

Verifying Properties of Delta-GOC-Ledger over BFT-Logs

Bachelor Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Computer Networks
<https://cn.dmi.unibas.ch/>

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

Pascal von Fellenberg
pascal.vonfellenberg@unibas.ch

September 9, 2026

Abstract

We propose a design of the δ -GOC-Ledger [6] over a byzantine fault tolerant log, which is largely based upon the design of 2P-BFT-Log proposed by Lavoie [7]. In doing so, we discovered the need for a new message type and introduced new invariants.

Additionally, we designed and implemented an algorithm that we argue verifies most of these invariants on a given the set of messages. The implementation was done over Git, where we used a more efficient encoding of the data than previous approaches.

The performance of this algorithm and our encoding was evaluated on a basic benchmark of randomly generated transactions. This algorithm was then optimized with what we argue is an optimization that improves the asymptotic running time of the algorithm in the case of the basic benchmark.

Acknowledgements

I thank Prof. Dr. Christian Tschudin and Dr. Erick Lavoie for being able to do this project under their supervision. I especially thank Dr. Erick Lavoie for his very helpful feedback and encouraging words when I got stuck on a problem or had a lack of motivation, and for being allowed to explore a topic as freely as I was.

Table of Contents

1	Introduction	1
2	Background	2
2.1	Append-only logs as Hash-Graphs	2
2.2	2P-BFT-Log	2
2.2.1	Messages	3
2.3	δ -GOC-Ledger	4
2.3.1	Delta Accounts	5
3	Design	6
3.1	Definitions	6
3.2	Choices	7
3.3	Detecting Byzantine Behaviour	7
3.4	Invariants	8
3.4.1	Invariants on message M and corresponding delta account	8
3.4.1.1	On Negative Balance	10
3.4.2	Invariants on Message M inherited from 2P-BFT-Log	10
3.4.3	Invariants related to Byzantine Behaviour	11
3.5	Syntax of Messages	12
3.6	Algorithm for Verification	12
3.6.1	Summary Data Structure	13
3.6.2	Creating the Summary of Message M	13
3.6.3	Verifying the invariants $M1$ to $M7$, and $F1$ to $F3$	14
3.6.4	Verifying the invariants $d1$ to $d7$	16
4	Implementation	19
4.1	Encoding Messages as Git Commits	19
4.2	Structure of Git Repository	20
4.3	Verification Algorithm	20
4.3.1	Summary Cache	21
4.4	Signature Verification	21
5	Evaluation	22
5.1	General Methodology	22

5.1.1	Generating the Benchmark Repository	22
5.1.2	Relevant Specs of System	22
5.2	Performance benchmark	23
5.2.1	Methodology	23
5.2.2	Results	23
5.2.2.1	Discussion	24
5.3	Storage Bench: Comparison of storage methods	24
5.3.1	Methodology	24
5.3.2	Results	25
5.3.2.1	Discussion	25
6	Related Work	26
7	Conclusion	27
7.1	Future Work	27
	Bibliography	29

1

Introduction

In this report we propose a design of the δ -GOC-Ledger [6] over a byzantine fault tolerant log, which is largely based upon the design of 2P-BFT-Log proposed by Lavoie [7]. In doing so, we discovered the need for a new message type (see section 3.2). In this report, the term “invariant” is considered synonymous with “property”.

In our design, we introduced to the best of our knowledge novel invariants (see section 3.4). These invariants are: $d1$ to $d7$, $M8$ and $F1$ to $F4$. To accommodate the new message type, we also adjusted the invariants $M3$ and $M4$ from the design of 2P-BFT-Log.

Additionally, we designed and implemented an algorithm that we argue verifies all of these invariants except for $F4$ on a given the set of messages. The implementation was done over Git, where we used a more efficient encoding of the data that needs to be tracked by the δ -GOC-Ledger by previous approaches [6]. The performance of this algorithm and our efficient encoding was evaluated on a basic benchmark of randomly generated transactions (see section 5). This algorithm was then optimized (see section 4.3.1) with what we argue is an optimization that improves the asymptotic running time of the algorithm (see section 5.2.2) in the case of the basic benchmark.

The implementation of the algorithm and source code of this report can be found on the following repository: <https://github.com/p-vf/verify-goc-ledger>

2

Background

In this section, relevant background to the concepts in this project are provided.

2.1 Append-only logs as Hash-Graphs

An append-only log is a distributed data structure that allows participants to share messages between each other. This is done by having each participant have a totally ordered set of messages that gets replicated to the other participants. This can be implemented using hash-graphs, where each message contains the hash of the previous message (if there is any) of the same author. By using a cryptographic hash, we know the messages cannot have been tampered with, as hash-collisions are computationally hard to find. By cryptographically signing the messages we can guarantee that the correct author created the message.

However one problem that arises is that an adversary E can create two messages that are concurrent to each other by referencing the same previous message and distributing each of those messages to different parts of the network. Once the correct participants have replicated these messages, this results in the set of messages belonging to E no longer being totally ordered, breaking the design of naive implementations of append-only logs. In this scenario we say that the log of E is *forked* and E is an *equivocator* [4].

If the append-only log is implemented as a hash-graph, dependencies between messages of different authors can be modelled by including a set of hashes into a message which represent its immediate dependencies. Such dependencies are necessary to order messages by causality, i.e. if there is a hash in the dependencies of message A belonging to message B, we know the message B must have been created before the message A (see happens-before relationship, eqn. 2.1).

2.2 2P-BFT-Log

2P-BFT-Log is a design that implements an append-only log data structure and behaves just like one as long as there are no forks. As soon as there is evidence of a fork in a log, any further messages of the corresponding author are ignored [7].

Field	Type	Meaning
M_{author}	String (Public Key)	author of the message
M_{prev}	MsgId or $\perp$	previous message from same author, $\perp$ (null) if none.
M_{deps}	Set of MsgId	State of logs from other authors on which the message depends
$M_{payload}$	String	Content of message, in our case representing either an account update or a byzantine acknowledgement (see section 3.5)
$M_{signature}$	String	Signature of the concatenation of the previous fields using the private key corresponding to M_{author} .
M_{date}	Timestamp	Date the message was created at

Table 2.1: Contents of a message according to the design of 2P-BFT-Log

2.2.1 Messages

The design of 2P-BFT-Log consists of several parts. First of all, each message has certain contents, each of which is called a field. The fields of a message and their corresponding type and meaning are listed in Table 2.1.

To identify and reference messages, the function `MsgId` is used. It is the hash derived from the concatenation of the contents of the message, i.e.

$$\text{MsgId}(M) \stackrel{\text{def}}{=} H(M_{author} \oplus M_{prev} \oplus M_{deps} \oplus M_{payload} \oplus M_{signature} \oplus M_{date})$$

where H is a cryptographic hash function.

Additionally there are some relevant relations that are defined on messages. The first one is the *happens-before* relationship and is defined as:

$$M \rightsquigarrow M' \stackrel{\text{def}}{=} \begin{cases} M = M'_{prev} = \perp \\ \text{MsgId}(M) = M'_{prev} \\ \text{MsgId}(M) \in M'_{deps} \\ \exists M'' : M \rightsquigarrow M'' \rightsquigarrow M' \end{cases} \quad (2.1)$$

Additionally, the author-specific happens-before relationship is defined as

$$M \rightsquigarrow^{\log} M' \stackrel{\text{def}}{=} \begin{cases} M = M'_{prev} = \perp \\ \text{MsgId}(M) = M'_{prev} \\ \exists M'' : M \rightsquigarrow^{\log} M'' \rightsquigarrow^{\log} M' \end{cases} \quad (2.2)$$

Based on the happens-before relationship, the partial order on messages ($\leq_{\mathcal{M}}$) is defined:

$$M \leq_{\mathcal{M}} M' \stackrel{\text{def}}{=} M = M' \vee M \rightsquigarrow M'$$

If neither M or M' is smaller than the other, they are called concurrent ($\parallel_{\mathcal{M}}$).

$$M \parallel_{\mathcal{M}} M' \stackrel{\text{def}}{=} M \not\leq_{\mathcal{M}} M' \wedge M' \not\leq_{\mathcal{M}} M$$

We define the strict causal history of message M as the set of messages that happened before message M :

$$\mathcal{H}_{<}(M) \stackrel{def}{=} \{M' : M' \rightsquigarrow M\}$$

Additionally we define the causal history of message M as

$$\mathcal{H}_{\leq}(M) \stackrel{def}{=} \{M' : M' \leq_{\mathcal{M}} M\}$$

If S is a set of messages, the (strict) causal history of S is overloaded and means the (strict) causal history of all the messages in S combined.

$$\mathcal{H}_r(S) \stackrel{def}{=} \bigcup_{M \in S} \mathcal{H}_r(M)$$

Where r is either $\leq$ or $<$.

