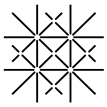


University of Basel

Delta-GOC-Ledger: Incremental Checkpointing and Lower Message Sizes for Grow-Only Counters Ledgers with Delta-CRDTs

Master's thesis

University of Basel – Faculty of Science
Department of Mathematics and Computer Science
Computer Networks Group
https://cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Jannick Heisch
jannick.heisch@unibas.ch

17.04.2024

Acknowledgments

I would like to express my gratitude to Prof. Dr. Christian Tschudin and Dr. Erick Lavoie for giving me the possibility to work on this thesis and for the dedicated support and guidance during this time.

I would also thank Dr. Erick Lavoie, whose expertise in this research field and his mentorship supported me throughout my thesis journey. I am particular thankful for the detailed feedback on earlier drafts of this thesis, as well as for the constructive discussions, which created many new ideas that influenced this work and motivated me.

Finally, I would like to thank Muriel Heisch for her feedback on the final draft of this thesis, especially regarding grammar and spelling.

Abstract

The Grow-Only-Counters-Ledger (GOC-Ledger) is a novel consensus-free replicated ledger based on a state-based Conflict-free Replicated Data Type (CRDT). It enables local crypto-tokens that impose lower infrastructure costs for low-volume intra-community transactions than existing blockchain projects. CRDTs ensure that all replicas in the system eventually end up in a consistent state. In order to achieve eventual consistency in state-based CRDTs, the replicas must periodically exchange their entire state. In the GOC-Ledger, the size of full states is unbounded and can become large depending on the number of transactions and accounts. Operation-based CRDTs only transfer operations in their update messages, leading to smaller message sizes, but require a reliable communication channel. Therefore, δ-based CRDTs have been proposed that unite the advantages of both approaches, by relying on the replication of small-states, while guaranteeing correct convergence on unreliable communication channels.

This thesis presents the δ-GOC-Ledger, a δ-based version of the existing state-based GOC-Ledger to reduce the communication overhead and increase scalability while still supporting unreliable communication channels. We prove the correctness and convergence of our design and formally show the relation to the state-based approach. A prototype based on Git is presented and demonstrates the implementation of CRDTs with Git. We assess both state-based and δ-based versions of the GOC-Ledger by simulating ERC-20 token transactions to quantify the reduction in message size that can be expected on real-world transactions and to evaluate the suitability of Git as a platform for implementing CRDTs.

We provide formal convergence proofs highlighting that the current state of the δ-based ledger can be incrementally computed. This is achieved by exploiting the associativity of the merge operation, which allows us to merge the latest full state with more recent delta states in any order. This approach enables efficient incremental computation of states from a causal history, as represented in a Git commit graph. Our analysis reveals that Git is quite effective at optimizing the size of state-based CRDT messages by already achieving an average size reduction of 24% compared to a naive approach. Nonetheless, implementing an application-specific δ-CRDT reduces the message size during incremental reconciliation by an additional 10-30%. Based on our results, we identify the overhead associated with the data representation using Git tree objects and highlight additional possibilities to further optimize the communication overhead.

Table of Contents

Acknowledgments ii

Abstract iii

List of Figures vi

1 Introduction 1
- 1.1 Contribution . . . 2
- 1.2 Outline . . . 2

2 Background 3
- 2.1 Grow-Only-Counters-Ledger . . . 3
- 2.2 Conflict-Free Replicated Data Types . . . 4
  - 2.2.1 State-based CRDTs . . . 4
  - 2.2.2 Operation-based CRDTs . . . 5
  - 2.2.3 Delta-State Replicated Data Type . . . 6
- 2.3 Happens-Before Relationship . . . 7
- 2.4 Append-only Log . . . 8
- 2.5 Git . . . 8

3 Delta-Grow-Only-Counters Ledger 10
- 3.1 System Model . . . 10
- 3.2 Delta Account . . . 11
  - 3.2.1 Partial Order . . . 12
  - 3.2.2 Delta-Mutators . . . 13
  - 3.2.3 Query Functions . . . 15
  - 3.2.4 Merging States . . . 16
  - 3.2.5 Account History . . . 17
- 3.3 Ledger . . . 19
- 3.4 Liveliness, Safety and Balance Properties . . . 21

4 Proofs 22
- 4.1 Definitions . . . 22
- 4.2 Accounts . . . 22
  - 4.2.1 Delta-mutators have the same effect as the state-based operators . . . 22

4.2.2 Account delta-mutators return minimal delta states . . . 24
4.2.3 Equality of state-based and delta-based account states . . . 26
4.3 Ledger . . . 28
4.3.1 Delta-mutators have the same effect as the state-based mutators . . . 28
4.3.2 Equality of state-based and delta-based ledger states . . . 29

5 Implementation 30
5.1 System Assumptions . . . 30
5.2 Commits as Delta States . . . 31
5.3 References as Frontier . . . 34
5.4 Replication . . . 35
5.5 Checkpointing . . . 36
5.6 State-based GOC-Ledger . . . 38

6 Evaluation 42
6.1 Methodology . . . 42
6.1.1 ERC-20 Transactions Dataset . . . 42
6.1.2 Simulation of Transactions . . . 43
6.1.3 Measurements . . . 44
6.2 Results . . . 45
6.2.1 Total Space for Storage and Full Replication . . . 45
6.2.1.1 Discussion . . . 48
6.2.2 Size for Incremental Updates . . . 50
6.2.2.1 Discussion . . . 50
6.2.3 Effectiveness of Git Default Compression and object-based architecture 51
6.2.3.1 Discussion . . . 53

7 Related Work 56

8 Conclusion 59
8.1 Future Work . . . 59
8.1.1 Git performance optimization . . . 60
8.1.2 Adversarial Environments . . . 60

Bibliography 61

Appendix A Notation 64

List of Figures

2.1 State-based replication of a grow-only counters dictionary between two replicas. . 5
2.2 Operation-based replication of a grow-only counters dictionary between two replicas. 6
2.3 δ-based replication of a grow-only counters dictionary between two replicas. . . . 7

3.1 Example of an account history depicted as a directed acyclic graph. 18

5.1 Example of a commit history for one token type. 33
5.2 Directory of Git references used for the implementation. 34
5.3 State representation in Git for state-based and delta-based GOC-Ledger. 39
5.4 Modified trees and blobs (in red) after a state update for the state-based GOC-Ledger. 41

6.1 Bundle file size. 46
6.2 Total size of different git objects. 46
6.3 Overview of the number of Git objects and deltas in the bundle file 47
6.4 Size of the incremental bundle file (containing 200 transactions each). 50
6.5 Size reduction of the delta-based approach compared to the state-based implementation (higher is better). 50
6.6 Repository size with uncompressed objects. 52
6.7 Compression factor of Git bundle file. 52
6.8 Size of blob and tree objects compared to a naive approach. 53

1 Introduction

Besides conventional transaction methods, blockchain technology gained popularity for performing secure global transactions between untrusted parties. Major blockchain projects such as Bitcoin and Ethereum rely on consensus algorithms that totally order operations among all replicas of the blockchain. This causes considerable costs, whether in terms of energy consumption in the case of Bitcoin's proof-of-work approach [27] or substantial capital investment in the case of Ethereum's proof-of-stake model [19].

For small communities, the costs of the required infrastructure of traditional blockchain projects for low-volume intra-community transactions are often prohibitive. Therefore, local crypto-tokens [19] have been proposed, which rely on the trust between the participants of a local economy. Unlike traditional blockchain projects that prevent double-spending, these tokens are based on the detection of double-spending, enabling cheaper and faster transactions for smaller ecosystems.

One token design that enables local crypto tokens is the Grow-Only-Counters-Ledger (GOC-Ledger) [18]. It is a consensus-free replicated ledger built upon state-based Conflict-free Replicated Data Types (CRDT) [28]. The GOC-Ledger represents different account state attributes (number of created, burned, transferred, and received tokens) as grow-only counters that can be combined to calculate the account balance. CRDTs ensure eventual consistency across all replicas without the need for costly consensus mechanisms between replicas. As a result, latency and costs are significantly reduced. This approach enables the transfer of local crypto tokens, especially for smaller communities that cannot afford the high transaction costs required in other financial systems. However, replicas have to exchange their entire state with other replicas periodically to achieve eventual convergence. The state size of the GOC-Ledger increases with the number of participants and transactions which results in communication overhead and limited scalability. In order to optimize the message sizes transferred between replicas, Delta-State Replicated Data Types (δ-CRDT) can be employed. These CRDTs only exchange the modified parts of the state, known as "deltas", between replicas instead of replicating the entire state. This approach significantly reduces the communication overhead [34].

In this thesis, the communication overhead of the existing GOC-Ledger design is optimized by introducing the δ-GOC-Ledger, a delta decomposition of the state-based ledger.

By only transferring delta states which are much smaller than the full account states, the size of messages replicated between peers is reduced. The correctness and convergence of the δ-GOC-Ledger are proven, as well as an implementation based on Git is presented and evaluated.

1.1 Contribution

To the best of our knowledge, this thesis introduces the δ-GOC-Ledger, the first consensus-free replicated ledger based on δ-CRDTs. We contributed a strong foundation for further optimizations by first formally designing the δ-based GOC-Ledger, then providing a prototype using Git, and finally conducting evaluations to show the estimated message size reduction with real transactions and to investigate how Git's default optimizations are influencing this result. We derive a δ-based design of the existing state-based GOC-Ledger [18] building upon the methods proposed by Almeida et al. [2] and prove that our design converges correctly and generates minimal delta states. We show how the account history can be represented as a directed acyclic graph (DAG). With this DAG, the relationship between the state-based and the δ-based approaches is demonstrated. This allows us to prove that both approaches converge to the same full state and we can adopt the already proven properties of the existing GOC-Ledger. Furthermore, the DAG reveals that not all delta states have to be merged in order to obtain the latest full state, but that it is sufficient to merge the last computed full state, which we call checkpoint, with all subsequent delta states.

We present a prototype that demonstrates how CRDTs can be implemented with Git and how observations from the design phase like checkpoints, i.e. incremental updates, and account history represented as a DAG, influence the implementation. In addition, we have developed a simulation application that translates real ERC-20 token transactions into GOC-Ledger token operations. This allows us to evaluate the implementations and compare the message sizes generated by the δ-based approach and state-based versions. The evaluation shows that both variants of the GOC-Ledger benefit from the compression techniques and object-based structure of Git. We also identified that Git implements a similar optimization as δ-based CRDTs for state-based CRDTs by only sending new objects while already existing objects are reused. However, implementing our δ-CRDT design with Git reduces the message sizes during replication by an additional 10-30%, because it generates smaller tree objects. We have identified the overhead of our implementation due to the data representation with Git tree objects and suggested additional ways to further reduce message sizes.

1.2 Outline

In the rest of this thesis, we first introduce important methods and concepts (Chapter 2), then present the δ-GOC-Ledger (Chapter 3), proofs of the claimed properties of our design (Chapter 4), an implementation based on Git (Chapter 5), an evaluation of our prototype (Chapter 6), a review of related work (Chapter 7), and finally we summarize the thesis and give an overview of future work (Chapter 8).

2
Background

This chapter introduces the necessary background concepts for this thesis, which are important for understanding the δ-GOC-Ledger design.

2.1 Grow-Only-Counters-Ledger

Banking systems are mainly used for financial transactions where ledgers are typically centralized, meaning that financial institutions maintain and manage all the transaction records and account balances. In contrast, decentralized networks rely on ledgers that are replicated among multiple participants. The most prevalent approaches involve consensus-based replicated ledgers based on blockchain technology, as seen in Bitcoin and Ethereum. These systems employ a consensus algorithm to order updates globally, preventing double-spending. However, these algorithms incur significant overhead and high costs [19].

To address this issue, consensus-free replicated ledgers have been introduced. This concept is based on the observation that the total ordering of operations within the blockchain is stronger than necessary and that sequential ordering of operations per account is sufficient to prevent double-spending [11]. Since this approach does not require consensus among replicas, it reduces latency and increases throughput of the system.

An innovative ledger based on the consensus-free approach is the Grow-Only-Counters-Ledger (GOC-Ledger), developed by Lavoie in 2023 [18]. This ledger implements local crypto tokens [19] specifically designed for low-value transactions within a community for which the existing blockchain infrastructure would be too expensive. The GOC-Ledger is a state-based CRDT combining multiple grow-only counters that are also state-based CRDTs. Various operations are defined for the creation, destruction, transfer, and acknowledgment of tokens, which modify the underlying account state. Each account state consists of a unique identifier (A_{id}), two grow-only counters that record the number of created ($A_{\uparrow}$) and destroyed ($A_{\downarrow}$) tokens, as well as two dictionaries of grow-only counters which track the number of tokens sent to ($A_{\rightarrow}$) and received from ($A_{\leftarrow}$) other accounts. The information stored in these grow-only counters can be combined to calculate the balance for each respective account by subtracting the number of burned and sent tokens from the number of created and received tokens, i.e. $balance(A) = A_{\uparrow} + A_{\leftarrow} - A_{\downarrow} - A_{\rightarrow}$. In comparison to other consensus-

free replicated ledgers, the GOC-Ledger represents a unique approach since the state-based CRDTs converge even on unreliable communication channels, without imposing specific requirements on the communication channel.

GOC-Ledger can only prevent overspending when all operations to the same account are sequential. In real peer-to-peer scenarios, adversarial replicas may disregard this requirement, which potentially leads to double-spending. Detecting such adversarial behavior [17] and developing methods to handle confirmed cases of double spending are still subject of ongoing research.

2.2 Conflict-Free Replicated Data Types

A fundamental issue in distributed systems is achieving consensus on data values among multiple replicas, because the concurrent modification of a value without coordination can easily lead to inconsistency among the replicas.

One way to achieve consistent states across all replicas is to enforce strong consistency [16]. A trivial strategy is to serialize the updates in a global total order, for example by implementing a central lock that a replica must acquire in order to modify a value. This ensures that only the replica with the lock can make modifications to a value, while others must wait until the lock is released again. However, this approach will limit the availability and latency as described by the CAP-Theorem [8].

Conflict-free Replicated Data Types (CRDTs) offer an alternative approach. This data structure allows concurrent modifications by multiple participants without the need for coordination. CRDTs guarantee eventual consistency, i.e. replicas in a distributed system might temporarily have different states, but they will eventually converge to a consistent state over time, even if network delays or partitions occur[28]. There are multiple forms of CRDTs described in the following.

2.2.1 State-based CRDTs

State-based CRDTs allow each replica to update its local state independently and concurrently with other participants. These replicas periodically exchange their entire local states with each other. A deterministic merging function is then used to combine the received states with their own, ensuring all replicas eventually reach a consistent state.

Figure 2.1 illustrates an example of a state-based dictionary of grow-only counters with two replicas. The arrows depict the local causal history and the nodes represent states. For each state, the applied operation leading to this state is shown above the node, while the resulting local full state is shown below. The *inc* operation increments the counter value of the specified key by the desired amount. In this example, the dictionary contains two keys A, and B, which are incremented by the respective replica. This CRDT's merge function ($\sqcup$) results in a state with the highest counter value for each key of both input states. Dotted arrows represent transferred update messages containing the information shown in the dotted box.

Initially, both replicas have the same state, a dictionary with keys for replicas A and

B and corresponding counter values. Then, both independently apply the *inc* operation on their local state to increment their counter, resulting in a new local state S_A^1 at replica A and S_B^1 at replica B. Afterward, both replicas exchange their full local state, which is then subsequently merged. We see that even if the states of the replicas were temporarily inconsistent, replicating and merging the full state of the other participant ensures that both reach a consistent state. Since the merge function computes the maximum counter values of both input states, the order and duplication of update messages do not affect the result.

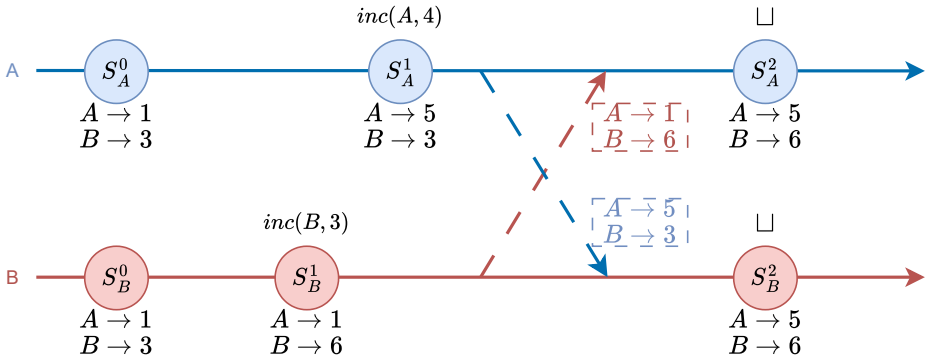


Figure 2.1: State-based replication of a grow-only counters dictionary between two replicas.

In general, a state-based object is a state-based CRDT if it satisfies all requirements of a monotonic join semi-lattice [28]. First, all possible states must be ordered by a partial order. Second, the merge function must compute the Least Upper Bound between two states within this partial order, ensuring that the merge operation is associative, commutative, and idempotent. Third, all defined mutators must be monotonically non-decreasing, i.e. they must result in a state that is equal to or larger than the previous state within the same partial order. Taken together, these three properties ensure that all replicas will always converge to a consistent state.

The advantage of these characteristics is that state-based CRDTs do not impose specific demands on the communication channel. The system still eventually converges to a consistent state, even over unreliable channels where messages may be lost, duplicated, or arrive out of order.

However, a major drawback of the state-based approach is that the entire state must be exchanged between replicas, resulting in the transmission of large messages. Depending on the application, this can lead to significant communication overhead, thereby potentially incurring high latency and limiting scalability.

2.2.2 Operation-based CRDTs

Instead of transmitting the entire local state as in the state-based approach, operation-based CRDTs transmit only the corresponding update operations with other replicas after modifying their local state. Operation-based CRDTs do not require a merge function, but rely on a causally ordered broadcast communication protocol for the exchange of update

operations. This protocol must guarantee that updates are delivered in a causally ordered sequence and need to avoid message loss or duplication to ensure that each update operation is delivered in-order and exactly once.

Operation-based CRDTs are characterized by a lower communication overhead, since only small update operations and not entire states have to be exchanged. However, this approach imposes stronger requirements on the underlying communication channel compared to state-based CRDTs.

Figure 2.2 showcases the same scenario as in Figure 2.1, but using an operation-based dictionary of grow-only counters. Since the same operations are applied, the resulting local states are equivalent to the state-based version. The key difference lies in the update messages. Instead of replicating the entire local state, only the executed operations are transmitted, resulting in smaller messages. However, if for example, the update message of replica B is accidentally duplicated, the counter value of key B at replica A, would be incremented twice resulting in a value of 9. Therefore, the underlying communication channel must implement additional mechanisms to prevent such scenarios.

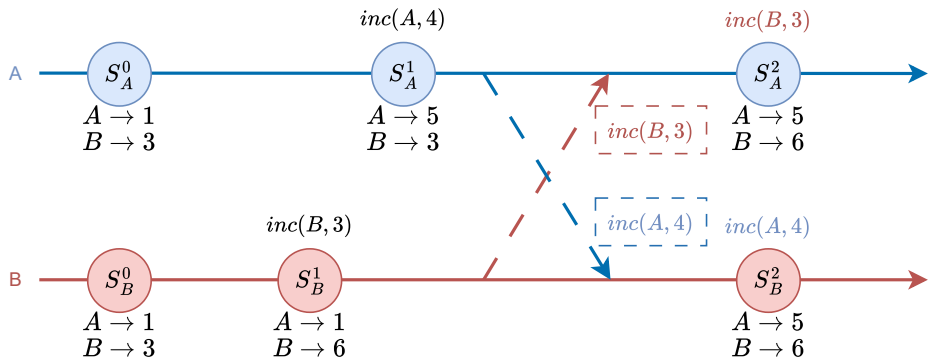


Figure 2.2: Operation-based replication of a grow-only counters dictionary between two replicas.

