On Liveness of Dynamic Storage

Alexander Spiegelman *
Viterbi EE Department
Technion, Haifa, Israel
sashas@tx.technion.ac.il

Idit Keidar
Viterbi EE Department
Technion, Haifa, Israel
idish@ee.technion.ac.il

Abstract

Dynamic distributed storage algorithms such as DynaStore, Reconfigurable Paxos, RAMBO, and RDS, do not ensure liveness (wait-freedom) in asynchronous runs with infinitely many reconfigurations. We prove that this is inherent for asynchronous dynamic storage algorithms. Our result holds even if only one process may fail, provided that machines that were successfully removed from the system's configuration can be switched off by a system administrator. To circumvent this result, we define a dynamic eventually perfect failure detector, and present an algorithm that uses it to emulate wait-free dynamic atomic storage. Though some of the previous algorithms have been designed for eventually synchronous models, to the best of our knowledge, our algorithm is the first to ensure liveness for all operations without restricting the reconfiguration rate.

*Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

1 Introduction

Many works in the last decade have dealt with *dynamic* reliable distributed storage emulation [26, 2, 14, 15, 9, 7, 21, 8, 18, 6, 5, 13, 17]. The motivation behind such storage is to allow new processes (nodes) to be phased in and old or dysfunctional ones to be taken offline. From a fault-tolerance point of view, once a faulty process is removed, additional failures may be tolerated. For example, consider a system that can tolerate one failure: once a process fails, no additional processes are allowed to fail. However, once the faulty process is replaced by a correct one, the system can again tolerate one failure. Thus, while static systems become permanently unavailable after some constant number of failures, dynamic systems that allow infinitely many reconfigurations can survive forever.

Previous works can be categorized into two main types: Solutions of the first type assume a churn-based model [19, 22, 25] in which processes are free to announce when they join the storage emulation [7, 5, 6, 4] via an auxiliary broadcast sub-system that allows a process to send a message to all the processes in the system, (which my be unknown to the sending processes). The second type solutions extend the register's API with a reconfiguration operation for changing the current configuration of participating processes [26, 2, 18, 13, 14, 15, 9], which can be only invoked by members of the current configuration. In this paper we consider the latter. Such an API allows administrators (running privileged processes), to remove old or faulty processes and add new ones without shutting down the service; once a process is removed from the current configuration, a system administrator may shut it down. Note that in the churn-based model, in contrast, if processes have to perform an explicit operation in order to leave the system (as in [7, 4]), a faulty process can never be removed. In addition, since in API-based models only processes that are already within the system invoke operations, it is possible to keep track of the processes in the system, and thus auxiliary broadcast is not required.

Though the literature is abundant with dynamic storage algorithms in both models, to the best of our knowledge, all previous solutions in asynchronous and eventually synchronous models restrict reconfigurations in some way in order to ensure completion of all operations. Churn-based solutions assume a bounded churn rate [7, 5, 4], meaning that there is a finite number of joining and removing processes in a given time interval. Some of the API-based solutions [26, 2, 18, 13] provide liveness only when the number of reconfigurations is finite, whereas others discuss liveness only in synchronous runs [14, 15, 9]. Such restrictions may be problematic in emerging highly-dynamic large-scale settings.

Baldoni et al. [5] showed that it is impossible to emulate a dynamic register that ensures completion of all operations without restricting the churn rate in asynchronous churn-based models in which processes can freely abandon the computation without an explicit leave operation. Since a leave and a failure are indistinguishable in such models, the impossibility can be proven using a partition argument as in [3].

In this paper we revisit this question in the API-based model. First, we prove a similar result for asynchronous API-based dynamic models, in which *one* unremoved process can fail and successfully removed ones can go offline. Specifically, we show that even the weakest type of storage, namely a *safe* register [20], cannot be implemented so as to guarantee liveness for all operations (i.e., wait-freedom) in asynchronous runs with an unrestricted reconfiguration rate. Note that this bound does not follow from the one in [5] since a process in our model can leave the system only after an operation that removes it successfully completes.

Second, to circumvent our impossibility result, we define a dynamic failure detector that can be easily implemented in eventually synchronous systems, and use it to implement dynamic storage. We present an algorithm, based on state machine replication, that emulates a strong shared object, namely a wait-free atomic dynamic multi-writer, multi-reader (MWMR) register, and ensures liveness for all operations without restricting the reconfiguration rate. Though a number of previous algorithms have been designed for eventually synchronous models [14, 15, 9, 7, 5, 21, 8], to the best of our knowledge, our algorithm is the first to ensure liveness of all operations without restricting the reconfigurations rate.

In particular, previous algorithms [14, 15, 9, 21, 8] that used failure detectors, only did so for reaching consensus on the new configuration. For example, reconfigurable Paxos variants [21, 8],

which implement atomic storage via dynamic state machine replication, assume a failure detector that provides a leader in every configuration. However, a configuration may be changed, allowing the previous leader to be removed (and then fail) before another process p (with a pending operation) is able to communicate with it in the old configuration. Though a new leader is elected by the failure detector in the ensuing configuration, this scenario may repeat itself indefinitely, so that p's pending operation never completes.

We, in contrast, use the failure detector also to implement a helping mechanism, which ensures that eventually some process will help a slow one before completing its own reconfiguration operation even if the reconfiguration rate is unbounded. Such mechanism is attainable in API-based models since only members of the current configuration invoke operations, and thus helping process can know which processes may need help. Note that in churn-based models in which processes announce their own join, implementing such a helping mechanism is impossible, since a helping process cannot possibly know which processes need help joining.

The remainder of this paper is organized as follows: In Section 2 we present the model and define the dynamic storage object we seek to implement. Our impossibility proof appears in Section 3, and our algorithm in Section 4. Finally, we conclude the paper in Section 5.

2 Model and Dynamic Storage Problem Definition

In Section 2.1, we present the preliminaries of our model, and in Section 2.2, we define the dynamic storage service.

2.1 Preliminaries

We consider an asynchronous message passing system consisting of an infinite set of processes Π. Processes may fail by crashing subject to restrictions given below. Process failure is modeled via an explicit fail action. Each pair of processes is connected by a communication link. A *service* exposes a set of *operations*. For example, a dynamic storage service exposes read, write, and reconfig operations. Operations are invoked and subsequently respond.

An *algorithm* A defines the behaviors of processes as deterministic state machines, where state transitions are associated with *actions*, such as send/receive messages, operation invoke/response, and process failures. A *global state* is a mapping to states from system components, i.e., processes and links. An *initial global state* is one where all processes are in initial states and all links are empty. A send action is *enabled* in state s if A has a transition from s in which the send occurs.

A *run* of algorithm A is a (finite or infinite) alternating sequence of global states and actions, beginning with some initial global state, such that state transitions occur according to A. We use the notion of *time t during a run r* to refer to the t^{th} action in r and the global state that ensues it. A *run fragment* is a contiguous subsequence of a run. An operation invoked before time t in run r is *complete* at time t if its response event occurs before time t in r; otherwise it is *pending* at time t. We assume that runs are *well-formed* [16], in that each process's first action is an invocation of some operation, and a process does not invoke an operation before receiving a response to its last invoked one.

