Generalized Lattice Agreement

[Extended Abstract]

Jose M. Faleiro
Microsoft Research India
t-josfal@microsoft.com

Sriram Rajamani
Microsoft Research India
sriram@microsoft.com

Kaushik Rajan
Microsoft Research India
krajan@microsoft.com

G. Ramalingam
Microsoft Research India
grama@microsoft.com

Kapil Vaswani
Microsoft Research India
kapilv@microsoft.com

ABSTRACT

Lattice agreement is a key decision problem in distributed systems. In this problem, processes start with input values from a lattice, and must learn (non-trivial) values that form a chain. Unlike consensus, which is impossible in the presence of even a single process failure, lattice agreement has been shown to be decidable in the presence of failures. In this paper, we consider lattice agreement problems in asynchronous, message passing systems. We present an algorithm for the lattice agreement problem that guarantees liveness as long as a majority of the processes are non-faulty. The algorithm has a time complexity of $O(N)$ message delays, where N is the number of processes. We then introduce the generalized lattice agreement problem, where each process receives a (potentially unbounded) sequence of values from an infinite lattice and must learn a sequence of increasing values such that the union of all learnt sequences is a chain and every proposed value is eventually learnt. We present a wait-free algorithm for solving generalized lattice agreement. The algorithm guarantees that every value received by a correct process is learnt in $O(N)$ message delays. We show that this algorithm can be used to implement a class of replicated state machines where (a) commands can be classified as *reads* and *updates*, and (b) all update commands *commute*. This algorithm can be used to realize serializable and linearizable replicated versions of commonly used data types.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication NetworksDistributed Systems; D.4.5 [**Software**]: Operating SystemsReliabilityFault Tolerance; E.1 [**Data**]: Data StructuresDistributed Data Structures

Keywords

replication, fault tolerance, lattice agreement

PODC'12, July 16–18, 2012, Madeira, Portugal.

1. INTRODUCTION

Lattice agreement is a key decision problem in distributed systems. In this problem, each process starts with an input value belonging to a lattice, and must learn an output value belonging to the lattice. The goal is to ensure that each process learns a value that is greater than or equal to its input value, each learnt value is the join of some set of input values, and all learnt values form a chain. Unlike consensus, which is impossible in the presence of even a single failure [5], lattice agreement is decidable, and asynchronous, wait-free algorithms for shared memory distributed systems have been proposed [4, 3, 7] (see Section 7).

In this paper, we consider the lattice agreement problem in asynchronous, message passing systems. We present a wait-free algorithm for solving a single instance of lattice agreement. We also study a generalization of lattice agreement, where each process *receives* a (potentially unbounded) sequence of values from an infinite lattice, and the goal is to learn a sequence of values that form a chain. As before, we wish to ensure that every learnt value is the join of some set of received values and that every received value is eventually included in a learnt value. We present a wait-free algorithm for solving generalized lattice agreement.

One of the challenges in extending the lattice agreement algorithm to generalized lattice agreement is ensuring liveness and progress because of the potential to iterate over an unbounded chain without learning new values. Our algorithm guarantees that every received value is learnt in $O(N)$ message delays, even in the presence of failures and concurrent proposals. Our algorithm is *adaptive* [3]: its complexity depends only on the number of participants actively contending at any given point in time. If only k processes receive values, then every received value is learnt in $O(k)$ message delays. In the extreme case, if only one process receives values, our algorithm guarantees that every value is learnt in 3 message delays (including the round trip from the client to the system). Our algorithms guarantee safety properties in the presence of arbitrary non-Byzantine failures (including process failures as well as message losses). The algorithms guarantee liveness and progress as long as a majority of processes are correct and messages between correct processes are eventually delivered.

Finally, we show that generalized lattice agreement can be used to implement a special class of replicated state machines, the main motivation for our work. State machine replication [8] is used to realize data-structures that tolerate process failures by replicating state across multiple processes. The key challenge here is to execute data-structure operations on all replicas so that the replicas converge and consistency (in the form of serializability or linearizability [6]) can be guaranteed. One common approach to state machine replication is to use distributed consensus [9] among the replicas to determine the order in which operations are to be performed. Unfortunately, the impossibility of consensus means that this approach cannot guarantee liveness in the presence of failures.

Operations of a state machine can be classified as *updates* (operations that modify the state) and *reads* (operations that do not modify the state). In this paper, we consider the problem of replicating state machines when all update operations commute with each other. Recent work [12] has shown how several interesting data-structures with update operations that logically do not commute, such as sets (with adds and deletes), sequences, certain kinds of key-value tables, and graphs can be realized using a more complex data-structure with commutative update operations. Thus, commutativity of updates is a reasonable assumption in several settings.

We show that generalized lattice agreement can be used to obtain an algorithm for this class of state machines that guarantees consistency and liveness in the presence of process failures (as long as a majority of the processes are non-faulty). We are not aware of any previous algorithm that guarantees both consistency and liveness for this problem. Algorithms have been previously proposed to exploit commutativity, including generalized consensus [10]. Generalized consensus is similar to generalized lattice agreement, with some significant differences. In generalized consensus, processes propose values belonging to a partially ordered set where not all pairs of elements have a least upper bound (or join). Processes then learn a sequence of values. Generalized consensus requires values learnt by different processes to have a least upper bound, but does not require these values to be comparable (i.e., form a chain). These differences between generalized consensus and generalized lattice agreement turn out to be crucial in achieving consistency with liveness.