2.2.3 Delta-State Replicated Data Type

Ensuring the specific requirements of operation-based CRDTs for the communication channel in distributed systems can be complex and may not be feasible depending on the application and infrastructure. To address this, Delta-State Replicated Data Type (δ-CRDT) has been proposed, which is a state-based CRDT approach that combines the advantages of both state-based and operation-based CRDTs [2]. Similar to the operation-based approach, delta-based CRDTs enable the transmission of small messages between replicas and, as with the state-based approach, they do not place any complex requirements on the communication channel. This is achieved by exchanging delta states, which contain the effect of recent update operations instead of replicating the complete state or individual update operations between replicas.

Similar to the state-based mutators (m), δ-CRDTs define delta-mutators (m^δ). These

delta-mutators can be applied on a state X, resulting in a delta state X^δ which only contains the effect of this mutator. By merging the delta with the full state, the effect encoded in the delta state is applied to the state i.e., $m(X) = m^\delta(X) \sqcup X$. Therefore, delta states are subsequently replicated, allowing other replicas to merge these deltas with their local state. Eventually, this will lead to a consistent state across all replicas. To achieve optimal message sizes, the delta-mutator should be minimal and not inflate the resulting delta states with redundant information, already contained in the full state. Therefore the minimal delta-mutator $m^{\delta\perp}$ yields the smallest possible delta state X^δ so that there exists no other delta-mutator that returns a smaller state while containing the same effect.

Delta states can be merged not only with full states but also with other delta states forming a delta group. This allows to bundle several small effects into one large effect. The delta group can potentially be smaller in size than the merged delta states if multiple updates modify the same part of the state, which further optimizes the message sizes replicated between replicas.

Figure 2.3 shows a δ-based version of the grow-only counters dictionary. The same example is illustrated as in the state-based (Figure 2.1) and operation-based versions (Figure 2.2). Here, the delta-mutator inc^δ is applied, which returns a delta state containing only the modified counter values. The resulting local state is the same as in the other approaches since the delta state encodes the same effect. However, during replication, only this delta state is transmitted instead of the entire state or the applied operations. Since the merge function of this approach also takes the maximum of each counter value of the input states, duplicated update messages do not change the resulting states.

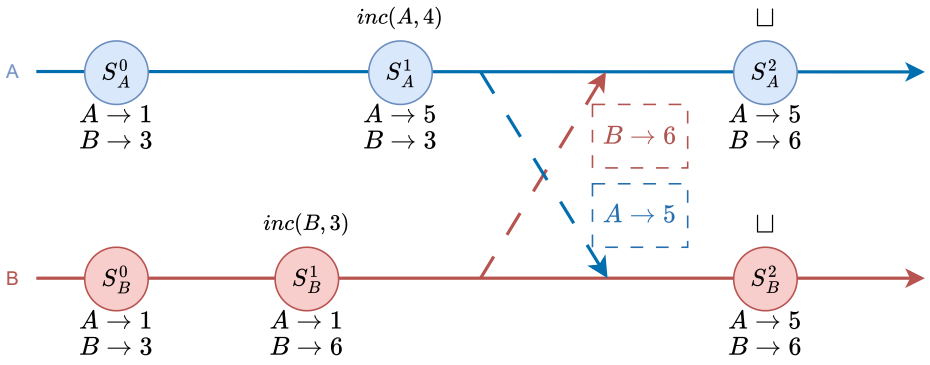


Figure 2.3: δ-based replication of a grow-only counters dictionary between two replicas.

2.3 Happens-Before Relationship

The happens-before relationship [16] is a fundamental concept for causal ordering events in distributed multiprocess systems. Informally, if an event a happens before event b, denoted as $a \to b$, it implies that a influences b directly or indirectly. The happens-before relationship fulfills the following properties:

- **Local total order:** If two events a and b occur in the same process and a is executed before b, then $a \rightarrow b$ must hold.
- **Synchronization order:** If a synchronization operation S_1 precedes another synchronization operation S_2, all events associated with S_1 happen before those associated with S_2.
- **Transitivity:** If event a happens before event b ($a \rightarrow b$), and event b happens before event c ($b \rightarrow c$) , then event a happens before event c ($a \rightarrow c$).

This way, the happens-before relationship establishes a partial order among events in a concurrent system. If neither $a \rightarrow b$ nor $b \rightarrow a$ can be established, the events are concurrent, written as $a \,||\, b$.

2.4 Append-only Log

Append-only logs are data structures that only permit the addition of new data exclusively to their end. Once the data is appended to such a log, it becomes immutable and cannot be altered or deleted. This characteristic facilitates replication and recovery by preserving a chronological record of messages while preventing retroactive modifications. This makes append-only logs useful for ensuring data consistency across replicated systems.

Due to these characteristics, append-only logs are often implemented in decentralized, replicated ledger applications. They are used in the form of blockchains in leading cryptocurrencies such as Bitcoin [22] and Ethereum [5]. In these cryptocurrencies, new transactions are verified through a consensus algorithm, then combined in blocks and finally appended to a replicated blockchain. Each new block contains a reference to the preceding block in the form of a cryptographic hash. This hash serves as a unique identifier for that block and also provides a way to verify the integrity of the data it contains. In addition, each block is digitally signed by the author's private key, enabling anyone to authenticate this signature with the corresponding public key. This cryptographically signed append-only log establishes self-certification [21]. As a result, transactions are irreversible and resistant to subsequent modifications and prevent other users from impersonating the author, ensuring secure and trusted transactions that are crucial to financial systems [23]. These properties are also useful for other decentralized applications that require the replication of messages between untrusted parties.

2.5 Git

Git [33] is a widely used distributed version control system. Unlike centralized systems, Git does not rely on a single server. Instead, it creates local copies (repositories) of the project on each user's machine. These repositories track changes over time by taking snapshots called commits, i.e. a snapshot of the current local file system. Furthermore, it keeps track of the history of commits, so that users can exchange their state with others and update their local state to the latest snapshot, merge changes, or undo changes by going back in the commit history. The most common platform implementing Git is GitHub, a high-availability replica

with additional access to control mechanisms and project management tools. In January 2023, GitHub reported that it has over 100 million developers and is one of the largest source code hosts, mainly used for the management of software development projects [7]. Due to its popularity, Git is available on almost all common platforms and is continuously updated and optimized.

In order to manage the version of files, Git operates on three different types of objects: blobs, trees, and commits [6]. The current files of the repository are stored in the local file system, the working directory, while the Git objects are stored in the ".git" folder. All objects in Git are identified by computing the SHA-1 hash of their content. When a file of the working directory is added to the repository, the current content of this file is stored as a blob, without any additional metadata such as its filename. If the file is modified and added again, a new blob will be created with the altered content, which also results in a new SHA-1 hash. Trees assign names to objects. They contain multiple entries, with each entry consisting of the SHA-1 reference to the object, the type of the object, and the assigned name. Trees can only contain blobs or other trees. This way, trees enable the assignment of blobs to the corresponding file name and the organization of files in a dictionary using nested trees. When creating a new snapshot of the local state, a commit object is created. It contains exactly one tree, the commit-tree, which refers to all committed files. Besides additional metadata, such as the name of the committer, timestamps, etc., the commit can contain hashes that refer to the parents of this commit. These references represent the happens-before relationship between this commit and its parent. This way, Git maintains the causal history in the form of a directed acyclic graph, also called the commit history. Since it is difficult to remember the hash of a commit, Git uses mutable references that map a name to the hash of a commit. Instead of traversing the whole commit history to find the latest version, the latest commit can be recovered from the corresponding reference. These references are also important for the synchronization process, in which replicas compare the referenced commits and identify the differences between the two repository states and subsequently only transmit the required objects. If the received changes cannot be applied without conflict, the affected files are highlighted and the user is prompted to resolve the issue with an additional merge commit.

The previously introduced Git objects are usually only used internally and abstracted away from the user by high-level APIs. However, by using lower-level APIs (plumbing commands), developers can design custom data structures on top of Git for purposes beyond file version control, while relying on the replication and reconciliation system of Git. This way, applications can be built on top of Git, such as the 2P-BFT-Log [17], which implements an append-only log by appending new commits to the commit history that points to the latest published message as a parent. By using the references as frontiers, i.e. pointers to the latest message of the log, this implementation can rely on the Git reconciliation protocol for replication, which compares the frontiers between replicas and then only replicates the difference.

3
Delta-Grow-Only-Counters Ledger

This chapter introduces the δ-GOC-Ledger, a delta-state decomposition of the state-based Grow-Only-Counters-Ledger [18] into a Delta-State Replicated Datatype [2]. In order for all replicas to eventually converge to a consistent state, CRDTs require that all possible states form a monotonic join semi-lattice, written S, to guarantee convergence [28]. For this, three conditions must hold: first, all possible states in S must be ordered by a partial order $\leq$. Second, all mutators i.e., all operations that modify the state, must be monotonic, ensuring that they result in a state that is equal or larger than the current one. Third, a merge function is defined that computes the Least Upper Bound between two states in S, which results in a state that is larger or equal to the two states. These conditions as well as the correctness of the ledger shape the design of the δ-GOC-Ledger.

The considered system model is described in the following section. Then, the delta-state replicated accounts and ledger with the corresponding delta-mutators, query functions, and merge operations are presented, highlighting similarities and differences to the existing state-based design. Finally, the properties of the delta-state ledger are discussed.

3.1 System Model

The considered distributed system consists of a dynamic set of replicas, each with its own local memory. There exists no global memory shared between the replicas, but the replicas can send update messages to each other. The network is not reliable, e.g. the update messages can be lost, re-ordered, or duplicated. However, it is assumed that an update message sent by a node will eventually be replicated across all replicas of the system after a finite amount of time. In addition, each node records its state on persistent storage, ensuring that in the event of a crash, the replica is able to eventually recover and resume the transmission of its latest updates. This guarantees that the updates performed by the replicas never get lost in the system.

The presented delta-state CRDT, like the state-based GOC-Ledger, is not compatible with adversarial replicas (see Section 3.4).

3.2 Delta Account

Similar to the original state-based GOC-Ledger, each account state A is composed of multiple grow-only counters: $A_{\uparrow}$ counts the number of tokens created. $A_{\downarrow}$ tracks how many tokens were burned. Furthermore, there are two dictionaries of grow-only counters. The dictionary of given tokens ($A_{\rightarrow}$) monitors the number of tokens transferred to other accounts. In the GOC-Ledger, tokens must be explicitly acknowledged in order to increase the balance of the receiver's account. Dividing the transfer of tokens into a sending and confirmation event allows other replicas to verify whether the recipient account has received the tokens and gives the recipient the option to ignore tokens. The amount of acknowledged tokens is stored in an additional dictionary ($A_{\leftarrow}$). By combining these counters, it is possible to compute the balance of this account. Due to the eventual consistency of the state-based ledger, all replicas that have received the same set of updates will compute the same balance for this account.

In contrast to the state-based approach, the δ-GOC-Ledger does not transmit the full account states. Only the effects of operations are sent to other replicas in the form of delta states (A^{δ}). Delta-accounts are similar to full accounts, but they only store a delta between full states and therefore they do not need to store all four account attributes, but only those that have been changed.

The notation used in the following sections is defined in Appendix A. In order to maintain the δ-GOC-Ledger design general and avoid being bound to a specific implementation, all missing fields are internally represented as zero or an empty set, but are not sent to other replicas. When a missing field of a delta state is accessed, the default value for this field as defined in Table 3.1 is returned. Since the merge function takes the larger value of both operands for each attribute, the default values are chosen so that they correspond to the smallest possible values for each field. This means that default values in a state do not affect the resulting account state if they are merged with another state. This approach avoids using other language-specific implementations such as *null* to represent missing fields, since depending on the language *null* may still induce storage overhead, which would make the delta accounts inefficient. It is assumed that missing fields of delta accounts do not require any memory space. Mathematically this can be expressed as a partial function that takes a delta account and is defined for all valid values of a given field. In the case of a valid value, the function returns this value. Otherwise, if the field is missing or contains invalid values, zero (for $A_{\uparrow}$ and $A_{\downarrow}$) or the empty set (for $A_{\leftarrow}$ and $A_{\rightarrow}$) is returned.

Account Field	**Default Value**
$A^{\delta}_{\uparrow}$	0
$A^{\delta}_{\downarrow}$	0
$A^{\delta}_{\rightarrow}$	$\emptyset$
$A^{\delta}_{\leftarrow}$	$\emptyset$

Table 3.1: Default values for missing account fields.

To initialize a delta account state with a given identifier, the $initialize^{\delta}_{\mathbb{A}}$ (Alg. 1) function is used. This method returns an account state with the designated identifier A_{id}, while all

other attributes are missing. The resulting delta-account is also called the default state A^0_{id} because, besides the account identifier, all other fields are missing and therefore correspond to the default values (Table 3.1). For this reason, A^0_{id} is the smallest of all possible account states. When merging the default state with any other state A', the resulting state is always A', since A' is greater than or equal to A^0_{id} ($A' \geq_{\mathbb{A}} A^0_{id}$). As all delta account states are instantiated with this method or are merged with a state created by it, every delta state always contains at least the identifier attribute A_{id}. Without this attribute, the state could not be uniquely associated with an owner, so that account balances from different owners are incorrectly merged.

Algorithm 1 Initialization of a delta account state

1: **function** INITIALIZE$^{\delta}_{\mathbb{A}}(id)$
2: $A^{\delta}_{id} \leftarrow id$ ▷ The other attributes are missing (do not take any memory space) and are therefore assigned to their default value (see Table 3.1).
3: **return** A^{δ}

3.2.1 Partial Order

To achieve convergence, one requirement is that all possible states of the CRDT can be organized in a semi-lattice S ordered by a partial order $\leq$. Because all missing fields in delta accounts are internally represented as their default value (see Table 3.1), all delta accounts can be represented as a full account. Therefore, they can be partially ordered by $\leq_{\mathbb{A}}$ (Algorithm 2) as in the state-based GOC-Ledger, which is already proven to be a valid partial order for full account states [18]. This partial order defines that an account state A is less or equal to another state A', if all identifiers of the $ackFrom$ and $giveTo$ dictionaries are a subset of those in A' and if all grow-only counters included in the state are less or equal the counters in state A'.

It should be noted that only a few fields are usually set for delta accounts, while many others are missing and therefore represented as zero or an empty set. When comparing a delta account A^{δ} resulting from delta-mutators with full accounts A, it often occurs that A^{δ} has only one counter that is larger than the corresponding counter in A, while all other counters are larger in A. Therefore, the $\leq_{\mathbb{A}}$ relation cannot be determined, which may lead to a concurrency between both states $A \,||\, A^{\delta}$ since neither is larger than the other.

Algorithm 2 Partial order between accounts and delta accounts as defined in the state-based GOC-Ledger [18]

1: **function** $\leq_{\mathbb{A}}(A, A')$ ▷ $A \leq_{\mathbb{A}} A'$, A and A' can be a full or a delta account state
2: $created \leftarrow A_{\uparrow} \leq A'_{\uparrow}$
3: $burned \leftarrow A_{\downarrow} \leq A'_{\downarrow}$
4: $given \leftarrow A_{\rightarrow *} \subseteq A'_{\rightarrow *} \wedge \bigwedge_{id \in A_{\rightarrow *}} A_{\rightarrow}[id] \leq A'_{\rightarrow}[id]$
5: $acked \leftarrow A_{\leftarrow *} \subseteq A'_{\leftarrow *} \wedge \bigwedge_{id \in A_{\leftarrow *}} A_{\leftarrow}[id] \leq A'_{\leftarrow}[id]$
6: **return** $A_{id} = A'_{id} \wedge created \wedge burned \wedge given \wedge acked$

3.2.2 Delta-Mutators

The original state-based GOC-Ledger introduces several operations that are performed on an account state, written A, and result in a modified account state A'. In the δ-based approach, these operations are redesigned so that instead of a full state, a delta state, written A^δ, is returned. This delta state includes only the necessary attributes required to achieve the equivalent account state A' when merged with the current state A, without containing any additional redundant information. For example, when tokens are created for an account using the state-based method, the resulting state contains all attributes, including those that are not affected by the operation. In contrast, the delta-mutator yields a state that only contains the created token field ($A^\delta_\uparrow$), while the other fields are omitted.

In the following, the delta decomposition proposed by Almeida et al. [2] is used to transform the existing state-based account mutators into delta-mutators. For each account operation op, a corresponding delta-mutator op^δ is designed such that $op(A) = A \sqcup op^\delta(A)$. This ensures that applying a delta-mutator to the account state and then merging it with the current state A yields the same resulting state as performing the original state-based operation. The delta-mutators for accounts are listed in Alg. 3 and described below. To streamline the presentation, the proofs that these are indeed correct delta mutators are given in Chapter 4 (see Proof 4.2.1).

The mutator $create^\delta$ (Alg. 3) takes a full account state A together with the desired amount of tokens to be created and returns a delta state. If the amount is valid, i.e. greater than zero and the account identifier is included in the set of allowed identifiers $\mathbb{C}$, then the resulting delta account A^δ consists only of the created token field, whereby the created counter is increased by the amount ($A^\delta_\uparrow = A_\uparrow + amount$). In the case the prerequisites for the created operation are not fulfilled, the account will remain in its current state without updating any counter. It consequently returns a delta state in which none of the fields are set, also called default state A^0. As explained previously, merging A^0 with any account state A will result in the same state A, because all properties of state A^0 contain the smallest possible value.

For burning tokens, the $burn^\delta$ mutator takes, similar to the $create^\delta$ operation, the current account state A and the requested amount of tokens to burn. There are also certain requirements that need to be met: the amount must be valid and the account needs a sufficient amount of tokens that can be burned. If the operation succeeds, a delta account A^δ is returned with only the burned field set to the updated burned counter $A^\delta_\downarrow = A_\downarrow + amount$. Otherwise, the mutator results in the default delta account A^0, which does not lead to a state modification when merged with the input state A.

The $giveTo^\delta$ mutator transfers the desired amount of tokens from account A to the account of the given id. If the balance of A is sufficient, the resulting delta state comprises a dictionary of grow-only counters containing exactly one key, the id of the receiver, whereby the corresponding value is the updated counter for this id. Otherwise, the mutator yields the default delta account A^0.

In the GOC-Ledger, tokens sent via the $giveTo^\delta$ operation are not automatically included in the balance of an account, but must first be acknowledged by the receiver. To achieve this, the $ackFrom^\delta$ mutator acknowledges all unacknowledged tokens sent from account B to A.

The *unackedFrom* operation, a query operation discussed in Section 3.2.3, determines the difference between the amount of received tokens from B and the amount of acknowledged tokens of A. If unacknowledged tokens exist, the mutator will return a delta state consisting of a dictionary of grow-only counters, which contains exactly one key, the identifier of account B together with the corresponding value set to the amount of acknowledged tokens. Otherwise, the mutator yields the default delta account A^0.

When decomposing the state-based GOC-Ledger, multiple delta-mutators may satisfy the requirement that merging the resulting delta account with the current state A produces the correct new State A', i.e. $op(A) = A \sqcup op^\delta(A)$. Even the existing state-based mutators are valid delta-mutators, but they do not provide any optimization benefits. Therefore, during the design phase of the δ-GOC-Ledger, we have ensured that all presented delta-mutators return minimal delta states that do not contain any redundant information (see Proof 4.2.2). This ensures that the delta design reduces the delta-account sizes and thus the communication overhead between peers as much as possible.

Note that the account mutators only operate on full account states. Using these operations on delta accounts may result in incorrect account states and consequently incorrect balances. For instance, consider an account state A with $A_\uparrow = 5$ and $A_\downarrow = 3$. Creating an additional token results in a delta account $A^\delta = create^\delta(A, 1)$ so that $A^\delta_\uparrow = 6$, while all other fields are dismissed. Performing a $burn^\delta$ operation on this delta account, $A^{\delta'} = burn^\delta(A^\delta, 2)$, and the subsequent merging of the delta accounts with the current account state A results in a state $A' = A \sqcup A^\delta \sqcup A^{\delta'}$, in which A' has a burn counter of $A'_\downarrow = 3$ instead of the expected amount of 5. The reason for this discrepancy lies in the removal of redundant information during the $create^\delta$ operation, which would be crucial for the subsequent correct application of the $burn^\delta$ operation.