We say that operation op_i *precedes* operation op_j in a run r, if op_i's response occurs before op_j's invocation in r. Operations op_i and op_j are *concurrent* in run r, if op_i does not precede op_j and op_j does not precede op_i in r. A *sequential run* is one with no concurrent operations. Two runs are *equivalent* if every process performs the same sequence of operations (with the same return values) in both, where operations that are pending in one can either be included in or excluded from the other.

2.2 Dynamic storage

The distributed storage service we consider is a *dynamic multi-writer, multi reader (MWMR) register* [2, 18, 13, 27, 24, 15], which stores a value v from a domain $\mathbb{V}$, and offers an interface for invoking *read*, *write*, and *reconfig* operations. Initially, the register holds some initial value $v_0 \in \mathbb{V}$. A *read* operation takes no parameters and returns a value from $\mathbb{V}$, and a *write* operation takes a value from $\mathbb{V}$ and returns "ok". We define *Changes* to be the set $\{remove, add\} \times \Pi$,

and call any subset of Changes a *set of changes*. For example, $\{\langle add, p_3\rangle, \langle remove, p_2\rangle\}$ is a set of changes. A *reconfig* operation takes as a parameter a set of changes and returns "ok". For simplicity, we assume that a process that has been removed is not added again.

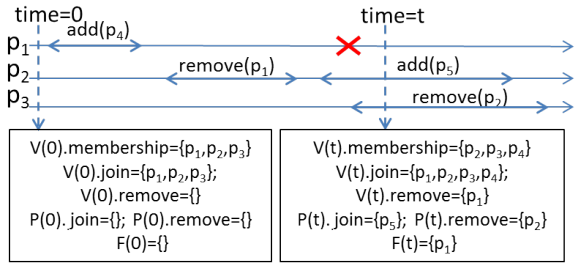

Figure 1: Notation illustration. $add(p)$ ($remove(p)$) represents $reconfig(\langle add, p\rangle)$ (respectively, $reconfig(\langle remove, p\rangle)$).

Notation For every subset w of $Changes$, the *removal set* of w, denoted $w.remove$, is $\{p_i | \langle remove, p_i\rangle \in w\}$; the *join set* of w, denoted $w.join$, is $\{p_i | \langle add, p_i\rangle \in w\}$; and the *membership* of w, denoted $w.membership$, is $w.join \setminus w.remove$. For example, for a set $w = \{\langle add, p_1\rangle, \langle remove, p_1\rangle, \langle add, p_2\rangle\}$, $w.join = \{p_1, p_2\}$, $w.remove = \{p_1\}$, and $w.membership = \{p_2\}$. For a time t in a run r, we denote by $V(t)$ the union of all sets q s.t. *reconfig(q)* completes before time t in r. A *configuration* is a finite set of processes, and the *current configuration at time* t is $V(t).membership$. We assume that only processes in $V(t).membership$ invoke operations at time t. The initial set of processes $\Pi_0 \subset \Pi$ is known to all and we say, by convention, that $reconfig(\{\langle add, p\rangle | p \in \Pi_0\})$ completes at time 0, i.e., $V(0).membership = \Pi_0$.

We define $P(t)$ to be the set of *pending changes* at time t in run r, i.e., the set of all changes included in pending reconfig operations. We denote by $F(t)$ the set of processes that have failed before time t in r; initially, $F(0) = \{\}$. For a series of arbitrary sets $S(t)$, $t \in \mathbb{N}$, we define $S(*) \triangleq \bigcup_{t\in\mathbb{N}} S(t)$. The notation is illustrated in Figure 1.

Correct processes and fairness A process p is *correct* if $p \in V(*).join \setminus F(*)$. A run r is *fair* if every send action by a correct process that is enabled infinitely often eventually occurs, and every message sent by a correct process p_i to a correct process p_j is eventually received at p_j. Note that messages sent to a faulty process from a correct one may or may not be received. A process p is *active* if p is correct, and $p \notin P(*).remove$.

Service specification A *linearization* of a run r is an equivalent sequential run that preserves r's operation precedence relation and the service's sequential specification. The sequential specification for a register is as follows: A read returns the latest written value, or v_0 if none was written. An MWMR register is *atomic*, also called *linearizable* [16], if every run has a linearization. Lamport [20] defines a *safe* single-writer register. Here, we generalize the definition to multi-writer registers in a weak way in order to strengthen the impossibility result. Intuitively, if a read is not concurrent with any write we require it to return a value that reflects some possible outcome of the writes that precede it; otherwise we allow it to return an arbitrary value. Formally: An MWMR register is *safe* if for every run r for every *read* operation rd that has no concurrent *writes* in r, there is a linearization of the subsequence of r consisting of rd and the *write* operations in r.

A *wait-free* service guarantees that every active process's operation completes regardless of the actions of other processes.

Failure model and reconfiguration The reconfig operations determine which processes are allowed to fail at any given time. Static storage algorithms [3] tolerate failures of a minority of their (static) universe. At a time t when no reconfig operations are ongoing, the dynamic failure condition may be simply defined to allow less than $|V(t)membership|/2$ failures of processes in $V(t).membership$. When there are pending additions and removals, the rule must be generalized to take them into account. For our algorithm in Section 4, we adopt a generalization presented in previous works [2, 18, 27, 1]:

Definition 1 (minority failures)**.** A model allows *minority failures* if at all times t in r, fewer than $|V(t).membership \setminus P(t).remove|/2$ processes out of $V(t).membership \cup P(t).join$ are in $F(t)$.

Note that this failure condition allows processes whose remove operations have completed to be (immediately) safely switched off as it only restricts failures out of the current membership and pending joins. We say that a service is *reconfigurable* if failures of processes in $V(t).remove$ are unrestricted.

In order to strengthen our lower bound in Section 3 we weaken the failure model. Like FLP [12], our lower bound applies as long as at least *one* process can fail. Formally, a failure is allowed whenever all failed processes have been removed and the current membership consists of at least three processes[1]. We call such a state "clean", captured by the following predicate: $clean(t) \triangleq (V(t).membership \cup P(t).join) \cap F(t) = \{\} \wedge |V(t).membership \setminus P(t).remove| \geq 3$. The minimal failure condition is thus defined as follows:

Definition 2 (minimal failure)**.** A model allows *minimal failure* if in every run r ending at time t when $clean(t)$, for every process $p \in V(t).membership \cup P(t)$, there is an extension of r where p fails at time $t+1$.

Notice that the minority failure condition allows minimal failure, and so all algorithms that assume minority failures [2, 18, 27, 1] are a fortiori subject to our lower bound, which is proven for minimal failures.

3 Impossibility of Wait-Free Dynamic Safe Storage

In this section we prove that there is no implementation of wait-free dynamic safe storage in a model that allows minimal failures. We construct a fair run with infinitely many reconfiguration operations in which a slow process p never completes its write operation. We do so by delaying all of p's messages. A message from p to a process p_i is delayed until p_i is removed, and we make sure that all processes except p are eventually removed and replaced.

Theorem 1. *There is no algorithm that emulates wait-free dynamic safe storage in an asynchronous system allowing minimal failures.*

Proof (Theorem 1). Assume by contradiction that such an algorithm A exists. We prove two lemmas about A.

