Byzantine Generalized Lattice Agreement

Giuseppe Antonio Di Luna, Emmanuelle Anceaume, Leonardo Querzoni

Byzantine Generalized Lattice Agreement

Giuseppe Antonio Di Luna
DIAG, Sapienza University of Rome
Italy
diluna@diag.uniroma1.it

Emmanuelle Anceaume[1]
CNRS, Univ Rennes, Inria, IRISA
France
Emmanuelle.Anceaume@irisa.fr

Leonardo Querzoni
DIAG, Sapienza University of Rome
Italy
querzoni@diag.uniroma1.it

***Abstract*—The paper investigates the Lattice Agreement (LA) problem in asynchronous systems. In LA each process proposes an element e from a predetermined lattice, and has to decide on an element e' of the lattice such that $e \leq e'$. Moreover, decisions of different processes have to be comparable (no two processes can decide two elements e' and e such that $(e \not\leq e') \wedge (e' \not\leq e)$).**

It has been shown that Generalized LA (i.e., a version of LA proposing and deciding on sequences of values) can be used to build a Replicated State Machine (RSM) with commutative update operations. The key advantage of LA and Generalized LA is that they can be solved in asynchronous systems prone to crash-failures (which is not the case with standard Consensus).

In this paper we assume Byzantine failures. We propose the Wait Till Safe (WTS) algorithm for LA, and we show that its resilience to $f \leq (n-1)/3$ Byzantine processes is optimal. We then generalize WTS obtaining a Generalized LA algorithm, namely GWTS. We use GWTS to build a RSM with commutative updates. Our RSM works in asynchronous systems and tolerates $f \leq (n-1)/3$ malicious entities. All our algorithms use the minimal assumption of authenticated channels. When the more powerful public signatures are available, we discuss how to improve the message complexity of our results (from quadratic to linear, when $f = \mathcal{O}(1)$). To the best of our knowledge this is the first paper proposing a solution for Byzantine LA that works on any possible lattice, and it is the first work proposing a Byzantine tolerant RSM built on it.

***Index Terms*—lattice agreement, replicated state machine, Byzantine faults**

I. Introduction

State machine replication (RSM) is today the foundation of many cloud-based highly-available products: it allows some service to be deployed such to guarantee its correct functioning despite possible faults. In RSM, clients issue operation requests to a set of distributed processes implementing the replicated service, that, in turn, run a protocol to decide the order of execution of incoming operations and provide clients with outputs. Faults can be accidental (e.g. a computer crashing due to a loss of power) or have a malicious intent (e.g. a compromised server). Whichever is the chosen fault model, RSM has proven to be a reliable and effective solution for the deployment of dependable services. RSM is usually built on top of a distributed Consensus primitive that is used by processes to agree on the order of execution of requests concurrently issued by clients. The main problem with this approach is that Consensus is impossible to achieve deterministically in a distributed settings if the system is asynchronous and even just a single process may fail by crashing [1]. This led the research community to study and develop alternative solutions based on the relaxation of some of the constraints, to allow agreement to be reached in partially synchronous systems with faulty processes by trading off consistency with availability.

An alternative approach consists in imposing constraints on the set of operations that can be issued by clients, i.e. imposing updates that commute. In particular, in 2012 Faleiro et al. [2] introduced a RSM approach based on a generalized version of the well known Lattice Agreement (LA) problem, that restricts the set of allowed update operations to commuting ones [3]. They have shown that commutative replicated data types (CRDTs) can be implemented with an RSM approach in asynchronous settings using the monotonic growth of a join semilattice, i.e., a partially ordered set that defines a join (least upper bound) for all element pairs (see Figure 1 for an example). A typical example is the implementation of a dependable counter with add and read operations, where updates (i.e. adds) are commutative.

In the LA problem, introduced by Attiya et al. [4], each process p_i has an input value x_i drawn from the join semilattice and must decide an output value y_i, such that *(i)* y_i is the join of x_i and some set of input values and *(ii)* all output values are comparable to each other in the lattice, that is form a chain in the lattice (see Figure 1). LA describes situations in which processes need to obtain some knowledge on the global execution of the system, for example a global photography of the system.

Differently from Consensus, LA can be deterministically solved in an asynchronous setting in presence of crash failures. Faleiro et al. [2] have shown that a majority of correct processes and reliable communication channels are sufficient to solve LA, while Garg et al. [5] proposed a solution that requires $\mathcal{O}(\log n)$ message delays, where n is the number of processes participating to the algorithm. The very recent solution of Skrzypczak et al. [6] considerably improves Faleiro's construction in terms of memory consumption, at the expense of progress.

In the Generalized Lattice Agreement (GLA) problem processes propose an infinite number of input values (drawn from an infinite semilattice) and decide an infinite sequence of output values, such that, all output values are comparable to each other in the lattice i.e. form a chain (as for LA); the sequence of decision values are non-decreasing, and every input value

[1]This work has been realized while the author was at Sapienza University of Rome, funded by the "Sapienza Visiting Professor Programme".

eventually appears in some decision values. Solving GLA in asynchronous distributed systems reveals to be very powerful as it allows to built a linearizable RSM of commutative update operations [2].

Despite recent advancements in this field, to the best of our knowledge no general solution exists that solves LA problems in an asynchronous setting with Byzantine faults. In the present paper we continue the line of research on LA in asynchronous message-passing systems by considering a Byzantine fault model, i.e. a model where processes may exhibit arbitrary behaviors. The contributions of this work can be summarized as follows:

- We first introduce a LA specification that takes into account Byzantine faults. Then we propose an algorithm, namely *Wait Till Safe* (WTS), which, in presence of less than $(n-1)/3$ Byzantine processes, guarantees that any correct process decides in no more than $5+2f$ message delays with a global message complexity in $\mathcal{O}(n^2)$ per process. We show that $(n-1)/3$ is an upper bound. The algorithm is wait-free, i.e., every process completes its execution of the algorithm within a bounded number of steps, regardless of the execution of other processes.
- We then go a step further by proposing an algorithm, namely *Generalized Wait Till Safe* (GWTS), to solve GLA in a Byzantine fault model. Our "wait until safe" strategy guarantees that each correct process performs an infinite sequence of decisions, and for each input received at a correct process, its value is eventually included in a decision. Our algorithm is wait-free and is resilient to $f \leq (n-1)/3$ Byzantine processes.
- We further present the construction of a RSM for objects with commuting update operations that guarantees both linearizability and progress in asynchronous environments with Byzantine failures.
- Finally, we sketch the main lines of a signature-based version of our algorithms which take advantage of digital signatures to reduce the message complexity to $\mathcal{O}(n)$ per process, when $f = \mathcal{O}(1)$.

To the best of our knowledge this is the first paper proposing a solution for Byzantine LA that works on any possible lattice, and it is the first work proposing a Byzantine tolerant RSM built on it.

