Asynchronous Resilient Linearizability

Sagar Chordia[1], Sriram Rajamani[2], Kaushik Rajan[2], G. Ramalingam[2], and Kapil Vaswani[2]

[1] IIT Bombay
[2] Microsoft Research India
chordiasagar14@gmail.com,(sriram,krajan,grama,kapilv)@microsoft.com

Abstract. In this paper we address the problem of implementing a distributed data-structure that can tolerate (non-byzantine) process failures in an asynchronous message passing system, while guaranteeing correctness (linearizability with respect to a given sequential specification) and resiliency (the operations are guaranteed to terminate, as long as a majority of the processes do not fail). We consider a class of data-structures whose operations can be classified into two kinds: *update* operations that can modify the data-structure but do not return a value and *read* operations that return a value, but do not modify the data-structure. We show that if every pair of update operations commute or nullify each other, then resilient linearizable replication is possible for the data-structure. We propose an algorithm for this class of data-structures with a message complexity of two message round trips for all read operations and O(n) round trips for all update operations. We also show that if there exists some reachable state where a pair of idempotent update operations neither commute nor nullify each other, resilient linearizable replication is not possible.

1 Introduction

In this paper, we focus on computation in a standard asynchronous message-passing distributed computing setting with n processes, in the presence of non-byzantine (stopping) process failures. The standard correctness criterion in this setting is linearizability [7] with respect to a given sequential specification. A desirable progress guarantee in this setting is *t-resiliency*, which guarantees that all operations terminate as long as no more than t processes fail. We consider the case where t is $\lceil n/2 \rceil - 1$: that is, we assume that a majority of the n processes do not fail. We address the question of when (i.e., for which data-structures or sequential specifications) it is possible to design a resilient linearizable algorithm in this setting.

We consider a class of data-structures whose operations can be classified into two kinds: *update* operations that can modify the data-structure but do not return a value and *read* operations that return a value, but do not modify the data-structure. We show that if every pair of update operations commute or nullify each other, then resilient linearizable replication is possible for the data-structure. We propose a algorithm for this class of data-structures with a message complexity of two message round trips for all read operations and O(n) round trips for all update operations. The algorithm is based on the insight that if all operations commute, the order in which operations are applied is irrelevant for the final state produced by a given set of update operations. This reduces the problem to that of ensuring that reads observe monotonically increasing sets of operations and respecting the real time ordering between non-concurrent operations. The extension for nullifying operations is more complex, but is based on the intuition that when an earlier operation is nullified by a later operation, the execution of the earlier operation is optional.

We also show that if there exists some reachable state where a pair of idempotent update operations neither commute nor nullify each other, resilient linearizable replication is not possible. This result is based on a reduction from consensus to resilient linearizable state machine replication.

These results show that resilient linearizable algorithms are possible for some interesting data-structures. We also show how these results can help design certain data-structure specifications so that a resilient linearizable implementation is possible, by addressing the design of a simple graph data-structure. We present two closely related graph specifications, where a resilient linearizable algorithm is possible for one specification but not the other.

2 The Problem

We assume a standard asynchronous computation setting with non-byzantine (stopping) process failures. We have n processes that communicate via messages. All messages are assumed to be eventually delivered, but no bound is assumed on the time taken for a message to be delivered and no assumptions are made about the order in which messages are delivered. We are interested in $(\lceil n/2 \rceil - 1)$-resilient algorithms: algorithms that can guarantee progress as long as a majority of the n processes do not fail. In the sequel, we will use the term resilient as short-hand for $(\lceil n/2 \rceil - 1)$-resilient.

State machine replication is a general approach for implementing data-structures that can tolerate process failures by replicating state across multiple processes. The key challenge in state machine replication is to execute data-structure operations on all replicas such that linearizability can be guaranteed.

A state machine models a system that implements an interface consisting of a set of procedures. Every procedure has a set of parameters and we assume that the parameters are of primitive type. In the sequel, we will use the term *operation* to refer to a tuple of the form $(p, a_1, \cdots, a_n)$ consisting of the name p of the procedure invoked as well as the actual values $a_1, \cdots, a_n$ of the parameters. A state machine $\mathfrak{m}$ consists of a set of states $\Sigma_{\mathfrak{m}}$. The semantics of an operation is given by a function that maps an input state to an output state as well as a return value.

UQ State Machines. In this paper, we consider a special class of state machines we refer to as *Update-Query* (UQ) state machines. We assume that operations of the state machine can be classified into two kinds: *updates* (operations that modify the state) and *queries* (also called *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, the operations on the data-structure are assumed to be deterministic. The semantics of an update operation *op* is given by a function $[\![op]\!] : \Sigma_{\mathfrak{m}} \rightarrow \Sigma_{\mathfrak{m}}$.

Thus, a UQ state machine does not allow for any operation that modifies the state and returns a value. While this is a convenient simplification, it does not restrict expressiveness, as long as we are able to associate every update operation invocation with an unique identifier. We can then use a separate query operation, with the unique identifier as a parameter, to obtain the return-value associated with the corresponding update operation.

Algorithm 1 NC State Machine Replication Algorithm (Process k)

```
 1: DistributedSet cset = {}
 2:
 3: procedure ExecuteUpdate(op)
 4:     let ts = get-time-stamp() in
 5:     cset.add ( (ts,op) )
 6:
 7: procedure State LinearizableRead()
 8:     let S = cset.read() in
 9:     return Apply(S)
10:
11: procedure int get-time-stamp()
12:     let S = cset.read() in
13:     return (S, k) // k is this process' unique-id
14:
15: procedure State Apply(S)
16:     let cmd_1, ··· , cmd_k = topological-sort(S, ≺_tn) in
17:     let (ts_i, op_i) = cmd_i in
18:     let s_0 = initial-state in
19:     let s_i = op_i(s_{i-1}) in
20:     return s_k
```

NC State Machines. We say that two update operations op_1 and op_2 *commute* iff $op_1(op_2(\sigma)) = op_2(op_1(\sigma))$ for every state σ. We say that an operation op_1 *nullifies* an operation op_2 iff $op_1(op_2(\sigma)) = op_1(\sigma)$ for every state σ: in other words, op_1 nullifies op_2 iff $op_1 \circ op_2 = op_1$. We write $op_1 \succcurlyeq op_2$ to denote that op_1 nullifies op_2. We say that two operations op_1 and op_2 *nullify* if $op_1 \succcurlyeq op_2$ or $op_2 \succcurlyeq op_1$.

A UQ state machine is said to be a *NC state machine* if for any pair of operations f and g, f and g commute or f nullifies g or g nullifies f.

Lemma 1. *$\succcurlyeq$ is a transitive relation: if $f \succcurlyeq g$ and $g \succcurlyeq h$, then $f \succcurlyeq h$.*

Proof. All proofs appear in the appendix.