Lemma 1.1. *Consider a run r of A ending at time t s.t. $clean(t)$, and two processes $p_i, p_j \in V(t).membership$. Extend r by having p_j invoke operation op at time $t+1$. Then there exists an extension of r where (1) op completes at some time $t' > t$,(2) no process receives a message from p_i between t and t', and (3) no process fails and no operations are invoked between t and t'.*

Proof (Lemma 1.1). By the minimal failure condition, p_i can fail at time $t+2$. Consider a fair extension σ_1 of r, in which p_i fails at time $t+2$ and all of its in-transit messages are lost, no other process fails, and no operations are invoked. By wait-freedom, op eventually completes at some time t_1 in σ_1. Since p_i fails and all its outstanding messages are lost, then from time t to t_1 in σ_1 no process receives any messages from p_i. Now let σ_2 be identical to σ_1 except that p_i does not fail, but all of its messages are delayed. Note that σ_1 and σ_2 are indistinguishable to all processes except p_i. Thus, op returns at time t_1 also in σ_2.

□ *Lemma 1.1*

Lemma 1.2. *Consider a run r of A ending at time t s.t $clean(t)$. Let $v_1 \in \mathbb{V} \setminus \{v_0\}$ be a value s.t. no process invokes $write(v_1)$ in r. If we extend r fairly so that p_i invokes $w = write(v_1)$ at time $t+1$ which completes at some time $t_1 > t+1$ s.t. $clean(t')$ for all $t < t' \leq t_1$ then in the run fragment between $t+1$ and t_1, some process $p_k \neq p_i$ receives a message sent by p_i.*

[1] Note that with fewer than three processes, even static systems cannot tolerate failures [3].

Proof (Lemma 1.2). Assume by way of contradiction that in the run fragment between $t+1$ and t_1 no process $p_k \neq p_i$ receives a message sent by p_i.

Consider a run r' that is identical to r until time t_1 except that p_i does not invoke w at time t. Now assume that some process $p_j \neq p_i$ invokes a *read* operation rd at time t_1+1 in r'. By the assumption, $clean(t_1)$ and therefore $clean(t_1+1)$. Thus, by Lemma 1.1, there is a run fragment σ beginning at the final state of r' (time t_1+1), where rd completes at some time t_2, s.t. between t_1+1 and t_2 no process receives a message from p_i. Since no process invokes $write(v_1)$ in r', and no writes are concurrent with the read, by safety, rd returns some $v_2 \neq v_1$.

Now notice that all global states from time t to time t_1 in r and r' are indistinguishable to all processes except p_i. Thus, we can continue run r with an invocation of read operation rd' by p_j at time t_1, and append σ to it. Operation rd' hence completes and returns v_2. A contradiction to safety.

□ *Lemma 1.2*

To prove the theorem, we construct an infinite fair run r in which a *write* operation of an active process never completes, in contradiction to wait-freedom.

Consider some initial global state c_0, s.t. $P(0) = F(0) = \{\}$ and $V(0).membership = \{p_1 \dots p_n\}$, where $n \geq 3$. An illustration of the run for $n = 4$ is presented in Figure 2. Now, let process p_1 invoke a write operation w at time $t_1 = 0$, and do the following:

Let process p_n invoke *reconfig(q)* where $q = \{\langle add, p_j\rangle | n+1 \leq j \leq 2n-2\}$ at time t_1. The state at the end of r is clean (i.e., $clean(t_1)$). So by Lemma 1.1, we can extend r with a run fragment σ_1 ending at some time t_2 when *reconfig(q)* completes, where no process $p_j \neq p_1$ receives a message from p_1 in σ_1, no other operations are invoked, and no process fails.

Then, at time t_2+1, p_n invokes *reconfig(q')*, where $q' = \{\langle remove, p_j\rangle | 2 \leq j \leq n-1\}$. Again, the state is clean and thus by Lemma 1.1 again, we can extend r with a run fragment σ_2 ending at some time t_3 when *reconfig(q')* completes s.t. no process $p_j \neq p_1$ receives a message from p_1 in σ_2, no other operations are invoked, and no process fails.

Recall that the minimal failures condition satisfies reconfigurability, i.e., all the processes in $V(t_3).remove$ can be in $F(t_3)$ (fail). Let the processes in $\{p_j \mid 2 \leq j \leq n-1\}$ fail at time t_3, and notice that the fairness condition does not mandate that they receive messages from p_1. Next, allow p_1 to perform all its enabled actions till some time t_4.

Now notice that at t_4, $|V(t_4).membership| = n$, $P(t_4) = \{\}$, $(V(t_4).membership \cup P(t_4).join) \cap F(t_4) = \{\}$, and $|V(t_4).membership \setminus P(t_4).removal| \geq 3$. We can rename the processes in $V(t_4).membership$ (except p_1) so that the process that performed the remove and add operations becomes p_2, and all others get names in the range $p_3 \dots p_n$. We can then repeat the construction above. By doing so infinitely many times, we get an infinite run r in which p_1 is active and no process ever receives a message from p_1. However, all of p_1's enabled actions eventually occur. Since no process except p_1 is correct in r, the run is fair. In addition, since $clean(t)$ for all t in r, by the contrapositive of Lemma 1.2, w does not complete in r, and we get a violation of wait-freedom.

■ *Theorem 1*

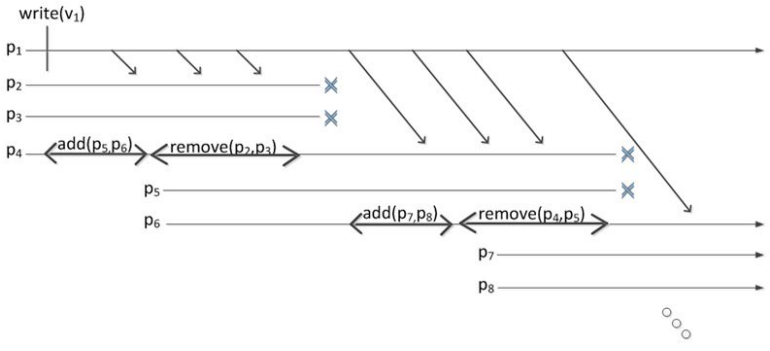

Figure 2: Illustration of the infinite run for $n = 4$.

4 Oracle-Based Dynamic Atomic Storage

We present an algorithm that circumvents the impossibility result of Section 3 using a failure detector. In this section we assume the minority failure condition. In Section 4.1, we define a dynamic eventually perfect failure detector. In Section 4.2, we describe an algorithm, based on dynamic state machine replication, that uses the failure detector to implement a wait-free dynamic atomic MWMR register. The algorithm's correctness is proven in Appendix A.

4.1 Dynamic failure detector

Since the set of processes is potentially infinite, we cannot have the failure detector report the status of all processes as static failure detectors typically do. Dynamic failure detectors addressing this issue have been defined in previous works, either providing a set of processes that have been excluded from or included into the group [23], or assuming that there is eventually a fixed set of participating processes [10]. In our model, we do not assume that there is eventually a fixed set of participating processes, as the number of reconfig operations can be infinite. And we do not want the failure detector to answer with a list of processes, because in dynamic systems, this gives additional information about participating processes that could have been unknown to the inquiring process, and thus it is not clear how such a failure detector can be implemented.

