Accountability and Reconfiguration: Self-Healing Lattice Agreement

Luciano Freitas de Souza
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Thibault Rieutord
CEA-List, Université Paris-Saclay, Palaiseau, France

Sara Tucci-Piergiovanni
CEA-List, Université Paris-Saclay, Palaiseau, France

Abstract

An *accountable* distributed system provides means to detect deviations of system components from their expected behavior. It is natural to complement fault detection with a reconfiguration mechanism, so that the system could heal itself, by replacing malfunctioning parts with new ones. In this paper, we describe a framework that can be used to implement a large class of accountable and reconfigurable replicated services. We build atop the fundamental lattice agreement abstraction lying at the core of storage systems and cryptocurrencies.

Our asynchronous implementation of accountable lattice agreement ensures that every violation of consistency is followed by an undeniable evidence of misbehavior of a faulty replica. The system can then be seamlessly reconfigured by evicting faulty replicas, adding new ones and merging inconsistent states. We believe that this paper opens a direction towards asynchronous "self-healing" systems that combine accountability and reconfiguration.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfiguration, accountability, asynchronous, lattice agreement

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.25

Funding Luciano Freitas de Souza was supported by Nomadic Labs, and Petr Kuznetsov by TrustShare Innovation Chair.

1 Introduction

There are two major ways to deal with failures in distributed computing:

Fault-tolerance: we anticipate failures by investing into replication and synchronization, so that the system's correctness is not affected by faulty components.

Accountability: we detect failures *a posteriori* and raise undeniable evidences against faulty components.

Accountability in computing has been proposed for generic distributed systems [18, 19] as a mechanism to detect deviations of system nodes from the algorithms they are assigned with. It has been shown that a large class of deviations of a given process from a given deterministic algorithm can be detected by maintaining a set of *witnesses* that keep track of all *observable* actions of the process and check them against the algorithm [20].

The generic approach can be, however, very expensive in practice and one may look for a more tractable, *application-specific* accountability mechanism. Indeed, instead of pursuing the ambitious goal of detecting deviations from the assigned algorithm, we might want to only care about deviations that violate the specification of the problem the algorithm is trying to solve.

The idea has been successfully employed in the context of Byzantine Consensus [11]. The accountable version of consensus guarantees correctness as long as the number of faulty processes does not exceed some fixed f. But if correctness is violated, e.g., honest processes take different decisions, then at least $f+1$ Byzantine processes are presented with undeniable evidences of misbehavior. This is not surprising: a decision in a typical f-resilient consensus protocol must receive *acknowledgements* from a *quorum* of processes, and any two quorums must have at least $f+1$ processes in common [31]. The fact that two processes took different decisions implies that at least $f+1$ processes in the intersection of the corresponding quorums *equivocated*, i.e., acknowledged conflicting decision values. Assuming that every decision is provided with a cryptographic *certificate* containing the set of signed acknowledgements from a quorum of processes, we can immediately construct a desired evidence. Polygraph [11], a recent accountable Byzantine Consensus protocol, naturally builds upon the classical PBFT protocol [9]. One may ask – okay, we have detected a faulty process, but what should we do next? Ideally, we would like to *reconfigure* the system by evicting the faulty process and *reinitializing* the system state.

Reconfigurable replicated systems [15, 16, 21, 35] allow the users to dynamically update the set of replicas. It has been recently shown that reconfiguration can be implemented in purely *asynchronous* environments [1, 2, 15, 21, 23, 35]. The idea was first applied to (read-write) storage systems [1, 2, 15], and then extended to max-registers [21, 35] and more general *lattice* data types, first in the crash-fault context [23] and then for Byzantine failures [24].

Contribution. In this paper, we propose a framework that can be used to implement a large class of replicated services that are both accountable and reconfigurable. Following recent work on reconfiguration [21, 23, 24], we build atop the fundamental *lattice agreement* abstraction. Lattice agreement [4, 14] (LA) takes arbitrary inputs in a *lattice* (a partially ordered set equipped with a *join* operator) and returns outputs that are (1) joins of the inputs, and (2) ordered with respect to the lattice partial order. The LA abstraction is weaker than consensus and can be implemented in an asynchronous system.

Lattice agreement appears to be a perfect match for both desired features: accountability and reconfiguration. Indeed, a quorum-based LA implementation enables detection of misbehaving parties: as soon as two correct users learn two incomparable values, they also obtain a proof of misbehavior of all replicas that *signed* both values. Furthermore, the very process of reconfiguration can be represented as agreement defined on a lattice of *configurations* [21, 23]. These two observations inspire the design of our system.

We propose an accountable *and* reconfigurable implementation that reaches agreement on a *joint* lattice: an object lattice (defining the current *state* of the replicated object) and a configuration lattice (defining the current *configuration* of the replicas). Assuming that the number of failures is less than half of the system size, our implementation is *alive*. It is also *safe* if only benign (crash) failures occur. Once safety is violated, i.e., two correct users learn two incomparable object states, some Byzantine replicas are inevitably confronted with an undeniable proof of misbehavior. The system is then seamlessly reconfigured by evicting the detected replicas, adding new ones and merging inconsistent states. Once the state is merged, the system comes back to providing safety and liveness, as long as no new replicas exhibits Byzantine behavior. Eventually all Byzantine replicas are detected and the system comes back to maintaining both liveness and safety.

Outdated configurations are harmless. Our system prevents users from accessing outdated configurations with the use of *forward-secure digital signature scheme* [5, 13]. A member of each new configuration is assigned a new secret key. Furthermore, honest members of an old

configuration are expected to destroy their old keys before moving to a new one. Thus, if they are later compromised, they will not be able to serve clients' requests, and the remaining Byzantine replicas will not constitute a quorum.

On Byzantine clients. For simplicity, our solution assumes that service replicas are subject to Byzantine failures, but clients are *benign*: they can only fail by crashing. This assumption has already been made in designs of fault-tolerant storage systems [29]. In our case, it precludes the cases when a Byzantine client brings the system into a compromised configuration or slows down the system by issuing excessive reconfiguration requests. In Appendix A we also describe a *one-shot* version of accountable lattice agreement, without reconfiguration, in which both clients and replicas can be Byzantine. Marrying reconfiguration and accountability in a *long-lived* service that can be accessed by Byzantine clients remains an important challenge. One way to address it is to assume an external *access control* mechanism [36] ensuring that only "authentic" configurations are accepted as inputs to the reconfiguration procedure. We discuss this issue in more detail in Section 6.

Summary. Altogether, we believe that this paper opens a new area of asynchronous "self-healing" systems that combine accountability and reconfiguration. Such a system either preserves safety and liveness or preserves liveness and compensates safety violations with eventual detection of Byzantine replicas. It also exports a reconfiguration interface that allows the clients to replace compromised replicas with new, correct ones. In this paper, we show that both mechanisms, accountability and reconfiguration, can be implemented in a purely asynchronous (in the modern parlance – *responsive*) way.

Road map. The rest of the paper is organized as follows. In Section 2, we introduce our system model. In Section 3, we state the problem of reconfigurable and accountable lattice agreement (RALA) and in Section 4.1, we describe our RALA implementation analysing its correctness. In Section 5, we discuss related work, and in Section 6 we present an overview of possible improvements and interesting open questions. In Appendix A, we present our one-shot accountable lattice agreement (A1LA) that assumes that both clients and replicas can be Byzantine and analyse its correctness.

2 System Model

We assume that the system is asynchronous and that it is composed by a set Π of processes that communicate over reliable message-passing channels exchanging authenticated messages. These processes are split into a set Σ of *replicas* that maintain a *replicated service* and a set Γ of *clients* that use the service. We assume the existence of a global clock with range $\mathbb{N}$, but the processes do not have access to it.