We say that a partial-ordering $\leq_s$ on a set of update operations (of the given state machine) is an NC-ordering if it satisfies the following conditions, where we write $x <_s y$ as shorthand for $x \leq_s y$ and $x \neq y$:

1. if $op_1 <_s op_2$, then op_2 nullifies op_1.
2. if $op_1 \not<_s op_2$ and $op_2 \not<_s op_1$, then op_1 and op_2 commute.

Lemma 2. *Every NC State Machine has a NC-ordering on the set of all its update operations.*

Many well known data-structures like read-write registers, read-write memory, counters, maps, sets are NC state machines. We present details of these and other such data-structures in the Appendix.

3 Replication For NC State Machines

We now describe our replication algorithm for an NC state machine. Assume that we have n replicas (processes). External clients may submit operations (either updates or reads) to any replica. Note that the same operation (e.g., an increment operation on a counter) may be invoked multiple times in an execution. We refer to each distinct invocation of an update operation as a *command*.

Our algorithm for NC state machine replication makes use of a resilient linearizable implementation of a distributed add-only set that provides an update operation *add*(v) to add an element v to the set and a read operation *read*() that returns the current value of the set. We describe an implementation of this data-type in Section 4.

Our replication algorithm for NC state machines is shown in Algorithm 1. The basic idea behind the algorithm is as follows. We utilize the add-only set to maintain the set of all commands executed so far, referred to as *cset* below. Executing an update operation essentially involves adding an element, representing this operation invocation, to *cset*. Read operations are realized by getting the current value S of *cset*, and then materializing the state σ_S corresponding to this set S of commands.

The key challenge is in defining σ_S so that the desired consistency criterion (linearizability) is satisfied.

Capturing Execution Ordering Constraints Between Non-Overlapping Operations

The definition of linearizability requires that the observed behavior of an execution π consisting of a set of operation-invocations is equivalent to that of some sequential execution π_s of the same set of operation-invocations. Furthermore, this sequential execution π_s must preserve the order of non-overlapping operation-invocations in π.

We associate a *timestamp* with each command. This timestamp serves two purposes. First, it lets us conservatively identify non-overlapping operaton-invocations, as explained soon. Second, it ensures that different invocations of the same operation are represented by different command instances. This is important since *cset* is a set and not a multi-set.

Specifically, a replica k that receives an update operation o augments it with a timestamp ts and represents the operation invocation as a pair (ts, op). We define the timestamp to be the pair consisting of the current value of *cset*, obtained by replica k via a read operation on the distributed-set, paired with the unique-id k of the replica. The replica-id distinguishes between different concurrent invocations of the same operation at different replicas. (Note that each replica processes its requests, both updates as well as queries, sequentially.) We refer to the ordered pair (ts, op) as an *update command*. Given any update command $c = (t, o)$, we define $op(c)$ to be o. The set *cset* is used to track the set of all executed update commands.

We define a relation $\prec_t$ on commands, as follows: $c_1 = ((cset_1, id_1), op_1) \prec_t c_2 = ((cset_2, id_2), op_2)$ iff $c_1 \in cset_2$. We say that $c_1 \parallel c_2$ iff $(c_2 \not\prec_t c_1) \wedge (c_1 \not\prec_t c_2)$. Note that these relations help determine whether two update commands are concurrent and the ordering relation between non-concurrent update commands, as follows:

Lemma 3. *For any two commands c_1 and c_2 in an execution, if c_1 completes before c_2 starts, then $c_1 \prec_t c_2$. Hence, if $c_1 \parallel c_2$, then the execution of c_1 and c_2 overlap.*

Lemma 4. *Let X denote the value of* cset *at some point during an execution and let Y denote the value of* cset *at a later point in the same execution. Then, (a) $X \subseteq Y$, and (b) there exists no $x \in X, y \in (Y \setminus X)$ such that $y \prec_t x$.*

Thus, even though *cset* is a set, the representation lets us determine the order in which non-overlapping operations must be executed.

Consistently Ordering Concurrent Operations

Linearizability permits an implementation to execute concurrent (or overlapping) operations in any order. The challenge, however, lies in ensuring that all replicas execute these operations in the same order. More precisely, we need a scheme that, given the value Y of $cset$ at any point in time, chooses an order in which concurrent operations in Y are executed so that the following constraints are satisfied: (a) Different processes that evaluate the same set Y of update commands produce the same state. (b) Let $Y_1 \subseteq Y_2 \cdots Y_k$ denote some sequence of values of $cset$ during an execution. The states obtained by evaluating each of the sets $Y_1, \cdots Y_k$ must correspond to states produced by the execution of increasing prefixes of a single sequential execution of the update commands in Y_k. Below we describe a way to order concurrent operations in a *cset* that satisfies both the above requirements. Consider concurrent operations in a *cset*.

Concurrent Commuting Operations. It is not necessary to determine the order in which two commuting update operations in a *cset* must be executed, as the resulting state is independent of the order in which commuting updates are applied.

Concurrent Non-Commuting Operations. However, we must determine a unique ordering among non-commuting concurrent update operations so that we can have a well-defined notion of the state σ_S corresponding to a set S of commands (i.e., to ensure requirement (a) above). We utilize the NC-ordering relation on the update operations for this purpose. Note that the NC-ordering is a static ordering relation between any pair of non-commuting update operations. Let $\prec_s$ be a NC partial-order on the set of all update operations.

Given a $cset$ value Y, we define the relation $\prec_n^Y$ on elements of Y recursively as follows:

$$c_1 \prec_n^Y c_2 \text{ iff } c_1 \parallel c_2 \wedge (op(c_1) \prec_s op(c_2)) \wedge (\nexists c_3 . c_2 \prec_n^Y c_3 \prec_t^* c_1).$$

More precisely, we define the relation $\prec_n^Y$ inductively, by considering pairs of elements (c_1, c_2) in topological sort order, with respect to $\prec_t$, so as to satisfy the above constraint. Intuitively, we consider any pair of commands c_1 and c_2 that are concurrent (i.e., $c_1 \parallel c_2$). If these commands do not commute, then we utilize the static nullification ordering relation between the operations of c_1 and c_2 to determine the $\prec_n^Y$ ordering between them. However, we do not add this extra ordering constraint if we have already established an ordering constraint $c_2 \prec_n^Y c_3$ that transitively (in combination with $\prec_t$ establishes an ordering between c_1 and c_2.

We further define a "combined" ordering relation $\prec_{tn}^Y$ to be the union of $\prec_t$ and $\prec_n^Y$:

$$c_1 \prec_{tn}^Y c_2 \text{ iff } c_1 \prec_t c_2 \vee c_1 \prec_n^Y c_2.$$

If no confusion is likely, we will abbreviate $\prec_n^Y$ to $\prec_n$ and $\prec_{tn}^Y$ to $\prec_{tn}$.

