Dynamic Atomic Storage without Consensus

MARCOS K. AGUILERA, Microsoft Research
IDIT KEIDAR, Technion
DAHLIA MALKHI, Microsoft Research
ALEXANDER SHRAER, Yahoo! Research

7

This article deals with the emulation of atomic read/write (R/W) storage in *dynamic* asynchronous message passing systems. In static settings, it is well known that atomic R/W storage can be implemented in a fault-tolerant manner even if the system is completely asynchronous, whereas consensus is not solvable. In contrast, all existing emulations of atomic storage in dynamic systems rely on consensus or stronger primitives, leading to a popular belief that dynamic R/W storage is unattainable without consensus.

In this article, we specify the problem of dynamic atomic read/write storage in terms of the interface available to the users of such storage. We discover that, perhaps surprisingly, dynamic R/W storage is solvable in a completely asynchronous system: we present DynaStore, an algorithm that solves this problem. Our result implies that atomic R/W storage is in fact easier than consensus, even in dynamic systems.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Shared memory*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage, distributed memories*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Algorithms, Design, Reliability, Theory

Additional Key Words and Phrases: Shared-memory emulations, dynamic systems, atomic storage

ACM Reference Format:
Aguilera, M. K., Keidar, I., Malkhi, D., and Shraer, A. 2011. Dynamic atomic storage without consensus. J. ACM 58, 2, Article 7 (April 2011), 32 pages.
DOI = 10.1145/1944345.1944348 http://doi.acm.org/10.1145/1944345.1944348

1. INTRODUCTION

Distributed systems provide high availability by replicating the service state at multiple processes. A fault-tolerant distributed system may be designed to tolerate failures of a minority of its processes. However, this approach is inadequate for long-lived systems, because over a long period, the chances of losing more than a minority inevitably increase. Moreover, system administrators may wish to deploy new machines due to increased workloads, and replace old, slow machines with new, faster ones. Thus, real-world distributed systems need to be *dynamic*, that is, adjust their membership over

A preliminary version of this article appears in *Proceedings of the 28th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC* 2009*)*.
Authors' addresses: M. K. Aguilera and D. Malkhi, Microsoft Research Silicon Valley, 1288 Pear Ave., Mountain View, CA 94043; I. Keidar, Department of Electrical Engineering, Technion, Haifa, 32000, Israel; A. Shraer, Yahoo! Research, 4401 Great America Parkway, Santa Clara, CA 95054; email: shralex@yahoo-inc.com.

DOI 10.1145/1944345.1944348 http://doi.acm.org/10.1145/1944345.1944348

time. Such dynamism is realized by providing users with an interface to reconfiguration operations that add or remove processes.

Dynamism requires some care. First, if one allows arbitrary reconfiguration, one may lose liveness. For example, say that we build a fault tolerant solution using three processes, p_1, p_2, and p_3. Normally, the adversary may crash one process at any moment in time, and the up-to-date system state is stored at a majority of the current configuration. However, if a user initiates the removal of p_1 while p_1 and p_2 are the ones holding the up-to-date system state, then the adversary may not be allowed to crash p_2, for otherwise the remaining set cannot reconstruct the up-to-date state. Providing a general characterization of allowable failures under which liveness can be ensured is a challenging problem.

A second challenge dynamism poses is ensuring safety in the face of concurrent reconfigurations, that is, when some user invokes a new reconfiguration request while another request (potentially initiated by another user) is under way. Early work on replication with dynamic membership could violate safety in such cases [Davcev and Burkhard 1985; Paris and Long 1988; El Abbadi and Dani 1991] (as shown in Yeger Lotem et al. [1997]). Many later works have rectified this problem by using a centralized sequencer or some variant of consensus to agree on the order of reconfigurations (see discussion of related work in Section 2).

Interestingly, consensus is not essential for implementing replicated storage. The ABD algorithm [Attiya et al. 1995] shows that atomic read/write (R/W) shared memory objects can be implemented in a fault-tolerant manner even if the system is completely asynchronous. Nevertheless, to the best of our knowledge, all previous dynamic storage solutions rely on consensus or similar primitives, leading to a popular belief that dynamic storage is unattainable without consensus.

In this work, we address the two challenges mentioned above, and debunk the myth that consensus is needed for dynamic storage. We first provide a precise specification of a dynamic problem. To be concrete, we focus on atomic R/W storage, though we believe the approach we take for defining a dynamic problem can be carried to other problems. We then present *DynaStore*, a solution to this problem in an asynchronous system where processes may undetectably crash, so that consensus is not solvable. We note that our solution is given as a possibility proof, rather than as a blueprint for a new storage system. Given our result that consensus-less solutions are possible, we expect future work to apply various practical optimizations to our general approach, in order to build real-world distributed services. We next elaborate on these two contributions.

Dynamic Problem Specification

In Section 3, we define the problem of an atomic R/W register in a dynamic system. Similarly to a static R/W register, the dynamic variant exposes a *read* and *write* interface to users, and atomicity [Lamport 1986] is required for all such operations. In addition, users can trigger reconfigurations by invoking *reconfig* operations, which return OK when they complete. Exposing *reconfig* operations in the model allows us to provide a protocol-independent specification of service liveness guarantees, as we explain next.