A straightforward approach to state machine replication using generalized consensus exploits commutativity as follows: different processes do not have to agree on the order in which commuting operations are performed. However, processes must still agree on the order of non-commuting operations. Even when all updates commute, read operations do not commute with update operations. The need to support non-commuting reads necessitates consensus in this approach and, hence, this approach cannot guarantee liveness.

The essence of our approach is to *separate* updates from reads. Note that the state of any replica corresponds to a *set-of-updates* (that have been executed to produce that state). We utilize generalized lattice agreement, on the power set lattice of the set-of-all-updates, to learn a chain of sets-of-updates. Each learnt set-of-updates represents a *consistent* state that can be used for reads. Specifically, every replica utilizes the state corresponding to its most recent learnt value to process read operations. More details on how we do this appears in the paper.

Note that generalized consensus cannot be used to separate updates from reads in this fashion. Suppose we apply generalized consensus for the state machine consisting of only update operations. Since all operations commute, generalized consensus guarantees liveness in this case. Unfortunately, the learnt values are not guaranteed to form a chain (with generalized consensus). Hence, different learnt values cannot be used to process read operations without compromising consistency.

As a simple example, consider a set data type that supports adds and reads. Assume clients issue two update commands, $\mathsf{Add}(1)$ and $\mathsf{Add}(2)$, and several reads, concurrently on an initially empty set. The adds commute with each other but the reads do not. The key to consistency is to ensure that two concurrent read operations do not return values $\{1\}$ and $\{2\}$. A conventional replicated state machine based on consensus must explicitly order reads with respect to updates, which ensures consistency but compromises liveness. In our implementation, replicas utilize our wait-free algorithm for generalized lattice agreement on the power set lattice $\mathcal{P}(\{\mathsf{Add}(1), \mathsf{Add}(2)\})$. Since lattice agreement guarantees that learnt values form a chain, it follows that no two reads can observe states $\{1\}$ and $\{2\}$. Furthermore, our implementation ensures that if a read returns some value of the set (say $\{1\}$), then any *subsequent* read observes a state produced by applying a larger set of commands (namely $\{1\}$, or $\{1, 2\}$).

To summarize, this paper makes the following three contributions:

- We provide a new algorithm for solving a single instance of the lattice agreement problem for asynchronous message passing system with $O(N)$ message delays. This is the first algorithm for this problem in an asynchronous message passing setting.
- We introduce the generalized lattice agreement problem and present an algorithm for this problem with an upper bound of $O(N)$ message delays for a value to be learnt.
- We use our algorithm for generalized lattice agreement to obtain an algorithm for replicated state machines with commuting updates. This is the first algorithm for this problem that guarantees both consistency and liveness.

2. SYSTEM MODEL

We are interested in lattice agreement problems in an asynchronous message passing system. We consider a system of N independent and asynchronous processes each of which have a unique identity. We assume processes are arranged as nodes in a complete graph and can communicate with each other by sending messages along edges. We make no assumptions about the order in which messages are delivered or about how long it takes for a message to be delivered. We consider non-Byzantine failures, both crash-failures of processes as well as message losses. We will refer to a process that does not fail (in a given execution) as a *correct* process.

3. LATTICE AGREEMENT

Problem Definition. Let $(\mathsf{L}, \sqsubseteq, \sqcup)$ be a semi-lattice with a partial order $\sqsubseteq$ and join (least-upper-bound) $\sqcup$. We say that two values u and v in L are *comparable* iff $u \sqsubseteq v$ or $v \sqsubseteq u$. In lattice agreement, each of the N processes starts with an initial value from the lattice and must learn values that satisfies the following conditions:

1. **Validity**. Any value learnt by a process is a join of some set of initial values that includes its own initial value.

2. **Stability**. A process can learn at most one value.

3. **Consistency**. Values learnt by any two processes are comparable.

4. **Liveness**. Every correct process eventually learns a value.

Most agreement problems are hard to solve in an asynchronous message passing system if a majority of the processes fail or if an arbitrary number of messages are lost [1]. For lattice agreement, we require that the safety properties hold even in the presence of arbitrary non-Byzantine failures (crash failures and message losses). Liveness must hold so long as a majority of processes are correct and messages between correct processes are eventually delivered.

Note that there is a subtle difference between the lattice agreement problem as stated here and the one proposed by Attiya *et al.* in [4]. Attiya *et al.* only require that any learnt value be less than or equal to the join of all initial values. The validity condition here is stricter, it restricts each learnt value to necessarily be a join of some initial values. This difference is not very significant. Any solution to the problem defined here is also a valid solution to the problem defined by Attiya *et al.*. All known algorithms for lattice agreement (for shared memory systems) as defined by Attiya *et al.* satisfy the stricter validity condition that we use here.

