Linearizable Replicated State Machines With Lattice Agreement

Xiong Zheng
Electrical and Computer Engineering Department, University of Texas at Austin, USA
zhengxiongtym@utexas.edu

Vijay K. Garg
Electrical and Computer Engineering Department, University of Texas at Austin, USA
garg@ece.utexas.edu

John Kaippallimalil
Wireless Access Laboratories, Huawei, USA
John.Kaippallimalil@huawei.com

Abstract

This paper studies the lattice agreement problem in asynchronous systems and explores its application to building a linearizable replicated state machine (RSM). First, we propose an algorithm to solve the lattice agreement problem in $O(\log f)$ asynchronous rounds, where f is the number of crash failures that the system can tolerate. This is an exponential improvement over the previous best upper bound of $O(f)$. Second, Faleiro et al have shown in [Faleiro et al. PODC, 2012] that combination of conflict-free data types and lattice agreement protocols can be applied to implement a linearizable RSM. They give a Paxos style lattice agreement protocol, which can be adapted to implement a linearizable RSM and guarantee that a command by a client can be learned in at most $O(n)$ message delays, where n is the number of proposers. Later, Xiong et al in [Xiong et al. DISC, 2018] gave a lattice agreement protocol which improves the $O(n)$ message delay guarantee to $O(f)$. However, neither of the protocols is practical for building a linearizable RSM. Thus, in the second part of the paper, we first give an improved protocol based on the one proposed by Xiong et al. Then, we implement a simple linearizable RSM using our improved protocol and compare our implementation with an open source Java implementation of Paxos. Results show that better performance can be obtained by using lattice agreement based protocols to implement a linearizable RSM compared to traditional consensus based protocols.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Lattice Agreement, Generalized Lattice Agreement, Replicated State Machine, Consensus

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.29

Related Version https://arxiv.org/abs/1810.05871

Funding This work was partially supported by NSF CSR-1563544,CNS-1812349, WNCG Agreement, and Cullen Trust Professorship.

1 Introduction

Lattice agreement, introduced in [2], to solve the atomic snapshot problem [1] in shared memory, is also an important decision problem in message passing systems. In this problem, n processes start with input values from a lattice and need to decide values which are comparable to each other in spite of f process failures, where n is the number of processes and f is the maximum number of failures in the system.

There are two primary applications of lattice agreement. First, Attiya et al [2] give a $\log n$ rounds algorithm to solve the lattice agreement problem in synchronous message systems and use it as a building block to solve the atomic snapshot problem. Second, Faleiro et al [6]

propose the problem of generalized lattice agreement (GLA), which is a generalization of lattice agreement problem for a sequence of inputs, and demonstrate that the combination of conflict-free data types (CRDT) [14, 15] and generalized lattice agreement protocols can be applied to implement a special class of RSM and provide linearizability [8]. We call this special class of state machines as Update-Query (UQ) state machines. The operations of UQ state machines can be classified into two kinds: updates (operations that modify the state) and queries or reads (operations that only return values and do not modify the state). An operation that both modifies the state and returns a value is not supported. In this paper, when we talk about linearizable RSMs, we mean UQ state machines. As shown in [6], to implement a linearizable RSM, we can first design the underlying data structure to be CRDT. This makes all update operations commute. Then, the generalized lattice agreement protocol is invoked for each operation to guarantee linearizability. In this paper, we call a linearizable RSM built by using the combination of CRDT and a GLA protocol as $LaRSM$.

RSM [13] is a popular technique for fault tolerance in a distributed system. Traditional RSMs typically enforce strong consistency among replicas by using a consensus based protocol to order all the requests from the clients. In this approach, each replica executes all the requests in an identical order to ensure that all replicas are at the same state at any given time. The most popular consensus based protocol for building a RSM is Paxos [9, 11]. In Paxos, processes are divided into three different roles: proposer, acceptor and learner. Proposers are responsible for proposing requests from clients to acceptors. Acceptors decide the order of a request and guarantee all learners learn a identical order of requests. When there are multiple proposers in the system, termination is not guaranteed in Paxos. Since the initial proposal of Paxos, many variants have been proposed. FastPaxos [10] reduces the typical three message delays in Paxos to two message delays by allowing clients to directly send commands to acceptors. MultiPaxos [4] is the typical deployment of Paxos in the industrial setting. It assumes that usually there is a stable leader which acts as a proposer, so there is no need for the first phase in the basic Paxos protocol. CheapPaxos [12] extends basic Paxos to reduce the requirement in the number of processors. Even though in the Paxos protocol, there could be multiple proposers, usually only one leader (proposer) is used in practice due to its non-termination problem when there are multiple proposers. The system performance is limited by the resources of the leader. Also, the unbalanced communication pattern limits the utilization of bandwidth available in all of the network links connecting the servers. SPaxos [3] is a Paxos variant which tries to offload the leader by disseminating clients to all replicas. However, the leader is still the only process which can order requests.

Since lattice agreement can be applied to implement a linearizable RSM, if we can solve lattice agreement efficiently, we may not need consensus in some cases. This is promising, since lattice agreement has been shown to be a weaker decision problem than consensus in theory. In synchronous systems, consensus cannot be solved in fewer than $f + 1$ rounds [5], but lattice agreement can be solved in $\log f + 1$ rounds [17]. In asynchronous systems, consensus cannot be solved even with one failure [7], whereas lattice agreement can be solved if a majority of processes is correct [6, 17].

The lattice agreement problem in asynchronous message systems is first studied by Faleiro et al in [6]. They present a Paxos style protocol when a majority of processes are correct. Their algorithm needs $O(n)$ asynchronous round-trips in the worst case. They also propose a protocol for generalized lattice agreement, adapted from their protocol for lattice agreement, which requires $O(n)$ message delays for a value to be learned. Later, a protocol which runs in $O(f)$ asynchronous round-trips was proposed by Xiong et al in [17]. They also give a protocol for generalized lattice agreement which improves the $O(n)$ message delays complexity to $O(f)$. In this work, we improve the upper bound for lattice agreement in asynchronous systems to $O(\log f)$, which is an exponential improvement.

