordering method provides different guarantees and trade-offs, which makes them suitable for different scenarios. The choice depends on the required consistency, coordination, and performance of the system.

2.2.2 Directed Acyclic Graph

Directed Acyclic Graph (DAG), is a type of graph structure where its nodes are connected with directed edges (edges that have a specific orientation) and with no cycles.

Formally, DAG is defined as:

Definition 2.5 (Directed Acyclic Graph (DAG)). A DAG is a directed graph $G = (V, E)$ where no cycles exist, *i.e.* there exists no sequence of edges of the form $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_1$.

We denote nodes whose in-degree is 0 and nodes whose out-degree is 0 as head and tail nodes, respectively.

An example of a DAG can be seen in Figure 2.1. In it, there are 5 nodes labelled n_1 through n_5 . n_1 is the only tail node of the DAG, and, n_4 and n_5 (the highlighted nodes) are both heads.

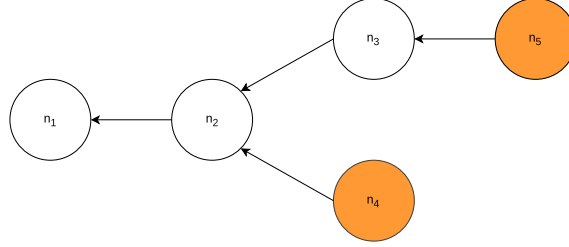


FIGURE 2.1: An example of a DAG where the heads are highlighted.

Because of its structure, DAGs are widely used for representing dependencies, precedence constraints, and hierarchical relationships [37–39]. Specifically, in distributed computation, they can be used to provide an explicit record of the operations performed on a system. This record can then be used for reproducibility and fault tolerance [40, 41].

For instance, the happens-before relationship (Definition 2.4) can be modelled by a DAG.

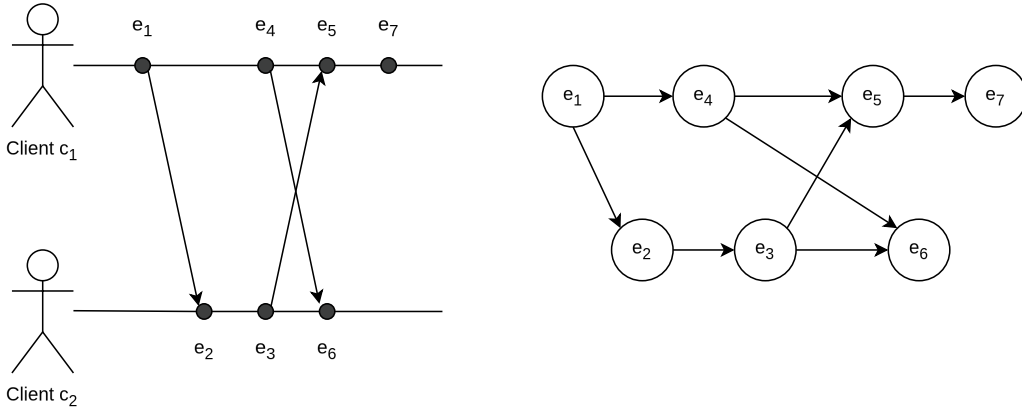


FIGURE 2.2: A system whose events are ordered through causal ordering, and the DAG that results from the ordering

Figure 2.2 shows a system with two clients, c_1 and c_2 , and the events that happen between the two. The causal ordering obtained by applying the definition can be visualized through the use of a DAG. With the resulting DAG we can more easily reason about the system, including, the possible order of events, and the state of the clients.

For instance, assuming c_1 has knowledge of all events, there are several different orders in which it can process events, such as, $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $\{e_1, e_2, e_4, e_3, e_6, e_5, e_7\}$. These

different orders are known as topological sorts of the DAG [42] or linearisations of the DAG [3].

To reason about these orders rigorously, we formally define a linearisation of a DAG as:

Definition 2.6 (Linearisation, or topological sort, of a DAG). Given a DAG $G = (V, E)$, a sequence $s = (v_1, v_2, \dots, v_n)$ of all vertices $v_i \in V$ is a linearisation, or topological sort, of G iff:

$$\forall (u, v) \in E, u \text{ appears before } v \text{ in } s.$$

Note that a DAG may have multiple valid linearisations.

Another instance where DAGs are used as explicit record of operations is Apache Spark [41], a distributed computing framework designed to efficiently handle a wide range of big data workloads. It builds a DAG of operations, known as the lineage graph, for every Resilient Distributed Dataset (RDD) (a read-only, partitioned collection of data that can be processed in parallel across a cluster). This lineage records all the transformations (like map, filter, or join) that were applied to the original data.

If a worker node fails and a partition of an RDD is lost, Spark uses its DAG to recompute only the missing partitions. It can rebuild the lost data by following the steps outlined in the DAG, often from data still in memory on other nodes or from the original input data. This process ensures that the results are reproducible and the system is highly resilient to failures.

However, DAGs by themselves do not provide any security properties for distributed systems. To address this we augment them and other relevant data structures with cryptographic authentication techniques introduced in Section 2.1.

2.3. Authenticated Data Type

Tamassia [2], defined Authenticated Data Types (ADTs) with the intent to secure data structures replicated on untrusted devices. This is achieved by combining cryptographic authentication techniques (introduced in Section 2.1) and techniques for replication and distributed systems (introduced in Section 2.2).

A common case where these issues arise is with the use of a cache server. In this scenario the cache server stores a copy of the server's data, and the user queries data from the cache server instead of the server. In case the cache server becomes compromised, responses could be modified or forged.

To solve this issue Tamassia proposes the following model. Given a structured collection $\mathcal{S}$, and three parties, the *source*, the *responder* and the *user*, where:

- *source* holds the original version of $\mathcal{S}$;
- *responder*, maintains a replica of $\mathcal{S}$, it receives the changes performed by *source* and the associated authentication information;
- finally, *user* queries $\mathcal{S}$, but instead of contacting *source* they contact *responder*.

In this model, upon making changes, *source* produces structure authentication information t , *i.e.*, a signed and time-stamped statement proving that *source* changed $\mathcal{S}$. *responder* receives the changes and associated t .

Responder in addition to returning the answer to *user* queries, will also return answer authentication information, which consists of the latest t and a proof p of the answer. The *user* can then use p to verify the correctness and completion of the received answer.

Thus, even if the responder is compromised, the user can detect any modification or forgery through p , and therefore only needs to trust the source. Furthermore, because t is timestamped, *user* can also detect when it is being served outdated data.

Having defined the ADT model, in the following sections we introduce Merkle trees and Merkle Directed Acyclic Graphs (Merkle-DAGs), two are structures often used as ADTs, that provide the necessary properties.

2.3.1 Merkle Tree

Merkle trees are a fundamental data structure used to efficiently verify the integrity of large volumes of data.

First introduced by Merkle, in [7], a Merkle Tree is a tree in which the leaf nodes are labelled with cryptographic hashes of a data block. Then, each internal node, is labelled with the cryptographic hash of the labels of its children.

This culminates in the root hash which stands at the top of the tree and is, effectively, a signature representing all the data blocks.

One notable advantage is its efficient verification of a data block in logarithmic time. To verify a specific data block the receiver needs only to verify the path from the root of

the tree to the data block [7, 43]. This advantage extends to changes, if a data block is altered, only a logarithmic number of hashes need to be recomputed.

Furthermore, each block can be verified independently, enabling parallel processing across multiple threads [44]. This property makes Merkle trees particularly well suited for distributed environments with large datasets.

In Figure 2.3 we can see how a Merkle tree is calculated. In this example we have 4 data blocks, labelled d_1 through d_4 . From these data blocks, hashes labelled h_1 through h_7 are calculated. The final hash h_7 is the root hash.

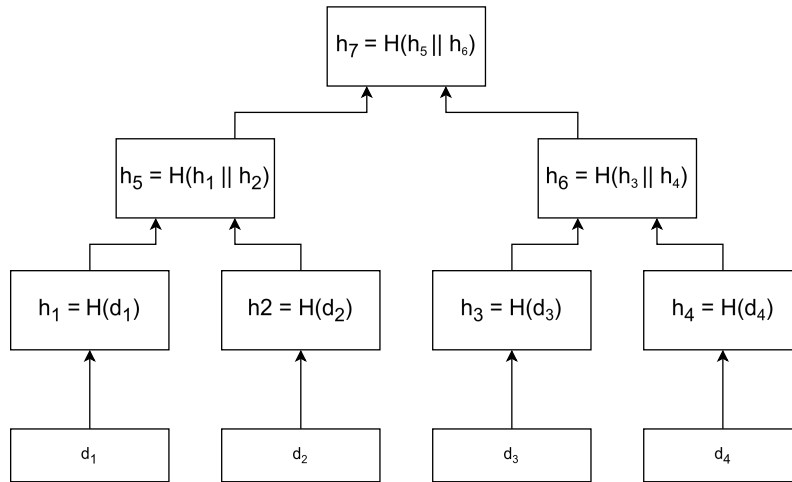


FIGURE 2.3: Example of how the root hash of a Merkle tree is calculated.

Consider the example in Figure 2.3. If a peer gets the root hash h_7 from a trusted source, it can then request the rest of the tree from any untrusted party. Suppose d_3 becomes corrupted and becomes d'_3 , when recalculating the root hash from the received nodes, h_3 changes, subsequently h_6 and h_7 are also changed. When the peer compares the recalculated hash to h_7 , the mismatch immediately reveals the tampering. The peer can then discard the incorrect data and request it from another source.

This approach allows for data stored or transferred between computers to be efficiently verified, ensuring that the blocks are undamaged and unaltered.

These properties explain why Merkle trees are widely used in peer-to-peer networks, having applications in:

- Bitcoin [8], where the Merkle tree root hash is added to each block to validate the integrity of the transactions of said block;

- the version 2 of the bit torrent specification [45], where the Merkle tree root hash is calculated from torrent data, such that the torrent file only needs to store the root hash;
- and, distributed databases like Apache Cassandra [46] and Amazon DynamoDB [47], where, by comparing the Merkle roots of their data, the nodes of the network can quickly identify and reconcile inconsistencies without having to transfer the entire dataset.

While Merkle trees provide an efficient way to authenticate general data, many applications require structures that can capture arbitrary dependencies. This motivates the use of Merkle Directed Acyclic Graphs, which extend Merkle trees to DAGs.

2.3.2 Merkle Directed Acyclic Graph

By combining Merkle Tree's cryptographic hashing with DAG's ability to branch and merge, we get a content-addressable graph where every node is uniquely identified by its data and links, this structure is referred to as Merkle Directed Acyclic Graph (Merkle-DAG) [6].

Formally Merkle-DAGs are defined as:

Definition 2.7 (Merkle Directed Acyclic Graph (Merkle-DAG)). A Merkle-DAG is a DAG $G = (V, E)$ where each vertex $v \in V$ is assigned a label $\ell(v)$ computed as follows:

- If v is a leaf node, $\ell(v) = H(d_v)$, where d_v is the data stored at v and H is a cryptographic hash function.
- If v is an internal node with children $C(v) = \{u_1, u_2, \dots, u_m\}$, then

$$\ell(v) = H(d_v || \ell(u_1) || \ell(u_2) || \dots || \ell(u_m))$$

where $||$ denotes concatenation.

For the labels to be well-defined, it is essential that ℓ enforces a consistent concatenation order of the children. Otherwise, a single node could produce different labels depending on the order in which its children are combined.

The heads of the DAG are identified by its hash $\ell(r)$, which uniquely represents the entire structure and its data.

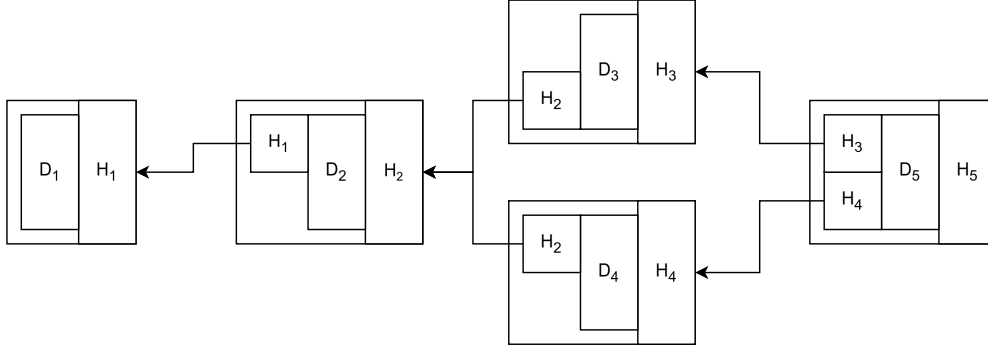


FIGURE 2.4: Example of a Merkle-DAG with five nodes, showing data D_i , hashes H_i , and parent-child dependencies.

Figure 2.4 shows a Merkle-DAG with five nodes, each storing data D_i and its hash H_i . Node 1 has no parent, so $H_1 = h(D_1)$. Node 2 depends on node 1, with $H_2 = h(D_2 \| H_1)$. Nodes 3 and 4 both reference node 2, giving $H_3 = h(D_3 \| H_2)$ and $H_4 = h(D_4 \| H_2)$. Finally, node 5 has two parents, nodes 3 and 4, so $H_5 = h(D_5 \| H_3 \| H_4)$.

Because of their design, Merkle-DAGs inherit the authenticity guarantees while also providing the flexibility and reusability of DAGs.

They are widely used in decentralized and distributed systems. For example, the Interplanetary File System (IPFS) [13] employs Merkle-DAGs to ensure content-addressable storage, enabling efficient de-duplication, verifiable retrieval, and robust sharing of large datasets. Another tool that uses Merkle-DAGs is Git [10]. Git uses Merkle-DAGs for controlling the versions of the project, enabling efficient branching, merging, and history traversal.

3. State Of the Art

Having established the fundamental concepts and structures in the preceding chapter, this chapter reviews existing work and research that form the foundation of this thesis.

We begin with Conflict-free Replicated Data Type (CRDT), describing how they ensure eventual consistency and detailing the different types of implementations. CRDTs provide a foundation for providing high availability and network partition tolerance in distributed systems.

Next, we introduce Proof-carrying Data, a technique for ensuring integrity and correctness of outsourced computations. While not directly applicable to our setting, its concept of recursive proofs is highly relevant, as it shows how correctness guarantees can be securely composed and verified across multiple steps.

Finally, we review a verifiable computation system for multiple clients, which demonstrates how integrity can be preserved even when clients act independently and the server may be untrusted.

3.1. Conflict-free Replicated Data Types

In real-world networks, it is normal to have data be replicated across various computers, as this practice has several different benefits. Namely, it improves performance as clients can access the machine that is geographically closest to them [32, 48]. Replication also ensures fault tolerance, as, if a node fails, other replicas can continue serving requests without data loss [48].

However, having data replicated across machines doesn't come without its challenges. A key challenge is synchronization: a client should be able to interact with any replica of the system without being aware of which replica is responding.

This is challenging because without proper coordination between each replica, it can lead to inconsistent states between them. Furthermore, in some cases, this inconsistency might not be resolvable.

For instance, consider two clients, client c_1 and c_2 , that interact with replicas r_1 and r_2 , respectively, by incrementing and decrementing a counter. If, at the same time, c_1 increments the counter and c_2 decrements the counter, r_1 and r_2 now have inconsistent

states, as the two replicas show different values for what should be the same counter. In this example, if replicas only keep the latest value, rather than tracking increments/decrements individually, merging is impossible as the original operations are lost, and no deterministic resolution exists.

The CAP theorem [49] states that a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance. In practice, network failures and message delays are unavoidable in distributed environments, meaning that systems must be able to continue operating correctly even when some nodes cannot communicate with others. This requirement—known as partition tolerance—forces designers to choose between consistency and availability.