Instead, our dynamic failure detector is queried separately about each process. For each query, it answers either *fail* or *ok*. It can be wrong for an unbounded period, but for each process, it eventually returns a correct answer. Formally, a *dynamic eventually perfect* failure detector, $\diamond P^D$, satisfies two properties:

- **Strong completeness:** For each process p_i that fails at time t_i, there is a time $t > t_i$ s.t. the failure detector answers *fail* to every query about p_i after time t.
- **Eventual strong accuracy:** There exists a time t, called the *stabilization time*, s.t. the failure detector answers *ok* to every query at any time $t' > t$ about a correct process in $V(t').join$.

Note that $\diamond P^D$ can be implemented in a standard way in the eventually (partially) synchronous model by pinging the queried process and waiting for a response until a timeout.

4.2 Dynamic storage algorithm

In Section 4.2.1 we give the overview of our algorithm and in Section 4.2.2 we present the full description.

4.2.1 Algorithm overview

The key to achieving liveness with unbounded reconfig operations is a novel helping mechanism, which is based on our failure detector. Intuitively, the idea is that every process tries to help all other processes it believes are correct, (according to its failure detector), to complete their concurrent operations together with its own. At the beginning of an operation, a process p queries all other processes it knows about for the operations they currently perform. The failure detector is needed in order to make sure that (1) p does not wait forever for a reply from a faulty process (achieved by strong completeness), and (2) every slow correct process eventually gets help (achieved by eventual strong accuracy).

State machine emulation of a register We use a state machine *sm* to emulate a wait-free atomic dynamic register, *DynaReg*. Every process has a local replica of *sm*, and we use consensus to agree on *sm*'s state transitions. Notice that each process is equipped with a failure detector FD of class $\diamond P^D$, so consensus is solvable under the assumption of a correct majority in a given configuration [21].

Each instance of consensus runs in some static configuration c and is associated with a unique timestamp. A process participates in a consensus instance by invoking a *propose* operation with

the appropriate configuration and timestamp, as well as its proposed decision value. Consensus then responds with a *decide* event, so that the following properties are satisfied: *Uniform Agreement* – every two decisions are the same. *Validity* – every decision was previously proposed by one of the processes in c. *Termination* – if a majority of c is correct, then eventually every correct process in c decides. We further assume that a consensus instance does not decide until a majority of the members of the configuration propose in it.

The *sm* (lines 2-5 in Algorithm 1) keeps track of dynaReg's value in a variable *val*, and the configuration in a variable *cng*, containing both a list of processes, *cng.mem*, and a set of removed processes, *cng.rem*. Write operations change *val*, and reconfig operations change *cng*. A consensus decision may bundle a number of operations to execute as a single state transition of *sm*. The number of state transitions executed by *sm* is stored in the variable *ts*. Finally, the array *lastOps* maps every process p in *cng.mem* to the sequence number (based on p's local count) of p's last operation that was performed on the emulated DynaReg together with its result.

Each process partakes in at most one consensus at a time; this consensus is associated with timestamp *sm.ts* and runs in *sm.cng.mem*. In every consensus, up to $|sm.cng.mem|$ ordered operations on the emulated DynaReg are agreed upon, and *sm*'s state changes according to the agreed operations. A process's *sm* may change either when consensus decides or when the process receives a newer *sm* from another process, in which case it skips forward. So *sm* goes through the same states in all the processes, except when skipping forward. Thus, for every two processes p_k, p_l, if $sm_k.ts = sm_l.ts$, then $sm_k = sm_l$. (A subscript i indicates the variable is of process p_i.)

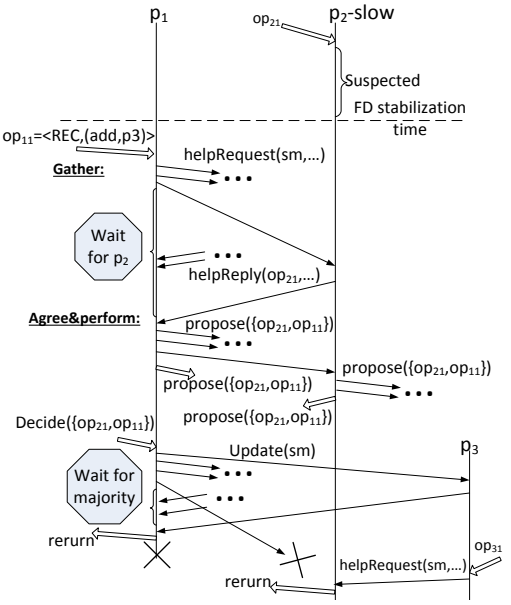

Figure 3: Flow illustration: process p_2 is slow. After stabilization time, process p_1 helps it by proposing its operation. Once p_2's operation is decided, it is reflected in every up-to-date *sm*. Therefore, even if p_1 fails before informing p_2, p_2 receives from the next process that performs an operation, namely p_3, an *sm* that reflects its operation, and thus returns. Line arrows represent messages, and block arrows represent operation or consensus invocations and responses.

Helping The problematic scenario in the impossibility proof of Section 3 occurs because of endless reconfig operations, where a slow process is never able to communicate with members of its configuration before they are removed. In order to circumvent this problem, we use FD to implement a helping mechanism. When proposing an operation, process p_i tries to help other processes in two ways: first, it helps them complete operations they may have successfully proposed in previous rounds (consensuses) but have not learned about their outcomes; and second, it proposes their new operations. To achieve the first, it sends a helping request with its *sm* to all other processes in $sm_i.cng.mem$. For the second, it waits for each process to reply with a help reply containing its latest invoked operation, and then proposes all the operations together. Processes may fail or be removed, so p_i cannot wait for answers forever. To this end, we use FD. For every process in $sm_i.cng.mem$ that has not been removed, p_i repeatedly inquires FD and waits either for a reply from the process or for an answer from FD that the process has failed. Notice that the strong completeness property guarantees that p_i will eventually continue, and strong accuracy guarantees that every slow active process will eventually receive help in case of endless reconfig operations.

Nevertheless, if the number of reconfig operations is finite, it may be the case that some slow process is not familiar with any of the correct members in the current configuration, and no other process performs an operation (hence, no process is helping). To ensure progress in such cases, every correct process periodically sends its *sm* to all processes in its *sm.cng.mem*.

State survival Before the reconfig operation can complete, the new sm needs to propagate to a majority of the new configuration, in order to ensure its survival. Therefore, after executing the state transition, p_i sends sm_i to $sm_i.cng$ members and waits until it either receives acknowledgements from a majority or learns of a newer sm. Notice that in the latter case, consensus in $sm_i.cng.mem$ has decided, meaning that at least a majority of $sm_i.cng.mem$ has participated in it, and so have learned of it.

Flow example The algorithm flow is illustrated in Figure 3. In this example, a slow process p_2 invokes operation op_{21} before FD's stabilization time, ST. Process p_1 invokes operation $op_{11} = \langle add, p_3 \rangle$ after ST. It first sends *helpRequest* to p_2 and waits for it to reply with *helpReply*. Then it proposes op_{21} and op_{11} in a consensus. When *decide* occurs, p_1 updates its sm, sends it to all processes, and waits for majority. Then op_{11} returns and p_1 fails before p_2 receives its update message. Next, p_3 invokes a *reconfig* operation, but this time when p_2 receives *helpRequest* with the up-to-date sm from p_3, it notices that its operation has been performed, and op_{21} returns.