2.3 δ -GOC-Ledger

The Grow-Only-Counter-Ledger (GOC-Ledger) is a consensus-free data structure that implements a ledger designed to be used to track transactions of crypto-tokens. A ledger needs to track which author owns how many tokens, to be able to check how many tokens they are able to spend in subsequent transactions. An account A is the unit of data that keeps track of this information. Multiple such accounts from different authors form the state of the ledger. More concretely, a ledger L is a mapping from author identities to their corresponding account.

In the case of the GOC-Ledger, this is realized by introducing 5 fields: A_{author} which is the author of this account, $A_{\uparrow}$ and $A_{\downarrow}$ which represent the number of created/destroyed tokens respectively, $A_{\rightarrow}$ and $A_{\leftarrow}$ which represent the number of given/received tokens respectively. $A_{\uparrow}$ and $A_{\downarrow}$ are simple grow-only counters whereas $A_{\rightarrow}$ and $A_{\leftarrow}$ are mappings from author ids to grow-only counters, which represent the number of tokens A_{author} has given to or acknowledged from the author respectively.

For ease of writing, the notation S_* is defined to be the domain of definition of a mapping S . E.g. if L is a ledger, L_* denotes the set of authors that have an account in this ledger.

The balance of an account A is defined as:

$$\text{BALANCE}(A) \stackrel{def}{=} A_{\uparrow} + \left(\sum_{a \in A_{\leftarrow} *} A_{\leftarrow}[a] \right) - A_{\downarrow} - \left(\sum_{a \in A_{\rightarrow} *} A_{\rightarrow}[a] \right)$$

Additionally there is a merge operation, denoted $\sqcup$, with which accounts can be merged. This merge operation is only defined on accounts of the same author and returns the account that contains the maximal values of all the grow only counters in both of the accounts and adds all the grow-only counters that are only in one of the two accounts. Formally, for

accounts A and B , the merged account $C = A \sqcup B$ is defined as:

$$\begin{aligned}
C_{author} &= A_{author} = B_{author} \\
C_{\uparrow} &= \max\{A_{\uparrow}, B_{\uparrow}\} \\
C_{\downarrow} &= \max\{A_{\downarrow}, B_{\downarrow}\} \\
C_{\rightarrow} &= \{x \mapsto A_{\rightarrow}[x] \mid x \in A_{\rightarrow*} \setminus B_{\rightarrow*}\} \\
&\quad \cup \{x \mapsto B_{\rightarrow}[x] \mid x \in B_{\rightarrow*} \setminus A_{\rightarrow*}\} \\
&\quad \cup \{x \mapsto \max\{A_{\rightarrow}[x], B_{\rightarrow}[x]\} \mid x \in B_{\rightarrow*} \cap A_{\rightarrow*}\} \\
C_{\leftarrow} &= \dots (\text{analogous to } C_{\rightarrow})
\end{aligned}$$

Furthermore, there is also a merge operation on a ledger and an account, also denoted $\sqcup$. The ledger and an account get merged by merging the account with the corresponding account in the ledger if the author is present and if not, adding the account to the ledger. Formally, if we have a ledger L and account A , $L' = L \sqcup A$ is defined as:

$$\begin{aligned}
L'[A_{author}] &= \begin{cases} L[A_{author}] \sqcup A & \text{if } A_{author} \in L_* \\ A & \text{otherwise} \end{cases} \\
L'[x] &= L[x] \quad \text{for any } x \in L_* \text{ with } x \neq A_{author}
\end{aligned}$$

Additionally, we define a repeated merge operation over a set of accounts $\mathcal{A} = \{A_1, \dots, A_n\}$ as the ledger after applying the account updates:

$$\bigsqcup_{A \in \mathcal{A}} A \stackrel{def}{=} ((\dots ((\emptyset \sqcup A_1) \sqcup A_2) \sqcup \dots) \sqcup A_{n-1}) \sqcup A_n \quad (2.3)$$

where the inner most left argument, denoted as the empty set, is the empty ledger, and we use the second definition of the $\sqcup$ operator, i.e. where the left argument is a ledger and the right argument an account.

2.3.1 Delta Accounts

To store the state of the ledger more efficiently with regards to space, only the grow-only counters that have changed have to be stored. This is done via delta accounts as proposed by Heisch [6]. A delta account A^δ has the same fields as a normal account, just that the fields $A_{\uparrow}^\delta$ and $A_{\downarrow}^\delta$ evaluate to the default value 0 if they are not stored and the fields $A_{\rightarrow}^\delta$ and $A_{\leftarrow}^\delta$ evaluate to $\emptyset$ when not stored. When merging delta accounts, one can use the merge operation on normal accounts, evaluating missing fields to their default value.

3

Design

In this chapter we present the design of the δ -GOC-Ledger over 2P-BFT-Log, including the choices that were made, the invariants on the design and the algorithm for verifying the invariants.

3.1 Definitions

First, we define some terms that are used in subsequent sections. We start with the signature check predicate, denoted SIGNED of message M . $\text{SIGNED}(M)$ is true iff $M_{\text{signature}}$ is a valid signature belonging to M_{author} .

$\text{SIGNED}_A(\mathcal{M})$ is defined as the set of messages in $\mathcal{M}$ that are created and signed correctly by author A , formally:

$$\text{SIGNED}_A(\mathcal{M}) \stackrel{\text{def}}{=} \{M \in \mathcal{M} \mid \text{SIGNED}(M) \wedge M_{\text{author}} = A\}$$

A fork proof is a set of messages which all have the same previous message. We define fork proofs as a predicate forkProof over a set of messages S .

$$\text{forkProof}(S) \stackrel{\text{def}}{=} \bigwedge_{M, M' \in S} \left(M_{\text{prev}} = M'_{\text{prev}} \wedge M_{\text{author}} = M'_{\text{author}} \right) \wedge |S| \geq 2$$

For ease of writing, we define some functions related to the immediate dependencies of a message. The function deps of message M is defined as the messages in its immediate dependencies.

$$\text{deps}(M) \stackrel{\text{def}}{=} \{M' \mid \text{MsgId}(M') \in M_{\text{deps}}\}$$

Similarly, the function prev of M denotes its previous message.

$$\text{prev}(M) \stackrel{\text{def}}{=} \begin{cases} M' & \text{if there exists } M' : \text{MsgId}(M') = M_{\text{prev}} \\ \perp & \text{otherwise} \end{cases}$$

A message M in our design can have one of two types; either it has the type account (denoted ACCOUNT) or it has the type byzantine acknowledgement (denoted BYZACK). The type is encoded within its payload and retrieved with $\text{MsgType}(M)$. The roles of these

message types is specified in section 3.2. The *acc* is the function that, given a message M , retrieves its account fields encoded in $M_{payload}$, the author identifier from M_{author} (i.e. $(acc(M))_{author} = M_{author}$), and is only defined if $MsgType(M) = ACCOUNT$. How this function is defined varies in each implementation, as the encoding of the account is not defined in the conceptual design of this data structure.