The following example illustrates the use of this recursive constraint, which is meant to ensure that the combined relation $\prec_{tn}$ does not have cycles. Consider an execution history consisting of three commands c_1, c_2, and c_3 such that $c_1 \parallel c_2$, $c_2 \parallel c_3$, while $c_1 \prec_t c_3$. Further assume that $op(c_3) \prec_s op(c_2) \prec_s op(c_1)$. In this case, we add the constraint $c_2 \prec_n c_1$, since c_2 and c_1 are concurrent and c_2 is nullified by c_1. However, we do not add the constraint $c_3 \prec_n c_2$ even though c_3 and c_2 are concurrent and c_3 is nullified by c_2 because we already have: $c_2 \prec_n c_1 \prec_t c_3$.

Lemma 5. *If $a \prec_n^Y b \prec_n^Y c$, then we must have $a \prec_{tn}^Y c$.*

Lemma 6. *$\prec_{tn}^Y$ is an acyclic relation. (In other words, the transitive closure of $\prec_{tn}^Y$ is irreflexive.)*

Let $\prec_{tn}^*$ denote the transitive closure of $\prec_{tn}$. We will write $a \parallel_{tn} b$ to denote that $(a \not\prec_{tn}^* b) \wedge (b \not\prec_{tn}^* a)$.

Lemma 7. *If $a \parallel_{tn} b$, then $op(a)$ and $op(b)$ commute.*

Given a sequence π of update commands, let $\mathrm{state}[\pi]$ denote the state produced by the execution of the updates in π in order.

Lemma 8. *Let S denote the value of cset at some point in an execution. Let π_1 and π_2 denote any two topological sort ordering of S with respect to the acyclic relation $\prec_{tn}$. Then,* $\mathrm{state}[\pi_1] = \mathrm{state}[\pi_2]$.

Given a set of update commands S, let $\overrightarrow{S}$ denote any sequence obtained by topologically sorting S with respect to the partial ordering $\prec_{tn}$. We define the state σ_S to be the state $\mathrm{state}[\overrightarrow{S}]$ obtained by executing the update commands in $\overrightarrow{S}$ in order. It follows from the previous lemma that σ_S is well-defined.

Consistency Across Csets. We now have a precise definition of the state σ_S produced by a set of commands S. This ensures that different replicas will produce the same state for the same set of commands. However, this is not sufficient for correctness. We need to establish that this way of constructing the state of a *cset* also ensures that the values produced by different sets of commands are consistent with each other. Note that as the *cset* is linearizable if two reads return different *csets* then one must necessarily be a subset of the other and all commands in the smaller *cset* will necessarily be $\prec_t$ or $\parallel$ with respect to commands in the larger *cset*.

Lemma 9. *Let $X \subseteq Y$ be two values of cset in a given execution. Then, $\prec_{tn}^X = \prec_{tn}^Y \cap (X \times X)$. (In other words, the ordering chosen between elements of X does not change over time.)*

Let $A \subset B$ denote two different values of *cset*. We need to ensure that the values σ_A and σ_B are consistent with each other. In particular, we need to show that σ_A and σ_B are states produced by executing some sequential executions π_A and π_B, respectively, where π_A is a prefix of π_B. We now show how we can construct these witness sequences π_A and π_B.

The simple case is when we can let π_A be $\overrightarrow{A}$ (a topological-sort ordering of A) and π_B be a topological-sort ordering of B that is also consistent with π_A. This can be done as long as we do not have a pair of operations $op_1 \in A$ and $op_2 \in (B \setminus A)$ such that $op_2 \prec_s op_1$.

The case where $op_2 \prec_s op_1$ for some $op_1 \in A$ and $op_2 \in (B \setminus A)$ requires more careful consideration. Note that the value σ_A is produced by executing op_1 but not op_2. However, our scheme above requires executing op_2 before op_1 when computing σ_B. We exploit the *nullification* property to deal with this issue. Note that the definition of a NC-ordering requires that op_1 nullify op_2 if $op_2 \prec_s op_1$. Hence, even though σ_A was computed without executing op_2, the nullification property guarantees that $\sigma_A = \sigma_{A'}$ where $A' = A \cup \{op_2\}$. Hence, we simply let π_B be $\overrightarrow{B}$ and we let π_A be the smallest prefix of π_B that includes all elements of A. We can show that the state produced by executing π_A is the same as the state σ_A produced by executing $\overrightarrow{A}$. Hence σ_A and σ_B are still consistent with each other. Based on the above discussion the following lemma can be proved.

Lemma 10. *Let $X \subseteq Y$ be two values of cset in a given execution. Then,* $\mathrm{state}[\overrightarrow{X}] = \mathrm{state}[\overrightarrow{Y} \downarrow X]$. *Where $\overrightarrow{Y} \downarrow X$ is the smallest prefix of $\overrightarrow{Y}$ that includes all elements of X.*

Theorem 1. *The NC state machine replication algorithm (Algorithm 1) is both linearizable and resilient.*

4 A Resilient Linearizable Add-Only Set

Our algorithm presented in Section 3 makes use of a resilient linearizable add-only set: a set data-structure that provides operations to add an element and to read the current value. Such a set implementation can be realized as sketched in Faleiro *et al.* [5], which presents a $(\lceil n/2 \rceil - 1)$-resilient algorithm for solving the generalized lattice agreement problem and shows how that can be used to implement a UQ state machine in which all update operations commute. The Faleiro *et al.* algorithm has a complexity of $O(n)$ message round trips for both reads and updates. We now describe how the Faleiro *et al.* algorithm can be modified so that a read incurs only a two message round trip while retaining the $O(n)$ complexity of an update command.

4.1 Notation

We make use of the following language constructs to keep the algorithm description simple and readable.

We introduce a Majority Vote construct "`QuorumVote` $[f]$ g" where f is a "remote delegate", denoting code to be executed at other processes and g is a "callback" that represents the code to be executed locally to process the return values of f. On invocation of the `QuorumVote` construct at process P a message containing sufficient information to execute the remote delegate is broadcast to all other processes. Every process that receives this message executes the delegate and sends back the result of evaluating this delegate to p. Process p evaluates every received value x by applying the function g to x. The execution of the `QuorumVote` construct terminates when a majority (at least $\lceil n+1 \rceil/2$) of the responses from other processes have been received and processed by p. Finally, every execution of f and g executes atomically.

The remote delegate may access/modify the state variables of the remote process where it is executed. The references to the variables of the remote process are denoted "`remote!var`". The remote-delegate may also contain read-only references to the variables of the local process (the process executing the `QuorumVote` construct), which are denoted "`local!var`". The values of these local variables are evaluated when the `QuorumVote` construct starts executing. The callback is only allowed read the return value of f and read/modify local state. As for the other code, any code that needs to execute atomically is explicitly wrapped in an "`atomic`" construct.