In each run, a process can be: (1) *correct* (C) if it faithfully follows the algorithm it is assigned with, (2) *benign* (B) if it can only deviate from the algorithm by prematurely stopping taking steps of its algorithm, or (3) *malicious* (M) or Byzantine if it skips steps or takes a step not prescribed by its algorithm.

We assume a *forward-secure digital signature scheme* [5, 6, 13, 28]. In the scheme, the public key of a process p is fixed while its secret key sk_t^p evolves with its *timestamp* t, a natural number bounded by a fixed natural parameter T, usually taken sufficiently large (e.g., 2^{64}), to accommodate any possible system lifetime. For any t', $t < t' \leqslant T$, the process can *update its secret key* and obtain $sk_{t'}^p$ from sk_t^p. However, we assume that it is computationally

infeasible to "downgrade" the key to a lower timestamp, from $sk^p_{t'}$ to sk^p_t. In particular, once a process updates its timestamp from t to $t' > t$, and then destroys sk^p_t, it is no longer able to sign messages with timestamp less than t', even if it turns Byzantine later.

More formally, we model a forward-secure signature scheme as an oracle which associates every process p with a timestamp t_p. The oracle provides process p with three operations: (1) $UpdateFSKeys(t)$ sets t_p to t if t is greater than the current value of t_p but less or equal to T (2) $FSSign(m, t)$ returns a signature s for message m and timestamp t, assuming $t \geqslant t_p$; (3) $FSVerify(m, t, s, q)$ returns *true* iff the message m provides a signature s generated by a valid call $FSSign(m, t)$ by process q.

We also make use of a weak broadcast primitive that ensures that once a correct process broadcasts a message, all correct processes eventually receive it, e.g., via a gossip mechanism. Notice that, unlike reliable broadcast [7, 8], we only require the primitive to disseminate messages broadcast by correct processes, not to make them eventually agree on the set of delivered ones.

We assume that all clients are benign. For the sake of simplicity, we assume that once a correct process learns an output, it eventually proposes a new input, and that there are only finitely many correct clients.[1]

3 Reconfigurable and Accountable Lattice Agreement: Specification

A lattice is a partially ordered set where any pair of elements has a unique *join*, or supremum, and a unique *meet*, or infinum. We denote $\mathcal{O}$ the object lattice corresponding to the data type the user wishes to implement using the system (such as a counter, a set or commit-abort) and K the configuration lattice.

A *configuration* κ is a finite set of pairs $(\sigma, inout)|\sigma \in \Sigma, inout \in \{+, -\}$. Intuitively, $(\sigma, +) \in \kappa$ means that σ has been earlier added to the configuration and $(\sigma, -)$ means that σ has been removed from it. We say that a replica σ is a member of κ if $(\sigma, +) \in \kappa$ and $(\sigma, -) \notin \kappa$.

$|\kappa|$ is defined as the cardinality of the set of pairs representing the configuration; $\kappa.excluded$ returns all the replicas excluded from it; $\kappa.included$ that were at some moment included on it; $\kappa.members := \kappa.included \backslash \kappa.excluded$. We only consider *well-formed* configurations κ: $\kappa.excluded \subseteq \kappa.included$ (a replica can be removed only if it has been previously added).

In the *reconfigurable accountable (long-lived) lattice agreement* ($RALA$) abstraction, defined on a product lattice $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times K, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{K})$, a client c_i periodically proposes inputs $(\iota, \kappa)|\iota \in \mathcal{O}, \kappa \in K$ to replicas in Σ and obtains, as output, a value $v \in \mathcal{L}$.

Additionally, the client locally maintains an *accusation set* $\alpha_i = (A, P)$ where $A \subset \Sigma$ is a set of replicas and $P \in \mathcal{P}$ is a *proof* (here $\mathcal{P}$ is the set of proofs). The system provides a Boolean map *verify-proof* : $(2^{\Pi} \times \mathcal{P}) \to \{true, false\}$ that can be used by any process or third party to *verify* a proof. For example, a proof can be a set of messages that, for every replica in $r \in A$, contains one or more messages signed by r that cannot be sent by r in any execution of our algorithm.

When a client c receives an input v from the upper-level application we say that c *proposes* v. When c outputs a value $v \in \mathcal{L}$, we say that c *learns* (or *decides*) v. When c sets its accusation set to (A, P), we say that c *accuses* A *with* P.

[1] Our specification can be easily refined to accommodate infinitely many correct clients under the assumption that the number of *concurrently* proposed values is bounded.

Given a client c_i, let $I^i = \langle(\iota_0^i, \kappa_0^i), (\iota_1^i, \kappa_1^i), \cdots\rangle$ denote the sequence of inputs and $\Upsilon^i = \langle \upsilon_0^i, \upsilon_1^i, \cdots\rangle$ denote the sequence of outputs. If for some client c_i and $k \in \mathbb{N}$, $\kappa_k^i \neq \kappa_{k+1}^i$, i.e., c_i proposes to change the configuration, we say that c_i *issues a reconfiguration request.*

Now a *RALA* system must ensure the following properties:

- **Validity.** Each value υ_k^i, $k \geqslant 0$, learned by a client c_i is a join of the k-prefix of its input sequence and some values from other client's inputs.
- **Completeness.** If a correct client learns a value that is incomparable with a value learnt by another correct client then it eventually accuses some replicas it had not yet accused before.

$$\forall c_i, c_j \in C \cap \Gamma, \forall k, l \in \mathbb{N}, \neg\left(\upsilon_k^i \sqsubseteq \upsilon_l^j \vee \upsilon_l^j \sqsubseteq \upsilon_k^i\right), \text{ where } c_i \text{ learns } \upsilon_k^i \text{ at time } t$$

$$\implies \exists t' > t: \ A^i[t] \subsetneq A^i[t']$$

- **Accusation Stability.** The accusation sets monotonically increase.

$$\forall c_i \in \Gamma, t, \ t' \in \mathbb{N}, t < t' : A^i[t] \subseteq A^i[t']$$

- **Accuracy.** If a client accuses a set of replicas A, then it has a valid proof against each replica in A:

$$\forall c_i \in \Gamma, \forall t \in \mathbb{N}, \textit{verify-proof}(A^i[t], P^i[t])$$

- **Authenticity.** It is computationally infeasible to accuse a benign process, i.e., to construct $P \in \mathcal{P}$ s.t. $\textit{verify-proof}(A, P) = \textit{true}$ and $A \cap B \neq \varnothing$.
- **Agreement.** The correct clients eventually agree on the replicas they accuse.

$$\forall t \in \mathbb{N}, \ \forall c_i, c_j \in \Gamma, \ \exists t' \in \mathbb{N}, \ t' > t: \ A^i[t] \subseteq A^j[t']$$

- **Liveness.** If the system reconfigures only finitely many times, every value proposed by a correct client is eventually included in the value learned by every correct client.

$$\forall c_i \in \Gamma, \forall k \in \mathbb{N}, \forall c_j \in \Gamma, \exists \ell \in \mathbb{N} | \iota_k^i \sqsubseteq \upsilon_l^j$$

A configuration κ is said to be *active* (at a given moment of time t) if (1) it is a join of configurations proposed and learnt by time t, (2) and no other correct process learns a configuration $\kappa' | \kappa \sqsubset \kappa'$ by time t. Liveness guarantees of our algorithm rely upon the following condition:

Configuration availability: For all times t, any configuration that is active at all $t' > t$ contains a majority of correct processes.