The set $max_A(\mathcal{M})$ is the set of message ids belonging to maximal messages in the set of messages $\mathcal{M}$ that were created by author A , when looking at the author-specific happens-before relationship $(\xrightarrow{\log})$.

$$max_A(\mathcal{M}) \stackrel{def}{=} \{MsgId(M) \mid M \in SIGNED_A(\mathcal{M}) \wedge \forall M' \in SIGNED_A(\mathcal{M}) : M \not\xrightarrow{\log} M'\}$$

There are multiple authors that we need to track of during verification. We combine this information in a frontier. A frontier is a map of author ids to a set of messages and is defined over a set of messages $\mathcal{M}$, written $F(\mathcal{M})$. The reason we map author ids to sets of messages as opposed to single messages is because an author may fork their log, creating multiple maximal elements in their log which we need to keep track of.

$$F(\mathcal{M}) \stackrel{def}{=} \{A \mapsto max_A(\mathcal{M}) \mid \exists M \in \mathcal{M} : A = M_{author} \wedge SIGNED(M)\} \quad (3.1)$$

3.2 Choices

In our design we made some choices. To save space, we only keep track of the necessary dependencies in a message, meaning only the participants of a transaction are allowed to be in the dependencies of the corresponding message. A consequence of this is that if this is the only way to replicate messages, proof of byzantine behaviour can only be shared if a transaction with actual tokens is made. This prompted us to add a second message type, the *BYZACK* message (a normal transaction has message type *ACCOUNT*). This message type is there to point out to other replicas that some of the participants behave badly, and to prevent colluders from being able to keep ignoring evidence of a fork.

3.3 Detecting Byzantine Behaviour

For specifying invariants related to byzantine behaviour, it is necessary to define what we consider byzantine behaviour. This is mostly based on the Blocklace paper by Almeida et al. [4], but was adapted to our design. The function *byz* of a set of messages S is the set of authors exhibiting byzantine behaviour in the set S and is defined as follows:

$$\begin{aligned} byz(S) \stackrel{def}{=} \{a \mid \exists M \in S : \\ & a = M_{author} \\ & \wedge SIGNED(M) \\ & \wedge (\\ & \quad \exists M' \in S : forkProof(\{M, M'\}) \\ & \quad \vee \neg valid(M) \\ &) \} \end{aligned}$$

Where *valid* is a predicate that is true iff the message is syntactically well formed (see section 3.5), and further invariants hold for the message specified in section 3.4. It is important to note that our design of the predicate *valid* does not follow the convention proposed in the design of the Blocklace paper that messages from known byzantine actors are ignored when computing the valid state of a CRDT, see section 3.4.1 for justification.

3.4 Invariants

In this section we introduce the invariants of our design. We start by introducing the invariants related to the account state. Then the modified invariants of the 2P-BFT-Log are listed and then, we introduce the invariants related to byzantine behaviour.

3.4.1 Invariants on message M and corresponding delta account

The following invariants only apply, if $\text{MsgType}(M) = \text{ACCOUNT}$. We decided that delta-accounts may update more than one counter. This was done to open up the possibility to safe space, as multiple updates can be put into one message, splitting the overhead of storing a message among the updates.

Let L_{before} be the state of the ledger before update A^δ was performed. Formally, this can be described as

$$L_{\text{before}} = \bigsqcup_{M' \in H_{\text{acc}}} \text{acc}(M') \quad (3.2)$$

where $H_{\text{acc}} = \{M' \in \mathcal{H}_<(M) \mid \text{MsgType}(M') = \text{ACCOUNT} \wedge \text{valid}(M')\}$ is the set of all delta accounts in the strict causal history of M , and $\bigsqcup$ is the iterated operator that applies the merging operations to the account states, resulting in a ledger (See Equation 2.3).

Since we merge account states no matter if the corresponding authors forked their log, this means that double spending is not prevented nor handled correctly: if a double spending happens, tokens get created out of thin air by the equivocator. This leads to byzantine processes being able to generate an arbitrary number of tokens. However fixing this issue is more involved: if we simply ignore messages that come from byzantine processes, correct replicas that received money from such byzantine replicas lose tokens, which might lead to a negative balance which is not allowed under normal circumstances. An economical design that handles double spending in some way, preventing correct replicas from losing tokens due to invalidated messages is required to fix this issue but is outside the scope of this project. L_{after} represents the ledger after A^δ has been applied to L_{before} , so $L_{\text{after}} = L_{\text{before}} \sqcup A^\delta$. Further, let $A^\delta = \text{acc}(M)$ be the delta account stored in M .

The invariant *d1* ensures that there are no unnecessary GOCs stored in A^δ . An unnecessary GOC is a GOC that doesn't increase the corresponding GOC in L_{before} when merged with it. This means each GOC in A^δ must be strictly greater than the previous value in L_{before} if it is defined.

d1 (minimal): If $M_{author} \in (L_{before})_*$, then all of the following holds, where $A_{old} = L_{before}[M_{author}]$:

- if $A_{\uparrow}^{\delta} \neq 0$, then $(A_{old})_{\uparrow} < A_{\uparrow}^{\delta}$,
- if $A_{\downarrow}^{\delta} \neq 0$, then $(A_{old})_{\downarrow} < A_{\downarrow}^{\delta}$,
- for all $a \in A_{\rightarrow*}^{\delta} \cap (A_{old})_{\rightarrow*}$, $(A_{old})_{\rightarrow}[a] < A_{\rightarrow}^{\delta}[a]$ holds,
- for all $a \in A_{\leftarrow*}^{\delta} \cap (A_{old})_{\leftarrow*}$, $(A_{old})_{\leftarrow}[a] < A_{\leftarrow}^{\delta}[a]$ holds.

The invariant *d5* ensures that at least one of the fields $A_{\rightarrow}^{\delta}$, $A_{\leftarrow}^{\delta}$, $A_{\uparrow}^{\delta}$, $A_{\downarrow}^{\delta}$ is defined, except for the first account message.

d5 (account not empty): If $M_{author} \in (L_{before})_*$, then $A_{\uparrow}^{\delta} \neq 0 \vee A_{\downarrow}^{\delta} \neq 0 \vee A_{\rightarrow}^{\delta} \neq \emptyset \vee A_{\leftarrow}^{\delta} \neq \emptyset$ holds.

The invariant *d2* ensures that accounts don't have negative balance. See section 3.4.1.1 for reasoning.

d2 (non-negative balance): The constraint $\text{BALANCE}(L_{after}[M_{author}]) \geq 0$ holds.

The invariant *d3* ensures that all the tokens that were acknowledged where actually given to the account.

d3 (valid acknowledgements) The constraint $A_{\leftarrow}^{\delta}[x] \leq (L_{before}[x])_{\rightarrow}[M_{author}]$ must hold for all $x \in A_{\leftarrow*}^{\delta}$.

Invariant *d4* ensures that only replicas that are permitted can create tokens. For the purposes of this report, we define the set *Creators* which contains all the authors that are allowed to create tokens.

d4 (valid creator) If $A_{\uparrow}^{\delta} \neq 0$, $A_{author}^{\delta} \in \text{Creators}$.

We want the dependencies that are necessary to track the origins of a transaction. However, we don't want unnecessarily many dependencies. For example if $acc(M)$ is a transaction between authors A , B and C , where A is the creator of the message, we want exactly one message in the immediate dependencies M_{deps} created by author B and exactly one created by C .

A message M' is relevant to message M iff message M acknowledges tokens from M_{author} .

d6 (relevant dependencies): all the immediate dependencies of M are relevant, i.e. for all $M' \in deps(M)$ it holds that $M'_{author} \in acc(M)_{\leftarrow*}$.

One thing to note here is that only the authors in the field $acc(M)_{\leftarrow*}$ get checked. If there is an author in $acc(M)_{\rightarrow*}$ that doesn't exist or has exhibited byzantine behaviour, this is semantically the same as if the tokens given to that non-existent author were destroyed (meaning $acc(M)_\downarrow$ being increased by this amount) and is always allowed. Furthermore, we need to check that there is at least one immediate dependency for each author mentioned in the field $acc(M)_{\leftarrow}$.

d7 (necessary dependencies): for any author $a \in acc(M)_{\leftarrow*}$, there exists a message M' with $M'_{author} = a$ and $MsgId(M') \in M_{deps}$

That there is at most one immediate dependency per author is covered by the invariant $M4$, which is introduced in section 3.4.2.

3.4.1.1 On Negative Balance

For this project, we assume that the predicate *valid* may depend on messages in the causal history from known equivocators to avoid negative balance in valid account messages, which is in contrast to the design described in the Blocklace paper [4], where the valid predicate may only depend on messages in the causal history from authors that don't exhibit byzantine behaviour. From this follows that if the balance of an account is negative, this can only have been caused by either a) the author overspending tokens or b) there being a fork in the causal history of the same account, where the double spending happened. In case a) the message is invalid and in case b) the message can also be viewed as invalid, because it is guaranteed to be created by a byzantine process, which means no message M with $MsgType(M) = ACCOUNT$ should ever reference it anyway. So in either case, a negative balance invalidates a message.