Traditional replication strategies, such as primary-backup and synchronous replication, [32] typically favour consistency, requiring coordination across replicas to ensure a single, globally consistent view of the data, but this comes at the cost of reduced availability during network partitions.

Replicated Data Type (RDT) [50] are data types designed to address this trade-off by giving up on strong consistency in favour of availability. Intuitively, an application that makes use of an RDT can update any replica of the data type concurrently, without coordinating with other replicas. Instead of enforcing immediate agreement, RDTs ensure eventual consistency by guaranteeing that all replicas will converge to the same state once updates have been propagated throughout the system.

However, while RDTs guarantee convergence under replication, the responsibility of handling conflicts is typically left to the application developer. This often results in ad hoc or case-specific resolution strategies, which can make reasoning about correctness and consistency difficult, especially in large-scale systems.

Conflict-free Replicated Data Types (CRDTs) are special instantiations of RDT that address this issue. Introduced by Shapiro et al., CRDTs [5] implement an algorithms that ensures that conflicts arising from the natural parallel use of replicas are automatically resolved. By removing the need for custom ad hoc logic, CRDTs offer a principled foundation for building reliable and scalable distributed systems.

CRDTs operate in environments where clients interact with replicas by invoking operations. A query operation reads the current state of a replica without producing side effects. An update operation mutates the local state of the source replica. How this

change is propagated to other replicas depends on the CRDT type: in some cases the full state is periodically disseminated, while in others the update is transmitted directly as an operation or as a smaller delta.

CRDTs were initially motivated by collaborative text editing and mobile computing [51], where users may edit data concurrently while offline. Since then, they have found applications in distributed chat systems, like Matrix [12] and in large-scale distributed databases, such as Redis, who integrates CRDTs to enable geo-distributed replication [52, 53], and Cosmos DB, who adopts CRDT-inspired policies to manage cross-region conflicts [54, 55]. Furthermore, frameworks such as Automerge [15] and Yjs [56] enable integration of CRDTs in more complex applications.

In the following sections we present the main types of CRDTs: State-based Convergent Replicated Data Type, Operation-based Commutative Replicated Data Type and Delta-Mutation Convergent Replicated Data Type. Their main difference lies in how they generate and propagate messages to maintain consistency across replicas. Some rely on transmitting full state, while others disseminate only operations or smaller state deltas. We will characterize each type, highlighting their advantages and limitations, and indicating typical use cases.

3.1.1 State-based Convergent Replicated Data Type

State-based Convergent Replicated Data Types (state-based CRDTs) [5], are CRDTs where each replica periodically transmits its local state to others, and upon receiving a state, a replica merges it with its own, guaranteeing that all replicas eventually reach the same result. This approach ensures convergence across replicas by relying on a well-defined merge function that combines local states.

State-based CRDTs [5] ensure convergence by defining each replica's state as an element of a mathematical structure known as a join semilattice. This structure guarantees that a merge function exists with the key properties of commutativity, associativity, and idempotence, which collectively ensure correct and deterministic merging of states across replicas.

The correctness of the merge function relies on these properties in distinct ways: idempotence prevents duplicate updates from affecting the result, commutativity ensures convergence regardless of the order in which states are received, and associativity allows updates

to be merged transitively across multiple replicas. Together, these properties make it possible to propagate state efficiently, even over unreliable networks, using patterns such as gossip protocols.

An example of a state-based CRDT is a Grow-only Counter (G-Counter) [57]. This example represents a monotonically increasing counter, its specification is defined in Figure 3.1.

<pre> increment(st, i): st[i] ← st[i] + 1 Return st merge(st, st'): For j ∈ N : st[j] ← max(st[j], st'[j]) Return st </pre>	<pre> value(st): sum ← 0 For j ∈ N : sum ← sum + st[j] Return st, sum </pre>
--	--

FIGURE 3.1: Grow-only Counter (G-Counter) Specification

The state st is an array of integers, where each replica maintains a unique entry indexed by its own identifier. To increment the counter, replicas use the `increment` operation, which increases the value of their own entry. Then, clients can obtain the value of the counter through the `value` operation, which returns the sum of all entries in the map. Finally, the CRDT merge function, and by extension the `merge` operation, consists of taking the maximum of each element of the array. Since only increments are allowed, this ensures that all updates are accounted for and that replicas eventually converge.

While there are many advantages to state-based CRDT, because the full state needs to be sent to other replicas to synchronize. Messages can be big, which places a huge strain in the network.

3.1.2 Operation-based Commutative Replicated Data Type

Operation-based Commutative Replicated Data Types (op-based CRDTs) [5] are another type of CRDT. In contrast to state-based CRDTs where the replicas send each other the full state, op-based CRDTs replicas propagate each operation that is applied on the CRDT.

Since operations are only the incremental changes of the state, they are, typically, smaller than the full state. Because of this, op-based CRDTs avoids the high bandwidth cost of sending the full state inherent to state-based CRDTs.

To support this behaviour, op-based CRDTs conceptually separate update operations into two phases: the “At Source” phase and the downstream phase. The At Source phase is triggered by a client request, during which the source replica applies the update to its local state and generates a message describing the operation. In the downstream phase, this message is sent to other replicas, which incorporate the operation into their own states. This two-phase approach ensures that updates are applied consistently across all replicas while minimizing the amount of data transmitted.

For instance an Observed-Remove Set (OR-Set) [57], is a op-based CRDT that represents a set, its specification is in Figure 3.2. To add elements to the set, through the `add` function, the source replica tags the added element with a unique identifier, the pair is then propagated, in the downstream phase, to other replicas. To remove an element from the set, through the `remove` function, the source replica first calculates the subset of elements that are to be removed, and propagates this subset to all other replicas. Even if additions and removals occur concurrently at different replicas, the identifiers ensure that the final merged set is consistent across all replicas once all operations are delivered.

<u>add(e):</u> At Source <code>add(st, e)</code> : $\alpha \leftarrow \text{unique}()$ $\text{st} \leftarrow \text{st} \cup \{(e, \alpha)\}$ Return <code>st, α</code>	<u>remove(e):</u> At Source <code>remove(st, e)</code> : $R \leftarrow \{(e, u) \mid \exists u : (e, u) \in \text{st}\}$ $\text{st} \leftarrow \text{st} \setminus R$ Return <code>st, R</code>
Downstream <code>add(st, (e, α))</code> : $\text{st} \leftarrow \text{st} \cup \{(e, \alpha)\}$ Return <code>st</code>	Downstream <code>remove(st, R)</code> $\text{st} \leftarrow \text{st} \setminus R$ Return <code>st</code>
<u>lookup(e):</u> Return <code>st, $\exists u : (e, u) \in \text{st}$</code>	

FIGURE 3.2: Op-based CRDT Observed-Remove Set (OR-Set) Specification

For op-based CRDTs to be successful, they require their operations to arrive exactly once and in causal order (see Definition 2.4). And, while, these requirements can be satisfied by relying on an reliable causally ordered broadcast communication protocol [5], this guarantee is hard to maintain as the number of participants increases. Notably, to prevent duplication, large logs must be maintained, even if TCP is used [58].

3.1.3 Delta-Mutation Convergent Replicated Data Type

Delta-Mutation Convergent Replicated Data Type (delta-based CRDT) [59, 60] seek to achieve a middle ground between state-based CRDT and op-based CRDT *i.e.*, they seek

the small message size of op-based CRDTs and the robustness of state-based CRDTs.

This is done by introducing a *delta-mutator* function. This function returns an incremental state, or delta, which is then propagated to other replicas. The key idea here is that deltas are themselves part of the same join-semilattice as the full state. The difference is that deltas only represent recent updates on the data. This means that we can still use the merge function defined for state-based CRDT.

An implementation of a delta-based CRDT can be seen in [59], similar to the example described in op-based CRDTs. Figure 3.3, shows the formal specification. This model's state consists of a set of tagged elements and another set of tags that act as tombstones. Tags, of the form *replica-id* $\times$ *integer*, consist of the replica's unique id as well as a monotonically increasing counter, and ensure they are unique for each element. To add an element e to the set, clients use the update operation *add* on replica i , which creates a globally unique tag by incrementing the highest tag present that was created by i . *remove* removes an element from the state at replica i by adding its tags to the tombstones set. Both *add* and *remove* apply their changes directly to the CRDT state and return objects, or deltas, that, as mentioned before, are part of the same join-semilattice as the full state. Trivially, the query operation *elements* returns all elements whose tags are not in the tombstones set. Finally, the merge function of this CRDT is simply the union of the respective sets that are both grow-only. In this example, instead of messages including the full state, they simply include all elements and tombstones added since the last message was generated.

$\frac{\text{add}((s, t), i, e):}{n \leftarrow \max(\{k \mid (i, k, _) \in s\})}$ $s \leftarrow s \cup \{(i, n + 1, e)\}$ $\text{return } (\{(i, n + 1, e)\}, \{\})$	$\frac{\text{remove}((s, t), i, e):}{R \leftarrow \{(j, n) \mid (j, n, e) \in s\}}$ $t \leftarrow t \cup R$ $\text{return } (\{\}, R)$
$\frac{\text{elements}((s, t), i):}{el \leftarrow \{e \mid (j, n, e) \in s \wedge (j, n) \notin t\}}$ $\text{return } el$	$\frac{\text{merge}((s, t), i, (s', t')):}{s \leftarrow s \cup s'}$ $t \leftarrow t \cup t'$ $\text{return } (s, t)$

FIGURE 3.3: OR-Set delta-based CRDT example.

Even so, the message size of delta-based CRDTs is still greater than that of op-based CRDTs, furthermore the size of the messages in most implementations increases based on the number of generated messages. However, recent studies in delta-based CRDT [61, 62] have developed a space efficient delta-based CRDT whose message size increases with the number of replicas of the system, instead of the number of messages generated.

Ditto DB [63] is an example of a database that uses delta-based CRDT. Ditto DB uses a nested CRDT map, along with version vectors to replicate diffs between peers in order for their state to converge.

Having established the distinctions between the different types of CRDTs, it is also important to consider the contexts in which each approach is most effective. Op-based CRDTs work best in systems where reliable message delivery can be ensured, as they transmit only the individual operations performed, keeping communication efficient. State-based CRDTs are better suited to unreliable networks, since they periodically exchange entire states, guaranteeing convergence even when some updates are lost. Delta-based CRDTs offer a middle ground, sending only small state differences, which makes them particularly useful in large-scale systems where bandwidth efficiency and fault tolerance are both essential.

CRDTs achieve Strong Eventual Consistency (SEC) without sacrificing availability, making them highly effective for reliable data replication. However, they are not designed to protect against adversarial settings, where nodes or third parties may act dishonestly.

This motivates the need for solutions that ensure integrity and correctness while supporting the operationalization of processes, such as merge functions. Specifically, we require systems with transitive properties, where computations can be securely composed and delegated. This led us to solutions such as Proof-carrying Data (PCD), which is a system for secure delegation of computations to untrusted entities that relies on recursive proofs of correctness.

3.2. Proof-carrying Data

Some computations require large amounts of computation power, which a client might not have access to. The challenge is to outsource computation to untrustworthy third-parties while still ensuring integrity and correctness of the computation, without having to recompute it locally.

To address this Chiesa and Tromer [64], developed a system for distributing computation among untrusting parties called Proof-carrying Data (PCD).

PCD seeks to reason on the output of a computation step by defining a property of the output data *a priori*. This property is a global invariant that must hold for all outputs of all computations of the system.

To achieve this, the output of a computation includes a proof, which signifies that the output complies with the specified property.

When the output of a computation is used as input of a second computation, the proof is used to verify that the property holds on the inputs, and, as such, the proof of the outputs of the second computation will, not only, attest that the output data of the second computation is valid but also that the proof of the inputs is also valid. Doing this creates a recursive proof that verifies the compliance of the property through the whole history of computation.

Another important result of [64] is that the size of the proof is poly logarithmic in the number of steps of the computation. Because each proof is succinct the verifier's workload remains efficient, making PCD suitable for long-running or distributed tasks.

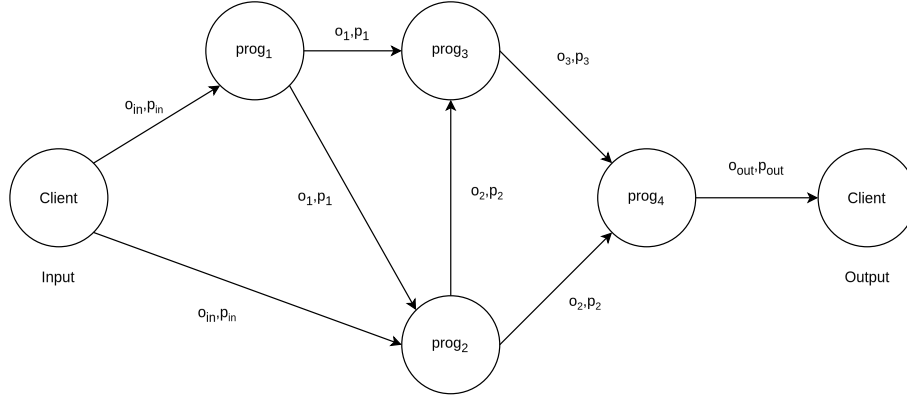


FIGURE 3.4: Example of a PCD system.

Consider the system illustrated in Figure 3.4. It is composed of four programs, $\text{prog}_0, \dots, \text{prog}_3$, each producing an output o_i together with a proof p_i . The proof p_i certifies that the output o_i satisfies the global invariant and that all preceding proofs in the chain were valid. When prog_1 consumes (o_{in}, p_{in}) as input, it verifies the proof p_{in} before producing its own pair (o_1, p_1) . This process continues through prog_2 and prog_3 , so that each step extends the proof chain recursively. Finally, the last program outputs (o_{out}, p_{out}) . The single proof p_{out} is sufficient for a verifier to be convinced of the correctness of the entire computation, from prog_0 to prog_4 , without re-executing any of the intermediate steps.

Beyond this scenario, PCD has a range of applications in distributed and decentralized systems. Prominent examples are blockchain scalability solutions, such as rollups and sidechains, where recursive proofs allow many transactions to be compressed into a single succinct proof [65, 66]. Another key domain is zero-knowledge frameworks such as

Nova [67] and RISC Zero’s zkVM [68]. These realize this PCD paradigm through recursive proofs, enabling verification of long-running or layered computations.

However, while PCD focuses on single-client verifiable computation guaranteeing correctness for operations executed by a single party, many real-world applications, such as collaborative document editing or multi-user databases, require multiple mutually trusting clients to perform operations concurrently on the same object.

3.3. Stateful Multi-client Verifiable Computation

This issue is addressed by Cachin et al. [3] who extends the idea of PCD to multiple clients.

Cachin et al. provides a framework in which mutually trusting clients can outsource the execution of a shared, state-based, computation to an untrusted server, while being able to verify the correctness of the server’s responses. The response to each of the clients operations is accompanied by a proof that ensures that the evolution of the state remains verifiable despite the server’s untrustworthiness.

The protocol is formalized around an ADT F . Each client maintains a local authenticator, a cryptographic commitment summarizing the current state and the history of operations it has observed. When a client performs an operation o , it sends o to the server, which returns the result r together with a proof π that the operation is consistent with the current state. The client verifies π and updates its authenticator, thereby extending a chain that cryptographically commits to the entire operation history.

To coordinate multiple clients, the system employs a *lock-step protocol*: operations are processed in rounds, with the server executing one operation per round and distributing the corresponding result and proof to all clients. Each client verifies the proof and updates its authenticator accordingly.

Cachin et al. further states that it is impossible to guarantee linearisability (Definition 3.1). This is because, a client can send an operation to the server but a malicious server can withhold that operation from the queries of other clients. Moreover, because clients, in this system, do not communicate between themselves, they cannot detect that some operations were omitted. This is known as a fork attack. It produces divergent authenticator chains, creating a fork in the operation history.