The only exception where the correct application of a delta mutator to a delta account always leads to a correct state is the repeated use of the $create^\delta$ operation without applying any other operation in-between. In this case, the necessary information for this particular operator is retained in the delta states. This is not possible for the other presented delta-mutators, since the information required to calculate the balance or the number of unacknowledged tokens is missing.

To establish convergence, all mutators must be monotonic, i.e. they must modify a state in such a way that the resulting state is equal or larger than the input state. Since the merging of the delta-states obtained from the delta-mutators with the corresponding input state results in the same state, as later shown in Proof 4.2.1, the mutators presented are equivalent to the original monotonic state-based mutators.

Algorithm 3 Delta-Mutators for Accounts

Require: $\mathbb{C}$, set of identifiers allowed to create tokens

1:
2: **function** CREATE$^{\delta}(A, amount)$
3: $A^{\delta} \leftarrow initialize^{\delta}_{\mathbb{A}}(A_{id})$
4: **if** $A_{id} \in \mathbb{C} \wedge amount > 0$ **then**
5: $A^{\delta}_{\uparrow} \leftarrow A_{\uparrow} + amount$
6: **return** A^{δ}
7:
8: **function** BURN$^{\delta}(A, amount)$
9: $A^{\delta} \leftarrow initialize^{\delta}_{\mathbb{A}}(A_{id})$
10: **if** $amount > 0 \wedge balance(A) \geq amount$ **then**
11: $A^{\delta}_{\downarrow} \leftarrow A_{\downarrow} + amount$
12: **return** A^{δ}
13:
14: **function** GIVETO$^{\delta}(A, amount, id)$
15: $A^{\delta} \leftarrow initialize^{\delta}_{\mathbb{A}}(A_{id})$
16: **if** $amount > 0 \wedge balance(A) \geq amount$ **then**
17: **if** $id \notin A_{\rightarrow *}$ **then**
18: $A^{\delta}_{\rightarrow}[id] \leftarrow amount$
19: **else**
20: $A^{\delta}_{\rightarrow}[id] \leftarrow A_{\rightarrow}[id] + amount$
21: **return** A^{δ}
22:
23: **function** ACKFROM$^{\delta}(A, B)$
24: $A^{\delta} \leftarrow initialize^{\delta}_{\mathbb{A}}(A_{id})$
25: **if** $unackedFrom(A, B) > 0$ **then**
26: **if** $B_{id} \notin A_{\leftarrow *}$ **then**
27: $A^{\delta}_{\leftarrow}[B_{id}] \leftarrow B_{\rightarrow}[A_{id}]$
28: **else**
29: $A^{\delta}_{\leftarrow}[B_{id}] \leftarrow max(B_{\rightarrow}[A_{id}], A_{\leftarrow}[B_{id}])$
30: **return** A^{δ}

3.2.3 Query Functions

Unlike the delta-mutators outlined in the previous section, query functions do not return delta states. Instead, they are used to retrieve useful information based on the current state of the account and only work on full account states. Because the full states are identical to the GOC-Ledger design, the same query functions are used.

The *balance* method (Alg. 4) retrieves the balance of the current account state. To compute the credit that increases the balance, the *create* counter and the $ackFrom$ counters of all identifiers are added together. Conversely, debits are determined by summing the *burn* counter and all *giveTo* counters. The current balance is then derived from the difference between credit and debit. As the correct calculation of the balance requires knowledge of all counter values, it can only be calculated from a full account state.

To query the number of unacknowledged tokens from a given account B to the current account A, the $unackedFrom$ method (Alg. 4) calculates the difference between the tokens sent from B to A and the tokens of B acknowledged by A. If all received tokens are already

acknowledged, this function returns zero. Like the *balance* operation, this is also a query function and can only be used for full account states for the same reasons.

Algorithm 4 Account Query Operations as defined in GOC-Ledger [18]

```
1: function BALANCE(A)
2:     debits ← A_↑ + Σ_{id∈A_←*} A_←[id]
3:     credits ← A_↓ + Σ_{id∈A_→*} A_→[id]
4:     return debits − credits
5:
6: function UNACKEDFROM(A,B)
7:     if A_id ∈ B_→* ∧ B_id ∈ A_←* then
8:         return B_→A_id − A_←B_id
9:     else if A_id ∈ B_→* then
10:        return B_→A_id
11:    else
12:        return 0
```

3.2.4 Merging States

The last requirement to achieve eventual consistency is the definition of a merge function that computes the Least Upper Bound between two states in the semi-lattice S, which results in a state that is larger or equal to the two states. As in the case of the partial order ($\leq_{\mathbb{A}}$), the merge function $\sqcup_{\mathbb{A}}$ (Algorithm 5) defined for full account states is also applicable for delta accounts due to the internal representation of delta states.

There are three different merge scenarios: First, a full account A can be merged with another full account state A' which results in A'', as in the state-based approach of the GOC-Ledger. Second, a delta-account A^δ can be merged with a full account A resulting in a full account A''. This corresponds to the application of the effect represented by A^δ to the current state A. The replicas therefore only need to send the deltas instead of the whole state. The receiver can then merge the delta accounts with their local full state. Third, delta accounts can also be merged with other delta accounts, combining the effect of both states in a single delta account. Instead of applying multiple delta-mutators and transmitting each resulting delta account separately to other replicas, the delta states can be locally merged together into one delta state and then sent to other peers. As an additional benefit, this may reduce the communication overhead when a set of operations modify the same underlying counters, as only the highest counter values will be transmitted.

By establishing the partial ordering, the monotonicity of mutators, and the merge function, a monotonic join semi-lattice is defined, which guarantees, that all replicas, once they eventually have received all state updates, will converge to a consistent state, thus achieving strong eventual consistency [28].

Due to the chosen representation of delta accounts, the partial order and merge functions of the state-based GOC-Ledger can also be used for delta states. This is possible for two reasons: first, our notation for delta-state accounts does not explicitly show missing creation and burned counters, respectively written $A_\uparrow$ and $A_\downarrow$, otherwise the ordering and

merge operations would have to explicitly test for their presence and replace their value with the corresponding default value. In contrast to our pseudo-code notation, an actual implementation has to handle these cases explicitly. Second, the original design of the GOC-Ledger only stores counters for transfers that actually happened between accounts and not for all possible accounts. This was initially done to support an open set of participants but is basically also a form of storage optimization analogous to what we are introducing with delta accounts.

Algorithm 5 Merge function as defined in GOC-Ledger [18]

```
1:  function ⊔_𝔸(A, A′)                ▷ Merges accounts, delta accounts or a combination of both
2:      assert(A_id = A′_id)
3:
4:      A″ ← initialize_𝔸(A_id)
5:      A″_↑ ← max(A_↑, A′_↑)
6:      A″_↓ ← max(A_↓, A′_↓)
7:
8:      R ← A_→* ∪ A′_→*
9:      for id in R do
10:         if id ∈ A_→* ∧ id ∈ A′_→* then
11:             A″_→[id] ← max(A_→[id], A′_→[id])
12:         else if id ∈ A_→* then
13:             A″_→[id] ← A_→[id]
14:         else
15:             A″_→[id] ← A′_→[id]
16:
17:     S ← A_←* ∪ A′_←*
18:     for id in S do
19:         if id ∈ A_←* ∧ id ∈ A′_←* then
20:             A″_←[id] ← max(A_←[id], A′_←[id])
21:         else if id ∈ A_←* then
22:             A″_←[id] ← A_←[id]
23:         else
24:             A″_←[id] ← A′_←[id]
25:
26:     return A″
```

3.2.5 Account History

The partial order $\leq_{\mathbb{A}}$ described in Section 3.2.1 is analogous to the happens-before relationship (Section 2.3) between account states, i.e. a state A that happens before the state A', which is itself the result of an operation on A, is also smaller or equal to A', denoted as $A \leq_{\mathbb{A}} A'$. To better understand the history of account states and the relation between the state-based and δ-based design, this relationship can be depicted as a directed acyclic graph (DAG), where the vertices denote account states and the directed edges represent the state transitions resulting from the delta mutators or merging states. This section introduces the formal notation for such a DAG representing account histories that is later used to implement the δ-GOC Ledger design via Git and enable incremental computation of full states based on previously computed earlier states (Section 5.5).

Consider a directed acyclic graph (DAG) $G = (V, E)$ that consists of a set of vertices (nodes) V and a set of directed edges E. Since this graph is a DAG, all edges have a single direction and there exists no directed path that starts and ends at the same node. Every vertice V represents an account state $A \in V$, which is included in the set of all possible account states for the identifier of this account ($\mathbb{A}^{\geq}_{id}$), i.e. $V \subseteq \mathbb{A}^{\geq}_{id}$. Because the identifier cannot be changed after its initialization, every account state has the same identifier. E consists of directed labeled edges that represent a transition from one account state $A^j \in V$ to another account state $A^i \in V$, i.e. $(A^j, A^i, l) \in E$ such that $A^i \neq A^j$. The edge label l indicates which type of operation leads to the new account state. There are two possible operations: if there is exactly one incoming edge, the edge is labeled with the delta state $A^{\delta^{j \to i}}$[1] that resulted from the application of a delta-mutator on the parent account state A^j. If there are two or more incoming edges, the edges are labeled with $\sqcup$ to indicate that the resulting account state A^i is the result of merging all parent account states.

If a node has no incoming edge, it represents the initial account state ($A^i = A^0_{id}$) and is unique. Without loss of generality, it is assumed that each such DAG has exactly one such node, since every account must first be initialized before any other operation can be performed, and concurrent initialization followed by operations is equivalent to concurrent operations on the same initial state. Given that both delta-mutators and the merge operation result in a state equal or larger than the preceding states, this graph also depicts the causal history of the account states, with the edges representing the happens-before relationship between the account states.

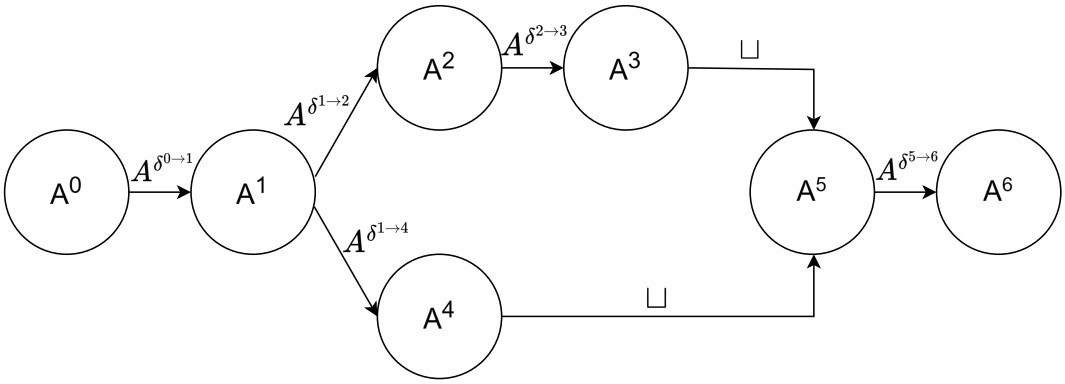


Figure 3.1: Example of an account history depicted as a directed acyclic graph.

Figure 3.1 illustrates an example of an account history represented as a DAG. The account state A^0 is the initialization state (state resulting from the execution of *initialize*) and therefore has no parent nodes. The other account states result from the application of different operations. In this example, $A^{\delta^{1\to2}}$ and $A^{\delta^{1\to4}}$ are performed simultaneously. Therefore the states A^2 and A^3 are concurrent to A^4, i.e. $A^2||A^4$ and $A^3||A^4$. The state A^6 is the most recent one. To calculate A^6, the preceding account state A^5 can be merged with the delta account $A^{\delta^{5\to6}}$ i.e., $A^6 = A^5 \sqcup A^{\delta^{5\to6}}$, as discussed in Section 3.2.2. Given that

[1] Note that the superscript only contains the start and resulting state to simplify the notation. It would be sufficient to enumerate the delta states, as the parent and resulting state are already given by the direction of the edge.

node A^5 has multiple incoming edges, its state can be calculated by merging its parents: $A^5 = A^3 \sqcup A^4$. Now, the two equations can be combined into $A^6 = A^3 \sqcup A^4 \sqcup A^{\delta^{5\to6}}$. The order of the merge is irrelevant because, the merge function is *commutative* and *associative*. This process can now be repeated recursively throughout the entire graph by expanding all account states with a merge of the parent states and the delta account state. The recursion will stop at the initial state A^0, because it has no parent nodes and cannot be further substituted. The equation eventually consists only of merges of the initial state A^0 and all applied delta accounts, as each edge was visited at least once: $A^6 = A^0 \sqcup A^{\delta^{0\to1}} \sqcup A^{\delta^{1\to2}} \sqcup A^{\delta^{2\to3}} \sqcup A^0 \sqcup A^{\delta^{0\to1}} \sqcup A^{\delta^{1\to4}} \sqcup A^{\delta^{5\to6}}$. The multiple occurrences of A^0 and $A^{\delta^{0\to1}}$ are caused by the concurrent states A^2 and A^4, which leads to branching in the graph. Therefore, recursive expansion runs through multiple paths between A^0 and A^6. However, these duplicates can be eliminated from the equation due to the *idempotence* of the merge function, yielding $A^6 = A^0 \sqcup A^{\delta^{0\to1}} \sqcup A^{\delta^{1\to2}} \sqcup A^{\delta^{2\to3}} \sqcup A^{\delta^{1\to4}} \sqcup A^{\delta^{5\to6}}$. This illustrates that the latest full state A^6 can be computed by merging all delta accounts of the account history with the initialize state A^0. We generalize this intuition and prove that it holds for any valid causal histories in Chapter 4 (see Proof 4.2.3).

Instead of expanding recursively until only the initial state A^0 remains, the expansion can also be performed until a previously computed full state, which we call *checkpoint*, is reached. In this way, applications can compute the latest state by merging checkpoint states with all delta states. This incremental computation of the latest state could be useful, for example, for systems with high transaction volumes and frequent queries of the current account state. By avoiding the necessity to traverse the entire account history to compute an updated state, this approach requires less computation and therefore provides lower latency.

3.3 Ledger

The previous section introduced the account states that can be modified using delta-mutators. However, a ledger comprises more than just a single account; it maintains records of a collection of multiple accounts. In the state-based GOC-Ledger, this is realized by a grow-only dictionary of accounts. The δ-GOC-Ledger employs the same delta decomposition technique as used for the accounts. The delta-based ledger (Alg. 6) is presented below.

Algorithm 6 Ledger

```
1:  function INITIALIZE
2:      L ← {}
3:      return L
4:
5:  function ADD^δ(L, A)
6:      L^δ ← initialize()
7:      if A_id ∉ L_* then
8:          L^δ[A_id] ← A
9:      else
10:         L^δ[A_id] ← L[A_id] ⊔_𝔸 A
11:     return L^δ
12:
13: function ≤_𝕃(L, L')
14:     return L_* ⊆ L'_* ∧ ⋀_{id∈L_*} L[id] ≤_𝔸 L'[id]
15:
16: function ⊔_𝕃(L, L')
17:     L'' ← initialize()
18:     I ← L_* ∪ L'_*
19:     for id in I do
20:         if id ∈ L_* ∧ id ∈ L'_* then
21:             L''[id] ← L[id] ⊔_𝔸 L'[id]
22:         else if id ∈ L_* then
23:             L''[id] ← L[id]
24:         else
25:             L''[id] ← L'[id]
26:     return L"
27:
28: function BALANCE(L)
29:     return {(id, balance(L[id])) for id in L_*}
```

Similar to the delta accounts, the operations of the ledger have been redefined as delta-mutators. These mutators yield a delta ledger state, which contains the effects of the application of this operation, and not the entire ledger state, the size of which increases linearly with the number of accounts managed by the ledger.

A ledger can be set up with *initialize*, which returns a ledger with an empty dictionary, i.e. with no accounts. As in the case of the delta account, it is assumed that the empty set is not sent explicitly, but a ledger without any information is interpreted implicitly as an empty dictionary, which ideally requires no storage.

Due to the implicit representation, the partial ordering of ledger states is determined by the same comparison function $\leq_{\mathbb{L}}$ used in the state-based ledger. For $L \leq_{\mathbb{L}} L'$ to be true, the account identifiers of L must be a subset of L' and for every account, the state in L is smaller ($\leq_{\mathbb{A}}$) than in L'. This comparison function also defines the ordering of delta-ledger states and full ledger states.

The merging of ledger states follows the same procedure as for the state-based version. The merge $L'' = L \sqcup_{\mathbb{L}} L'$ iterates through the union of account identifiers of both states. If an identifier is present in only one of the states, the corresponding account state is copied

to the new state L''. However, if both L and L' contain the identifier, $L''[id]$ is assigned to the merged account state of both states. The merge with a delta state L^δ represents the application of the delta mutator's effect, encoded in L^δ to the current state.

The *add* operation of the GOC-Ledger is decomposed into a delta-mutator using the same approach as for the account operations. This function adds an account A to the given ledger L or if this account is already included, updates the account state by merging it with the existing one. The add^δ mutator returns a delta-ledger state containing a dictionary with exactly one key, the identifier of the account A_{id} and the associated updated account state A. When merged with the current state L, the returned delta account has the same effect as the state-based operation (see Proof 4.3.1).

The query operation *balance* returns the current balances of all accounts managed by the state L. Similar to the delta accounts, its result is only valid for full ledger states.

3.4 Liveliness, Safety and Balance Properties

As shown in Proof 4.2.3 and Proof 4.3.2, merging all delta states with the initialization state results in the most recent full account state, which is identical to the state achieved with the state-based GOC-Ledger operations. This equality implies that the delta-state GOC-ledger complies with all its properties, including the liveliness and safety properties, as well as the non-zero balance conditions.

This concludes our presentation of the design of the delta GOC-Ledger, which separates the effects of account operations in the form of delta states from the full state. This way, replicas only need to replicate the delta states instead of the full state, thereby reducing communication overhead. Since the delta-mutators have the same effect as their state-based counterpart, the underlying ledger state stays unaffected. This also implies that the presented design inherits the properties of the state-based design and converges to a consistent state across all replicas. The proofs supporting these claims will be presented in the next chapter.

4
Proofs

In this chapter, the properties of the δ-GOC-Ledger presented in the previous chapter are formally proven. The following proofs adhere to the structured proof format proposed by Lamport [15].

4.1 Definitions

Firstly, additional notation is defined, which is important for the subsequent proofs.

1. $\mathbb{A}$ is the set of all possible account states as defined in the GOC-Ledger [18].
2. $\mathbb{A}_{id}^{\geq 0}$ is the subset of all possible account states with the same identifier id i.e., $\mathbb{A}_{id}^{\geq 0} \subset \mathbb{A}$ such that for every state $A \in \mathbb{A}$, $A_{id} = id$. The superscript (≥ 0) indicates that all counters of an account state are strictly growing.
3. $A_{id}^0 = intialize_{\mathbb{A}}(id)$ is the smallest element in $\mathbb{A}_{id}^{\geq 0}$. Since all counters of A_{id}^0 are the smallest possible and the merge results in the largest counter values of the two operands, merging any other account state $A \in \mathbb{A}_{id}^{\geq 0}$, with $A \neq A_{id}^0$ will result in A.

4.2 Accounts

4.2.1 Delta-mutators have the same effect as the state-based operators

DEFINE: 1. $op = \{create, burn, giveTo, ackFrom\}$, set of all defined state-based mutators for accounts, defined in GOC-Ledger [18].
2. $op^\delta = \{create^\delta, burn^\delta, giveTo^\delta, ackFrom^\delta\}$, set of all defined delta-mutators for accounts, introduced in Section 3.2.2.