Algorithm. Our algorithm for lattice agreement is shown in Algorithm 1. For convenience, we consider two kinds of processes, (1) proposer processes, each of which has an initial value *initialValue*, and learns a single output value *outputValue*, and (2) acceptor processes, which help proposer processes learn output values. The algorithm permits the same process to play the roles of both a proposer and an acceptor. Let N_p denote the number of proposer processes and let N_a denote the number of acceptor processes. In this formulation, our liveness guarantees require a majority of acceptor processes to be correct.

In our formal description of the algorithm, we present every process as a collection of *guarded actions* (loosely in the style of IO-automata). A guarded action consists of a guard (precondition) predicate and an effect. We also give a name to every action to facilitate our subsequent discussion. An action is said to be *enabled* (for a process) if its corresponding precondition is true at the process. A precondition of the form $?M(x) \&\& P(x)$ is said to be true iff the process has an input message $M(x)$ satisfying the predicate $P(x)$. Here M is a message name (tag) and x is the parameter value carried by the message. A process repeatedly selects any enabled action and executes its effect *atomically.* Whenever a process executes an action whose guard includes a message predicate $?M(x)$, the matching message $M(x)$ is *consumed* and removed from the input channel.

Every proposer begins by proposing its value to all acceptors (see Propose in Algorithm 1). Each proposal is associated with a proposal number that is unique to each proposer. Proposal numbers are not required to be totally ordered. The only purpose they serve is to uniquely identify proposals made by a proposer.

An acceptor may *accept* or *reject* a proposed value. Every acceptor ensures that all values that it accepts form a chain in the lattice. It does this by tracking the largest value it has accepted so far in the variable *acceptedValue*. When it gets a proposal, it accepts the proposed value iff it is greater than or equal to *acceptedValue*, and send an acknowledgment back to the proposer. If it rejects a proposed value, it sends back the *join* of the proposed value and *acceptedValue* back to the proposer along with the rejection.

This guarantees that all values accepted by a single acceptor are comparable. However, different acceptors may accept incomparable values. A value is said to be a *chosen* value iff it is accepted by a majority of acceptors. Note that any two chosen values have at least one common acceptor. Hence, any two chosen values are guaranteed to be comparable. Proposers simply count the number of acceptances every proposed value gets and whenever a proposed value gets a majority of acceptances, it knows that its proposed value is a chosen value. The proposer then executes the Decide() action, and declares the current proposed value as its output value.

This approach ensures the safety requirements of the problem. We now show how proposers deal with rejections to ensure liveness and termination. A proposer waits for a quorum of acceptors to respond to its proposal. If all these responses are acceptances, the proposer is done (since its value has been included in an output value). Otherwise, it *refines* its proposal by replacing its current value with the join of its current value and all the values it received with its rejection responses. It then goes through this process all over again, using its current value as the new proposed value. Once a proposer proposes a new value, it ignores responses it may receive for all previous proposals. This approach ensures termination since all values generated in the algorithm belong to a finite sub-lattice of L (namely, the lattice L' consisting of all values that can be expressed as the join of some set of input values).

Time complexity We now establish the complexity of the lattice agreement algorithm. In particular, we measure the time complexity [1] defined as the time it takes for a correct process to learn an output value, under the assumption that every message sent to a correct process is delivered in one unit of time. We present a simple informal argument here to show that the time complexity of the lattice agreement algorithm is $O(N)$ and defer the formal treatment to section 5. From the algorithm it can be seen that every time a proposer performs the action Refine(), it proposes a value that is strictly greater than the previously proposed value. As every proposed value is a join of some initial values, in the worst case, the N^{th} proposal will be the join of all initial values. Such a proposal has to be accepted. As the time between successive proposals is two units and at most N proposals are made the time complexity of the algorithm is $O(N)$. Note that if only k processes propose values, then the complexity is $O(k)$.

Algorithm 1 Lattice Agreement

```
// Proposer process
int $UID$ // Unique id for a process
enum $\{passive, active\}$ $status = passive$
int $ackCount, nackCount, activeProposalNumber = 0$
L $initialValue$ // Initial value of the process
L $proposedValue, outputValue = \perp$

action Propose()
guard: $activeProposalNumber = 0$
effect:
    $proposedValue = initialValue$
    $status = active$
    $activeProposalNumber{+}{+}$
    $ackCount = nackCount = 0$
    Send Proposal($proposedValue, activeProposalNumber, UID$) to all $Acceptors$

action ProcessACK($proposalNumber, value, id$)
guard: ?ACK($proposalNumber, value, id$) && $proposalNumber = activeProposalNumber$
effect: $ackCount{+}{+}$

action ProcessNACK($proposalNumber, value$)
guard: ?NACK($proposalNumber, value$) && $proposalNumber = activeProposalNumber$
effect:
    $proposedValue = proposedValue \sqcup value$
    $nackCount{+}{+}$

action Refine()
guard: $nackCount > 0$ && $nackCount + ackCount \geq \lceil (N_a + 1)/2 \rceil$ && $status = active$
effect:
    $activeProposalNumber{+}{+}$
    $ackCount = nackCount = 0$
    Send Proposal($proposedValue, activeProposalNumber, UID$) to all $Acceptors$

action Decide()
guard: $ackCount \geq \lceil (N_a + 1)/2 \rceil$ && $status = active$
effect:
    $outputValue = proposedValue$
    $status = passive$

// Acceptor process
L $acceptedValue = \perp$

action Accept($proposalNumber, proposedValue, proposerId$)
guard: ?Proposal($proposalNumber, proposedValue, proposerId$) && $acceptedValue \sqsubseteq proposedValue$
effect:
    $acceptedValue := proposedValue$
    Send ACK($proposalNumber, proposedValue, proposerId$) to $proposerId$

action Reject($proposalNumber, proposedValue, proposerId$)
guard: ?Proposal($proposalNumber, proposedValue, proposerId$) && $acceptedValue \not\sqsubseteq proposedValue$
effect:
    $acceptedValue := acceptedValue \sqcup proposedValue$
    Send NACK($proposalNumber, acceptedValue$) to $proposerId$
```