3.4.2 Invariants on Message M inherited from 2P-BFT-Log

- $M1$ (*single previous dependency*): either $M_{prev} = \perp$ or $M_{prev} = MsgId(M') : M'$ exists and is valid.
- $M2$ (*single author*): if $M_{prev} = MsgId(M')$ then $M'_{author} = M_{author}$.
- $M3$ (*valid external dependencies*): for all $msgId \in M_{deps}$, there exists a M' such that $MsgId(M') = msgId$ and $M'_{author} \neq M_{author}$. If $MsgType(M) = ACCOUNT$, $valid(M')$ must be true.
- $M4$ (*single author dependencies*): for any possible *author* not equal to M_{author} , and any $M' \in deps(M)$, the following must hold: If $MsgType(M) = ACCOUNT$, then there must be at most one M' such that $M'_{author} = author$.
- $M5$ (*self-certifying*): $M_{signature}$ is consistent with M_{author} , i.e. $SIGNED(M)$ is true.
- $M6$ (*acyclic*): M_{prev} does not reference a successor of M , i.e. $(M_{prev} \neq M' : M \xrightarrow{log} M')$.
- $M7$ (*monotonic dependencies*): For all M' with $M \xrightarrow{log} M'$, the following must hold: for all $(D, D') \in deps(M) \times deps(M')$ with $D_{author} = D'_{author}$, $D \leq_{\mathcal{M}} D'$ holds.

M8 (non-decreasing dates): if $M_{prev} \neq \perp$, $(prev(M))_{date} \leq M_{date}$ must hold

We adjusted M3 to ignore validity when $MsgType(M) = BYZ_ACK$, as byzantine acknowledgements are allowed to have invalid messages as dependencies (see section 3.4.3). Furthermore, we adjusted M4 to only apply if $MsgType(M) = ACCOUNT$, because if the message is a byzantine acknowledgement, we want to be able to have two dependencies from the same author in case of a fork. We also added the invariant M8 to allow for sorting the messages of one author by date.

3.4.3 Invariants related to Byzantine Behaviour

We want to ensure that authors don't make any transactions with an author that has created messages that exhibit byzantine behaviour. To enforce this, we check that any author that is part of the transaction must not exhibit byzantine behaviour in the set of messages known to author at the time (denoted $\mathcal{H}_{\leq}(M)$). This invariant is only enforced on messages of type *ACCOUNT*.

F1 (No transactions after byzantine message):

If $MsgType(M) = ACCOUNT$, then the following must hold: for any author $a \in byz(\mathcal{H}_{<}(M))$, there must be no message $M' \in deps(M)$ with $M'_{author} = a$.

We also want to ensure that participants don't make transactions with authors that haven't acknowledged the byzantine behaviour present in the message graph. This means that if there is byzantine behaviour in the set of messages reachable from a message, all its immediate dependencies must know about this (see invariant F2). An exception is made for byzantine acknowledgement messages, as - by design - they cannot satisfy this criteria.

F2 (byzantine acknowledgement before transaction):

If $MsgType(M) = ACCOUNT$ then for any message $M' \in deps(M)$,

$$byz(\mathcal{H}_{<}(M)) = byz(\mathcal{H}_{\leq}(M'))$$

must hold.

To ensure that byzantine acknowledgement messages only have necessary immediate dependencies (in the same sense as before), we check that if any of the messages in the immediate dependencies were to be removed, the set of byzantine authors would shrink (see invariant F3).

F3 (necessary dependencies of acknowledgement):

If $MsgType(M) = BYZ_ACK$ then for all $M' \in deps(M)$, the condition

$$byz(\mathcal{H}_{\leq}((deps(M) \cup \{prev(M)\}) \setminus \{M'\})) \subset byz(\mathcal{H}_{<}(M))$$

must hold.

The invariant $F4$ ensures that additionally, any message that is in the dependencies must be a minimal message that proves its author byzantine.

F4 (minimal dependencies of acknowledgement):

If $\text{MsgType}(M) = \text{BYZ_ACK}$ then for all $M' \in \text{deps}(M)$, the condition

$$\text{byz}(\mathcal{H}_{<}(M) \setminus \{M'\}) \subset \text{byz}(\mathcal{H}_{<}(M))$$

must hold.

Our algorithm does not verify the invariant $F4$, but $F3$ which is quite similar, but easier to verify. In $F3$, there is no requirement for minimality of the message M' . This means that any of the messages that get added to the causal history by M' must make a new author byzantine, not just the message M' .

If we don't check $F4$ this allows byzantine colluders to pollute the causal history with arbitrarily many messages after a fork in a log. To show this, we can take a look at an example; If E is a byzantine node and C its colluder, where E has forked their log and created many messages after this on either end. A colluder can then include any of these messages in the causal history with a byzantine acknowledgement message. So long as this acknowledgement is the first message that acknowledges the byzantine behaviour of E , this message is labelled as valid by $F3$. If we look at $F4$, we see that such behaviour would not be allowed. C would have to acknowledge the first message that proves E 's byzantine behaviour. However, colluders may still include arbitrarily many messages into their causal history after a fork by only including messages from one branch of the fork and not acknowledging another branch, or they may keep creating bogus identities that create invalid or concurrent messages which get included by correct replicas as evidence for byzantine behaviour. The latter is analogous to the attack proposed by Lavoie [8].

3.5 Syntax of Messages

For a message M to be well formed, M_{payload} must contain a valid string that encodes either a delta account or a byzantine acknowledgement. The requirements on the syntax are implemented in a predicate *wellFormed* which is true iff its argument is a syntactically well formed message according to the implementation specific encoding that is used. How this is implemented in our case can be seen in Section 4.1.

3.6 Algorithm for Verification

To verify a replica, its messages are visited in an arbitrary ascending topological order according to the relation $\rightsquigarrow$, meaning if $M \rightsquigarrow M'$, M must be visited before M' . Each message gets checked for the invariants and if it is valid, it is added to the set of valid message ids F_{valid} .

The function `VERIFYMESSAGE` takes as argument the message M to be verified, the message cache C_{msg} and the set of already validated messages, F_{valid} and is listed in Algorithm 2.

Algorithm 1 Basic structure of the verifying algorithm

```

1: function VERIFYREPLICA(set of messages  $\mathcal{M}$ )
2:    $F_{valid} \leftarrow \emptyset$  ▷ The set of ids corresponding to valid messages
3:    $C_{msg} \leftarrow \emptyset$  ▷ A mapping from message ids to messages
4:   for all  $M \in \mathcal{M}$  in a topological order do
5:      $C_{msg}[\text{MsgId}(M)] \leftarrow M$ 
6:     if VERIFYMESSAGE( $M, C_{msg}, F_{valid}$ ) then
7:        $F_{valid} \leftarrow F_{valid} \cup \{\text{MsgId}(M)\}$  ▷ add  $M$  as valid message
8:   return  $F_{valid}$ 

```

First off, we check that M is syntactically well formed, i.e. $\text{wellFormed}(M)$ is true if not, we return false. For readability the rest of the function VERIFYMESSAGE was split up into multiple conditions which return false if violated. These conditions are listed in sections 3.6.3 and 3.6.4.

Algorithm 2 Basic structure of the function verifying individual messages

```

function VERIFYMESSAGE(message  $M$ , message cache  $C_{msg}, F_{valid}$ )
  if  $\neg \text{wellFormed}(M)$  then
    return false
  ... ▷ checks that return false if one of  $M1$  to  $M8$ ,  $F1$  to  $F3$  violated (Section 3.6.3)
  if  $\text{MsgType}(M) = \text{ACCOUNT}$  then
    ... ▷ checks that return false if one of  $d1$  to  $d7$  violated (Section 3.6.4)
  return true

```

3.6.1 Summary Data Structure

In the verification algorithm, we use a data structure that we call a summary. The summary is an accumulation of information necessary to verify the invariants $M7$, $F1$, $F2$, $F3$ and the invariants related to the delta account state on message M . A summary S contains the fields listed in Table 3.1¹, and is always defined with respect to a message M , we write $S = \text{CREATESUMMARY}(M)$. An explicit procedure for CREATESUMMARY is given in Algorithm 3.