Instead Cachin et al. ADTs guarantee fork-linearisability (Definition 3.2). Under this model, the most a malicious server can do is isolate the view of each client, meaning each

client only sees the updates they performed.

Definition 3.1 (Linearisability). A concurrent history H of operations is linearisable with respect to a type F [69] if there exists a sequential permutation $\pi(H)$, such that:

1. The real-time order of operations in H is preserved in $\pi(H)$; and
2. $\pi(H)$ respects the sequential specification of F .

In other words, each operation appears to take effect atomically at some point between its invocation and its response.

Definition 3.2 (Fork-Linearisability). A concurrent history H is fork-linearisable with respect to type F [4, 70] if, for each client C_i , there exists a subsequence of H_i of H consisting only of complete operations and a sequential permutation $\pi_i(H_i)$, such that:

1. All complete operations in H occurring in client C_i are in H_i ,
2. $\pi_i(H_i)$ preserves the real-time order of H_i ,
3. $\pi_i(H_i)$ satisfies sequential specification of F ,
4. All views S_i are mutually consistent: if two views S_i and S_j both contain a common operation o , then S_i and S_j agree on the sequence of all operations up to and including o .

Intuitively, fork-linearisability allows divergence of client views (a "fork"), but once views diverge they cannot re-merge. Fork linearisability is weaker than linearisability because it requires consistency with respect to permutations of sub-sequences of history, as opposed to requiring consistency of the whole history.

4. Replicated Authenticated Data Type

We consider a distributed system with multiple clients and multiple replicas of a server. In this system, clients independently issue updates and queries to the state which is stored by the server, and do not have affinity with any specific replica. To ensure proper consistency, replicas may communicate with each other. Communication is asynchronous and reliable.

This model generalizes prior work on multi-client abstract data types [3], which consider a single server, as well as the distributed replication setting of CRDTs (Section 3.1).

In this chapter, we first provide a formal definition of a Replicated Authenticated Data Type (RADT). Then, we show how a RADT can be implemented on top of an Authenticated Merkle Directed Acyclic Graph (Auth-MDAG), a structure that implements an authenticated Merkle-DAG (Section 2.3.2) with MACs, and how this structure can be further extended by incorporating the semantics of a CRDT, yielding an Authenticated Conflict-free Replicated Data type (Auth-CRDT).

4.1. Replicated Authenticated Data Type

Our system allows for the update/query interaction with an Replicated Authenticated Data Type (RADT) in two steps: updates are first prepared by clients and then processed by the replicas, while queries are first processed by replicas and then decoded by clients.

As such, clients display the following Application Programming Interface (API)

- $\text{CInit}(\lambda, i, N)$ takes security parameter λ , client identifier i and a set of replicas N , producing a client state st_C and optional public and private parameters $\text{p}_{\text{pub}}, \text{p}_{\text{priv}}$
- $\text{CSet}(\text{st}_C, \text{p}_{\text{priv}}, \text{p}_{\text{pub}})$ takes client state st_C and parameters $\text{p}_{\text{priv}}, \text{p}_{\text{pub}}$, and produces updated client state st_C
- $\text{CUpdate}(\text{st}_C, j, op, v, d)$ takes client state st_C , replica identifier j , operation parameter op and value v , and optional dependency parameter d , and produces update data u and updated client state st_C
- $\text{CQuery}(\text{st}_C, j, op, q)$ takes client state st_C , replica identifier j , operation parameter op and an encoded query from the replica q , and produces the query output o

Additionally, replicas are expected to propagate updates to other replicas. The process and frequency can vary widely, depending on the network logic of the system. To encompass all possible behaviours, we propose general operations for state propagation and subsequent merge. Replicas display the following API.

- $\text{RInit}(N, C, \mathbf{p}_{\text{pub}})$ takes a set of replicas N , a set of clients C and public parameters $\mathbf{p}_{\text{pub}}$, and initializes replica state st_R
- $\text{RUpdate}(\text{st}_R, i, u)$ takes replica state st_R , client identifier i and update data u , and produces an updated replica state st_R
- $\text{RQuery}(\text{st}_R, i, op)$ takes replica state st_R , client identifier i and operation parameter op , and produces an encoded query q
- $\text{Propagate}(\text{st}_R, j)$ takes replica state st_R and replica identifier j , and produces an updated replica state st_R and propagation message m
- $\text{Merge}(\text{st}_R, j, m)$ takes replica state st_R , replica identifier j and propagation message m , and produces an updated replica state st_R

A RADT system Π is defined as a set of these nine client and replica algorithms. These algorithms operate within the environment illustrated in Figure 4.1, and it is as follows:

The system is initialized by invoking `init`, which generates initial states for all client and replica. During this phase, each client's public and private parameters are created and subsequently shared with all other clients.

Once initialized, clients interact with replicas through the `update` and `query` operations. The `update` operation produces update data that is transmitted to the corresponding replica, where it is incorporated into the Authenticated Merkle Directed Acyclic Graph (Auth-MDAG). The `query` operation retrieves the current state of the replica by applying all updates from the Auth-MDAG and performing the requested query on the resulting state.

Replica synchronization is achieved through the `merge` operation. In this process, a replica generates a propagate message that is sent to another replica, which integrates it into its local state.

$\text{init}(\lambda, C, N, t):$ $\text{prms} \leftarrow []$ For $i \in C$: $(\text{st}_i, \text{prms}_i) \leftarrow_{\$} \text{CInit}(\lambda, i, N)$ $\text{prms} \leftarrow (\text{prms}_i : \text{prms})$ For $i \in C$: $\text{st}_i \leftarrow \text{CSet}(\text{st}_i, \text{prms})$ For $i \in N$: $\text{st}_i \leftarrow_{\$} \text{RInit}(C, N)$	$\text{update}(i, j, op, v, d):$ $(\text{st}_i, u) \leftarrow \text{CUpdate}(\text{st}_i, j, op, v, d)$ $\text{st}_j \leftarrow \text{RUpdate}(\text{st}_j, j, u)$ Return u $\text{query}(i, j, op):$ $q \leftarrow \text{RQuery}(\text{st}_j, i)$ $o \leftarrow \text{CQuery}(\text{st}_i, j, op, q)$ Return o
$\text{merge}(i, j):$ $(\text{st}_i, m) \leftarrow \text{Propagate}(\text{st}_i, j)$ $\text{st}_j \leftarrow \text{Propagate}(\text{st}_j, i, m)$	

FIGURE 4.1: RADT environment: client-replica interaction(top), replica-replicas interaction(bottom).

4.2. Authenticated Merkle Directed Acyclic Graphs

As a central building block, we introduce an Authenticated Merkle Directed Acyclic Graph (Auth-MDAG), a direct adaptation of the classical Merkle-DAG (Definition 2.7, Section 2.3.2) where each hash is replaced by a cryptographic attestation of authenticity. Functionally, this will provide the same robustness guarantees as a Merkle-DAG, with the additional guarantee that only legitimate clients are able to add nodes to it.

An Auth-MDAG Θ provides the same operations as a traditional DAG structure (Definition 2.5, Section 2.2.2), with three additional operations. $\text{Gen}(\lambda)$ takes security parameter λ and generates secret cryptographic material. $\text{EAuth}(k, h, v)$ produces an authentication tag for data v and its dependencies h , which can then be added to the Auth-MDAG. $\text{Ver}(k, d)$ iteratively verifies the authentications associated with all events in d .

Alongside our definition, we propose a straightforward construction using MACs (Definition 2.3, Section 2.1.2). Let $\Delta = \{\text{Gen}, \text{MAC}, \text{Ver}\}$ denote a secure MAC scheme, k a key produced by Gen , and DAG d . Every event v and dependencies h is accompanied by MAC tag t . If an event has no precedents, the produced tag is simply the result of $\text{MAC}(k, v)$. $\text{EAuth}(k, h, v)$ produces an authentication tag that is the result of $\text{MAC}(k, v \parallel \text{order}(h))$. Here $\text{order}(h)$ is a lexicographical ordering of all events in h . This operation is necessary to produce a valid tag to accompany the event for DAG append operations. For convenience, we also propose $\text{DAGAuth}(k, d)$, which takes a non-authenticated DAG and produces an authenticated DAG, replacing all events (v, h) with tuples of the form $((v, t), h)$, calculating the authentication as previously described. Finally, $\text{Ver}(k, d)$ iteratively verifies

the MACs associated with all events in d . Its behaviour is straightforward: every event e without precedents is checked with the MAC $\text{Ver}(k, e)$, and all subsequent events are checked considering their precedents, in the same lexicographical ordering in which they are generated.

Clients and replicas each maintain an Auth-MDAG representing the set of observed updates. However, only clients possess the shared symmetric key and can therefore generate authenticated nodes, ensuring the integrity and authenticity of all update events. Replicas store and propagate the update operations received from clients, presenting it during query and propagate operations. Merge and update operations are appended to the local DAG. Our construction is illustrated in Figure 4.2.

On the client side, CInit has the first client ($i = 0$) generate the k , which is given to all others in CSet . It also prepares an empty DAG for all replicas, representing the local view of the client for each replica. CUpdate produces valid MACs for all events in d , joining them with the local view of the DAG. Event e and its tag t is then appended to the DAG. The provided inputs and produced authentications are output as update data. CQuery takes an authenticated DAG q , and rejects if either the local view is not a subset of, or equal to said DAG, or if the DAG verification procedure on q fails. Otherwise, it updates its local view to q , and creates an empty state over which it will sequentially apply all events of q . It then outputs the result of the query operation on said state.

On the replica side, RInit initializes its own view as an empty authenticated DAG. RUpdate takes update data, and includes it into the local DAG, merging with the optional DAG parameter d , and appending update event e . RQuery simply produces the local DAG as output. Propagate produces the local view as update message, and Merge calculates the new DAG by joining the received DAG with the replica state.

The intuition for the security of this construction is straightforward. Existential unforgeability of MAC ensures that only valid updates are accepted by the clients. Furthermore, the sequential construction of the authenticated DAG, verified in Ver , ensures causal dependency of said events is maintained, preventing valid updates from being re-ordered. This restricts any adversary behaviour that does not result in $\perp$ to providing DAGs that either match the expected protocol behaviour, or are outdated versions, i.e. subsets of the expected DAG.

The Auth-MDAG thus provides a verifiable structure in which every operation can be

$\text{CInit}(\lambda, i, N):$ $\text{prms} \leftarrow \epsilon; d \leftarrow []$ If $i = 0$: $\text{prms} \leftarrow \text{Gen}(\lambda)$ For $j \in N$: $d[j] \leftarrow \{\}$ return (prms, d, i) $\text{CSet}(\text{st}, \text{prms}):$ $(\cdot, d, i) \leftarrow \text{st}$ return (prms, d, i)	$\text{CUpdate}(\text{st}, j, op, v, d):$ $(\text{prms}, d, i) \leftarrow \text{st}$ $v' \leftarrow (i, op, v)$ $d' \leftarrow \Theta.\text{DAGAuth}(\text{prms}, d)$ $d[j] \leftarrow d[j] \cup d'$ $h \leftarrow \text{heads}(d[j])$ $t \leftarrow \Theta.\text{EAuth}(\text{prms}, h, v')$ $d[j] \leftarrow d[j]; ((v', t), h)$ Return $(d', ((v', t), h))$	$\text{CQuery}(\text{st}, j, op, q):$ $(\text{prms}, d) \leftarrow \text{st}$ If $d[j] \not\subseteq q \vee$ $\Theta.\text{Ver}(\text{prms}, d) = \perp$: Return $\perp$ $d[j] \leftarrow q$ $\text{st}_q \leftarrow \epsilon$ For $e \in q$: $\text{st}_q \leftarrow \text{apply}(e, \text{st}_q)$ Return $\text{query}(\text{st}_q, op)$
$\text{RInit}(N, C):$ Return $\{\}$	$\text{RUpdate}(\text{st}, i, u):$ $(d, e) \leftarrow u$ Return $(\text{st} \cup d); e$ $\text{RQuery}(\text{st}, i, op):$ Return st	$\text{Propagate}(\text{st}, j):$ Return st $\text{Merge}(\text{st}, j, m):$ Return $\text{st} \cup m$

FIGURE 4.2: Auth-MDAG-based algorithms implementing the RADT API: client interface (top) and replica interface (bottom).

traced and authenticated through cryptographic links. It establishes integrity and authenticity guarantees for all updates within a distributed setting, forming the foundation upon which more expressive, convergent data types can be constructed.

4.3. Authenticated Conflict-free Replicated Data type

While the Auth-MDAG guarantees that all operations are verifiable, it does not, by itself, provide convergence in the presence of concurrent updates. Since updates are represented as independent nodes in the authenticated structure, concurrent operations may produce distinct and potentially unmergeable branches in the DAG. As a result, although every update can be verified, the overall state cannot be deterministically derived without an additional mechanism for resolving conflicts.

CRDTs, as defined in Section 3.1, are replicated data structures designed to provide deterministic convergence in distributed systems without requiring coordination. Each replica maintains a local state that can be updated independently, and states are merged using well-defined methods. This ensures that all replicas converge to the same state, regardless of the order or timing of updates. CRDTs therefore allow distributed systems to achieve SEC while supporting concurrent modifications.

To address the limitations of the Auth-MDAG, we integrate a CRDT in the client's state,

forming an Authenticated Conflict-free Replicated Data type (Auth-CRDT). In this approach, **query** and **apply** are treated as state-management procedures over the CRDT. This change shifts the focus from directly manipulating the authenticated structure to managing the client's replica state, allowing the Auth-MDAG to act as a verifiable log of operations. Consequently, consistency is ensured primarily by the CRDT's convergence guarantees, and the authenticated structure maintained by the Auth-MDAG can be simplified, for example, by tracking only updates that have not yet been applied. Additionally, in the context of CRDTs, the state size grows much slower, often times remaining constant, as in the case of G-Counter CRDT, than the number of updates.

We define an Auth-CRDT as a combination of a CRDT with a Auth-MDAG. An Auth-CRDT inherits the properties of both components, providing verifiable operations alongside the convergence guarantees of the underlying CRDT. In this construction, CRDT updates are mapped directly to Auth-MDAG nodes, by setting op to the identifier of the update operation, and including the parameters of the operation in the update. Upon querying, a linearisation of the Auth-MDAG nodes can be applied, in order, to an empty CRDT state to reconstruct a state that is equivalent to that of any other replica sharing the same Auth-MDAG.

Take the G-Counter CRDT is a state-based CRDT shown in Section 3.1, where each client has a counter for every client. It has the operations **increment**, where a client's local counter is incremented, and the **value** operation, used to obtain the value of the counter.

According to our environment Figure 4.1, the operations would be as follows:

Client i , with state st_C , increments the counter on replica j with state st_R , by calling $CUpdate(st_C, j, "inc", i, \epsilon)$, generating update data $u = (((i, "inc", i), t), h)$, where t is the MAC tag and h parents of the new Auth-MDAG node. Then u would be sent to j , where $RUpdate(st_R, j, u)$ is called, appending u to its the Auth-MDAG.

Client i , with state st_C can then obtain the value of the counter from replica j with state st_R , by calling $RQuery(st_R, i, j)$. j answers i with encoded query q , which is used to call $CQuery(st_C, j, "val", q)$. This applies all **increment** updates included in q to an empty G-Counter state, generating the state where the value **query** is performed, returning the value of the counter.

Furthermore, notice how repeated queries from the same replica will lead to encoded queries q_0 and q_1 , where $q_0 \subseteq q_1$, i.e., all updates in q_0 are included in q_1 . As a result,

client can save the CRDT state that results of the application of the updates in q_0 , and subsequently apply the remaining updates $\{u \mid u \in q_0 \wedge u \notin q_1\}$, which results in a state equivalent to applying all updates in q_1 .