This is a conventional assumption in asynchronous reconfigurable systems [1, 23, 35]. The intuition behind it is the following. If an active configuration remains active forever, i.e., it is never superseded, then it should contain enough correct replicas. On the other hand, a once active but later superseded configuration may contain arbitrarily many Byzantine processes: the clients' requests will be served by the new configuration.

Notice that the properties above imply that either the values learnt by correct processes are comparable or eventually some Byzantine replicas are detected. If from some point on, no more Byzantine faults take place, we ensure that all new learnt values are comparable. Our requirement of finite number of reconfigurations is standard in the corresponding literature [2, 23, 35] and, in fact, can be shown to be necessary [34]. In practice, we ensure liveness in "sufficiently long" time intervals without reconfiguration.

Notice that the choice of new configurations to propose is left entirely to the clients, as long as the condition above is satisfied. In Section 6, we discuss possible reconfiguration strategies the clients may want to choose. However, it is important to emphasize that regardless of this strategy, **the system does not allow the accused replicas to affect the system's safety and liveness anymore**.

4 Reconfigurable and Accountable Lattice Agreement: Implementation

4.1 Algorithm

Our RALA implementation is given in Algorithm 2, Algorithm 3, and Algorithm 4. We assume that every method in the algorithms is executed by the process *sequentially*, without being interrupted by other methods of this process. Moreover, we consider that the processes *ignore* accused replicas, messages with invalid signatures and messages whose signatures do not match the configuration content.

Algorithm 1 Example of Verify-proof Operation.

```
operation Verify-Proof(accusation(A,P))
1   foreach Process b ∈ A do
2       let MSG be the union of all messages by b in P
3           Check if every m in MSG has a valid signature continue if not
4           Get all ACKs in MSG and check if they are comparable, continue if not
5           Get all Proposal in MSG and check if they obey the description continue if not
6           Get all Decision in MSG and check if their ACKs hold continue if not
7           return false
8   return true
```

Overview. The clients propose values to the replicas which can either *accept* them by issuing an ACK or *reject* them by issuing a NACK. Once enough responses are gathered by the proposing client, it can accordingly either proceed to learn the value it proposed or to refine its proposal so it contains the missing information replicas raised. If no malicious replica tries to deviate, the values learnt are comparable and no accusations are raised. On the other hand, once a replica induces clients to learn incomparable values it is eventually detected and an accusation against it is produced.

The following definitions and boolean map are used in the algorithm and proofs of correctness:

▶ **Definition 1** (S satisfies configurations). *Let S be a set of replicas, κ a configuration, and D a set of configurations. We say that S* satisfies *D* upon *κ iff, for each $d \subseteq D$, S contains a majority of replicas in each configuration in the set $\kappa \cup \cup d$.*

▶ **Definition 2** (Pending Configurations). *A configurations is called* pending *as long as a client has received it but has not yet included it in the most recent decided configuration (lines 38 and 44). This set is comprised of the current client proposal, as well as configurations coming from ACKs (line 13 and 24).*

The map $verify\_maj : (\Sigma \times K \times 2^K) \to \{true, false\}$ where $verify\_maj(S, \kappa, D) = true$ iff S satisfies D upon κ. This map is used to indicate that the client gathered all the responses it needed.

Ledgers. Every client maintains a local *ledger*, called *ackL*, reserved to keep track of signed ACK messages the client received and their senders. Also, clients and replicas maintain two more ledgers to register the values introduced in the system by their origin processes called *objL* and *confL*. By indexing a ledger l by a process p ($l[p]$), one can recover all the values signed by p present in l.

Algorithm 2 Reconfigurable Accountable Lattice Agreement: Code for client c part 1.

Local variables:
status, initially waiting { Boolean indicating status: waiting or proposing }
dest, initially $\varnothing$ { Set of replicas that must be contacted }
nackBool, initially false { Flag indicating whether a NACK has already been received or not }
activePropNb, initially -1 { Index of the current active proposal }
activeOutNb, initially 0 { Index of the next value to be learnt }
propV, initially $\perp$ { Value currently being proposed }
objL, initially empty { Ledger matching object values in the system to their original proposer }
confL, initially containing *Initial Conf* signed by c { Analogous to objL for configurations }
ackL, initially empty { Ledger matching acks to the replicas that issued them }
pendConf, initially $\varnothing$ { Set of pending configurations }
RESPSet, initially $\varnothing$ { Set of replicas that responded }
lastDec, initially $(\perp, intialConfig)$ { Last decided value }
Input:
inBuffer { Values received by the client from an external source to insert in the system }
Outputs:
outV, initially $\perp$ { Array of values learnt by the client }
accusation, initially $\varnothing$ { Set of accusations issued by the client }

upon *status* = *waiting* AND *inBuffer* $\neq \perp$
9 *extract* and *sign* objects from *inBuffer* and *include* them to *objL*
10 *extract* and *sign* configurations from *inBuffer* and *include* them to *confL*
11 Propose

operation *Propose*
12 $propV := extractLedger(objL, confL, c)$
13 include $propV.conf$ to *pendConfSet*
14 $status := proposing$
15 $activePropNb := activePropNb + 1$
16 clear *ackL[activeOutNb]*
17 clear *RESPSet*
18 $nackBool := false$
19 $dest := propV.conf.included - lastDec.conf.excluded$
20 *multicast* $\langle PROPOSAL, (objL, confL, lastDec, activePropNb)\rangle$ to replicas in *dest*

upon $verify\_maj(RESPSet, lastDec.conf, pendConf) = true$
21 **if** $nackBool = true$ **then** Propose **else** Decide

upon *receive* $\langle ACK, (HASH(propV), lastDec, pendConf', activePropNb)\rangle$
from replica r AND $status = proposing$ AND $r \notin ackL$ AND $r \in dest$
22 **if** $propV.conf \in pedingConf'$ **then**
23 append r's ACK message to $ackL[activeOutNb]$
24 include elements from $pendConf'$ which aren't subset of $lastDec.conf$ in *pendConfSet*
25 append r to *RESPSet*
26 **else** $\langle ACCUSATION, (accusation)\rangle$
27 include (r, ACK) to *accusation*
28 *broadcast* $\langle ACCUSATION, (accusation)\rangle$

Issuing a proposal. A client starts in a *waiting* status and listens for values in its *inBuffer* to include them in a new proposal (lines 9 and 10), not taking any values from the buffer while the executing a *proposal*. Additionally, it must also listen for decisions made by other clients (lines 45 and 46) including them in its proposal, preventing malicious replicas from keeping values from it. It then proceeds to multicast its $propV$ to the replicas that might satisfy the pending configurations (variable *dest*) it has seen upon the last decided configuration it came by (line 20) and waits for them to respond.