Clearly, the progress of such service is conditioned on certain failure restrictions in the deployed system. A fault model specifies the conditions under which progress is guaranteed. It is well understood how to state a liveness condition of the static version of this problem: t-resilient R/W storage guarantees progress if fewer than t processes crash. For an n-process system, it is well known that t-resilient R/W storage exists when $t < n/2$, and does not exist when $t \geq n/2$ [Attiya et al. 1995]. A dynamic fault model serves the same purpose, but needs to additionally capture changes introduced by the user through the *reconfig* interface. Under reasonable use of *reconfig*, and some restricted fault conditions, the system will make progress. For example, an administrative-user can deploy machines to replace faulty ones, and thereby enhance system longevity. On the other hand, if used carelessly, reconfiguration might cause the service to halt, for example, when servers are capriciously removed from the system.

Suppose the system initially has four processes $\{p_1, p_2, p_3, p_4\}$ in its configuration (also called *view*). Initially, any one process may crash. Suppose that p_1 crashes. Then, additional crashes would lead to a loss of liveness. Now suppose the user requests to reconfigure the system to remove p_1. While the request is pending, no additional crashes can happen, because the system must transfer the up-to-date state from majority of the previous view to a majority of the new one. However, once the removal is completed, the system can tolerate an additional crash among the new view $\{p_2, p_3, p_4\}$. Overall, two processes may crash during the execution. Viewed as a simple threshold condition, this exceeds a minority threshold, which contradicts lower bounds. The liveness condition we formulate is therefore not in the form of a simple threshold; rather, we require crashes to occur gradually, contingent on reconfigurations.

A dynamic system also needs to support additions. Suppose the system starts with three processes $\{p_1, p_2, p_3\}$. In order to reconfigure the system to add a new process p_4, a majority of the view $\{p_1, p_2, p_3\}$ must be alive to effect the change. Additionally, a majority of the view $\{p_1, p_2, p_3, p_4\}$ must be alive to hold the state stored by the system. Again, the condition here is more involved than a simple threshold. That is, if a user requests to *add* p_4, then while the request is pending, a majority of both old and new views need to be alive. Once the reconfiguration is completed, the requirement weakens to a majority of the new view.

Given these, we state the following requirement for liveness for dynamic R/W storage: At any moment in the execution, let the current view consist of the initial view with all completed reconfiguration operations (add/remove) applied to it. We require that the set of crashed processes and those whose removal is pending be a minority of the current view, and of any pending future views. Moreover, like previous reconfigurable storage algorithms [Lynch and Shvartsman 2002; Gilbert et al. 2003], we require that no new *reconfig* operations will be invoked for “sufficiently long” for the started operations to complete. This is formally captured by assuming that only a finite number of *reconfig* operations are invoked.

Note that a dynamic problem is harder than the static variant. In particular, a solution to dynamic R/W is a fortiori a solution to the static R/W problem. Indeed, the solution must serve read and write requests, and in addition, implement reconfiguration operations. If deployed in a system where the user invokes only read and write requests, and never makes use of the reconfiguration interface, it must solve the R/W problem with precisely the same liveness condition, namely, tolerating any minority of failures. Similarly, dynamic consensus is harder than static consensus, and is therefore a fortiori not solvable in an asynchronous setting with one crash failure allowed. As noted above, in this paper, we focus on dynamic R/W storage.

DynaStore: Dynamic Atomic R/W Storage

Our algorithm does not need consensus to implement reconfiguration operations. Intuitively, previous protocols used consensus, virtual synchrony, or a sequencer, in order to provide processes with an agreed-upon sequence of configurations, so that the membership views of processes do not diverge. The key observation in our work is that it is sufficient that such a sequence of configurations exists, and there is no need for processes to know precisely which configurations belong to this sequence, as long as they have some assessment which includes these configurations, possibly in addition to others that are not in the sequence. In order to enable this property, in Section 4 we introduce *weak snapshots*, which are easily implementable in an asynchronous system. Roughly speaking, such objects support *update* and *scan* operations accessible by a given set of processes, such that *scan* returns a set of updates that, if non-empty, is guaranteed to include the *first update* made to the object (but the object cannot identify which update that is).

In DynaStore, which we present in Section 5, each view w has a weak snapshot object $ws(w)$, which stores reconfiguration proposals for what the next view should be. Thus, we can define a unique global sequence of views, as the sequence that starts with some fixed initial view, and continues by following the first proposal stored in each view's ws object. Although it is impossible for processes to learn what this sequence is, they can learn a DAG of views that includes this sequence as a path. They do this by creating a vertex for the current view, querying the ws object, creating an edge to each view in the response, and recursing. Reading and writing from a chain of views is then done in a manner similar to previous protocols, for example, Lynch and Shvartsman [2002], Gilbert et al. [2003], Chockler et al. [2005], and Rodrigues and Liskov [2003, 2004].

Summary of Contributions

In summary, our work makes two contributions.

—We define a dynamic R/W storage problem that includes a clean and explicit liveness condition, which does not depend on a particular solution to the problem. The definition captures a dynamically changing resilience requirement, corresponding to reconfiguration operations invoked by users. The approach easily carries to other problems, such as consensus. As such, it gives a clean extension of existing static problems to the dynamic setting.

—We discover that dynamic R/W storage is solvable in a completely asynchronous system with failures, by presenting a solution to this problem. Along the way we define a new abstraction of weak snapshots, employed by our solution, which may be useful in its own right. Our result implies that the dynamic R/W is weaker than the (dynamic) consensus problem, which is not solvable in this setting. This was known before for static systems, but not for the dynamic version. The result counters the intuition that emanates from all previous dynamic systems, which used agreement to handle configuration changes.