This is guaranteed by the design of state-based CRDTs, as they are defined through a merge operation that is commutative, associative, and idempotent. This ensures that all replicas converge to the same state regardless of the order or duplication of received updates. Therefore, if $q_0 \subseteq q_1$, applying the updates in q_1 or in $q_1 \cup u \mid u \in q_0 \wedge u \notin q_1$ produces an equivalent state, and consequently, identical query results.

For op-based CRDTs, our DAG guarantees that the causal dependencies of each update are respected. This ensures that the set $u \mid u \in q_0 \wedge u \notin q_1$ contains only updates that are either causally dependent on or concurrent with those in q_0 . If the updates are dependent, enforcing that they are applied after the updates of q_0 yields an equivalent state, and thus identical query results. If they are concurrent, applying them after q_0 remains valid, as any causal linearisation of concurrent operations in an op-based CRDT leads to an equivalent state. Hence, querying either state produces the same output.

This composition results in a system that ensures integrity against untrusted replicas through the Auth-MDAG, while maintaining availability and resilience to network partitions through the CRDT.

5. Implementation

This chapter focuses on the practical implementation of the model introduced in the previous chapter. These implementations draw inspiration from local-first applications [14], enabling clients to operate fully offline while servers support fast incremental querying.

As such, our clients will have both a CRDT front-end and an Auth-MDAG back-end. Figure 5.1 shows a diagram of the architecture that our implementations follow. The user makes changes to the CRDT and passes the messages it generates, whether operations, state or deltas, to the back-end. The back-end holds the Auth-MDAG and the authenticated key, and, upon receiving the messages from the front-end, it generates updates which are appended to its Auth-MDAG. Finally, the replica entity, who cannot generate new valid updates, sends the authenticated updates to other clients and/or replicas.

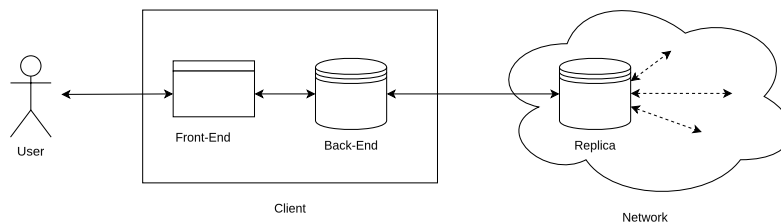


FIGURE 5.1: Architecture of our implementations.

This architecture allows clients to hold a CRDT that, unlike the model presented in Chapter 4 is consistent with the Auth-MDAG, avoiding repeated applications of the same update. This is achieved by applying received updates to the client’s CRDT only when a query is performed.

Recent studies have explored space-efficient CRDTs, whose propagation messages do not grow with the number of updates [62]. However, a practical implementation that supports complex operations is not currently available. Therefore, for our implementation, we use Automerge [15], a CRDT library specifically designed for building local-first software.

In the following sections we first define an API that the implementations follow and then discuss our four practical implementations:

- **baseline** implementation is a direct implementation of a Merkle-DAG, having no guarantees of integrity or authenticity.

- **authenticated** implementation augments **baseline** with MACs, implementing the Auth-MDAG construction, ensuring integrity and authenticity.
- **cached** implementation further extends **authenticated** by implemented optimizations mentioned in the previous chapter.
- **matrix** implementation leverages the Matrix event graph [11] to maintain internal state of clients and replicas.

These implementations are incremental, with each building on its predecessor. As such, we define all methods for **baseline**, while for the subsequent implementations we describe only the modifications and extensions relative to the previous version.

5.1. Implementation Interface

Because of the differences between the formal model and our practical implementations, we define a unified API that all implementations follow.

As previously mentioned, our practical implementations adhere to the local-first philosophy, which requires that clients operate independently and verify updates without relying on servers. To support this, the implementations maintain state for both clients and replicas.

In our implementations, the client functions are adapted from the formal model as follows: **CInit** no longer requires the parameter N (the number of replicas); **CSet** remains unchanged; **CUpdate** modifies the local state directly, eliminating the need for replica-related parameters such as j and d ; and **CQuery** operates on the local state, so parameters q and j are no longer required.

Furthermore, **CQuery** and **RQuery** output a modified state of the client or replica, respectively. This is because, although querying a CRDT or an Auth-MDAG does not change their state, practical implementations may still modify the entity's state, for example, for caching purposes.

To fully support local-first operation, we define two additional client-side functions, **CPropagate** and **CMerge**, which correspond to **Propagate** and **Merge** from the formal model but are executed by clients.

- **CInit**(λ, i) takes security parameter λ and client identifier i , producing a client state st_C and optional public and private parameters $\text{p}_{\text{priv}}, \text{p}_{\text{pub}}$

- $\text{CSet}(\text{st}_C, \text{p}_{\text{priv}}, \text{p}_{\text{pub}})$ takes client state st_C and parameters $\text{p}_{\text{priv}}, \text{p}_{\text{pub}}$, and produces updated client state st_C
- $\text{CUpdate}(\text{st}_C, \text{op}, v)$ takes client state st_C , operation parameter op and value v , and produces update data u and updated client state st_C
- $\text{CQuery}(\text{st}_C, \text{op})$ takes client state st_C and operation parameter op , and produces updated client state st_C and the query output o
- $\text{CPropagate}(\text{st}_C, j)$ takes client state st_C and replica identifier j , and produces an updated client state st_C and propagation message m
- $\text{CMerge}(\text{st}_C, j, m)$ takes client state st_C , replica identifier j and propagation message m , and produces an updated client state st_C

On the replica side, the functions are adapted similarly. RInit no longer requires the parameters N or C , for the same reason that CInit omits N . For consistency, Propagate and Merge are renamed to RPropagate and RMerge , avoiding confusion with the client functions. In addition, RPropagate and RMerge can now take an identifier corresponding to either a replica or a client.

- $\text{RInit}(\text{p}_{\text{pub}})$ takes public parameters p_{pub} , and initializes replica state st_R
- $\text{RUpdate}(\text{st}_R, i, u)$ takes replica state st_R , client identifier i and update data u , and produces an updated replica state st_R
- $\text{RQuery}(\text{st}_R, i, \text{op})$ takes replica state st_R , client identifier i and operation parameter op , and produces an updated replica state st_R and an encoded query q
- $\text{RPropagate}(\text{st}_R, (i, j))$ takes replica state st_R and either a client identifier i or replica identifier j , and produces an updated replica state st_R and propagation message m
- $\text{RMerge}(\text{st}_R, (i, j), m)$ takes replica state st_R , either a client identifier i or replica identifier j , and propagation message m , and produces an updated replica state st_R

Remark 5.1. It is possible to have stateless clients. In that scenario, similarly to our theoretical model, Clients use RQuery to obtain the full DAG, which is then passed to CQuery .

5.2. Base implementation

The **baseline** implementation serves as the foundation for the subsequent implementations, focusing on straightforward choices of data structures and algorithm design. This implementation does not include MACs, and therefore gives no guarantees of authenticity or integrity. Nevertheless, it serves as a reference for our experimental evaluation of Chapter 6.

In the following subsections we showcase the data types used to implement the methods of our API, and describe the design choices made in their implementation.

5.2.1 Cursor

In our system, a client's DAG may contain nodes that the client has not yet processed. This occurs because replicas may freely merge state with a client, whereas the client applies the merged updates to its CRDT only when a query is performed.

The **Cursor** is a data type that forms part of the client's state and stores the heads of the DAG of updates that has been applied to the CRDT.

Formally, we define the **Cursor** type as:

Field	Type	Description
<i>heads</i>	Set of Hash	List of the hashes of head nodes

TABLE 5.1: **baseline** **Cursor** definition.

5.2.2 Node

Node represents the nodes of the DAG, each object contains update data, as well as the identifier of its parents.

Additionally, **Node** includes the identifier of the client that created the node which is used to distinguish between parallel nodes that were created with the same update data and parents.

Formally, we define the type **Node** as a record with the following fields:

To create a new node we define the function $new-node(st_C, op, v)$, which takes in client state st_C and private parameter p_{priv} , and produces a **Node**, specifically the new parents of the node are defined by the client's **Cursor** of st_C .

Field	Type	Description
<i>data</i>	Binary String	Data stored in the node
<i>parents</i>	List of Hash	Parents of the node
<i>client_id</i>	identifier	identifier of the client that created the node
<i>id</i>	hash	Hash of the node's <i>data</i> and <i>parents</i> fields

TABLE 5.2: baseline Node definition.

Creating a new node involves calculating the hash of the parents, the data and the client identifier, as such the time complexity of *new-node* is $\mathcal{O}(k + m)$, where k is the number of parents, m is the length of the data.

5.2.3 MerkleDag

The central data structure of our implementation is the **MerkleDag**, replicated across all participants to store **Node** objects and maintain their causal relationships.

When creating a new **Node**, it is crucial to know the current head nodes of the DAG, as they determine the parent relationships for the new node. To facilitate this, the **MerkleDag** includes a *heads* field, which keeps track of the current heads for later use. We will describe in detail how these heads are used during node creation in the following sections.

MerkleDag is formally defined as:

Field	Type	Description
<i>nodes</i>	HashMap of Node	Map from hash to the corresponding node
<i>heads</i>	HashSet of hashes	Set of hashes of nodes with in-degree 0

TABLE 5.3: baseline MerkleDag definition.

As each node already stores its hash, we can make use of **HashMaps** and **HashSets** whose insert and lookup operations have expected $\mathcal{O}(1)$ time complexity [71].

The API methods **CInit**, **RInit** and **CSet** are not described in detail, as their functionality is trivially to instantiate a new Merkle-DAG. Once the first client calls **CInit**, it invokes **CSet** to distribute public and private parameters p_{pub} and p_{priv} , respectively. Other methods associated with this implementation are described in the following sections.

5.2.3.1 Insertion Operations

When inserting data into a **MerkleDag**, we first create a **Node** object that wraps the data, then, add the node to the **MerkleDag**. This process is further detailed in Algorithm 1.

Algorithm 1: CUpdate procedure

Input: st_C , op : operation parameter, v : Data**Output:** st_C , u

```

1  $(D, c, prms) \leftarrow st_C$ ;
2  $node \leftarrow new\_node((D, c, prms), v, op)$ ;
3  $D.nodes.insert(node.hash, node)$ ;
4  $D.heads \leftarrow \{\}$ ;
5  $D.heads.insert(node)$ ;
6  $c.heads \leftarrow$  ;
7  $c.heads.insert(node)$ ;
8 return  $(D, c, prms), node$ 

```

When inserting a new node into the DAG, its in-degree is zero, making it a new head. Each parent of the inserted node is removed from the *heads* field, as their in-degree increases and they are no longer heads.

Because of the *heads* field, trivializing the gathering of nodes with in-degree 0, the time complexity of CUpdate the same as that of *new-node*, $O(|v| + |D.heads|)$.

In our construction, $|D.heads|$ is bounded by the total number of clients, $|C|$, as only clients can create new updates and a client cannot create 2 parallel updates, this is ensured by Algorithm 1. In practice, not all clients make simultaneous requests, so $|D.heads|$ is typically smaller than $|C|$. Yielding an effective complexity of $\mathcal{O}(1)$ in the common case.

Inserting a node in a replica is similar to Algorithm 1, except the node is provided, as a parameter, by the client. Because of this the replica may have nodes in *heads* that are not parents of the received node. As such, the replica needs to remove the parents of the received node from *heads*. This process is shown in Algorithm 2.

Algorithm 2: RUpdate procedure

Input: st_R , op : operation parameter, n : Node**Output:** st_R

```

1  $(D) \leftarrow st_R$ ;
2 foreach  $parent$  in  $n.parents$  do
3    $D.heads.remove(parent)$ ;
4  $D.nodes.insert(n.hash, n)$ ;
5  $D.heads.insert(n)$ ;
6 return  $(D)$ 

```

The time complexity of **RUpdate** depends on the size of $n.heads$, as each parent in $heads$ must be removed from the DAG's $head$ set. Therefore, the complexity is $\mathcal{O}(|n.parents|)$.

5.2.3.2 Query Operations

The **CQuery** operation evaluates the DAG by iteratively applying all operations to an initially empty state, producing a state corresponding to the evaluation of all nodes. In other words, **CQuery** computes a linearisation of the DAG. By definition, a linearisation corresponds to a topological sorting of the DAG (Definition 2.6), so we can use Kahn's topological sorting algorithm [42] to compute it efficiently. When computing a **CQuery**, cursor $heads$ field is set to DAG $heads$.

Kahn's algorithm requires an initial list of nodes with in-degree zero, which is readily provided by the **MerkleDag**'s $heads$ field. The algorithm has time complexity $\mathcal{O}(V + E)$, where V is the number of vertices and E the number of edges in the DAG. Consequently, **CQuery** has time complexity $\mathcal{O}(n + p)$, where n is the number of nodes and p the total number of parent references.

Clients can request the DAG from replicas via **RQuery**. This operation simply returns the replica's full **MerkleDag**, resulting in a complexity of $\mathcal{O}(n)$.

5.2.3.3 Merge and Messages

Propagate and Merge are functions used to synchronize **MerkleDag** across replicas and client back-ends. In this instance, both **CPropagate** and **RPropagate** return the complete **MerkleDag** to the recipient.

Upon receiving a message, the appropriate merge function is executed: **CMerge** if the recipient is a client back-end, or **RMerge** if the recipient is a replica.

The merge procedure is inspired by Khan's topological sorting algorithm. Evaluation begins at the $heads$ of the DAG, which are initially added to a stack. While the stack is not empty nodes are processed iteratively. For each node, if it is not already present in the Auth-MDAG, it is added to the new DAG and its parents are added to the stack.

CMerge is shown in Algorithm 3.

Algorithm 3: CMerge and RMerge Procedure**Input:** st, j : Replica Id, m : Propagate Message**Output:** MerkleDag

```

1  $(D, \dots) \leftarrow st$ ;      // only Auth-MDAG is required, for clients cursor and
   params are ignored
2  $D' \leftarrow D$ ;
3  $S \leftarrow m.heads$ ;
4 while  $S$  is not empty do
5    $n \leftarrow S.pop()$ ;
6   if  $D.nodes.lookup()$  then
7      $D'.heads.remove(n)$ ;
8   else
9      $D'.nodes.insert(n)$ ;
10    foreach  $p$  in  $n.parents$  do
11       $S.insert(p)$ ;
12  $D'.heads \leftarrow D'.heads \cup m.heads$ ;
13 return  $D'$ ;

```

The complexity of this method is determined by iterating through all parents of all the nodes in m . Consequently, the worst-case time complexity is $\mathcal{O}(n' + p')$, where n' is the number of nodes in m and p' is the number of parents references.

5.3. Authenticated implementation

This implementation augments the **baseline** by providing guarantees of authentication and integrity through the use of MACs. While the basic operations and data structures remain the same, each **Node** is now cryptographically signed to prevent tampering and ensure authenticity and integrity.

During *CInit*, clients generate a secret key k , which is subsequently used to sign every new **Node**. The first client to generate k invokes *CSet* to securely distribute the key to all other clients.

5.3.1 Node

Node is augmented with a MAC generated from the *data*, *parents* and *client_id* fields, using a password that only the clients have.

The new data type specification is as follows:

Field	Type	Description
<i>data</i>	Binary String	Data stored in the node
<i>parents</i>	List of Hash	Parents of the node
<i>client_id</i>	Identifier	identifier of the client that created the node
► <i>id</i>	Hash	MAC of the node

► indicates a field introduced or changed from the previous definition.

TABLE 5.4: `authenticated Node` definition.

For a MAC scheme $(\text{Gen}, \text{MAC}, \text{Vrf})$, A node $u = (data, parents, client_id, id)$ is *valid* iff $id = \text{MAC}(k, data || client_id || \text{order}(parents))$ where $\text{order}(parents)$ is the lexicographical ordering of the parent identifiers array.