The construct `asyncmap f` executes the remote-delegate `f` in every process. It is asynchronous: the construct completes execution once it sends the necessary messages and does not wait for the completion of the remote delegate execution.

The construct `when cond stmt` is a conditional atomic statement that executes when `cond` is true. It is equivalent to `atomic { if (cond) then stmt else retry }` .

4.2 The Algorithm

Read The local variable `current` of every process represents the latest value of the set that it is aware of. A process p processes a request for the current value of the set as follows (as shown in procedure `read`). It sends a request to every (other) process to get their own copy of `current`. It computes the union of the values returned by a majority of the processes. Once the responses of a majority of the processes have been received and processed, p has a correct (linearizable) value to be returned. However, before returning this value, it broadcasts this value to all other processes. Every recipient updates its own value of `current` (to be the union of its current value and the new value) and sends an ack back. Once p receives an ack back from a majority of the processes, it can complete the read operation.

Add Every process tracks the elements to be added to the set using a variable `buffer`. A process p processes a request to add an element e to the set by first adding it to its own buffer and then broadcasting a request to all other processes to add e to their own buffers. Elements to be added to the set are processed in batches by each process sequentially. If p is in the middle of processing the previous batch of elements (as indicated by the status variable `passive` being false), it then waits until the previous batch is processed.

Every process uses a local variable `proposed` to store its proposed (i.e., candidate) next value for the set and a local variable `accepted` that represents the join of all the proposed values it has seen so far. Process p begins by adding all elements in its buffer to the proposed new value. It then sends the proposed value to all other processes. Every recipient compares the newly proposed value with its accepted value. If the newly proposed value is a superset of its accepted value, it sends back an ACK. Otherwise, it sends back a NACK. In either case, it updates its accepted value to include the newly proposed value.

```
1  Set current = {}, proposed = {}, accepted = {};
2  boolean passive = true;
3
4  Set read() {
5     Set result = {};
6     QuorumVote [remote!current] (λx. result := result ∪ x);
7     QuorumVote [remote!current := remote!current ∪ local!result] (λack. ())
8     return result;
9  }
10
11 void add(e) {
12    atomic {buffer := buffer ∪ {e}};
13    asyncmap [remote!buffer := remote!buffer ∪ {local!e}];
14    when (passive} { passive := false; }
15    atomic {proposed := proposed ∪ buffer};
16    while (e ∉ current) {
17       NACKrecvd = false;
18          QuorumVote [let x = remote!accepted ⊆ local!proposed in
19            remote!accepted := remote!accepted ∪ local!proposed;
20            if (x) then (ACK,remote!accepted) else (NACK,remote!accepted)]
21          (λ(x,s). if (x = NACK) then NACKrecvd := true proposed := proposed ∪ s);
22       if (!NACKrecvd) then current = current ∪ proposed;
23    }
24       QuorumVote [remote!current := remote!current ∪ local!current] (λack. ())
25    passive := true;
26 }
```

Fig. 1. The Add-Only Set.

If process p gets back responses from a majority of the processes, and these are all ACKs, then p has succeeded. It updates its current value to be the last value it proposed. If p receives any NACKs, then it updates its proposed value to include the accepted value indicated by the NACK.

Process p stops the iterative loop when the element e to be added is contained in its current value. At this point, p broadcasts its current value and waits until a majority of the processes update their own current value appropriately. Then, the add operation is complete.

4.3 Correctness and Complexity

Consider any history (i.e., execution) of the algorithm. The following terminology is relative to a given history.

We refer to the execution of the `QuorumVote` in line [18] as a *proposal round* and the value of `local!proposed` as the *proposed* value of the round. We say that the proposal round is *successful* if it terminates without receiving any NACK and we say that the proposal round *failed* otherwise. We identify any successful proposal round by a pair (P, Q), where P is the proposed value and Q is the set of processes that accepted the proposal with an ACK. Note that Q constitutes a quorum: i.e., it consists of a majority of the processes. We say that a set value P has been *chosen* if it is the proposed value of a successful proposal.

The following key property of the algorithm is the basis for correctness. If (P_1, Q_1) and (P_2, Q_2) are two successful proposals in a single execution, then P_1 and P_2 must be comparable: that is, either $P_1 \subseteq P_2$ or $P_2 \subseteq P_1$. (This follows since $Q_1 \cap Q_2$ must be non-empty, as both Q_1 and Q_2 consist of a majority of the processes. Since every process ensures that the values it ACKs form an increasing chain, the result follows.) It follows that all chosen values form a chain in the powerset lattice.

We say that a set value P has been *learnt* iff P is a chosen value and the value of `current` $\supseteq P$ for a majority of the processes. It follows that the set of all learnt values also form a chain. The *maximal learnt value*, at any point in time, represents the latest learnt value: it represents the current state/value of the distributed set.

It can be shown that the following properties hold:

1. Any chosen value consists only of elements e for which an `add` operation has been invoked.
2. The value of the variable `current`, of any process, is always a chosen value.
3. When an invocation of `add`(e) completes, e belongs to the maximal learnt value (as ensured by line [24]).
4. The value R returned by a `read` operation is a learnt value.
5. The value R returned by an invocation of `read` contains the maximal learnt value at the point of the invocation of the `read` operation.
6. The value R returned by an invocation of `read` is contained in the maximal learnt value at the point of completion of the `read` operation. (as ensured by line [7]).

Linearizability We can show that the given history is linearizable by constructing an equivalent sequential history as follows.

1. For any two operations `add(x)` and `add(y)`, we order `add(x)` before `add(y)` if there exists a chosen value that contains `x` but not `y`.
2. For any two read operations op_1 and op_2, we order op_1 before op_2 if the value returned by op_1 is properly contained in the value returned by op_2.
3. For any two operations `add(x)` and `read()`, we order the add operation before the read operation iff the read operation returns a value containing `x` and the add operation was initiated before the read operation completed.

Resiliency and Complexity Every invocation of `QuorumVote` is guaranteed to terminate as long as a majority of the processes are correct and all messages between correct processes are eventually delivered (and every correct process eventually processes all received messages). It follows that every read operation terminates in two message round-trip delays. The proof of termination of the while loop in the add operation is more involved.

A proposal round in this loop may fail if multiple incomparable values are being concurrently proposed (by different processes). In the worst case, all of these concurrent proposals may fail. However, whenever a proposal by a process fails, a strictly greater value will be proposed by the same process in the next proposal round. As a result, it can be shown that we can have at most n successive proposal rounds before at least one of the processes succeeds in its proposal. Since every add operation begins by broadcasting the value to be added to all other processes, and any process that successfully completes a proposal round is guaranteed to include all values it has received in its next proposal, every add operation is guaranteed to terminate. A careful analysis shows that the complexity of the add operation is O(n) message delays.