Algorithm 3 Reconfigurable Accountable Lattice Agreement: Code for client c part 2.

upon *receive* $\langle NACK, (HASH(propV), \Delta objL', \Delta confL', activePropNb)\rangle$ from replica r
AND $status = proposing$ AND $r \in dest$
29 $nackV := extractLedger(\Delta objL', \Delta confL', r)$
30 **if** $nackV \sqsubseteq propV$ **return**
31 $objL := objL \cup objL'$
32 $confL := confL \cup confL'$
33 $nackBool := true$
34 append r to $RESPSet$

operation *Decide*
35 $outV[activeOutNb] := propV$
36 *broadcast* $\langle DECISION, (objL, confL, ackL[activeOutNb])\rangle$
37 $lastDec := outV[activeOutNb]$
38 $pendConfSet := \varnothing$
39 $activeOutNb := activeOutNb + 1$
40 $status := waiting$

upon *receive* $\langle DECISION, (objL', confL', ackL')\rangle$ from client c'
41 $outV' := extractLeger(objL', confL')$
42 $lastDecOld := lastDec$
43 $lastDec := lastDec \cup outV'$
44 Eliminate from *pendConf* subsets of *lastDec.conf*
45 $objL := objL' \cup objL$
46 $confL := confL' \cup confL$
47 $\forall i \mid outV' \not\sqsubseteq outV[i]$ && $outV[i] \not\sqsubseteq outV'$
48 **let** $M = \{m | m \in ackL[i]$ && $m \in ackL'$ && $m \notin accusation\}$
49 **foreach** $m \in M$ **do** include $(m, \{ackL[m], ackL'[m]\})$ to $accusation$
50 **if** $|M| > 0$ **then** *broadcast* $\langle ACCUSATION, (accusation)\rangle$
51 **if** $lastDec.conf \not\sqsubseteq lastDecOld.conf \vee outV' \not\sqsubseteq propV$ **then** Propose

operation *extractLedger* $(objL', confL', sender)$
52 **if** $\exists$ *process* $p \in objL'$ or $confL'$ with invalid signature **then**
53 $accusation := accusation \cup \{(sender, getMSG(objL') \cup getMSG(confL')\}$
54 *broadcast* $\langle ACCUSATION, (accusation)\rangle$
55 **return** $\varnothing$
56 **let** $receivedValue = (\sqcup[v | \exists p, objL'[p] = v], \sqcup[c | \exists p, confL'[p] = c])$
57 **return** $receivedValue$

upon *receive* $\langle ACCUSATION, (accusation')\rangle$ from client q
58 $\Delta Proof := \varnothing$
59 **foreach** process b accused in *accusation'* with p and who isn't present in $accusation$ **do**
60 include (b, p) in $\Delta Proof$
61 **if** $\Delta Proof \neq \varnothing$ **then**
62 $accusation := accusation \cup \Delta Proof$

Treating Client Proposals. The replicas that receive the proposal extract the value from the ledger (line 65). This makes use of the operation *extractLedger* which verifies that all the values came from existing clients, making these values valid. Each replica then checks whether the new proposal contains the join values it has already seen proposed ($repV$), in which case they ack it (line 73) or not, in which case they nack it (line 76), sending a complement to the ledger allowing the client to update its proposal. Benign replicas always forward their keys, destroying the old ones in the process, before responding to clients (line 72).

A replica cannot provide a client with outdated information because the timestamp used in the signature of its messages is only valid if it has been forwarded to the content it proposes and cannot be rolled back. Moreover, if a benign replica sees that a client isn't aware of a decision it has already come by, it will ignore the client proposal until it includes newer information (line 63).

The function getMSG (line 53) takes a set of input values and returns the set of proposals or NACK messages that originally contained them.

Algorithm 4 Reconfigurable Accountable Lattice Agreement: Code for replica r.

Local variables:
objL, initially empty { Object Ledger }
confL, initially empty { Configuration Ledger }
$repV$ initially $\bot$ { Value held by replica }
pendConf, initially $\varnothing$ { Pending Configurations }
lastDec, initially $\bot$
signature timestamp t_r initially $|Initial\ Configuration|$

sign all outgoing messages m with $FSSign(m, t_r)$

upon *receive* $\langle PROPOSAL, (objL', confL', lastDec', activePropNb')\rangle$ from client c
63 **if** $lastDec' \sqsubseteq lastDec$ **then return**
64 $lastDec := lastDec \cup lastDec'$
65 $propV' := extractLedger(objL', confL')$
66 $objL := objL \cup objL'$
67 $confL := confL \cup confL'$
68 **if** $repV \sqsubseteq propV'$ **then**
69 $repV := propV'$
70 Include $propV'.conf$ to *pendConf*
71 $t_r := |repV.conf|$
72 $UpdateFSKeysDestroyOld(t_r)$
73 send $\langle ACK, (HASH(propV'), lastDec, pendConf, activePropNb')\rangle$ to c
74 **else**
75 $repV := repV \sqcup propV'$
76 send $\langle NACK, (HASH(propV'), objL - objL', confL - confL', activePropNb')\rangle$ to c

upon *receive* $\langle DECISION, (objL', confL', ackL')\rangle$ from client c
77 $lastDec := lastDec \cup extractLedger\ (objL', confL')$
78 Eliminate from *pendConf* subsets of $lastDec.conf$
79 $objL := objL' \cup objL$
80 $confL := confL' \cup confL$

operation $extractLedger\ (objL', confL')$
81 **let** $receivedValue = (\sqcup[v|\exists p, objL'[p] = v], \sqcup[c|\exists p, confL'[p] = c])$
82 **return** $receivedValue$

Treating Replica Responses. Once the client gets an ACK from a replica, it includes the message in its ackL (line 23) and registers the replica in its response set (line 25). Upon reception of a NACK, a client complements its *objL* and *confL* (lines 31 and 32) and sets the NACK bool, including the replica in its response set (lines 33 and 34). When a client sees that it has gathered responses from a set of replicas that satisfies the pending configurations, it proceeds to check its NACK bool, as the presence of a NACK means that it cannot decide yet, and if it is false, then it will decide (line 21).

Each proposal gets a unique number (*activePropNb*) so clients consider only reactions to the active proposal, ignoring late messages they might receive. Clients also ignore messages coming from replicas they already accused, as well as messages signed using timestamps that do not correspond to the configuration in their contents.

A client either waits until it gets responses from a set of replicas (keeping track via *RESPSet*) that satisfies the pending configurations or until it gets a newer decided configuration from another client broadcast. It is necessary to get majorities in all those combinations of system configurations because the client doesn't know if any combination of them was learnt by another client and must be sure that it has reached all possible active configurations that can be learnt before it learns one by itself. Furthermore, the state transfer from one replica to another will be directly provided by this procedure, as once a client learns a configuration the object information is already in place, which is one of the advantages of this solution. It becomes then necessary to keep track of the state of the system by including information about which was the last combination of decisions seen (variable *lastDec*), as well as pending configurations (variable *pendConf*).

Issuing and Treating Convictions. We keep an array of all output values instead of just the current one, as well as their corresponding ack ledgers, indexing the currently active entry by *activeOutNb*. This is necessary in order to monitor that after long delays in the network when two correct clients re-establish their connection they can still check if in this period their decisions were comparable (line 47) and be able to accuse processes that lead them to this incomparable state. The clients broadcast their accusations as well as their decisions.

They avoid issuing redundant accusations by keeping track of the variations (line 61). If a process gets new misbehavior proofs, it includes them on its accusations (line 62). Algorithm 1 shows one possible implementation of the *verify_proof* function, checking that every issued accusation was made after a process tried to forge some signature, issued incoherent ACKs, tried to input values in the system in discordance with the specification or decided something without gathering the necessary acknowledgements.

Once a replica is accused by a client, the client begins to ignore the replica and the underlining application can for instance issue a reconfiguration effectively replacing it by one or more new replicas. The clients will then eventually learn comparable values once they join their values and the malicious replicas that try to subvert the system have been accused.

4.2 Correctness

We claim that the system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 solves RALA.

▶ **Definition 3.** *Let's define a state s as being the value of the variable propV. A state s is considered as decided when the first client c in state s broadcasts its decision at line 36. Moreover, we define s.lastDec and s.pendConf as being the value of these variables on the client c at the moment of the state decision. Finally, s.confComb is defined as the set* $\{s.lastDec.conf \cup \{\cup d\} | d \in 2^{s.pendConf}\}$

▶ **Definition 4.** *We define then a graph G_s whose vertices are the different decided states of the system plus the state $(\bot, initConf)$ and whose edges exist between two vertices s and s' whenever the following is true:*

$$s \to s' \Leftrightarrow s \sqsubsetneq s' \wedge s.conf \in s'.confComb$$

▶ **Lemma 5.** *At every benign replica, the variables repV, pendConf and lastDec are monotonically increasing.*

Proof. The variable *repV* is updated in line 69 where it is assigned a new value which has passed the test in line 68, so the new value contains the old one.