Although [6] has demonstrated that generalized lattice agreement protocol can be applied to implement a linearizable RSM, both the protocols proposed in [6] and [17] are impractical. This is due to the following reason. In both protocols, each process has an accept command which keeps track of all received proposal values. When the protocols are applied to implement a linearizable RSM, this accept command is a set which records all previously proposed commands. When a process rejects a proposal, it has to send back this whole set. Even worse, this set keeps increasing as more commands arrive from clients. In this work, we propose an improved algorithm for the generalized lattice agreement problem, which is specifically designed to make it practical to build a linearizable RSM.

In summary, this paper makes the following contributions:

- We present an algorithm, *AsyncLA*, to solve the lattice agreement in asynchronous system in $O(\log f)$ rounds, where f is the maximum number of crash failures in the system. This bound is an exponential improvement to the previously known best upper bound of $O(f)$ by [17].
- We give an improved algorithm for the generalized lattice agreement protocol based on the one proposed in [17] to make it practical to implement a linearizable RSM.
- We implement a simple linearizable RSM in Java by combining a CRDT map data structure and our improved generalized lattice agreement algorithm. We demonstrate its performance by comparing with SPaxos. Our experiments show that LaRSM achieves around 1.3x times throughput than SPaxos and a lower operation latency in normal case.

2 System Model and Problem Definitions

2.1 System Model

We consider a distributed message passing system with n processes, $p_1, \ldots, p_n$, in a completely connected topology. We only consider asynchronous systems, which means that there is no upper bound on the time for a message to reach its destination. The model assumes that processes may have crash failures but no Byzantine failures. The model parameter f denotes the maximum number of processes that may crash in a run. We do not assume that the underlying communication system is reliable.

2.2 Lattice Agreement

In the lattice agreement problem, given a join semi-lattice $(X, \leq, \sqcup)$ with $\leq$ as the partial order and $\sqcup$ as the join operation, each process p_i proposes a value x_i in X and must decide on some output y_i also in X. An algorithm solves the lattice agreement problem if the following properties are satisfied:

Downward-Validity: For all correct processes $i \in [1..n]$, $x_i \leq y_i$.
Upward-Validity: For all correct processes $i \in [1..n]$, $y_i \leq \sqcup\{x_1, ..., x_n\}$.
Comparability: For any two correct $i \in [1..n]$ and $j \in [1..n]$, either $y_i \leq y_j$ or $y_j \leq y_i$.

2.3 Generalized Lattice Agreement

In the generalized lattice agreement problem [6], each process may receive a possibly infinite sequence of values belong to a lattice at any point of time. Let x_i^p denote the ith value received by process p. The aim is for each process p to learn a sequence of output values y_j^p which satisfies the following conditions:

Validity: Any learned value y_j^p is a join of some subset of received input values.
Stability: The value learned by any correct p is non-decreasing: $j < k \implies y_j^p \leq y_k^p$.

Comparability: Any two values y_j^p and y_k^q learned by any two correct processes p and q are comparable.

Liveness: Every value x_i^p received by a correct process p is eventually included in some learned value y_k^q of every correct process q: i.e, $x_i^p \leq y_k^q$.

3 Asynchronous Lattice Agreement

In this section, we give an algorithm to solve the lattice agreement problem in asynchronous systems which only needs $O(\log f)$ asynchronous rounds. The proposed algorithm is inspired by the algorithm for synchronous setting in [17]. The basic idea is to apply a *Classifier* procedure, which is associated with a specific threshold value, to divide processes into *master* and *slave* groups and ensure that any process in the *master* group have values great than or equal to any process in the *slave* group. Then, by recursively applying a *Classifier* procedure within each subgroup, eventually all processes have comparable values. Equivalently, we can think of the above recursive procedure as letting all processes traverse through a virtual binary *Classifier* tree. Each node of this tree has a *Classifier* procedure with a specific threshold value. When traversing through a node in this tree, the processes will invoke the *Classifier* procedure at this node. Some processes will be classified as *master* and go to the right child, and others will be classified as *slave* and go to the left child. The threshold value associated with the *Classifier* procedure is used to decide which processes should be classified as *master* and which should be classified as *slave*. By carefully setting the threshold value for each *Classifier* procedure in the virtual tree, we can make sure all processes eventually have comparable values.

The main difficulty of the above recursive procedure lies in constructing a *Classifier* procedure to divide a group of processes into two subgroups such that processes in one subgroup have values greater than or equal to processes in the other subgroup. In synchronous systems, [17] gives a very simple procedure. First, each process sends its value to all processes within the same group. Then, it takes join of all values received from processes in the same group. If the join is greater than the threshold value, it is classified as *master* and updates its value to be the join. Otherwise, it is classified as *slave* and keeps its value unchanged. We can easily see that the value of each *slave* process must be received by each *master* process, since the system is synchronous. Thus, each *master* process has value greater than or equal to each *slave* process. In asynchronous systems, however, it is not straightforward to design such a *Classifier* procedure, since we cannot guarantee that the value of a *slave* process is received by each *master* process. Our primary idea for such a *Classifier* procedure in asynchronous systems is as follows. In an asynchronous system, at each round, each process can only wait for $n - f$. If we can guarantee that (1) the value of a *slave* process is stored in at least $n - f$ processes at some point, and (2) each *master* process reads from at least $n - f$ processes, then the value of each *slave* process must be known by each *master* process. This claim holds since any two group of $n - f$ processes have at least one process in common, if we assume $f < \frac{n}{2}$.

The virtual *Classifier* tree is built based on the knowledge of the height of the input lattice, which is unknown. Thus, instead of directly agreeing on the input value lattice, we first agree on a view lattice, which has a known maximum height.

We associate each process p_i with a view v_i, which is an array composed of n entries. Each entry of the view corresponds to the input value of each process known by p_i. Initially, $v_i[i] = x_i$ and $\forall j \neq i, v_i[j] = \perp$, where x_i is the input value of p_i. We say $\perp$ is smaller that any input value. For any two views v and u, we say v *dominates* u, if for all i, $v[i] \geq u[i]$.