2. RELATED WORK

Several existing solutions can be viewed in retrospect as solving a dynamic problem. Most closely related are works on reconfigurable R/W storage. RAMBO [Lynch and Shvartsman 2002; Gilbert et al. 2003] solves a similar problem to the one we have formulated above; other works [Martin and Alvisi 2004; Rodrigues and Liskov 2003; 2004] extend this concept for Byzantine fault tolerance. All of these works have processes agree upon a unique sequence of configuration changes. Some works use an auxiliary source (such as a single reconfigurer process or an external consensus algorithm) to determine configuration changes [Lynch and Shvartsman 1997, 2002; Englert and Shvartsman 2000; Gilbert et al. 2003; Martin and Alvisi 2004; Rodrigues and Liskov 2004], while others implement fault-tolerant consensus decisions on view changes [Chockler et al. 2005; Rodrigues and Liskov 2003]. In contrast, our work implements reconfigurable R/W storage without any agreement on view changes.

Since the closest related work is on RAMBO, we further elaborate on the similarities and differences between RAMBO and DynaStore. In RAMBO, a new configuration can be proposed by any process, and once it is installed, it becomes the current configuration. In DynaStore, processes suggest changes and not configurations, and thus, the current configuration is determined by the set of all changes proposed by complete reconfigurations. For example, if a process suggests to add p_1 and to remove p_2, while another process concurrently suggests to add p_3, DynaStore will install a configuration including both p_1 and p_3 and without p_2, whereas in RAMBO there is no guarantee that any future configuration will reflect all three proposed changes, unless some process explicitly proposes such a configuration. In DynaStore, a quorum of a configuration is any majority of its members, whereas RAMBO allows for general quorum-systems, specified explicitly for each configuration by the proposing process. In both algorithms, a non-faulty quorum is required from the current configuration. A central idea in allowing dynamic changes is that a configuration can be replaced, after which a quorum of the old configuration can crash. In DynaStore, a majority of a current configuration C is allowed to crash as soon as C is no longer current, that is, when a *reconfig* operation proposing a new membership change completes at one of the processes. Notice that a *reconfig* operation in DynaStore involves communication with a majority of C and the new configuration (for state-transfer) allowing any minority of C to crash at any time. In RAMBO, C must be garbage-collected at every nonfaulty process $p \in C$, and all *read* and *write* operations that began at p before C was garbage-collected must complete. Thus, whereas in DynaStore the conditions allowing a quorum of C to fail can be evaluated based on events visible to the application, in RAMBO these conditions are internal to the algorithm. Moreover, if some process $p \in C$ might fail, it might be impossible for other processes to learn whether a quorum of C is still needed. Assuming that all quorums required by RAMBO and DynaStore are responsive, both algorithms require additional assumptions for liveness. In both, the liveness of *read* and *write* operations is conditioned on the number of reconfigurations being finite. In addition, in both algorithms, the liveness of reconfigurations does not depend on concurrent *read* and *write* operations. However, whereas reconfigurations in RAMBO rely on additional synchrony or failure-detection assumptions required for consensus, reconfigurations in DynaStore, just like its *read* and *write* operations, only require the number of reconfigurations to be finite.

View-oriented group communication systems provide a membership service whose task is to maintain a dynamic view of active members. These systems solve a dynamic problem of maintaining agreement on a sequence of views, and additionally provide certain services within the members of a view, such as atomic multicast and others [Chockler et al. 2001; Birman et al. 2010]. Maintaining agreement on group membership in itself is impossible in asynchronous systems [Chandra et al. 1996]. However, perhaps surprisingly, we show that the dynamic R/W problem is solvable in asynchronous systems. This appears to contradict the impossibility but it does not: We do not implement group membership because our processes do not have to agree on and learn a unique sequence of view changes. Furthermore, unlike to group communication systems we do not expose views to the application and views are only used internally in the analysis. Processes running our algorithm maintain a local estimate of the current view of the system, however such views do not necessarily correspond to a view of the system as visible to any external observer (some membership changes may not have been acknowledged to a user). Local estimates at different processes may diverge and re-merge over time when no new membership changes are proposed for a sufficiently long period of time.

The State Machine Replication (SMR) approach [Lamport 1998; Schneider 1990] provides a fault tolerant emulation of arbitrary data types by forming agreement on a sequence of operations applied to the data. Paxos [Lamport 1998] implements SMR, and allows one to dynamically reconfigure the system by keeping the configuration itself as part of the state stored by the state machine. Another approach for reconfigurable SMR is to utilize an auxiliary configuration-master to determine view changes, and incorporate directives from the master into the replication protocol. This approach is adopted in several practical systems, for example, Lee and Thekkath [1996], MacCormick et al. [2004], and van Renesse and Schneider [2004], and is formulated in Lamport et al. [2009]. Naturally, a reconfigurable SMR can support our dynamic R/W memory problem. However, our work solves it without using the full generality of SMR and without reliance on consensus.