The other updates involving these variables in lines 64, 70 and 75 are joins where one of the operands is the old value, so the new values must contain the old ones. ◀

▶ **Lemma 6.** *Given decided states $\overline{s}$, s and s' in G_s, if $\overline{s} \to s$, $\overline{s} \to s'$, $s'.lastDec \sqsubseteq s$, $s.lastDec \sqsubseteq s'$ then either there is an edge between s and s' or there is an accusation.*

Proof. From $\overline{s} \to s$, $\overline{s} \to s'$ we derive that:

$$\overline{s}.conf \in s.confComb \wedge \overline{s}.conf \in s'.confComb \implies \overline{s}.conf \in s.confComb \cap s'.confComb$$

Because the decision only happens after triggering the event that begins in line 21, then it must be that the clients who decided these states got responses from replicas forming majority quorums in $\overline{s}.conf$ and they must therefore intersect in at least a replica r.

Let us assume for now that r followed the algorithm and behaved correctly. Let c_s be the client that decided s and $c_{s'}$ be the client that decided s'. Assuming w.l.o.g that the replica r served the client c_s before, using lemma 5 and observing that s and s' correspond to the first decision of these values, $s \sqsubsetneq s'$. Since s' passed the test in line 63 in replica r, it means that $s.lastDec \sqsubseteq s'.lastDec$, moreover because we assume that $s'.lastDec \sqsubseteq s$ we can write:

$$s.conf = \bigsqcup(\{s.lastDec.conf\} \cup s.pendConf) \sqsubseteq \bigsqcup(\{s'.lastDec.conf\} \cup s.pendConf)$$

$$\sqsubseteq \bigsqcup(\{s.conf\} \cup s.pendConf) = s.conf$$

Furthermore, we see that all pending configurations in s which weren't included by the last decided configuration in s' must also be pending in s' because this information will be carried by the ack from replica r (line 24). We can then conclude:

$$s.conf = \bigsqcup(\{s'.lastDec.conf\} \cup s.pendConf)$$

$$= \bigsqcup(\{s'.lastDec.conf\} \cup \{u \in s.pendConf, u \not\sqsubseteq s'.lastDec.conf\})$$

$$\in \bigsqcup(\{s'.lastDec.conf\} \cup C | C \sqsubseteq s'.pendConf) = s'.confComb$$

Hence there is an edge from s to s' in G_s in this scenario.

If the replica r didn't follow the algorithm and issued incomparable acks, then this event shall be detected and r will be accused. r's ack would be included in both clients c_s and $c_{s'}$ *ackLs* being broadcasted together with the decision in line 36 and once the first client who decided and then received the other's decision go through the line 49, it would find r in both ledgers and accuse it. ◀

▶ **Lemma 7.** *All the infinite connected components of G_s have the same suffix.*

Proof. Because we assume that the system reconfigures finitely many times, there is a point where all the decisions regarding the different configurations the system passed, which were broadcasted in line 36, arrive at the recipient clients. They'll process the configuration included in these values in lines 43 and 46 and the use of the forward secure signatures will prevent them from processing messages of old replicas. As a result, all the clients will have the same values of *lastDec.conf* and *propV.conf*, meaning that they will contact the same replicas and need to form a majority in the active configuration $lastDec.conf \sqcup propV.conf$, where their majority quorums intersect. As seen in the lemma 6 if any malicious replica tries to issue incomparable ACKs the clients will accuse it and ignore it from this point onward. They will retry the proposal again until none of the replicas in the majority misbehaves.

Let s be the first state decided in this scenario, henceforth the states will be totally ordered, sharing the same configuration which is always present in confComb, meaning that these states are connected. The graph will only have one growing branch and any new state s' in it will be a descendant of s. Finally, this branch will be infinite because the clients never stop proposing. ◀

▶ **Theorem 8.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* validity*.*

Proof. The learnt values by a client are extracted from its *objL* and *confL*.

First of all, at the beginning of a proposal (lines 9 and 10) the value present in the input buffer is read and put into the ledgers, guaranteeing that when a decision is made it shall be present in it.

These variables are then modified in lines 31, 32, 45 and 46. Therefore, the values included into them either come from the the input buffer from the clients where they are signed, or they are informed by replicas nacking proposals after passing signature check or by the information of other clients decisions. We conclude that the values learnt always come from the client input buffers directly or indirectly. ◀

▶ **Theorem 9.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* completeness*.*

Proof. By Lemma 6 whenever the graph G_s forks an accusation is issued. Each fork occur when clients learn incomparable values and are caused by some Byzantine replicas which are eventually accused. Moreover, by lemma 7 the system cannot be indefinetely forked and all inconsistencies are eventually solved when no new accusations are issued as required. ◀

▶ **Theorem 10.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* accusation stability*.*

Proof. The set of accused processes is reflected in the algorithm via the variable *accusation* which is updated in lines 49, 53, and 62. As one can see, they either attribute this variable to a union where one of the operands is itself or a value is explicitly included into it. Therefore after each update the new value must, by the definition of these operations, include the old one, i.e. the accusation set is monotonically increasing. ◀

▶ **Theorem 11.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* accuracy*.*

Proof. An accusation can be issued in lines 27, 49 and 53.

The first occurrence checks that an ACK was issued but the matching configuration proposal wasn't added by the replying replica as described in line 70 meaning that this line was skipped.

On the second case the decision of incomparable values requires, as seen earlier in Theorem 9 that a replica acknowledged incomparable values, violating the behavior of benign replicas described by Lemma 5. Having two ACKs signed by the same process for incomparable values characterises an irrefutable proof and the replicas in it will be a non-empty subset of M because they deviated from the algorithm.

On the last case a replica will be caught providing fake signatures, which is by itself enough to accuse it as this is a clear deviation from the algorithm. ◀

▶ **Theorem 12.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* authenticity*.*

Proof. Authenticity follows from our cryptographic assumptions and specially from three properties of the underlying system:

- Every message contains a signature;
- The signatures can be verified by a public function;
- No other process can sign on behalf of a correct process. ◀

▶ **Theorem 13.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides* agreement*.*

Proof. Agreement is a direct consequence of the dissemination of information implemented in the algorithm. Every accusation is broadcasted and every message containing an accusation is analysed and, if it holds, leads to the adoption of the information (line 60). ◀

▶ **Theorem 14.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 is* alive*.*

Proof. After the system stops reconfiguring a client can eventually receive the last configuration learnt by the broadcast in line 36 and then every client will contact active configurations. If a client then starts a proposal when receiving a value v in its input buffer, a majority of replicas in all active configurations shall eventually respond to this client, following our majority of correct replicas assumption in this scenario. From this point on, all decisions shall contain v as the last learnt configuration all clients contact will provide a majority of replicas that include v. ◀

5 Related Work

Accountability. In security terms, accountability ensures that the actions of an entity can be traced solely to that entity. This supports non-repudiation, deterrence, fault detection, and after-action recovery. Distributed computing research has focused for many years on failure detection [10,12,22], a close relative of accountability. By identifying faulty processes, failure detection helps the distributed computation to make progress in a safe way, but does not provide evidences of misbehaviors that can be verified by a third party. To the best of our knowledge, PeerReview [20] was the first proposing a general solution to provide accountability as an add-on feature for any distributed protocol. In PeerReview each process in the system records messages in tamper-evident logs: an auditor can challenge a process,

retrieve its logs, and simulate the original protocol to ensure that the process behaved correctly. By doing so, any observable deviating action can be traced back to at least one Byzantine process that was responsible for it. The main issue is that for an auditor to prove that a process is Byzantine it must receive a response to the challenge from the process. If no response is received, the auditor cannot determine whether the process is faulty or not. As a result, some Byzantine processes might be suspected forever and never proven guilty. This limitation is common to distributed protocols that are not designed to provide accountability.