Consider the lattice formed by the initial views of all processes with the order defined by the domination relation, i.e, $v \leq u$ iff u dominates v. We call this lattice the view lattice. This view lattice has its smallest element (or bottom) equal to $[\perp, ..., \perp]$ and the top element equal to $[x_1, ..., x_n]$. The height of the view lattice is n (the length of the longest chain). We say v and u are comparable if either $v \leq u$ or $u \leq v$. The join of any two views is defined as the component-wise maximum. The height of a view v, denoted as $h(v)$, is defined as the number of components which are not $\perp$, i.e, the number of processes whose values are contained in this view. Since the ith entry of any view is either the input value of p_i or $\perp$, if a view $v \leq u$, then view u contains all input values contained in view v. That is, in the original input lattice, we have $\sqcup\{v[i] : i \in [1..n]\} \leq \sqcup\{u[i] : i \in [1..n]\}$ if $v \leq u$. Thus, if all correct processes can output comparable views from the view lattice, they can output comparable values from the input value lattice by taking join of all values contained in its output view. Therefore, in our algorithm, instead of directly working on the input value lattice, we apply the *Classifier* technique on the view lattice. The *Classifier* procedure is shown in Fig. 1. The main algorithm, *AsyncLA*, is shown in Fig. 2.

3.1 The Classifier Procedure

The *Classifier* procedure has three input parameters: the input view, the threshold value, and the round number. Each process keeps a label. Whenever a process invokes the *Classifier* procedure, it passes its current view, its label and the current round number as the parameters. We say any two processes p_i and p_j are in the same group at a certain round if they invoke the *Classifier* procedure with the same threshold value, i.e, they have the same label. When processes are classified into different subgroups, they update their labels accordingly (to be explained later). Since processes pass their labels as the threshold value of the *Classifier* procedure, we use label or threshold value interchangeably henceforth. Details of the *Classifier* procedure for p_i at round r are shown as below:

Line 0: p_i set its $acceptVal_r$ to be empty. This $acceptVal_r$ set is used to record all the $< view, label >$ pairs received from all processes at round r via *write* or *read* messages. Note that this $acceptVal_r$ also includes $< view, label >$ pairs received from processes that are not in the same group as p_i.

Line 1-2: p_i sends a *write* message with its current view v and the threshold value (current label) k to all processes and waits for $n - f$ *write_ack*s. This step is to ensure that the value and label of p_i is in the $acceptVal_r$ set of $n - f$ processes.

Line 3-5: p_i sends a *read* message with its current round number r to all processes and waits for $n - f$ *read_ack*s. It collects all received views associated with the same label k in a set U, i.e, collects all views from processes within the same group. It may seem that *lines* 3-5 perform the same functionality as *lines* 1-2 and there is no need to have both. However, this part is actually the key of the *Classifier* procedure. The reason will be clear in the correctness proof section.

Line 6-14: p_i performs classification based on the views received from processes in the same group. Let w be the join of all received views in U. If the height of w is greater than k, then p_i sends a *write* message with w, k and r to all and waits for $n - f$ *write_ack*s with round number r. Then in *line* 10-12, it takes the join of w and all the views contained in the *write_ack*s from the same group, denoted as w'. It returns $(w', master)$ as output of the *Classifier* procedure in which *master* indicates its classified into *master* group in the next round. Otherwise, it returns its own input view v and *slave*.

When p_i receives a *write* message for round r_j from p_j, it includes the $< view, label >$ pair contained in the message into its $acceptVal_{r_j}$ set and sends a *write_ack* message containing the current $acceptVal_{r_j}$ back. When p_i receives a *read* message for round r_j from

p_j, it sends a *read_ack* message containing its current $acceptVal_{r_j}$ back. Basically, the *write* message is used to ensure at least $n - f$ processes know the current view and label of a process and the *read* message is used to retrieve the knowledge of at least $n - f$ processes.

Note that when a process which is invoking the *Classifier* at round r receives a *write* or *read* message with a round number $r' > r$, it buffers this message and delivers it when it reaches round r'.

Classifier(v, k, r)**:**
v: input view
k: threshold value
r: round number

0: $acceptVal_r := \emptyset$ // set of <view, threshold> pairs.

/* write */
1: Send $write(v, k, r)$ to all
2: wait for $n - f$ $write\_ack(-, -, r)$

/* read */
3: Send $read(r)$ to all
4: **wait** for $n - f$ $read\_ack(-, -, r)$
5: Let U be views contained in received acks with label equals k
/* Classification */
6: Let $w := \sqcup\{u : u \in U\}$
7: **if** $h(w) > k$ /* height of w is greater than its label */
8: Send $write(w, k, r)$ to all
9: wait for $n - f$ $write\_ack(-, -, r)$
10: Let U' be views contained in received acks with label equals k /* views received in the same group */
11: Let $w' := w \sqcup \{u : u \in U'\}$
12: **return** (w', *master*)
13: **else**
14: **return** (v, *slave*)

Upon receiving $write(v_j, k_j, r_j)$ **from** p_j
$acceptVal_{r_j} := acceptVal_{r_j} \cup < v_j, k_j >$
Send $write\_ack(acceptVal_{r_j}, r_j)$ **to** p_j

Upon receiving $read(r_j)$ **from** p_j
Send $read\_ack(acceptVal_{r_j}, r_j)$ **to** p_j

Figure 1 *The Classifier Procedure.*

3.2 Algorithm *AsyncLA*

Now let us look at the main algorithm, *AsyncLA*. The basic idea of *AsyncLA* is to let all processes recursively invoke the *Classifier* procedure with a carefully set threshold value. Let y_i denote the output value of p_i. Let v_i^r denote its view at the beginning of round r. The algorithm for p_i proceeds in asynchronous rounds. The algorithm runs in $\log f + 1$ rounds.

At round 0, all processes exchange their views. The purpose of round 0 is to allow us to construct the virtual *Classifier* tree with height equal to $\log f$. In such case, the recursive invocation of the *Classifier* procedure terminate in $\log f$ rounds. The reason is as follows. After round 0, the view of each correct process must have height at least $n - f$ in the view lattice. Since the height of the view lattice is n, the join-closed subset that includes all current views after round 0 (which is also a lattice) has height at most f. Then we can construct a the binary *Classifier* tree with height equals to $\log f$ by setting the threshold value of the root *Classifier* to be $\frac{(n-f)+n}{2} = n - \frac{f}{2}$ in the virtual tree. We say the root *Classifier* is at level 1. For any node at level r of the tree with threshold value k, we set the threshold value of its right child to be $k + \frac{f}{2^{r+1}}$ and the threshold value of its left child to be $k - \frac{f}{2^{r+1}}$. Thus, we can easily see that the height of the tree is $\log f$. Note that the initial label of each process is $n - \frac{f}{2}$, which means all processes are at the root of the tree.

