Byzantine Lattice Agreement in Asynchronous Systems

Xiong Zheng
Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Vijay Garg
Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Abstract

We study the Byzantine lattice agreement (BLA) problem in asynchronous distributed message passing systems. In the BLA problem, each process proposes a value from a join semi-lattice and needs to output a value also in the lattice such that all output values of correct processes lie on a chain despite the presence of Byzantine processes. We present an algorithm for this problem with round complexity of $O(\log f)$ which tolerates $f < \frac{n}{5}$ Byzantine failures in the asynchronous setting without digital signatures, where n is the number of processes. This is the first algorithm which has logarithmic round complexity for this problem in asynchronous setting. Before our work, Di Luna et al give an algorithm for this problem which takes $O(f)$ rounds and tolerates $f < \frac{n}{3}$ Byzantine failures. We also show how this algorithm can be modified to work in the authenticated setting (i.e., with digital signatures) to tolerate $f < \frac{n}{3}$ Byzantine failures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine Lattice Agreement, Asynchronous

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.4

Related Version Full version hosted on `https://arxiv.org/abs/2002.06779`.

Funding *Vijay Garg*: This work was supported in parts by the National Science Foundation Grants CNS-1812349, CNS-1563544, and the Cullen Trust Endowed Professorship.

1 Introduction

In distributed systems, reaching agreement in the presence of process failures is a fundamental task. Understanding the kind of agreement that can be reached helps us understand the limitation of distributed systems with failures. Consensus [15] is the most fundamental problem in distributed computing. In this problem, each process proposes some input value and has to decide on some output value such that all correct processes decide on the same valid output. In synchronous message systems with crash failures, consensus cannot be solved in fewer than $f + 1$ rounds [9]. In asynchronous systems, consensus is impossible in the presence of even one crash failure [11]. The k-set agreement [5] is a generalization of consensus, in which processes can decide on at most k values instead of just one single value. The k-set agreement cannot be solved in asynchronous systems if the number of crash failures $f \geq k$ [3, 12]. The paper [6] shows that k-set agreement problem cannot be solved by less than $\lfloor \frac{f}{k} \rfloor$ rounds if $n \geq f + k + 1$ in crash failure model. The lattice agreement problem was proposed by Attiya et al [1] to solve the atomic snapshot object problem in shared memory systems. In this problem, each process $i \in [n]$ has input x_i and needs to output y_i such that the following properties are satisfied. 1) **Downward-Validity**: $x_i \leq y_i$ for each correct process i. 2) **Upward-Validity**: $y_i \leq \sqcup\{x_i \mid i \in [n]\}$. 3) **Comparability**: for any two correct processes i and j, either $y_i \leq y_j$ or $y_j \leq y_i$.

Attiya et al in [1] present a generic algorithm to transform any protocol for the lattice agreement problem to a protocol for implementing an atomic snapshot object in shared memory systems. This transformation can be easily implemented in message passing systems by replacing each read and write step with sending "*read*" and "*write*" messages to all and waiting for acknowledgements from $n-f$ different processes. Conversely, if we can implement an atomic snapshot object, lattice agreement can also be solved easily both on shared-memory and message passing systems with only crash failures. Thus, solving the lattice agreement problem in message passing systems is equivalent to implementing an atomic snapshot object in message passing systems with only crash failures.

Using lattice agreement protocols, Faleiro et al [10] give procedures to build a special class of linearizable and serializable replicated state machines which only support query operations and update operations but not mixed query-update operations. Later, Xiong et al [19] propose some optimizations for their procedure for implementing replicated state machines from lattice agreement in practice. They propose a method to truncate the logs maintained in the procedure in [10]. The recent paper [17] by Skrzypczak et al proposes a protocol based on generalized lattice agreement [10], which is a multi-shot version of lattice agreement problem, to provide linearizability for state based conflict-free data types [16].

In message passing systems with crash failures, the lattice agreement problem is well studied [1, 19, 20, 13]. The best upper bound for both synchronous systems and asynchronous systems is $O(\log f)$ rounds. In the Byzantine failure model, a variant of the lattice agreement problem is first studied by Nowak et al [14]. Then, Di Luna et al [8] propose a validity condition which still permits the application of lattice agreement protocol in obtaining atomic snapshots and implementing a special class of replicated state machines. They present an $O(f)$ rounds algorithm for the Byzantine lattice agreement problem in asynchronous message systems. For synchronous message systems, a recent preprint by Xiong et al [18] gives three algorithms. The first algorithm takes $O(\sqrt{f})$ rounds and has the early stopping property. The second and third algorithm takes $O(\log n)$ and $O(\log f)$ rounds but are not early stopping. All three algorithms can tolerate $f < \frac{n}{3}$ failures. They also show how to modify their algorithms to work for authenticated settings and tolerates $f < \frac{n}{2}$ failures. The preprint by Di Luna et al [7] presents an algorithm which takes $O(\log f)$ rounds and tolerates $f < \frac{n}{4}$ failures and shows how to improve resilience to $f < \frac{n}{3}$ by using digital signatures.

In this work, we present new algorithms for the Byzantine lattice agreement (BLA) problem in asynchronous message systems. In this problem, each process $i \in [n]$ has input x_i from a join semi-lattice $(X, \leq, \sqcup)$ with X being the set of elements in the lattice, $\leq$ being the partial order defined on X, and $\sqcup$ being the join operation. The lattice can be infinite. Each process i has to output some $y_i \in X$ such that the following properties are satisfied. Let C denote the set of correct processes in the system and f_a denote the actual number of Byzantine processes in the system.

Comparability: For all $i \in C$ and $j \in C$, either $y_i \leq y_j$ or $y_j \leq y_i$.

Downward-Validity: For all $i \in C$, $x_i \leq y_i$.

Upward-Validity: $\sqcup\{y_i \mid i \in C\} \leq \sqcup(\{x_i \mid i \in C\} \cup B)$, where $B \subset X$ and $|B| \leq f_a$.