The rest of this paper is organized as follows: Section II discussed the related works; Section III describes the system model, Section IV formalises LA and illustrates the necessity of at least $3f+1$ processes; Section V and Section VI introduce the algorithms for Byzantine LA and Byzantine GLA, respectively; Section VII describe the construction of a byzantine tolerant RSM; Section VIII sketches a signature-based variant of our solutions and, finally, Section IX concludes the paper. For space reason some details and proofs are omitted and can be found in the full version [7].

II. Related Work

Lattice Agreement has been introduced by Attiya et al. [4] to efficiently implement an atomic snapshot object [8], [9].

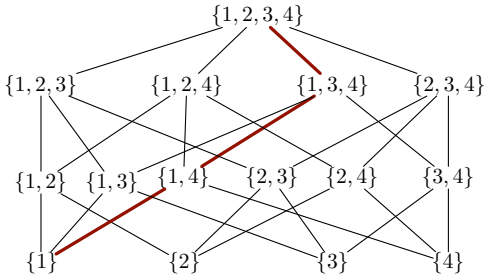

Fig. 1. Hasse diagram of the semilattice induced over the power set of $\{1,2,3,4\}$ using the union operation as join. Taken two elements e, e' of the lattice if $e < e'$, then e appears lower in the diagram than e' and there is an "upward" path, going from lower points to upper points, connecting e to e' (e.g., $\{1\} \leq \{1,3,4\}$, but $\{2\} \not\leq \{3\}$). Any two elements e, e' of the semilattice have a join $e \oplus e' = e \cup e'$ and $e \oplus e' \geq e, e'$ (e.g., $\{1\} \oplus \{2,3\} = \{1,2,3\}$). The red edges indicate the chain (i.e., sequence of increasing values) selected by the Lattice Agreement protocol.

Their construction is such that each *scan* or *collect* operation requires $\mathcal{O}(1)$ executions of LA and uses $\mathcal{O}(n)$ read/write registers. Then Faleiro [2] have shown that GLA is a very interesting abstraction to build RSMs with strong consistency properties, i.e., linearizability of its operations, and liveness guarantees in asynchronous systems. Very recently Nowak and Rybicki [10] have studied LA in presence of Byzantine faults. Their specifications of LA is more restrictive than the one we propose since it does not allow decisions to contain values proposed by Byzantine processes. We argue that our Lattice Agreement specification is more fit to build RSM on top of the LA algorithm. Removing a value proposed by a Byzantine process might not be desirable: think about an RSM that implements an object shared by different organizations, it could be a breach of contract to selectively avoid certain updates even when the sender misbehaved. A second reason is more technical and it stems from the interaction between the impossibility result introduced by the specifications of [10] and how an RSM is implemented using GLA [2]. Following [2] to implement the RSM we take the power set of all possible updates and we construct a lattice on it using as join the union operation. As an example, let us suppose that we want to build a set counter data type, and let us assume that clients issue four update operations add(1), add(2), add(3), add(4) interleaved with reads. In this case our semilattice is the one in Figure 1, and a Lattice Agreement algorithm will ensure that each read will see values on a single chain, the red one in the figure. Thanks to this, different reads will see "growing" versions of the counter that are consistent snapshots (e.g., if someone reads $\{1\}$, the other could read $\{1,4\}$, but it can not read $\{4\}$). Such a semilattice has a breadth[1] of 4, actually, each semilattice obtained using as join operation the union over the power set of a set of k different values has breadth k. Therefore to satisfy the specifications of [10] using the semilattice in Figure 1 we should have at least 5 processes participating to Lattice Agreement. Unfortunately, it is often the case that the number of possible update operations is larger

[1]Formally, the breadth of a semilattice $L = (V, \oplus)$ is the largest n, such that taken any set of $U \subseteq V$ of size $n+1$ we have $\bigoplus U = \bigoplus K$ where K is a proper subset of U.

than the processes running the LA. In the set counter data type, we may have an $\mathsf{add}(x)$ for any $x \in \mathbb{N}$, in such setting the specifications of [10] is impossible to implement. Our specifications circumvent such impossibility.

III. Model

We have a set $P : \{p_0, p_1, \ldots, p_{n-1}\}$ of processes. They communicate by exchanging messages over asynchronous authenticated reliable point-to-point communication links (messages are never lost on links, but delays are unbounded). The communication graph is complete: there is a communication link between each pair of processes.

We have a set $F \subset P$ of Byzantine processes, with $|F| \leq f$. Byzantine processes deviate arbitrarily from the algorithm. We assume $|P| \geq 3f + 1$ (we show that this is necessary in Section IV-A). The set of correct processes is $C = P \backslash F$.

IV. The Byzantine Lattice Agreement Problem and The Necessity of $3f + 1$

Each process $p_i \in C$ starts with an initial input value $pro_i \in E \subseteq V$. Values in V form a join semi-lattice $L = (V, \oplus)$ for some commutative join operation $\oplus$: for each $u, v \in V$ we have $u \leq v$ if and only if $v = u \oplus v$. Given $V' = \{v_1, v_2, \ldots, v_k\} \subseteq V$ we have $\bigoplus V' = v_1 \oplus v_2 \oplus \ldots \oplus v_k$.

The task that processes in C want to solve is the one of Lattice Agreement, and it is formalised by the following properties:

- **Liveness:** Each process $p_i \in C$ eventually outputs a decision value $dec_i \in V$;
- **Stability:** Each process $p_i \in C$ outputs a unique decision value $dec_i \in V$;
- **Comparability:** Given any pair $p_i, p_j \in C$ we have that either $dec_i \leq dec_j$ or $dec_j \leq dec_i$;
- **Inclusivity:** Given any correct process $p_i \in C$ we have that $pro_i \leq dec_i$;
- **Non-Triviality:** Given any correct process $p_i \in C$ we have that $dec_i \leq \bigoplus(X \cup B)$, where X is the set of proposed values of all correct processes ($X : \{pro_i |$ with $p_i \in C\}$), and $B \subseteq E$ is $|B| \leq f$.

In the rest of the paper we will assume that L is a semi-lattice over sets (V is a set of sets) and $\oplus$ is the set union operation. This is not restrictive, it is well known [11] that any join semi-lattice is isomorphic to a semi-lattice of sets with set union as join operation.

A. Necessity of at least $3f + 1$ processes

We first show that our specification can only be satisfied when there are at least $3f + 1$ processes. The proof is omitted and it can be found in the extended version.