4.2.2 Detailed description

The data structure of process p_i is given in Algorithm 1. The type Ops defines the representation of operations. The emulated state machine, sm_i, is described above. Integer $opNum_i$ holds the sequence number of p_i's current operation; ops_i is a set that contains operations that need to be completed for helping; the flag $pend_i$ is a boolean that indicates whether or not p_i is participating in an ongoing consensus; and $myOp_i$ is the latest operation invoked at p_i.

Algorithm 1 Data structure of process p_i

1: $Ops \triangleq \{\langle RD, \perp \rangle\} \cup \{\langle WR, v \rangle \mid v \in \mathbb{V}\} \cup \{\langle REC, c \rangle \mid c \subset Changes\}$
2: $sm_i.ts \in \mathbb{N}$, initially 0
3: $sm_i.value \in \mathbb{V}$, initially v_0
4: $sm_i.cng = \langle mem, rem \rangle$, where $mem, rem \subset \Pi$, initially $\langle \Pi_0, \{\} \rangle$
5: $sm_i.lastOps$ is a vector of size $|sm_i.cng.mem|$, where $\forall p_j \in sm_i.cng.mem$, $sm_i.lastOps[j] = \langle num, res \rangle$, where $num \in \mathbb{N}$, $res \in \mathbb{V} \cup \{\text{“ok”}\}$, initially $\langle 0,$ “ok” $\rangle$
6: $pend_i \in \{true, false\}$, initially $false$
7: $opNum_i \in \mathbb{N}$, initially 0
8: $ops_i \subset \Pi \times Ops \times \mathbb{N}$, initially {}
9: $myOp_i \in operation$, initially $\perp$

The algorithm of process p_i is presented in Algorithms 2 and 3. We execute every event handler, (operation invocation, message receiving, and consensus decision), atomically excluding wait instructions; that is, other event handlers may run after the handler completes or during a wait (lines 16,18,27 in Algorithm 2). The algorithm runs in two phases. The first, *gather*, is described in Algorithm 2 lines 11–16 and in Algorithm 3 lines 52–58. Process p_i first increases its operation number $opNum_i$, writes op together with $opNum_i$ to the set of operations ops_i, and sets $myOp_i$ to be op. Then it sends $\langle$“helpRequest”$, \ldots\rangle$ to every member of $A = sm_i.cng.mem$ (line 15), and waits for each process in A that is not suspected by the FD or removed to reply with $\langle$“helpReply”$, \ldots\rangle$. Notice that sm_i may change during the wait because messages are handled, and p_i may learn of processes that have been removed.

When $\langle$“helpRequest”$, num, sm\rangle$ is received by process $p_j \neq p_i$, if the received sm is newer than sm_j, then process p_j adopts sm and abandons any previous consensus. Either way, p_j sends $\langle$“helpReply”$, \ldots\rangle$ with its current operation $myOp_j$ in return.

Upon receiving $\langle$“helpReply”$, opNum_i, op, num\rangle$ that corresponds to the current operation number $opNum_i$, process p_i adds the received operation op, its number num, and the identity of the sender to the set ops_i.

At the end of this phase, process p_i holds a set of operations, including its own, that it tries to agree on in the second phase (the order among this set is chosen deterministically, as explained below). Note that p_i can participate in at most one consensus per timestamp, and its propose might end up not being the decided one, in which case it may need to propose the same

operations again. Process p_i completes *op* when it discovers that *op* has been performed in sm_i, whether by itself or by another process.

The second phase appears in Algorithm 2 lines 17–28, and in Algorithm 3 lines 31–51. In line 17, p_i checks if its operation has not been completed yet. In line 18, it waits until it does not participate in any ongoing consensus ($pend_i$=*false*) or some other process helps it complete *op*. Recall that during a wait, other events can be handled. So if a message with an up-to-date sm is received during the wait, p_i adopts the sm. In case *op* has been completed in sm, p_i exits the main while (line 19). Otherwise, p_i waits until it does not participate in any ongoing consensus. This can be the case if (1) p_i has not proposed yet, (2) a message with a newer sm was received and a previous consensus was subsequently abandoned, or (3) a *decide* event has been handled. In all cases, p_i marks that it now participates in consensus in line 20, prepares a new request Req with the operations in ops_i that have not been performed yet in sm_i in line 27, proposes Req in the consensus associated with $sm_i.ts$, and sends $\langle$"propose",$\ldots\rangle$ to all the members of $sm_i.cng.mem$.

Algorithm 2 Process p_i's algorithm: performing operations

```
10: upon invoke operation(op) do
                                                          ▷ phase 1: gather
11:     opNum_i ← opNum_i + 1
12:     ops_i ← {⟨p_i, op, opNum_i⟩}
13:     myOp_i ← op
14:     A ← sm_i.cng.mem
15:     for all p ∈ A
            send ⟨"helpRequest", opNum_i, sm_i⟩ to p
16:     for all p ∈ A wait for ⟨"helpReply", opNum_i, ...⟩ from p or p is suspected or p ∈ sm_i.cng.rem
                                                          ▷ phase 2: agree&perform
17:     while sm_i.lastOps[i].num ≠ opNum_i
18:         wait until ¬pend_i or sm_i.lastOps[i].num = opNum_i
19:         if sm_i.lastOps[i].num = opNum_i then break
20:         pend_i ← true
21:         Req ← {⟨p_j, op, num⟩ ∈ ops_i | num > sm_i.lastOps[j].num}
22:         propose(sm_i.cng, sm_i.ts, Req)
23:         for all p ∈ sm_i.cng.mem send ⟨"propose", sm_i, Req⟩ to p
24:     if op.type = REC
25:         ts ← sm_i.ts
26:         for all p ∈ sm_i.cng.mem send ⟨"update", sm_i, opNum_i⟩ to p
27:         wait for ⟨"ACK", opNum_i⟩ from majority of sm_i.cng.mem or sm_i.ts > ts
28:     return sm_i.lastOps[i].res

29: periodically:
30:     for all p ∈ sm_i.cng.mem send ⟨"update", sm_i, ⊥⟩ to p
```

When $\langle$"propose",$sm, Req\ldots\rangle$ is received by process $p_j \neq p_i$, if the received sm is more updated than sm_j, then process p_j adopts sm, abandons any previous consensus, proposes Req in the consensus associated with $sm.ts$, and forwards the message to all other members of $sm_j.cng.mem$. The same is done if sm is identical to sm_j and p_j has not proposed yet in the consensus associated with $sm_j.ts$. Otherwise, p_j ignores the message.

The event $decide_i(sm.cng, sm_i.ts, Req)$ indicates a decision in the consensus associated with $sm_i.ts$. When this occurs, p_i performs all the operations in Req and changes sm_i's state. It sets the value of the emulated DynaReg, $sm_i.value$, to be the value of the *write* operation of the process with the lowest id, and updates $sm_i.cng$ according to the *reconfig* operations. In addition, for every $\langle p_j, op, num\rangle \in Req$, p_i writes to $sm_i.lastOps[j]$, num and *op*'s response, which is "ok" in case of a *write* or a *reconfig*, and $sm_i.value$ in case of a *read*. Next, p_i increases $sm_i.ts$ and sets $pend_i$ to false, indicating that it no longer participates in any ongoing consensus.