5.3.2 MerkleDag

In this context, an Auth-MDAG is considered valid if every node it contains is valid (*i.e.*, each node is correctly signed with the client's key k , see Section 5.3.1). Validity therefore entails both authentication and integrity.

All client operations preserve the invariant that the DAG remains valid: `CUpdate` generates a new hash which, by construction, is signed with k and therefore valid. `CQuery` operates exclusively on the existing DAG, does not produce any side effects. that the DAG is valid, `CQuery` cannot compromise its validity.

On the other hand, `CMerge` requires modification to preserve validity. The change is shown in Algorithm 4. And it ensures that the client's local DAG remains valid under all permitted operations.

Algorithm 4: Authenticated CMerge Procedure

Input: st_C, j : Replica Id, m : Propagate Message**Output:** st_C

```

1   $(D, c, prms) \leftarrow st_C$ ;
2   $D' \leftarrow D$ ;
3   $S \leftarrow m.heads$ ;
4  while  $S$  is not empty do
5       $n \leftarrow S.pop()$ ;
6      if  $D.nodes.lookup()$  then
7           $D'.heads.remove(n)$ ;
8      else
9          // Check authenticity of node using MAC
10         if  $n.id \neq MAC(prms, n.data \parallel order(n.parents))$  then
11             return  $D$ ; // Abort: falsification detected
12          $D'.nodes.insert(n)$ ;
13         foreach  $p$  in  $n.parents$  do
14              $S.insert(p)$ ;
15  $D'.heads \leftarrow D'.heads \cup m.heads$ ;
16 return  $(D', c, prms)$ ;

```

The complexity of this method is determined by iterating through all parents of all the nodes in m . Consequently, the worst-case time complexity is $\mathcal{O}(n' + p')$, where n' is the number of nodes in m and p' is the number of parents references.

The complexity of the **baseline** implementation of CMerge is $\mathcal{O}(n' + p')$ where n' is the number of nodes in m and p' is the number of parent references. However since clients now need to verify each update, the complexity of the CMerge of **authenticated**, becomes $\mathcal{O}(n' + v + p')$ where n' and p' maintain their previous definition, and v is the total size of update messages.

5.4. Caching Optimization

Consider an example scenario with two replicas r_1 and r_2 , maintaining subsets of the system DAG. The system DAG consists of seven nodes labelled 1 through 7, its structure is illustrated in Figure 5.2. Initially, r_1 's state includes nodes 1 through 5 and r_2 's state includes node 1, 2 and 3. At this stage, r_1 generates a propagate message m_1 for r_2 .

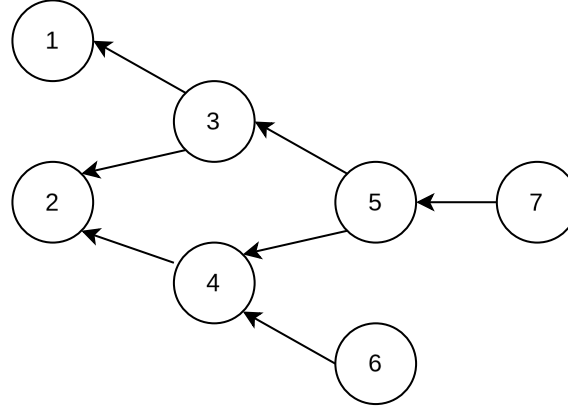


FIGURE 5.2: An example system DAG with seven nodes labelled 1 through 7.

Subsequently, r_1 receives nodes 6 and 7 and generates a second propagate message m_2 for r_2 .

In the case of **authenticated**, the full state is included in propagate messages, as such, $m_1 = \{1, 2, 3, 4, 5\}$ and $m_2 = \{1, 2, 3, 4, 5, 6, 7\}$. Between the two messages, five nodes where repeated.

This happens because, sending the full **MerkleDag** leads to repeated messages. As such, in this chapter we introduce **cached**, which showcases optimizations with the objective of reducing the size of messages generated with **CPropagate** and **RPropagate**.

To achieve this, we add two new fields to **MerkleDag**, *linear* and *cache*. The former is an array that stores the hashes of the nodes in the order that they arrived in. The latter is a map from machine ids(replica or client) to tuples of the form (hash map, integer), where, the integers indicates the index of the last merged node between the 2 clients, the hash map represent the heads that the target machine has knowledge of.

Field	Type	Description
<i>nodes</i>	HashMap of Node	Map from hash to the corresponding node
<i>heads</i>	HashSet of hashes	Set of hashes of nodes with in-degree 0
► <i>linear</i>	Array of Node	An array that records the sequence in which nodes were received.
► <i>cache</i>	Map from replica/client id to (Set of Node, Integer)	A map of information that our MerkleDag maintains about the target replica or client.

► indicates a field introduced or changed from the previous definition.

TABLE 5.5: **cached MerkleDag** definition.

Because when a node is inserted into the DAG it's parents have to already be inside the DAG, this means that *linear* stores a valid linearisation, which we can use to:

1. Efficiently identify which nodes have already been sent to a user/replica,
2. Optimize CQuery to $\mathcal{O}(n)$ time complexity as its process is trivialized by using *linear* to apply the operations, as opposed to calculating a linearisation each time it is called.

Furthermore, we added a new bookkeeping field to **Node**, the *index* field. This field stores the index that was given to this node, *i.e.*, if a node has *index* = 10, this means that this node was the 10th node to arrive to this entity.

However, while receiving new updates cannot invalidate *linear*, it is necessary to ensure that CMerge adds the merged nodes to *linear* in the correct order. To achieve this we add a new field to **Node**, the *layer* field. This field stores an integer and its value is obtained by incrementing by one the maximum layer of all of the node's parents. An example of how *layer* is calculated is shown in Figure 5.3.

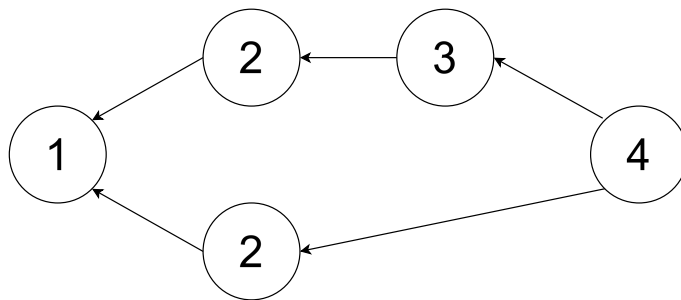


FIGURE 5.3: Minimal example of how layer property works, the labels of each node are their layer value.

Field	Type	Description
<i>data</i>	Binary String	data stored in the node
<i>parents</i>	List of Hash	parents of the node
<i>client_id</i>	Identifier	identifier of the client that created the node
<i>id</i>	Hash	MAC of the node
► <i>index</i>	integer	index of the node in the <i>linear</i> array
► <i>layer</i>	integer	layer of the node

► indicates a field introduced or changed from the previous definition.

TABLE 5.6: **cached Node** definition.

In Merge procedures, we can use this field along with a binary heap (as opposed to the previous implementation's stack) with a custom comparison function where nodes with a higher *layer* value are selected first. This ensures that all nodes of layer i will be processed before those of layer $i - 1$. And as such, we can use a simple array to store the order in which the nodes were processed and, upon completing the state merge, add to *linear* the reverse of that array. This process is described in Algorithm 5.

Algorithm 5: Correction of the CMerge procedure**Input:** st_C, j : Replica Id, m : Propagate Message**Output:** st_C

```

1   $(D, c, prms) \leftarrow st_C$ ;
2   $D' \leftarrow D$ ;
3   $BH \leftarrow m.heads$ ;
4   $arr \leftarrow []$ ;
5  while  $BH$  is not empty do
6       $n \leftarrow BH.pop()$ ;
7       $arr.push(n)$ 
8      if  $D.nodes.lookup()$  then
9           $D'.heads.remove(n)$ ;
10     else
11         // Check authenticity of node using MAC
12         if  $n.id \neq MAC(prms, n.data \parallel order(n.parents))$  then
13             return  $D$  ;                               // Abort: falsification detected
14          $D'.nodes.insert(n)$ ;
15         foreach  $p$  in  $n.parents$  do
16              $BH.insert(p)$ ;
17  $D'.heads \leftarrow D'.heads \cup m.heads$ ;
18  $D'.linear \leftarrow D.linear \parallel arr.reverse()$ ;
19 return  $(D', c, prms)$ ;

```

Finally, in the RPropagate method, we make use of the target's id j , to identify which nodes the target has knowledge of, and propagate only those we know the target doesn't have knowledge of. This process is described in Algorithm 6.

Algorithm 6: RPropagate procedure

Input: st_R, j **Output:** Propagate Message: Array of Node

```

1   $(D) \leftarrow st_R$ ;
2   $index \leftarrow D.linear.length() - 1$ ;
3   $ret \leftarrow []$ ;
4   $(thead, tindex) \leftarrow D.cache[j]$ ;
5   $visited \leftarrow [tindex : index]$ ;
6  while  $index > tindex$  do
7       $node \leftarrow D.linear[index]$ ;
8      if  $node \in theads \vee visited[index - tindex]$  then
9          foreach  $p$  in  $node.parents$  do
10              $pi \leftarrow D.nodes.lookup(p).index$ ;
11              $visited[pi - tindex] \leftarrow true$ ;
12      else
13           $ret.push(node)$ ;
14       $index - -$ ;
15  $return \leftarrow reverse(ret)$ ;
16  $D.cache[j] \leftarrow (D.heads.l.length() - 1)$ ;
17 return  $return$ 

```

Revisiting the example of Figure 5.2. In the initial state, replica r_1 's *linear* field is set to $[2, 1, 4, 3, 5]$. And replica r_2 's *linear* field set to $[2, 1, 3]$. At this stage, r_1 generates a propagate message m_1 targeted at r_2 . Since r_1 has not previously exchanged messages with r_2 , its local r_1 's cache is empty and, consequently, m_1 includes all nodes known to r_1 . Upon successfully delivered, r_1 updates its cache, $cache[2] \leftarrow (\{5\}, 5)$.

Subsequently, r_1 receives nodes 6 and 7, r_1 *linear* becomes $[2, 1, 4, 3, 5, 7, 6]$, and the *heads* is updated to $\{6, 7\}$.

When r_1 prepares m_2 , following the procedure of Algorithm 6, the optimization utilizes the cached information. In this instance, r_2 starts from the node whose *index* = 7 (node 6), its index is lower than *tindex* (*tindex* = 5), it is not a head and it is not visited, thus it gets included in the message. Similarly, node 7 whose *index* = 6 satisfy the same conditions and is also added to the message. Once the iteration reaches node 5 we have *index* = 5 $\not>$ *tindex*, terminating the while loop. The *ret* array is reversed and returned,

producing a propagation message consisting only of nodes 6 and 7.

This optimization yields significant performance improvements when two entities (client or replica) exchange messages frequently, as both index and heads of the target will be more accurate.

Merge functions then use the messages generated by Algorithm 6 to merge state. Furthermore they update the heads associated with the replica who generated the merge message with the heads that came in the message.

The complexity of this method is dominated by, in the worst case, having to traverse the full *linear* array while having to lookup each node, thus the time complexity for this method is $\mathcal{O}(p + n)$. Amortized across a sequence of calls, each node and each parent-edge is processed at most once, thus, the total work over the whole sequence is $\mathcal{O}(p + n)$, *i.e.*, the complexity per call is $\mathcal{O}(1)$

In addition, this optimization can be applied to RQuery, guaranteeing the same complexity.

Following these caching concepts, we can further improve CQuery by using the clients **Cursor**. By adding a *index* field to **Cursor**, we can simply return to the client all nodes of the **MerkleDag**'s *linear* after the cursor's *index* field.

Field	Type	Description
	<i>heads</i>	Set of Hash List of the hashes of head nodes
►	<i>index</i>	integer of the last query in the <i>linear</i> array

TABLE 5.7: cached **Cursor** definition.

Although a single call to CQuery may take $\mathcal{O}(n)$ time in the worst case, each node of the linear array is transferred to the client at most once. Hence, over a sequence of operations, the amortized time complexity of CQuery is $\mathcal{O}(1)$.

5.5. Matrix

Matrix [11, 12] is a decentralized, topic-based publish-subscribe middleware for communication and data storage, particularly used for secure instant messaging. Unlike traditional messaging systems that rely on a single, centralized server, Matrix operates on a federated network of homeservers, where each server independently stores user data and participates in message delivery. This architecture allows users on different servers to communicate seamlessly, much like how email works between different service providers, while also

enabling anyone to host a homeserver, ensuring decentralization by design rather than relying on a single central authority.

The protocol’s core data structure is the Matrix Event Graph (MEG), which is a DAG that acts as a CRDT. Ensuring a consistent and causally-ordered history of events across all participants [12].

Our implementation, **matrix**, leverages Matrix solely as a distributed transport and synchronization layer: a dedicated Matrix room represents the application state and the protocol handles real-time replication and conflict resolution. We do not interact with the MEG directly because MEG is only exposed inside Matrix homeservers. Manipulating MEG therefore would require implementing (or heavily modifying) a full Matrix homeserver to gain access to its internal data structures and life cycle. This would introduce substantial protocol, security, and maintenance complexity outside the scope of this project. By using Matrix at the client/room level instead, **matrix** reuses the existing, well-tested replication and conflict-resolution semantics without forcing us to become homeserver implementers.

The core of our implementation is the encoding of our application’s Auth-MDAG onto the Matrix event system. To achieve this, we model each node of our Auth-MDAG as a custom Matrix event. Specifically, we define a new event type **UpdateEvent** and send it to our designated Matrix room via the Client-Server API. The content field of our event contains the serialized Auth-MDAG node, including its cryptographic hash and the hashes of its parent nodes.

As a result, sending a new node, through **CUpdate**, is as simple as creating and dispatching a single Matrix event. The underlying Matrix protocol then ensures this event, and thus the new Merkle node, is reliably replicated and ordered across all room participants. This effectively turns the MEG into our Auth-MDAG.

With this in mind, similar to the previous implementations, each client maintains a local **MerkleDag** state that is continuously reconstructed from the replicated MEG.

CInit and **CSet** correspond to creating (or joining) a designated Matrix room and recording its identifier within the client state.

CPropagate, **CMerge** and all replica logic, are realized by the Matrix synchronization mechanism: each homeserver propagates new events to all participating replicas, and clients incorporate received events into their local **MerkleDag** state.

6. Experimental Setup

This chapter benchmarks the execution time of our different implementations. Specifically, we evaluate how the different implementations behave under the same conditions. We use these benchmarks to assess the overhead of an instance of the proposed system in a simple, controlled scenario.

We begin by outlining the system architecture and the methodology used to conduct our benchmarking experiments. The evaluation is subsequently divided into micro benchmarks and macro benchmarks. The former focuses on controlled scenarios designed to isolate and assess individual operations. The latter captures more realistic usage patterns, simulating multi-user environments and extended interaction sessions.

6.1. Benchmarks Architecture

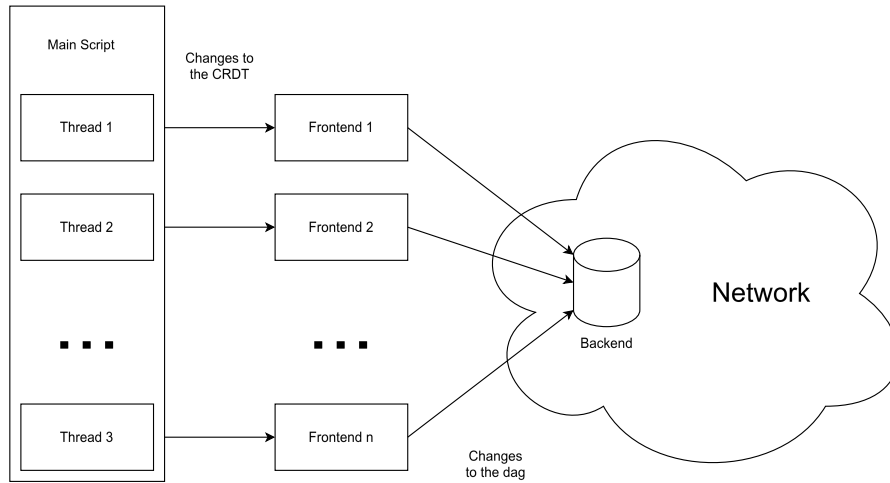


FIGURE 6.1: Architecture of benchmarks.