5 Impossibility Results

Suppose we have a state machine with two operations op_1 and op_2 such that they do not commute with each other and neither operation nullifies the other operation. Our algorithm from Section 3 does not apply in this case. We now show that if we make somewhat stronger assumptions about op_1 and op_2 no resilient linearizable algorithm is possible for such a state machine.

Consider a state machine with an initial state σ_0. Let op_1 and op_2 be two operations on the state machine. Let σ_i denote the state $op_i(\sigma_0)$ and let $\sigma_{i,j}$ denote the state $op_j(op_i(\sigma_0))$. We say that op_1 and op_2 are *2-distinguishable* in state σ_0 iff $\{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\} \cap \{\sigma_2, \sigma_{2,1}, \sigma_{2,2}\} = \phi$. Note that this essentially says the following: the state produced by execution of op_1, optionally followed by the operation op_1 or op_2, is distinguishable from the state produced by the execution of op_2, optionally followed by the operation op_1 or op_2.

Theorem 2. *A state machine with 2-distinguishable operations op_1 and op_2 in its initial state can be used to solve consensus for 2 processes. Thus, it has a consensus number of at least 2.*

Proof. Assume that we have a resilient linearizable implementation of the given state machine. Reduction 1 shows how we can solve binary consensus for two processes using the state machine implementation. Consider the execution of Reduction 1 by two processes p and q. Since the state machine implementation is resilient, the above algorithm will clearly terminate (unless the executing process fails).

We first show that when neither process fails, both processes will decide on the same value (*agreement*) and that this value must be one of the proposed values (*validity*). Let s_x denote the value read by process x (in line [3]). To establish agreement, we must show that $s_p \in \{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$ iff $s_q \in \{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$.

Reduction 1 2-distinguishable ops in initial state

```
1: procedure Consensus (Boolean b)
2:     if (b) then op1() else op2() endif
3:     s = read()
4:     return(s ∈ {σ1, σ1,1, σ1,2})
```

Reduction 2 k-distinguishable ops in initial state

```
1: procedure Consensus (Boolean b)
2:     if (b) then op1() else op2() endif
3:     s = read()
4:     return(s ∈ Σ1)
```

Let f_x denote the update operation performed by process $x \in \{p, q\}$ (in line [2]). Without loss of generality assume that the update operation f_p executes before f_q (in the linearization order). If f_q executes before the read operation by p, then both processes will read the same value and agreement follows.

Thus, the only non-trivial case (for agreement) is the one where p executes its read operation before q executes its update operation (f_q). Thus, $s_p = f_p(\sigma_0)$ while $s_q = f_q(f_p(\sigma_0))$. Without loss of generality, we can assume that the operation f_p is op_1 (since the other case is symmetric). Operation f_q can, however, be either op_1 or op_2.

Thus, $s_p = \sigma_1$, while s_q is either $\sigma_{1,1}$ or $\sigma_{1,2}$. Hence, agreement holds even in this case.

As for validity: note that this algorithm *decides* on the value proposed by the process that first executes its update operation. Specifically: the value read by either process will belong to $\{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$ iff the first update executed is op_1.

This shows that both validity and agreement holds when both processes are correct. If either of the two processes fails, then agreement is trivially satisfied. Validity holds just as explained above.

We can extend the above result to n processes as follows. Let $\gamma = [e_1, \cdots, e_k]$ be a sequence where each element e_i is either op_1 or op_2. Define $\gamma(\sigma)$ to be $e_k(\cdots(e_1(\sigma))\cdots)$. Define $first(\gamma)$ to be e_1. Let Γ_k denote the set of all non-empty sequences, of length at most k, where each element is either op_1 or op_2.

We say that op_1 and op_2 are k-distinguishable in state σ_0 if for all $\gamma_1, \gamma_2 \in \Gamma_k$, $\gamma_1(\sigma_0) = \gamma_2(\sigma_0)$ implies $first(\gamma_1) = first(\gamma_2)$. In other words, consider two sequences γ_1 and γ_2 in Γ_k such that $first(\gamma_1) \neq first(\gamma_2)$. Then, the final states produced by executing the sequences of operations γ_1 and γ_2 will be different. Loosely speaking, we can say that the effect of the first operation executed has a "memory effect" that lasts for at least $k-1$ more operations.

Define Σ_i to be $\{\gamma(\sigma_0) \mid \gamma \in \Gamma_k, first(\gamma) = op_i\}$, where $i \in \{1, 2\}$. Note that op_1 and op_2 are k-distinguishable in state σ_0 iff Σ_1 and Σ_2 are disjoint.

Theorem 3. *A state machine with k-distinguishable operations op_1 and op_2 in its initial state can be used to solve consensus for k processes. Thus, it has a consensus number of at least k.*

Proof. We use Reduction 2 a generalization of our previous reduction scheme. The proof follows as before.

We now show that the k-distinguishability condition reduces to a simpler non-commutativity property for *idempotent* operations. We say that an operation op is *idempotent* if repeated executions of the operation op have no further effect. We formalize this property as follows. Let γ be a sequence of operations. Define $\gamma!op$ to be the sequence obtained from γ by omitting all occurrences of op except the first one. We say that op is idempotent if: for all sequences γ, $\gamma(\sigma_0) = (\gamma!op)(\sigma_0)$.

Let op_1 and op_2 be two idempotent operations. Then, for any $k \geq 2$, op_1 and op_2 are k-distinguishable in σ_0 iff op_1 and op_2 are 2-distinguishable in σ_0. This condition can be further simplified to: $\{\sigma_1, \sigma_{1,2}\} \cap \{\sigma_2, \sigma_{2,1}\} = \phi$.

Note that the above condition can be equivalently viewed as follows:

1. op_1 and op_2 behave differently in σ_0: $op_1(\sigma_0) \neq op_2(\sigma_0)$.
2. op_1 and op_2 do not commute in σ_0: $op_1(op_2(\sigma_0)) \neq op_1(op_2(\sigma_0))$.
3. op_1 does not nullify op_2 in σ_0: $op_1(op_2(\sigma_0)) \neq op_1(\sigma_0)$.
4. op_2 does not nullify op_1 in σ_0: $op_2(op_1(\sigma_0)) \neq op_2(\sigma_0)$.

Note that the notions of commutativity and nullification used above are with respect to a single initial state.

Note that state machines (or interfaces) in a distributed setting are often designed to be *idempotent* (i.e., all its operations are designed to be idempotent) since a client may need to issue the same operation multiple times (when it does not receive a response back) in the presence of message failures. This may simply require clients to associate a unique identifier to each request they make so that the system can easily identify duplicates of the same request. (Recall that an operation, as defined earlier, includes all the parameters passed to a procedure.)