Finally, after *op* is performed, p_i exits the main while. If *op* is not a *reconfig* operation, then p_i returns the result, which is stored in $sm_i.lastOps[i].res$. Otherwise, before returning, p_i has to be sure that a majority of $sm_i.cng.mem$ receives sm_i. It sends $\langle$"update",$sm,\ldots\rangle$ to all the processes in $sm_i.cng.mem$ and waits for $\langle$"ACK",$\ldots\rangle$ from a majority of them. Notice that it may be the case that there is no such correct majority due to later reconfig operations and

failures, so, p_i stops waiting when a more updated sm is received, which implies that a majority of $sm_i.cng.mem$ has already received sm_i (since a majority is needed in order to solve consensus).

Upon receiving $\langle$"update", $sm, num\rangle$ with a new sm from process p_i, process p_j adopts sm and abandons any previous consensus. In addition, if $num \neq \perp$, p_j sends $\langle$"ACK", $num\rangle$ to p_i (Algorithm 3 lines 59–63).

Beyond handling operations, in order to ensure progress in case no operations are invoked from some point on, every correct process periodically sends $\langle$"update", $sm, \perp\rangle$ to all processes in its $sm.cng.mem$ (Algorithm 2 line 30).

Algorithm 3 Process p_i's algorithm: event handlers

```
31: upon decide_i(sm_i.cng, sm_i.ts, Req) do
32:     W ← {⟨p, value⟩ | ⟨p, ⟨WR, value⟩, num⟩ ∈ Req}
33:     if W ≠ {}                                   ▷ deterministically choose one of the writes to be the last
34:         sm_i.value ← value with smallest p in W
35:     for all ⟨p_j, op, num⟩ ∈ Req                                              ▷ apply op to sm
36:         if op.type = WR
37:             sm_i.lastOps[j] ← ⟨num, "ok"⟩
38:         else if op.type = RD
39:             sm_i.lastOps[j] ← ⟨num, sm_i.value⟩
40:         else
41:             sm_i.cng.rem ← sm_i.cng.rem ∪ {p | ⟨remove, p⟩ ∈ op.changes}
42:             sm_i.cng.mem ← sm_i.cng.mem ∪ {p | ⟨add, p⟩ ∈ op.changes} \ sm_i.cng.rem
43:             sm_i.lastOps[j] ← ⟨num, "ok"⟩
44:     sm_i.ts ← sm_i.ts + 1
45:     pend_i ← false

46: upon receiving ⟨"propose", sm, Req⟩ from p_j do
47:     if (sm_i.ts > sm.ts) or (sm_i.ts = sm.ts ∧ pend_i = true) then return
48:     sm_i ← sm
49:     pend_i ← true
50:     propose(sm_i.cng, sm_i.ts, Req)
51:     for all p ∈ sm_i.cng.mem send ⟨"propose", sm_i, Req⟩ to p

52: upon receiving ⟨"helpRequest", num, sm⟩ from p_j do
53:     if sm_i.ts < sm.ts then                                                   ▷ learn new sm
54:         sm_i ← sm
55:         pend_i ← false
56:     send ⟨"helpReply", num, myOp_i, opNum_i⟩

57: upon receiving ⟨"helpReply", opNum_i, op, num⟩ from p_j do
58:     ops_i ← ops_i ∪ ⟨p_j, op, num⟩

59: upon receiving ⟨"update", sm, num⟩ from p_j do
60:     if sm_i.ts < sm.ts then                                                   ▷ learn new sm
61:         sm_i ← sm
62:         pend_i ← false
63:     if num ≠ ⊥ then send ⟨"ACK", num⟩ to p_j
```

5 Conclusion

We proved that in an asynchronous API-based reconfigurable model allowing at least one failure, without restricting the number of reconfigurations, there is no way to emulate dynamic safe wait-free storage. We further showed how to circumvent this result using a dynamic eventually perfect failure detector: we presented an algorithm that uses such a failure detector in order to emulate a wait-free dynamic atomic MWMR register.

Our dynamic failure detector is (1) sufficient for this problem, and (2) can be implemented in a dynamic eventually synchronous [11] setting with no restriction on reconfiguration rate. An interesting question is whether a weaker such failure detector exists. Note that when the reconfiguration rate is bounded, dynamic storage is attainable without consensus, thus such a failure detector does not necessarily have to be strong enough for consensus.

References

[1] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer, et al. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 2010.

[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 2011.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[4] Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L Welch. Simulating a shared register in an asynchronous system that never stops changing. In *International Symposium on Distributed Computing*, pages 75–91. Springer, 2015.

[5] Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 639–647. IEEE, 2009.

[6] Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Regular register: an implementation in a churn prone environment. In *International Colloquium on Structural Information and Communication Complexity*, pages 15–29. Springer, 2009.

[7] Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *Parallel and Distributed Systems, IEEE Transactions on*, 2012.

[8] Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. Virtually synchronous methodology for dynamic service replication. 2010.

[9] Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M Musial, and Alex A Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1):100–116, 2009.

[10] Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 2001.

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

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

[13] Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*. Springer, 2015.

[14] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*. IEEE Computer Society, 2003.

[15] Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 2010.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, July 1990.

[17] Leander Jehl and Hein Meling. The case for reconfiguration without consensus. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.

[18] Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, 2015.

[19] Steven Y Ko, Imranul Hoque, and Indranil Gupta. Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 259–268. IEEE, 2008.

[20] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[21] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[22] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 233–242. ACM, 2002.

[23] Kal Lin and Vassos Hadzilacos. Asynchronous group membership with oracles. In *Distributed Computing*. Springer, 1999.

[24] Nancy Lynch and Alex A Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, pages 173–190. Springer, 2002.

[25] Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and AE Abbadi. From static distributed systems to dynamic systems. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 109–118. IEEE, 2005.

[26] Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. LADIS '10.

[27] Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *OPODIS*, 2015.

A Correctness Proof

In Section A.1 we prove that our algorithm satisfies atomicity, and in Section A.2 wait-freedom.

A.1 Atomicity

Every operation is uniquely defined by the process that invoked it and its local number. During the proof we refer to operation *op* invoked by process p_i with local number $opNum_i = n$ as the tuple $\langle p_i, op, n\rangle$. We begin the proof with three lemmas that link completed operations to *sm* states.

Lemma 1.3. *Consider operation op invoked by some process p_i in r with local number $opNum_i = n$. If op returns in r at time t, then there is at least one request* Req *that contains $\langle p_i, op, n\rangle$ and has been chosen in a consensus in r before time t.*

Proof. When operation *op* return, $sm_i.lastOps[i].num = n$ (line 17 or 18 in Algorithm 2). Processes update *sm* during a decide handler, or when a newer *sm* is received, and the first update occurs when some process p_j writes n to $sm_j.lastOps[i].num$ during a decide handler. In the decide handler, n is written to $sm.lastOps[i].num$ when the chosen request in the corresponding consensus contains $\langle p_i, op, n\rangle$.