The first two requirements are straightforward. Upward-Validity requires that the total number of values that can be introduced by Byzantine processes into the decision value of correct processes can be at most the number of actual Byzantine processes in the system. One may argue that if a Byzantine process proposes the largest element of the input lattice, then correct processes may always decide on the largest element. For applications, we can impose an additional constraint on the initial proposal of all processes. In the case of a Boolean lattice, we can require that the initial proposal for any process must be a singleton. More generally, we can impose the requirement that the initial proposal of any process must have the height less than some constant.

Our contribution is summarized in Table 1. First, we present an algorithm for the BLA problem in asynchronous systems without the digital signatures assumption which takes $O(\log f)$ rounds and $f < \frac{n}{5}$. The algorithm achieves exponential improvement in round complexity compared to the previous best algorithm in [8]. Then, we show how to improve the resilience to $f < \frac{n}{3}$ with the digital signatures assumption. The round complexity of our algorithm matches the best round complexity achieved in synchronous model [18].

Table 1 Our Results.

Model	Digital Signatures?	Reference	Rounds	Resilience
Synchronous	No	[18]	$O(\log f)$	$f < \frac{n}{3}$
	Yes		$O(\log f)$	$f < \frac{n}{2}$
Asynchronous	No	[8]	$O(f)$	$f < \frac{n}{3}$
	No	This paper	$O(\log f)$	$f < \frac{n}{5}$
	Yes			$f < \frac{n}{3}$

2 System Model

We assume a distributed asynchronous message system with n processes with unique ids in $\{1, 2, ..., n\}$. The communication graph is completely connected, i.e., each process can send messages to any other process in the system. We assume that the communication channel between any two processes is reliable. There is no upper bound on message delay. We assume that processes can have Byzantine failures but at most $f < n/3$ processes can be Byzantine in any execution of the algorithm. We use parameter f_a to denote the actual number of Byzantine processes in a system. By our assumption, we must have $f_a \leq f$. Byzantine processes can deviate arbitrarily from the algorithm. We say a process is correct or non-faulty if it is not a Byzantine process. We consider both systems with and without digital signatures. In a system with digital signatures, Byzantine processes cannot forge the signature of correct processes.

3 Algorithm for the Asynchronous model without Digital Signatures

In this section, we present an algorithm for the BLA problem in asynchronous systems which takes $O(\log f)$ rounds of asynchronous communication and tolerates $f < \frac{n}{5}$ Byzantine failures. Our algorithm applies a recursive approach similar to the algorithms designed for crash failure model in [20], which is inspired by the algorithm in [2] designed for atomic snapshot objects in shared memory systems. The high level idea of the recursive approach is to apply a classifier procedure to divide a group of processes into the slave subgroup and the master subgroup and update their values such that the values of the slave group is less than the values of the master group. Then, by recursively applying such a classifier procedure within each subgroup, eventually all processes have comparable values. In crash failure model, the classifier procedure only needs to guarantee the following two properties: (C1) The value of a correct slave process is at most the value of any correct master process, (C2) The size of the union of all values of correct slave processes is at most k, which is a threshold parameter associated with the classifier procedure and serves as knowledge threshold.

Suppose we have a classifier procedure in the crash failure model with properties (C1) and (C2). The binary tree in Fig. 1 shows how processes invoke the classifier procedure recursively. Each node in the tree represents a classifier procedure with its threshold parameter k shown

above the node. Before all processes traverse the tree and recursively invoke the classifier procedures along the way, an initial round is used to let all processes exchange their input values. After the initial round, each process obtains at least $n - f$ values. The threshold parameters of classifier procedures in the tree are set in a binary way with low equal to $n - f$ and high equal to n. The threshold parameter of the classifier procedure at the root node is set as $n - \frac{f}{2}$. Given a node with threshold parameter equal to k, the threshold parameter of its left child node and right child node are set as $k - \frac{f}{2^{r+1}}$ and $k + \frac{f}{2^{r+1}}$, respectively. Then, all processes traverse the binary tree starting from the root and invoke the classifier procedures along the way. After a specific classifier procedure invocation, processes classified as slave traverses to the left subtree and processes classified as master traverse to the right subtree. We can observe that all labels in the binary tree up to level $\log f$ are unique. The above properties (C1) and (C2) of the classifier procedure and our method to set the threshold parameter of each classifier procedure in the tree guarantee that 1) at level $\log f + 1$, processes in different nodes have comparable values, 2) at level $\log f + 1$, processes within the same node must have the same value. This will be formally proved when we present our algorithm.

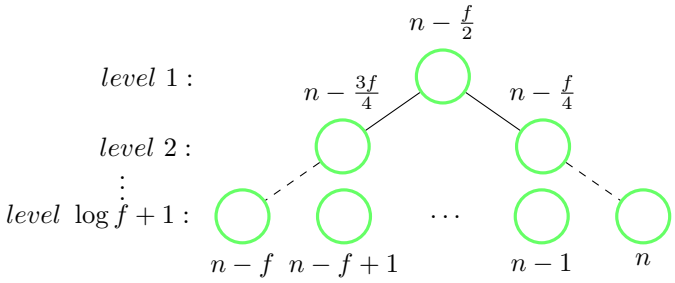

Figure 1 The Classification Tree.

In presence of Byzantine processes, (C1) and (C2) are not enough for recursively applying such classifier procedure within each subgroup. A Byzantine process in a slave group can introduce new values which are not known by some master process. To prevent that from happening, we introduce the notion of admissible values for a group (to be formally defined later), which is the set of values that processes in this group can ever have. We present a Byzantine tolerant classifier procedure with threshold parameter k which provides the following properties: (B1) Each correct slave process has $\leq k$ values and each correct master process has $> k$ values. (B2) The admissible values of the slave group is a subset of the value of any correct master process. (B3) The union of all admissible values in the slave group has size $<$ the threshold parameter k.