4. GENERALIZED LATTICE AGREEMENT

We now generalize the lattice agreement problem, allowing processes to accept a possibly infinite sequence of input values.

Problem Definition. Let L be a join semi-lattice with a partial order $\sqsubseteq$. Consider a system with N processes. Each process p may receive an input value belonging to the lattice (from a client) at any point in time. There is no bound on the number of input values a process may receive. Let v_i^p denote the i-th input value received by a process p. The objective is for each process p to learn a sequence of output values w_j^p that satisfy the following conditions:

1. **Validity**. Any learnt value w_j^p is a join of some set of received input values.
2. **Stability**. The value learnt by any process p increases monotonically: $j < k \Rightarrow w_j^p \sqsubseteq w_k^p$.
3. **Consistency**. Any two values w_j^p and w_k^q learnt by any two processes are comparable.
4. **Liveness**. Every value v_i^p received by a correct process p is eventually included in some learnt value w_k^q of every correct process q: i.e., $v_i^p \sqsubseteq w_k^q$.

As in lattice agreement, we require that the safety properties hold even in the presence of arbitrary non-Byzantine failures (crash failures and message losses), and liveness must hold as long as a majority of processes are correct and messages between correct processes are eventually delivered.

Algorithm. The algorithm for solving one instance of lattice agreement can be extend to solve generalized lattice agreement. The extensions are described in Algorithm 2. The proposer process has two new actions Receive() and Buffer(). In generalized lattice agreement problem, input values belonging to the lattice arrive, over time, in an unbounded fashion. We model this using the action Receive(), which can be executed an unbounded number of times. In addition, the action Propose() changes to the pseudo code shown in Algorithm 2. All the other actions, ProcessACK(), ProcessNACK(), Refine() and Decide() are the same as shown in Algorithm 1. For convenience, we introduce a new type of processes called *Learners*, which learn values chosen by acceptors. We change the Accept action in acceptors to send acknowledgments to all learners.

The goal of generalized lattice agreement is to ensure that learnt values form a chain and every input value is eventually learnt. Our algorithm achieves this in two stages. First, the algorithm ensures that a received value is eventually included in the proposed value of some proposer. Second, the algorithm ensures that every value proposed by a proposer is eventually learnt, using repeated iterations of the basic lattice agreement algorithm.

The first goal above can be trivially achieved by a proposer if it replaces its current value by the join of its current value and the received value whenever any new value is received. Unfortunately, this can cause non-termination in the second stage algorithm. Recall that the informal termination argument for the basic lattice agreement algorithm exploits the fact that all computed values belong to a finite lattice. This is no longer true if every received value is immediately incorporated into proposed values.

As an example, consider a system with two proposers p_1 and p_2 and three acceptors, a_1, a_2 and a_3. Let L be a power set lattice $\mathcal{P}(\mathcal{I})$ defined over natural numbers. Consider an execution in which p_1 first proposes the value $\{1\}$, which is accepted by a_1 and a_2 (hence chosen). Subsequently p_2 proposes the value $\{2\}$. Since $\{2\} \not\sqsubseteq \{1\}$, at least one of the acceptors a_1 or a_2 will reject the proposal and update its accepted value to $\{1, 2\}$. Before p_2 refines and sends its proposal, assume p_1 receives a value $\{3\}$ and proposes $\{1, 3\}$. Since $\{1, 3\} \not\sqsubseteq \{1, 2\}$, at least one of the acceptors a_1 or a_2 will reject the proposal and update its accepted value to $\{1, 2, 3\}$. Furthermore, since L is an infinite lattice, it is easy to see that so long as proposers keep including newly received values in their proposals this process can continue without any value being chosen.

This lack of termination due to conflicting proposals is reminiscent of non-termination in Paxos. However, for the lattice agreement problem, we can guarantee termination by *controlling* when proposers can incorporate received values into new proposals. In our algorithm, a proposer *buffers* all new values it receives until it *successfully completes* its participation in a round of the basic lattice agreement problem instance. The variable *bufferedValues* contains the join of all values received since the last Propose. A proposer is said to successfully complete a round when it receives a majority of responses to its proposal that are all acceptances. When this happens, the proposer includes *bufferedValues* into its *next* proposed value during the subsequent execution of the action Propose() and moves into its next round.

With this modification, we can show that at least one of the proposers will successfully complete its round and process its buffered values. However, we cannot guarantee that all proposers will successfully complete their rounds and process their buffered values. For example, the same proposer p may successfully complete all the rounds and keep introducing new received values into the next round. As a result, none of the other proposers may ever receive a majority of acceptances to their proposals.