Theorem 1. *Let $\mathcal{A}$ be any asynchronous distributed algorithm that solves Byzantine Lattice Agreement when up to f are Byzantine processes. We have that $\mathcal{A}$ needs at least $3f + 1$ processes. This holds even if we drop the Inclusivity property from the specification.* □

V. Algorithm *Wait Till Safe* (WTS)

The *Wait Till Safe* algorithm (Algorithms 1 and 2) is divided in two phases: an initial *Values Disclosure Phase* where processes are asked to declare to the whole system values they intend to propose, and then a *Deciding Phase* where processes agree on which elements of the lattice can be decided on the basis of the proposed values. For the sake of clarity, processes are divided in *proposers* that propose an initial value, and then decide one decision value, and *acceptors* which help proposers decide. This distinction does not need to be enforced during deployment as each process can play both roles at the same time.

The main idea in the *Values Disclosure Phase* is to make any proposer disclose its proposed value by performing a Byzantine reliable broadcast. The reliable broadcast prevents Byzantine processes from sending different messages to processes [12], [13]. The exact specification of this broadcast is in [14]. In the pseudocode the broadcast primitive is represented by the RBCast (used for reliably broadcast messages) function and RBCastDel event (that indicates the delivery of a message sent with the reliable broadcast).

Values delivered at each process are saved in a SvS (Safe-values Set). A process moves to the next phase as soon as he receives values from at least $(n - f)$ proposers. Waiting for $(n - f)$ messages is not strictly necessary, but allows us to show a bound of $\mathcal{O}(f)$ on the message delays of our algorithm. Note that, from this point on, some operations of Phase 1 could run in parallel with Phase 2. Thanks to *Value Disclosure Phase* a process is *committed* to its value and cannot change its proposal or introduce a new one during the *Deciding Phase*. During this latter phase, correct processes only handle messages that contain values in SvS, i.e. messages for which the $SAFE()$ predicate is true. Messages that do not satisfy this condition are buffered for later use, i.e. if and when all the values they contain will be in SvS.

The *Deciding Phase* is an extension of the algorithm described in [2] with a Byzantine quorum and additional checks used to thwart Byzantine attacks. Each correct proposer p sends its $Proposed$ value to acceptors in a request message (Line 19). An acceptor receiving a request, sends an ack if the previously $Accepted\_set$ is a subset of the value contained in the request, and updates its $Accepted\_set$ using the $Proposed$ set in the request (initially, the $Accepted\_set$ of an acceptor is the empty set). Otherwise, the acceptor sends a nack containing the $Accepted\_set$, and it updates its $Accepted\_set$ with the union of the value contained in p's request and its $Accepted\_set$. The proposer p decides if it receives $\lfloor (n+f)/2 \rfloor + 1$ acks. In case p receives a nack, then p updates $Proposed$ set by taking the union of it and the value contained in the nack. Each time a proposer updates its $Proposed$ set it issues a new request.

A. Safety properties

Observation 1. *Given any correct process p_j its SvS contains at most one value for each process in P.*

Algorithm 1 *WTS (Wait Till Safe)* -Alg. for Proposer p_i

1: $proposed\_value = pro_i$
2: $init\_counter = ts = 0$
3: $Proposed = Ack\_set = SvS = Waiting\_msgs = \emptyset$
4: $state = disclosing$
5: ▷ Values Disclosure Phase
6: **upon event** $proposed\_value \neq \perp$
7: $Proposed = Proposed \cup proposed\_value$
8: $proposed\_value = \perp$
9: RBCAST($< dis\_phase, proposed\_value >$) to all
10: **upon event** RBCASTDEL FROM SENDER($< dis\_phase, value >$)
11: **if** $value$ is a element of E **then**
12: **if** $state = disclosing$ **then**
13: $Proposed = Proposed \cup value$
14: $SvS = SvS \cup value$
15: $init\_counter = init\_counter + 1$
16: ▷ Deciding Phase
17: **upon event** $init\_counter \geq n - f$ WHEN $state = disclosing$
18: $state = proposing$
19: BROADCAST($< ack\_req, Proposed, ts >$) to all Acceptors
20: **upon event** DELIVERY FROM SENDER(m)
21: $Waiting\_msgs = Waiting\_msgs \cup \{m\}$
22: **upon event** $\exists m \in Waiting\_msgs \,|\, SAFE(m) \wedge state = proposing \wedge m =< ack, \cdot, ts' > \wedge ts' = ts$ FROM SENDER
23: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
24: $Ack\_set = Ack\_set \cup \{< ack, sender >\}$
25: **upon event** $\exists m \in Waiting\_msgs \,|\, SAFE(m) \wedge state = proposing \wedge m =< nack, Rcvd, ts' > \wedge ts' = ts$ FROM SENDER
26: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
27: **if** $Rcvd \cup Proposed \neq Proposed$ **then**
28: $Proposed = Rcvd \cup Proposed$
29: $Ack\_set = \emptyset$
30: $ts = ts + 1$
31: BROADCAST($< ack\_req, Proposed, ts >$) to all Acceptors
32: **upon event** $|Ack\_set| \geq \lfloor (n+f)/2 \rfloor + 1$ WHEN $state = proposing$
33: $state = decided$
34: $decision_i = Proposed$
35: DECIDE($decision_i$)
36: **function** SAFE(m)
37: **if** the lattice element contained in m is a subset of SvS **then**
38: **return** True
39: **else**
40: **return** False

Algorithm 2 *WTS (Wait Till Safe)* - Algorithm for Acceptor process p_i

1: $Accepted\_set = Waiting\_msgs = \emptyset$
2: SvS ▷ Reference to SvS in the corresponding Proposer (Each process is a Proposer and an Acceptor)
3: **upon event** DELIVERY FROM SENDER(m)
4: $Waiting\_msgs = Waiting\_msgs \cup \{m\}$
5: **upon event** $\exists m \in Waiting\_msgs \,|\, SAFE(m) \wedge m =< ack\_req, Rcvd, x >$ FROM SENDER
6: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
7: **if** $Accepted\_set \subseteq Rcvd$ **then**
8: $Accepted\_set = Rcvd$
9: SEND TO SENDER ($< ack, Accepted\_set, x >$)
10: **else**
11: SEND TO SENDER ($< nack, Accepted\_set, x >$)
12: $Accepted\_set = Accepted\_set \cup Rcvd$

The above observation derives from the specification of reliable broadcast, and the fact that in the disclosure phase each participating process broadcasts a single value. We say that a message m containing a set of proposed values is "safe" for a process p_i if such set of values is contained in SvS. It is immediate from function at Lines 36-40 that proposers (in state $proposing$) change their $Proposed$ only when they receive safe messages. The analogous holds for the $Accepted\_set$ of acceptors.

We say that a value v receives m acks if it is contained in a $Proposed$, that is in turn contained in ack messages in the form $< ack, \cdot, ts >$ sent by m acceptors. The same meaning is intended when we say that $Proposed$ receives acks.