Suppose now we have a Byzantine tolerant classifier which guarantees the above properties.The main algorithm, shown in Fig. 2, proceeds in asynchronous rounds. The Byzantine tolerant classifier procedure can take multiple rounds. For ease of presentation, we call each round in the classifier as a subround. Each process i maintains a value set V_i which contains a set of values and is updated at each round by invoking the classifier procedure. Each process i has a label l_i, which is used as the threshold parameter when it invokes the classifier procedure. Initially, each process has the same label $k_0 = n - \frac{f}{2}$. The label of a process is updated at each round according to the classification tree. Each process i also keeps track of a map S_i, which we call *the safe value map*. $S_i[k]$ denotes the set of values that process i considers valid for label k. This safe value map is used by process i to restrict the admissible values of a group. In the main algorithm, a process uses the reliable broadcast primitive efined by Bracha [4] to send its value. When process i receives a value

broadcast by process j that is not in $S_i[j]$, it will not send echo this value to other processes. In the reliable broadcast primitive, a process uses **RB_broadcast** to send a message and uses **RB_broadcast** to reliably deliver a message. This primitive guarantees many nice properties. In our algorithm, we need the following two main properties: 1) If a message is reliably delivered by some correct process, then this message will eventually be reliably delivered by each correct process. 2) If a correct process reliably delivers a message from process p, then each correct process reliably delivers the same message from p.

Code for process i:
x_i: input value y_i: output value
l_i: label of process i. Initially, $l_i = k_0 = n - \frac{f}{2}$:
V_i^r: value set held by process i at round r of the algorithm
Map S_i: $S_i[k]$ denote the safe value set for group k

/* Initial Round */
1: **RB_broadcast**(x_i), wait for $n - f$ **RB_deliver**(x_j) from p_j
2: Set V_i^1 as the set of values reliably delivered
/* Round 1 to $\log f$ */
3: **for** $r := 1$ to $\log f$
4: $(V_i^{r+1}, class) := Classifier(V_i^r, l_i, r)$
5: **if** $class = master$ **then** $l_i := l_i + \frac{f}{2^{r+1}}$
6: **else** $l_i := l_i - \frac{f}{2^{r+1}}$
7: **end for**
8: $y_i := \sqcup\{v \in V_i^{\log f+1}\}$

Upon RB_deliver(x_j) from p_j
$S_i[k_0] := S_i[k_0] \cup x_j$

Figure 2 $O(\log f)$ Rounds Algorithm for the BLA Problem.

In the initial round at lines 1-2, process i **RB_broadcast** its input x_i to all and waits for **RB_deliver** from $n - f$ different processes. Then, it updates its value set to be the set of values reliably delivered at this round. When reliable delivering a value, process i adds this value into its safe value set for the initial group $k_0 = n - \frac{f}{2}$. The reliable delivery procedure is assumed to be running in background. So, the safe value set for the initial group keeps growing. By the properties of reliable broadcast, this safe value set can only contain at most one value from each process. This is used to ensure **Upward-Validity**.

After the initial round, we can assume that all values in the initial safe value set of each process are unique, which can be done by associating the sender's id with the value. At line 3-8, process i executes the classifier procedure (to be presented later) for $\log f$ rounds. At each round, it invokes the classifier procedure to decide whether it is classified as a slave or a master and then updates its value accordingly. At round r, if process i is a master, it updates its label to be $l_i := l_i + \frac{f}{2^{r+1}}$. Otherwise, if updates its label to be $l_i := l_i - \frac{f}{2^{r+1}}$.

By applying properties (B1)-(B3), we can show that any two correct process i and j in the same group at the end of round $\log f$ must have the same set of values. For any two processes in different group, by recursively applying property (B2), the values of one process must be subset of the values of the other process.

3.1 The Byzantine Tolerant Classifier

We present a classifier procedure that satisfies (B1)-(B3), shown in Fig. 4. It is inspired by the asynchronous classifier procedure given in [19] for the crash failure model. In the classifier procedure, each process stores a set of values received from other processes. We say a process writes a value to at least $n - f$ processes if it sends a "*write*" message containing the value to all processes and waits for $n - f$ processes to send acknowledgement back. We say a process reads from at least $n - f$ processes if it sends a "*read*" message to all and waits for at least $n - f$ processes to send their current values back. We say a process performs a write-read step if it writes its value to at least $n - f$ processes and reads their values.

In the asynchronous classifier procedure for the crash failure model [19], to divide a group into a slave subgroup and a master subgroup, each process in the group first writes its value to at least $n - f$ processes and then reads from at least $n - f$ processes. After that, each process checks whether the union of all values obtained has size greater than the threshold parameter k or not. If true, it is classified as a master process, otherwise, it is classified as a slave process. Slave processes keep their values the same. To guarantee the value of each slave process is $\leq$ the value of each master process, each master process performs a write-read step to write the values obtained at the read step to at least $n - f$ processes and read the values from them. Then it updates its value to be the union of all values read. The second read step guarantees the size of the union of values of slave processes is $< k$, since the last slave process which completes the write step must have read all values of slave processes.

Constructing such a classifier procedure in presence of Byzantine processes is much more difficult. In order to adapt the above procedure to work in Byzantine setting, we need to address the following challenges. First, in the write step or read step, when a process waits for at least $n - f$ different processes to send their values back, a Byzantine process can send arbitrary values. Second, simply ensuring that the values of a slave process is a subset of values of each master process is not enough, since a Byzantine process can introduce some values unknown to a master process in the slave group. For example, even if we can guarantee that the current value of each slave process is less that the value of each master process, in a later round, a Byzantine process can send some new value to a slave process which is unknown to some master process. This is possible in an asynchronous systems since messages can be arbitrarily delayed. Third, ensuring that the union of all values in the slave group has size at most k is quite challenging. A simple second read step does not work any more since the last process which completes the write step might be a Byzantine process.

To prevent the first problem, in the Byzantine classifier procedure, when a process wants to perform a write step or read step, it applies the reliable broadcast primitive to broadcast its value. When a process waits for values from at least $n - f$ processes, it only accepts a value if the value is a subset of the values reliably delivered by this process. By property of reliable broadcast, this ensures that each accepted value must be reliably broadcast by some process, which prevents Byzantine processes from introducing arbitrary values.