3.6.2 Creating the Summary of Message M

To create the summary (see Algorithm 3), we first have to initialize the fields and then iterate over all the messages in $\mathcal{H}_<(M)$ in topological order with the condition that their signature is valid. If we focus on the second statement (line 24) and the following in the for-loop, we can see how the frontier is built up over the iterations. The goal is to store the maximal messages of each author in the frontier $\text{Summary}_{\text{frontier}}$. If we assume that L contains the maximal messages up to but not including N , we know that if N_{prev} is not in L , this means N is concurrent to the messages in L , which means we need to add it to L . L gets then stored back in $\text{Summary}_{\text{frontier}}$, which means we build the frontier of $\mathcal{H}_<(M)$ incrementally. If the size of this resulting L is greater than 1, we know the creator of N

¹ The fields S_{account} and S_{received} represent the relevant parts of L_{before} in Equation 3.2

Field	Type	Meaning
$S_{frontier}$	Map of String (author) to Set of Message	Frontier of messages $F(\mathcal{H}_{<}(M))$ as in Equation 3.1. Needed to keep track of what messages M_{author} must have known about before creating M .
$S_{byzantine}$	Set of String (author)	Set of authors that exhibit byzantine behaviour in $\mathcal{H}_{<}(M)$. Needed for the check that M isn't an acknowledgement of tokens from a known byzantine author.
S_{byz_acked}	Set of String (author)	Set of byzantine authors in $\mathcal{H}_{<}(M)$ acknowledged by M_{author} . Needed for the check that if $\text{MsgType}(M) = \text{BYZ_ACK}$, it must only have new byzantine authors in its dependencies.
$S_{account}$	Account	The result of merging all the valid account messages in $\mathcal{H}_{<}(M)$ that belong to M_{author} . Needed for checking non-negative balance and that counter updates are not redundant.
$S_{received}$	Map of String (author) to Number	Maps authors to the number of tokens that were given by them to M_{author} . Needed for checking the validity of any acknowledgement in M if $\text{MsgType}(M) = \text{ACCOUNT}$ and $\text{acc}(M)_{\leftarrow} \neq \emptyset$.

Table 3.1: Fields in a summary object of message M .

must have forked their log, thus N_{author} is added to $\text{Summary}_{byzantine}$.

Now we come to the first part of the loop, which only runs if N is a valid message and is there to update $\text{Summary}_{account}$, $\text{Summary}_{received}$ and $\text{Summary}_{byz_acked}$. If N is an account message we do the following: if N comes from the same author as M , we update $\text{Summary}_{account}$ accordingly, otherwise $\text{Summary}_{received}$ is updated with the number of tokens given to M_{author} (if any).

If N is a byzantine acknowledgement message that comes from the same author as M , we acknowledge all the authors that appear in the dependencies by updating $\text{Summary}_{byz_acked}$.

3.6.3 Verifying the invariants $M1$ to $M7$, and $F1$ to $F3$

The simple invariants were re-expressed in terms of C_{msg} , M and F_{valid} and listed in Table 3.2. Each of the simple conditions corresponds to an if statement that returns false if the condition is met.

The code snippet in Algorithm 4 validates invariant $M4$. If the message is an account message, it checks if an author occurs more than once (see line 5) and if the author of the message occurs in its own dependencies (see line 7) and If either of these conditions is true, the message violates $M4$ so we return false. If the message is a byzantine acknowledgement, it is only checked whether an author occurs more than twice in dependencies. Additionally the mapping *Authors* of authors to their corresponding set of messages in the dependencies is computed, which is used by following code snippets.

To check the invariants $F1$, $F3$ and $M7$ (see Algorithm 5), we have to iterate over the authors

Algorithm 3 Function that creates summary based on a message

```

1: function CREATESUMMARY(message  $M$ , set of messages  $F_{valid}$ , map  $C_{msg}$ )
   Precondition: all messages in  $\mathcal{H}_<(M)$  have been checked, i.e. for any message  $N \in \mathcal{H}_<(M)$ ,  $valid(N) \Leftrightarrow N \in F_{valid}$  holds.
2:    $Summary_{account} \leftarrow \text{INITIALIZEACCOUNT}(M_{author})$ 
3:    $Summary_{received} \leftarrow \emptyset$  ▷ Map of author to number
4:    $Summary_{frontier} \leftarrow \emptyset$  ▷ Map of author to log
5:    $Summary_{byzantine} \leftarrow \emptyset$ 
6:    $Summary_{byz\_acked} \leftarrow \emptyset$ 
7:   for all  $N \in \mathcal{H}_<(M)$  with SIGNED( $N$ ) in topological order do
8:     if  $N \in F_{valid}$  then
9:       if  $MsgType(N) = ACCOUNT$  then
10:         $A^\delta \leftarrow acc(N)$ 
11:        if  $N_{author} = M_{author}$  then
12:           $Summary_{account} \leftarrow Summary_{account} \sqcup A^\delta$ 
13:        else
14:          if  $M_{author} \in A_{\rightarrow*}^\delta$  then
15:            if  $N_{author} \in Summary_{received}$  then
16:               $old \leftarrow Summary_{received}[N_{author}]$ 
17:            else
18:               $old \leftarrow 0$ 
19:             $Summary_{received}[N_{author}] \leftarrow \max(old, A_{\rightarrow}^\delta[M_{author}])$ 
20:          else ▷  $MsgType(N) = BYZ\_ACK$ 
21:            if  $N_{author} = M_{author}$  then
22:              for all  $msgId \in N_{deps}$  do
23:                 $Summary_{byz\_acked} \leftarrow Summary_{byz\_acked} \cup \{(C_{msg}[msgId])_{author}\}$ 
24:             $L \leftarrow \emptyset$ 
25:            if  $N_{author} \in (Summary_{frontier})_*$  then
26:               $L \leftarrow Summary_{frontier}[N_{author}]$ 
27:             $L \leftarrow (L \setminus \{N_{prev}\}) \cup \{MsgId(N)\}$ 
28:            if  $N \notin F_{valid} \vee |L| > 1$  then
29:               $Summary_{byzantine} \leftarrow Summary_{byzantine} \cup \{N_{author}\}$ 
30:             $Summary_{frontier}[N_{author}] \leftarrow L$ 
31:   return  $Summary$ 

```

that are in the immediate dependencies of M , which is $Authors_*$. If M is an account message and one of the authors is byzantine according to the summary of M , then M is invalid (see line 5), because it violates $F1$. If M is a byzantine acknowledgement but the author was either already acknowledged as byzantine, or not byzantine, then this message is invalid as well (see line 8) because it violates $F3$. Furthermore if any of the immediate dependencies is not part of the maximal messages in their corresponding log, then the message is invalid (see line 9) because it violates $M7$.

To verify $F2$, we have to check that every author that exhibited byzantine behaviour (contained in $Summary_{byzantine}$) was also acknowledged (contained in $Summary_{byz_acked}$). If any of the authors in this frontier are byzantine but not acknowledged as such, $F2$ is violated.

Invariant	Violated when
$M1$	$M_{prev} \neq \perp \wedge M_{prev} \notin F_{valid}$
$M2$	$M_{prev} \neq \perp \wedge (C_{msg}[M_{prev}])_{author} \neq M_{author}$
$M3$	$M_{deps} \not\subseteq F_{valid} \wedge \text{MsgType}(M) = \text{ACCOUNT}$ $\vee M_{deps} \not\subseteq (C_{msg})_*$
$M4$	see Algorithm 4
$M5$	$\neg \text{SIGNED}(M)$
$M6$	never, because message graph is a cryptographic hash-graph
$M7, F1 - F4$	see Algorithm 5
$M8$	$(C_{msg}[M_{prev}])_{date} > M_{date}$

Table 3.2: Invariants expressed in terms of C_{msg} , M and F_{valid} **Algorithm 4** Code snippet verifying $M4$

```

...
1:  $Authors \leftarrow \emptyset$  ▷ map of author to set of message ids
2: for all  $msgId \in M_{deps}$  do ▷  $M4$ 
3:    $Author \leftarrow (C_{msg}[msgId])_{author}$ 
4:   if  $\text{MsgType}(M) = \text{ACCOUNT}$  then
5:     if  $Author \in Authors_*$  then
6:       return false
7:     if  $Author = M_{author}$  then
8:       return false
9:   if  $Author \notin Authors_*$  then
10:     $Authors[Author] \leftarrow \emptyset$ 
11:     $Authors[Author] \leftarrow Authors[Author] \cup \{msgId\}$ 
12:    if  $|Authors[Author]| > 2$  then
13:      return false
...