Reduction 3 2-distinguishable operations in idempotently reachable state

```
1:  procedure Consensus (Boolean b)
2:      f_1(); ···; f_m();
3:      if (b) then op_1() else op_2() endif
4:      s = read()
5:      return(s ∈ {op_1(σ), op_2(op_1(σ))})
```

Theorem 4. *A state machine with 2-distinguishable idempotent operations op_1 and op_2 in its initial state can be used to solve consensus for any number of processes. Thus, it has a consensus number of ∞.*

Extension. The above theorems immediately tell us that resilient linearizable implementations of certain data-structures or state-machines are not possible in an asynchronous model of computation (in the presence of process failures). The above theorem requires 2-distinguishable idempotent operations in the initial state. We can generalize this to state-machines where such operations exist in states other than the initial states.

We say that a state σ is a *reachable* state iff there exists a sequence of operations γ such that $\sigma = \gamma(\sigma_0)$. We say that a state σ is an *idempotently reachable* state iff there exists a sequence of idempotent operations γ such that $\sigma = \gamma(\sigma_0)$.

Theorem 5. *Consider a state machine that has an idempotently reachable state σ and two idempotent operations op_1 and op_2 such that op_1 and op_2 are 2-distinguishable in σ. Then, the given state machine can be used to solve consensus for any number of processes. Thus, it has a consensus number of ∞.*

Proof. Since σ is reachable, there exists a sequence γ_0 of idempotent operations $[f_1, \cdots, f_m]$ such that $[f_1, \cdots, f_m](\sigma_0) = \sigma$. We use the following reduction: The proof follows as before. Let γ_1 denote the sequence of update operations γ_0 followed by op_1. Let γ_2 denote the sequence of update operations γ_0 followed by op_2. Note that every process executes either the sequence γ_1 or γ_2 followed by a read. The idempotence property lets us ignore repeated execution of the same operation. Consider the first process p that executes statement 3. We can show that at this point, the state must be σ (the state produced by the sequence γ_0). Execution of statement 3 by p will produce either state $op_1(\sigma)$ or $op_2(\sigma)$. Suppose p executes op_1 producing state $op_1(\sigma)$. The only subsequent operation that can change the state is op_2, which will produce the state $op_2(op_1(\sigma))$. Thus, the state read in line 4 by any process will belong to the set $\{op_1(\sigma), op_2(op_1(\sigma))\}$. Dually, if p executes op_2, then the state read in line 4 by any process will belong to the set $\{op_2(\sigma), op_1(op_2(\sigma))\}$. It follows that all processes will decide on the same value, depending on the operation p executes.

Idempotent stacks, certain forms of multi-writer registers and many other examples are impossible to realize in a linearizable and resilient manner. Please refer to the Appendix A for details.

Generalization. Our preceding results assume that the state machine includes a read operation that returns the entire state. It is possible to generalize the definitions and proofs to deal with state machines that provide restricted read operations. In particular, the notion of 2-distinguishability requires that resulting states must be distinguishable by some read operation.

6 Applications

Both our positive result (Theorem 1) and negative result (Theorem 5) may be of help in carefully crafting data-structure APIs so as to enable a resilient linearizable implementation. We illustrate this by considering the design of a graph data-structure API.

Graph-1. Let U denote any countable set of vertex identifiers (such as the natural numbers or integers). The *graph* data-structure provides the following update operations:

$$U = \{\ \text{removeVertex}(u) \mid u \in U\ \} \cup \{\ \text{addEdge}(u, v), \text{removeEdge}(u, v) \mid u, v \in U\ \}$$

The state consists of only a set of edges. The formal specification of the operations is shown in Specification 1.

Specification 1 Graph-1
1: Set $\langle U \times U \rangle$ E;
2: addEdge(u, v) { $E = E \cup \{(u, v)\}$ }
3: removeEdge(u, v) { $E = E \setminus \{(u, v)\}$ }
4: removeVertex(u)
5: { $E = \{(x, y) \in E \mid x \neq u, y \neq u\}$ }

Specification 2 Graph-2
1: Set $\langle U \rangle$ V; Set $\langle U \times U \rangle$ E;
2: addEdge(u, v) { $V = V \cup \{u, v\}$; $E = E \cup \{(u, v)\}$ }
3: removeEdge(u, v) { $E = E \setminus \{(u, v)\}$ }
4: removeVertex(u)
5: {$V = V \setminus \{u\}$; $E = \{(x, y) \in E \mid x \neq u, y \neq u\}$ }

Most of the graph operations commute with each other. The only non-commuting operations are discussed below. Operations $\mathrm{addEdge}(u, v)$ and $\mathrm{removeEdge}(u, v)$ nullify each other. Operation $\mathrm{removeVertex}(u)$ nullifies the operations $\mathrm{addEdge}(u, x)$ and $\mathrm{addEdge}(x, u)$ (for any x).

It follows that Graph-1 is a NC state machine and that a resilient linearizable implementation of Graph-1 is possible. However, we now present a very similar specification for a Graph for which no resilient linearizable algorithm is possible.

Graph-2. This specification provides the same set of operations as Graph-1. However, the semantics of the operations are slightly different. The state in this case consists of both a set of vertices V and a set of edges E. The operations maintain the invariant that for any edge $(u, v) \in E$, the vertices u and v are in V (a sort of *referential integrity* constraint). The formal specification is shown in Specification 2. Turns out, Graph-2 is not a NC state machine. The operations $op_1 = \mathrm{addEdge}(u, v)$ and $op_2 = \mathrm{removeVertex}(u)$ neither commute nor nullify in state $G = (\{u, w\}, \{(u, w)\})$ consisting of two vertices u and w and the edge (u, w). It can be verified that the operations op_1 and op_2 are 2-distinguishable in state G. It follows from Theorem 5 that a idempotent version of Graph-2 cannot be realized resiliently.

Discussion. At first glance, it might appear as though the key difference between Graph-1 and Graph-2 is that Graph-2 maintains a set of vertices in addition to a set of edges. Even though Graph-1 does not provide for an explicit representation of the set of vertices, the set of edges implicitly encodes a set of vertices as well (namely the endpoints of the edges in the edge-set). Graph-1 even permits encoding of graphs with isolated vertices (as a self-loop of the form (u, u)). Thus, Graph-1 is, in some sense, as expressive as Graph-2. The key difference between Graph-1 and Graph-2 that leads to the possibility/impossibility distinction above is the subtle change in the semantics of the operations.

7 Related Work

State machine replication is a general approach for implementing data-structures that can tolerate process failures. One common way to implement state machine replication is by using consensus to order all commands If the state machine is deterministic, each correct process is guaranteed to generate the same responses and reach the same state. As consensus is impossible in the presence of process failures [6] this approach does not guarantee progress.

Shapiro *et al.* [9] exploit properties of data structures like commutativity to build efficient replicated data structures. However, they do not seek to achieve linearizability. Many of the implementations they propose are not linearizable.