To tackle the second and third problem, the key idea is to restrict the values that a Byzantine process, which claims itself to be a slave process, can successfully reliable broadcast in later rounds. To achieve that, first we require that that a slave process can only reliable broadcast the value that it has reliably broadcast in the previous round. This prevents Byzantine processes from introducing arbitrary new values into a slave group. Second, we require each process which claims itself as a slave process to prove that it is indeed classified as a slave at the previous round when it tries to reliable broadcast a value at the current round by presenting the set of values it used to do classification. To enforce the above two requirements, we add a validity condition when a process echoes a message in the reliable

broadcast primitive. However, this is not enough, since the value of a Byzantine slave process might not be known to a master process if the value of the Byzantine process is arbitrarily delayed. To ensure that the value a Byzantine process reliably broadcast is read by each correct master process, we force a Byzantine process who wants to be able to reliable broadcast a value in the slave group at next round to actually write its value to at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes, i.e., at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes must have received the value of a Byzantine process before each correct master process tries to read from at least $n - 2f$ correct processes. These two sets of correct processes must have at least one correct process in common since $f < \frac{n}{5}$.

BRB_broadcast$(type, pf, v, k, r)$
type denotes the type of the message to be sent, either "*write*" or "*read*"
pf is an array which is a proof of sender's group identity
v is the value to be sent, k is the label of the sender, r is the round number

Broadcast INIT$(i, type, pf, v, k, r)$ to all

Upon receiving INIT$(j, t_j, pf_j, v_j, k_j, r_j)$
  if (first reception of INIT$(j, t_j, -, -, -, r_j)$
    wait until $valid(t_j, pf_j, v_j, k_j, r_j)$ /* The *valid* function is defined in Fig. 5 */
    Broadcast ECHO(j, t_j, v_j, k_j, r_j)

Upon receiving ECHO$(j, t_j, pf_j, v_j, k_j, r_j)$
  if ECHO$(j, t_j, pf_j, v_j, k_j, r_j)$ is received from at least $\lfloor \frac{n+f}{2} \rfloor + 1$ different processes $\wedge$ READY(j, v_j, k_j, r_j) has not yet broadcasted
    Broadcast READY$(j, t_j, pf_j, v_j, k_j, r_j)$

Upon receiving READY$(j, t_j, pf_j, v_j, k_j, r_j)$
  if READY$(j, t_j, pf_j, v_j, k_j, r_j)$ received from $f + 1$ processes $\wedge$ READY$(j, t_j, pf_j, v_j, k_j, r_j)$ has not been broadcasted
    Broadcast READY$(j, t_j, pf_j, v_j, k_j, r_j)$
  if READY$(j, t_j, pf_j, v_j, k_j, r_j)$ received from $2f + 1$ processes $\wedge$ $(j, t_j, pf_j, v_j, k_j, r_j)$ has not been delivered
    BRB_deliver$(j, t_j, pf_j, v_j, k_j, r_j)$

Figure 3 Bounded Reliable Broadcast.

Each process which is classified as master is not required to prove its group identity but the value it tries to broadcast has to be a subset of safe value sets of correct processes. To ensure that the value of a slave process is less than the value of a master process, a master process needs to do a write-read step after it is classified as a master process.

Bounded Reliable Broadcast. Before explaining the Byzantine classifier procedure in detail, we modify the reliable broadcast primitive by adding a condition when a process echoes a broadcast message. This condition restricts the admissible values for each group. For completeness, the modified reliable broadcast procedure is shown in Fig. 3. When a process reliable broadcasts a value, it also includes the round number, its current label and a proof of its group identity. The proof is an array of size n denoting the values read by the sender at previous round, which will be explained in detail when we present the classifier

procedure.When a process i receives a broadcast message from process j, it waits for the validity condition to hold and then echoes the message. We say a process BRB_broadcasts a message if it executes **BRB_broadcast** procedure with the message. We say a process BRB_delivers a message if it executes **BRB_deliver** with this message.

Groups and Admissible Values. In our algorithm, each process i has a label l_i, which serves as the threshold when it invokes the classifier procedure. The notion of group defined as below is based on labels of processes.

▶ **Definition 1** (group). *A group is a set of processes which have the same label. The label of a group is the label of the processes in this group. The label of a group is also the threshold value processes in this group use to do classification.*

We also use label to indicate a group. A process is in group k if its message is associated with label k. Initially all processes are within the same group with label $k_0 = n - \frac{f}{2}$. The label of each process is updated at each round based on the classification result. For group k at round r, let $s(k,r) = k - \frac{f}{2^{r+1}}$ and $m(k,r) = k + \frac{f}{2^{r+1}}$. We introduce the notion of admissible values for a group, which is the set of values that processes in the group can ever have.

▶ **Definition 2** (admissible values for a group). *The admissible values for a group G with label k is the set of values that can be reliably delivered with label k if they are reliably broadcast by some process (possibly Byzantine) with label k.*

In our classifier, each process in group k updates its value set to a subset of the values which are reliably delivered with label k. Thus, the value set of each process in group k must be a subset of the admissible values for group k.

3.2 The Classifier Procedure

The classifier procedure for process $i \in [n]$, shown in Fig. 4, has three input parameters: V is the current value set of process i, k is the threshold value used to do the classification, which is also the current label of process i, and r is the round number.

In lines 1-2, process i writes its current value set to at least $n - f$ processes by using the **BRB_broadcast** procedure to send a "*write*" message. If process i is classified as a slave at the previous round, it needs to include the array of values it read from at least $n - f$ processes at previous round as a proof of its group identity. This proof is used by every other process in the *valid* function to decide whether to echo the "*write*" message or not. When process i BRB_delivers a "*write*" message with label k at round r, it includes the value in it into its safe value set for group $m(k,r)$. The safe value set is used to restrict the set of values that can be delivered in the master group $m(k,r)$. Due to this step, we can see that the admissible values in the master subgroup must be a subset of the admissible values at the current group. Process i also includes the value contained in the "*write*" message into $ACV_i^r[k]$, which stores the set of values reliably delivered with label k at round r.