```

3.6.4 Verifying the invariants $d1$ to $d7$

The following code snippets are wrapped inside a big if statement on the condition that the message has message type *ACCOUNT*. We assume that the variables A_{old} , A^δ and A_{new} have been computed as in Algorithm 6, and *Summary* has been computed as in Algorithm 5. A_{old} represents the old account state and is retrieved from *Summary_{account}*. If M is the first message of an author, i.e. the author of this message is not in the frontier *Summary_{frontier}*, we assign the null value to A_{old} to allow the first messages to be treated differently (otherwise A_{old} would just be an empty account). A^δ represents the newly added delta account and A_{new} is the state of the account after applying A^δ .

As before, many invariant checks are simple if-conditions, which are re-expressed in terms of A_{old} , A^δ , A_{new} , C_{msg} and M , and listed in Table 3.3.

To check $d6$ and $d7$ (see Algorithm 7) we create the set of authors *RelevantAuthors* that are relevant to the transaction, which are the authors from which tokens are acknowledged. We then iterate over the authors occurring in the immediate dependencies and in each iteration we do the following: if the author is in the set, we remove it from the set and go on to the next iteration. If the author is not in the set, we know that this author is either not relevant or appeared more than once in the dependencies, in either case the message is invalid.

To check $d7$, i.e. that all necessary dependencies are present, we can check whether the set of authors relevant to the transaction was fully consumed in the preceding loop. If this set

Algorithm 5 Code snippet verifying $M7$, $F1$, $F2$, and $F3$

Precondition: *Authors* has been calculated (as in Algorithm 4)

...

```

1: Summary  $\leftarrow$  CREATESUMMARY(M)
2: for all Author  $\in$  Authors* do
3:   if MsgType(M) = ACCOUNT then
4:     if Author  $\in$  Summarybyzantine  $\vee$  Author  $\notin$  Summarybyzantine then ▷ F1
5:       return false
6:   else ▷ MsgType(M) = BYZ_ACK
7:     if Author  $\in$  Summarybyz_acked then ▷ F3
8:       return false
9:     if Authors[Author]  $\not\subseteq$  Summaryfrontier[Author] then ▷ M7
10:      return false
11: if MsgType(M) = ACCOUNT then ▷ F2
12:   if Summarybyzantine  $\neq$  Summarybyz_acked then
13:     return false

```

...

Algorithm 6 Code snippet assigning relevant account states

Precondition: *Summary* has been computed (see Algorithm 5)

...

```

if Mauthor  $\in$  (Summaryfrontier)* then
  Aold  $\leftarrow$  Summaryaccount
else
  Aold  $\leftarrow \perp$ 
Aδ  $\leftarrow$  acc(M)
Anew  $\leftarrow$  Summaryaccount  $\sqcup$  Aδ

```

...

Invariant	Violated when
$d1$	$A_{old} \neq \perp \wedge ($ $(A_{\uparrow}^{\delta} \neq 0 \wedge A_{\uparrow}^{\delta} \leq (A_{old})_{\uparrow})$ $\vee (A_{\downarrow}^{\delta} \neq 0 \wedge A_{\downarrow}^{\delta} \leq (A_{old})_{\downarrow})$ $\vee (\exists a \in A_{\rightarrow*}^{\delta} : a \in (A_{old})_{\rightarrow*} \wedge A_{\rightarrow}^{\delta}[a] \leq (A_{old})_{\rightarrow}[a])$ $\vee (\exists a \in A_{\leftarrow*}^{\delta} : a \in (A_{old})_{\leftarrow*} \wedge A_{\leftarrow}^{\delta}[a] \leq (A_{old})_{\leftarrow}[a])$ $)$
$d2$	BALANCE(<i>A</i> _{new}) < 0
$d3$	see Algorithm 8
$d4$	$A_{\uparrow}^{\delta} \neq 0 \wedge M_{author} \notin \text{Creators}$
$d5$	$A_{old} \neq 0 \wedge A_{\downarrow}^{\delta} = A_{\uparrow}^{\delta} = 0 \wedge A_{\rightarrow}^{\delta} = A_{\leftarrow}^{\delta} = \emptyset$
$d6, \delta7$	see Algorithm 7

Table 3.3: Invariants expressed in terms of A_{old} , A^{δ} , A_{new} , C_{msg} and M

is not empty, this means an author is missing in the immediate dependencies, thus M must be invalid.

To check $d3$ (see Algorithm 8), we must go through the authors that A^{δ} acknowledges tokens from and check that these authors gave at least the acknowledged number of tokens to the account.

Algorithm 7 Code snippet verifying $d6$ and $d7$

Precondition: *Authors* has been calculated (as in Algorithm 4)

...

$RelevantAuthors \leftarrow A_{\leftarrow}^{\delta}$

for all $a \in Authors_*$ **do** $\triangleright d6$

if $a \notin RelevantAuthors$ **then**

return false

$RelevantAuthors \leftarrow RelevantAuthors \setminus \{a\}$

if $RelevantAuthors \neq \emptyset$ **then** $\triangleright d7$

return false

...

Algorithm 8 Code snippet verifying $d3$

Precondition: *Authors* has been calculated as in Algorithm 4, *Summary* has been calculated as in Algorithm 5.

...

for all $Author \in A_{\leftarrow}^{\delta}$ **do** $\triangleright \delta3$

if $Author \notin (Summary_{received})_* \vee Summary_{received}[Author] < A_{\leftarrow}^{\delta}[Author]$ **then**

return false

...

Algorithm 9 Function that initializes account

1: **function** INITIALIZEACCOUNT(author S)

2: $A_{author} \leftarrow S$

3: $A_{\uparrow} \leftarrow 0$

4: $A_{\downarrow} \leftarrow 0$

5: $A_{\rightarrow} \leftarrow \emptyset$ $\triangleright$ map of author to number

6: $A_{\leftarrow} \leftarrow \emptyset$ $\triangleright$ map of author to number

7: **return** A

4

Implementation

We implemented the verification algorithm in Python. We made this choice because Python is ported to many platforms and because its integer implementation allows for arbitrarily large numbers.

4.1 Encoding Messages as Git Commits

The encoding of a message M as a Git commit can be seen in table 4.1. Now we specify further what $M_{payload}$ must look like.

If the message represents a byzantine acknowledgement, the payload must be empty. If the message represents delta account A^δ , the payload contains an object encoded in the json format [5], where the values representing the fields $A^\delta_{\uparrow}$, $A^\delta_{\downarrow}$, $A^\delta_{\leftarrow}$, and $A^\delta_{\rightarrow}$ are named “c”, “d”, “a”, and “g” respectively. The values of “c” and “d” are positive numbers representing their GOCs and the values of “a” and “g” must be non-empty json objects that contains a member (name/value pair) for each author in $A^\delta_{\leftarrow*}$ and $A^\delta_{\rightarrow*}$ respectively where the name is the author identifier and the value a positive number representing the corresponding GOC. The serialized json object must not contain any newlines, as we retrieve the message content using the subject of the message (format placeholder %s in git) which is newline delimited [1].