If a proposer does not successfully complete its round, then it will never include its buffered values into the second stage. Hence, values it receives may never be learnt. We get around this problem by making a proposer broadcast any new value it receives to all other proposers. Other proposers treat this broadcast value just like any other values they receive. Effectively all values are sent to all proposers. Since we have a liveness guarantee that at least some correct proposer will eventually successfully complete its current round and move on to its next round, and messages between correct processes are eventually delivered, we can guarantee that all received values are included in the second stage and eventually learnt.

In the next section, we formalize the notion of a round and will see that there is a distinction between a process completing a round and a process *successfully* completing a round. We will show that every process is guaranteed to complete every round, and at least one process is guaranteed to *successfully* complete a given round.

5. SAFETY AND LIVENESS

We now establish the safety and liveness properties of the generalized lattice agreement algorithm. We start with some definitions. An execution of the algorithm consists of each process executing a sequence of execution-steps. An execution-step (of a single process) consists of the execution of a single action. We identify the action executed in a step (as indicated in our algorithm description) by the name of the action and its parameters: e.g., Receive(v). If a process has one or more enabled actions, it selects one of its enabled actions (non-deterministically) and executes it atomically and repeats this process. A process may *fail* at any point in time (after which it does not execute any more execution-steps). We will use the term *correct* process (or *non-failed* process) to refer to a process that does not fail during the execution under consideration. Note that a process may fail in the middle of an execution step, and our algorithm guarantees safety properties even in this case.

Recall that N_a denotes the number of acceptor processes and N_p denotes the number of proposer processes. We will

Algorithm 2 Generalized Lattice Agreement

```
// Proposer Process
// All variables and actions specified in Algorithm 1 except Propose() are also included.
L $bufferedValues = \bot$

procedure ReceiveValue($value$)
    Send InternalReceive($value$) to all $Proposers \setminus \{UID\}$
    $bufferedValues = value \sqcup bufferedValues$

action Receive($value$)
guard: ?ExternalReceive($value$)
effect: ReceiveValue($value$)

action Buffer($value$)
guard: ?InternalReceive($value$)
effect: $bufferedValues = value \sqcup bufferedValues$

action Propose()
guard: $status = passive$ && $proposedValue \sqcup bufferedValues \sqsupset proposedValue$
effect:
    $proposedValue = proposedValue \sqcup bufferedValues$
    $status = active$
    $activeProposalNumber{+}{+}$
    $ackCount = nackCount = 0$
    Send Proposal($proposedValue$, $activeProposalNumber$, $UID$) to all $Acceptors$
    $bufferedValues = \bot$

// Acceptor Process
// All variables and actions specified in Algorithm 1 except Accept() are also included.
action Accept($proposalNumber$, $proposedValue$, $proposerId$)
guard: ?Proposal($proposalNumber$, $proposedValue$, $proposerId$) && $acceptedValue \sqsubseteq proposedValue$
effect:
    $acceptedValue := proposedValue$
    Send ACK($proposalNumber$, $proposedValue$, $proposerId$) to $proposerId$ and all $Learners$

// Learner process
L $learntValue = \bot$
int $ackCount$[int,int] // all initially 0

action Learn($proposalNumber$, $value$, $proposerId$)
guard: ?ACK($proposalNumber$, $value$, $proposerId$)
effect:
    $ackCount[proposerId][proposalNumber]{+}{+}$
    if ($ackCount[proposerId][proposalNumber] \geq \lceil (N_a + 1)/2 \rceil$ && $learntValue \sqsubset value$) then
        $learntValue = value$
    endif

procedure L LearntValue()
effect:
    return $learntValue$
```

assume in the sequel that N_p is at least 2. (Otherwise, the problem and proofs are trivial.)

A set S of acceptors is said to be a *quorum* iff $|S| \geq \lceil (N_a + 1)/2 \rceil$. Note that by definition any two quorums have a non-empty intersection.

A value $v \in$ L is said to be a *received value* if and when some proposer executes the step Receive(v). A value v is said to have been *initiated* if some proposer executes an Propose step where it proposes v. A value v is said to have been *proposed* if some proposer executes either an Propose or Refine step where it proposes v. A proposal (n, v, id) is said to have been *chosen* if some quorum of acceptors execute Accept(n, v, id). A value v is said to have been *chosen* if some proposal (n, v, id) has been chosen. A value v is said to have been *decided* if and when some proposer executes a Decide(v) step. A value v is said to have been *learnt* by a learner process p if and when p executes the step Learn(w) for some $w \sqsupseteq v$. (Note that the definition of a learnt value differs from the preceding definitions, to be consistent with the problem definition and requirements.)

5.1 Safety

Simpler proofs have been omitted in the sequel.

Theorem 5.1. *The value of proposedValue (of any proposer), acceptedValue (of any acceptor), and learntValue (of any learner) can all be expressed as the join of some subset of previously received input values.*

Proof. Follows by induction on the length of execution. □

Corollary 5.2. *Validity: Any learnt value is a join of a subset of received values.*

Theorem 5.3. *Stability: The value of learntValue of any learner increases monotonically over time.*

Lemma 5.4. *Values acknowledged by an acceptor increase monotonically: for any two steps* $\mathsf{Accept}(n, v, id)$ *and* $\mathsf{Accept}(n', v', id')$ *executed by the same acceptor (in that order),* $v \sqsubseteq v'$.