The benchmarking process replaces the user with a script, as shown in Figure 6.1, as such, it involves three parties: the generation script, the front-ends and a back-end.

The generation script manages different threads, where each thread generates changes that are sent to the different front-end instances. Each front-end instance, keeps a CRDT and applies the changes issued by the generation script, then the front-end generates diffs which are sent to the back-end and appended to the DAG.

6.2. System Configuration and Reproducibility

To ensure the reproducibility of our system, we provide full details of the software environments used in both the front-end and back-end components. All relevant source

code, dependency configurations, and instructions for building and running the system are included in the project repository [72].

All experiments and deployments were performed on a consistent development environment. Versioning is strictly documented to avoid discrepancies across environments.

All experiments were conducted on a desktop system equipped with an AMD Ryzen 5 5600G processor (12 cores, 4.46 GHz), an AMD Radeon RX 6600 GPU, and 20 GB of RAM.

The front-end was developed in JavaScript using the following tools and dependencies:

- `npm` version 10.9.2
- `automerger` version 2.2.8
- All other dependencies and exact versions are specified in the `package.json` file.

The back-end system was built using the Rust programming language with the following toolchain:

- `rustc` 1.86.0-nightly (124cc9219 2025-02-09)
- `cargo` 1.86.0-nightly (2928e3273 2025-02-07)
- `matrix-sdk` version 0.7.1
- All additional dependencies and versions are explicitly declared in the `Cargo.toml` file.

6.3. Micro benchmarking

Micro benchmarks were designed to quantify latency and cost of individual operations in a controlled environment. To this end, we execute sequences of random CRDT `update` operations on the front end, where each update is transmitted to the server. After every batch of n updates the front end issues a `query` operation. This arrangement isolates per-operation costs by minimizing interference from unrelated system components.

We exclude `matrix` from micro benchmarks because even in a minimal configuration a matrix server introduces substantial protocol- and infrastructure-level overheads that would confound measurements intended to capture the cost of the elementary operations. In

contrast, our custom implementations permit direct measurement of operation latency within the system under test.

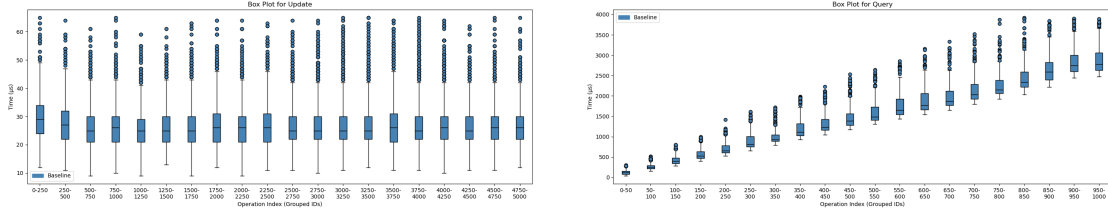
For the experimental evaluation defined of this section, we use $n = 5$, *i.e.*, after 5 updates, the front-end performs a query. This sequence is repeated 1000 times, resulting in a total of 1000 query and 5000 update.

With each operation we measure the time taken for the operation to be complete on the local DAG copy. For instance, for the **update** operation, we measure the time needed for the node to be generated and appended to the DAG. And, for the **query** operation, we measure the time needed to iterate through the nodes of the DAG, finishing when the collection of queried nodes is returned to the client.

For readability we removed the top 0.5% of results from the figures. For completeness, the complete graphs are shown in Appendix A.

In the next sections we go over each of the implementations while showing the results obtained. Then, at the end, we provide a comparison of the results obtained highlighting their differences.

6.3.1 baseline



(A) Box plot of **update** execution times.

(B) Box plot of **query** execution times.

FIGURE 6.2: Micro benchmarks results of the **baseline** implementation.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1-250]	24	29	34	79	117	155
Box 5 [1001-1250]	21	25	30	606	656	783
Box 10 [2251-2500]	21	26	31	1281	1386	1561
Box 15 [3501-3750]	21	26	31	1922	2034	2290
Box 20 [4751-5000]	22	26	30	2639	2792	3105

TABLE 6.1: **baseline** micro benchmark details table.

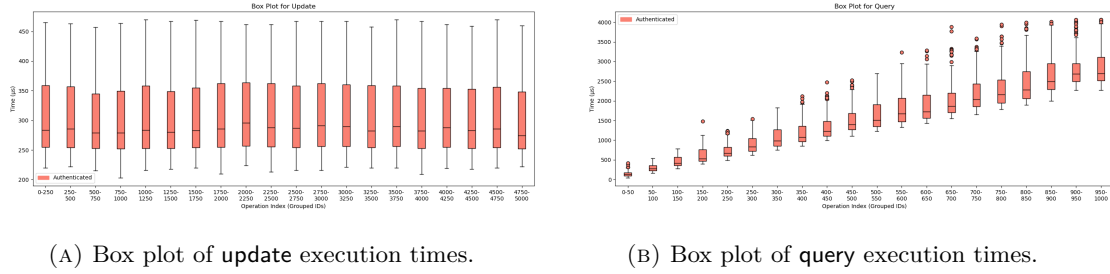
Micro-benchmark results for the **baseline** implementation are shown in Figure 6.2, and detailed in Table 6.1.

The **update** operation (Figure 6.2a), shows an asymptotically constant latency, fluctuating between $20\mu\text{s}$ to $30\mu\text{s}$, with a median around $26\mu\text{s}$.

The **query** operation (Figure 6.2b) on the other hand, exhibits a linear increase over time. The median increases from $117\mu\text{s}$ in the first 250 operations to $2792\mu\text{s}$ by the end of the benchmark.

Overall, the results confirm the analysis in Section 5.2, with **update** exhibiting constant-time latency and **query** growing linearly with the number of operations.

6.3.2 authenticated



(A) Box plot of **update** execution times.

(B) Box plot of **query** execution times.

FIGURE 6.3: Micro benchmark results for **authenticated** implementation.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1-250]	255	285	360	91	135	177
Box 5 [1001-1250]	253	285	358	594	670	822
Box 10 [2251-2500]	256	288	363	1265	1399	1683
Box 15 [3501-3750]	256	292	359	1854	2037	2431
Box 20 [4751-5000]	252	275	348	2521	2707	3168

TABLE 6.2: **authenticated** micro benchmark details table.

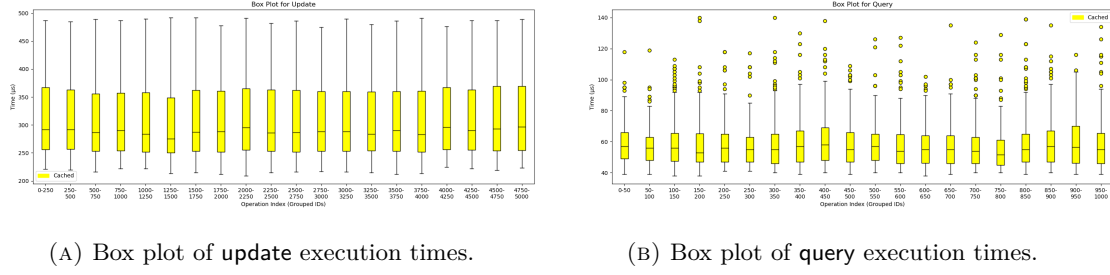
Micro-benchmark results for the **authenticated** implementation are shown in Figure 6.3, and detailed in Table 6.2.

The **update** operation (Figure 6.3a) shows asymptotically constant latency, fluctuating between $255\mu\text{s}$ and $360\mu\text{s}$, with a median around $280\mu\text{s}$.

By contrast, the **query** operation (Figure 6.3b) exhibits a linear increase in latency. The median grows from $135\mu\text{s}$ in the first 250 operations to $2707\mu\text{s}$ by the end of the benchmark, which is consistent with the expected $\mathcal{O}(n)$ behaviour described in Section 5.2.3.2.

Overall, these results confirm the theoretical analysis of the **authenticated** implementation.

6.3.3 cached

(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE 6.4: Micro benchmarks results of the **cached** implementation.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1-250]	256	293	369	49	57	66
Box 5 [1001-1250]	252	284	359	47	56	65
Box 10 [2251-2500]	253	286	363	47	55	66
Box 15 [3501-3750]	253	291	361	46	54	63
Box 20 [4751-5000]	255	298	370	46	55	66

TABLE 6.3: **cached** micro benchmark details table.

Micro-benchmark results for the **cached** implementation are shown in Figure 6.4, and detailed in Table 6.3.

As in other benchmarks, we removed outliers beyond the top 0.5%. For this experiment, this corresponds to 125 update entries and 24 query entries.

The **update** operation (Figure 6.4a) shows asymptotically constant latency, fluctuating between 252 μ s and 370 μ s, with a median around 290 μ s.

Similarly, the **query** operation (Figure 6.4b) also remains constant throughout the benchmark, fluctuating between 46 μ s and 66 μ s, with a median around 56 μ s.

Overall, these results confirm that the caching optimization introduced in Section 5.4 is effective, yielding constant-time queries as expected.

6.3.4 Micro Benchmark Results Discussion

Finally, in Figure 6.5 we have the results of the experiments of **baseline** (Section 6.3.1), **authenticated** (Section 6.3.2), and **cached** (Section 6.3.3) plotted in the same graph for easier comparison.

In Figure 6.5a we can see that the **update** operation is constant in all evaluated implementations. But, while the **cached** and the **authenticated** implementations have an average latency value of 300 μ s, the **baseline** implementation has an average latency of 40 μ s.

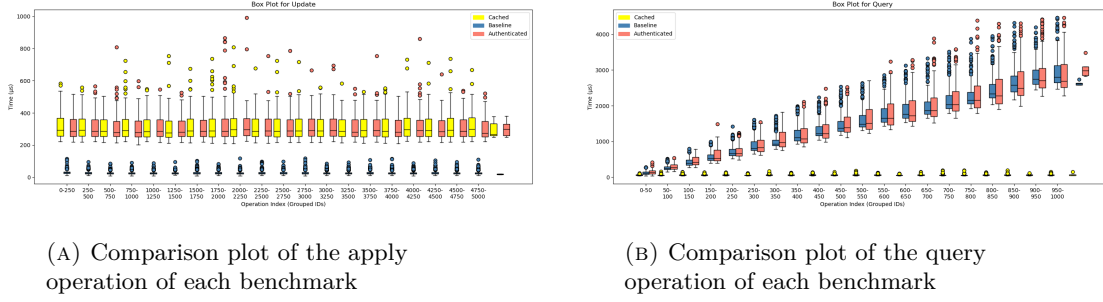


FIGURE 6.5: Comparison of the micro benchmark results of the different implementations.

This is expected as **baseline** implementation is the only implementation that doesn't generate a MAC tag for the new node, which is an event that takes considerable amount of time.

In Figure 6.5b we can see that the **query** operation is constant for the **cached** implementation but it is linear for both **baseline** and **authenticated** implementations. The latter 2 implementations have similar latency values for the **query** operation.

Overall, these results highlight the trade-offs between the implementations. The **baseline** implementation has the lowest update latency by avoiding MAC generation and thus does not provide authenticity, but its query latency grows linearly. Both **authenticated** and **cached** ensure authenticity through MAC generation, resulting in higher update latency, with **cached** achieving constant-time queries while **authenticated** retains linear query growth. This comparison demonstrates how authentication and state-management choices directly influence performance.

6.4. Macro benchmarking

For the macro-benchmark evaluation, our goal is to understand how the system behaves under conditions that resemble realistic deployment scenarios. Unlike the micro-benchmarks, which focus on isolated operations and controlled environments, the macro-benchmarks are designed to capture the performance of the full protocol stack when exercised by workloads derived from real-world data.

After defining the workload, we present the results of our macro-benchmark experiments, followed by a detailed discussion and interpretation. Similarly to micro benchmarks, we remove the top 0.5% of results to improve figure clarity. The full graphs are provided in Appendix A.

6.4.1 Workload Generation

We generate workloads based on FMKe benchmark framework [73], it is a benchmark that is designed to emulate the system that manages medical information of Denmark. This framework emulate how a collection of medical professionals manages prescriptions. For instance, a patient goes to the hospital where a doctor creates a prescription for him. Then, the patient goes to a pharmacy to collect their medicine. If all medicine in the prescription is collected then the pharmacy marks the prescription as processed.

To achieve this, FMKe defines the operations: get pharmacy prescriptions, get prescription medication, get staff prescriptions, create prescription, get processed pharmacy prescriptions, process prescription and update prescription medication. Each of these operations have different parameters and different frequency in a workload. This data is shown in Table 6.4. Furthermore, we include a column that specifies the operation each of these methods calls when they are executed. For example, when getting a pharmacy prescription, we first query our DAG.

Operation	Frequency	Parameters	DAG Method
Get pharmacy prescriptions	27%	pharmacy	Query
Get Prescription Medication	27%	prescription	Query
Get Staff Prescriptions	14%	medical staff	Query
Create Prescription	8%	patient, medical staff, pharmacy	Update
Get Processed Pharmacy Prescriptions	7%	pharmacy	Query
Get Prescription	4%	prescription	Query
Process Prescription	4%	prescription	Update
Update Prescription Medication	4%	prescription	Update
Get Patient	4%	patient	Query
Update Prescription Medication	4%	prescription	Update

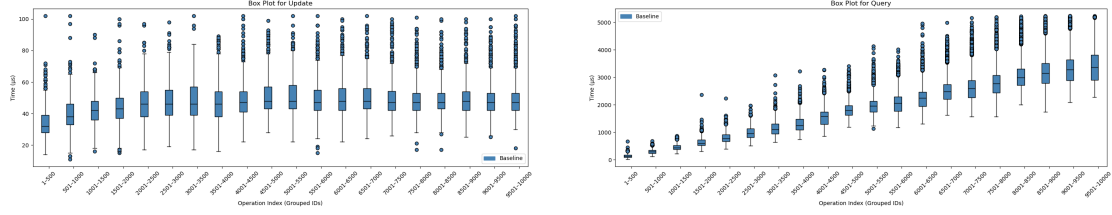
TABLE 6.4: FMKe operation table.

The parameters column represents the information needed to perform that operation, these parameters are populated by choosing at random from Table 6.5 according to a Zipf law [74] such that the parameters choice reflects the natural distribution of the data, where the most frequent element appears roughly twice as much as the second most frequent.

Entity	Number
Patients	1,000,000
Hospitals	50
Pharmacies	300
Doctors	5,000

TABLE 6.5: Number of entities of the workload

6.4.2 baseline

(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE 6.6: Macro benchmark results for **baseline**.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1–500]	28	32	39	95	135	179
Box 5 [2001–2500]	38	46	54	673	777	922
Box 10 [4501–5000]	43	48	58	1628	1805	1974
Box 15 [7001–7500]	42	47	55	2267	2601	2885
Box 20 [9501–10000]	42	47	54	2932	3410	3901

TABLE 6.6: Results of the **baseline** macro benchmark.

Results of macro benchmarks of the **baseline** implementation are shown in Figure 6.6 and detailed in Table 6.6.

For the **update** operation (Figure 6.6a) the median latency of the first 500 operations is 32 μs . Over the next 2000 operations, as the number of operations increases, the latency rises to 45 μs , after which it stabilizes for the remainder of the benchmark.