From line 3 to line 4, process i reads values from at least $n - f$ processes by using the **BRB_broadcast** procedure to send a "*read*" message to all. In the *valid* function, each process j echos a "*read*" message from process i only if it has BRB_delivered the "*write*" message from process i sent at line 2. This step is used to ensure that for any process, possibly Byzantine, to read from other processes, it must have written its value to at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes, otherwise it cannot have enough processes echo its "*read*" message in the **BRB_broadcast**. When process i BRB_delivers a "*read*" message with label k from process j at round r, it records the set of values it has reliably delivered with

Classifier(V, k, r) **for** p_i**:**
V: input value set k: threshold value r: round number
/* Each process $i \in [n]$ keeps track of the following variables */
Array LB_i^r. $LB_i^r[j]$ denotes the label of process j sent along its values at round r
Map S_i. $S_i[k]$ denotes a safe value set for group k
Map ACV_i^r. $ACV_i^r[k]$ denotes the set of values accepted with label k, initially $ACV_i^r[k] := \emptyset$
Map RV_i^r. $RV_i^r[j]$ denote the values process i read from process j at round r at line 4
Map RT_i^r. $RT_i^r[j]$ denote the values process j read from process i at round r.

/* write step*/
1: **if** $isSlave(i, k, r)$ **then** $pf := RV_i^{r-1}$ **else** $pf := \emptyset$
2: BRB_broadcast("$write$", pf, V, k, r), wait for $wack(-, r)$ from $n - f$ different processes

/* read step*/
3: BRB_broadcast("$read$", $-, -, k, r$), wait for $n - f$ $rack(R_j, r)$ s.t. $R_j \subseteq ACV_i^r[k]$ from p_j
4: Set $RV_i^r[j] := R_j$ if $R_j \subseteq ACV_i^r[k]$, otherwise $RV_i^r[j] := \emptyset$

/* Classification */
5: Let $T := \bigcup_{j=1}^{n} RV_i^r[j]$
6: **if** $|T| > k$
/* write-read step */
7: Send $master(T, k, r)$ to all, wait for $n - f$ $mack(R_j, r)$ from p_j s.t. $R_j \subseteq ACV_i^r[k]$
8: Define $T' := \cup\{R_j \mid R_j \subseteq ACV_i^r[k], j \in [n]\}$
9: **return** (T', *master*)
10: **else**
11: **return** (V, *slave*)

Upon BRB_Deliver$(j, type, -, v, k, r)$
if $type =$ "$write$"
$S_i[m(k, r)] := S_i[m(k, r)] \cup v$ /* Construct safe value set for group $m(k, r)$ */
$ACV_i^r[k] := ACV_i^r[k] \cup v$
$LB_i^r[j] := k$ /* Record the label of a process at round r */
Send message $wack(-, r)$ to p_j
elif $type =$ "$read$"
$RT_i^r[j] := ACV_i^r[k]$
Send message $rack(ACV_i^r[k], r)$ to p_j

Upon receiving $master(T, k, r)$ from p_j
wait until $T \subseteq ACV_i^r[k]$
Send message $mack(ACV_i^r[k], r)$ to p_j

Figure 4 *The Byzantine Tolerant Classifier Procedure.*

label k in $RT_i^r[j]$. Then process i sends back a *rack* message along with the set of reliably delivered values with label k at round r to process j. At line 3, after the "*read*" message is sent, process i has to wait for valid *rack* message from $n-f$ processes. A *rack* message is valid if the value set contained in it is a subset of $ACV_i^r[k]$, which is the set of values reliably delivered with label k at round r. Consider a $rack(R_j, r)$ message from a correct process j. Since j is correct, each value in R_j must have been reliably delivered by process j. By property of reliable broadcast, each value in R_j will eventually be reliably delivered by process i, thus $R_j \subseteq ACV_i^r[k]$. Thus, eventually process i can obtain $n-f$ valid *rack* message. To implement line 3, we need a concurrent thread to check the wait condition whenever a new message is reliably delivered and added into $ACV_i^r[k]$. At line 4, process i records the set of valid R_j's obtained at line 3 into array RV_i^r. So, this array stores the values reliably delivered with label k that process i read from all processes. This array is used to do classification in line 5-11 and also used as the proof of group identity of process i when it writes at next round.

Line 5-11 is the classification step. Process i is classified as a master process if the size of the union of valid values obtained in the read step is greater than its label k, otherwise, it is classified as a slave process. If it is classified as a slave process, it returns its input value set. If it is classified as a master process, process i performs a write-read step by sending a *master* message which includes the set of values it uses to do classification to all and wait for $n-f$ valid *mack* message back at line 7. Similar to line 3, a *mack* message is valid if each value contained in it has been reliably delivered with correct label. When a process receives a *master* message with value set T and label k at round r, it first waits until all values in T are reliably delivered. Then it sends back a *mack* message along with the set of values reliably delivered with label k at round r. The waiting is used to ensure that each value in T is valid, i.e., be reliably delivered, because a Byzantine process can send arbitrary values in its *master* message at line 7. By a similar reasoning as line 3, process i will eventually obtain valid *mack* message from at least $n-f$ different processes. After the write-read step, at line 8, process i updates its value set to be the union of values obtained at line 7.

function $valid(j, type, pf, v, k, r)$ for process i:
  if $(type = \text{“}write\text{”} \wedge \neg isSlave(j,k,r) \wedge v \subseteq S_i[k])$
    $\vee$ $(type = \text{“}write\text{”} \wedge isSlave(j,k,r) \wedge \text{BRB\_deliver}(j, \text{“}write\text{”}, -, v, LB_i^{r-1}[j], r-1)$
      $\wedge\ pf[i] = RT_i^{r-1}[j] \wedge |\bigcup_{j=1}^{n} pf[j]| \leq LB_i^{r-1}[j])$
    $\vee$ $(type = \text{“}read\text{”} \wedge \text{BRB\_deliver}(j, \text{“}write\text{”}, -, -, k, r))$
    return $True$
  else
    return $False$

function $isSlave(j, k, r)$ for process i:
  if $k = LB_i^{r-1}[j] - \frac{f}{2^r}$
    return $True$
  else
    return $False$

Figure 5 The *valid* Function.