Polygraph [11] equips Byzantine Consensus with an accountability mechanism. As in our system, the very messages sent during the protocol execution carry the necessary information to construct a proof in case of Consensus agreement violation. This way, there is no need to query processes to collect evidences and construct a proof. Fairledger [25] and LLB (Long-Lived Blockchain) [32] are consensus-based state-machine replication protocols that are able to detect consistency violations in consensus instances and reconfigure themselves. In contrast, we do not rely on consensus for reconfiguration and propose a purely asynchronous accountable and reconfigurable service.

Lattice agreement. Attiya et al. [4] introduced the (one-shot) lattice agreement abstraction and, in the shared-memory context, described a wait-free reduction of lattice agreement to atomic snapshot. Falerio et al. [14] introduced the long-lived version of lattice agreement (adopted in this paper), called generalized lattice agreement, and described an asynchronous message-passing implementation of lattice agreement assuming a majority of correct processes. In the Byzantine failure model, Di Luna et al [27] proposed for the first time a solution for Byzantine asynchronous generalized lattice agreement, later improved by [36]. All these algorithms propose a fault-tolerant approach where safety and liveness are guaranteed with $f < n/3$ Byzantine processes and authenticated channels. In our accountability approach, liveness and recovery from safety violations are guaranteed with $f < n/2$ Byzantine processes and authenticated channels.

Asynchronous reconfiguration. Dynastore [1] was the first solution emulating a reconfigurable atomic read/write register without consensus: clients can asynchronously propose incremental additions or removals to the system configuration. Since proposals commute, concurrent proposals are collected together without the need of deciding on a total order. In [21] it has been observed that asynchronous reconfiguration can be handled using an external reliable lattice-agreement object. Reconfigurable lattice agreement [23] enables reconfigurable versions of a large class of objects and abstractions, including state-based CRDTs [33], atomic-snapshot, max-register, conflict detector and commit-adopt.

In the Byzantine fault model, Dynamic Byzantine storage [3, 30] allows a trusted *administrator* to issue ordered reconfiguration calls that might also change the set of replicas. More recently, [24] describes a generic Byzantine fault-tolerant reconfigurable lattice agreement, implemented without assuming a trusted administrator.

The reconfiguration technique used in this paper takes inspiration from [23] while been enriched with the use of forward-secure signatures as proposed in [24] to protect the system from Byzantine replicas belonging to old configurations. Note that none of the cited work provide proof-of-misbehavior of Byzantine processes.

6 Concluding remarks

In this paper, we propose the first design of an asynchronous replicated system that not only detects misbehavior that affects its safety properties, but is also able to mitigate misbehaving replicas by reconfiguration. Compare to earlier [16, 25] and concurrent [32] work, we do not employ consensus to agree on the evolving configurations. The algorithm described in this paper can be improved and generalized in multiple ways. Below we discuss some of them.

Garbage collection. In the current version of our algorithm, every process locally maintains a complete history of updates, and periodic reinitialization of the system is an important issue. In particular, it appears challenging to reinitialize the set of accusations, as a slow client may never be able to be convinced that a compromised replica is not trustful anymore. One may think, e.g., of a periodic instances of a consensus protocol among the clients to agree on the new initial system state, running in parallel with our algorithm. Altogether, periodic "truncation" of the ever-growing state in an asynchronous protocol remains an interesting question for the future work.

Complexity. Similar to earlier solutions of (generalized) lattice agreement [14, 23], the latency of learning a value in our algorithm (in the number of asynchronous query-response rounds, assuming that the configuration does not change) is proportional to the number of concurrently proposed values. It remains unclear if there is an asymptotically faster algorithm. There are interesting solutions for *one-shot* Byzantine lattice agreement that take $\log k$ rounds for k proposed values [37], but we do not have a comparable long-lived implementation.

For simplicity, in our algorithm, the sizes of messages grow linearly with the number of distinct values learnt by the clients. One can improve this by sending relative updates instead of complete histories in PROPOSE and DECISION messages. The size of ACK and NACK messages already grow much slower, as they use digests of corresponding proposals and only contain information about changes: in the case of ACKs, these changes consist of the pending configurations since last decision and in the case of NACKs – with respect to the proposed value the replica is responding to. An ACCUSATION message has asymptotic complexity of an ACK message. The issue of maintaining bounds on ever-growing message sizes is related to the more general question of garbage-collection and reinitialization.

Clients: Byzantine and heavy. Early proposals of quorum-based fault-tolerant storage systems typically assumed that clients are benign (see, e.g., [29]). While the effect of Byzantine *writers* can be mitigated using erasure coding [17] or voting [26], it appears nontrivial to handle malicious reconfiguration requests. Indeed, a Byzantine client can block progress by plunging the system in constant reconfiguration, or break safety of the replicated data by rendering the system to a compromised configuration. How to handle such attacks is an intriguing challenge.

Assuming that the clients are benign enables assigning them with a major part of the total work. This results in linear message complexity: the replicas only passively respond to clients' queries.

Alternatively, we may follow earlier work on asynchronous Byzantine reconfiguration [24], and assume an external *access control* mechanism ensuring that inputs from the clients (including reconfiguration calls) are "acceptable". In particular, the proposed configurations should satisfy the configuration availability condition (Section 3): every combination of

candidate configurations should contain enough correct replicas. Also, the access control mechanism should provide a verification procedure that would allow the third party to verify validity of reconfiguration requests. The clients are then only responsible for submitting valid to the set of replicas. The resulting algorithm will, however, likely to be more costly in terms of message complexity, as each reconfiguration will have to handle each of the valid requests.

Our algorithm can also be easily extended to accommodate partitions of the clients into (benign) administrators and (Byzantine-prone) users, along the lines of [25, 30].

For completeness, in Appendix A, we describe a specification and a corresponding implementation of a *one-shot* lattice agreement abstraction that assumes that both clients and replicas can be Byzantine. Our system is particularly well suited for the client-administrator approach as the reconfiguration requests are issued by the proposing entities (in this case an administrator) and not the entities maintaining the system (the replicas).

Reconfiguration strategy. In this paper, we delegated the task of choosing new configurations to the clients. The clients are free to reconfigure the system even if no new misbehaving replicas are detected. The only requirement we impose on the configurations proposed by the clients is that resulting configurations must remain available (Section 3). But one may think of more explicit reconfiguration strategies. For example, each time a new misbehaving replica is detected, it is replaced with a new one taken from a "pool" of correct replicas (a similar approach is proposed in LLB [32] for consensus-based reconfiguration).

References

1 M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.

2 E. Alchieri, A. Bessani, F. Greve, and J. da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 26:1–26:17, 2017.

3 L. Alvisi, D. Malkhi, E. Pierce, M. K. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.

4 H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.

5 M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.

6 X. Boyen, H. Shacham, E. Shen, and B. Waters. Forward-secure signatures with untrusted update. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 191–200, 2006.

7 G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, Nov. 1987.

8 C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

9 M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

10 T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

11 P. Civit, S. Gilbert, and V. Gramoli. Polygraph: Accountable byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413. IEEE, 2021.