Definition 1. *(Committed value) A value v is committed if it received $\lfloor (n+f)/2 \rfloor + 1$ acks.*

Definition 2. *(Committed proposal) A $Proposed$ is committed if it received $\lfloor (n+f)/2 \rfloor + 1$ acks.*

Lemma 1. *Let t be the first time at which a value v is committed, we have that any $Proposed$ committed after t contains v.*

Proof. Value v received at least $\lfloor (n-f)/2 \rfloor + 1$ acks from acceptors in C (see Alg. 1 Line 32). These acceptors have inserted v in their $Accepted\_set$ (see Alg. 2 Line 8). Thus by time t a quorum of acceptors $Q_1 \subseteq C$ has v in their $Accepted\_set$. Let $Proposed$ be a value committed after t, then, by the same above reasoning, we have that $Proposed$ received acks from a set of correct acceptors $Q_2 \subseteq C$, with $|Q_2| \geq \lfloor (n-f)/2 \rfloor + 1$. Since $\exists p \in Q_1 \cap Q_2$ (recall that $|C| = (n-f)$) we have that $Proposed$ contains v: p sent an ack, thus has passed the if at Line 7 of Alg 2. □

Observation 2. *Given any correct process p_j its $decision_j$ has been committed.*

Theorem 2. *Let us consider a set of processes, of size at least $(3f+1)$, executing WTS algorithm. Algorithm WTS enforces: (1) Comparability; (2) Inclusivity; (3) Non-Triviality; (4) Stability.*

Proof. We prove each property separately.

1) is implied by Lemma 1 and Observation 2.
2) derives from the fact that a proposer never removes a value from $Proposed$ and from Line 7.
3) the bound on B derives from the safety of messages and Observation 1, the fact that $dec_i \leq \bigoplus(X \cup B)$ derives from the fact that a correct process insert in its proposal only values received by messages and its initial proposed value.
4) is ensured by Line 33 in proposers. □

Note that the Inclusivity and the Comparability imply that, when all correct proposers decide, then each value proposed by some correct will be in a decision and that there exists a proposer whose decision includes all values proposed by correct proposers.

B. Liveness properties

Lemma 2. *Each message sent by a correct process is eventually safe for any other correct process.*

Proof. If a correct process p_i sent a message m then the set of values contained in m is a subset of SvS of p_i. Note that SvS is only updated as result of the reception of a message reliably broadcast in the disclosure phase (Line 14). For the properties of the broadcast eventually each other correct process will obtain a SvS that contains the set of values in m, making m safe. □

Lemma 3. *A correct proposer refines its proposal (executing Line 31) at most f times.*

Proof. Each time the proposer executes Line 31 it passes the if at Line 27, thus increasing its proposed set. However, its first proposal, in Line 19, contains at least $|X \cup B| - f$ values. Since there are at most $|X \cup B|$ safe values (from Oservation 1), the claim follows. □

Lemma 4. *If there is a time t after which a correct proposer p_i in state $proposing$ cannot execute Line 31, then p_i eventually decides.*

Proof. Let $< ack\_req, Proposed, ts >$ be the last ack request message sent by p_i. Since p_i does not execute Line 31 it means that either it does not receive any nack, or that any nack it receives does not allow him to pass the if Line 27. Since p_i is correct its message $< ack\_req, Proposed, ts >$ will reach each correct acceptor. By hypothesis each of them will send a ack, otherwise p_i should be able to execute Line 31 (they all handle the ack request by Lemma 2). Once p_i receives the acks from the set of correct acceptors, it will handle them, Lemma 2, and decide. □

In the next Theorem (Th. 3) we show that each correct process eventually commits and decides, we also bound the number of delays needed, by each correct proposer, to reach a decision.

Theorem 3. *Let us consider a set of processes, of size at least $3f+1$, executing WTS algorithm. Every correct proposer decides in at most $2f + 5$ message delays.*

Proof. The Byzantine reliable broadcast at Line 9 takes at most 3 message delays. Therefore after three rounds each correct process starts its first proposal. Each refinement takes at most 2 message delays, the time needed to broadcast and receive a response. There are f refinements, see Lemma 3, executed in at most $2f + 2$ message delays, and thus by Lemma 4 after $2f + 5$ message delays a correct decides. □

Note that Theorem 3 implies the Liveness property of our Lattice Agreement specification.

1) Message complexity: The Byzantine Reliable broadcast used at Line 9 costs $\mathcal{O}(n^2)$ messages [14], this cost dominates the other algorithm operations: in the $2f+5$ delays needed to reach the decision at most $\mathcal{O}(f \cdot n)$ messages are generated.

Algorithm 3 *GWTS* -Algorithm for proposer process p_i

1: $proposed\_value = pro_i$
2: $Batch[\forall r \in \mathbb{N}] = SvS[\forall k \in \mathbb{N}] = \emptyset$ ▷ Array of value sets, one batch for each round
3: $Counter[\forall r \in \mathbb{N}] = 0$ ▷ Array of numbers, one for each round
4: $r = -1$
5: $ts = 0$
6: $Proposed = Decided = Waiting\_msgs = Ack\_history = \emptyset$
7: $state = newround$
8: **upon event** NEW VALUE(v)
9: $Batch[r+1] = Batch[r+1] \cup \{v\}$
10: ▷ Values Disclosure Phase
11: **upon event** $state = newround$
12: $state = disclosing$
13: $r = r + 1$
14: $Proposed = Proposed \cup Batch[r]$
15: RBCAST$(< dis_phase, Batch[r], r >)$ to all
16: **upon event** RBCASTDEL FROM SENDER$(< dis_phase, Set, round >)$
17: **if** $state = disclosing \wedge Set$ is an element of the lattice **then**
18: $Proposed = Proposed \cup Set$
19: $SvS[round] = SvS[round] \cup Set$
20: $Counter[round] = Counter[round] + 1$
21: ▷ Deciding Phase
22: **upon event** $Counter[r] \geq n - f$ WHEN $state = disclosing$
23: $state = proposing$
24: $ts = ts + 1$
25: BROADCAST$(< ack_req, Proposed, ts, r >)$ to all Acceptors
26: **upon event** DELIVERY OR RBCASTDEL FROM SENDER(m)
27: $Waiting\_msgs = Waiting\_msgs \cup \{m\}$
28: **upon event** $\exists m \in Waiting\_msgs | SAFE(m) \wedge state = proposing \wedge m =< nack, Rcvd, ts', r' > \wedge ts = ts' \wedge r = r'$
29: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
30: **if** $Rcvd \cup Proposed \neq Proposed$ **then**
31: $Proposed = Rcvd \cup Proposed$
32: $ts = ts + 1$
33: BROADCAST$(< ack_req, Proposed, ts, r >)$ to all Acceptors
34: **upon event** $\exists m \in Waiting\_msgs | SAFE(m) \wedge state = proposing \wedge m =< ack, Accepted\_set, dst, snd, ts', r' > \wedge m$ WAS DELIVERED WITH RBCASTDEL
35: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
36: $Ack\_history = Ack\_history \cup \{m\}$
37: **upon event** $< ack, Accepted\_set, destination, \cdot, timestamp, r' >$ APPEARS $\lfloor (n+f)/2 \rfloor + 1$ TIMES IN $Ack\_history$
38: **if** $Decided \subseteq Accepted\_set \wedge state = proposing \wedge r' = r$ **then**
39: DECIDE$(Accepted_set)$
40: $Decided = Accepted\_set$
41: $state = newround$
42: **function** SAFE(m)
43: **if** the lattice element contained in m is a subset of $SvS[r]$ **then**
44: **return** True
45: **else**
46: **return** False