Faleiro *et al.* [5] show that a weaker form of agreement namely lattice agreement and a generalized version of it (GLA) can be solved in asynchronous message passing systems. They also show that GLA can be used to implement linearizable and resilient UQ state machines as long as all updates commute. This paper shows that even data structures in which not all updates commute can be implemented in a linearizable and resilient manner. In addition we show that certain UQ state machines are impossible to implement in a linearizable and resilient manner.

Wait free implementations of other specific data structures like atomic snapshot objects have been studied in literature [4, 8]. Attiya *et al.* [2] show how a wait free linearizable atomic register for a shared memory system can be emulated in a message passing system so long as only a minority of processes fail.

Our feasibility result is closely related to the result of [1] that wait-free linearizable algorithms are possible in a shared-memory setting for a similar class of problems. The key differences are that we address the problem in a message-passing model. Our approach distinguishes updates from reads, unlike [1]. This also allows us to achieve a more efficient algorithm for read-operations. We also present impossibility results, which are new. In the context of shared memory systems, Vechev et al [3] show that it is impossible to build a linearizable implementation of an object with a non-commutative method without using strong synchronization (barrier, fence or locks)

References

1. J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, SPAA '90, pages 340–349, New York, NY, USA, 1990. ACM.
2. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42, 1995.
3. Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.
4. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distrib. Comput.*, 8, March 1995.
5. 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*, PODC '12, pages 125–134, New York, NY, USA, 2012. ACM.
6. 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.
7. 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.
8. Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 723–732, New York, NY, USA, 2005. ACM.
9. 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.

A Examples

We present several well-known data-structures that are NC state machines as well some which have 2-distinguishable operations. (In the sequel, U and Q denote the sets of update operations and query operations respectively for each example.)

Read-Write Register. A read-write register provides operations to write a value to a register and operations to read the current value of the register. Let V denote the set of storable values. Then, we have $U = \{\ \mathrm{write}(v) \mid v \in V\ \}$ and $Q = \{\ \mathrm{read}()\ \}$. This is in NC. Only writes are update operations and every write operation nullifies every other write operation. (Note that we consider $\mathrm{write}(1)$ and $\mathrm{write}(2)$ to be two different operations. "*write*" by itself is not an operation. Thus, more formally, the operation $\mathrm{write}(i)$ nullifies the operation $\mathrm{write}(j)$ for any values of i and j.) Note also that any total-ordering on U is a NC-ordering.

Read-Write Memory. A read-write memory is essentially a collection of read-write registers. The permitted update operations are write operations that write a new value to a given location. Let L denote the set of locations, and V the set of values. Then, $U = \{\ \mathrm{write}(\ell, v) \mid \ell \in L, v \in V\ \}$ and $Q = \{\ \mathrm{read}(\ell) \mid \ell \in L\ \}$.

In this example, update operations on different memory locations commute with each other, while update operations on the same memory location nullify each other. For each location ℓ, let $\leq_\ell$ denote any total ordering on the set of update operations on ℓ. Then, the union $\cup_{\ell \in L} \leq_\ell$ is a NC-ordering for this state machine.

Atomic Snapshot Object. An atomic snapshot object is the same as a read-write memory as far as update operations are concerned. It differs from a read-write memory in providing a read operation that can return the values of all memory locations (in one operation). Here, $U = \{\ \mathrm{write}(\ell, v) \mid \ell \in L, v \in V\ \}$ and $Q = \{\ \mathrm{read}(\ell) \mid \ell \in L\ \} \cup \{\ \mathrm{snapshot}()\ \}$.

Counter. Consider a Counter that provides update operations to increment and decrement the value of a single counter. Here, $U = \{\ \mathrm{incr}(), \mathrm{decr}()\ \}$ and $Q = \{\ \mathrm{read}()\ \}$. All update operations commute in this case. Thus, the empty relation is a NC-ordering for this state machine.

Resettable Counter. This extends the Counter interface by providing an additional update operation to reset the value of the counter to zero. Here, $U = \{\ \mathrm{incr}(), \mathrm{decr}(), \mathrm{reset}()\ \}$ and $Q = \{\ \mathrm{read}()\ \}$. In this case, all increment and decrement operations commute with each other. The reset operation nullifies all increment and decrement operations. Consider the partial-order

$$\leq = \{\ (\mathrm{incr}(), \mathrm{reset}()), (\mathrm{decr}(), \mathrm{reset}()), (\mathrm{reset}(), \mathrm{reset}())\ \}.$$

$\leq$ is a NC-ordering for this state machine. Note that this data-structure can be generalized to support a write operation that writes a specific new value to the counter. This generalized version is also a NC state machine.

Union-Find. In a *Union-Find* data-structure, every element is initially in an equivalence class by itself. A *union* operation (on two elements) merges the equivalence classes to which the two elements belong into one equivalence class. A *find* operation returns an unique representative element of the equivalence class. We make this specification deterministic by assuming that a total-ordering exists on the set of all elements and that the *find* operation returns the minimum element (with respect to this ordering) of the equivalence class to which the parameter belongs. Let V denote the set of all elements (which is assumed to be statically fixed). Here, $U = \{\ \mathrm{union}(u, v) \mid u, v \in V\ \}$ and $Q = \{\ \mathrm{find}(u) \mid u \in V\ \}$. All the update operations of this data-structure commute. Thus, the empty relation is a NC-ordering for this state machine.

Maps, Sets, and Heaps. Several other well-known data-structures are similar to the preceding examples. A *map* (or key-value store) with update and lookup operations is the same as a read-write memory. A *set* with add, remove, and membership test operations is a special case of the read-write memory where the set of values V is $\{\ true, false\ \}$. A *heap* (or *priority-queue*) with add, remove, and findmin operations is the same as a set with regards to the update operations, and differs only in providing a more complex query operation.

Multi-Register Write. Assume that we have a set of registers (or memory locations) L. In addition to being able to write atomically to any single register, we would like to support the ability to write values to a set of registers atomically. However, rather than permit an atomic write to any arbitrary set of registers, we consider the case where an atomic write is permitted only to certain sets of registers. Let $W \subseteq 2^L$ denote the set of all sets $X \subseteq L$ such that an atomic write to all registers in X is permitted. Whether we can realize this data-structure depends on the set W.

Improper Overlap: We say that two sets X and Y have an *improper overlap* if their intersection is non-empty, but neither set contains the other. If the set W includes two sets X and Y that have an improper overlap, then the update operations that write to these sets of registers are 2-distinguishable. As a result, it is not possible to have a resilient linearizable implementation of such a data-structure.

Proper Nesting: If the set W does not have any elements with an improper overlap, then the corresponding state machine is a NC state machine. Hence, a resilient linearizable implementation of the data structure is possible.