12 C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Koutnetzov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing*, 2004.

13 M. Drijvers, S. Gorbunov, G. Neven, and H. Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

14 J. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.

15 E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.

16 S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010.

17 G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *(DSN*, pages 135–144. IEEE Computer Society, 2004.

18 A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, Dec. 2009.

19 A. Haeberlen, P. Kuznetsov, and P. Druschel. The case for byzantine fault detection. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep'06)*, Nov. 2006.

20 A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.

21 L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

22 K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *Comput. J.*, 46(1):16–35, 2003.

23 P. Kuznetsov, T. Rieutord, and S. Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.

24 P. Kuznetsov and A. Tonkikh. Asynchronous reconfiguration with byzantine failures. In H. Attiya, editor, *DISC*, volume 179 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

25 K. Lev-Ari, A. Spiegelman, I. Keidar, and D. Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *OPODIS*, volume 153 of *LIPIcs*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

26 B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. In *ICDCS*, page 34. IEEE Computer Society, 2006.

27 G. A. D. Luna, E. Anceaume, and L. Querzoni. Byzantine generalized lattice agreement. In *IPDPS*, pages 674–683. IEEE, 2020.

28 T. Malkin, D. Micciancio, and S. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 400–417. Springer, 2002.

29 J. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In D. Malkhi, editor, *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2002.

30 J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 325–334. IEEE, 2004.

31 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

32 A. Ranchal-Pedrosa and V. Gramoli. Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain. *CoRR*, abs/2007.10541, 2020.

33 M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

34 A. Spiegelman and I. Keidar. On liveness of dynamic storage. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 356–376, 2017.

35 A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

36 X. Zheng and V. K. Garg. Byzantine lattice agreement in asynchronous systems. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

37 X. Zheng and V. K. Garg. Byzantine lattice agreement in asynchronous systems. In Q. Bramas, R. Oshman, and P. Romano, editors, *OPODIS*, volume 184 of *LIPIcs*, pages 4:1–4:16, 2020.

A Accountable Lattice Agreement with Byzantine Clients

In this section, we discuss a *one-shot* static version of accountable lattice agreement that considers that both clients and replicas might be Byzantine.

A.1 Problem statement

The *general accountable one-shot lattice agreement* (*A1LA*) abstraction, defined on a lattice $(\mathcal{L}, \sqsubseteq)$, takes, as a single input, an element in $\mathcal{L}$ and produces, as an output, a pair of an element in $\mathcal{L}$ and a set of *accusations*. Again, an accusation is a pair (A, P) where $A \subset \Pi$ and P is a *proof of misbehavior*. And we assume that the proof can be independently verified by a third party through a boolean map *verify-proof*$: (2^{\Pi} \times \mathcal{P}) \rightarrow \{true, false\}$.

We say that this version is general because both clients and replicas can be malicious. The system contains N replicas where a majority of them are correct. Let $U \subseteq B$ be the finite set of benign clients that proposed values in that run, and let u_i denote the value proposed by a process $p_i \in U$. Let V_B and V_C be the sets of, respectively, benign and correct clients that learned values in that run, and let v_i denote the value learned by a process $p_i \in V_B$ (obviously, $V_C \subseteq V_B \subseteq U$). The A1LA abstraction satisfies the following properties:

- **Validity.** The value learnt by a benign client c_i with input value u_i is a join of values proposed by clients in U (including c_i), at most $|M|$ values coming from M, and u_i:

  $$\forall c_i \in V_B : u_i \sqsubseteq v_i \wedge v_i \sqsubseteq \sqcup(\{u_j | c_j \in U\} \cup F),\ F \subseteq \mathcal{L},\ |F| \leqslant |M|.$$

- **Consistency.** Either the values learned by the correct clients are totally ordered:

  $$\forall c_i, c_j \in V_C : v_i \sqsubseteq v_j \vee v_i \sqsubseteq v_j.$$

  or every correct process eventually accuses a set of processes.
- **Accuracy.** If a benign process p_i accuses A (with P), then A is a subset of M and P contains a proof against each process in A.

  $$\forall p_i \in B, A \subseteq M \wedge \textit{verify-proof}(A, P)$$

- **Authenticity.** It is computationally infeasible to construct $A \cap B \neq \varnothing$ and $P \in \mathcal{P}$ such that *verify-proof*$(A, P) = true$.
- **Liveness.** If a correct client proposes a value, it eventually learns a value or accuses a set of processes.

One can see that a benign client can accuse a set of processes only if there is at least one Malicious process ($M \neq \varnothing$) and in the absence of malicious processes no proofs of misbehavior are created. The Consistency property guarantees that either *correct* clients learn comparable values or some malicious process will be accused. Notice that we cannot

avoid executions in which a *Benign* but not correct client learns a value that is inconsistent with a value learnt by another benign client if there are malicious processes without issuing accusations.

A.2 The algorithm

Our solution to A1LA is presented in Algorithm 5, Algorithm 6 and Algorithm 7. As before, each method is executed in its entirety without being interrupted. Each client might be in an *active* state where it proposes values and takes steps towards learning something new, re-proposing if necessary or in a *passive* state where it only reacts to other client proposals. On start-up clients that receive input values from the application sign them and put them to their input ledgers (line 84) and proceed to propose it to the system by multicasting (91).

When a replica receives a proposal with a ledger, it extracts the proposed value by the other process merging the new inputs to its own ledger (117). Before treating the ledger a verification is made to guarantee its integrity (113). Once the message is validated, there might be inconsistencies in the ledger introduced by malicious processes that tried to insert more than one value in the system and an accusation might be issued (118). Otherwise the ledger is consistent and an ACK can then be produced if the proposal comprises the previously received ones (126), otherwise a NACK shall be sent informing the proposer of values that it didn't include in its proposal via a complement to its ledger (130).

Received ACKs are discarded if they correspond to old proposals, or if they come from a process whose ACK has already been accounted in the current proposal, or if they don't match the proposed value they were sent for. Note that the latter is in itself a sign of byzantine behavior but doesn't constitute an irrefutable proof of misbehavior as the conflicting messages, i.e. the proposal and the ACK, are signed by different processes. When the ACK is valid it is included in the *ackL* (97) and the respective counter is incremented (98). Similarly, outdated NACKs are ignored when received, as well as empty ledgers, ledgers who don't include new values (101). Once the ledger is validated, the proposed value is set to include the information patch (87) and the counter of NACKs is incremented (103).

After a client has received responses from at least a majority of replicas (99 and 104) for its proposal it proceeds to evaluate if it can decide on a value or not, in case it has not accused any malicious processes along the way. If a NACK has been received, it tries to learn its new proposal (92) which includes the missing values in the previous attempt, otherwise it decides upon its proposal and broadcasts it alongside the ledger of ACKs it has collected (96). At this point clients who are proposing incomparable values to the one decided check the ACKs they received so far as byzantine processes can lead the system to decide incomparable values by issuing contradictory ACKs for different processes, which can be detected at this point and lead to their accusation (110).

Algorithm 5 Accountable One-Shot Lattice Agreement: Code for client c part 1.