VI. ALGORITHM *Generalized Wait Till Safe* (GWTS)

A. The Generalised Byzantine Lattice Agreement Problem

In the generalised version of our problem, each process p_i receives, asynchronously, input values from an infinite sequence $Pro_i = \langle pro_0, pro_1, pro_2, \ldots \rangle$ and it must output an infinite number of decision values $Dec_i = \langle dec_0, dec_1, dec_2, \ldots \rangle$. The sequence of decisions has to satisfy the following properties:

- **Liveness:** each correct process $p_i \in C$ performs an infinite sequence of decisions $Dec_i = \langle dec_0, dec_1, dec_2, \ldots \rangle$;

Algorithm 4 *GWTS* - Algorithm for Acceptor process p_i

1: $Accepted\_set = Waiting\_msgs = Ack\_history = \emptyset$
2: $SvS[]$ ▷ Reference to SvS in the corresponding Proposer
3: $Safe\_r = 0$ ▷ Max round for which it is safe to process messages
4: **upon event** DELIVERY OR RBCASTDEL FROM SENDER(m)
5: $Waiting\_msgs = Waiting\_msgs \cup \{m\}$
6: **upon event** $\exists m \in Waiting\_msgs | SAFEA(m) \wedge r \leq Safe\_r \wedge m =< ack\_req, Rcvd, ts, r >$
7: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
8: **if** $Accepted\_set \leq Rcvd$ **then**
9: $Accepted\_set = Rcvd$
10: RBCAST($< ack, Accepted\_set, sender, p_i, ts, r >$) to all
11: **else**
12: SEND TO SENDER ($< nack, Accepted\_set, ts, r >$)
13: $Accepted\_set = Accepted\_set \cup Rcvd$
14: **upon event** $\exists m \in Waiting\_msgs | SAFEA(m) \wedge r \leq Safe\_r \wedge m =< ack, Accepted\_set, destination, sender, ts, r > \wedge m$ WAS DELIVERED WITH RBCASTDEL
15: $Waiting\_msgs = Waiting\_msgs \setminus \{m\}$
16: $Ack\_history = Ack\_history \cup \{< ack, Accepted\_set, destination, sender, ts, r, >\}$
17: **upon event** $< ack, Accepted\_set, destination, \cdot, ts, r >$ APPEARS $\lfloor (n+f)/2 \rfloor + 1$ TIMES IN $Ack\_history$
18: **if** $r = Safe\_r$ **then**
19: $Safe\_r = Safe\_r + 1$
20: **function** SAFEA(m)
21: **if** $\exists r$ such that lattice element contained in m is a subset of $SvS[r]$ **then**
22: **return** True
23: **else**
24: **return** False

- **Local Stability:** For each $p_i \in C$ its sequence of decisions is non decreasing (i.e., $dec_h \subseteq dec_{h+1}$, for any $dec_h \in Dec_i$);
- **Comparability:** Any two decisions of correct processes are comparable, even when they happen on different processes;
- **Inclusivity:** Given any correct process $p_i \in C$, if Pro_i contains a value pro_k, then pro_k is eventually included in $dec_h \in Dec_i$;
- **Non-Triviality:** Given any correct process $p_i \in C$ if p_i outputs some decision dec_k at time t, then $dec_k \leq \bigoplus(Prop[0:h] \cup B[0:b])$, where, $Prop[0:h]$ is the union of the prefixes, until index h, of all sequences Pro_i of correct processes; and, $B[0:b]$ is the union of all prefixes, until index b, of f infinite sequences B_i, one for each Byzantine process.

Intuitively, with Non-Triviality we are bounding the number of values that the Byzantine processes could insert in any decision to a finite number of values.

B. Algorithm Description

The pseudocode of GWTS is in Algorithms 3-4. *Generalized Wait Till Safe* algorithm is an extension of the WTS algorithm based on the same batching approach proposed in [2]. Input values at proposers are batched until a new decision round starts. Each decision round follows the two-phases approach of WTS. Note that rounds are executed asynchronously at each proposer.[2]

Compared to WTS, an additional challenge to be faced is to prevent adversarial processes from indefinitely postponing the decision of correct processes. A uncareful design could allow Byzantine proposers to continuously pretend to have decided, thus jumping to new rounds, and clogging the proposers with a continuous stream of new values. This would make acceptors to continuously nack proposals of correct processes. We solve this problem through the acceptors. Acceptors will hep a new proposal to be decided in round $r \geq 1$ when, and if, in round $(r-1)$ a proposal has been accepted by at least a (Byzantine) quorum of acceptors (i.e., $safe\_r = r$). In order for this to work we make acceptors to reliably broadcast their ack messages, in this way the acceptance of proposals is made public. Any correct proposer can decide, in a round r, any proposal that has been correctly accepted in round r, even if it was not proposed by itself (provided that such decision preserves the Local Stability).

[2]The Byzantine reliable broadcast primitive used in [14] is designed to avoid possible confusion of messages in round based algorithms. That is exactly what we need.

C. Safety properties

The proof of the safety properties of GWTS is analogous to the proof contained in Section V-A. From the properties of reliable broadcast we have the following:

Observation 3. *For each correct process p_i and each round r, the set $SvS[r]$ contains at most n sets.*

Theorem 4. *Let us consider a set of processes, of size $3f + 1$, executing GWTS algorithm. Algorithm GWTS enforces: (1) Comparability; (2) Non-Triviality; (3) Stability.*

Proof. We prove each property separately.

1) is implied by Lemma 1 and Observation 2.
2) For the Non-Triviality we have to show that for each correct proposer p_i and each $dec_t \in Dec_i$ it holds $dec_t \leq \bigoplus(Prop[0:h] \cup B[0:b])$. First notice that dec_t may only contain values that are present in $\bigcup_{r' \in [0,r]} SvS[r']$ with r rounds in which dec_k happens. This derives from

the fact that a correct process, at a given round, only handles messages that contain safe values. Let $W_r = \bigcup_{r' \in [0,r]} SvS[r']$, it is immediate to see that W_r contains at most the union of all values in the prefixes $Prop_i[0:h]$ from some index h, which correspond precisely to all values proposed by correct processes until round r. It is also immediate that W_r contains at most the union of all values that Byzantine proposers have reliably broadcast in the first r disclosure phases: this is equivalent to say that it contains a prefix of all the infinite sequences of values that Byzantine cumulatively broadcast in the disclosure phases of our algorithm. From these arguments the Non-Triviality follows.