ASSUME: 1. $A \in \mathbb{A}_{id}^{\geq 0}$
2. $amount \in \mathbb{R}$
3. $m \in op$ and $m^\delta \in op^\delta$ (m is the same operation in both)

PROVE: $m(A) = m^\delta(A) \sqcup A$

PROOF:

$\langle 1 \rangle 1$. CASE: $A' = create(A, amount)$ and $A'' = A^\delta \sqcup A$, with $A^\delta = create^\delta(A, amount)$

$\langle 2 \rangle 1$. CASE: $A_{id} \in \mathbb{C} \wedge amount > 0$

⟨3⟩1. $A'_{\uparrow} = A_{\uparrow} + amount$, all other properties of A remain unchanged in A'
Because of the effect of *create*.
⟨3⟩2. $A^{\delta}_{\uparrow} = A_{\uparrow} + amount$, all other properties of A^{δ} are equal to A^0_{id}
Because of the effect of $create^{\delta}$.
⟨3⟩3. $A''_{\uparrow} = A^{\delta}_{\uparrow}$, all other properties of A'' are the same as those of A
Counter values after a merge are always the largest of those of the two operands and all counter values of A are equal or larger than those of A^0_{id} because $A \geq_{\mathbb{A}} A^0_{id}$.
⟨3⟩4. Q.E.D.
$A' = A''$ because $A''_{\uparrow} = A'_{\uparrow} = A_{\uparrow} + amount$ and all other properties of A' and A'' are equal to those of A.
⟨2⟩2. Case: $A_{id} \notin \mathbb{C} \vee amount \leq 0$
⟨3⟩1. $A' = A$
Because of the effect of *create*.
⟨3⟩2. $A^{\delta} = A^0_{id}$
Because of the effect of $create^{\delta}$.
⟨3⟩3. $A'' = A$
Since the merge function computes the least upper bound of both operands and $A \geq_{\mathbb{A}} A^0_{id}$, the resulting state A'' is equal to A.
⟨3⟩4. Q.E.D.
$A' = A''$, because $A'' = A' = A$.
⟨1⟩2. Case: $A' = burn(A, amount)$ and $A'' = A^{\delta} \sqcup A$, with $A^{\delta} = burn^{\delta}(A, amount)$
Idem ⟨1⟩1 by replacing $A''_{\uparrow}$ for $A''_{\downarrow}$ and *create* by *burn*.
⟨1⟩3. Case: $A' = giveTo(A, amount, id)$ and $A'' = A^{\delta} \sqcup A$, with $A^{\delta} = giveTo^{\delta}(A, amount, id)$
⟨2⟩1. Case: $amount > 0 \wedge balance(A) \geq amount$
⟨3⟩1. Case: $id \notin A_{\rightarrow *}$
⟨4⟩1. $A'_{\rightarrow *} = A_{\rightarrow *} \cup \{id\}$, with $A'_{\rightarrow}[id] = amount$, all other properties of A remain unchanged in A'.
Because of the effect of *giveTo*.
⟨4⟩2. $A^{\delta}_{\rightarrow *} = \{id\}$, with $A^{\delta}_{\rightarrow}[id] = amount$, all other properties of A^{δ} are equal to A^0_{id}.
Because of the effect of $giveTo^{\delta}$.
⟨4⟩3. $A''_{\rightarrow *} = A_{\rightarrow *} \cup A^{\delta}_{\rightarrow *}$, with $A''_{\rightarrow}[id] = A^{\delta}_{\rightarrow}[id]$. All other properties of A'' are equal to those of A.
The merge joins the set of identifiers of the giveTo attribute of both operands. Because id is only contained in A^{δ}, the value of $A''_{\rightarrow}[id]$ is the same as in A^{δ}.
⟨4⟩4. Q.E.D.
$A' = A''$ because $A''_{\rightarrow *} = A'_{\rightarrow *} = A_{\rightarrow *} \cup \{id\}$ and $A''_{\rightarrow}[id] = A'_{\rightarrow}[id] = amount$ and all other properties of A' and A'' are equal to those of A.
⟨3⟩2. Case: $id \in A_{\rightarrow *}$
⟨4⟩1. $A'_{\rightarrow}[id] = A_{\rightarrow}[id] + amount$, all other properties of A remain unchanged in A'.
Because of the effect of *giveTo*.
⟨4⟩2. $A^{\delta}_{\rightarrow}[id] = A_{\rightarrow}[id] + amount$, all other properties of A^{δ} are equal to A^0_{id}
Because of the effect of $giveTo^{\delta}$.

⟨4⟩3. $A''_{\rightarrow}[id] = A^{\delta}_{\rightarrow}[id]$

Because $A^{\delta}_{\rightarrow}[id] > A_{\rightarrow}[id]$ and the resulting state of the merge function consists of counters that are larger or equal than those of the operands.

⟨4⟩4. Q.E.D.

$A' = A''$ because $A''_{\rightarrow}[id] = A'_{\rightarrow}[id] = A_{\rightarrow}[id] + amount$ and all other properties are equal to A.

⟨2⟩2. Case: $amount \leq 0 \vee balance(A) < amount$

⟨3⟩1. $A' = A$

Because of the effect of $giveTo$.

⟨3⟩2. $A^{\delta} = A^{0}_{id}$

Because of the effect of $giveTo^{\delta}$.

⟨3⟩3. $A'' = A$

Merging any state A with A^{0}_{id} results in the same state A.

⟨3⟩4. Q.E.D.

$A' = A''$ because $A'' = A' = A$.

⟨1⟩4. Case: $A' = ackFrom(A, B)$ and $A'' = ackFrom^{\delta}(A, B) \sqcup A$

Idem ⟨1⟩3 by replacing $A''_{\rightarrow}$ for $A''_{\leftarrow}$ and $giveTo$ by $ackFrom$.

⟨1⟩5. Q.E.D.

For every delta-mutator in ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, the resulting delta-account merged with the base state A leads to the same result as the state-based mutators. Therefore, $create^{\delta}$, $burn^{\delta}$, $giveTo^{\delta}$ and $ackFrom^{\delta}$ have the same effect on the account state as their state-based counterparts when merged with the current state A.

4.2.2 Account delta-mutators return minimal delta states

When decomposing a state-based CRDT into a Delta State Replicated Datatype, there are multiple valid delta-mutators. Even the trivial decomposition, where the state-based mutator m and the delta-mutator m^{δ} are equal for every possible account state $m(A) = m^{\delta}(A)$, is a valid decomposition. However, the state returned by a delta-mutator should be minimized to achieve the best possible optimization. Therefore, delta-mutators should be preferred, which return the smallest possible states and do not leak any redundant information that is already contained in the account state A. In the following, it is shown that the designed delta-mutators for account states are indeed returning minimal delta-states.

Define: 1. $op = \{create, burn, giveTo, ackFrom\}$, set of all defined state-based mutators for accounts, defined in GOC-Ledger [18].

2. $op^{\delta} = \{create^{\delta}, burn^{\delta}, giveTo^{\delta}, ackFrom^{\delta}\}$, set of all defined delta-mutators for accounts, introduced in Section 3.2.2.

3. $\mathbb{OP}^{\delta}$ is the set of all possible delta-mutators (valid or not), which implies that $(op \cup op^{\delta}) \subseteq \mathbb{OP}^{\delta}$.

Assume: 1. $A \in \mathbb{A}^{\geq 0}_{id}$

2. $m \in op$ and $m^{\delta} \in op^{\delta}$ (m is the same operation in both)

3. $\mathbb{A}^{\delta}$ is the set of all possible delta states, i.e. for all $A^{\delta'} \in \mathbb{A}^{\delta}$, $A^{\delta'} = m^{\delta'}(A)$ such that $m^{\delta'} \in \mathbb{OP}^{\delta}$ and $m(A) = A^{\delta'} \sqcup_{\mathbb{A}} A$

PROVE: $A^\delta = m^\delta(A)$ is the smallest state of $\mathbb{A}^\delta$.
PROOF SKETCH: By contradiction. Since at most a single counter has been modified in A^δ compared to A and that counter is equal to the corresponding counter of A, then a smaller delta state $A^{\delta''} <_{\mathbb{A}} A^\delta$ cannot be in $\mathbb{A}^\delta$, because it would contradict $m(A) = A^{\delta''} \sqcup_{\mathbb{A}} A$.
PROOF:
⟨1⟩1. CASE: $A^\delta = create^\delta(A, amount)$ and $\nexists A^{\delta''} \in \mathbb{A}^\delta$, with $A^{\delta''} <_{\mathbb{A}} A^\delta$
  ⟨2⟩1. CASE: $A_{id} \in \mathbb{C} \wedge amount > 0$
    ⟨3⟩1. $A^\delta_\uparrow = A_\uparrow + amount$, all other attributes are equal to A^0_{id}
      Because of the effect of $create^\delta$.
    ⟨3⟩2. $A^{\delta''}_\uparrow < A^\delta_\uparrow$, all other attributes of $A^{\delta''}$ are equal to A^0_{id}
      Assume by contradiction that such a state $A^{\delta''}$ exists. In order to make $A^{\delta''} <_{\mathbb{A}} A^\delta$ true, at least one counter of $A^{\delta''}$ must be smaller than A^δ. Since all attributes besides the create counter $A^\delta_\uparrow$ are equal to those of A^0_{id}, $A^{\delta''}_\uparrow$ must be smaller than $A^\delta_\uparrow$.
    ⟨3⟩3. Q.E.D.
      Since $A' = create(A, amount)$, with $A'_\uparrow = A_\uparrow + amount$, the state $A^{\delta''}$ cannot be in $\mathbb{A}^\delta$, because $A^{\delta''}_\uparrow < A'_\uparrow$, thus contradicting $create(A) = A^{\delta''} \sqcup_{\mathbb{A}} A$. Therefore A^δ is the smallest state of $\mathbb{A}^\delta$.
  ⟨2⟩2. CASE: $A_{id} \notin \mathbb{C} \vee amount \leq 0$
    ⟨3⟩1. $A^\delta = A^0_{id}$
      Because of the effect of $create^\delta$.
    ⟨3⟩2. Q.E.D.
      Assume by contradiction that $A^{\delta''}$ exists. Then $A^{\delta''}$ must be smaller than A^0_{id}. However, by definition A^0_{id} is the smallest state that $create^\delta$ can return. Therefore, A^δ is the smallest state of $\mathbb{A}^\delta$.
⟨1⟩2. CASE: $A^\delta = burn^\delta(A, amount)$ and $\nexists A^{\delta''} \in \mathbb{A}^\delta$, with $A^{\delta''} <_{\mathbb{A}} A^\delta$
  Idem ⟨1⟩1, by replacing $A^\delta_\uparrow$ with $A^\delta_\downarrow$ and *create* by *burn*.
⟨1⟩3. CASE: $A^\delta = giveTo^\delta(A, amount, id)$ and $\nexists A^{\delta''} \in \mathbb{A}^\delta$, with $A^{\delta''} <_{\mathbb{A}} A^\delta$
  ⟨2⟩1. CASE: $amount > 0 \wedge balance(A) \geq amount$
    ⟨3⟩1. CASE: $id \notin A_{\rightarrow *}$
      ⟨4⟩1. $A^\delta_{\rightarrow *} = \{id\}$, with $A^\delta_\rightarrow[id] = amount$. All other properties are equal to A^0_{id}
        Because of the effect of $giveTo^\delta$.
        ⟨5⟩1. CASE: $A^{\delta''}_{\rightarrow *} \subset A^\delta_{\rightarrow *}$, all other properties of $A^{\delta''}$ are equal to A^0_{id}
          Assume by contradiction that $A^{\delta''}$ exists, then to make $A^{\delta''} <_{\mathbb{A}} A^\delta$ true, $A^{\delta''}_{\rightarrow *}$ must be a strict subset of $A^\delta_{\rightarrow *}$. Because $A^\delta_{\rightarrow *}$ contains only one identifier, $A^{\delta''}_{\rightarrow *} = \emptyset$.
        ⟨5⟩2. CASE: $A^{\delta''}_\rightarrow[id] < A^\delta_\rightarrow[id]$, all other properties of $A^{\delta''}$ are equal to A^0_{id}
          Assume by contradiction that $A^{\delta''}$ exists, then to make $A^{\delta''} <_{\mathbb{A}} A^\delta$ true, $A^{\delta''}_{\rightarrow *}[id]$ must be smaller than $A^\delta_{\rightarrow *}[id]$.
        ⟨5⟩3. Q.E.D.
          By contradiction. In either case ⟨5⟩1 and ⟨5⟩2, $A^{\delta''}$ contradicts the requirement $giveTo^\delta(A, amount, id) = A^{\delta''} \sqcup_{\mathbb{A}} A$ and therefore is not in $\mathbb{A}^\delta$. This means that A^δ is the smallest possible state in $\mathbb{A}^\delta$.
    ⟨3⟩2. CASE: $id \in A_{\rightarrow *}$

⟨4⟩1. $A^{\delta}_{\rightarrow *} = \{id\}$, with $A^{\delta}_{\rightarrow}[id] = A_{\rightarrow}[id] + amount$. All other properties are equal to A^{0}_{id}

Because of the effect of $giveTo^{\delta}$.

⟨4⟩2. Q.E.D.

Idem ⟨3⟩1 because the keys in $A^{\delta}_{\rightarrow *}$ are equal. The only difference is the value for the key id. However, also in this case a smaller state $A^{\delta''}$ must have a smaller value for id, which contradicts the assumed requirements for $A^{\delta''}$.

⟨2⟩2. Case: $amount \leq 0 \vee balance(A) < amount$

⟨3⟩1. $A^{\delta} = A^{0}_{id}$

Because of the effect of $giveTo^{\delta}$.

⟨3⟩2. Q.E.D.

Assume by contradiction that $A^{\delta''}$ exists. Then $A^{\delta''}$ must be smaller than A^{0}_{id}. However, by definition A^{0}_{id} is the smallest state in $\mathbb{A}^{\delta}$. Therefore, A^{δ} is the smallest state of $\mathbb{A}^{\delta}$.

⟨1⟩4. Case: $A^{\delta} = ackFrom^{\delta}(A, B)$ and $\nexists A^{\delta''} \in \mathbb{A}^{\delta}$, with $A^{\delta''} <_{\mathbb{A}} A^{\delta}$

Idem ⟨1⟩3 by replacing $A^{\delta}_{\rightarrow}$ with $A^{\delta}_{\leftarrow}$ and $giveTo$ by $ackFrom$.

⟨1⟩5. Q.E.D.

In ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 was shown that there does not exist any other delta state that is smaller than the delta state A^{δ} returned by the presented delta mutators.

It follows from this result that the delta states returned by our delta mutators (op^{δ}) use the least amount of space, because the relative space usage is proportional to the ordering between delta states, i.e. a larger delta state has more counters with non-default values and/or counters with larger values that takes either the same or more space to represent.

4.2.3 Equality of state-based and delta-based account states

Define: 1. $op^{\delta} = \{create^{\delta}, burn^{\delta}, giveTo^{\delta}, ackFrom^{\delta}\}$, set of all defined delta-mutators for accounts introduced in Section 3.2.2.

2. $G = (V, E)$ is a directed acyclic graph (DAG) representing the account history as introduced in Section 3.2.5, with:
   a) V a set of vertices representing account states such that $V \subseteq \mathbb{A}^{\geq 0}_{id}$.
   b) E a set of edges representing transitions between states such that $(A^j, A^i, l) \in E$ is a directed edge that starts from state $A^j \in V$, ends on state $A^i \in V$, and is labelled with operation l.
   c) l is an operation, either:
      i. $\sqcup$ if it represents a merge of multiple parent states.
      ii. the delta state $A^{\delta^{j \rightarrow i}}$ that resulted from the application of a delta-mutator $m^{\delta} \in op^{\delta}$ on A^j, i.e. $A^{\delta^{j \rightarrow i}} = m^{\delta}(A^j)$.
3. $P(A^i) = \{A^j \mid (A^j, A^i, l) \in E\}$, the direct parents of account state A^i
4. $\mathcal{H}(A^i) = \{A^j \mid \text{there exists a directed path from } A^j \text{ to } A^i\} \cup \{A^i\}$, the causal history of account state A^i.
5. $\mathcal{A}(A^i) = \{A^j \mid \text{transitive parents of } A^i\} \cup \{A^i\}$, the set of causal states of A^i.

6. $\mathcal{A}^\delta(A^i) = \{A^{\delta^{j\to k}} \mid A^k \in \mathcal{A}(A^i) \wedge (A^j, A^k, A^{\delta^{j\to k}}) \in E\}$, set of causal delta-states of A^i.

ASSUME: 1. For any state $A^i \in V$, exactly one of the three following cases apply:

a) A^i has no incoming edges and is the initial state ($A^i = A^0_{id}$), therefore it is unique and has no parents ($P(A^0_{id}) = \emptyset$).

b) A^i has exactly one incoming edge starting from A^j labeled $A^{\delta^{j\to i}}$ and is the result of applying a delta mutator on its parents, i.e. $A^i = A^{\delta^{j\to i}} \sqcup_{\mathbb{A}} A^j$.

c) A^i has two or more incoming edges all labeled with $\sqcup$ and is the result of merging all its parents, i.e. $A^i = \bigsqcup_{A^k \in P(A^i)} A^k$.

PROVE: $A^i = A^0_{id} \sqcup_{\mathbb{A}} \bigsqcup_{A^{\delta^{j\to k}} \in \mathcal{A}^\delta(A^i)} A^{\delta^{j\to k}}$

PROOF:

⟨1⟩1. **Expansion**:

Starting from $A^i = A^i$, the following cases are recursively applied on any full state A^k present on the right-hand side of the equation until only full states of the form A^0_{id} and delta states of the form $A^{\delta^{j\to k}}$ remain.

CASE: $A^k = A^0_{id}$: Base case of the recursion, stop (because $P(A^0_{id}) = \emptyset$ and cannot be expanded any further).

CASE: $A^k = A^{\delta^{j\to k}} \sqcup_{\mathbb{A}} A^j$: Delta-mutator case, expand A^j.

CASE: $A^k = \bigsqcup_{A^l \in P(A^k)} A^l$: Merge case, expand every A^l.

⟨1⟩2. **Reduction**:

After the expansion step in ⟨1⟩1, all states between A^i and A^0_{id} were traversed. As a result, every delta state in $\mathcal{A}^\delta(A^i)$ and the initial state A^0_{id} remain on the right-hand side of the equation. Since the merge function $\sqcup_{\mathbb{A}}$ is commutative and associative, this can be combined in $A^0_{id} \sqcup_{\mathbb{A}} \bigsqcup_{A^{\delta^{j\to k}} \in \mathcal{A}^\delta(A^i)} A^{\delta^{j\to k}}$. If merge cases occur during the expansion step, i.e. there is branching in the graph G, several paths between A^i and A^0_{id} are traversed during the expansion, which leads to multiple occurrences of the same delta states and the initial state A^0_{id} on the right-hand side of the equation. However, due to the idempotency of the merge function, these redundant occurrences can be eliminated.

⟨1⟩3. Q.E.D.

Applying the expansion step ⟨1⟩1 followed by the reduction step ⟨1⟩2 results in the equation $A^i = A^0_{id} \sqcup_{\mathbb{A}} \bigsqcup_{A^{\delta^{j\to k}} \in \mathcal{A}^\delta(A^i)} A^{\delta^{j\to k}}$.

The expansion need not always be done until only A^0_{id} full states remain. It can also be performed only until the last computed full states, i.e. *checkpoints*. We call the set of checkpoints a *frontier* and we define it as a subset of the causal states of A^i, i.e. $\mathcal{F}(A^i) \subseteq \mathcal{A}(A^i)$, with its corresponding set of causal delta states $\mathcal{F}^\delta(A^i) = \bigsqcup_{A^k \in \mathcal{F}(A^i)} \mathcal{A}^\delta(A^k)$.

Using these definitions, the previous equality can therefore be rewritten as:

$$A^i = \bigsqcup_{A^k \in \mathcal{F}(A^i)} A^k \sqcup_{\mathbb{A}} \bigsqcup_{A^{\delta^{j\to k}} \in (\mathcal{A}^\delta(A^i) \setminus \mathcal{F}^\delta(A^i))} A^{\delta^{j\to k}} \tag{4.1}$$

The proof follows easily from the previous one.

4.3 Ledger

4.3.1 Delta-mutators have the same effect as the state-based mutators

DEFINE: 1. $\mathbb{L}$ is the set of all possible ledger states as defined in the GOC-Ledger [18].
2. $L \in \mathbb{L}$
3. $A \in \mathbb{A}_{id}^{\geq 0}$

PROVE: $add(L, A) = add^{\delta}(L, A) \sqcup L$

PROOF:

$\langle 1 \rangle 1$. CASE: $L' = add(L, A), L^{\delta} = add^{\delta}(L, A), L'' = L^{\delta} \sqcup_{\mathbb{L}} L$

$\langle 2 \rangle 1$. CASE: $A_{id} \notin L_*$

$\langle 2 \rangle 2$. $L'_* = L_* \cup \{A_{id}\}$, with $L'[A_{id}] = A$, while all other properties of L remain unchanged in L'.