Local variables:
status, initially passive { *Boolean for the current state: passive or active* }
ackCnt, initially 0 { *The number of* acks *received for the active proposal* }
nackCnt, initially 0 { *The number of* nacks *received for the active proposal* }
activePropNb, initially -1 { *The number of* nacks *received for the active proposal* }
propV, initially $\perp$ { *The value being proposed* }
inL, initially empty { KV table (ledger) init. w/ proposed values signed by their originators }
ackL, initially empty { *Key value table holding received signed acks by replicas* }
Input:
initV { *Value initially proposed by the process, provided by external source* }
Outputs:
outV, initially $\perp$ { *Value learnt by the client* }
accusation, initially $\varnothing$ { *Proofs of misbehavior gathered* }

upon *startup* **if** $initV \neq \perp$
83 $propV := initV$
84 include *signed* $initV$ to inL
85 Propose

operation *Propose*
86 $status := active$
87 $propV := extractLedger(inL, c)$
88 $activePropNb := activePropNb + 1$
89 clear $ackL$
90 $ackCnt := nackCnt := 0$
91 *multicast* $\langle PROPOSAL, (inL, activePropNb)\rangle$ to *Servers*

operation *EvaluateDecision*
92 **if** $nackCnt > 0$ **then** Propose
93 **else**
94 $outV := propV$
95 $status := passive$
96 *multicast* $\langle DECISION, (outV, ackL)\rangle$ to *Servers*

upon *receive* $\langle ACK, (propV, activePropNb)\rangle$
from given replica r AND $r \notin ackL$ $status = active$
97 append q's ack to $ackL$
98 $ackCnt := ackCnt + 1$
99 **if** $ackCnt + nackCnt \geqslant \lceil \frac{N+1}{2} \rceil$ **then** EvaluateDecision

upon *receive* $\langle NACK, (\Delta Ledger, activePropNb)\rangle$ from process q
AND $status = active$ AND $\Delta Ledger \neq \varnothing$
100 $\Delta Value = extractLedger(\Delta Ledger, q)$
101 **if** $\Delta Value \sqsubseteq propV$ **return**
102 $inL := inL \cup (inL')$
103 $nackCnt := nackCnt + 1$
104 **if** $ackCnt + nackCnt \geqslant \lceil \frac{N+1}{2} \rceil$ **then** EvaluateDecision

Algorithm 6 Accountable One-Shot Lattice Agreement: Code for client c part 2.

upon *receive* $\langle DECISION, (outV', ackL')\rangle$ from process q
105 **if** $\exists (p, v) \in ackL' | v \neq outV'$
106 $accusation := accusation \cup \{(q, DECISION)\}$
107 $status = passive$
108 **return**
109 **if** $outV' \not\sqsubseteq propV$ && $propV \not\sqsubseteq outV'$ **then**
110 **let** $M = \{m | m \in ackL$ && $ackL'\}$ **do**
111 **foreach** $m \in M$ **do** include $(b, \{ackL[b], ackL'[b]\})$ to $accusation$
112 **if** $|M| > 0$ **then** $status := passive$

operation $extractLedger\ (inL', sender)$
113 **if** $\exists$ *process* $p \in inL'$ with invalid signature **then**
114 $accusation := accusation \cup \{(sender, getPropNACKMSG(inL')\}$
115 $status := passive$
116 **return** $\varnothing$
117 $inL'' := inL \cup (inL')$
118 **let** $M = \{m | m \in inL''$ && $|inL''[m]| > 1\}$ **do**
119 **foreach** $m \in M$ **do** include $(m, getPropNACKMSG(inL''[m]))$ to $accusation$
120 $status := passive$
121 **return** $\varnothing$
122 **let** $receivedValue = \sqcup[v | \exists p, inL'[p] = v]$
123 **return** $receivedValue$

Algorithm 7 Accountable One-Shot Lattice Agreement: Code for replica r.

Local variables:
inL, initially empty
{ *Key value table holding initially proposed values signed by their originators* }
$repV$ initially $\perp$ { *Value held by the replica* }
$accusation$, initially $\varnothing$ { *Proofs of misbehavior gathered by the replica* }

upon *receive* $\langle PROPOSAL, (inL', activePropNb')\rangle$ from process q
124 $propV' := extractLedger(inL')$
125 **if** $repV \sqsubseteq propV'$ **then**
126 send $\langle ACK, (propV', activePropNb')\rangle$ to q
127 $repV := propV'$
128 **else**
129 $repV := repV \sqcup propV'$
130 send $\langle NACK, (inL - inL', activePropNb')\rangle$ to q

operation $extractLedger\ (inL', sender)$
{ Identitical to client operation with the same name without lines 115 and 120 }

A.3 Correctness

We claim that the algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 solves the General Accountable One-Shot Lattice Agreement.

▶ **Lemma 15.** *At every benign process, the variable propV is monotonically increasing.*

Proof. The variable is updated only in line 87. As one can see, it is a join operation where one of the operands is the previous value, hence the new value by definition contains the old value. ◀

▶ **Lemma 16.** *If a benign process p learns a value v, then v cannot contain two or more values signed by the same process.*

Proof. A process adds values to its proposal *propV*, when it receives a *nack* response. The insertion is then subject to verification following line 118. Process p will only proceed to a deciding a value if the list comprehension yields an empty list, in which case there is at most one value introduced on its proposal per process in the system. ◀

▶ **Lemma 17.** *At every benign process, the variable repV is monotonically increasing.*

Proof. The variable is updated in lines 127 and 129. The first update assigns to this variable a new value which has passed the test in line 125, so the new value contains the old one. Similarly to *propV*, the second update is a join where one of the operands is the old value, so the new value must contain the old one. ◀

▶ **Theorem 18.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides* consistency*.*

Proof. Suppose, by contradiction, that two benign processes p and q learned two incomparable values v' and v''. The majority that acknowledged v' at p must intersect with the majority that acknowledged v'' at q. Let r be any process in the intersection. If r is not Byzantine then by Lemma 17, v' and v'' must be comparable and *consistency* will hold.

Otherwise, r must have acked incomparable values, which shall be detected in line 110 meaning that the processes that output incomparable values will accuse r. ◀

▶ **Theorem 19.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides* validity*.*

Proof. The inclusion of the process own proposal follows from Lemma 15 with the initialisation of *propV* to *initV*, remarking that *outV* is but one of the values taken by *propV*.

As for the cap on the number of values coming from byzantine processes, suppose that there are at least $|M| + c$, where $c \in \mathbb{N}^*$, values coming from byzantine processes. It means that at least one byzantine process b signed two or more initial values that are output by a benign process. Because of Lemma 16, this is impossible. ◀

▶ **Theorem 20.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides* accuracy*.*

Proof. An accusation can be issued in lines 114, 119, 111.

On the first case it will have a proposal signed by a process which doesn't hold valid signed origins for its values. One of the values can come from the process itself, in which case a benign process would have signed it and put in its ledger on the initialisation. The other values must come through NACKs that also provide signed origins obtained in line 117

which benign processes include in its ledger. Having a proposal signed by a process which provided fake signatures to the origin of its values consist as an irrefutable proof and M' will be a non-empty subset of M.

The second scenario tracks processes that have inserted more than one value in the system. A benign process would only do it once during initialisation and having two inclusions signed by the same process consists as irrefutable proof and M' will be a non-empty subset of M.

Finally, the decision of incomparable values requires, as seen earlier in Theorem 18 that a process acknowledged incomparable values, violating the behavior of benign processes described by Lemma 17. Having two ACKs signed by the same process for incomparable values characterises as irrefutable proof and M' will be a non-empty subset of M. ◀

▶ **Theorem 21.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides* authentiticy*.*

Proof. This property is exactly the same as Theorem 12. ◀

▶ **Theorem 22.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides* liveness*.*

Proof. Following Theorem 19 combined with Lemma 15, each run can have at most $|U|$ different values being proposed. Since by the end of this many proposals a client shall propose the join of all these values, it will get ACKs from a majority of processes proceeding to learn a value or gather enough information for accusing at least one byzantine process, as at this byzantine clients must have introduced more than one value in the system for the proposal not to go through and Theorem 20 holds. ◀