An alternative way to break the minority barrier in R/W emulation is by strengthening the model using a failure detector. Delporte-Gallet et al. [2010] identify the weakest failure detector for solving R/W memory with arbitrary failure thresholds. Their motivation is similar to ours – solving R/W memory with increased resilience threshold. Unlike our approach, they tackle more than a minority of failures right from the outset. They identify the *quorums failure detector* as the weakest detector required for strengthening the asynchronous model, in order to break the minority resilience threshold. Our approach is incomparable to theirs, that is, our model is neither weaker nor stronger. On the one hand, we do not require a failure detector, and on the other, we allow the number to failures to exceed a minority only after certain actions are taken. Moreover, their model does not allow for additions as ours does. Indeed, our goal differs from Delporte-Gallet et al. [2010], namely, to model dynamic reconfiguration in which resilience is adaptive to actions by the processes. It is an interesting future direction to define a quorum failure detector corresponding to the adaptive failure model used in this article.

3. DYNAMIC PROBLEM DEFINITION

We specify a read/write service with atomicity guarantees. The storage service is deployed on a collection of processes that interact using asynchronous message passing. We assume an unknown, unbounded and possibly infinite universe of processes Π, subject to crash failures. Communication links between all pairs of processes do not create, duplicate, or alter messages. Moreover, the links are reliable: if a process p_i sends a message m to a process p_j and neither p_i nor p_j crash then p_j eventually receives m.[1]

Executions and Histories. System components, namely the processes and the communication links between them, are modeled as I/O Automata [Lynch 1996]. An automaton has a state, which changes according to *transitions* that are triggered by *actions*, which are classified as input, output, and internal.[2] A *protocol* P specifies the behaviors of all processes. An *execution* of P is a sequence of alternating states and actions, such that state transitions occur according to the specification of system components. The occurrence of an action in an execution is called an *event*.

The application interacts with the service via *operations* defined by the service interface. As operations take time,[3] they are represented by two events – an *invocation* (input action) and a *response* (output action). A process p_i interacts with its incoming link from process p_j via the $receive(m)_{i,j}$ input action, and with its outgoing link to p_j via the $send(m)_{i,j}$ output action. The failure of process p_i is modeled using the input action $crash_i$, which disables all actions at p_i. In addition, p_i can disable all input actions using the internal action $halt_i$.

A *history* of an execution consists of the sequence of invocations and responses occurring in the execution. An operation is *complete* in a history if it has a matching response. An operation o *precedes* another operation o' in a sequence of events σ, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, if o precedes o' in σ then o precedes

[1] This requirement can be weakened to account for processes that have not yet joined or have left the system. The issue of message reliability in a dynamic setting was studied in the context of group communication systems [Chockler et al. 2001].

[2] A minor difference from I/O Automata as defined in Lynch [1996], is that in our model input actions can be disabled, as explained below. Note that we do not make use of any I/O Automata property that may be affected by this difference.

[3] By slight abuse of terminology, we use the terms *operation* and *operation execution* interchangeably.

o' in π. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent operations.

We assume that executions of our algorithm are *well-formed*, that is, the sequence of events at each client consists of alternating invocations and matching responses, starting with an invocation. Finally, we assume that every execution is *fair*, which means, informally, that it does not halt prematurely when there are still steps to be taken or messages to be delivered (see the standard literature for a formal definition [Lynch 1996]).

Service Interface. We consider a multi-writer/multi-reader (MWMR) service, from which any process may read or write. The service stores a value v from a domain $\mathcal{V}$ and offers an interface for invoking *read* and *write* operations and obtaining their result. Initially, the service holds a special value $\perp \notin \mathcal{V}$. When a read operation is invoked at a process p_i, the service responds with a value x, denoted $read_i() \rightarrow x$. When a write is invoked at p_i with a value $x \in \mathcal{V}$, denoted $write_i(x)$, the response is OK. We assume that the written values are unique, that is, no value is written more than once. This is done so that we are able to link a value to a particular write operation in the analysis, and can easily be implemented by having *write* operations augment the value with the identifier of the writer and a local sequence number.

In addition to *read* and *write* operations, the service exposes an interface for invoking reconfigurations. We define $Changes \stackrel{def}{=} \{Remove, Add\} \times \Pi$. We informally call any subset of Changes a *set of changes*. A *view* is a set of changes. A *reconfig* operation takes as parameter a set of changes c and returns OK. We say that a change $\omega \in Changes$ is *proposed* in an execution if a $reconfig_i(c)$ operation is invoked at some process p_i such that $\omega \in c$.

Intuitively, only processes that are members of the current system configuration should be allowed to initiate actions. To capture this restriction, we define an output action *enable operations*; the *read*, *write* and *reconfig* input actions at a process p_i are initially disabled, until an *enable operations* event occurs at p_i.

Safety Specification. The sequential specification of the service indicates its behavior in sequential executions. It requires that each *read* operation returns the value written by the most recent preceding *write* operation, if there is one, and the initial value $\perp$ otherwise.

Atomicity [Lamport 1986], also called linearizability [Herlihy and Wing 1990], requires that for every execution, there exist a corresponding sequential execution, which preserves the real-time order, and which satisfies the sequential specification. Formally, let σ_{RW} be the subsequence of a history σ consisting of all events corresponding to the *read* and *write* operations in σ, without any events corresponding to *reconfig* operations. Linearizability is defined as follows:

Definition 3.1 (*linearizability* [*Herlihy and Wing* 1990]). A history σ is *linearizable* if σ_{RW} can be extended (by appending zero or more response events) to a history σ', and there exists a sequential permutation π of the subsequence of σ' consisting only of complete operations such that:

(1) π preserves the real-time order of σ; and
(2) The operations of π satisfy the sequential specification.