From round 1 to $\log f$, each process simply traverses the virtual *Classifier* tree. At round r, each process is at level r of the tree. For process p_i, it invokes the *Classifier* procedure with its current view v_i^r, current label l_i and r as parameters. Based on the output of the

Classifier procedure, p_i adjust its label to be the threshold value of the *Classifier* procedure it will invoke at next round. If it is classified as master, then it increases its label by $\frac{f}{2^{r+1}}$, i.e, it goes to the right subtree of the virtual *Classifier* tree. Otherwise, it reduces its label by $\frac{f}{2^{r+1}}$, i.e, it goes to the left subtree of the virtual *Classifier* tree. At the end of round $\log f$, p_i outputs the join of all values contained in its current view as its decision value.

AsyncLA(x_i) for p_i:	/* Round 1 to $\log f$ */
x_i: input value	$v_i^1 := \sqcup\{u \mid u \in U\}$
y_i: output value	**for** $r := 1$ to $\log f$
$l_i := n - \frac{f}{2}$ // label of p_i	$(v_i^{r+1}, class) := Classifier(v_i^r, l_i, r)$
v_i^r: the view of p_i at the beginning of round r, an array of size n.	**if** $class = master$
Initially, $v_i^0[i] = x_i \wedge v_i^0[j] = \perp, \forall j \neq i$	$l_i := l_i + \frac{f}{2^{r+1}}$
	else
/* Round 0 */	$l_i := l_i - \frac{f}{2^{r+1}}$
Send *value*$(v_i^0, 0)$ to all	**end for**
wait for $n - f$ messages of form *value*(-, 0)	Let $V_i := v_i^{\log f + 1}$
Let U denote the set of all received values	$y_i := \sqcup\{V_i[j] : j \in [1..n]\}$

Figure 2 Algorithm *AsyncLA*.

3.3 Proof of Correctness

Let us prove the correctness of *AsyncLA*. Let w_i^r be the value of w at *line* 6 of the *Classifier* procedure at round r. Let G be a group of processes at round r. Recall that a group G at round r is a set of processes which have the same label at round r. The label of a group is the label of the processes in this group. Let $M(G)$ and $S(G)$ be the group of processes which are classified as *master* and *slave*, respectively, when they run the *Classifier* procedure in group G. Recall that $h(v)$ denote the height of view v in the view lattice. The following lemma presents the key properties of the *Classifier* procedure, with detailed proof given in the full paper [16].

▶ **Lemma 1.** *Let G be a group at round r with label k. Let L and R be two nonnegative integers such that $L \leq k \leq R$. If $L < h(v_i^r) \leq R$ for every process $i \in G$, and $h(\sqcup\{v_i^r : i \in G\}) \leq R$, then*

(p1) *for each process $i \in M(G)$, $k < h(v_i^{r+1}) \leq R$*
(p2) *for each process $i \in S(G)$, $L < h(v_i^{r+1}) \leq k$*
(p3) *$h(\sqcup\{v_i^{r+1} : i \in M(G)\}) \leq R$*
(p4) *$h(\sqcup\{v_i^{r+1} : i \in S(G)\}) \leq k$, and*
(p5) *for each process $i \in M(G)$, $v_i^{r+1} \geq \sqcup\{v_i^{r+1} : i \in S(G)\}$*

Once we have the above lemma, the correctness proof for *AsyncLA* follows in a similar fashion as the synchronous algorithm in [17]. We only give the primary lemmas and put the detailed proof in the full paper [16].

▶ **Lemma 2.** *Let G be a group of processes at round r with label k. Then*

(1) *for each process $i \in G$, $k - \frac{f}{2^r} < h(v_i^r) \leq k + \frac{f}{2^r}$*
(2) *$h(\sqcup\{v_i^r : i \in G\}) \leq k + \frac{f}{2^r}$*

▶ **Lemma 3.** *Let i and j be two processes that are within the same group G at the end of round $r = \log f$. Then v_i^{r+1} and v_j^{r+1} are equal.*

Proof. (Sketch of Proof) Intuitively, after round 0, the join-closed subset that includes all current views after round 0 (which is also a lattice) has height at most f. We know that the height interval of values in a group are shrinking by a factor of 2, from Lemma 2. Thus, after $\log f$ rounds, any two processes in a same group must have a same value. ◀

▶ **Lemma 4.** *Let process i decides on y_i. Let G be a group at round r such that $i \in S(G)$, then $y_i \leq \sqcup\{v_i^{r+1} : i \in S(G)\}$.*

Proof. Immediate from $(p2)$ and $(p4)$ of Lemma 1. ◀

Since the value of a process is non-decreasing at each round, from $(p5)$ of Lemma 1 and Lemma 4, we have that once two processes are classified into two subgroups, their values must be comparable. We immediately have the following lemma.

▶ **Lemma 5.** *Let i and j be any two processes in two different groups G_i and G_j at the end of round $\log f$, then y_i is comparable with y_j.*

▶ **Theorem 6.** *Algorithm* AsyncLA *solves the lattice agreement problem in $O(\log f)$ round-trips when at least a majority of processes are correct.*

Proof. *Down-Validity* holds since the value held by each process is non-decreasing. *Upward-Validity* follows because each learned value must be the join of a subset of all initial values which is at most $\sqcup\{x_1, ..., x_n\}$. For *Comparability*, from Lemma 3, we know that any two processes which are in the same group at the end of *AsyncLA*, they must have equals values. For any two processes which are in two different groups, from Lemma 5 we know they must have comparable values. ◀

3.4 Complexity Analysis

Each invocation of the *Classifier* procedure takes at most three round-trips. Therefore, $\log f$ invocation of *Classfier* results in at most $3 * \log f$ round-trips. Thus, the total time complexity is $3 * \log f + 1$ round-trips. For message complexity, each process sends out at most 3 *write* and *read* messages and at most $3 * n$ *write_ack* and *read_ack* messages. Therefore, the message complexity for each process is $O(n * \log f)$.