Git Commit Field	Purpose
<i>author</i>	M_{author} as <i>author.name</i> , the field <i>author.email</i> is left empty
<i>committer</i>	<i>committer.name</i> contains “_” as git does not allow it to be empty, <i>committer.email</i> is empty
<i>parents</i>	First parent represents M_{prev} , while the rest represent M_{deps} ;
<i>message</i>	contains $M_{payload}$
<i>tree-hash</i>	contains hash of the empty tree
<i>signature</i>	contains $M_{signature}$

Table 4.1: Encoding of messages as git commits

4.2 Structure of Git Repository

The structure of the git repository that contains the data after verification is shown in Figure 4.1. Each invalid commit is referenced by the reference `invalid/<commit-id>` where `<commit-id>` is the invalid commit. Additionally, we store for each author the maximal commit that was checked by the verification algorithm in the reference `<author-id>/validated`, where `<author-id>` represents the identity of the author. Since we use ssh public keys as author identities, there may be sequences like `“//”` in these author identities which are not allowed as part of the references by git, so we actually use the author identities encoded as base64 strings.

```

refs/
├── heads/
│   ├── invalid/
│   │   ├── <commit-id>
│   │   └── ...
│   ├── <author-id>/
│   │   └── validated
│   └── ...
└── ...

```

Figure 4.1: References directory in Git repository.

4.3 Verification Algorithm

To retrieve messages in a reverse topological order, we use the command `git log` with the flags `--reverse` and `--topo-order` and, using format placeholders, we retrieve the relevant fields (see table 4.2). The commits and corresponding delta accounts are cached in a dictionary.

Placeholder	Meaning
%H	Hash (identifier) of commit
%T	Hash of tree object belonging to the commit
%P	Concatenated hashes of parents of commit, separated with spaces
%an, %cn	Author/Committer name respectively
%ae, %ce	Author/Committer email respectively
%at, %ct	Author/Committer date respectively
%G?	Signature validity, replaced with “G”, good signature, “U” for signature with unknown validity. In our case “U” is sufficient, however this does not check if the author of the commit matches the creator of the signature.
%s	Subject line of commit message, terminated by new-line character.

Table 4.2: Relevant format placeholders used with Git, in `--format` option.

4.3.1 Summary Cache

Now instead of always recreating the summary from scratch, we implemented an optimization that uses a mapping from authors to a modified version of their last summaries to reuse the already computed data. Let $S = \text{CREATE_SUMMARY}(M)$ where M is a message, then we compute the modified summary S' that gets added to the cache by adding M to the summary in the same way that the individual messages in the loop of `CREATE\_SUMMARY` get added. This message is then part of the message frontier, i.e. $\text{MsgId}(M) \in S'_{\text{frontier}}[M_{\text{author}}]$.

The modified summaries in the summary cache can then be used to compute $\text{CREATE_SUMMARY}(M')$, where M' is a message, more efficiently, if the modified summary S' in the cache corresponding to author M'_{author} has $S'_{\text{frontier}}[M'_{\text{author}}] = \{M'_{\text{prev}}\}$. This is the case if the previously visited message of author M'_{author} corresponds to M'_{prev} and M'_{author} did not acknowledge a fork of their own.

Once this condition is true, we can compute $\text{CREATE_SUMMARY}(M')$ by only traversing the messages that come after all of the messages in S'_{frontier} . The command to retrieve these messages in git is `git log <m> ^<f1> ^<f2> ... ^<fn>`, where `<fi>` represent the message ids in S'_{frontier} and `<m>` represents $\text{MsgId}(M')$. The impact of this optimization can be seen in the Section 5.2.2.

4.4 Signature Verification

There is no way to get the signer of a commit using a format placeholder in the Git CLI. We can only check that a signature is valid, but not that M_{author} is the creator of $M_{\text{signature}}$ (see %G? in Table 4.2), which makes this check alone useless. To remedy this, we run the `git show` command for each commit that has a valid signature (based on the signature validity retrieved with the format placeholder %G?) with the `--format=raw` option, which prints the raw commit info including the base64 encoding of the signature, from which we can extract the signer and check that it matches M_{author} .

5

Evaluation

When evaluating our design, we focused on two metrics. The first was performance of the verification algorithm and the second was efficiency of storage. Concerning performance, there are two things we are interested in: how is the scaling behaviour of the running time as we increase the number of messages we verify? and how does the optimization mentioned in Section 4.3 affect the runtime? Concerning storage, we compare how much less storage on the hard disk our design uses by storing account data in the message of a commit compared to storing the data in the tree and blob objects as in the delta-GOC-Ledger by Heisch [6].

5.1 General Methodology

5.1.1 Generating the Benchmark Repository

To get a baseline performance and storage measurement, we created a benchmark that contains random transactions between a fixed number of authors, where the number of transactions per author follows a heavy-tailed distribution (a Pareto-distribution in our case), as it often does in real-world scenarios [9]. In the performance benchmark, the number of simulated transactions is varied to indicate asymptotic scaling of the runtime of different parts of the code. Additionally there is an acknowledgement delay, meaning tokens that were given don't get acknowledged immediately. However the messages are assumed to always be replicated instantly, meaning all the participants know about the latest messages when creating a new one. Note that this doesn't mean that messages are totally ordered, since the immediate dependencies of an account message are only of the relevant participants of a transaction.

5.1.2 Relevant Specs of System

The Benchmarks are run on a Lenovo ThinkPad X13 2-in-1 with the specs shown in Table 5.1. One important thing to note when using Windows Subsystem for Linux 2 (WSL2) as a virtual machine is that the performance is degraded significantly when accessing files from a mounted directory [3]. We run the benchmark with associated data in the home directory, where this issue doesn't arise.

OS	Windows Subsystem for Linux version 2.7.10.0 running Ubuntu 24.04.4 LTS with Microsoft Windows 11 Pro as host OS
System Type	x64-based PC
Processor	Intel(R) Core(TM) Ultra 5 125U, 1300 Mhz, 12 Cores, 14 Logical Processors
Installed Physical Memory (RAM)	16.0 GB
Git version	2.43.0

Table 5.1: relevant specs of the hard- and software used to perform benchmarks

5.2 Performance benchmark

5.2.1 Methodology

We generate repositories with the number of messages varying from 300 to 900 with a step size of 300 and the number of authors fixed at 30. To measure the runtime of different parts of the code, we use the `perf_counter` function from the `time` module in the standard library of python [2]. We split the code into various non-overlapping sections that get measured individually and get a label in the plot. The code is then run with and without the optimization mentioned in section 4.3. The benchmarks were only run once, which explains some of the variance in the results.

5.2.2 Results

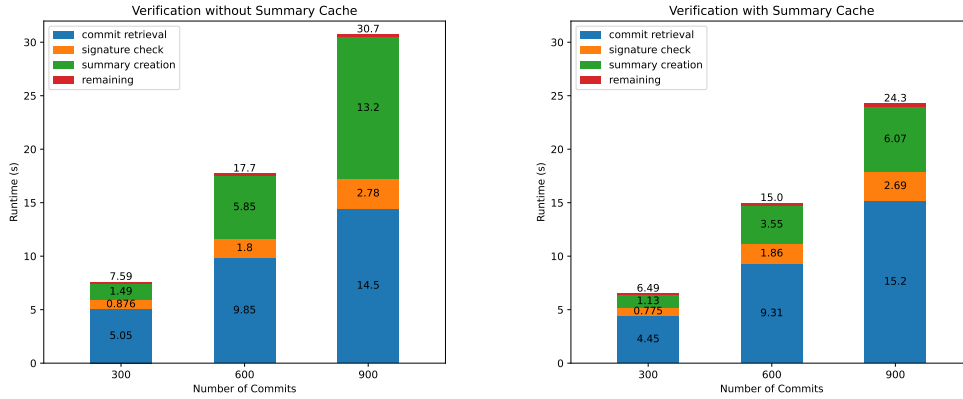


Figure 5.1: Performance measurements on baseline repository with and without the summary cache optimization

The results of the performance benchmark done on the baseline repository are shown in figure 5.1. The sections corresponding to “signature check”, “commit retrieval”, “summary creation” and “remaining” are the total time spent respectively checking the signatures of commits, retrieving all the commits initially, creating the summary and the remaining time not measured by any of the other ones. Most of the remaining time will be taken up by checking the invariants given the summary data structure. If we take a look at the verification algorithm without the summary cache optimization, we see that the total

running time (top of the bars) increases more than proportionally with respect to the number of commits. The section that is responsible for this scaling is mostly “create summary”. However if we take a look at the graph with the summary cache optimization, we see that “create summary” is no longer that big of a part of the total runtime and increases less quickly.