3) is ensured by line 38 in proposers. □

D. Liveness properties

We say that a correct process p_i joins a round r if it sends a message $< disclosure\_phase, \cdot, r >$. Similarly a correct process p_i proposes a Set at round r if it sends a message $< ack\_req, Set, \cdot, r >$.

We say that a message m is safe for a process p_i at round r, if $SvS[r]$ of p_i contains of all values contained in m.

Lemma 5. *Each message sent by a correct process at round r is eventually safe, at round r, for any other correct process.*

Proof. Same as Lemma 2 □

Definition 3. *We say that a round r has a "legitimate end" if there exists a proposal that has been committed at round r.*

(See Definition 2 of committed proposal)

Definition 4. *Round r is a legit round, at time t, if, either, r is 0 or $r-1$ had a legitimate end before time t.*

Definition 5. *An acceptor trusts round r if its $Safe\_r \geq r$.*

Lemma 6. *If r is a legit round, then eventually any correct acceptor will trust round r.*

Proof. The proof is by induction on r.

- **Base case**: for round $r = 0$ each acceptor has $Safe\_r$ initialized to 0.
- **Inductive case**: By inductive hypothesis we have that eventually any acceptor sets $Safe\_r = r-1$. Moreover, we have that, by definition of legit, round $(r-1)$ had a legitimated end. This means that $2f+1$ acceptors reliably broadcast $< ack, Accepted\_set, destination, \cdot, ts, r-1 >$, and at least one of them is correct process p, thus the message $< ack, Accepted\_set, destination, \cdot, ts, r-1 >$ is safe for p at round $r-1$. Any acceptor with $Safe\_r = r-1$ upon receipt of these messages will eventually process them, by Lemma 5, and it will set $Safe\_r = r$, see procedure starting at line 17. □

Note that by Lemma 6 we have that any legit round will be eventually trusted by all acceptors. Moreover, we can show that if r is a non-legit round at time t than it will not be trusted, at time t, by any correct acceptor.

Lemma 7. *If r is a non-legit round, at time t, that is, $r \neq 0$ and $(r-1)$ has not had a legitimate end before time t, then any correct acceptor has $Safe\_r < r$.*

Proof. The proof derives immediately from the definition of legitimate end and from line 17 in the acceptor code. □

Definition 6. *A value v has been disseminated, by time t, if, by time t, it was contained in a safe ack_req message for some round r and it has been received by $\lfloor (n+f)/2 \rfloor + 1$ correct acceptors that trusted round r.*

The observation below is a strengthen version of Lemma 1.

Observation 4. *If a value v has been disseminated by time t, then any proposal committed after time t will contain v.*

Proof. The proof is immediate by observing that a disseminated value is in the $Accepted\_set$ of $\lfloor (n-f)/2 \rfloor + 1$ correct acceptors (either by line 9 or 13 of Algorithm 4), and by using an argument similar of Lemma 1 proof. □

Lemma 8. *If round r has a legitimate end and at least $(n-f)$ correct proposers joined round r, then eventually any correct proposer, that joined round r, will decide in round r, and join round $(r+1)$.*

Proof. The proof is by contradiction. Let p_i be a correct process that joined round r but has not yet decided in round r. Note that p_i has to be in state $proposing$: by hypothesis $(n-f)$ correct proposers joined r, thus the guard at line 22 of Alg 3 has to be eventually triggered.

Since r has a legitimate end then there are $\lfloor (n+f)/2 \rfloor + 1$ reliable broadcast of messages $< ack, Accepted\_set, destination, \cdot, ts, r >$, and at least one of them has been generated by a correct process p, thus it is safe for p at round r. By Observation 4 we have that $Decided \subseteq Accepted\_set$, and, by Lemma 5, upon receipt of these messages p_i decides and joins round $r+1$. □

Lemma 9. *If r is a legit round, then any correct proposer eventually joins it.*

Proof. The proof is by induction on round number.

- **Base case**: round $r = 0$, by assumption it is a legit round, and by algorithm construction each correct proposer joins $r = 0$.
- **Inductive case**: The inductive hypothesis is that $(r-1)$ is a legit round and that each correct proposer joined it. We assume that r is a legit round, thus round $(r-1)$ had a legitimate end. Lemma 8 and the inductive hypothesis imply that any correct proposer joins r. □

Lemma 10. *If r is a legit round, then it will eventually have a legitimate end. Moreover, each correct proposer executes line 31, while its round variable is r, at most f times (that is it refines its proposal at most f times during its participation to round r).*

Proof. First observe, by Lemma 6 that each correct acceptor eventually trusts round r. Then observe that, until r does not

have a legitimate end, by Lemma 7, no correct acceptor will trust any round $r' > r$. Thus they will not process any message coming from round r'.

The above and Observation 3 bound the number of changes that correct acceptors perform in round r on their $Accepted\_set$ to a finite number. Therefore, there exists a time t after which each correct acceptor does not change anymore its $Accepted\_set$.

If a correct proposer, that joined round r, issues an ack_req after time t, then, by Lemma 5, and the above reasoning we have that such request will be committed. Once committed round r has a legitimate end. Now by Lemma 9 we have that eventually any correct proposer joins round r. It remains to show that some correct process issues a request after time t. Note that, when joining a new round, each correct process proposes its value. This proposal either is committed or refined (upon execution of line 31). In case of refinement a new set is immediately proposed. This ensures that either something was committed in round r before time t, or that something will be proposed after t. In both cases round r will have a legitimate end. Recall that all correct proposers eventually join round r (by Lemma 9), each of them only proposes the value constituted by $Batch[r]$, by the property of the Byzantine reliable broadcast and the safety of messages also Byzantine processes are constrained to propose at most f different values in round r. This means that a decision in round r can contain at most n new values with respect to the decision at round $(r-1)$. Since when a correct proposer executes line 25 it passed the guard at line 22, it is obvious that there at most f values missing in its proposal. From this Lemma 10, and thus the bound on the number of executions of line 31, follows immediately. □

Lemma 11. *if a correct process p_i joins a round r at time t, it also proposes, in round r, all values in Pro_i received before time t.*

Proof. Observe that a correct process joins a round r only if $(r-1)$ had a legitimate end. From Lemma 9 we have that all correct processes will eventually join round r, thus the if at line 22 will be passed. From the above, the atomicity of the local procedures, and the fact that a correct process cumulates in its $Proposed\_set$ all previous batches never removing any value (see line 18), our claim follows. □

From Lemma 11 we have the following observation:

Observation 5. *Given any correct process $p_i \in C$ and any value $v \in Pro_i$, we have that v is eventually disseminated.*

Theorem 5. *Let us consider a set of processes, of size $3f+1$, executing GWTS algorithm. We have that any run of GWTS ensures Liveness and Inclusivity.*

Proof. We prove each property separately:

1) Liveness: It is enough to show that there is an infinite sequence of legit rounds. This derives from Lemma 10, combining it with Lemma 8 and a simple induction on the round number. Lemmas 8, 9 ensure that in each round of such sequence all correct proposers decide.
2) Inclusivity: it derives from Observations 5, 4 and the fact that the sequence of decisions is infinite (see above). □

E. Message Complexity

GWTS executes a possibly infinite sequence of decisions. Thus, we restrict our message complexity analysis to the number of messages needed for each decision. The messages are counted per proposer, we include messages created by correct acceptors in response to proposer actions. Each proposer decides exactly once for each algorithm round. Therefore, we count messages from start to end of a generic round. A proposer has to reliably broadcast its batch (line 15 -cost $\mathcal{O}(n^2)$), it has to broadcast its proposal (line 25 - cost $\mathcal{O}(n)$), then, in the worst case, it refines its proposal at most f times (see Lemma 10 -line 33 - cost $\mathcal{O}(n)$), however each ack from a correct acceptor has to be reliably broadcast (line 10 - cost $\mathcal{O}(n^2)$). The total cost is therefore upper-bounded by $\mathcal{O}(f \cdot n^2)$.

VII. Byzantine tolerant RSM

We are interested in wait-free implementations of linearizable replicated state machines for commutative update operations in the Byzantine model.

A. Specification of the Byzantine tolerant RSM

The replicated state machine is composed of n replicas, which start in the initial empty state s_0. Among them, up to $f \leq (n-1)/3$ replicas may exhibit Byzantine failures. The RSM exposes two operations, update and read, such that the update operation with command cmd modifies the current state s of the RSM by applying cmd to s but does not return any value, while the read operation returns the current state of the RSM. The state of the RSM at time t is a set of update commands applied to the initial state s_0 until time t. Note that being the RSM commutative the order in which updates are applied does not matter. Clients may trigger an infinite number of read and update operations. We assume that each command is unique (which can be easily done by tagging it with the identifier of the client and a sequence number). We do not make any assumptions regarding clients behavior: they can exhibit arbitrary behaviors (e.g., invoke an update operation with some arbitrary command, or modify the read and update code). We do not limit the number of Byzantine clients. Hence, to prevent Byzantine clients from jeopardizing the state of the RSM through arbitrary commands, commands are locally executed by clients: the RSM provides clients with a set of updates and clients locally execute them. For readibility reasons, the value returned by the execution of a set of commands is equal to the set of commands. The following properties formalise the behavior of read and update operations during any execution run by correct clients:

- **Liveness** Any update and read operation completes;
- **Read Validity:** Any value returned by a read reflects a state of the RSM;

- **Read Consistency:** Any two values returned by any two reads are comparable;
- **Read Monotonicity:** For any two reads r_1 and r_2 returning value v_1 and v_2 respectively, if r_1 completes before r_2 is triggered then $v_1 \subseteq v_2$;
- **Update Stability:** If update u_1 completes before update u_2 is triggered then every read that returns a value that includes the command of u_2 also include the command of u_1;
- **Update Visibility:** If update u completes before read r is triggered then the value returned by r includes the command of u.

B. Implementation of the Byzantine tolerant RSM

As previously introduced, our general idea to implement a wait-free and linearizable replicated state machine resilient to Byzantine failures in an asynchronous system is to apply Generalized Lattice Agreement on the power set of all the update commands. GWTS is executed by the replicas of the state machine (for simplicity reasons replicas play the role of both proposers and acceptors). The update and read operations are presented in Algorithms 5 and 6 respectively. The update operation consists in submitting the new command cmd to generalized Lattice Agreement so that eventually the new state of each (correct) replica includes cmd. This is achieved by triggering the execution of NEW VALUE with $\{cmd\}$ as parameter at any subset of $(f+1)$ replicas (so that at least one correct replica will execute it), see Line 3. The update operation completes when some correct replica modifies its local state with cmd, that is, decides a decision value that includes cmd (Line 4). This preserves the order of non-overlapping update operations. The read operation consists in an update operation followed by a confirmation step. The update is triggered with a special value nop that locally modifies a replica's state as for an ordinary command cmd but is equivalent to a nop operation when executed. When the update completes, any decision value decided by a correct replica can be returned by the read operation. Since up to f Byzantine replicas may provide any value, a confirmation request for each of these $(f+1)$ decision values is sent to all replicas (Line 8). A replica acknowledges $Accepted\_set$ if $Accepted\_set$ has been accepted by $\lfloor (n+f)/2 \rfloor + 1$ acceptors, which ensures that $Accepted\_set$ has effectively been decided in GWTS. The value returned by the read operation is the result of the execution of the first decision value confirmed by $(f+1)$ replicas, i.e, the first decision value confirmed by at least one correct replica. This ensures that a read operation will return a value that reflects the effect of the last update operation. From an implementation point of view, the confirmation step requires to add two lines of code in Algorithm 3. Specifically, when a proposer receives a confirmation request for decision value $Accepted\_set$, then it acknowledges the request if $< ack, Accepted\_set, \cdot, \cdot, ts, r >$ appears $\lfloor (n+f)/2 \rfloor + 1$ times in its $Ack\_history$ set for a fixed combination of ts and round r. The pseudocode of this modification is in the full version.

Algorithm 5 *RSM - Update algorithm at a client*

```
1: procedure UPDATE(cmd)
2:     DecSet = ∅
3:     NEW VALUE ({cmd}) at (f + 1) REPLICAS
4:     wait until |DecSet| ≥ f + 1
5:     upon event RECEIPT FROM replica < DECIDE, Accepted_set, replica > WITH
       cmd ∈ Accepted_set
6:         DecSet = DecSet ∪ < DECIDE, Accepted_set, replica >
```

Algorithm 6 *RSM - Read algorithm at client c*

```
1: procedure READ
2:     DecSet = ConfSet = ∅
3:     NEW VALUE ({nop_{c,r}}) at (f + 1) REPLICAS
4:     upon event RECEIPT FROM replica < DECIDE, Accepted_set, replica > WITH nop_{c,r} ∈ Accepted_set
5:         DecSet = DecSet ∪ < DECIDE, Accepted_set, replica >
6:     upon event |DecSet| ≥ f + 1
7:         for all Accepted_set ∈ DecSet do
8:             SEND(< CNFREQ, Accepted_set >) to all replicas
9:     upon event RECEIPT FROM replica < CNFREP, Accepted_set, replica >
10:        ConfSet = ConfSet ∪ < CNFREP, Accepted_set, replica >
11:    upon event < CNFREP, Accepted_set, . > APPEARS f+1 TIMES IN ConfSet
12:        return EXECUTE (Accepted_set)
```

Theorem 6. *Given the wait-free Byzantine generalized Lattice Agreement (GLA) algorithm whose pseudocode is given in Alg. 3, 4, the above transformation yields a wait-free linearizable replicated state machine for commutative update operations. This transformation requires one execution of the Byzantine GLA algorithm.*

Proof. The proof consists in showing that (1) liveness, (2) read validity, (3) read consistency, (4) update stability, (5) read monotonicity, and (6) update visibility properties holds. We prove each property separately.