Active Processes. We assume a non-empty view *Init*, which is initially known to every process in the system. We say, by convention, that a *reconfig*(*Init*) completes by time 0. A process p_i is *active* if p_i does not crash, some process invokes a *reconfig* operation to add p_i, and no process invokes a *reconfig* operation to remove p_i. We do not require all processes in Π to start taking steps from the beginning of the execution, but instead we assume that if p_i is active then p_i takes infinitely many steps (if p_i is not active, then it may stop taking steps).

Dynamic Service Liveness. We first give preliminary definitions, required to specify service liveness. For a set of changes w, the *removal-set of* w, denoted $w.remove$, is the set $\{i \mid (Remove, i) \in w\}$. The *join set of* w, denoted $w.join$, is the set $\{i \mid (Add, i) \in w\}$. Finally, the *membership of* w, denoted $w.members$, is the set $w.join \setminus w.remove$.

At any time t in the execution, we define $V(t)$ to be the union of all sets c such that a *reconfig(c)* completes by time t. Thus, $V(0) = Init$. Note that removals are permanent, that is, a process that is removed will never again be in members. More precisely, if a reconfiguration removing p_i from the system completes at time t_0, then p_i is excluded from $V(t).members$, for every $t \geq t_0$.[4] Let $P(t)$ be the set of *pending changes* at time t, that is, for each element $\omega \in P(t)$ some process invokes a *reconfig*(c) operation such that $\omega \in c$ by time t, and no process completes such a *reconfig* operation by time t. Denote by $F(t)$ the set of processes that crashed by time t.

Intuitively, any pending future view should have a majority of processes that did not crash and were not proposed for removal; we specify a simple condition sufficient to ensure this. A dynamic R/W service guarantees the following liveness properties:

Definition 3.2 (*Dynamic Service Liveness*). If at every time t in the execution, fewer than $|V(t).members|/2$ processes out of $V(t).members \cup P(t).join$ are in $F(t) \cup P(t).remove$, and the number of different changes proposed in the execution is finite,[5] then the following hold:

(1) Eventually, the *enable operations* event occurs at every active process that was added by a complete *reconfig* operation.
(2) Every operation invoked at an active process eventually completes.

4. THE WEAK SNAPSHOT ABSTRACTION

A weak snapshot object S accessible by a set P of processes supports two operations, $update_i(c)$ and $scan_i()$, for a process $p_i \in P$. The $update_i(c)$ operation gets a value c and returns OK, whereas $scan_i()$ returns a set of values. Note that the set P of processes is fixed (i.e., *static*). We require the following semantics from *scan* and *update* operations:

- PR1 (integrity) Let o be a $scan_i()$ operation that returns C. Then, for each $c \in C$, an $update_j(c)$ operation is invoked by some process p_j prior to the completion of o.
- PR2 (validity) Let o be a $scan_i()$ operation that is invoked after the completion of an $update_j(c)$ operation, and that returns C. Then, $C \neq \emptyset$.
- PR3 (monotonicity of scans) Let o be a $scan_i()$ operation that returns C and let o' be a $scan_j()$ operation that returns C' and is invoked after the completion of o. Then, $C \subseteq C'$.
- PR4 (non-empty intersection) There exists c such that for every $scan()$ operation that returns $C \neq \emptyset$, it holds that $c \in C$.
- PR5 (termination) If some majority M of processes in P does not crash, then every $scan_i()$ and $update_i(c)$ invoked by any process $p_i \in M$ eventually completes.

Although these properties bear resemblance to the properties of atomic snapshot objects [Afek et al. 1993], PR1-PR5 define a weaker abstraction: we do not require that all updates are ordered as in atomic snapshot objects, and even in a sequential

[4]In practice, one can add back a process by changing its id.
[5]In reality, liveness would still hold even with an infinite number of reconfigurations, provided that each operation is concurrent with a finite number of reconfigurations. It is easy to show that this requirement is necessary for liveness.

ALGORITHM 1: Weak snapshot - code for process p_i

```
operation update_i(c)
  if collect() = ∅ then
    Mem[i].Write(c)
  end if
  return OK

operation scan_i()
  C ← collect()
  if C = ∅ then return ∅
  C ← collect()
  return C

procedure collect()
  C ← ∅;
  for each p_k ∈ P
    c ← Mem[k].Read()
    if c ≠ ⊥ then C ← C ∪ {c}
  return C
end
```

execution, the set returned by a *scan* does not have to include the value of the most recently completed *update* that precedes it (validity only requires that *some* value is returned). Intuitively, these properties only require that the "first" *update* is seen by all *scan*s that see any *update*s. As we shall see below, this allows for a simpler implementation than of a snapshot object. In particular, in a sequential execution our algorithm only records the value of the first *update* whereas subsequent updates have no effect.

DynaStore will use multiple weak snapshot objects, one of each view w. The weak snapshot of view w, denoted $ws(w)$, is accessible by the processes in $w.members$. To simplify notation, we denote by $update_i(w, c)$ and $scan_i(w)$ the *update* and *scan* operation, respectively, of process p_i of the weak snapshot object $ws(w)$. Intuitively, DynaStore uses weak snapshots as follows: in order to propose a set of changes c to the view w, a process p_i invokes $update_i(w, c)$; p_i can then learn proposals of other processes by invoking $scan_i(w)$, which returns a set of sets of changes.