Because of the effect of *add*.

$\langle 2 \rangle 3$. $L^{\delta}_* = L_* \cup \{A_{id}\}$

Because of the effect of add^{δ}.

$\langle 2 \rangle 4$. $L''_* = L^{\delta}_*$

By definition, the merge function always results in a state that is equal or larger than both operand states. Since $L_* \subset L^{\delta}_*$, L is a smaller state than L^{δ} and therefore $L''_* = L^{\delta}_*$.

$\langle 2 \rangle 5$. Q.E.D.

$L' = L''$ because $L''_* = L'_* = L_* \cup \{A_{id}\}$.

$\langle 2 \rangle 2$. CASE: $A_{id} \in L_*$

$\langle 2 \rangle 3$. $L'[A_{id}] = L[A_{id}] \sqcup_{\mathbb{A}} A$, while all other properties of L remain unchanged in L'.

Because of the effect of *add*.

$\langle 2 \rangle 4$. $L^{\delta}[A_{id}] = L[A_{id}] \sqcup_{\mathbb{A}} A$

Because of the effect of add^{δ}.

$\langle 2 \rangle 5$. $L''[A_{id}] = L^{\delta}[A_{id}]$, while all other properties of L remain unchanged in L'.

Because L^{δ} contains only one account state (A_{id}) and this account state is equal or larger than in L.

$\langle 2 \rangle 6$. Q.E.D.

$L' = L''$ because $L''[A_{id}] = L'[A_{id}] = L[A_{id}] \sqcup_{\mathbb{A}} A$ and all properties of L' and L'' are equal to those of L.

$\langle 1 \rangle 2$. Q.E.D.

As shown in $\langle 1 \rangle 1$, the delta-mutator add^{δ} leads to the same ledger state as the state-based operator, when merged with the current state L and therefore has the same effect.

4.3.2 Equality of state-based and delta-based ledger states

Because proving the equality of state-based and delta-based ledger states follows a similar approach to the account state equality proof (Proof 4.2.3), we provide only a sketch proof here.

Proof sketch: Analogous to the equivalence proof of account states (refer to Proof 4.2.3), a directed acyclic graph (DAG) can be used to represent the history of ledger states, with the vertices denoting ledger states and the edges indicating either a merge or a delta ledger state. The equation to be proven can then be achieved by performing the same expansion and reduction steps as for the account states.

5 Implementation

This chapter introduces an implementation of the previously presented δ-GOC-Ledger design using Git [33]. It allows users to create new accounts, initialize tokens, and perform all defined token operations via a terminal user interface. The implementation is based on Git and Bash, which can be easily ported to other programming languages. Furthermore, Git allows us to use the optimized Git operations for the reconciliation and checkpointing process. This prototype serves as an initial platform to explore the suitability of Git for implementing Delta CRDTs with a similar structure as the δ-GOC-Ledger, which is analyzed in the evaluation Chapter 6.

Each replica can initialize an author, which includes generating an ed25519 keypair, setting up a local Git repository, and committing an alias message that can be used to publish a user's self-chosen alias. The implementation supports the initialization of multiple independent token types by any user to support *local crypto-tokens* [19], where each token is identified by the hash of their initialization message. The implementation uses the combination of the author's public key and the hash of the token initialization message as the unique identifier of an account (see Algorithm 1), i.e. the account ID. Once initialized, an author can perform any operation supported by the δ-GOC-Ledger. The Git push/pull model is used to synchronize the replicas in the system, while the implementation performs additional checks on the received updates to verify their correctness and ensure that the replicas stay in a correct state.

Because integers in Bash are limited to a specific system-dependent range[2], all arithmetic operations are performed using Python, which can handle larger numbers without overflow issues. In the following sections, important parts of the implementation are presented in depth.

5.1 System Assumptions

The implementation is designed for a distributed system with an arbitrarily large set of replicas. Each author is identified by a public key, while the corresponding private key is

[2] typically $2^{63} - 1$ on modern 64-bit systems

kept secret by the owner. In addition, it is assumed that each delta state created by the same author is ordered sequentially within its log, resulting in a total order. This means that there is no forking in a log, which could otherwise lead to *fork-based double-spending* [19]. However, this is a limitation of the implementation and not of Git or the δ-GOC-Ledger. There are known algorithms, such as the *2P-BFT-Log* [17] that could be used in order to detect forking.

5.2 Commits as Delta States

Every account operation results in a delta state, which is implemented as a Git commit object that contains various attributes: the commit's author that created contributions included within the commit, the name of the committer, the commit tree, an SHA-1 that points to a tree object associated with this commit, none or multiple parents of the commit and a commit message. Table 5.1 provides an overview of how these attributes are used for the implementation of delta states. The author name of every commit is set to the ed25519 public key of the account. Because authors always commit updates themselves, the author of the commit and the name of the committer are identical.

Commit attribute	**Purpose**
Author name	Public key of the account that created the delta state.
Committer name	Same as author name.
Commit tree	Hash referencing the tree-object that represents the delta state.
Parents	References to other commits this delta state relies on.
Commit message	Only used in special cases, e.g. when storing token alias.
Signature	Cryptographic signature of all other commit attributes using the private key associated with the public key of the account.

Table 5.1: Overview of the purpose of each relevant commit attribute.

The object-based design of Git makes it possible to map the data structure used in the δ-GOC-Ledger to Git objects in a straightforward way. Each commit points to a commit tree that contains the actual data of the delta state in the form of tree and blob objects. This tree can include up to four entries, each representing a different attribute of the delta state. The *created* and *burned* entries point to a blob object containing the corresponding grow-only counter. The $giveTo$ and $ackFrom$ fields refer to trees, with each entry containing a reference to the blob that stores the counter for the corresponding account ID. If a specific field is missing in a delta state, the corresponding entry is also missing in the commit tree. In such cases, accessing those fields is interpreted as retrieving the default values defined in Section 3.2. If the state has no attributes at all, for example after initializing an account, the commit points to the empty tree.

When initializing a new token, a commit is created that stores the alias of the token as a commit message. This initial commit does not reference any parent commits, as this is the first operation for this token. Subsequent operations result in commits that store the corresponding delta state in a tree as described above. The commit message is not used, but could be useful for other applications or debugging purposes.

Subsequent delta states always keep a reference to the previous delta state of this account

as a parent, forming an append-only log-like structure for this account ID. In addition, when acknowledging tokens transferred from another account, reference is also made to the latest commit of this account in order to encode the causal relationship between the transfer and the acknowledge operations. This way, the commit history forms a directed acyclic graph (DAG) for each token type, similar to the DAG representing the account history (see Section 3.2.5), with the commits representing the delta accounts and the pointers to previous commits depicting their causal relationship. Due to the characteristics of δ-CRDTs, the order of received delta states and the number of duplicates are irrelevant for the resulting state, because the merge is commutative, associative, and idempotent. However, the established causal relationship between commits is important for the replication with Git (see Section 5.4) and for efficiently calculating full account states (see Section 5.5). Furthermore, the self-certifying properties of the resulting append-only logs lead to a chain of trust, whereby every replica can verify that the delta states are indeed created by the corresponding author. Modifying or excluding parts of the log would break the chain of trust. Consequently, the merge of multiple delta states to further reduce the update message sizes is not possible with the current implementation. Merging would combine multiple commits into one, altering the author's log.

Each commit is signed with the private key of the author who generated the associated delta state. This verifies that the delta state was indeed created by this author and prevents any subsequent modifications to the commit. Every valid commit must satisfy the following criteria: 1) The signature of the author can be verified by using the public key contained in the author name field. 2) If the commit does not represent the initialize state, then there must exist a path in the commit history to exactly one commit with no parents, the initialize commit of the corresponding token.

These criteria ensure that a message is correctly forged. However, they do not guarantee that the included delta state is correct with respect to the properties of the GOC-Ledger, which is verified during the creation of checkpoints (see Section 5.5).

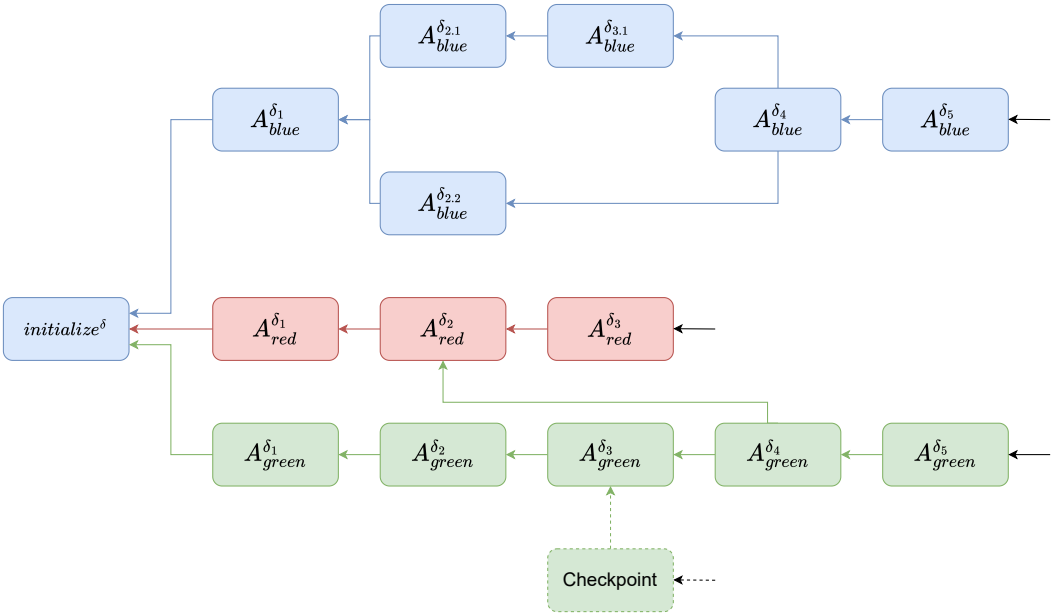


Figure 5.1: Example of a commit history for one token type.

Figure 5.1 shows an example of a commit history for a single token type that is used by multiple authors. Each box corresponds to a commit that represents a delta state resulting from an account operation. The directed edges represent the parents of the commit, while the different colors indicate the author of the commit. This graph only shows a commit history for one token type. The implementation allows the authors to interact with multiple token types simultaneously, resulting in one commit history per token type, which are independent from each other.

The commits of the blue author correspond to the account history shown in Figure 3.1. Similar to this graph, the commit history of the blue author is a directed acyclic graph. However, in the commit history, the nodes represent delta states instead of full account states and the edges correspond to the causal relationship between delta states. Because the account history includes concurrent states after state A^1 for a single account, the corresponding commit history includes branching after $A^{\delta_1}_{blue}$. Concurrent states are not supported in the current implementation of the δ-GOC-Ledger, but could be handled with 2P-BFT-Log [17]. At the time of publication, it is assumed that the other authors behave correctly and form an append-only log, where the first commit references the initialization message of the token and the subsequent commits reference the latest commit of the log. In addition, $A^{\delta_4}_{green}$ references a commit of the red author. This indicates that this delta state is a result of the $ackFrom^{\delta}$ operation, where the green author has acknowledged the tokens received from red.

The graph also shows a checkpoint for the green author that was computed locally, which is explained in Section 5.5. The black edges on the right-hand side of the graph represent Git references that are explained in detail in Section 5.3.

5.3 References as Frontier

In order to maintain the current ledger state of the replicas, for every token type, the last commit of each author is stored as a Git reference. These references serve as the frontier of the local state, which is important for retrieving the latest full account state and for the replication process. Since references are technically a mapping of names to commit hashes, we also use them as a fast lookup table to retrieve the hash of the corresponding author alias. Figure 5.2 provides a graphical overview of the implemented reference file structure.

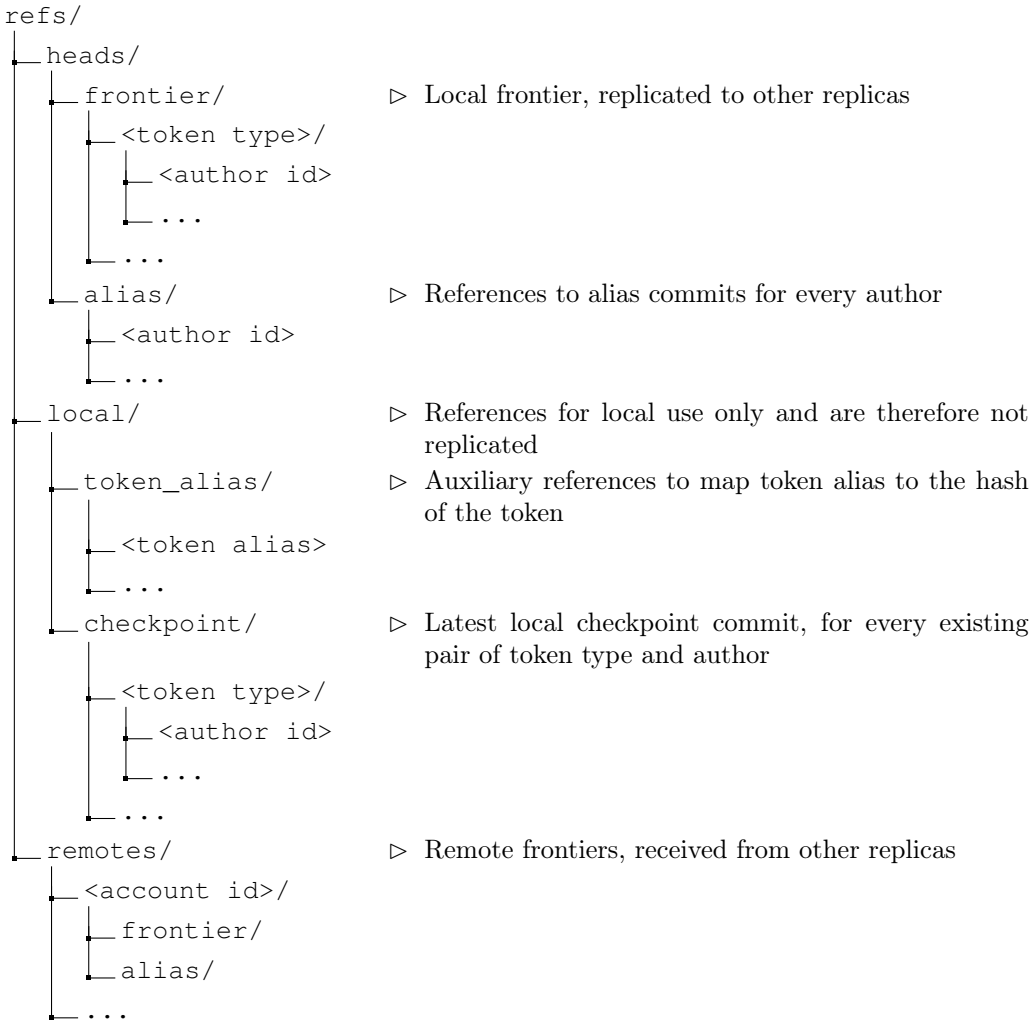


Figure 5.2: Directory of Git references used for the implementation.

In Git, references are usually saved in the `refs` directory, with local references stored in `heads` and those of other replicas in `remotes`. To better distinguish the references that are not synchronized with other replicas, for example, local auxiliary data, the implementation additionally introduces the `local` folder. The references stored in `heads` are split into two directories: `frontier` and `alias`.

In the `ledger` directory, references are stored, which are also synchronized with other replicas. For each existing token type, the tips are stored in their own directory within the

`frontier` directory, i.e. for every author that interacted with this token type, there is a reference pointing to the latest message of this author. This way, the latest commit for a specific account ID can be easily retrieved. In addition, the tips of all token types combined serve as the frontier of the local repository state, which is important for the replication process (Section 5.4). A similar approach is also used for maintaining the aliases of the accounts. Here, the latest alias commit for each author is stored in a reference named after the public key of the author within the `alias` directory. This enables other replicas to quickly retrieve the current alias of a replica.

The `local` directory stores references to auxiliary data that is important for local operations. Inside the `checkpoint` subdirectory, references to the latest calculated checkpoints are stored for each account ID. Additionally, to efficiently translate token aliases to their corresponding hash, each known token type has a reference named after its alias stored in `token_alias` and points to the commit of the initialize operation. The references in `local` are only used locally and are not replicated to other remotes, because the information they contain is extracted from the delta states of the δ-GOC-Ledger and can therefore be computed independently at each replica.

During replication, the frontiers received from other replicas are stored in `remotes`. Within this directory, for each replica, a folder is created named after its public key, which contains a copy of its local frontier. These references are then later used to update the local repository state in `frontier`. The combination of the signed commits by the author's private key and the reference names that contain the author's public key establishes *self-certification* [21] of the frontier references. As a result, if an adversary attempts to update a reference to a commit that originates from another author than the one specified in the reference name, the other replicas can easily determine that the signature of the commit cannot be verified by the public key and therefore refuse the update of the frontier.

5.4 Replication

The implementation relies on Git's synchronization protocol in order to exchange delta states between replicas. By adding another replica as a Git remote, it can synchronize with this remote using any protocol that is supported by Git. The replication process is divided into two steps: the pull/push phase and the merge phase.

Updates can be exchanged with other replicas by either pushing or pulling data. In both cases, the updated references of remotes are stored in their directory and corresponding Git objects are exchanged. During this process, the local references that represent the current state remain unchanged and are later updated in a separate merge step. When pushing updates to another replica, the sender signs the data by using its private key. The receiving replica relies on Git hook scripts to verify the signature of the received update. References are only updated if the signature matches the author ID contained in the corresponding remote reference directory that the sender wants to update. In addition, the receiver checks that updates pushed by the sender are fast-forwards, i.e. that the current commit is causally an ancestor of the updated reference. Therefore, a sender can only update existing references to a commit that causally happened after the commit of the current recipient's reference by

sending all causally preceding updates. If the signature is invalid or the push is non-fast-forward, the update process is canceled. This prevents anyone from impersonating another replica during the push process and ensures that the sending side is not able to truncate the commit history. Pushing an additional branch is also a non-fast-forward update, causing a replica to remain in the same fork of the branch. Consequently, the implementation is not robust against concurrent operations from the same account ID. Furthermore, the implementation allows to pull updates of other replicas by using the Git fetch operation that functions in a similar way to the push, but in the reverse direction. However, compared to the push operation, fetching updates provides lower security guarantees. At the time of publication, there exists no mechanism for signing the fetch update in Git, so the receiver cannot verify that the remote is authorized to update the references. Therefore, it is recommended to only use the push operation or integrate an external authorization protocol to mitigate this issue. For optimized replication, the provided pull and push operations can be used with well-known *gossip protocols* [20].

After the pull/fetch step, the received remote frontier is stored in `refs/remotes/<remote-name>/` and it is guaranteed that this update is a fast-forward. However, it is not verified that all references point to correctly signed commits, or that the referenced commit is indeed created by the account specified in the branch name. This validation is performed in a subsequent merge step. In the merge step, the received updates are first verified before updating the local references. Since the commits included in the local frontier are already trusted, only the signature of new commits needs to be verified. Instead of traversing the whole commit history, which would scale poorly for bigger systems, the δ-GOC-Ledger implementation first determines the difference between the local and the received remote frontier in order to identify the new commits that are not yet included in the local frontier. These commits are identified with the `git log` operation, whereby the local and remote references are specified as arguments. This method returns a list of all commits between the two frontiers, which are then validated. After the signature verification process, the remote references are merged with a local `git fetch` operation, which also verifies that the merged updates causally happened after the local frontier. If the signature verification fails or the update is not a fast-forward, the process is canceled to maintain the correctness of the local state. The reception and merge of updates are independent processes that can be deferred. However, in this implementation, the reception of updates will automatically trigger the merge function via Git hook scripts for convenience reasons, instead of requiring the user to explicitly invoke the merge after each update.