4 Improved Generalized Lattice Agreement Protocol for RSM

In this part, we give optimizations for the generalized lattice agreement protocol proposed in [17] (referred as GLA) to implement a linearizable RSM. Inside GLA, a $f + 1$ round-trips asynchronous lattice agreement protocol proposed in the same paper is embedded. We do not change their $f + 1$ round-trips asynchronous protocol to our $O(\log f)$ protocol for reasons as follows. First, our primary goal in this section is not to give a new protocol for the generalized lattice agreement problem, yet to make it practical to implement. The optimizations we make in this section do not involve the lattice agreement protocol part. Keeping the simple $f + 1$ round-trips algorithm would make things easier. Second, in practice, the $f + 1$ round-trips algorithm is favorable compared to the $O(\log f)$ algorithm due to its smaller constant.

The optimized protocol, GLA_Δ, is shown in Fig. 3 with the two main changes marked using Δ. Note that although we only have two primary changes compared to GLA, we claim those changes are the key for its applicability in building a linearizable RSM. The basic idea of GLA_Δ is to invoke a separate lattice agreement instance for a set of concurrent commands. To ensure *Comparability* and *Stability* of generalized lattice agreement, we associate each

lattice agreement instance with a distinct sequence number and ensure that lattice agreement instance with higher sequence number only starts running after the instance with smaller sequence number has completed. In this way, we guarantee that any value learned by higher sequence lattice agreement instance is greater than or equal to the value learned by lower sequence lattice agreement instance. Along with the *Comparability* property of each lattice agreement instance, we can obtain *Comparability* and *Stability* for the generalized lattice agreement problem. Details are given in correctness proof section.

In GLA_Δ, the sequence number assigned for lattice agreement instances starts from 0 and increase by 1 when a new lattice agreement instance is created. Each process keeps an integer s, which is the next available sequence number. Since different processes might be executing lattice agreement instance with different sequence number, *maxSeq* is used to record the largest sequence number known by a process. *buffVal* stores all the commands received which need to be learned. *LV* denotes the mapping from a sequence number to its corresponding learned command set. *acceptVal* stores all commands received from all processes via proposal messages.

***GLA*$_\Delta$** for p_i:
$s := 0$ // sequence number
$maxSeq$:= -1 // largest sequence number seen
$buffVal := \bot$ // commands buffer
$LV := \bot$ // map from seq to learned commands set
$acceptVal := \bot$ // current accepted commands set
$active := false$ //proposing status

Procedure *ReceiveValue*(v):
  $buffVal := buffVal \sqcup v$

on receiving $prop(v_j, r, s')$ from p_j:
  if $s' < s$
    $buffVal := buffVal \sqcup v_j$ Δ_2
    Send $ACK(\text{“}decide\text{”}, LV[s'], r, s')$
    return
  $maxSeq := \max\{s', maxSeq\}$
  wait until $s' = s$
  if $acceptVal \subseteq v_j$
    Send $ACK(\text{“}accept\text{”}, -, r, s')$
    $acceptVal := v_j$
  else
    Send $ACK(\text{“}reject\text{”}, acceptVal, r, s')$

Procedure *Agree():*
guard: $(active = false) \wedge (buffVal \neq \bot \vee maxSeq \geq s)$
  $active := true$
  $acceptVal := buffVal \sqcup acceptVal$
  $buffVal := \bot$
  /* Lattice Agreement with sequence number s */
  for $r := 1$ **to** $f + 1$
    $val := acceptVal$
    Send $prop(val, r, s)$ to all
    wait for $n - f$ $ACK(-, -, r, s)$
    Let V be values in *reject ACKs*
    Let D be values in *decide ACKs*
    Let *tal* be the number of *accept ACKs*
    if $|D| > 0$
      $val := \sqcup\{d \mid d \in D\}$; **break**
    else if $tal > \frac{n}{2}$
      break
    else
      $acceptVal := acceptVal \sqcup \{v \mid v \in V\}$
    end
  $LV[s] := val$
  $acceptVal := acceptVal$ - $LV[s-1]$ Δ_1
  $s := s + 1$
  $active := false$

Figure 3 Algorithm GLA_Δ.

When a process receives a command v from a client, it invokes *ReceiveValue(v)* to include v into its buffer (*buffVal*).

Each process invokes the *Agree()* procedure to start a new lattice agreement instance. This procedure is automatically executed when the guard condition is satisfied. Inside the *Agree()* procedure, a process first updates its *acceptVal* to be the join of current *acceptVal* and *buffVal*. Then, it starts a lattice agreement instance with the next available sequence number. The lattice agreement instance runs the $f + 1$ round-trips protocol in [17]. At each round of the lattice agreement, a process sends its current *acceptVal* to all and waits for $n - f$ *ACK*s. If it receives any *decide ACK*, it decides on the join of all *decide* values. If it receives a majority of *accept ACK*s, it decides on its current value. Otherwise, it updates its *acceptVal* to be the join of all received values and starts next round. When a process receives a proposal from some other process, if the proposal is associated with a smaller sequence number, then it sends *decide ACK*s back with its decided value for that sequence number and includes the received value into its own buffer set. Otherwise, it waits until its current sequence number to reach the sequence number associated with the proposal. Then, it checks whether the proposed value contains its current *acceptVal*. If true, the process sends back a *accept ACK*. Otherwise, it sends back a *reject ACK* along with its current *acceptVal*. When a process completes lattice agreement for sequence number s, it stores learned values in *LV[s]* and removes all learned values for sequence number $s - 1$.

Now we explain the our proposed improvements in detail.