Implementation. Our implementation of *scan* and *update* is shown in Algorithm 1. It uses an array *Mem* of $|P|$ single-writer multi-reader (SWMR) atomic registers, where all registers are initialized to $\perp$. Such registers support *Read*() and *Write*(c) operations such that only process $p_i \in P$ invokes $Mem[i].Write(c)$ and any process $p_j \in P$ can invoke $Mem[i].Read()$. The implementation of such registers in message-passing systems is described in the literature [Attiya et al. 1995].

A $scan_i()$ reads from all registers in *Mem* by invoking *collect*, which returns the set C of values found in all registers. After invoking *collect* once, $scan_i()$ checks whether the returned C is empty. If so, it returns $\emptyset$, and otherwise invokes *collect* one more time. An $update_i(c)$ invokes *collect*, and in case $\emptyset$ is returned, writes c to $Mem[i]$. If *collect*() returns a non-empty set, the *update* simply returns OK. Intuitively, in this case another *update* is already the "first" and there is no need to perform a *Write* since future *scan* operations would not be obligated to observe it. In DynaStore, this happens when some process has already proposed changes to the view, and thus, the weak snapshot does not correspond to the most up-to-date view in the system and there is no need to propose additional changes to this view.

4.1. Correctness of Algorithm 1

Standard emulation protocols for atomic SWMR registers [Attiya et al. 1995] guarantee integrity (property PR1) and termination (property PR5). We next show that Algorithm 1 preserves properties PR2-PR4. We assume that all registers in *Mem* are initialized to $\perp$ and that no process invokes *update*($\perp$), which is indeed preserved by DynaStore.

Notice that at most one *Mem*[i].*Write* operation can be invoked in the execution, since after the first *Mem*[i].*Write* operation completes, any *collect* invoked by p_i (the only writer of this register) will return a non-empty set and p_i will never invoke another *Write*. Informally, this together with atomicity of all registers in *Mem* implies properties PR2-PR3. We start the formal proof of these two properties by showing that each register *Mem*[i] can be assigned at-most one noninitial value.

LEMMA 4.1. *For any* $i \in P$, *the following holds: (a) if Mem*[i]*.Read() is invoked after the completion of Mem*[i]*.Write*(c)*, and returns* c'*, then* $c' = c$*; and (b) if two Mem*[i]*.Read() operations return* $c \neq \perp$ *and* $c' \neq \perp$*, then* $c = c'$.

PROOF. Recall that only p_i can write to *Mem*[i] (by invoking an *update* operation). We next show that *Mem*[i].*Write* can be invoked at most once in an execution. Suppose for the sake of contradiction that *Mem*[i].*Write* is invoked twice in the execution, and observe the second invocation. Section 5.3 mentions our assumption of a mechanism that always completes a previous operation on a weak snapshot object, if any such operation has been invoked and did not complete (because of restarts), whenever a new operation is invoked on the same weak snapshot object. Thus, when *Mem*[i].*Write* is invoked for the second time, the first *Mem*[i].*Write* has already completed. Before invoking the *Write*, p_i completes *collect*, which executes *Mem*[i].*Read*. By atomicity of *Mem*[i], since the first *Write* to *Mem*[i] has already completed writing a non-$\perp$ value, *collect* returns a set containing this value, and the condition in line 2 in Algorithm 1 evaluates to FALSE, contradicting our assumption that a *Write* was invoked after the *collect* completes.

(a) follows from atomicity of *Mem*[i] since *Mem*[i].*Write* is invoked at most once in the execution. In order to prove (b), notice that if $c \neq c'$, since p_i is the only writer of *Mem*[i], this means that both *Mem*[i].*Write*(c) and *Mem*[i].*Write*(c') are invoked in the execution, which contradicts the fact that *Mem*[i].*Write* is invoked at most once in the execution. □

The next lemma proves that Algorithm 1 preserves validity (property PR2).

LEMMA 4.2. *Let* o *be a* $scan_i()$ *operation that is invoked after the completion of an* $update_j(c)$ *operation, and that returns* C*. Then* $C \neq \emptyset$.

PROOF. Since $update_j(c)$ completes, either *Mem*[i].*Write*(c) completes or *collect* returns a non-empty set. In the first case, when o reads from *Mem*[i] during both first and second *collect*, the *Read* returns c by Lemma 4.1. The second case is that *collect* completes returning a non-empty set. Thus, a read from some register *Mem*[j] during this *collect* returns $c' \neq \perp$. By atomicity of *Mem*[j] and Lemma 4.1, since o is invoked after $update_j(c)$ completes, any read from *Mem*[j] performed during o returns c'. Thus, in both cases the first and second *collect* during o return a non-empty set, which means that $C \neq \emptyset$. □

Similarly, we next show that Algorithm 1 preserves monotonicity of scans (property PR3).

LEMMA 4.3. *Let* o *be a* $scan_i()$ *operation that returns* C *and let* o' *be a* $scan_j()$ *operation that returns* C' *and is invoked after the completion of* o*. Then* $C \subseteq C'$.