This update process ensures that the local state of the replica always remains correct and trusted, but the correctness of the received delta states is not verified. This validation takes place during the creation of checkpoints, discussed in the next section.

5.5 Checkpointing

In order to retrieve the latest full state of an account, all the delta states affecting this account need to be merged. This means, that all commits in the append-only log of this account ID need to be traversed. This implementation uses `git log -first-parent`

`refs/heads/frontier/<token type>/<author ID>` to retrieve a list of all commits between the initialize state and the latest commit associated with this account ID in the frontier. By iterating through this list, each contained delta state is then merged as described in section 3.2.5, resulting in the current full state for this account ID. However, this implies that the whole log must be traversed for each state query, which leads to poor scaling in view of the continuously increasing number of commits.

For this reason, the presented implementation introduces the technique of checkpointing, formally explained in Section 3.2.5. When computing the full state of an account, the resulting state is stored in a local commit that refers to the latest delta state considered by this checkpoint as its parent. Additionally, a reference is stored in `heads/local/checkpoint`, enabling quick access to the computed state. If a full state is requested, it is first checked whether a reference to a checkpoint already exists. If not, a new checkpoint is created by merging all delta accounts between the initialize state and the latest delta state. If a checkpoint exists and the delta state referenced by the checkpoint commit is equal to the latest delta state, the stored checkpoint is up-to-date and represents the latest state of the account. However, if the checkpoint is not pointing to the latest delta state, it needs to be updated. For this, all the delta states between the current checkpoint and the latest delta account of the local frontier are computed by using `git log –first-parent refs/local/checkpoint/<token type>/<author ID>..refs/heads/frontier /<token type>/<author ID>`. These delta states are then subsequently merged with the current checkpoint, resulting in a new checkpoint that represents the current full account state. A new checkpoint commit is generated and the local reference is updated. The previous checkpoints and the corresponding Git objects are outdated and no longer needed. Since these old checkpoints are no longer referenced by any object or reference, they can be safely removed by periodically calling Git's garbage collection (`git prune`), which limits the local storage overhead.

In Figure 5.1, an example of a checkpoint in the commit history is shown. There, $A^{\delta_3}_{green}$ is referenced by the checkpoint commit via the dotted edge, indicating that the included full account state was calculated up to this delta state. The incoming black edge of the checkpoint is the reference stored in `refs/local/checkpoints` to quickly retrieve the latest checkpoint. When an operation requires the latest full state of the green author's account, the system will identify a difference between the latest checkpoint and the frontier. As a result, a new checkpoint will be generated by merging the old checkpoint with $A^{\delta_4}_{green}$ and $A^{\delta_5}_{green}$.

It is assumed that a replica is only interested in a subset of all available accounts, so maintaining the most recent checkpoint for every available author and token type would result in unnecessary overhead. Therefore, checkpoints are computed on-demand, i.e. when a token operation requires the most recent full state of an account. Furthermore, the correctness of the received delta states is not verified upon reception, but rather during the computation of a checkpoint. This validation verifies the following criteria: 1) All intermediary merges always result in a positive balance 2) For every acknowledgment counter, there exists an account state of the sender that causally happened before the acknowledgment with a giveTo counter larger or equal of the number of acknowledged tokens. If those requirements are not

met, the checkpoint process is canceled and an error is returned. This way, the application can rely on the correctness of the checkpoints and only the states required by the replica are computed, which further reduces the computation overhead.

5.6 State-based GOC-Ledger

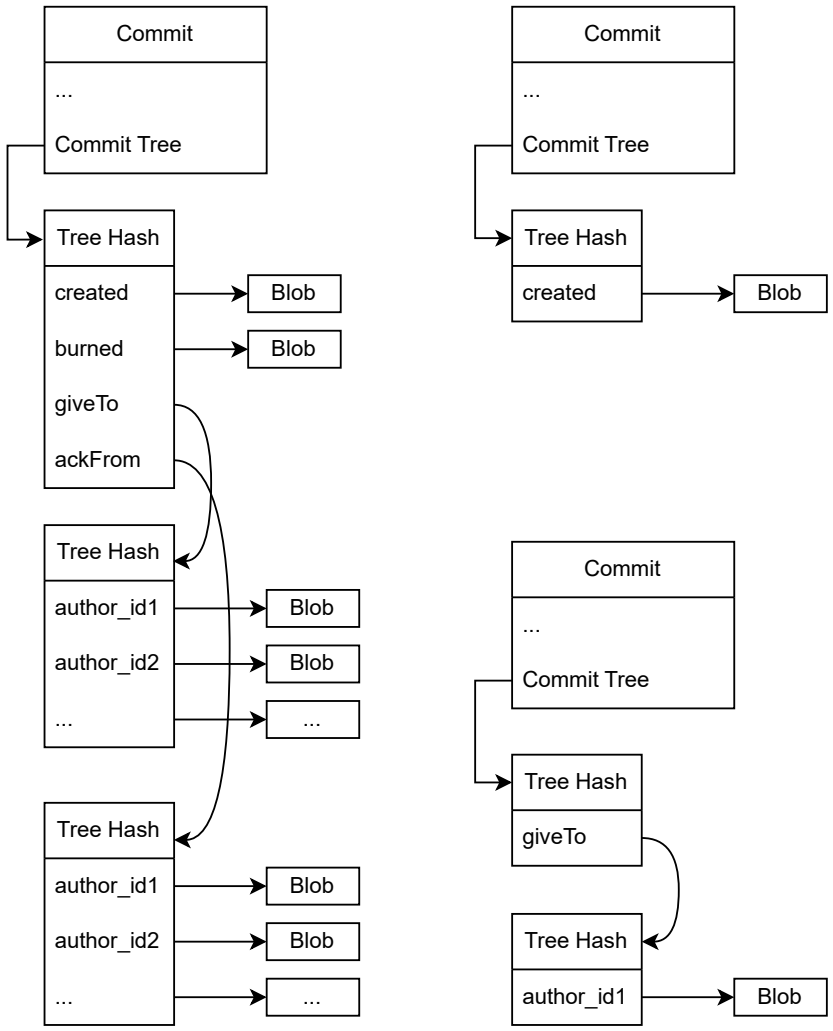
A state-based adaptation of the δ-based implementation was developed to measure the size of the messages generated by the state-based GOC-Ledger. This state-based version is similar to the δ-based implementation and can reuse most of the logic, including the reference system and the replication process. The key difference, however, is that the commits now represent a full state. Therefore, every commit must contain all four attributes of an account state. Consequently, all token operations need to copy the full current state, modify the affected values and then return a full state.

Since the simulation program verifies all transactions during the evaluation and thus creates a trustworthy environment, the state-based implementation assumes that the commit history corresponds to the partial ordering of account states $\leq_{\mathbb{A}}$. Therefore, the latest commit represents the largest state that includes the information of all preceding states. However, an implementation for untrusted systems can be added in a straightforward way by implementing a similar checkpointing strategy as in the δ-based approach (see Section 5.5).

Figure 5.3 illustrates how account states are stored in the state-based and δ-based implementations of the GOC-Ledger. Commit and tree objects are represented as lists, where the header indicates the type of the object and its corresponding hash, while the list entries correspond to the attributes contained in the commit or the entries of the tree object. The pointers depict the references stored as the SHA-1 Hash of the referenced object.

Both versions store a state as a commit, which includes, besides the other commit attributes, a pointer to a tree object that contains the different attributes of this state. However, while in the state-based approach the commit tree always contains an entry for all four account attributes, the delta-states in the δ-based version only consist of fields that were changed by a delta mutator. This means that every commit of a full state includes three tree objects (one for account attributes, one for giveTo and one for ackFrom) and an unlimited number of blobs, since the giveTo and ackFrom trees can refer to counters of an unbounded number of authors. The δ-GOC-Ledger approach only requires one tree object and one blob, as demonstrated by the upper commit in Figure 5.3(b), when the delta state is a result of the $create^{\delta}$ operator. Using the $burn^{\delta}$ mutator results in a similar commit. In the case of $giveTo^{\delta}$ and $ackFrom^{\delta}$, the resulting commit consists of two trees and only one blob, as shown in the lower commit in Figure 5.3(b). Therefore, the size of tree objects in the delta-based implementation has a fixed upper bound, while the tree objects representing a full state are unbounded and larger, especially for accounts that have many keys in the giveTo and ackFrom dictionaries.

In naive implementations, the resulting full account states would be significantly larger than the delta states. However, because of the object-oriented design of Git, not all objects included in the commit tree will be stored again. As explained in Section 2.5, all objects are identified by their SHA-1 Hash. This hash stays the same as long as the content stored



(a) state-based (b) delta-based

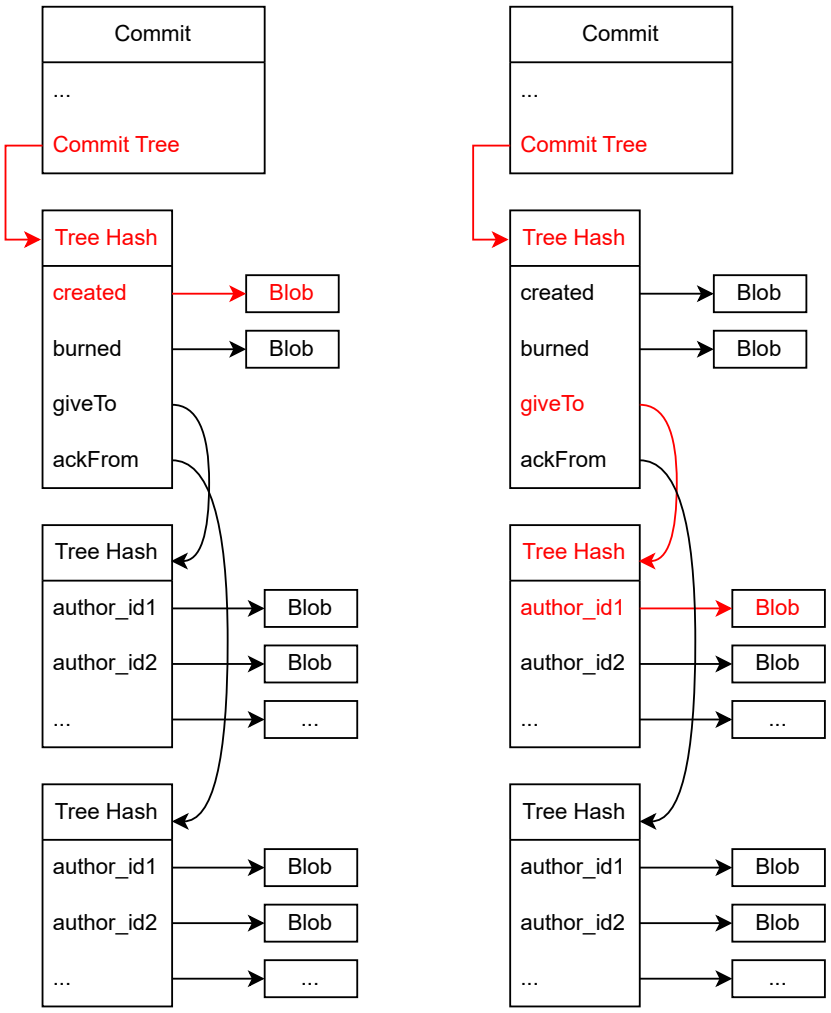
Figure 5.3: State representation in Git for state-based and delta-based GOC-Ledger.

in the tree or blob object remains unchanged, i.e. the dictionary or counter is unmodified. Therefore, a commit tree that points to an unchanged field, will refer to the same hash as the previous state. The object with this hash already exists and thus does not need to be sent again. Figure 5.4 shows how the different token operations affect the underlying Git objects of the resulting state commit. The objects that have changed in comparison to the previous state are highlighted in red. The create operation (Figure 5.4(a)) increases the corresponding counter and thus creates a new blob. The commit tree must also be modified to point to the hash of the generated blob, which leads to a new tree object. Since the other fields are not affected by the *create* operation, the corresponding objects remain unchanged. Therefore, after a *create* operation, the resulting commit includes exactly one new tree object and one new blob object. This also applies to the *burn* mutator. The *giveTo* operation also modifies a blob. This blob is referenced by a tree that maps the author ID to the blob that contains the corresponding counter. In this case, two new tree objects, the commit tree and the tree containing the giveTo entries, and one blob are generated. When

the *ackFrom* operation is performed, it leads to a similar case, where instead of the giveTo tree, the tree containing the references for the ackFrom counters must be generated.

Therefore, the number of modified objects that need to be transmitted is the same as in the delta-state implementation of the GOC-Ledger. This means that the object-based design of Git and the referencing of those via their hash lead to a similar optimization as seen in *Delta-State Replicated Datatypes* for such data structures, where only the updated objects are sent. One major difference, however, is the different size of the tree objects in the state-based and δ-based implementations, as discussed above. When a tree is modified, the newly generated tree must encompass all entries of the previous version in addition to the modified one. Therefore, the δ-based implementation further reduces the communication overhead by using smaller tree objects, especially for accounts with large giveTo and ackFrom dictionaries.

If multiple counters within the same or different account states have the same value, only a single blob containing this integer value is generated and the trees containing these counters reference the same hash multiple times. Hence, the reuse of a blob object for various counter values is another optimization achieved through the design of Git and applies to both state-based and δ-based implementations.



(a) *create* operation (b) *giveTo* operation

Figure 5.4: Modified trees and blobs (in red) after a state update for the state-based GOC-Ledger.

6
Evaluation

In this chapter, we compare the δ-GOC-Ledger and state-based implementation of the GOC-Ledger presented in Chapter 5. The primary goal of the evaluation is to determine to what extent the δ-based approach optimizes the amount of data that needs to be transmitted between replicas. For this reason, we developed a simulator that emulates real ERC-20 [35] transactions using the GOC-Ledger. We compare both implementations along three main dimensions: 1) How much total space is required to replicate or store the full causal history? 2) What is the size of incremental updates transmitted between replicas? 3) How effective are the compression techniques and data representation natively implemented by Git?

First, the methods used to obtain the results are explained in Section 6.1. The results for each dimension are then presented and discussed in Section 6.2.

6.1 Methodology

This section provides an overview of the methodology used for the results. To gain insights that take into account the practical relative distribution of operations and interaction patterns between accounts, real token transactions should be used for the evaluation. However, since there is no dataset for local crypto-tokens because the design has not yet been deployed in practice, we have instead used real-world traces of transactions that follow the ERC-20 token standard [35], which supports similar operations to the GOC-Ledger but instead represent transactions within a single global community.

ERC-20 is a technical standard for fungible tokens implemented as smart contracts on the Ethereum blockchain. In this context, smart contracts [30] are essentially self-executing programs that are deployed on blockchains and define the rules and functionalities of the token. ERC-20 introduces a standardized API for such tokens, including methods for initializing and transferring tokens and for querying the balance of different accounts.

6.1.1 ERC-20 Transactions Dataset

For simulating ERC-20 Tokens transactions, the database provided by Blockchair[35] is used. This dataset keeps a record of all transactions of existing ERC-20 tokens since 2016.

For every single transaction, various data attributes are stored. An overview of those is provided in Table 6.1. Some fields are not used for the simulation such as information about the block of the transaction or the token decimals, which is not directly supported by the implementation.

Dataset field	Type	Description
block_id	int	The position of the block containing the transaction in the Ethereum blockchain.
transaction_hash	string	Cryptographic hash of the block containing the transaction.
time	string	Date of creation of the block containing the transaction.
token_address	string	Unique identifier of the token.
token_name	string	Alias of the token.
token_symbol	string	Symbol of the token.
token_decimals	uint8	The number of decimal places associated with this token.
sender	string	Unique identifier of the sender of the transaction.
recipient	string	Unique identifier of the recipient of the transaction.
value	uint256	The amount of tokens transferred in the transaction.

Table 6.1: Overview of the various fields of the dataset.

Since the dataset contains information on transactions over more than seven years with millions of accounts interacting with each other, replaying all transactions would be computationally prohibitive and beyond the scope intended for the deployment of local crypto-tokens. Therefore, a subset of transactions was chosen and we randomly picked the 27th of April 2017 for simulation, which contains enough data for meaningful insights and viable runtime. On that day, a total of 14782 transactions were performed on 81 different tokens that affected around 8000 accounts.

6.1.2 Simulation of Transactions

To simulate the ERC-20 tokens transactions with the GOC-Legder, a Python script was developed that parses the dataset, translates the transactions into GOC token operations, and finally measures different metrics. First, the dataset is analyzed and different auxiliary data structures are created, containing various information used in the following simulation step. The simulation is decomposed into four different phases:

1. **Author Initialization:** First, all senders and recipients that are involved in the transactions must be initialized. The existing alias system is used to map the generated author public key to the corresponding Ethereum account address.

2. **Token Initialization:** In addition, the various tokens must be initialized as well. Therefore, each token in the data set is initialized in the simulation by the first author who interacts with this token. Similar to the authors, the token alias is used to map the GOC token to its corresponding Ethereum token address.

3. **Token Creation:** In order to simulate all transactions, the sender must have enough tokens to perform the token operations. Since the simulation focuses on a specific day within the dataset, the starting balance for each account is unknown. Therefore, all

accounts begin with zero tokens. Additionally, the ERC-20 token is based on a smart contract that can include the creation of tokens for different accounts during their initialization or define additional functions to create tokens. This information is not contained in the dataset and would require the decompilation and analysis of all smart contracts, which is beyond the scope of this evaluation. To prevent accounts from exceeding their balance when performing transactions, the simulation first calculates the required number of necessary tokens to successfully perform all transactions for that day and creates them before simulating the transactions in the next phase.

4. **Transaction Simulation:** After the previous steps, every account is initialized and has sufficient balance to perform the ERC-20 transactions of the dataset. Every transaction of the selected day is simulated, whereby three different scenarios can arise:

   1) If the ERC-20 transaction is performed between two existing accounts, the sender first performs a giveTo operation, followed by an acknowledgment of the recipient. This approach leads to more acknowledgment operations than required because the recipient could receive tokens in multiple giveTo operations and acknowledge them in a single operation. However, the ERC-20 tokens do not need to be acknowledged, therefore there is no data available when a recipient should acknowledge.

   2) If the transactions in the dataset are sent from the *zero-address* ("0x0", a virtual token address) to an existing account, then this represents a mint operation. This process is not defined in the original ERC-20 specifications, but minting is a method available in many ERC-20 libraries, such as OpenZeppelin [24]. It is very similar to the creation operation of the GOC-Ledger, where accounts that have the right permission can create new tokens. Therefore, a create operation is performed during the simulation, where the recipient of the transaction creates the specified amount of tokens.

   3) If the transaction is sent from an existing account to the *zero-address*, it represents a burn operation. As for the mint operation, the burn operation is an extension of the existing ERC-20 specification. It is similar to the burn operation of the GOC-Ledger and can therefore be mapped in a straightforward way.

At the end of the simulation, the balance of all accounts of the GOC-Ledger is compared to the expected simulated balance to ensure that the simulation was performed successfully. The different assumptions made during the simulation cause the balance of the accounts to differ from the corresponding ERC-20 accounts. However, this difference does not affect the result of the evaluation, since our focus is on comparing the state-based and δ-based versions of the GOC-Ledger and not the comparison between the GOCC-Ledger and the ERC-20 tokens.

6.1.3 Measurements

The ERC-20 dataset described above only contains information about the transactions, but there exists no information on how the data was replicated among all the participants of the system. Therefore, a special version of the presented GOC-ledger implementations

is used for the simulation, which performs token operations of multiple authors within a single repository instead of creating one repository per author. This reduces the simulation time, as there is no need to transfer and merge updates. However, the underlying logic remains unchanged, meaning that all generated states/commits and thus also their sizes are identical to the previously introduced implementations. This allows us to analyze the size of the update message that needs to be transferred between two replicas, independently of specific replication algorithms.

To measure the data size that needs to be transferred between two replicas, the simulation periodically creates bundle files of the current repository state. Since bundle files are compressed by using the same method as used when synchronizing updates with another repository via Git push or fetch, their size reflects the actual amount of data that needs to be transmitted during replication. Every 200 transactions, two bundle files are generated: One bundle file contains the entire ledger commit history, which is transmitted if a new replica is onboarding the system, and also represents the compressed size of only the essential information required to store transactions with our design. The second bundle file contains the difference between the last recorded repository state and the current state, which corresponds to a reasonable estimate of the amount of data that needs to be transmitted to another replica when incrementally reconciling their state. In the absence of real-world deployment data, we estimated the latter to be 200 transactions.