4.1 Truncate the Accept Command Set

Let us first look at the challenges of directly applying the GLA protocol in [17] or the one in [6] to implement a linearizable RSM. In a RSM, each input value is a command from a client. Thus, the input lattice is a finite boolean lattice formed by the set of all possible commands. The order in this lattice is defined by the set inclusion, and the join is defined as the union of two sets. This boolean input lattice poses a challenge for both the algorithms in [6] and [17]. In these algorithms, for each process there is an accept command set (*acceptVal*), which stores the join of whatever value the process has accepted. Now since the join is defined as union in the RSM setting, this set keeps increasing. For example, in the original algorithm given in [17], it does not include the line marked as Δ_1. Suppose, we have three processes p_1, p_2 and p_3 which handles commands from clients. Suppose p_1, p_2 and p_3 first receive commands $\{a\}$, $\{b\}$ and $\{c\}$, respectively. They start the lattice agreement instance with the sequence number 0 and learn $\{a\}$, $\{a, b\}$ and $\{a, b, c\}$ respectively for the sequence number 0. After that, p_1, p_2 and p_3 receive $\{d\}$, $\{e\}$, and $\{f\}$ as input, respectively. Now, they start a lattice agreement instance with the sequence number 1. In order to ensure *Comparability* and *Stability* of GLA, the accept command set for sequence number 1 have to include the largest learned value of sequence 0, which is $\{a, b, c\}$, although each process only proposes a single command. Therefore, the accept command set keeps increasing. This problem makes applying lattice agreement to implement a linearizable RSM impractical.

To tackle the above problem, we need to truncate the accept command set. A naive way is to remove all learned commands in the accept command set when proposing for the next available sequence number. This way does not work. Suppose we have two processes: p_1, p_2 and p_3. They propose $\{a\}$, $\{b\}$ and $\{c\}$, respectively for sequence number 0. After execution of lattice agreement for sequence number 0, suppose p_1, p_2 and p_3 both have learned value set and accept command set to be $\{a\}$, $\{a, b, c\}$, and $\{a, b, c\}$, respectively. It is easy to verify this case is possible for an execution of lattice agreement. When completing sequence number 0, all processes remove learned value set for sequence number 0 from their accept command set. Thus, the accept command set of all the three processes becomes to be empty. Now, suppose p_1, p_2 and p_3 start to propose for sequence number 1 with new

commands $\{d\}$, $\{e\}$ and $\{f\}$. Since the accept command sets of p_2 and p_3 do not contain value $\{b\}$ and $\{c\}$, p_1 will never be able to learn $\{b\}$ and $\{c\}$. Thus, learned command set of p_1 for sequence 1 and the learned command set of p_2 and p_3 for sequence 0 are incomparable. Instead of removing all learned commands from the accept command set, we propose to remove all learned commands for the sequence numbers smaller than the largest learned sequence number from the accepted command set. In order to achieve this, the line marked by Δ_1 in the pseudocode is added, compared to the original algorithm in [17]. In this line, after a process has learned a value set for sequence number s, it removes the learned value set corresponding to sequence number $s-1$ from its accept command set.

4.2 Remove Forwarding

In both the algorithms of [6] and [17], a process has to forward all commands it receives to all other processes to ensure liveness. This forwarding results in load that is multiplied many fold, since many processes may propose the same request. In [17], this forwarding is to ensure that the commands proposed by slow processes can also be learned. However, for the fast processes, there is no need to forward their requests to others because they can learn requests quickly. Therefore, instead of forwarding every request to all servers, we require that when a process receives some proposal with a smaller sequence number than its current sequence number, it sends back a *decide* message and also includes the received proposal value into its buffer set. These values will be proposed by the server in its next sequence number. In this way, only when a process is slow, its value will be proposed by the fast processes. This change is shown as addition of the line marked by Δ_2.

4.3 Proof of Correctness

Let us prove the correctness of GLA_Δ. Although we only have two primary changes compared to the algorithm in [17], the correctness proof is quite different due to the modification marked by Δ_1. Let $LV_p[s]$ denote the learned value of process p for sequence number s. Let $LearnedVal_s^p$ denote all the learned values of process p after completing lattice agreement for sequence number s. Thus, $LearnedVal_s^p = \sqcup\{LV_p[t] : t \in [0...s]\}$. Due to the page limit, we put the proof for Lemma 7-9 in the full paper [16].

▶ **Lemma 7.** *For any sequence number s, $LV_p[s]$ is comparable with $LV_q[s]$ for any two processes p and q.*

▶ **Lemma 8.** *For any sequence number s, $LearnedVal_p^s \subseteq LearnedVal_q^{s+1}$ for any two correct processes p and q.*

▶ **Lemma 9.** *For any sequence number s and s', $LearnedVal_p^s$ and $LearnedVal_q^{s'}$ are comparable for any two correct processes p and q.*

▶ **Theorem 10.** *Algorithm GLA_Δ solves the generalized lattice agreement problem when a majority of processes is correct.*

Proof. *Validity* holds since any learned value is the join of a subset of values received. *Stability* follows from Lemma 8. *Comparability* follows from Lemma 9. *Liveness* follows from the termination of lattice agreement. ◀

5 LaRSM vs Paxos

In this section, we compare LaRSM and Paxos from both theoretical and engineering perspective. Table 1 shows the theoretical perspective. The main difference between Paxos and LaRSM lies in their termination guarantee. In the worst case, Paxos may not terminate (∞ message delays), though very unlikely. Whereas, LaRSM always guarantee termination in at most $O(\log f)$ message delays. This difference is because Paxos is consensus based and LaRSM is lattice agreement based. In the best case, both Paxos and LaRSM need three message delays. One limitation of LaRSM is that it is only applicable to UQ state machines.

For engineering perspective, Paxos is typically deployed with only one single proposer (leader) due to its non-termination. Only the leader can handle handle requests from clients. Thus, in a typical deployment the leader becomes the bottleneck, i.e, the throughput of the system is limited by the leader's resources. Besides, the unbalanced communication pattern limits the utilization of bandwidth available in all of the network links connecting the servers. In LaRSM, however, there could be multiple proposers since termination is guaranteed, which can simultaneously handle requests from clients and may yield better throughput. In the failure case, a new leader needs to be elected in Paxos and there could be multiple leaders in the system. During this time, Paxos generally takes longer to terminate because of conflicting proposals. However, a failure of a replica in LaRSM has limited impact on the whole system. This is because other replicas can still handle requests from clients as long as less than a majority of replicas has failed.

Table 1 Paxos vs LaRSM.