PROOF. If $C = \emptyset$, the lemma trivially holds. Otherwise, consider any $c \in C$. Notice that c is returned by a *Read* r from some register *Mem*$[k]$ during the second *collect* of o. Atomicity of *Mem*$[k]$ and Lemma 4.1 guarantee that every *Read* r' from the same register invoked after the completion of r returns c. Both times *collect* is executed during o', it reads from *Mem*$[k]$ and since o' is invoked after o completes both times a set containing c is returned from *collect*, that is, $c \in C'$. □

The key to showing non-empty intersection (property PR4) is to observe that every *scan*() operation that returns a non-empty set executes *collect* twice. Let us focus on the first *collect* that completes in the execution returning some non-empty set C and denote this *collect* by α. Notice that any *scan*() operation returning a non-empty set starts at least one *collect* after α completes. We show that this means that any value returned by α in the set C also appears in any non-empty set returned by a *scan*() in the execution, guaranteeing that such sets have a non-empty intersection.

LEMMA 4.4. *There exists c such that for every scan() operation that returns* $C' \neq \emptyset$*, it holds that* $c \in C'$.

PROOF. Let o be the first $scan_i()$ operation during which *collect* in line 7 returns a non-empty set, and let $C \neq \emptyset$ be this set. Let o' be any *scan*() operation that returns $C' \neq \emptyset$. We next show that $C \subseteq C'$, which means that any $c \in C$ preserves the requirements of the lemma. Since $C' \neq \emptyset$, the first invocation of *collect*() during o' returns a non-empty set. By definition of o, the second *collect* during o' starts after the first *collect* of o completes. For every $c \in C$, there is a *Mem*$[k]$.*Read()* executed by the first *collect* of o that returns $c \neq \perp$. By Lemma 4.1 and atomicity of *Mem*$[k]$, a *Read* from the same register performed during the second *collect* of o' returns c. Thus, $C \subseteq C'$. □

5. DYNASTORE

This section describes DynaStore, an algorithm for multi-writer multi-reader (MWMR) atomic storage in a dynamic system, which is presented in Algorithm 2. A key component of our algorithm is a procedure *ContactQ* (lines 68-80) for reading and writing from/to a quorum of members in a given view, used similarly to the *communicate* procedure in ABD [Attiya et al. 1995]. When there are no reconfigurations, *ContactQ* is invoked twice by the *read* and *write* operations – once in a read-phase and once in a write-phase. More specifically, both *read* and *write* operations first execute a read-phase, where they invoke *ContactQ* to query a quorum of the processes for the latest value and timestamp, after which both operations execute a write-phase as follows: a *read* operation invokes *ContactQ* again to write-back the value and timestamp obtained in the read-phase, whereas a *write* operation invokes *ContactQ* with a higher and unique timestamp and the desired value.

To allow reconfiguration, the members of a view also store information about the current view. They can change the view by modifying this information at a quorum of the current view. We allow the reconfiguration to occur concurrently with any *read* and *write* operations. Furthermore, once reconfiguration is done, we allow future reads and writes to use (only) the new view, so that processes can be expired and removed from the system. Hence, the key challenge is to make sure that no reads linger behind in the old view while updates are made to the new view. Atomicity is preserved using the following strategy.

—The read-phase is modified so as to first read information on reconfiguration, and then read the value and its timestamp. If a new view is discovered, the read-phase repeats with the new view.
—The write-phase, which works in the last view found by the read-phase, is modified as well. First, it writes the value and timestamp to a quorum of the view, and then, it

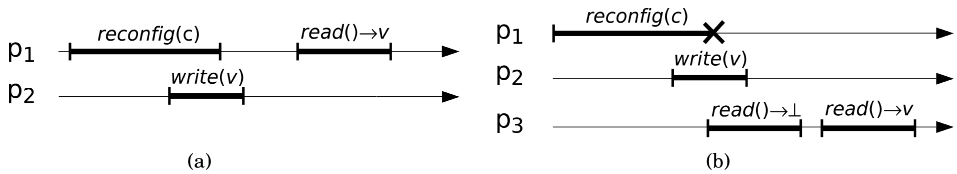

Fig. 1. Two scenarios that illustrate operation flow in DynaStore. (a) *reconfig*(c) operation from configuration c_1 to c_2 (where $c_2 = c_1 \cup c$) is concurrent with *write*(v). DynaStore ensures that one of them writes the value v in configuration c_2. (b) In this scenario, the *reconfig*(c) fails. DynaStore ensures that either the first *read*() completes in c_1, or *write*(v) writes the value v in c_2.

reads the reconfiguration information. If a new view is discovered, the protocol goes back to the read-phase (the write-phase begins again when the read-phase ends).

—The *reconfig* operation has a preliminary phase, writing information about the new view to the quorum of the old one. It then continues by executing the phases described above, starting in the old view.

The core of a read-phase is procedure *ReadInView*, which reads the configuration information (line 34) and then invokes *ContactQ* to read the value and timestamp from a quorum of the view (line 35). It returns a non-empty set if a new view was discovered in line 34. Similarly, procedure *WriteInView* implements the basic functionality of the write-phase, first writing (or writing-back) the value and timestamp by invoking *ContactQ* in line 42, and then reading configuration information in line 43 (we shall explain lines 39-40 in Section 5.3).

We next give intuition into why the above regime preserves read/write atomicity, by considering the simple case where only one reconfiguration request is ever invoked, *reconfig*(c), from c_1 to c_2 (where $c_2 = c_1 \cup c$); we shall refer to this reconfiguration operation as *RC*. Figure 1(a) depicts a scenario where *RC*, invoked by process p_1, completes while a second process p_2 concurrently performs a *write*(v) operation. In our scenario p_2 is not initially aware of the existence of c_2, and hence the *write* operation performs a write-phase W writing in c_1 the value v with timestamp *ts*. After the *write* completes, p_1 executes a *read* operation, which returns v (the only possible return value according to atomicity). The *read* operation starts with a read-phase which operates in c_2 – the latest view known to p_1. Therefore, for v to be returned by the *read*, our algorithm must make sure that v and *ts* are transferred to c_2 by either *RC* or the *write* operation.