To gain a better insight into the behavior of the underlying Git objects and compression, the tool *git-sizer* [10] is used to measure various repository metrics, including the number and size of different Git objects.

6.2 Results

This section presents and discusses the measurement results obtained from the simulation. First, the size of the bundle file that contains the entire transaction history is analyzed. The incremental update behavior of the GOC-Ledger implementations is then presented, i.e. the data that is transferred periodically to other replicas. Finally, the advantages of Git's object representation and compression are examined.

In each of the following figures, the red area indicates the author and token initialization phase of the simulation (see Section 6.1.2), followed by the blue area representing the token creation phase. After these phases, the simulation executes actual transactions, including create and burn, as well as giveTo and ackFrom operations. For the evaluation, the term token operation refers to all operations defined for the GOC-ledger (*initialize*, *create*, *burn*, *giveTo*, and *ackFrom*) and the initialization of an author.

6.2.1 Total Space for Storage and Full Replication

The following analysis examines the total bundle file size, which contains the entire token history. The graph in Figure 6.1 shows the bundle file size in KB for the state-based and delta-based versions of the GOC-Ledger throughout the entire simulation.

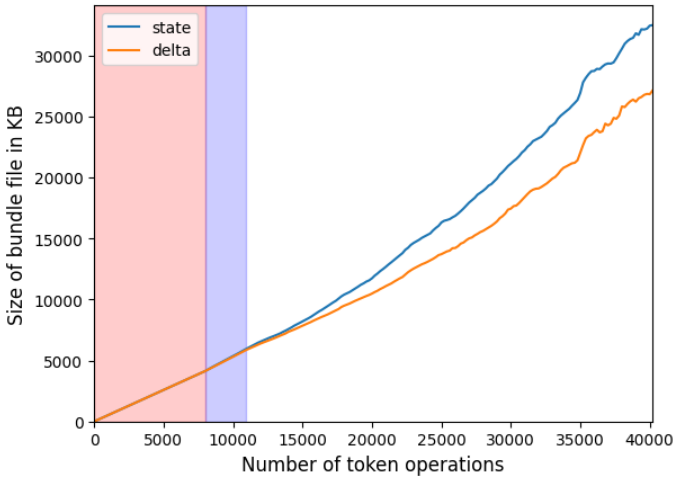


Figure 6.1: Bundle file size.

Overall, the δ-based implementation consistently results in lower or equal bundle file sizes compared to the state-based version. During the initialization phase, the state-based and delta-based versions have identical bundle file sizes, which grow linearly with the number of token operations. In the token creation phase, the state-based bundle file becomes slightly larger than the delta approach, but both versions still follow a relatively linear growth. At the beginning of the transaction phase, the size difference between the two versions is insignificant. However, the delta-based implementation maintains a linear growth in size, while the state-based approach reveals a slightly superlinear increase. This leads to an increasing difference between the bundle file sizes of both versions the more transactions are simulated.

Figure 6.2 shows the total uncompressed size of the different Git objects contained in the bundle file for different numbers of simulated token operations. Since the size of blobs and commits is nearly identical for the state and delta approaches, the measurements of both versions are combined into a single curve for these objects.

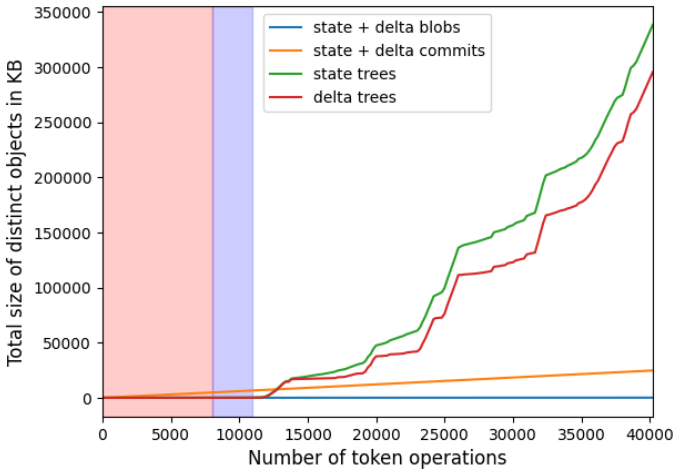
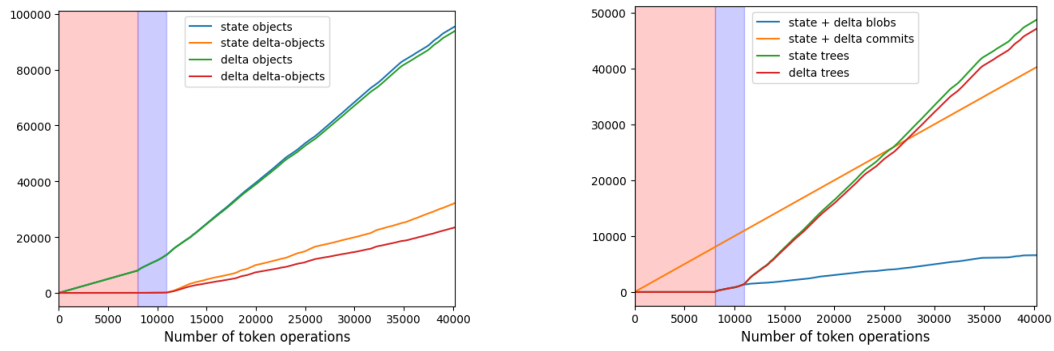


Figure 6.2: Total size of different git objects.

The growth of the blob objects for both versions is barely visible in the figure, as they are relatively small in size compared to the other object types. The size of commits is slightly larger than the blobs and scales linearly with the number of token operations. The most significant difference between the state-based and δ-based versions lies in the size of tree objects. At the beginning, during the initialization and creation phases, both versions are nearly identical. However, during the transaction phase, the total tree size shows a significant growth in size. The growth indicates a quadratic pattern, but further experiments with more transactions are required to confirm this observation. The trees in the delta approach require less space than the state-based variant. This difference does not consistently increase with the number of transactions, although it becomes more significant in comparison to the beginning of the transaction phase.

Figure 6.3 shows the number of Git objects stored in the bundle file for both state-based and delta-based implementations and Figure 6.3(a) provides an overview of the total number of Git objects, as well as the number of objects stored in delta chains during Git's delta-compression. The bundle files of both versions contain almost the same number of Git objects. However, the state-based version shows a slightly faster increase of objects in the transaction phase than the δ-based implementation. During the initialization phase, both versions indicate a linear growth in object count with the number of operations, followed by a steeper linear increase. Git does not employ delta chains in this phase. Only in the transaction phase, an increasing number of objects can be stored in delta chains. The state-based variant employs more Git delta objects compared to the delta-based one. The quantity of delta objects used in both implementations grows linearly with the number of token operations and the difference between the two versions also steadily increases.



(a) Total number of Git objects and delta objects

(b) Number of blob, tree and commit objects

Figure 6.3: Overview of the number of Git objects and deltas in the bundle file

Figure 6.3(b) offers a more detailed look at how the total number of objects is distributed among the three different Git object types used in the implementations. In both versions, the number of blob and commit objects remains identical throughout the simulation and is therefore summarized in a single curve. During the initialization phase, no blob and tree objects are created. As the simulation progresses, the number of blob objects gradually increases with the number of token operations, but is insignificant compared to the other object types. In both implementations, the number of commits follows a linear pattern and

is equal to the number of executed token operations throughout the simulation. The key difference between the state-based and δ-based versions is the number of tree objects. At the start of the simulation, both versions have the same number of tree objects. But during the transaction phase, the bundle file of the state-based implementation contains slightly more tree objects than in the δ-based variant. The deviation increases slightly with the number of executed token operations. While the number of commits dominates at the beginning, tree objects later become more prominent due to their faster growth rate during the simulation of transactions.

6.2.1.1 Discussion

Overall, the results presented above demonstrate that the δ-based implementation consistently generates smaller bundle files than the state-based version. This means that less data needs to be transferred when replicating the entire ledger state to other replicas. Furthermore, the size gain of the δ-based implementation improves as more token operations are performed, indicating superior scaling behavior compared to the state-based version.

In the initialization phase, both versions require a similar amount of data because the author and token aliases are published in a similar way. The only difference is that for the initialization of a token, the state-based version generates a tree object, in which all fields correspond to the default value of a state. However, this tree is generated only once and can be reused (see Section 5.6) for the initialization of all other tokens and therefore does not significantly influence the bundle file size. In this phase, the size of the commit objects dominates because, besides the additional metadata, they contain the alias name as their commit message. In this evaluation, all alias names correspond to Ethereum addresses that have a fixed size of 40 characters (as a hexadecimal representation). Real aliases chosen by users may be shorter and differ in size.

During the token creation phase, a similar behavior is observed. The bundle file size of both versions is nearly identical because the state-based and δ-based implementations generate states, which contain a create counter with the same value, thus creating the same blob objects and a new tree object that refers to this updated blob. In the delta case, this tree only contains one entry, while the one in the state-based approach contains all four state attributes, thus resulting in a slightly larger tree object. Therefore, the bundle file size of the state-based implementation is slightly higher. However, this difference in size is insignificant and barely visible in the presented figures compared to the difference in the transaction phase. It is important to mention that the creation phase, i.e. the execution of subsequent *create* operations without any transaction in between, is not a real-world usage scenario for financial systems, but is required for setting up the required simulation environment. Nevertheless, this demonstrates the behavior of a system in which new initialized tokens are created without any prior transactions.

The key difference between the two approaches becomes evident in the transaction phase of the simulation. During this phase, the state-based implementation requires significantly more data to represent the same ledger state, as shown by the increasing difference in bundle file size (Figure 6.1). By analyzing the total size of the different Git object sizes (Figure 6.2), we can see that only the size of the tree objects affects the difference between both versions,

while the size and number of the other objects are almost identical. Both implementations have the same number of commits, which is equal to the number of token operations, since each token operation leads to a new state that is represented as a commit. Additionally, the sizes of blob objects are the same, because both implementations transmit the same counter values, as explained above. The crucial difference lies in the number and size of the tree objects. As expected, the state-based approach generates more and larger tree objects, since the ackFrom and giveTo trees contain multiple entries in addition to the modified entry, while in the δ-based version these trees contain only one entry, resulting in a smaller object (see Section 5.6).

When comparing the sizes of the different Git objects in Figure 6.2, the size of the tree object has the most significant impact on the size of the overall bundle file, while the sizes of commit and blob objects are almost negligible. This is further highlighted when examining the size of the resulting uncompressed repository in Figure 6.6, where the curve of the tree size in Figure 6.2 is almost proportional to the size of the repository. Furthermore, the blob and commit objects show only sublinear growth, while the total tree object size increases roughly quadratically with the number of token operations. These observations show that the size of the tree objects is the most dominant and thus limiting factor for scalability in both state-based and delta-based implementations. Because the delta-based approach results in smaller tree objects, the scalability is significantly increased in contrast to the state-based approach.

During the creation of bundle files or the synchronization of the current local state with other replicas, Git compresses the objects using Zlib and delta compression. It analyzes the objects and identifies delta objects, i.e. objects that can be expressed as a modification of an already existing object, which are more efficiently stored in delta chains. Figure 6.3(a) shows that the state-based version offers more possibilities for delta compression than the δ-based implementation. Git balances the delta compression's performance overhead and the achieved compression gain [9]. However, understanding which characteristics of the state-based implementation favor the creation of delta chains would require a deeper analysis of the delta compression algorithm of Git, which is beyond the scope of this evaluation.

To summarize, the Git δ-GOC-Ledger implementation requires less data to be transferred between replicas than the state-based approach. The difference between the state-based and δ-based implementations is mainly due to the size of the tree objects, which are generally larger in the state-based approach. Since the number of entries in the trees is bounded in the δ-based implementation, while the number of entries in the state-based version grows on average, our design scales significantly better with the number of executed operations. However, due to the significant growth in size, the tree objects impose a scalability limit for both versions. The analyzed bundle file contains the entire ledger history that is transferred when a new replica is onboarded, but does not represent the behavior when incremental updates are exchanged. This scenario is analyzed in the next section.

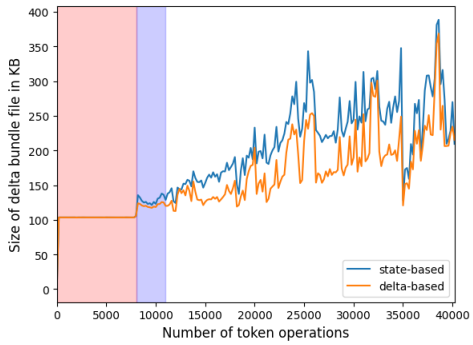


Figure 6.4: Size of the incremental bundle file (containing 200 transactions each).

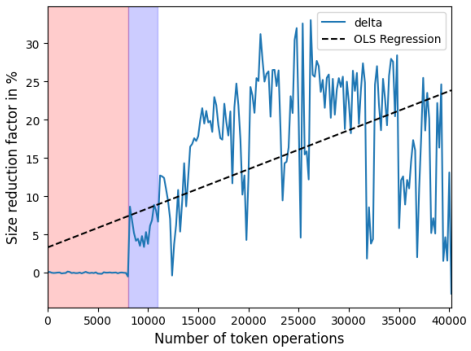


Figure 6.5: Size reduction of the delta-based approach compared to the state-based implementation (higher is better).

6.2.2 Size for Incremental Updates

This section presents the results regarding the amount of data required to be transmitted to a replica that incrementally synchronizes its state every 200 token operations. Figure 6.4 shows the size of the incremental bundle file for both the state-based and δ-based GOC-Ledger implementations. During the initialization phase, both versions result in incremental bundle files of constant and identical size. Then, the bundle file size starts to grow. This increase is not constant, but rather fluctuates. However, as more transactions are performed, the peak of the incremental bundle file size increases. The incremental bundle file of the δ-based implementation consistently remains smaller than the state-based version, with one exception at around 40000 token operations, where the δ-based bundle file is marginally larger. The ratio between the state-based and delta-based incremental bundle file size is shown in Figure 6.5. This indicates how much the bundle file of the delta approach is reduced in contrast to the state-based implementation. In the initialization phase, no size reduction is observed. After this phase, the δ-based version consistently leads to a reduction in size, except for the outlier at 35000 and 40000 token operations, which results in a larger bundle file than the state-based version. The reduction in size fluctuates significantly, where it reaches reduction factors of up to almost 50%. The black dotted line represents the Ordinary Least Squares (OLS) linear regression of the size reduction. It indicates the trend of size reduction of the delta-based approach as the number of executed token operations increases. It can be observed that the file size reduction is steadily growing with the number of token operations.

6.2.2.1 Discussion

In this scenario, it is observed that the delta-based implementation results in overall smaller updates than the state-based version. These findings are consistent with the analysis of the total bundle file containing the entire ledger history (see Section 6.2.1). The identified outlier at 40000 operations may be caused by the compression used by Git to transmit updates. To better understand this phenomenon, further investigation of the compression algorithms is required, but is beyond the scope of this analysis.

Compared to the previous section, this represents a more typical replication case that

occurs during normal ledger operation. In general, the reduction in update size varies considerably and is sometimes only slight, while in other cases it is almost halved. The fluctuation in gain is expected because the evaluation is based on real-world transactions that are not homogeneous. In both implementations of the GOC-Ledger, the size of the resulting update depends heavily on the performed token operation and the accounts involved. As explained in Section 5.6, performing a create or burn operation results in a smaller update than the giveTo or ackFrom operation because only one tree and one blob object are modified. In the latter case, two tree objects and one blob object are generated. Furthermore, the size of the resulting tree objects that represent the giveTo and ackFrom fields of the account state in the state-based approach, highly depends on the account on which the token operation was executed. If the account has already given tokens to or acknowledged tokens from many other different accounts, these trees will contain more entries than an account that has not yet performed any transactions, resulting in larger updates. Additionally, Git does not need to resend already existing objects that can be reused in new account states, which impacts the total update size. This is discussed in Section 6.2.3. Therefore, in scenarios where the update sizes of the state-based and δ-based versions are nearly equal, most of the create or burn operations or giveTo and ackFrom operations are executed on accounts with transactions with few other accounts. The δ-based approach is particularly advantageous when performing giveTo or ackFrom operations on account states that contain numerous entries in the ackFrom or giveTo dictionary. In such cases, there is no need to include all associated account IDs in the tree objects, which leads to a more concise representation of the current state and therefore results in a significant reduction in update size.

The results of the linear regression indicate that the reduction in update size of the delta-based implementation compared to the state-based version increases with the number of executed token operations. This suggests that the delta-based implementation scales significantly better with incremental updates than the state-based version, which is also confirmed by the measurement results of the total bundle file in Section 6.2.1. However, as the coefficient of determination for the linear regression used is only around 0.35, we cannot determine if the curve generated by the regression will maintain the same gradient if additional token operations are performed.

6.2.3 Effectiveness of Git Default Compression and object-based architecture

This section presents the measurement results on the compression and object reuse of Git, analyzing how these properties impact the size of replicated update messages.

Figure 6.6 displays the size of the repository, which only includes uncompressed objects. Only objects that are important for the ledger state are taken into account, but local objects such as checkpoints are not included in the measurements. The repository size of the two versions does not differ during the initialization and token creation phase. But the δ-based implementation results in a smaller overall repository size than the state-based approach when simulating transactions. This difference increases as more token operations are executed.

The size reduction achieved by compressing the Git objects and references into a bundle

Figure 6.6: Repository size with uncompressed objects.

file (see Figure 6.1) instead of storing them uncompressed in the repository is shown in Figure 6.7. At the beginning of the simulation, the size reduction by the bundle file is extremely high. The measurement at zero token operations is not included in this graph, because it results in a reduction of around 13000% and thus this outlier would distort further analysis. Overall, there is always a compression gain in both implementations of the GOC-Ledger. In the initialization and creation phases, the size reduction is smaller. But in the transaction phase, the compression factor increases from 122% at the beginning to almost 600% at the end of the simulation. Until 13000 token operations, the compression used by Git achieves the same size reduction. Between 13000 and 32000, the Git compression then leads to a higher reduction factor for the state-based implementation than for the δ-based version. However, this difference decreases with the number of token operations. At the end of the simulation, starting from 32000 token operations, the δ-based bundle file archives a slightly better compression compared to the state-based file.

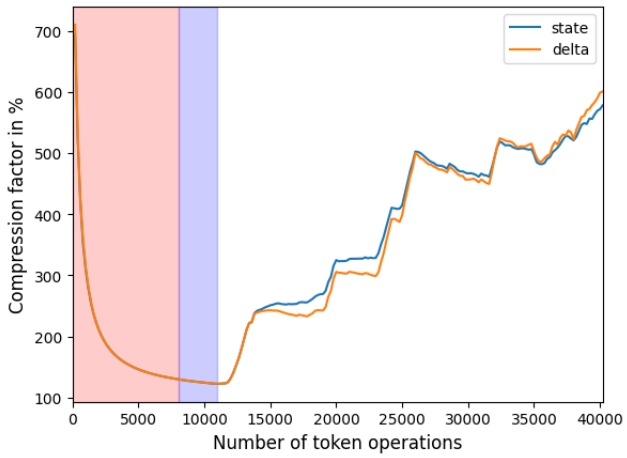
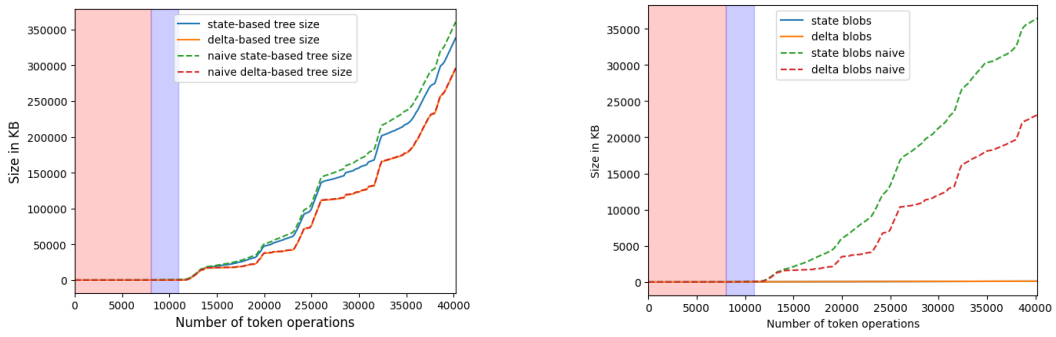


Figure 6.7: Compression factor of Git bundle file.