5.2.2.1 Discussion

The reason “summary creation” is the biggest part of the time besides “commit retrieval”, is because we use subprocesses to recreate the summary, which happens for each commit, even with the summary cache optimization.

If we take a look at the graph without the summary cache optimization, we see that the label “summary creation” scales quite quickly. This is because for each commit we recreate the summary from scratch, which means we have to look at the whole causal history, which comprises only of commits that were already visited. This is a lot of unnecessary work and intuitively leads to an asymptotic scaling which is roughly quadratic in terms of the number of commits.

Once the summary cache optimization is applied, we see that this scaling is less aggressive, yet still not proportional to the number of commits. This is not because we occasionally miss the cache: since there are no forks, cache misses are impossible (except for the initial messages of an author). We believe this scaling is this way because the more commits we have visited previously, even if we get a summary cache hit, the more additional commits may need to be traversed to create the summary.

5.3 Storage Bench: Comparison of storage methods

5.3.1 Methodology

We take the simulation described in section 5.1.1 and use it to create repositories with the same encoded transactions, the number of messages being 1000 and the number of authors being 30. The only difference between them is how the messages are stored and how many dependencies each message has.

In the encoding labelled “Message”, the account data is stored as described in 3.5. In the “Tree” encoding, the account data is stored as nested tree and blob objects, as described by Heisch [6]. More concretely json objects in our encoding map to tree objects whereas GOCs are represented as blob objects which contain the bytes that represent the number they store (using the functions `int.to_bytes` and `int.from_bytes` in the standard library of python to encode and decode respectively).

The number of dependencies is also varied. By “Extra Deps.” we mean that a message stores a reference for each other author pointing to the last known message that the creator knew about, as opposed to just the last messages of the authors that appear in the transaction.

To measure the size of the generated repositories, we use the unix command `du` (disk usage) on the corresponding `.git` directory with the flag `-s` to get the total storage size this directory takes up. This value is in the column “Effective Size”. The command `git gc`

Method		Effective Size	
Encoding	Extra Deps.	Compressed	Uncompressed
Tree	Yes	2268	23228
Tree	No	2044	21392
Message	Yes	1860	8624
Message	No	1572	6772

Table 5.2: Result of storage benchmark, numbers in kB.

is used to compress the git repository. We measure each size before and after running this command which correspond to the columns “Uncompressed” and “Compressed” respectively.

5.3.2 Results

When measuring the size of the generated git repositories, we get the result shown in Table 5.2. We see that the compressed versions of the tree encoding are less than 10% of their uncompressed versions, while the compressed versions of the message encoding take up more than 20% (21% and 23%) of the uncompressed ones. Additionally, we see the repositories without the extra dependencies are between about 78% and 92% of the size of their corresponding repository with the extra dependencies. We also see that encoding the data as messages and storing no extra dependencies (our approach) uses about 69% of the space that uses the tree encoding with extra dependencies (proposed by Heisch [6]) uses when looking at the effective size after compressing the files. The difference is bigger if we take a look at the effective size before compressing: then our approach only uses about 29% of the approach proposed by Heisch.

5.3.2.1 Discussion

We see that the space saved by encoding account data in the message of the commit is significant, especially when looking at the uncompressed version of the repositories. This is the case, because there are a lot of extra objects that need to be stored in the case of the tree encoding: for each GOC value, we store a blob which is identified with a 32-byte SHA256 value. This value has to be stored in the tree objects. One advantage that the tree encoding could have is deduplication of equivalent values. However by the time deduplication would be effective, i.e. if the number of bytes stored in them is greater than the identifier (greater than 32 bytes), the numbers stored in the objects are unlikely to be equivalent, so deduplication does not apply.

The extra dependencies don’t take up much of the storage space, however this heavily depends on the number of authors present, so the space savings could be more significant in real-world use cases.

The tree encoding gets compressed more effectively than the message encoding. We think this is the case because git packs the tree and blob files into files without explicitly storing the identifiers of these objects. Still, our encoding is more effective at storing the data in the benchmark, even after compression.

6

Related Work

The design of the delta-GOC-Ledger, as well as of the 2P-BFT-Log was taken from the Masters Thesis of Heisch [6] and the article of Lavoie [7] respectively. What was attempted in this report was to unify the two designs. To do this, we took inspiration from the Blocklace paper by Almeida et al. [4]. Additionally, none of the papers mentioned above described a verification procedure, which we specified and implemented. We also extended the designs by limiting the number of dependencies per transaction and adding a new message type.

7

Conclusion

In this report, we specified new invariants that need to be considered when implementing the δ -GOC-Ledger over a byzantine fault tolerant log such as 2P-BFT-Log and how they interact. We then specified and implemented an algorithm that verifies these invariants. Additionally, we implemented an optimization that reduced the rate of increase of the running time with respect to the number of commits significantly. We also adapted the encoding of the account data in commit objects from the originally proposed design of Heisch and were able to achieve about a 70% reduction in storage space when uncompressed and about a reduction of 30% when compressed, while also being easier to retrieve using the Git CLI.

7.1 Future Work

Considering the design of the algorithm, there are some things to be done; the verification algorithm still has to be extended to include invariant $F4$, which we argue should guarantee byzantine repellance, as defined in the Blocklace paper [4]. Furthermore, the design of our algorithm is not proven to verify the mentioned invariants. Additionally, we have to keep track of an increasing amount of state as more commits get verified (for example C_{msg} and F_{valid}). This could be solved by using some sort of reference counting technique to drop the entries in these sets that don't get referenced by new commits.

Considering the implementation of the algorithm, we think the creation of the summary can be improved by running the commit traversal in the same process as the verification algorithm, rather than calling the Git CLI through the subprocess module of python. This is possible, because at the time of creating a summary, we have all the data that is needed to traverse the commit graph in C_{msg} .

Considering the performance evaluation of our algorithm, we only took a look at a very basic benchmark that does not include byzantine behaviour. To remedy this, a benchmark would have to be created that includes forks and other byzantine behaviour.

Considering the encoding of delta accounts as commit messages, instead of using json serialization, we could use a more efficient custom format.

Considering the design of the invariants, the problem of double spending still persists and has to be solved by designing a system that prevents correct replicas from losing tokens or

tokens being created out of thin air.

Bibliography

- [1] `git/pretty.c` at `d35c5399e3e54ac277bb391fc2f6be3e816d312b` · `git/git` — `github.com`. <https://github.com/git/git/blob/d35c5399e3e54ac277bb391fc2f6be3e816d312b/pretty.c#\{L\}1763>. [Accessed 29-07-2026].
- [2] `time` — Time access and conversions — `docs.python.org`. https://docs.python.org/3/library/time.html#time.perf_counter_ns. [Accessed 14-07-2026].
- [3] `[wsl2] filesystem performance is much slower than wsl1 in /mnt` · Issue #4197 · `microsoft/WSL` — `github.com`. <https://github.com/microsoft/{W}{S}{L}/issues/4197>. [Accessed 14-07-2026].
- [4] Paulo Sérgio Almeida and Ehud Shapiro. The blocklace: A byzantine-repelling and universal conflict-free replicated data type, 2025. DOI: 10.48550/arXiv.2402.08068.
- [5] Douglas Crockford. McKeeman Form — `crockford.com`. <https://www.crockford.com/mckeeman.html>. [Accessed 29-07-2026].
- [6] Jannick Heisch. Delta-goc-ledger: Incremental checkpointing and lower message sizes for grow-only counters ledgers with delta-crdts, 2024. URL https://cn.dmi.unibas.ch/fileadmin/user_upload/redesign-cn-dmi/pubs/theses/master/Heisch-Delta-GOC-Ledger.pdf. [Accessed 24-08-2026].
- [7] Erick Lavoie. 2p-bft-log: 2-phase single-author append-only log for adversarial environments, 2023. DOI: 10.48550/arXiv.2307.08381.
- [8] Erick Lavoie. Memory-exhaustion attack on the blocklace byzantine-repelling conflict-free replicated data type, 2026. DOI: 10.48550/arXiv.2607.15185.
- [9] Dorit Ron and Adi Shamir. *Quantitative Analysis of the Full Bitcoin Transaction Graph*, pages 6–24. 11 2012. ISBN 978-3-642-39883-4. DOI: 10.1007/978-3-642-39884-1_2.