In contrast, the **query** operation (Figure 6.6b) exhibits a linear behaviour. The median latency increases from less than 135 μs at the beginning to a median of 1800 μs after 2500 operations, eventually reaching over 3400 μs . Furthermore, interquartile range widens considerably, indicating greater variability in query performance as the number of **updates** grows.

These results confirm the expected behaviour indicated in Section 5.2. Specifically, the **update** operation exhibits constant-time performance, while the **query** operation shows a linear-time increase. Furthermore, these results are consistent with those observed in **baseline** micro-benchmarks (Section 6.3.1).

6.4.3 authenticated

Results of macro benchmarks of the **authenticated** implementation are shown in Figure 6.7 and detailed in Table 6.7.

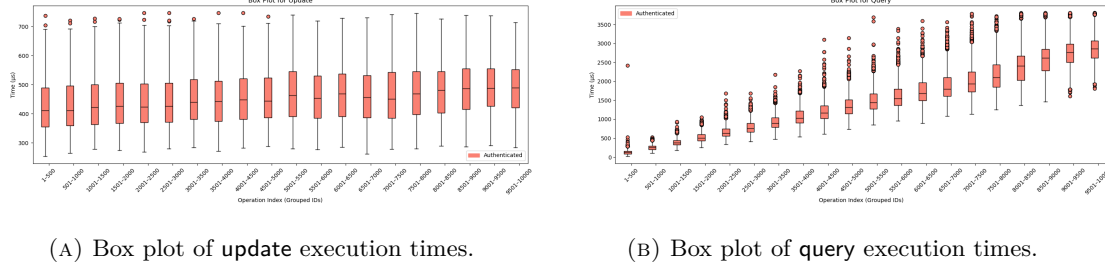


FIGURE 6.7: Macro benchmark results for Authless

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1–500]	356	411	491	87	122	162
Box 5 [2001–2500]	371	423	506	558	630	745
Box 10 [4501–5000]	386	446	525	1146	1312	1513
Box 15 [7001–7500]	386	452	544	1725	1935	2248
Box 20 [9501–10000]	422	492	555	2632	2873	3114

TABLE 6.7: Results of the Authless Macro benchmark

For the **update** operation (Figure 6.7a) the median increases gradually, starting from 411 μ s and finishing at 492 μ s.

For the **query** operation (Figure 6.7b) the median rises steadily over the course of the benchmark, starting from 122 μ s and finishing at 2872 μ s.

These results confirm the expected behaviour indicated in Section 5.3. Specifically, as with **baseline**, the **update** operation exhibits constant-time performance, while the **query** operation shows a linear-time increase. However, the **update** latency of **authenticated** is significantly higher than that of **baseline**, due to the overhead of MAC generation and the associated authenticity guarantees. Overall, the results are consistent with **authenticated** micro benchmarks (Section 6.3.2) and illustrate the trade-off between maintaining authenticity and achieving low update latency.

6.4.4 cached

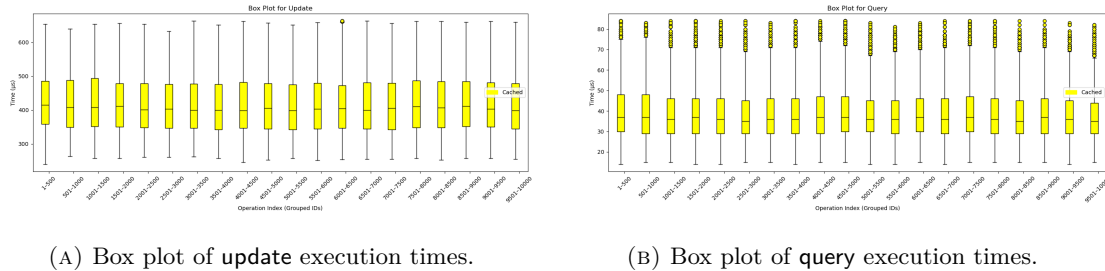


FIGURE 6.8: Macro benchmark results for cached.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1–500]	359	416	487	30	38	49
Box 5 [2001–2500]	350	401	478	29	36	46
Box 10 [4501–5000]	344	402	478	30	37	47
Box 15 [7001–7500]	344	407	483	30	37	47
Box 20 [9501–10000]	347	402	480	29	35	44

TABLE 6.8: Detailed results of **cached** macro benchmark.

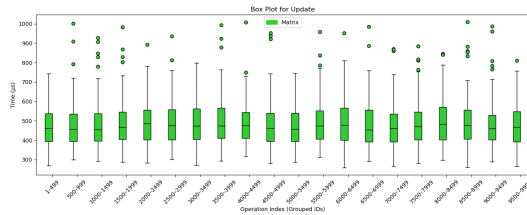
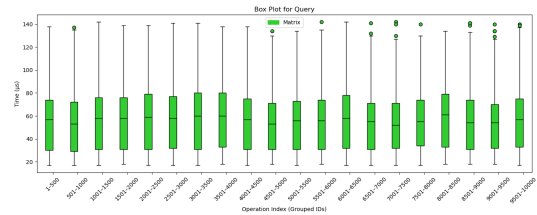
Macro benchmark results for the **cached** implementation are shown in Figure 6.8 and detailed in Table 6.8.

For **update** operation (Figure 6.8a) median stays consistent over the duration of the benchmark. Specifically, latency, for the initial 500 operations, is $416\mu\text{s}$, and, at the benchmark’s end, latency is $402\mu\text{s}$.

Similar to **update** operation, The latency of the **query** operation (Figure 6.8b), constant throughout the benchmark. However, the latency of each **query** is significantly lower than that of a **update**, sitting at around $37\mu\text{s}$ which is around ten times better than the latency of each **update**.

Just like **baseline** and **authenticated**, **cached**’s macro benchmarks further solidify both the complexity calculated in Section 5.4, and the micro benchmark results obtained in Section 6.3.3.

6.4.5 matrix

(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE 6.9: Macro benchmark results for **matrix**.

Macro-benchmark results for the **matrix** implementation are shown in Figure 6.9 and detailed in Table 6.9.

For the **update** operation (Figure 6.9a), latency remains stable throughout the benchmark. Specifically, the median latency hovers around $470\mu\text{s}$, with no observable growth over time.

Box	Update			Query		
	Q1	Median	Q3	Q1	Median	Q3
Box 1 [1–499]	394	462	539	30	57	74
Box 5 [2000–2498]	400	484	556	31	59	80
Box 10 [4499–5000]	396	461	541	31	53	71
Box 15 [7001–7500]	386	458	532	32	52	71
Box 20 [9501–10000]	392	469	550	33	57	75

TABLE 6.9: Detailed results of **matrix** macro benchmarks.

Similarly, the latency of **query** operations (Figure 6.9b) also remains constant. Here, the median latency is approximately $55\mu\text{s}$, which is significantly lower than that of **update** operation.

These results demonstrate that the **matrix** implementation achieves steady performance for both **update** and **query** operations throughout the experiment, validating its suitability for macro-scale workloads.

6.4.6 Discussion

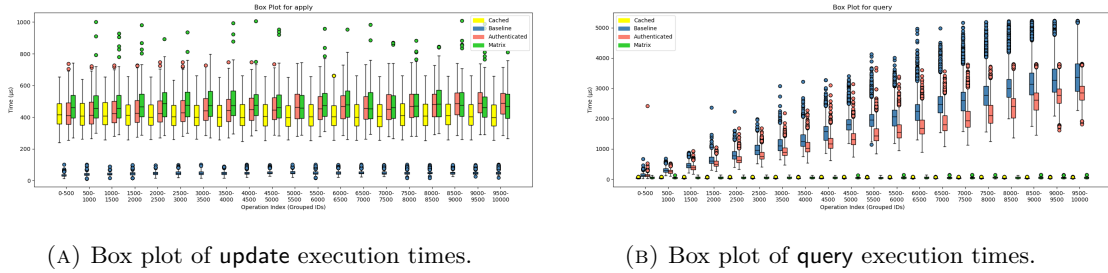
(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.

FIGURE 6.10: Comparison of the macro benchmark results of the different implementation.

The macro benchmark results (comparison shown in Figure 6.10) are consistent with the results observed in micro benchmarks Section 6.3.4 and with the differences previously mentioned in the implementation chapter, Chapter 5.

First, as shown by Figure 6.10a, **baseline**'s **update** operations are significantly less expensive than that other implementations, because it is the only implementation that does not use MACs.

Second, as shown by Figure 6.10b, **baseline**'s and **authenticated**'s **query** operations grow linearly with the number of updates, while **cached** and **matrix** are constant, demonstrating that the caching optimization of Section 5.4 works properly.

Finally, there is no observable asymptotic difference between `cached` and `matrix`. The performance curves of the two implementations overlap closely, indicating that while this approach does impose the expected authentication burden typical of cryptographic techniques of this kind, it does not incur additional costs. As such, it demonstrates potential for use in systems that prioritize integrity without compromising performance.

7. Conclusion and Future Work

Distributed systems increasingly rely on replicated data structures to provide high availability and responsiveness. This thesis addresses the challenge of building replicated, authenticated data structures that support these conditions on environments with multiple clients and multiple servers. Previous work in verifiable data structures, such as [3] that focuses on single-server settings. The challenges of multiple concurrently updating clients. While CRDTs provide strong guarantees for conflict-free replication, they do not inherently ensure authenticity or tamper-evidence.

Our research extends these ideas to a multi-client setting, providing a framework for Auth-CRDTs built on top of an Auth-MDAG. This construction allows multiple clients to perform operations concurrently while guaranteeing both convergence and the authenticity of each update.

Specifically, in this thesis, we developed a general framework for RADT, beginning with a formal definition of an Auth-MDAG and extending it by integrating the semantics of a CRDT to produce Auth-CRDT. Alongside these formal definitions, we designed and implemented multiple practical instantiations of Auth-CRDT, each representing an incremental improvement over the previous version. The final implementation leveraged MEG as a transport layer, demonstrating the feasibility of deploying Auth-CRDT in a distributed environment.

In addition to the formal and practical constructions, we performed a comprehensive experimental evaluation of all implementations. These experiments empirically demonstrated the relative performance and shortcomings of each approach, as well as the computational and communication complexity of individual operations. Together, these results provide both a rigorous theoretical foundation and practical insights into the behaviour and performance of Auth-CRDT under realistic conditions.

Despite the contributions of this work, several limitations remain. First, the full capabilities of the Matrix back-end could not be utilized, as implementing a homeserver was outside the scope of this thesis. Second, the Auth-MDAG grows linearly with the number of updates, which affects both storage requirements and network bandwidth, although our optimizations mitigate this effect to some extent. Third, the security and correctness properties of the Auth-MDAG were extrapolated, but not formally proven.

Several directions remain for extending and improving the framework presented in this thesis.

First, formal proofs of security and correctness for the Auth-MDAG and the resulting Auth-CRDT constructions could be developed. Such proofs would provide stronger guarantees for authenticity and tamper-evidence, addressing one of the primary theoretical gaps in the current work.

While Automerge provides a CRDT with robust support for complex data types and conflict-free replication, its operations result in a DAG that grows linearly with the number of updates. Future work could explore integrating more space-efficient CRDTs to reduce storage and bandwidth overhead, potentially combined with DAG compression techniques to further improve scalability.

Third, the current implementation assumes pre-shared secrets for authentication. Future work could explore mechanisms for securely distributing shared secrets among clients or transitioning to a public-key cryptography approach, improving scalability and deployment flexibility.

Finally, defining concrete application scenarios where the Auth-MDAG and Auth-CRDT constructions provide clear benefits would help validate the practical relevance of this framework. Potential use cases include collaborative editing systems, distributed databases, and verifiable ledgers, where tamper-evidence and conflict-free replication are critical.

Overall, this thesis demonstrates that it is possible to design and implement replicated data structures that are both conflict-free and tamper-evident in a multi-client multi-server distributed setting. By integrating formal constructions, practical implementations, and empirical evaluation, our work provides a foundation for Auth-CRDT that can inform the development of secure and reliable collaborative and distributed systems in the future.

References

- [1] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno, “Hash first, argue later: Adaptive verifiable computations on outsourced data,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1304–1316. [Cited on page 1.]
- [2] R. Tamassia, “Authenticated data structures,” in *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2003, pp. 2–5. [Cited on pages 1, 11, and 12.]
- [3] C. Cachin, E. Ghosh, D. Papadopoulos, and B. Tackmann, “Stateful multi-client verifiable computation,” in *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*. Springer, 2018, pp. 637–656. [Cited on pages 1, 2, 11, 24, 26, and 61.]
- [4] C. Cachin, A. Shelat, and A. Shraer, “Efficient fork-linearizable access to untrusted shared memory,” in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 129–138. [Cited on pages 1 and 25.]
- [5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 2011, pp. 386–400. [Cited on pages 1, 17, 18, 19, and 20.]
- [6] H. Sanjuan, S. Poyhtari, P. Teixeira, and I. Psaras, “Merkle-crds: Merkle-dags meet crds,” *arXiv preprint arXiv:2004.00107*, 2020. [Cited on pages 2 and 14.]
- [7] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Advances in Cryptology — CRYPTO ’87*, C. Pomerance, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. [Cited on pages 2, 12, and 13.]
- [8] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Available at SSRN 3440802*, 2008. [Cited on pages 2 and 13.]
- [9] K. Gai, J. Guo, L. Zhu, and S. Yu, “Blockchain meets cloud computing: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2009–2030, 2020. [Cited on page 2.]

- [10] J. Courtiel, P. Dorbec, and R. Lecoq, “Theoretical analysis of git bisect,” *Algorithmica*, vol. 86, no. 5, p. 1365–1399, Dec. 2023. [Online]. Available: <http://dx.doi.org/10.1007/s00453-023-01194-0> [Cited on pages 2 and 15.]
- [11] The Matrix.org Foundation CIC, *Matrix Specification, Version v1.16*, <https://spec.matrix.org/v1.16/>, 2025, accessed: 2025-09-23. [Cited on pages 2, 34, and 47.]
- [12] F. Jacob, C. Beer, N. Henze, and H. Hartenstein, “Analysis of the matrix event graph replicated data type,” *IEEE access*, vol. 9, pp. 28 317–28 333, 2021. [Cited on pages 2, 3, 18, 47, and 48.]
- [13] J. Benet, “Ipfs - content addressed, versioned, p2p file system,” 2014. [Online]. Available: <https://arxiv.org/abs/1407.3561> [Cited on pages 2 and 15.]
- [14] M. Kleppmann, A. Wiggins, P. Van Hardenberg, and M. McGranaghan, “Local-first software: you own your data, in spite of the cloud,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 154–178. [Cited on pages 3 and 33.]
- [15] Automerge Contributors, “Automerge: A local-first crdt library for collaborative applications,” <https://automerge.org/>, 2025, javaScript and Rust implementations with transport-agnostic sync. [Cited on pages 3, 18, and 33.]
- [16] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007. [Cited on pages 5 and 7.]
- [17] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, “Performance of the most common non-cryptographic hash functions,” *Software: Practice and Experience*, vol. 44, no. 6, pp. 681–698, 2014. [Cited on page 5.]
- [18] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018. [Cited on pages 6 and 7.]
- [19] R. Rivest, “The md5 message-digest algorithm,” Tech. Rep., 1992. [Cited on pages 6 and 7.]
- [20] X. Wang, D. Feng, X. Lai, and H. Yu, “Collisions for hash functions md4, md5, haval-128 and ripemd,” *Cryptology ePrint Archive*, 2004. [Cited on page 6.]
- [21] X. Wang and H. Yu, “How to break md5 and other hash functions,” in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2005, pp. 19–35.