The *valid* function is defined in Fig. 5. In the this function, we first consider the "*write*" messages. If the message has been sent by a process that claims to be a master, then it is considered valid if the value v in this message is contained in the safe value set $S_i[k]$. If

the message has been sent by a process that claims to be a slave, then process i checks (1) whether process i has BRB_delivered the "*write*" message containing the same value at the previous round, (2) whether the i^{th} entry in pf array matches the value process j read from i in the previous round, and (3) whether the the number of values contained in the proof pf is at most k. The condition (1) ensures that a slave process sends the same value as the previous round since a correct slave process must keep its value same as in the previous round. The condition (2) ensures that the proof sent by the slave process uses values that it read at round $r-1$. The condition (3) checks that the sender classified itself correctly.

If the message is a "*read*" with label k at round r, process i considers it as valid if it BRB_deliverd a "*write*" message with label k at round r from the sender. This is used to make sure that the sender (possibly Byzantine) must complete its write step in line 1-2 before trying to read at line 3-4.

The *isSlave* function invoked in the *valid* function simply checks whether the label of the sender matches the label update rule by comparing it with the label at previous round.

3.3 Proof of Correctness

We first define the notion of *committing* a message. Due to space limitation, we omit the proof of most lemmas. The notations used in our proof are listed in Table. 2.

▶ **Definition 3.** *We say a process* **commits** *a message if it reliably broadcasts the message and the message is reliably delivered. A process* **commits** *a message at time t if this message is reliably delivered by the first process at time t.*

Table 2 Notations.

Variable	Definition
G	A group of processes at round r with label k
$slave(G)$	The slave subgroup of G, i.e., the processes with label $s(k,r)$ at round $r+1$
$master(G)$	The master subgroup of G, i.e., the processes with label $m(k,r)$ at round $r+1$
V_i^r	The value set of process i at the beginning of round r
S_i^r	The safe value map of process i at the beginning of round r $S_i^r[k]$ is the safe value set of process i for group k at the beginning of round r
U_k^r	The set of admissible values for group k at round r, i.e., the set of values that can be committed along with a "*write*" message at round r with label k

By properties of reliable broadcast, we observe that each process (possibly Byzantine) can commit at most one "*write*" message and at most one "*read*" message at each round. Define $s(k,r) = k - \frac{f}{2^{r+1}}$ and $m(k,r) = k + \frac{f}{2^{r+1}}$. The variables we use in the proof are shown in Table. 2. Consider the classification step in group k at round r. The following lemma shows that if a Byzantine process wants to commit a "*write*" message m at round $r+1$ with a slave label, then it must commit a "*write*" message m' which contains the same value as m and a "*read*" message at round r with label k. Also, it must commit its "*read*" message before its "*write*" message at round r with label k.

▶ **Lemma 4.** *Suppose that process i (possibly Byzantine) commits a write message $(i, \text{"}write\text{"}, -, V_i, s(k,r), r+1)$. Then*

1) *The message $(i, \text{"}read\text{"}, -, -, k, r)$ and the message $(i, \text{"}write\text{"}, -, V_i, k, r)$ must be committed by process i.*

2) *Let t denote the time that message $(i, \text{“}read\text{”}, -, -, k, r)$ is committed. Then, the message $(i, \text{“}write\text{”}, -, V_i, k, r)$ must have been reliably delivered by at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes before time t.*

The following lemma shows that the classifier provides the properties we defined.

▶ **Lemma 5.** *Let G be a group at round r with label k. Let L and R be two nonnegative integers such that $L < k \leq R$. If $L < |V_i^r| \leq R$ for each correct process $i \in G$, and $|U_k^r| \leq R$, then*

(p1) *For each correct $i \in master(G)$, $k < |V_i^{r+1}| \leq R$*
(p2) *For each correct $i \in slave(G)$, $L < |V_i^{r+1}| \leq k$*
(p3) $U_{s(k,r)}^{r+1} \subseteq U_k^r$
(p4) $U_{m(k,r)}^{r+1} \subseteq U_k^r$
(p5) $|U_{m(k,r)}^{r+1}| \leq R$
(p6) $|U_{s(k,r)}^{r+1}| \leq k$
(p7) *For each correct $j \in master(G)$, $U_{s(k,r)}^{r+1} \subseteq V_j^{r+1}$*
(p8) *Each correct $i \in slave(G)$ can commits its value set at round $r+1$, i.e., $V_i^{r+1} \subseteq U_{s(k,r)}^{r+1}$*
(p9) *Each correct $j \in master(G)$ can commit its value set at round $r+1$, i.e., $V_j^{r+1} \subseteq U_{m(k,r)}^{r+1}$*
(p10) $|\cup \{V_i^{r+1} \mid i \in slave(G) \cap C\}| \leq k$ *(p11)* $|\cup \{V_i^{r+1} \mid i \in master(G) \cap C\}| \leq R$