- (1) Liveness of the update operation is straightforward from Theorem 5. For the read operation, liveness holds from update liveness and from the fact that among the $(f+1)$ received values, at least one is the decision value of a correct replica, which by Lines 37- 39 of Algorithm 3, has been accepted and reliably broadcast to all proposers by $(2f+1)$ acceptors, and thus reliably delivered by all proposers (by Liveness of Reliable Broadcast);
- (2) Straightforward from the fact that the value returned by a read is a decision value.
- (3) Straightforward from Theorem 4;
- (4) By Observation 4 and by the fact that a read operation begins with an update operation;
- (5) By applying the same argument as for update stability, read monotonicity holds.
- (6) By applying the same argument as for update stability, update visibility holds. □

The proof of the following Lemma is omitted and can be found in the full version.

Lemma 12. *The above transformation is resilient to Byzantine clients.* □

VIII. Safety by Signature Algorithm - An algorithm with improved message complexity

In this Section we will discuss how to decrease the message complexity using signatures, as usual we assume that signatures cannot be forged by Byzantine processes. Our Safe by Signature (SbS) algorithm has a message complexity of $\mathcal{O}(n)$ when $f = \mathcal{O}(1)$. Details and proofs can be found in the extended version of the paper. SbS is divided in three phases:

- *Init*:in this phase each process broadcasts a signed version of its initial proposed value. A process collects these messages until it sees $n - f$ of it.
- *Safetying*: At the end of the init phase a correct process has a certain set of values. The purpose of the safetying phase is to make at least $|X \cup B| - 2f$ such values safe. A value v is safe if we are sure that no other process can see a different value v' that is also safe and has been initially sent by the same sender. Safetying is done by performing a broadcast of signed values obtained in the init phase towards the acceptors. Each acceptor keeps a set of values candidates to be safe. Once an acceptor receives a set of values from a proposer, it examines each value contained. If for a value v it has value sent by the same sender in its candidate set(check done using signature), it adds v to its candidate set. Otherwise, if it exists a v' from the same sender, it adds (v, v') to a temporary set of conflicts. The acceptor replies back to the proposer by sending a signed message containing the set received, and the set of conflicts. A proposer possesses a proof of safety for a value v if it receives $\lfloor (n+f)/2 \rfloor + 1$ messages from different acceptors in which v never appears as a conflict. The intuition behind this phase is that if the same Byzantine process injects two (or more) values signed by him in the init phase, then at most one of them could manage to get a correct proof of safety.
- *Proposing*: This phase is identical to the corresponding in WTS. The only difference is that correct proposers and acceptors refuse to process a message that contains a value without an attached proof of safety.

a) Message Complexity: The asymptotical complexity of our algorithm is the same of WTS once the cost of the Byzantine Reliable Broadcast has been removed, that is $\mathcal{O}(f \cdot n)$.

A. Generalising SbS

Adapting the SbS algorithm to its generalised version, while keeping the message complexity improvement, needs a special attention to substitute the reliable broadcast used to acknowledge in the GWTS (line 10 of Algorithm 4). We would like to replace such broadcast with a single point-to-point message. We do this by forcing an acceptor to sign its, now poin-to-point, ack. Intuitively, a proposer is able to prove others that it received an ack for its proposal. Each correct proposer broadcasts a special decided message before deciding, such message has attached all the acks used to decide. This would allow proposers and acceptors that receive a decided message to know that the sender of such message was allowed to decide by the algorithm rules (recall that acks are now signed). Additional details can be found in the full version.

a) Message Complexity: The message complexity follows the same analysis of Section VI-E, the removal of the Byzantine reliable broadcast leads to $\mathcal{O}(f \cdot n)$ messages per decision.

IX. Conclusions

We investigated Byzantine Lattice Agreement and we used it to build a byzantine tolerant RSM with commutative updated. Our main future line of investigation is to understand whether a message delay of $\mathcal{O}(f)$ is necessary or not. In the crash-stop model $\mathcal{O}(\log f)$ delays are sufficient [15]. A first step would be to investigate if the technique in [15] could be "Byzantined" while preserving the desirable delay. A final target is to understand the necessary number of message delays: even in the crash-stop model such knowledge is still missing.

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, 1985.

[2] J. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani, "Generalized lattice agreement," in *PODC*, 2012.

[3] N. M. Preguiça, C. Baquero, and M. Shapiro, "Conflict-free replicated data types crdts," in *Encyclopedia of Big Data Technologies*, 2019.

[4] H. Attiya, M. Herlihy, and O. Rachman, "Atomic snaphots using lattice agreement," *Dist. Comp.*, vol. 8, pp. 121–132, 1995.

[5] X. Zheng, C. Hu, and V. K. Garg, "Lattice agreement in message passing systems," in *DISC*, 2018.

[6] J. Skrzypczak, F. Schintke, and T. Schuütt, "Brief announcement: Linearizable state machine replication of state-based crdts without logs," in *PODC*, 2019.

[7] G. Di Luna, E. Anceaume, and L. Querzoni, "Byzantine generalized lattice agreement," https://arxiv.org/abs/1910.05768.

[8] A. J. H., "Composite registers," *Dist. Comp.*, vol. 6, no. 3, pp. 141–154, 1993.

[9] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *J. ACM*, vol. 40, no. 3, pp. 973–890, 1993.

[10] T. Nowak and J. Rybicki, "Byzantine approximate agreement on graphs," in *DISC*, 2019.

[11] J. B. Nation, "Notes on lattice theory," http://math.hawaii.edu/~jb/math618/Nation-LatticeTheory.pdf.

[12] G. Bracha, "Asynchronous byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, 1987.

[13] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Dist. Comp.*, vol. 2, no. 2, 1987.

[14] H. Mendes, M. Herlihy, N. H. Vaidya, and V. K. Garg, "Multidimensional agreement in byzantine systems," *Dist. Comp.*, vol. 28, no. 6, pp. 423–441, 2015.

[15] X. Zheng, V. K. Garg, and J. Kaippallimalil, "Linearizable replicated state machines with lattice agreement," https://arxiv.org/abs/1810.05871, 2018, arXiv:1810.05871.