Lemma 5.5. *Any two chosen values u and v are comparable.*

Proof. Let $Quorum(u)$ be the quorum of acceptors that acknowledged a proposal with value u. Let $Quorum(v)$ be the quorum of acceptors that acknowledged a proposal with value v. Let $A = Quorum(u) \cap Quorum(v)$. By the definition of a quorum, A must be non-empty. Consider any acceptor $a \in A$. a must have acknowledged both u and v. Therefore, from Lemma 5.4, $u \sqsubseteq v$ or $v \sqsubseteq u$. □

Theorem 5.6. *Consistency : Any two values u and v learnt by two different processes are comparable.*

Proof. Follows from Lemma 5.5 since every learnt value is a chosen value. □

5.2 Liveness

We now show that our algorithm guarantees liveness (subject to our failure model described earlier) and derive its complexity. The key idea behind the proof is to establish that any execution can be partitioned into a sequence of *rounds* such that:

- A new value is chosen in every round, and
- Every round is guaranteed to terminate. In particular, we show that each proposer can propose at most N_p+1 times within a single round.
- Every value proposed in a round is included in the value chosen in the same round.
- If a value has been received by all correct processes in a round, then it is included in a proposal in the next round.

All of the following results apply to a given execution.

Lemma 5.7. *Values proposed by a single proposer strictly increase over time: if v_i is the value proposed by proposer p in proposal number i and v_{i+1} is the value proposed by p in proposal number $i+1$, then $v_i \sqsubset v_{i+1}$.*

Definition Let $v_1 \sqsubset v_2 \sqsubset \cdots \sqsubset v_k$ be the set of all values chosen in a given execution. Define v_0 to be $\bot$. We partition the sequence of execution-steps of a proposer p into rounds as follows. Let v denote the value p's *proposedValue* at the end of an execution-step s. Step s is said to belong to the initial (dummy) round 0 if $v = \bot$. Step s is said to belong to the (last) round $k+1$ if $v \not\sqsubseteq v_k$. Otherwise, step s belongs to the unique round r that satisfies $v \sqsubseteq v_r$ and $v \not\sqsubseteq v_{r-1}$. We refer to round $k+1$ as an *incomplete* round and every other round as a *completed* round.

Note that the above definition of rounds is consistent with the ordering of events implied by message delivery: if execution-step e_1 sends a message that is processed by an execution-step e_2, then e_1 belongs to the same round as e_2 or an earlier round.

Lemma 5.8. *Let v be a value proposed in a completed round r and let v_r be the value chosen in round r. Then, $v \sqsubseteq v_r$.*

Proof. Follows from the definition of rounds. □

Lemma 5.9. *A proposer executes at most one* Decide *step and at most one* Propose *in a round.*

Proof. A proposer must alternate between Propose and Decide steps. Suppose a proposer initiates v_1, then decides v_2, then initiates v_3 and then decides v_4. We must have $v_1 \sqsubseteq v_2 \sqsubset v_3 \sqsubseteq v_4$. Note that v_2 must be a chosen value and, hence, $\mathsf{Decide}(v_2)$ must mark the end of a round for the proposer. The result follows. □

Lemma 5.10. *Assume that a proposer makes at least two proposals in a round r. Let w_1 and w_2 denote the first and second value it proposes. Let v_{r-1} be the value chosen in round $r-1$. Then, $v_{r-1} \sqsubseteq w_2$.*

Proof. The proposer must have received responses from a quorum of acceptors for proposal w_1 before it proposes w_2. At least one of these acceptors, say a, must have acknowledged the value v_{r-1}. Furthermore, a must have acknowledged v_{r-1} before responding to proposal w_1. (Otherwise, we would have $w_1 \sqsubseteq v_{r-1}$, which contradicts the definition of rounds.) Hence, regardless of whether a ACKs or NACKs w_1, we have $v_{r-1} \sqsubseteq w_2$. □

Lemma 5.11. *A proposer can execute at most N_p+1 proposals in a single round.*

Proof. Consider a round r in which a proposer p proposes a sequence of values $w_1 \sqsubset w_2 \sqsubset \cdots \sqsubset w_k$. Let *Prev* denote the set of all values initiated in a round $r' < r$. Let *Curr* denote the set of all values initiated in round r. Note that *Curr* can contain at most one value per proposer (from Lemma 5.9). Define *All* to be *Prev* $\cup$ *Curr*.

Note that each proposed value w_i can be expressed as the join of some subset of values in *All*. (This follows similar to Theorem 5.1.) Define $covered_j$ to be $\{u \in All \mid u \sqsubseteq w_j\}$. Thus, we have $w_j = \bigsqcup covered_j$.

It follows from above that $covered_1 \subset \cdots \subset covered_k$ is a strictly increasing sequence of subsets of $All = Prev \cup Curr$.