Idempotent UQ Stack. Consider a stack API that exposes operations to push an element onto the stack, pop the top element of the stack (but not return it), and a read operation to return the top element of the stack. Consider the operations op_1 and op_2 that, respectively, push the values 1 and 2 onto the stack. We can show that these operations are 2-distinguishable. Hence, it is not possible to realize a resilient linearizable implementation of this data-structure.

B Proofs

Proof of Lemma 1 Let $f \gtrdot g$ and $g \gtrdot h$. Then, for any state σ,

$$\begin{aligned} f(h(\sigma)) &= f(g(h(\sigma)) && (\text{since } f \gtrdot g) \\ &= f(g(\sigma)) && (\text{since } g \gtrdot h) \\ &= f(\sigma) && (\text{since } f \gtrdot g) \end{aligned}$$

□

Proof of Lemma 2 We show how the desired partial-ordering can be constructed inductively, by considering the update operations one at a time. Let $\leq$ be a NC-ordering on a subset S of the update operations of the given state machine. Let op be an update operation not in S. We define the partial-ordering $\leq'$ on $S \cup \{op\}$ as follows. Let $L = \{\ op' \in S \mid op \gtrdot op'\ \}$. Let $G = \{\ op' \in (S \setminus L) \mid op' \gtrdot op\ \}$. Define $\leq'$ to be $\leq \cup \{(op', op) \mid op' \in L\} \cup \{(op, op)\} \cup \{(op, op') \mid op' \in G\}$. We can show that $\leq'$ is a NC-ordering for $S \cup \{op\}$. It follows that a NC-ordering exists for the set of all update operations. (We note that the ordering can be mathematically defined even if the set of update operations is infinite, but enumerable.) □

Proof of Lemma 5 We prove this by induction. For purposes of induction, we exploit the $\prec_t$ partial-ordering. For any x, define $rank(x)$ to be the cardinality of the set $\{y \mid y \prec_t x\}$. For any triple (x, y, z), define the rank to be $rank(x) + rank(y) + rank(z)$.

Let (a, b, c) be the triple with the smallest rank satisfying $a \prec_n^Y b \prec_n^Y c$ but not $a \prec_{tn}^Y c$. We now derive a contradiction.

We have $a \not\prec_t c$, by assumption. We must have $c \not\prec_t a$, since, otherwise, we would not have $a \prec_n b$ by definition of $\prec_n$. This implies that $a \parallel c$. But we also have $op(a) \prec_s op(b) \prec_s op(c)$. This implies that $op(a) \prec_s op(c)$.

Thus, we have $a \parallel c$ and $op(a) \prec_s op(c)$. This implies that we have $a \prec_n c$, unless there exists some x such that $c \prec_n x \prec_t a$. But such an x cannot exist: otherwise, we have $b \prec_n c \prec_n x$. Since (b, c, x) is a triple with smaller rank than (a, b, c), it must satisfy the lemma (by hypothesis). Hence, we must have $b \prec_{tn} x$. But this contradicts the addition of the $a \prec_n b$ edge. □

Proof of Lemma 6 Consider the graph consisting of Y as the set of vertices, with a directed edge $a \rightarrow b$ (for any pair of vertices a and b) iff $a \prec_{tn} b$. We want to show that this graph has no cycles. If $a \prec_t b$, we refer to $a \rightarrow b$ as a t-edge, and if $a \prec_n b$, we refer to $a \rightarrow b$ as a n-edge.

Part 1: No cycles of 2 or 3 edges. We first show that the graph has no cycles consisting of 3 edges or fewer. It is straightforward to check that the graph cannot have a cycle consisting of only t-edges or a cycle consisting of only n-edges. The definition of the $\prec_n$ relation explicitly avoids creating cycles of the form $a \prec_t b \prec_n a$ or $a \prec_t b \prec_t c \prec_n a$ or $a \prec_t b \prec_n c \prec_n a$. It follows that the graph has no cycles of 3 or fewer edges.

Part 2: No cycles with more than 3 edges. We now show that if the graph has a cycle with $n > 3$ edges, then there must exist another cycle with less than n edges. By repeatedly applying this construction, we can create a cycle with 3 edges or fewer, which leads to a contradiction.

Case 2a. Note that $a \prec_t b \prec_t c$ implies $a \prec_t c$. Hence, if the cycle has 2 or more consecutive t-edges, they can be replaced by a single t-edge, producing a smaller cycle.

Case 2b. If $a \prec_n b \prec_n c$, then we must have $a \prec_{tn} c$ from Lemma 5. Hence, we can replace 2 consecutive n-edges in the cycle by a single edge.

Case 2c. Consider any sequence $a \prec_t b \prec_n c \prec_t d$. In this case, we must have $a \prec_t d$ (since, loosely speaking, $begin(d) > end(c) > begin(b) > end(a)$). Hence, we can replace this sequence by the single edge from a to d.

It follows from the above three cases that we can repeatedly contract any cycle with more than 3 edges until we get a cycle with 3 or fewer edges, which is impossible.

It follows that $\prec_{tn}^Y$ is an acyclic relation. □

Proof of Lemma 8 Follows since the different orderings π_1 and π_2 differ only in the order of concurrent commuting operations. Specifically, let $\pi_1 = a_1 a_2 \cdots a_n$ and let $\pi_2 = b_1 b_2 \cdots b_n$. Let i be the first position where the sequences π_1 and π_2 differ. Thus, we have $a_1 \cdots a_{i-1} = b_1 \cdots b_{i-1}$ and $a_i \neq b_i$.

There must be some $k > i$ such that $a_i = b_k$. For any $i \leq j < k$, we must have $b_j \parallel_{tn} b_k$ since these two commands are ordered differently in π_1 and π_2. Hence, commands b_j and b_k must commute. Hence, the sequences $b_i \cdots b_{k-1} b_k$ and $b_k b_i \cdots b_{k-1}$ must be equivalent. □

Proof of Lemma 9 Consider $a, b \in X$. Whether we add an edge $a \prec_n^X b$ depends only on the set of elements $\{x \in X \mid x \prec_t a \vee x \prec_t b\}$. It follows from Lemma 4 that $\{x \in X \mid x \prec_t a \vee x \prec_t b\} = \{x \in Y \mid x \prec_t a \vee x \prec_t b\}$. Hence, $a \prec_n^X b$ iff $a \prec_n^Y b$. □

Proof of Theorem 1 Resiliency of the algorithm follows from the fact that the distributed set implementation is resilient. Next we prove linearizability. Consider any execution π. Let S_a denote the set of all update commands initiated in π. Let S_c denote the set of all update commands that have completed execution in π. Let S denote the set $\{c \in S_a \mid \exists c' \in S_c.\ c \leq_s c'\}$. Let π_s denote any topological-sort ordering of S with respect to the $\prec_{tn}$ ordering. All the reads can be shown to be consistent with this sequential execution of the update operations.

□