To investigate to what extent the GOC-Ledger implementations benefit from reusing already existing Git objects from previous states, Figure 6.8 compares the actual total size

of Git objects against a naive approach in which all objects of a new commit must be resent, even if they already exist. Figure 6.8(a) compares the total size of tree objects of both versions against a corresponding naive replication approach. A similar behavior as in Figure 6.2 can be observed. The tree size in the state-based version is always equal to or larger than the tree objects in the delta approach. The naive implementations are always equal to or larger than the actual measured size. In the delta implementation, the naive approach is almost always equal to the actual size throughout the simulation. However, in the state-based implementation, the size of the trees begins to deviate from the naive approach after 14000 token operations. This difference gradually grows as the number of executed token operations increases.

Figure 6.8(b) focuses on the size of the blob objects. The graph clearly shows that the total blob size of the naive approach is considerably larger in both versions. Although in the normal implementation the blob size is negligible compared to the other objects, the total size of blob objects in the naive representation is larger than the naive tree object size. The naive blob size of both state-based and δ-based versions is equal up to around 14000 token operations. After that, however, the two naive versions begin to differ, and the difference grows significantly with the number of token operations. The size of the naive state-based tree grows almost linearly with the number of operations, increasing much faster than the δ-based naive tree size. By the end of the simulation, the δ-based total naive blob size achieves a size reduction of 81% compared to the naive state-based approach.



(a) Tree objects

(b) Blob objects

Figure 6.8: Size of blob and tree objects compared to a naive approach.

6.2.3.1 Discussion

The results indicate that both the state-based and δ-based versions significantly benefit from Git's compression method. Besides the large peak of the compression rate at the beginning, it can be observed that the compression rate achieved during the initialization and creation phases is relatively low compared to the rates achieved during the transaction phase. This means that in the initial phases, when data is mainly stored as commit messages, the compression leads to a comparatively low size reduction (see Figure 6.7). However, when transactions are simulated, the compression rate increases significantly, indicating that data stored in Git objects may be compressed more efficiently. Additionally, Git's compression

of updates that are sent to other replicas also includes delta compression. Figure 6.3(a) shows that Git delta-objects could be identified during the transaction phase, which further increases the compression ratio. Overall, both implementations benefit almost equally from the compression and the compression rate increases with the number of executed token operations. At the end of the simulation, the δ-based implementation begins to benefit slightly more from the compression than the state-based version. But more experiments with additional transactions are necessary to determine whether this trend will continue. The outliers at the beginning of the simulation are caused by the memory overhead of the repository directory structure, which does not affect the compressed bundle file and thus leads to extremely high size reduction factors. However, this overhead is static and therefore becomes insignificant compared to the size of the actual data when more token operations are executed. The efficient compression of Git objects is not only important for reducing the message size transmitted between replicas, but also for the local memory overhead. Git performs periodic maintenance on the local repository, which includes the compression of objects in a similar way to bundle files. This reduces the local memory footprint and improves the performance for accessing Git objects.

In addition to compression, Git offers further advantages, as it reuses exiting objects and only transmits missing objects to other replicas due to its reference system and object-oriented structure. The graphs in Figure 6.8 demonstrate that both the state-based and delta-based GOC-Ledger implementations benefit from this optimization. The impact of reusing tree objects is minimal in the state-based version, while the delta-based implementation shows no difference in the tree size of the naive approach. For a tree to be reused in the delta implementation, either at least two different giveTo operations to the same recipient must result in the same giveTo counter for that identifier, or multiple operations must acknowledge tokens from the same sender, resulting in the same ackFrom counter value for that sender ID. The probability that such scenarios occur in real-world transactions is very low, thus the simulation could not reveal any benefits for the δ-based approach. However, the reuse of blob objects is crucial for reducing the overall state size for both versions. The state-based implementation benefits significantly more from the reuse of blobs than the δ-based implementation. This behavior is expected because the state-based approach contains full states that encompass all different grow-only counters as blobs. As only the hash of the blob that contains the modified counter value changes, it is the only blob object that is sent to other replicas. This shows that Git needs to replicate far fewer blob objects than a naive implementation by referencing already existing objects via their hash. Because all delta states in the implementation only contain exactly one modified counter value, the difference between the naive delta blob size and the actual size indicates that there must exist collisions of counter values between different states. Consequently, different operations result in the same counter value for various fields. The same blob object is thus referenced multiple times, but only needs to be transmitted once. This further reduces the size of update messages that are replicated between replicas.

The total size of blob objects is almost ten times smaller than the size of trees. This means that reusing tree objects has a larger impact on optimizing the message sizes than blobs. However, the difference between the naive tree object size and the tree size of the

implementation is significantly smaller than the difference observed in blob objects. Therefore, Git natively achieves an average message size reduction of 24% in the state-based implementation compared to a naive approach.

To summarize, these observations demonstrate that Git's optimizations, which are based on object compression and reuse of objects, have a significant impact on the overall ledger state size of both the δ-based and state-based implementations. Although the compression rate is almost equal in both versions, the state-based version benefits significantly more from the object-oriented structure of Git. By only referencing the hash of the blob that contains the corresponding counter value, unmodified counter values do not need to be transferred again. This indicates that Git provides a similar optimization to the Delta-State Replicated Data Types for state-based CRDTs, which are implemented in a similar data structure as presented in Section 5.6. However, the tree objects of the state-based implementation still need to reference all blobs for each counter value, resulting in considerably larger tree objects. Therefore, the δ-based implementation still reduces the communication overhead significantly and scales better with the number of token operations.

7
Related Work

In this section, existing research projects and literature related to this thesis are presented and reviewed. To the best of our knowledge and based on the review of *CRDT.tech* [14], which provides a list of research publications on CRDTs, the δ-GOC-Ledger is the first known consensus-free replicated ledger based on Delta-State Replicated Datatypes.

Conflict-free Replicated Data Types is a current method of replicating data among peers that relies on eventual consistency without the need for consensus [28]. The δ-based CRDT represents a novel approach to optimize state-based CRDTs and is still part of active research [2]. In general, there exists significantly fewer applications based on δ-CRDTs than on state- or operation-based CRDTs. Nevertheless, a few industry applications support δ-based CRDTs, such as AKKA, a toolkit that facilitates the development of concurrent, distributed and fault-tolerant applications that implement various known CRDT datatypes [1]. IPFS, a protocol for sharing information in a decentralized way, also supports δ-CRDTs for distributing key-value stores among replicas [3, 12]. Most of the recent research in this area has focused on supporting JSON by combining multiple δ-based CRDTs [4, 25]. But there exists no known consensus-free replicated ledger based on δ-CRDTs. Therefore, the main focus of this thesis lies on the design of δ-based CRDTs in the domain of financial token systems, as well as evaluating its benefits compared to a state-based version.

The implemented append-only log per author and token type is inspired by the feeds used in Secure Scuttlebutt (SSB) [31], a decentralized peer-to-peer communication protocol. In SSB, each author has an append-only log to which new messages can be appended. The properties of the log ensure that messages cannot be subsequently modified by signing new log entries and containing the cryptographic hash of the previous message. In this way, each replica can trust the messages in the author's log up to the latest replicated message.

Δ-CRDT is an extension of δ-CRDT that further optimizes the state size replicated between peers. Usually with the δ-CRDT approach, the replicas need to periodically send their full state, similar to the state-based approach. This ensures that replicas that have not synchronized any updates for a longer period of time still converge, as the new delta states may not include older changes. To optimize this process, in the Δ-CRDTs, all replicas maintain the current version of their local state using a logical clock, called the version vector. During the replication process, the replicas compare each other's version vectors

to determine the minimal delta state that needs to be transmitted. As a result, only this single delta state is then sent to the other replica instead of the entire local state, thereby decreasing the size of the transmitted message. In order to compute this minimal delta state, additional metadata needs to be maintained, which grows over time. Therefore, this data is removed periodically, with the risk that a replica may not be able to send this minimal delta anymore and must resend the full state, as in state-based CRDTs. This approach is similar to the replication process presented in Section 5.4. The references serve as a version vector pointing to the last known local state of the replica, also called frontiers. The parent-child relationship of the commits, stored as a hash graph [13], represents the causal history [26] of the delta states. Git's reconciliation protocol then compares the local frontier with the remote frontier, i.e. computes the difference between both states and only replicates missing commits. Although git compresses these objects as efficiently as possible, unlike the Δ-CRDT, this approach transmits all missing delta states but does not combine them into a single minimal state. A similar approach could be added to the Git implementation of the δ-GOC-Ledger, where the required delta states are merged into a single delta state that represents the minimal needed state between the replicas. However, if only such deltas instead of the entire delta state history were transmitted, the immutable, self-certifying append-only-log per account would no longer be consistent, resulting in an interrupted chain of trust, which would be crucial for financial transactions as in the GOC-Ledger. The same scenario would occur if all data related to the causal history of delta states were removed without additional mechanisms.

At the time of writing, there are no known implementations of consensus-free replicated ledgers using Git, probably because the main use case for Git is to provide a multi-user version control system for files. Additionally, we could not find any implementations of δ-based CRDTs build on Git. But there are some applications that rely on Git and its reconciliation protocol to implement decentralized peer-to-peer systems. One example is Radicle [32], an open-source, peer-to-peer code collaboration stack built on Git. It uses a similar referencing and message signing technique, as described in Section 5.3, to achieve trust through self-certification. Additionally, all modifications are stored as commits, ordered as a directed acyclic graph, similar to the presented Git implementation. This graph is then partially ordered and concurrent states are merged consistently across all replicas, leading to a similar data structure as state-based CRDTs. However, the documentation does not specify the implemented merge function, nor does it validate the properties of this function. In Radicle there exist also possible cases where no merge function can be applied, leading to a conflict that must be resolved by the user, thus contradicting the principle of CRDTs. Our Git implementation of the δ-GOC ledger has instead proven convergence and liveness properties. It also explores how the data structure of a CRDT can be mapped to Git objects and quantifies the efficiency of this approach.

Another application is the Git implementation of the 2P-BFT-Log [17], which provides an eventually consistent append-only log, even in the presence of forks of Byzantine processes. It also implements the properties of self-certifying branches in a similar way as the presented implementation. Additionally, it is one of the first implementations of state-based CRDTs, which are proven to converge. However, the implementation transfers data via commit

messages and does not encode the information as blob or tree objects. The implementations of this thesis, in contrast, represent states as a composition of blobs and trees, which allows us to explore the efficency of representing CRDT states as Git objects. Furthermore, this approach is evaluated to quantify the size of data transmitted between replicas.

8 Conclusion

This thesis introduces the δ-GOC-Ledger, a consensus-free replicated ledger based on Delta-State Replicated Datatypes. This approach reduces the communication overhead when replicating data compared to the original state-based version and at the same time guarantees that eventual convergence is achieved even on unreliable communication channels. We provided a strong formal foundation by proving all properties and showed how the delta-based CRDT results in the same full state as in the state-based version, thereby inheriting the same properties. To compare the two approaches, we built prototypes using Git for both the state-based and δ-based versions and simulated real-world transactions. By analyzing the message sizes during the simulation, we discovered that implementing the state-based GOC-Ledger using Git achieves an average size reduction of 24% compared to a naive implementation. However, implementing the Delta-GOC-Ledger reduces the message sizes by an additional 10-30%. Therefore, the δ-based ledger significantly reduces the message sizes that need to be replicated, especially when the number of transactions grows. In addition, we discovered that Git may be a promising tool for implementing CRDTs with a similar design as the GOC-Ledger due to its compression and reconciliation algorithms. However, we also highlighted the limitations of the data representation using Git tree objects.

Since the δ-GOC-Ledger leads to smaller overall message sizes, it offers a substantial improvement in scalability compared to the existing state-based design, especially for systems with a high number of transactions and participants. Therefore, the δ-GOC-Ledger provides a strong foundation for future practical applications of the GOC-Ledger on real-world peer-to-peer systems.

8.1 Future Work

This thesis provides a foundation for further research. In the following, interesting ideas for future work are presented.

8.1.1 Git performance optimization

The implementation presented is an initial prototype of the δ-GOC-Ledger based on Git, which has the potential to be further optimized. The evaluation of the implementation revealed that the current approach of representing states as a composition of Git tree and blob objects has certain benefits (see Section 6.2.3), but we could also observe that the scalability of the system is limited due to the increasing size of the tree objects. Therefore, other ways of storing state information with Git can be explored and evaluated to overcome this limitation. One approach could be to encode the delta-state information as a Git commit message instead of relying on tree objects. Additionally, the commits contain metadata that is redundant. For example, it is obvious that the commits within one append-only log are created by the same author and signed by the same cryptographic key. Similar optimizations as used in tinySSB [29], where several metadata of a log entry is derived from its context, could be implemented to further reduce the message sizes transmitted between replicas. Further methods for truncating the commit history can also be explored to reduce local memory footprint and facilitate onboarding.

8.1.2 Adversarial Environments

The presented prototype of the δ-GOC-ledger currently does not support adversarial behavior in the form of concurrent delta states from one author, i.e. forking of the author's append-only log of a token type. In such cases, other replicas will stay on the first received branch of the fork and will ignore any updates on other branches. This behavior is also seen in other replicated systems based on append-only logs such as *Secure Scuttlebutt* [31], but in untrusted peer-to-peer systems, such Byzantine behavior cannot be prevented. However, the underlying δ-GOC-Ledger design supports concurrent states. Therefore, the current implementation could be extended by additional methods to detect and handle forking. One approach could be the implementation of the *2P-BFT-Log* [17], which provides eventual consistent append-only logs, even in the presence of forks. However, if a fork in the log is detected, it could cause *fork-based double-spending*. The solution to such scenarios in the context of local crypto-tokens is still an open research question.

Bibliography

[1] akka. Ditributed Data – Akka Documentation. URL https://doc.akka.io/docs/akka/current/typed/distributed-data.html. Retrieved: 17.02.2024.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. Journal of Parallel and Distributed Computing, 111:162–173, 2018. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2017.08.003. URL https://www.sciencedirect.com/science/article/pii/S0743731517302332.

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

[4] Amos Brocco. Melda: A general purpose delta state JSON CRDT. In 9th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2022, pages 1–7. ACM, April 2022. doi: 10.1145/3517209.3524039.

[5] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. white paper, 3(37):2–1, 2014.

[6] Scott Chacon and Ben Straub. Pro git, pages 414–447. Springer Nature, 2014.

[7] Thomas Dohmke. 100 million developers and counting, 2023. URL https://github.blog/2023-01-25-100-million-developers-and-counting/. Retrieved: 08.01.2024.

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

[9] Git. Git - git-pack-objects Documentation. URL https://www.git-scm.com/docs/git-pack-objects. Retrieved: 16.03.2024.

[10] GitHub. git-sizer. URL https://github.com/github/git-sizer. Retrieved: 05.02.2024.

[11] Saurabh Gupta. A non-consensus based decentralized financial transaction processing model with support for efficient auditing. Master thesis, Arizona State University, 2016.

[12] ipfs. go-ds-crdt. URL https://github.com/ipfs/go-ds-crdt. Retrieved: 17.02.2024.

[13] Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In 9th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2022, pages 8–15. ACM, April 2022. doi: 10.1145/3517209.3524042.

[14] Martin Kleppmann, Annette Bieniusa, and Marc Shapiro. CRDT Papers. URL https://crdt.tech/papers.html. Retrieved: 10.03.2023.

[15] Leslie Lamport. How to write a 21 st century proof. Journal of fixed point theory and applications, 11:43–63, 2012.

[16] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System, page 179–196. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450372701. URL https://doi.org/10.1145/3335772.3335934.

[17] Erick Lavoie. 2P-BFT-Log: 2-Phase Single-Author Append-Only Log for Adversarial Environments, 2023. arXiv:2307.08381.

[18] Erick Lavoie. GOC-Ledger: State-based Conflict-Free Replicated Ledger from Grow-Only Counters, 2023. arXiv:2305.16976.

[19] Erick Lavoie and Christian Tschudin. Local Crypto-Tokens for Local Economics. In Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good, DICG '22, page 43–48, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450399289. doi: 10.1145/3565383.3566113. URL https://doi.org/10.1145/3565383.3566113.

[20] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), pages 301–310. IEEE, 2007.

[21] David Mazières and M. Frans Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, EW 8, page 118–125, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 9781450373173. doi: 10.1145/319195.319213. URL https://doi.org/10.1145/319195.319213.

[22] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Decentralized business review, 2008.

[23] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton University Press, 2016.

[24] OpenZeppelin. ERC 20 - OpenZeppelin Docs. URL https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20Mintable. Retrieved: 05.03.2024.

[25] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. DSON: JSON CRDT using delta-mutations for document stores. Proceedings of the VLDB Endowment, 15(5):1053–1065, January 2022. doi: 10.14778/3510397.3510403.

[26] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Distributed computing, 7:149–174, 1994. doi: 10.1007/BF02277859.

[27] Johannes Sedlmeir, Hans Ulrich Buhl, Gilbert Fridgen, and Robert Keller. The energy consumption of blockchain technology: Beyond myth. Business & Information Systems Engineering, 62(6):599–608, 2020.

[28] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, INRIA, July 2011. URL https://hal.inria.fr/inria-00609399.

[29] ssbc/tinySSB. tinySSB - the LoRa descendant of Secure Scuttlebutt. URL https://github.com/ssbc/tinySSB. Retrieved: 13.03.2024.

[30] Nick Szabo. Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16), 18(2):28, 1996.

[31] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications. In Proceedings of the 6th ACM Conference on Information-Centric Networking, ICN '19, page 1–11, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369701. doi: 10.1145/3357150.3357396. URL https://doi.org/10.1145/3357150.3357396.

[32] The Radicle Team. Radicle Protocol Guide. URL https://radicle.xyz/guides/protocol. Retrieved: 18.03.2024.

[33] Linus Torvalds, Junio Hamano, et al. git, 2005. URL https://git-scm.com/. Retrieved: 07.01.2024.

[34] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342964. doi: 10.1145/2911151.2911163. URL https://doi.org/10.1145/2911151.2911163.

[35] Fabian Vogelsteller and Vitalik Buterin. ERC-20: Token Standard, 2015. URL https://eips.ethereum.org/EIPS/eip-20. Retrieved: 22.10.2023.

A
Notation

In addition to the common mathematical conventions, we use the following notation to improve the readability and comprehensibility of our algorithms and proofs.

- A represents a **full account state** as described in [18], which includes several attributes denoted as subscript:
  - $\mathbf{A}_{id}$: the identifier of the account.
  - $\mathbf{A}_{\uparrow}$: the number of tokens created
  - $\mathbf{A}_{\downarrow}$: the amount of tokens burned
  - $\mathbf{A}_{\leftarrow}$: a dictionary, where $\mathrm{A}_{\leftarrow}[id]$ is the number of tokens received by id
  - $\mathbf{A}_{\rightarrow}$: a dictionary, where $\mathrm{A}_{\rightarrow}[id]$ is the number of tokens transferred to id
- A^{δ} represents a **delta account state**, resulting from the application of a delta-mutator on a full account state or by joining several delta states. It can contain the same attributes as full account states and is denoted with the same subscript. When trying to access an attribute that is not included in the delta account state, the default value defined in Table 3.1 is implicitly returned.
- To access a value stored in a **dictionary** D, it can be retrieved by accessing the corresponding key, written as $D[key]$. The set of included keys is denoted as D_*. An empty dictionary is written as $\{\}$ or $D_* = \emptyset$.
- $variable \leftarrow value$: this represents a **variable assignment**. The variable name on the left-hand side is assigned a new value from the expression on the right-hand side.