Now, we show that $covered_2 \supseteq Prev$. Let $v \in Prev$. Since v was initiated in an earlier round $r' < r$, we must have $v \sqsubseteq v_{r-1}$, where v_{r-1} is the value chosen in round $r-1$. It follows from Lemma 5.10 that $v_{r-1} \sqsubseteq w_2$.

Putting these together, it follows that the strictly increasing sequence $covered_1 \subset \cdots \subset covered_k$ can have a length at most $N_p + 1$.

□

Liveness Assumptions. Note that all the preceding results hold in the presence of arbitrary failures and message losses. The following progress guarantees, however, require the following extra assumptions: (a) A majority of acceptor processes are correct, (b) At least one proposer process is correct, (c) At least one learner process is correct, and (d) All messages between correct processes are eventually delivered. (Note that if every process simultaneously plays the roles of proposer, acceptor, and learner, then the first three assumptions simplify to the assumption that a majority of the processes are correct.)

Lemma 5.12. *Every round eventually terminates.*

Proof. Follows from Lemma 5.11. □

Lemma 5.13. *Every initiated value is eventually included in a chosen value.*

Proof. Follows from Lemma 5.12 and Lemma 5.8. □

Lemma 5.14. *Every value received by a correct process is eventually proposed.*

Proof. Consider a value u received by some correct process. By definition of Receive, u is sent to all proposers. It is eventually delivered to all correct proposers. If any correct proposer P that receives u is passive when it receives u, it will initiate u. However, if P is active, then it is in the middle of some round. It follows from Lemma 5.11 that eventually some proposer P' must execute a Decide step. Therefore, some correct process will eventually propose u. □

Theorem 5.15. *Every value received by a correct process is eventually learnt by every correct learner.*

Proof. Implied by Lemma 5.13 and Lemma 5.14. Note that once a value is chosen, a correct learner will eventually receive the acknowledgments sent for the chosen value and learn it. □

5.3 Time Complexity

We now establish the complexity of the generalized lattice agreement algorithm, in terms of the number of message delays (as described in Section 3). Recall that when a new value v is received, it is first sent to all correct proposers via the action *NewInput*. This takes one unit of time. Consider the last proposer to receive this value. By Lemma 5.11 we know that within the next N_p+1 proposals (i.e., $2\times(N_p+1)$ units of time) some proposer will decide and will propose v in the next round. By Lemma 5.8 and Lemma 5.11, v will be chosen in the next $N_p + 1$ proposals. Every chosen value will be learnt by the learners after one message delay. Putting this all together it can be seen that every value received by a correct process is learnt in $O(N)$ units of time. Lemma 5.11 also implies that if only k processes receive values, each received value will be chosen in $O(k)$ message delays. In the extreme case, if only one process receives values, each received value can be learnt in three message delays (from the client to the proposer, from the proposer to acceptors, and acceptors to client, assuming the client acts as a learner). This is one message delay more than the fast path of two message delays in Paxos [11]. Improving this bound while preserving wait-freedom remains an open problem.

6. STATE MACHINE REPLICATION USING LATTICE AGREEMENT

State machine replication is a general approach for implementing data-structures that can tolerate process failures by replicating state across multiple processes. In one common approach to state machine replication, replica processes receive requests or commands from clients, utilize consensus to decide on the order in which commands must be processed, and apply commands to the local replica of the state machine in that order. If the state machine is deterministic and no Byzantine faults occur, each correct process is guaranteed to generate the same responses and reach the same state. Unfortunately, the undecidability of consensus means that this approach cannot guarantee liveness in the presence of failures.

In this paper, we consider a special class of state machines. We first assume that operations of the state machine can be classified into two kinds: *updates* (operations that modify the state) and *reads* (operations that do not modify the state, but return a value). Thus, an operation that modifies the state and returns a value is not permitted. Furthermore, we assume that all update operations commute with each other and are deterministic. Several data types such as sets, sequences, certain types of key-value tables, and graphs [12] can be designed with commuting updates.

There are several approaches for implementing such state machines, each with different consistency and performance characteristics. One approach is to allow each replica process to process reads and updates in arbitrary order. This approach requires no co-ordination between processes and guarantees that as long as all commands are eventually delivered, all correct processes *eventually* reach the same state. However, this approach does not provide strong consistency guarantees, such as linearizability or serializability, for reads.

Both linearizability and serializability guarantee that the observed behavior of the replicated state machine on some set of (possibly concurrent) operations is the same as the behavior of the state machine (with no replication) for some sequential execution (the "witness") of the same set of operations. Linearizability provides the additional guarantee that any two temporally non-overlapping operations (in the execution) occur in the same order in the "witness".

One approach to guarantee linearizability, based on generalized consensus, is for processes to agree on a partial order on the commands that totally orders every read command with every update command that it does not commute with. This alternative guarantees linearizability but requires the use of consensus to compute the partial order, which is impossible in the presence of failures.