Proof.
(p1)-(p5): Implied by how processes are classified as slave or master in the classifier.
(p6): (Sketch) Let P denote the set of processes who can commit a write message at round $r+1$ with label $s(k,r)$. Part 2) of Lemma 4 implies that the write message of each $i \in P$ at round r must have been reliably delivered by at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes. Let $l \in P$ be the last process s.t its write message at round r is reliably delivered by at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes. Process l must have read all the values written by processes in P at round r due to quorum intersection. Due to quorum intersection and the condition to which processes echo write messages, process l must have read all values in $U_{s(k,r)}^{r+1}$ at round r and l is classified as slave at round r, which indicates that $U_{s(k,r)}^{r+1}$.
(p7): (Sketch) Let P denote the set of processes who commit a “*write*” message at round $r+1$ with label $s(k,r)$. Lemma 4 implies that the write message of each process in P must have been reliably delivered by at least $\lfloor \frac{n+f}{2} \rfloor + 1 - f$ correct processes. The condition to which correct processes echo write messages implies that at round $r+1$, each process in P sends the same value as round r in its write message. Quorum intersection guarantees that each master process must have read the values of each process in P in its reading step at round r. Thus, $U_{s(k,r)}^{r+1} \subseteq V_j^{r+1}$ for each j.
(p8): Since process i is correct, at round r, it must read from at least $n-2f$ correct processes. Let Q denote this set of correct processes. Then, at round $r+1$, each process in Q will echo i's write message. Thus, there will be $\geq n-2f$ echo messages. Since $f < \frac{n}{5}$, we have $n - 2f \geq \lfloor \frac{n+f}{2} \rfloor + 1$. Hence, the write message of i will be eventually reliably delivered.
(p9): (Sketch) Any value in V_j^{r+1} will eventually be reliably delivered by each correct process and be included into the safe value set of each correct process for the group with label $m(k,r)$. V_j^{r+1} will be reliable delivered by each correct process at round $r+1$.
(p10)-(p11): $(p10)$ is implied by $(p8)$ and $(p6)$. $(p11)$ is implied by $(p9)$ and $(p5)$. ◀

The following lemma shows that the value set of a correct process is non-decreasing.

▶ **Lemma 6.** *For any correct process i and round r, $V_i^r \subseteq V_i^{r+1}$.*

The following lemma is used later to show that processes in the same group at the end of the algorithm must have the same set of values.

▶ **Lemma 7.** *Let G be a group of processes at round r with label k. Then*
(1) *for each correct process $i \in G$, $k - \frac{f}{2^r} \leq |V_i^r| \leq k + \frac{f}{2^r}$*
(2) $|U_k^r| \leq k + \frac{f}{2^r}$

Proof. By induction on round number r and apply $(p1)$-$(p2)$ and $(p5)$-$p(6)$ of Lemma 5. ◀

▶ **Lemma 8.** *Let i and j be two correct processes that are within the same group G with label k at the beginning of round $\log f + 1$. Then $V_i^{\log f+1}$ and $V_j^{\log f+1}$ are equal.*

Proof (Sketch). By applying Lemma 7 at round $\log f$ within the parent group of G, we can show that $k' < |V_i^{\log f+1}| \leq k' + 1$ and $|\cup\{V_i^{\log f+1}, V_j^{\log f+1}\}| \leq k' + 1$, where k' is the label of the parent group of G. Thus, $V_i^{\log f+1} = V_j^{\log f+1}$. ◀

▶ **Lemma 9** (Comparability). *For any two correct process i and j, y_i and y_j are comparable.*

Proof. If process i and j are in the same group at the beginning of round $\log f + 1$, then by Lemma 8, $y_i = y_j$. Otherwise, let G be the last group that both i and j belong to. Suppose G is a group with label k at round r. Suppose $i \in slave(G)$ and $j \in master(G)$ without loss of generality. Then, $V_i^{\log f+1} \subseteq U_{s(k,r)}^{r+1} \subseteq V_j^{r+1} \subseteq V_j^{\log f+1}$, by $(p8)$, $(p6)$ $(p7)$ and $(p5)$ of Lemma 5 and Lemma 6. ◀

▶ **Theorem 10.** *There is an $O(\log f)$ rounds algorithm for the BLA problem in asynchronous systems which can tolerate $f < \frac{n}{5}$ Byzantine failures, where n is the number of processes in the system. The algorithm takes $O(n^2 \log f)$ messages.*

4 An $O(\log f)$ Rounds Algorithm for the Authenticated BLA Problem

In this section, we present an $O(\log f)$ round algorithm for the BLA problem in authenticated (i.e., assuming digital signatures and public-key infrastructure) setting that can tolerate $f < \frac{n}{3}$ Byzantine failures by modifying the Byzantine tolerant classifier procedure in previous section. The Byzantine classifier procedure in authenticated setting is shown in Fig. 6. The primary difference lies in what a process does when it reliably delivers some message and the validity condition for echoing a broadcast message. The basic idea is to let a process sign the *ack* message that it needs to send. Each process uses the set of signed *ack* messages as proof of its completion of a write step or read step. In this section, we use $\langle x \rangle_i$ to denote a message x signed by process i, i.e., $\langle x \rangle_i = \langle x, \sigma \rangle$, where σ is the signature produced by process i using its private signing key. We say a message is correctly signed by process i if the signature within the message is a correct signature produced by process i.

The Authenticated Byzantine Tolerant Classifier. The classifier in the authenticated setting is shown in Fig. 6. The primary difference between the classifier in previous section and the authenticated classifier is that in the authenticated classifier each process uses signed messages as proof of its group identity.

At lines 1-2, each process writes its current value set by using the **BRB_broadcast** procedure to send a "*write*" message. If the process is a slave process, it also includes the set of at least $n - f$ signed *rack* messages it received at the previous round as a proof that it is indeed classified as a slave. At line 2, each process waits for correctly signed *wack* message from at least $n - f$ different processes. This set of signed *wack* message is used as the proof

of its completion of the write step when this process tries to read from other processes. When a process BRB_delivers a "*write*" message, it performs similar steps as the algorithm in previous section except that it sends a signed *wack* message back.