- [22] M. Stevens, A. Lenstra, and B. De Weger, “Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2007, pp. 1–22. [Cited on page 6.]
- [23] R. C. Merkle, *Secrecy, authentication, and public key systems*. Stanford university, 1979. [Cited on page 6.]
- [24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Keccak sponge function family main document,” *Submission to NIST (Round 2)*, vol. 3, no. 30, pp. 320–337, 2009. [Cited on pages 6, 7, and 8.]
- [25] “Secure Hash Standard,” National Institute of Standards and Technology (NIST), Tech. Rep. FIPS PUB 180-1, Apr 1995. [Online]. Available: <https://csrc.nist.gov/publications/detail/fips/180/1/archive/1995-04-17> [Cited on page 7.]
- [26] “Secure Hash Standard,” National Institute of Standards and Technology (NIST), Tech. Rep. FIPS PUB 180-2, Aug 2002. [Online]. Available: <https://csrc.nist.gov/publications/detail/fips/180/2/archive/2002-08-01> [Cited on page 7.]
- [27] “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” National Institute of Standards and Technology (NIST), Tech. Rep. FIPS PUB 202, Aug 2015. [Online]. Available: <https://csrc.nist.gov/publications/detail/fips/202/final> [Cited on pages 7 and 8.]
- [28] M. Bellare, R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication,” in *Annual international cryptology conference*. Springer, 1996, pp. 1–15. [Cited on page 8.]
- [29] J. Black and P. Rogaway, “Cbc macs for arbitrary-length messages: The three-key constructions,” in *Annual International Cryptology Conference*. Springer, 2000, pp. 197–215. [Cited on page 8.]
- [30] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, “Umac: Fast and secure message authentication,” in *Annual international cryptology conference*. Springer, 1999, pp. 216–233. [Cited on page 8.]
- [31] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 5th ed. Pearson Education, 2010, discussion of the User Datagram Protocol. [Cited on pages 8 and 9.]

- [32] A. D. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011. [Cited on pages 9, 16, and 17.]
- [33] J. Postel, “Transmission control protocol,” RFC 793, September 1981. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc793.html> [Cited on page 9.]
- [34] L. Lamport, “Time, clocks and the ordering of events in a distributed system,” *Communications of the ACM* 21, 7 (July 1978), 558–565. Reprinted in several collections, including *Distributed Computing: Concepts and Implementations*, McEntire et al., ed. IEEE Press, 1984., pp. 558–565, July 1978, 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/> [Cited on page 9.]
- [35] —, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pp. 51–58, 2001. [Cited on page 9.]
- [36] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319. [Cited on page 9.]
- [37] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using dependency models to manage complex software architecture,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 167–176. [Online]. Available: <https://doi.org/10.1145/1094811.1094824> [Cited on page 10.]
- [38] R. Elmasri, *Fundamentals of database systems*. Pearson Education India, 2008.
- [39] I. G. Smit, J. Zhou, R. Reijnen, Y. Wu, J. Chen, C. Zhang, Z. Bukhsh, Y. Zhang, and W. Nuijten, “Graph neural networks for job shop scheduling problems: A survey,” *Computers & Operations Research*, vol. 176, p. 106914, 2025. [Cited on page 10.]
- [40] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Cited on page 10.]

- [41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing,” in *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, 2012, pp. 15–28. [Cited on pages 10 and 11.]
- [42] A. B. Kahn, “Topological sorting of large networks,” *Commun. ACM*, vol. 5, no. 11, p. 558–562, Nov. 1962. [Online]. Available: <https://doi.org/10.1145/368996.369025> [Cited on pages 11 and 39.]
- [43] G. Becker, “Merkle signature schemes, merkle trees and their cryptanalysis,” *Ruhr-University Bochum, Tech. Rep.*, vol. 12, p. 19, 2008. [Cited on page 13.]
- [44] A. Flangas, A. Cuellar, M. Reyes, and F. C. Harris Jr, “Parallelized c implementation of a merkle tree,” in *ITNG 2021 18th International Conference on Information Technology-New Generations*. Springer, 2021, pp. 107–114. [Cited on page 13.]
- [45] B. Cohen. (2017) The bittorrent protocol specification v2. BitTorrent Enhancement Proposal (BEP) 52. [Online]. Available: https://www.bittorrent.org/beps/bep_0052.html [Cited on page 14.]
- [46] Apache Software Foundation, “Apache Cassandra,” 2025, accessed: 2025-09-16. [Online]. Available: <https://github.com/apache/cassandra> [Cited on page 14.]
- [47] Amazon Web Services. (2025) Amazon DynamoDB. Amazon. [Online]. Available: <https://aws.amazon.com/dynamodb/> [Cited on page 14.]
- [48] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013. [Online]. Available: <https://doi.org/10.1145/2491245> [Cited on page 16.]
- [49] E. A. Brewer, “Towards robust distributed systems,” in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343–477. [Cited on page 17.]
- [50] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, “Replicated data types: specification, verification, optimality,” *SIGPLAN Not.*, vol. 49, no. 1, p. 271–284,

- Jan. 2014. [Online]. Available: <https://doi.org/10.1145/2578855.2535848> [Cited on page 17.]
- [51] P. S. Almeida, “Approaches to conflict-free replicated data types,” *ACM Computing Surveys*, vol. 57, no. 2, pp. 1–36, 2024. [Cited on page 18.]
- [52] Redis Documentation, “Under the hood: Redis crdts,” <https://redis.io/resources/under-the-hood/>, 202x, explains CRDT-based architecture used in Redis Enterprise for geo-distributed Active-Active replication. [Cited on page 18.]
- [53] Redis Labs, “Diving into conflict-free replicated data types (crdts),” <https://redis.io/blog/diving-into-crdts/>, 2022, blog post detailing CRDT implementations in Redis Enterprise. [Cited on page 18.]
- [54] Microsoft Azure Blog, “Conflict-free replicated data types (crdts) in azure cosmos db,” <https://azure.microsoft.com/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/>, 2020, describes built-in CRDT support in Cosmos DB’s multi-master system. [Cited on page 18.]
- [55] Microsoft Learn, “Conflict resolution policies in azure cosmos db,” <https://learn.microsoft.com/azure/cosmos-db/conflict-resolution-policies>, 2024, describes conflict resolution, including CRDT-based mechanisms, in Cosmos DB. [Cited on page 18.]
- [56] Yjs Contributors, “Yjs: A high-performance, modular crdt framework for real-time collaboration,” <https://github.com/yjs/yjs>, 2025, supports shared types, offline editing, rich text, and P2P sync. [Cited on page 18.]
- [57] M. Shapiro, “A comprehensive study of convergent and commutative replicated data types,” 2011. [Cited on pages 19 and 20.]
- [58] P. Helland, “Idempotence is not a medical condition: An essential property for reliable systems,” *Queue*, vol. 10, no. 4, pp. 30–46, 2012. [Cited on page 20.]
- [59] P. S. Almeida, A. Shoker, and C. Baquero, “Efficient state-based crdts by delta-mutation,” in *International Conference on Networked Systems*. Springer, 2015, pp. 62–76. [Cited on pages 20 and 21.]
- [60] A. van der Linde, J. a. Leitão, and N. Preguiça, “ Δ -crdts: making -crdts delta-based,” in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, ser. PaPoC ’16. New York,

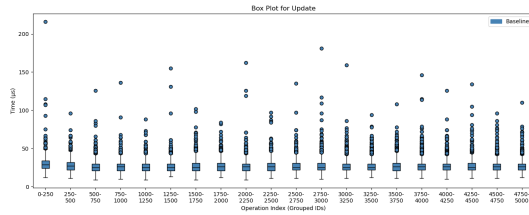
- NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2911151.2911163> [Cited on page 20.]
- [61] A. Rinberg, T. Solomon, R. Shlomo, G. Khazma, G. Lushi, I. Keidar, and P. Ta-Shma, “Dson implementation,” <https://github.com/crdt-ibm-research/json-delta-crdt/tree/3967a7998016f73c3866b68314fe4eb20991f689><https://www.nasa.gov/nh/pluto-the-other-red-planet>, 2021, accessed: 29-07-2025. [Cited on page 21.]
- [62] —, “Dson: Json crdt using delta-mutations for document stores,” *Proceedings of the VLDB Endowment*, vol. 15, no. 5, pp. 1053–1065, 2022. [Cited on pages 21 and 33.]
- [63] Ditto, “An inside look at ditto’s delta state crdts,” <https://www.ditto.com/blog/an-inside-look-at-dittos-delta-state-crdts>, 2023. [Cited on page 22.]
- [64] A. Chiesa and E. Tromer, “Proof-carrying data and hearsay arguments from signature cards.” in *ICS*, vol. 10, 2010, pp. 310–331. [Cited on pages 22 and 23.]
- [65] A. Garoffolo, D. Kaidalov, and R. Oliynykov, “Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains,” 2020. [Online]. Available: <https://arxiv.org/abs/2002.01847> [Cited on page 23.]
- [66] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner, “Proof-carrying data without succinct arguments,” in *Annual International Cryptology Conference*. Springer, 2021, pp. 681–710. [Cited on page 23.]
- [67] A. Kothapalli, S. Setty, and I. Tzialla, “Nova: Recursive zero-knowledge arguments from folding schemes,” in *Annual International Cryptology Conference*. Springer, 2022, pp. 359–388. [Cited on page 24.]
- [68] RISC Zero Team, “Recursive proving,” <https://dev.risczero.com/api/recursion>, 2025, accessed: 2025-09-22. [Cited on page 24.]
- [69] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990. [Cited on page 25.]
- [70] D. Mazieres and D. Shasha, “Building secure file systems out of byzantine storage,” in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, 2002, pp. 108–117. [Cited on page 25.]

- [71] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022. [Cited on page 37.]
- [72] J. Azevedo, “Authenticated crdt implementation,” <https://github.com/Azeved00/> Thesis, 2025, accessed: 27-09-2025. [Cited on page 50.]
- [73] G. Tomás, P. Zeller, V. Balesgas, D. Akkoorath, A. Bieniusa, J. Leitão, and N. Preguiça, “Fmke: A real-world benchmark for key-value data stores,” in *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, 2017, pp. 1–4. [Cited on page 55.]
- [74] M. N. and, “Power laws, pareto distributions and zipf’s law,” *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005. [Online]. Available: <https://doi.org/10.1080/00107510500052444> [Cited on page 55.]

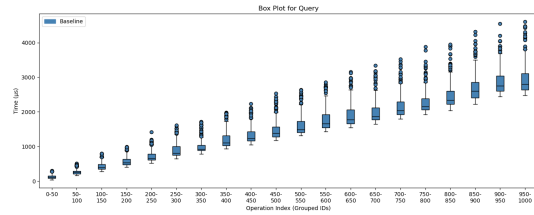
A. Full Graphs

For completeness, this appendix presents all benchmark graphs, including the removed outliers that were omitted for readability. Each figure illustrates the same measurements as discussed in Chapter 6.

A.1. Micro Benchmarks

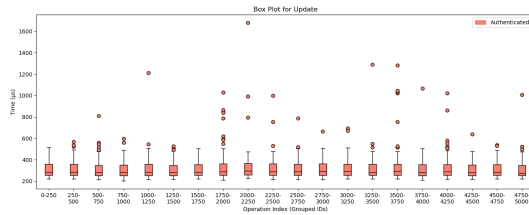


(A) Box plot of update execution times.

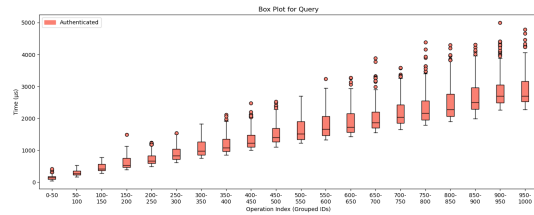


(B) Box plot of query execution times.

FIGURE A.1: Micro benchmarks results of the **baseline** implementation.

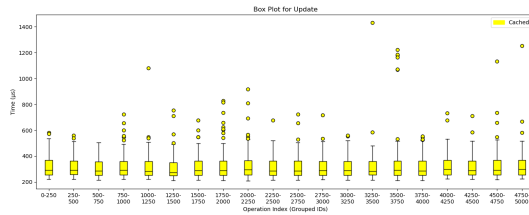


(A) Box plot of update execution times.

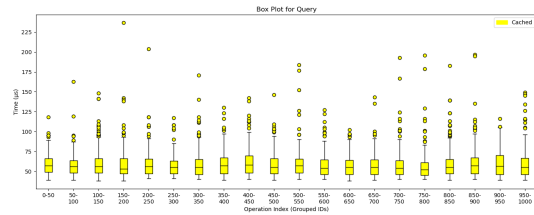


(B) Box plot of query execution times.

FIGURE A.2: Micro benchmark results for **authenticated** implementation.



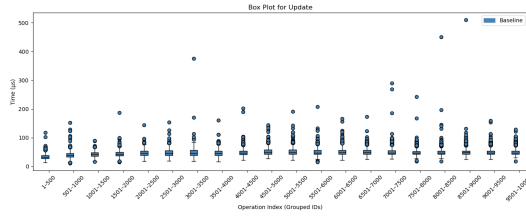
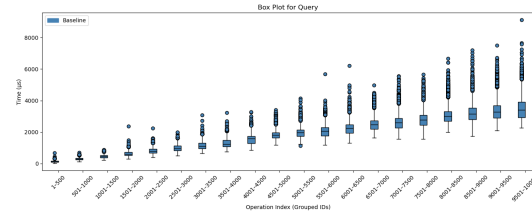
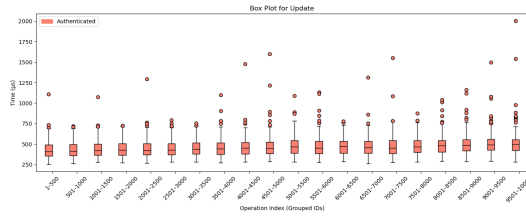
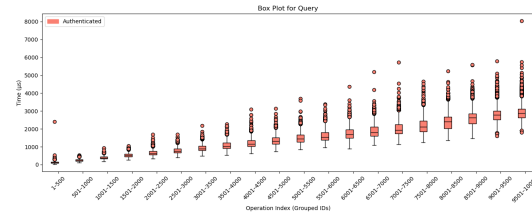
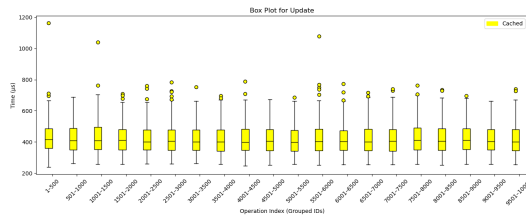
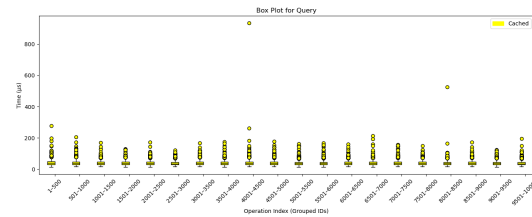
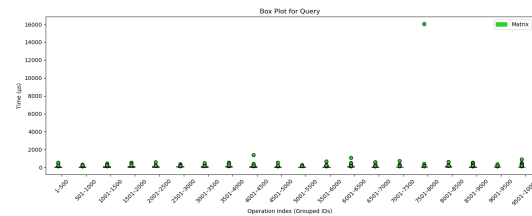
(A) Box plot of update execution times.



(B) Box plot of query execution times.

FIGURE A.3: Micro benchmarks results of the **cached** implementation.

A.2. Macro Benchmarks

(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE A.4: Macro benchmark results of the **baseline** implementation.(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE A.5: Macro benchmark results of the **authenticated** implementation.(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE A.6: Macro benchmark results of the **cached** implementation.(A) Box plot of **update** execution times.(B) Box plot of **query** execution times.FIGURE A.7: Macro benchmark results of the **matrix** implementation.