Serializability Using Lattice Agreement. Algorithm 3 describes a wait-free algorithm for state machine replication based on generalized lattice agreement that guarantees serializability. In this algorithm, the lattice L is defined to be the power set of all update commands with the partial order $\sqsubseteq$ defined to be set inclusion. (We use the term "update command" to refer to *instances* of update operations.) We

Algorithm 3 Serializable ReplicatedStateMachine

```
procedure ExecuteUpdate($cmd$)
    ReceiveValue( $\{cmd\}$)

procedure State Read()
    return Apply(LearntValue())
```

Algorithm 4 Linearizable ReplicatedStateMachine

```
procedure ExecuteUpdate($cmd$)
    ReceiveValue( $\{cmd\}$)
    waituntil $cmd \in$ LearntValue()

procedure State Read()
    ExecuteUpdate(CreateNop())
    return Apply(LearntValue())
```

refer to a set of update commands as a *cset*. In this setting, update commands can be executed by proposers and read commands can be executed by learners. A proposer executes an update command *cmd* by simply executing the procedure ReceiveValue($\{cmd\}$), taking the singleton set $\{cmd\}$ to be a newly proposed value. Reads are processed by computing a state that reflects all commands in the learnt *cset* (applied in arbitrary order).

Due to the properties of generalized lattice agreement, it is easy to see that this algorithm is wait-free and serializable, with both reads and updates requiring $O(N)$ message delays. Furthermore, this algorithm guarantees *progress*: every update operation will eventually be reflected in all read operations (subject to our failure model). Note that the execution of an update command *cmd* completes (from a client's perspective) when the set $\{cmd\}$ has been sent to all proposers. For simplicity, assume that each process acts as a proposer, acceptor and learner. Progress is guaranteed as long as message delivery is reliable and at least a majority of the processes are correct.

Linearizability Using Lattice Agreement. We now extend the preceding algorithm to guarantee linearizability, as shown in Algorithm 4. For simplicity, we assume that each process acts as a proposer, acceptor and learner. The first challenge is to preserve the order of non-overlapping update operations: if one replica completes the execution of an update operation c_1 before another replica initiates the execution of an update operation c_2, then we must ensure that c_1 occurs before c_2 in the linearization order. This is not guaranteed by the serializable algorithm presented above.

We extend the execution of an update operation as follows. When a process executes an update command *cmd*, it includes the command in the next proposal (as before), and then *waits* until the command has been learnt. This preserves the order of non-overlapping update operations.

The second challenge concerns a read operation that is initiated after an update operation completes. In this case, we need to ensure that the value returned by the read operation reflects the effects of the update operation. We assume that the set of update commands includes a special *no-op* command which does not modify the state. Reads are processed by creating a new instance of the *no-op* command, executing this command, and then computing a state that reflects all commands in the learnt *cset*(applied in arbitrary order).

Optimizations. There are several simple ways of optimizing the basic algorithms presented above. Every invocation of Apply does not have to recompute state by executing all commands in LearntValue(). Instead, the implementation can exploit the monotonic nature of learnt values and the commutativity of update commands to incrementally compute state.

7. RELATED WORK

As mentioned before, the lattice agreement problem has been studied previously in an asynchronous shared memory setting [4, 3, 7]. In [7], Inoue *et al.* propose a lattice agreement algorithm which requires $O(M)$ register operations when processes are assigned identifiers in the range $[1-M]$. The problem of assigning N processes names in a range $[0-poly(N)]$ is referred to as the re-naming problem [2] and has been studied before. The best known algorithms for assigning identifiers in the range $[0-O(N)]$ have a complexity of $O(Nlog(N))$. Hence the complexity of the Inoue *et al.* algorithm expressed in terms of number of participating processes is $O(Nlog(N))$, if the cost of naming is taken into account. Algorithms whose complexity depends only on the number of participant processes and not on the range of identifiers assigned to processes are called *range-independent* algorithms [3]. Further, an algorithm is *adaptive* if its complexity only depends on the number of participants actively contending at any given point in time [3]. The Inoue *et al.* algorithm is not adaptive. The algorithm proposed in [3] is both adaptive and range independent, it has a complexity of $O(Nlog(N))$ where N is the number of active participants. The construction of the algorithm requires a series of re-naming algorithms that are carefully put together to obtain a range independent and adaptive lattice agreement algorithm.

Some of the shared memory algorithms can be translated into message passing algorithms using emulators [1]. In particular, any shared memory algorithm that only uses single-writer multiple-reader atomic registers can be translated into a message passing algorithm with each read and write operation requiring only constant ($O(1)$) message delays. A direct emulation of the above algorithms using emulators from [1] leads to asynchronous message passing algorithms with complexity $O(Nlog(N))$.

In this paper we provide an asynchronous message passing algorithm for lattice agreement with complexity $O(N)$. Our algorithm is *range independent*, as it only requires the identifiers of all participating processes be unique and does not rely on any renaming steps. Our algorithm is also *adaptive* since the number of proposal refinements executed by a proposer depends only on the number of active proposers. We further show that the algorithm can be extended, without changing the complexity, to a generalization of the lattice agreement problem where processes receive a (potentially unbounded) sequence of values from a lattice and learn a sequence of values that form a chain.

8. REFERENCES

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42, 1995.

[2] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an

asynchronous environment. *J. ACM*, 37:524–548, July 1990.

[3] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31, February 2002.

[4] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distrib. Comput.*, 8, March 1995.

[5] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 2(32):374–382, April 1985.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, July 1990.

[7] Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG '94. Springer-Verlag, 1994.

[8] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[9] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, May 1998.

[10] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, April 2005.

[11] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (104):67–88, 2011.