Classifier(V, k, r)**:**
V: input value set k: threshold value r: round number
Each process $i \in [n]$ keeps track of the same variables as the classifier in Fig. 4
Set RV_i^r, which stores the set of signed *rack* message in the read step of previous round

1: **if** $isSlave(k, r)$ **then** $pf := RV_i^{r-1}$ **else** $pf := \emptyset$
2: BRB_broadcast("$write$", pf, v, k, r), wait for $n - f$ valid $\langle wack(-, r)\rangle_j$ from p_j
3: Let W denote the set of $\langle wack\rangle_j$ delivered at line 2

4: Send $read(W, k, r)$ to all, wait for $n - f$ valid $\langle rack(R_j, r)\rangle_j$ s.t. $R_j \subseteq ACV_i^r[k]$ from p_j
5: Set $RV_i^r := \{\langle rack(R_j, r)\rangle_j \mid R_j \subseteq ACV_i^r[k]\}$

6: Let $T := \cup\{R_j \mid R_j \subseteq ACV_i^r[k]\}$
7: **if** $|T| > k$ /* Size of T is greater than the threshold */
8: Send $master(T, k, r)$ to all, wait for $n - f$ $mack(R_j, r)$ s.t. $R_j \subseteq ACV_i^r[k]$ from p_j
9: Define $T' := \cup\{R_j \mid R_j \subseteq ACV_i^r[k]\}$
10: **return** (T', *master*)
11: **else**
12: **return** (V, *slave*)

Upon BRB_deliver(j, t, v, k, r)
if $t =$ "$write$"
$S_i[m(k, r)] := S_i[m(k, r)] \cup v$, $ACV_i^r[k] := ACV_i^r[k] \cup v$
Send message $\langle wack(ACV_i^r[k], r)\rangle_i$ to p_j

Upon receiving $read(W, k, r)$ from p_j
if $validSignature$("$read$", j, W, r)
Send message $\langle rack(ACV_i^r[k], r)\rangle_i$ to p_j

Upon receiving $master(T, k, r)$ from p_j
wait until $T \subseteq ACV_i^r[k]$
Send message $mack(ACV_i^r[k], r)$ to p_j

Figure 6 *The Authenticated Byzantine Tolerant Classifier.*

At line 4-5, each process reads from at least $n - f$ processes. Different from the classifier procedure in previous section, each process directly sends a read message along with the set of correctly signed *wack* messages obtained at line 2 to all (instead of using the **BRB_broadcast** procedure). When a process receives a "*read*" message with label k for round r, if uses the *validSignature* function to check whether the "*read*" message contains correctly signed *wack* message for round r from at least $n - f$ different processes. If so, it sends back to the sender a signed *rack* message along with the reliably delivered values with label k at round r. This ensures that if a process (possibly Byzantine) tries to read from correct processes, it must complete its write step first.

The classification step from line 6-12 is the same as the classification step of the algorithm in previous section. A mater process performs a write-read step by sending a *master* message along the set of value obtained at line 6. Then it waits for $n - f$ valid *mack* messages and updates its value set to be the set of values contained in these messages. When a process receives a *master* message, it performs the same steps as in the classifier in previous section.

The *valid* function is different from the one given in previous section. First, only "*write*" messages are reliably broadcast. Second, the proof is a set of signed *rack* messages instead of an array in previous section. To verify the proof, the valid function invokes the *validSignature* function to check whether the proof contains correctly signed *rack* message for previous round from at least $n - f$ different processes.

```
function valid(j, type, pf, v, k, r):
    if (type = "write" ∧ ¬isSlave(j, k, r) ∧ v ⊆ S_i[k])
        ∨ (type = "write" ∧ isSlave(j, k, r) ∧ BRB_deliver(j, "write", −, v, LB_i^{r−1}[j], r − 1)
              ∧validSignature("write", pf, r) ∧ pf contains at most k distinct values
        return True
    else
        return False

function validSignature(type, pf, r):
    if (type = "write" ∧ pf contains correctly signed rack(−, r − 1) from n − f processes)
        ∨  (type = "read" ∧ pf contains correctly signed wack(−, r) from n − f processes)
        return True
    else
        return False
```

Figure 7 The *valid* Function.

For the proof of correctness, we just need to prove the classifier procedure satisfies the properties given Lemma 5 under the assumption that $f < \frac{n}{3}$.

▶ **Lemma 11.** *Properties* $(p1) - (p11)$ *of Lemma 5 hold for the authenticated Byzantine tolerant classifier.*

▶ **Theorem 12.** *There is an* $O(\log f)$ *rounds algorithm for the BLA problem in authenticated asynchronous systems which can tolerate* $f < \frac{n}{3}$ *Byzantine failures, where* n *is the number of processes in the system. The algorithm takes* $O(n^2 \log f)$ *messages.*

5 Conclusion

In this paper, we present an $O(\log f)$ rounds algorithm for the Byzantine lattice agreement problem in asynchronous systems which can tolerates $f < \frac{n}{5}$ Byzantine failures. We also give an $O(\log f)$ rounds algorithm for the authenticated setting that can tolerate $f < \frac{n}{3}$ Byzantine failures. One open problem left is to design an algorithm which has resilience of $f < \frac{n}{3}$ and takes $O(\log f)$ rounds.

References

1. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
2. Hagit Attiya and Ophir Rachman. Atomic snapshots in o (n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
3. Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.
4. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
5. Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.
6. Soma Chaudhuri, Maurice Herlihy, Nancy A Lynch, and Mark R Tuttle. Tight bounds for k-set agreement. *Journal of the ACM (JACM)*, 47(5):912–943, 2000.
7. Giuseppe Antonio Di Luna, Emmanuelle Anceaume, Silvia Bonomi, and Leonardo Querzoni. Synchronous byzantine lattice agreement in $\mathcal{O}(\log f)$ rounds. *arXiv preprint arXiv:2001.02670*, 2020.
8. Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. *arXiv preprint arXiv:1910.05768*, 2019.
9. Danny Dolev and H Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
10. Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
11. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
12. Maurice Herlihy, Sergio Rajsbaum, and Mark R Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 133–142, 1998.
13. Marios Mavronicolasa. A bound on the rounds to reach lattice agreement. *http://www.cs.ucy.ac.cy/ mavronic/pdf/lattice.pdf*, 2018.
14. Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
15. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
16. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.
17. Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable state machine replication of state-based crdts without logs. *arXiv preprint arXiv:1905.08733*, 2019.
18. Xiong Zheng and Vijay K Garg. Byzantine lattice agreement in synchronous systems. In *34nd International Symposium on Distributed Computing (DISC 2020)*, 2020.
19. Xiong Zheng, Vijay K. Garg, and John Kaippallimalil. Linearizable Replicated State Machines With Lattice Agreement. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
20. Xiong Zheng, Changyong Hu, and Vijay K Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.