□

Lemma 1.4. *For two processes p_i, p_j, let t be a time in a run r in which neither p_i or p_j is executing a decide handler. Then at time t, if $sm_i.ts = sm_j.ts$, then $sm_i = sm_j$.*

Proof. We prove by induction on timestamps. Initially, all correct processes have the same *sm* with timestamp 0. Now consider timestamp TS, and assume that for every two processes p_i, p_j at any time not during the execution of decide handlers, if $sm_i.ts = sm_j.ts = TS$, then $sm_i = sm_j$. Processes increase their $sm.ts$ to $TS + 1$ either at the end of a *decide* handler associated with TS or when they receive a message with *sm* s.t. $sm.ts = TS + 1$. By the agreement property of consensus and by the determinism of the algorithm, all the processes that perform the *decide* handler associated with TS perform the same operations, and therefore move *sm* (at the end of the handler) to the same state. It is easy to show by induction that all the processes that receive a message with *sm* s.t. $sm.ts = TS + 1$ receive the same *sm*. The lemma follows.

□

Observation 1. *For two process p_i, p_j, let sm_1 and sm_2 be the values of sm_j at two different times in a run r. If $sm_1.ts \geq sm_2.ts$, then $sm_1.lastOps[i].num \geq sm_2.lastOps[i].num$.*

Lemma 1.5. *Consider operation $\langle p_i, op, opNum_i\rangle$ invoked in r with $opNum_i = n$. Then $\langle p_i, op, n\rangle$ is part of at most one request that is chosen in a consensus in r.*

Proof. Assume by way of contradiction that $\langle p_i, op, n\rangle$ is part of more than one request that is chosen in a consensus in r. Now consider the earliest one, *Req*, and assume that it is chosen in a consensus associated with timestamp TS. At the end of the *decide* handler associated with timestamp TS, $sm.lastOps[i].num = n$ and the timestamp is increased to $TS + 1$. Thus, by Lemma 1.4 $sm.lastOps[i].num = n$ holds for every *sm* s.t. $sm.ts = TS+1$. Consider now the next request, Req_1, that contains $\langle p_i, op, n\rangle$, and is chosen in a consensus. Assume that this consensus associated with timestamp TS', and notice that $TS' > TS$. By the validity of consensus, this request is proposed by some process p_j, when $sm_j.ts$ is equal to TS'. By Observation 1, at this point $sm_j.lastOps[i].num \geq n$, and therefore p_j does not include $\langle p_i, op, n\rangle$ in Req_1 (line 27 in Algorithm 2). A contradiction.

□

Based on the above lemmas, we can define, for each run r, a linearization σ_r, where operations are ordered as they are chosen for execution on *sm*'s in r.

Definition 3. For a run r, we define the sequential run σ_r to be the sequence of operations decided in consensus instances in r, ordered by the order of the chosen requests they are part of in r. The order among operations that are part of the same chosen request is the following: first all *writes*, then all *reads*, and finally, all *reconfig* operations. Among each type, operations are ordered by the process ids of the processes that invoked them, from the highest to the lowest.

Note that for every run r, the sequential run σ_r is well defined. Moreover, σ_r contains every completed operation in r exactly once, and every invoked operation at most once.

In order to prove atomicity we show that (1) σ_r preserves r's real time order (lemma 1.6); and (2) every *read* operation rd in r returns the value that was written by the last *write* operation that precedes rd in σ_r, or $\perp$ if there is no such operation (lemma 1.7).

Lemma 1.6. *If operation op_1 returns before operation op_2 is invoked in r, then op_1 appears before op_2 in σ_r.*

Proof. By Lemma 1.3, op_1 is part of a request Req_1 that is chosen in a consensus before op_2 is invoked, and thus op_2 cannot be part of Req_1 or any other request that is chosen before Req_1. Hence op_1 appears before op_2 in σ_r.

□

Lemma 1.7. *Consider read operation $rd = \langle p_i, RD, n\rangle$ in r, which returns a value v. Then v is written by the last write operation that precedes rd in σ_r, or $v = \perp$ if there is no such operation.*

Proof. By Lemmas 1.3 and 1.5, rd is part of exactly one request Req_1 that is chosen in a consensus, associated with some timestamp TS. Thus $sm.lastOps[i]$ is set to $\langle n, val\rangle$ in the *decide* handler associated with TS. By Lemma 1.4, $sm.lastOps[i] = \langle n, val\rangle$ for all sm s.t. $sm.ts = TS + 1$. By Lemma 1.5 and since we consider only well-formed runs, $sm_i.lastOps[i] = \langle n, val\rangle$ when rd returns, and therefore rd returns val. Now consider three cases:

- There is no *write* operation in Req_1 or in any request that was chosen before Req_1 in r. In this case, there is no *write* operation before rd in σ_r, and no process writes to $sm.value$ before $sm.lastOps[i]$ is set to $\langle n, val\rangle$, and therefore, rd returns $\perp$ as expected.
- There is a *write* operation in Req_1 in r. Consider the *write* operation w in Req_1 that is invoked by the process with the lowest id, and assume its argument is v'. Notice that w is the last *write* that precedes rd in σ_r. By the code of the *decide* handler, $sm.value$ equals v' at the time when $sm.lastOps[i]$ is set to $\langle n, val\rangle$. Therefore, $val = v'$, rd returns the value that is written by the last *write* operation that precedes it in σ_r.
- There is no *write* operation in Req_1, but there is a request that contains a *write* operation and is chosen before Req_1 in r. Consider the last such request Req_2, and consider the *write* operation w invoked by the process with the lowest id in Req_2. Assume that w's argument is v', and Req_2 was chosen in a consensus associated with timestamp TS' (notice that $TS' < TS$). By the code of the *decide* handler and Lemma 1.4, in all the sm's s.t. $sm.ts = TS' + 1$, the value of $sm.value$ is v'. Now, since there is no *write* operation in any chosen request between Req_2 and Req_1 in r, no process writes to $sm.value$ when $TS' < sm.ts < TS$. Hence, when $sm.lastOps[i]$ is set to $\langle n, val\rangle$, $sm.value$ equals v', and thus $val = v'$. Therefore, rd returns the value that is written by the last *write* operation that precedes rd in σ_r.

□

Corollary 1. *Algorithms 1–3 implement an atomic storage service.*

A.2 Liveness

Consider operation op_i invoked at time t by a correct process p_i in run r. Notice that r is a run with either infinitely or finitely many invocations. We show that, in both cases, if p_i is active in r, then op_i returns in r.

We associate the addition or removal of process p_j by a process p_i with the timestamp that equals $sm_i.ts$ at the time when the operation returns. The addition of all processes in P_0 is associated with timestamp 0.

First, we consider runs with infinitely many invocations. In Lemma 1.8, we show that for every process p, every sm associated with a larger timestamp than p's addition contains p in $sm.cng.mem$. In Observation 1, we show that in a run with infinitely many invocations, for every timestamp ts, there is a completed operation that has a bigger timestamp than ts at the time of the invocation. Moreover, after the stabilization time of the FD, operations must help all the slow active processes in order to complete. In Lemma 1.9, we use the observation to show that any operation invoked in a run with infinitely many invocations returns.