Properties	**Paxos**	**LaRSM**
Consistency	Linearizability	Linearizability
Underlying Protocol	Consensus	Lattice Agreement
Best Case #Message Delays	3	3
Worse Case #Message Delays	∞	$O(\log f)$
Applicable to All Sate Machines	Yes	Only UQ State Machines

6 Evaluation

In this section, we evaluate the performance of LaRSM and compare with SPaxos. To implement LaRSM, we also propose some practical optimizations for the procedure proposed in [6] to implement a RSM by combining CRDT and a generalized lattice agreement protocol, which can be found in the full paper [16].

Although the lattice agreement protocol proposed in this paper has round complexity of $O(\log f)$, it has a large constant, which is only advantageous when the number of replicas is large. In practical cases, the number of replicas is usually small, often 3 to 5 nodes. Thus, we adopt the lattice agreement protocol from [17] which runs in $f + 1$ asynchronous round-trips in our implementation. In order to evaluate LaRSM, we implemented a simple RSM which stores a Java hash map data structure. We implement the hash map date structure to be a CRDT by assigning a timestamp to each update operation and maintain the last writer wins semantics. We measure the performance of SPaxos and our implementation in the following three perspectives: performance in the normal case (no crash failure), performance in failure case, and performance under different work loads.

All experiments are performed on Amazon's EC2 micro instances, which have variable ECUs (EC2 Compute Unit), 1 vCPUs, 1 GBytes memory, and low to moderate network performance. All servers run Ubuntu Server 16.04 LTS (HVM) and the socket buffer sizes are equal to 16 MBytes. All experiments are performed in a LAN environment with all processes distributed among the following three availability zones: US-West-2a, US-West-2b and US-West-2c.

The keys and values of the map are string type. We limit the range of keys to be within 0 to 1000. Two operations are supported: update and get. The update operation changes the value of a specific key. The get operation returns the value for a specific key. A client execute one request per time and starts executing next request when it completes the current one. The request size is 20 bytes. For each request, the server returns a response to indicates its completion. In order to compare with SPaxos, we set its crash model to be CrashStop. In this model, SPaxos would not write records into stable storage. In SPaxos, batching and pipelining are implemented to increase the performance of Paxos. There are some parameters related to those two modules: the batch size, batch waiting timeout and the window size. The batch size controls how many requests the batcher needs to wait before starting proposing for a batch. The batch waiting timeout controls the maximum time the batch can wait for a batch. The window size is the maximum number of parallel proposals ongoing. We set the batch size to be 64KB, which is the largest message size in a typical system. We set the batch timeout according to the number of clients from 0 to 10 at most. The window size is set to 2 because we found that increasing the window size further does not improve the performance in our evaluation.

6.1 Performance in Normal Case

In this experiment, we build a RSM system with three instances. We measure the throughput of the system and latency of operations while increasing the number of clients. The load from the clients are composed of 50% writes and 50% reads. The left part of Fig. 4 shows the throughput of SPaxos and LaRSM. The throughput is measured by the number of requests handled per second by the system. The latency is the average time in milliseconds taken by the clients to complete execution of a request. We can see from Fig. 4, as we increase the number of clients, the throughput of both SPaxos and LaRSM increases until there are around 1000 clients. At that point, the system reaches its maximum handling capability. If we further increase the clients number, the throughput of both LaRSM and SPaxos does not change in a certain range and begins to decrease. This is because both systems do not limit the number of connections from the client side. A large number of clients connection results in large burden on IO, decreasing the system performance. Comparing SPaxos and LaRSM, we can see that LaRSM always has better throughput than SPaxos.

The right part of Fig. 4 shows the latency of LaRSM and SPaxos. In both LaRSM and SPaxos, read and write perform the same procedure, thus their latency should be similar. So, in our evaluation, we just use operation latency. From Fig. 4, we find that operation latency of LaRSM keeps increasing. As we increase the number of clients, the latency of SPaxos decreases first up to some point and then begins to increase. This performance is because the latency is the average response time of all clients and SPaxos has a batching module which batches multiple requests from different clients to propose in a single proposal. Thus, initially when there are very few clients, they can only propose a small number of requests in a single proposal, which makes the latency relatively higher. While the number of clients increases, more requests can be proposed in one single batch, thus the average latency for one client decreases. If the number of clients increases further, the handling capability limit

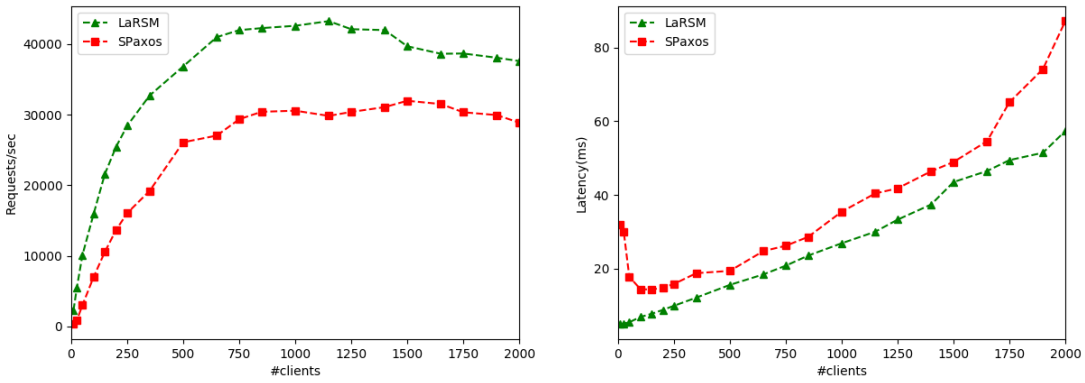

Figure 4 Throughput and latency of LaRSM and SPaxos with increasing number of clients.

of the system increases the operation latency. Comparing SPaxos and LaRSM, we find that the latency of LaRSM is always around 5ms smaller.

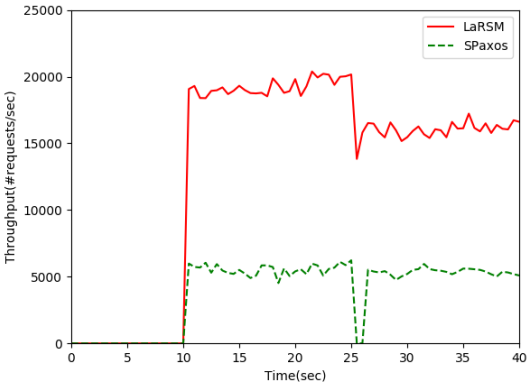

Figure 5 Throughput in Case of Failure.

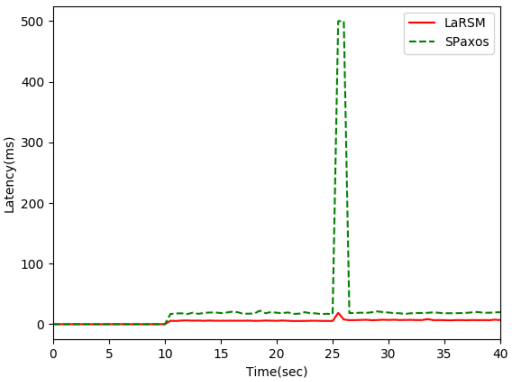

Figure 6 Latency in Case of Failure.

6.2 Performance in Failure Case

In this section, we evaluate the performance of both LaRSM and SPaxos in the case of failure. In this experiment, the RSM system is composed of five replicas. There are 100 clients that keep issuing requests to the system. In LaRSM, since all replicas perform the same role and can handle requests from the clients concurrently. Thus, for loading balancing, each client randomly selects a replica to connect. Each client has a timeout, unlike SPaxos, this timeout is typically small. Timeout on an operation does not necessarily mean failure of the connected replica. It might also be due to an overload of the replica. In this case, the client randomly chooses another replica to connect. However, in SPaxos, the timeout set for a client is usually used to suspect the leader. That is, when an operation times out, most likely the leader has failed. Thus, the timeout in SPaxos is typically large.

We run the simulation for 40 seconds. The first 10 seconds is for the system to warm up, so we do not record the throughput and latency data. A crash failure is triggered at 25th second after the start of the system. For LaRSM, we randomly shut down one replica since

all replicas are performing the same role. For SPaxos, we shut down the leader, since crash of a follower does not have much impact on the system. Figure 5 shows the throughput of both LaRSM and SPaxos. Figure 6 shows the latency change. From Fig. 5 and Fig. 6, for LaRSM we can see that when the failure occurs, the throughput drops sharply from around 20K requests/sec to around 15K requests/sec, but not to 0. However, the throughput of SPaxos drops to 0 when leader fails. The latency of LaRSM only increases slightly, whereas the latency of SPaxos goes to infinity (Note that in the figure it is shown as around 500ms). This is because when leader fails, SPaxos stops ordering requests, thus no requests are handled by the system. For LaRSM, the clients which are connected to the failed replica, timeout on their current requests and then randomly connect to another replica. As discussed before, this timeout is usually much smaller than the timeout for suspecting a failure in SPaxos. Thus, the latency of a client in LaRSM only increases by a small amount. After the failure, the throughput of LaRSM remains around 16K requests/sec, which is because now there is one less replica in the system and the handling capability of the system decreases. For SPaxos, after a new leader is selected, the throughput increases to be a level slightly smaller than the throughput before the failure and the latency also decreases to be slightly higher than the latency before the failure. We also find that even though the throughput of LaRSM drops when a failure occurs, it still has better throughput than SPaxos, which indicates the better performance of LaRSM.

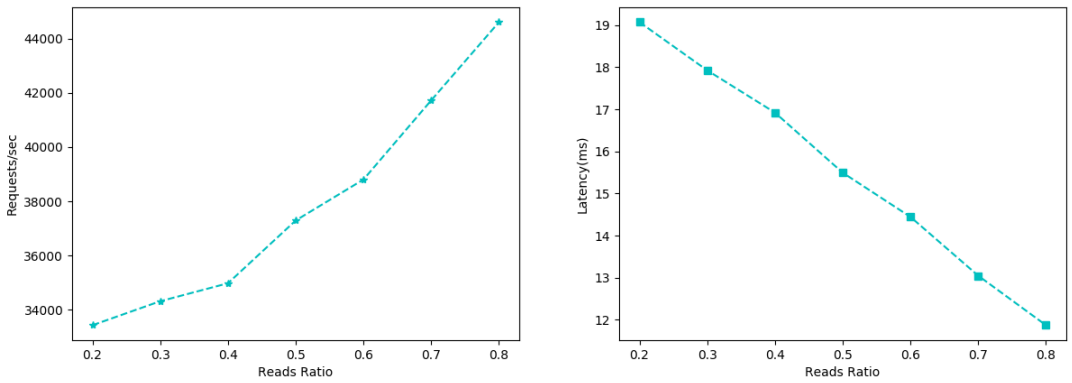

Figure 7 Throughput and Latency Under Different Reads Ratio.

6.3 Performance under Different Loads

In this part, we evaluate the performance of LaRSM on different types of work loads. This evaluation is done in a system of three replicas with 500 clients that keep issuing requests. We measure the throughput and latency as we increase the ratio of reads in a work load. The left part and right part of Fig. 7 give the throughput and latency change, respectively. It is shown in Fig. 7 that as the ratio of reads increases in a work load, the throughput of the system increases and the operation latency decreases. This confirms our optimization for the procedure to implement a linearizable RSM. As the reads ratio increases, the writes ratio decreases. Note that in a lattice agreement instance the input lattice is formed only by all the writes. When the number of writes is small, the proposal command set would be small and the message size would be small as well. Thus, the system can complete a lattice agreement instance faster. This shows that the performance LaRSM is even better for settings with fewer writes.

7 Conclusion

In this paper, we first give an algorithm to solve the lattice agreement problem in $O(\log f)$ rounds asynchronous rounds, which is an exponential improvement compared to previous $O(f)$ upper bound. We also give some optimizations for the GLA protocols proposed in literature. Evaluation results show that using lattice agreement to build a linearizable RSM has better performance than conventional consensus based RSM technique. Specifically, our implementation yields around 1.3x times throughput than SPaxos and incurs smaller latency in normal case.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

2 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

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 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

5 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

6 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.

7 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.

8 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

9 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

10 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

11 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

12 Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*, pages 307–314. IEEE, 2004.

13 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

14 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.

15 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. 2011.

16 Xiong Zheng, Vijay K Garg, and John Kaippallimalil. Linearizable replicated state machines with lattice agreement. *arXiv preprint arXiv:1810.05871*, 2018.

17 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.