Next, we consider runs with finitely many invocations. We show Lemma 1.10 that eventually all the active members of the last sm adopt it. Then, in Lemma 1.11, we show that every operation invoked by an active process completes. Finally, Theorem 2, stipulates that the algorithm satisfies wait-freedom.

Lemma 1.8. *Assume the addition of p_i is associated with timestamp TS in run r. If p_i is active, then $p_i \in sm.cng.mem$ for every sm s.t. $sm.ts \geq TS$.*

Proof. The proof is by induction on $sm.ts$. **Base:** If $p_i \in P_0$, then $p_i \in sm.cng.mem$ for all sm s.t. $sm.ts = 0$. Otherwise, $\langle add, p_i \rangle$ is part of a request that is chosen in a consensus associated with timestamp $TS' = TS - 1$, and thus, by Lemma 1.4, $p_i \in sm.cng.mem$ for all sm s.t. $sm.ts = TS' + 1 = TS$. **Induction:** Process p_i is active, so no process invokes $\langle remove, p_i \rangle$, and therefore, together with the validity of consensus, no chosen request contains $\langle remove, p_i \rangle$. Hence, if $p_i \in sm.cng.mem$ for sm with $sm.ts = k$, then $p_i \in sm.cng.mem$ for every sm s.t. $sm.ts = k + 1$.

□

Claim 1. *Consider a run r of the algorithm with infinitely many invocations. Then for every time t and timestamp TS, there is a completed operation that is invoked after time t by a process with $sm.ts > TS$ at the time of the invocation.*

Proof. Recall that r is well-formed and only processes in $V(t).join$ can invoke operations at time t. Therefore, there are infinitely many completed operations in r. Since a finite number of operations are completed with each timestamp, the claim follows.

□

Lemma 1.9. *Consider an operation op_i invoked at time t by an active process p_i in a run r with infinitely many invocations. Then op_i completes in r.*

Proof. Assume by way of contradiction that p_i is active and op_i does not complete in r. Assume w.l.o.g. that p_i's addition is associated with timestamps TS and op_i is invoked with $opNum_i = n$. Consider a time $t' > t$ after p_i invokes op_i and the FD has stabilized. By Claim 1, there is a completed operation op_j in r, invoked by some process p_j at a time $t'' > t'$ when $sm_j.ts > TS$, whose completion is associated with timestamp TS'. By Lemma 1.8, $p_i \in sm_j.cng.mem$, at time t''. Now by the algorithm and by the eventual strong accuracy property of the FD, p_j proposes op_j and op_i in the same request, and continues to propose both of them until one is selected. Note that it is impossible for op_j to be selected without op_i since any process that helps p_j after stabilization also helps p_i. Hence, since op_j completes, they are both performed in the same *decide* handler. The run is well-formed, so p_i does not invoke operations that are associated with $opNum_i > n$. Hence, following the time when op_i is selected, for all sm s.t. $sm.ts > TS'$, $sm.lastOps[i].num = n$. Now, again by Claim 1, consider a completed operation op_k in r, that is invoked by some process p_k at time t''' after the stabilization time of the FD s.t. $sm_k.ts > TS'$

at time t'''. Operation op_k cannot complete until p_i receives p_k's sm. Therefore, p_i receives sm s.t. $sm.ts \geq TS'$, and thus $sm.lastOps[i].num = n$. Therefore, p_i learns that op_i was performed, and op_i completes. A contradiction.

□

We now proceed to prove liveness in runs with finitely many invocations.

Definition 4. For every run r of the algorithm, and for any point t in r, let TS_t be the timestamp associated with the last consensus that made a decision in r before time t. Define sm^t, at any point t in r, to be the sm's state after the completion of the decide handler associated with timestamp TS_t at any process. By Lemma 1.4, sm^t is unique. Recall that sm^0 is the initial state.

Claim 2. *For every run r of the algorithm, and for any point t in r, there is a majority of $sm^t.cng.mem$ M s.t. $M \subseteq (V(t).membership \cup P(t).join) \setminus F(t)$.*

Proof. By the code of the algorithm, for every run r and for any point t in r, $V(t).membership \subseteq sm^t.cng.mem$ and $sm^t.cng.mem \cap V(t).remove = \{\}$. The claim follows from failure condition.

□

Observation 2. *Consider a run r of the algorithm with finitely many invocations. Then there is a point t in r s.t. for every $t' > t$, $sm^t = sm^{t'}$. Denote this sm to be $\hat{sm}$.*

The following lemma follows from Lemma 1.4, Claim 2, and the periodic update messages; for space limitations, we omit its proof.

Lemma 1.10. *Consider a run r of the algorithm with finitely many invocations. Then eventually for every active process $p_i \in \hat{sm}.cng.mem$, $sm_i = \hat{sm}$.*

Lemma 1.11. *Consider an operation op_i invoked at time t by an active process p_i in a run r with finitely many invocations. Then op_i completes in r.*

Proof. By Lemma 1.8, $p_i \in \hat{sm}.cng.mem$, and by Lemma 1.10, there is a point t' in r s.t. $sm_i = \hat{sm}$ for all $t \geq t'$. Assume by way of contradiction that op_i does not complete in r. Therefore, op_i is either stuck in one of its waits or continuously iterates in a while loop. In each case, we show a contradiction. Denote by *con* the consensus associated with timestamp $\hat{sm}.ts$. By definition of $\hat{sm}$, no decision is made in *con* in r.

- Operation op_i waits in line 16 (Algorithm 2) forever. Notice that $\hat{sm}.cng.rem$ contains all the process that were removed in r, so, after time t', p_i does not wait for a reply from a removed process. By the strong completeness property of FD, p_i does not wait for faulty processes forever. A contradiction.

- Operation op_i waits in line 18 (Algorithm 2) forever. Notice that from time t' till p_i proposes in *con*, $pend_i$=*false*. Therefore, p_i proposes in *con* in line 22 (Algorithm 2), and waits in line 18 after the propose. By Observation 2, there is a majority M of $\hat{sm}.cng.em$ s.t. $M \subseteq V(t).membership \cup P(t).join \setminus F(t)$. Therefore, by the termination of consensus, eventually a decision is made in *con*. A contradiction to the definition of $\hat{sm}$.

- Operation op_i remains in the while loop in line 17 (Algorithm 2) forever. Since it does not waits in line 18 (Algorithm 2) forever, op_i proposes infinitely many times, and since each propose is made in a different consensus and p_i can propose in a consensus beyond the first one only once a decision is made in the previous one, infinitely many decisions are made in r. A contradiction to the definition of $\hat{sm}$.

- Operation op_i waits in line 27 (Algorithm 2) forever. Consider two cases. First, $sm_i \neq \hat{sm}$ when p_i performs line 26 (Algorithm 2). In this case, p_i continues at time t', when it adopts $\hat{sm}$, because $sm_i.ts > ts$ hold at time t'. In the second case ($sm_i = \hat{sm}$ when p_i performs line 26), p_i sends *update* message to all processes in $\hat{sm}.cng.mem$, and waits for a majority to reply. By Observation 2, there is a correct majority in $\hat{sm}.cng.mem$, and thus p_i eventually receives the replies and continues. In both cases we have contradiction.

Therefore, p_i completes in r.

□

We conclude with the following theorem:

Theorem 2. *Algorithms 1–3 implement wait-free atomic dynamic storage.*