There are two possible cases with respect to *RC*. The first case is that *RC*'s read-phase observes W, that is, during the execution of *ContactQ* in the read-phase of *RC*, p_1 receives v and *ts* from at least one process. In this case, *RC*'s write-phase writes-back v and *ts* into c_2. The second case is that *RC*'s read-phase does not observe W. In this case, as was explained previously, our algorithm must not allow the *write* operation to complete without writing the value and timestamp to a quorum of the new view c_2. We next explain how this is achieved. Since *RC*'s read-phase does not observe W, when *RC* invokes *ContactQ* during its read-phase, W's execution of *ContactQ* writing a quorum of c_1 has not completed yet. Thus, W starts to read c_1's configuration information after *RC*'s preliminary phase has completed. This preliminary phase writes information about c_2 to a majority of c_1. Therefore, W discovers c_2 and the write operation continues in c_2.

Figure 1(b) considers a different scenario, where p_1 fails before completing *RC*. Again, we assume that p_2 is not initially aware of c_2, and hence the *write* operation performs a write-phase W in c_1 writing the value v with timestamp *ts*. Concurrently with p_2's *write*, p_3 invokes a *read* operation in c_1. Atomicity of the register allows this *read* to return either v or $\perp$, the initial value of the register; in the scenario depicted in

Figure 1(b) $\perp$ is returned. After the *write* operation completes, p_3 invokes a second *read* operation, which returns v (the only possible value allowed by atomicity for this *read*). There are two cases to consider, with respect to the view in which the first read executes its final phase. The simple case is when this view is c_1. Then, the second *read* starts by executing a read-phase in c_1 and hence finds out about v.

The second case is more delicate, and it occurs when the first *read* completes in c_2. Recall that this *read* returns $\perp$ and thus it does not observe W and the latest value v. Nevertheless, since the second *read* starts with a read-phase in c_2, the algorithm must ensure that v is stored at a quorum of c_2. This is done by the *write* operation, as we now explain. Since the first *read* operation starts in c_1 but completes in c_2, it finds c_2 when reading the reconfiguration information during a read-phase R in c_1. Since R does not observe W, it must be that W completes its *ContactQ* writing a majority of c_1 only after R invokes its *ContactQ* reading from a majority of c_1. Since R inspects reconfiguration information *before* invoking *ContactQ* while W does so *after* completing *ContactQ*, it must be that W starts inspecting reconfiguration information after R has finished inspecting reconfiguration information. Monotonicity of scans (property PR3) guarantees that W finds all configuration changes observed by R, and hence finds out about c_2. Consequently, the *write* operation continues in c_2 and completes only after writing v in c_2. Here, it is important that the read-phase reads reconfiguration information *before* it performs *ContactQ*, while the write-phase reads reconfiguration information *after* it performs *ContactQ*. This inverse order is necessary to ensure atomicity in this scenario.

In these examples, additional measures are needed to preserve atomicity if several processes concurrently propose changes to c_1. Thus, the rest of our algorithm is dedicated to the complexity that arises due to multiple contending reconfiguration requests. Our description is organized as follows: Section 5.1 introduces the pseudo-code of DynaStore, and clarifies its notations and atomicity assumptions. Section 5.2 presents the DAG of views, and shows how every operation in DynaStore can be seen as a traversal on that graph. Section 5.3 discusses *reconfig* operations. Section 5.4 presents the notion of established views, which is central to the analysis of DynaStore. Formal proofs are given in Section 5.5.

5.1. DynaStore Basics

DynaStore uses *operations*, *upon clauses*, and *procedures*. Operations are invoked by the application, whereas upon-clauses are triggered by messages received from the network: whenever a process p_i receives a message m from p_j (through a $receive(m)_{i,j}$ input action), m is stored in a buffer (this is not showed in the pseudo-code) and the upon-clause is an internal action enabled when some condition on the message buffer holds. Procedures are called from an operation. Operations and local variables at process p_i are denoted with subscript i.

Whereas upon-clauses are atomic, for simplicity of presentation, we do not formulate operations as atomic actions in the pseudo-code (with slight abuse of the I/O automata terminology), and operations sometimes block waiting for a response from a majority of processes in a view (in lines 31, 75, 54, 34 and 43), either explicitly (in lines 31 and 75), or in the underlying implementation of a SWMR register (e.g., Attiya et al. [1995]) which is used in the construction of weak-snapshots. Note, however, that it is a trivial exercise to convert the pseudo-code to the I/O automata syntax, as each operation is atomic until it blocks waiting for a majority and thus the operation can be devided into multiple atomic actions: initially an action corresponding to the code that precedes the *wait* statenent executes, and when messages are received from a majority, the upon-clause receiving the messages uses an additional internal flag to enable the execution of the operation part following the *wait*, which forms another atomic action, and disable code which precedes the *wait*.

Operations and upon-clauses access different variables for storing the value and timestamp[6]: v_i and ts_i are accessed in upon-clauses, whereas operations (and procedures) manipulate v_i^{max} and ts_i^{max}. Procedure *ContactQ* sends a write-request including v_i^{max} and ts_i^{max} (line 72) when writing a quorum, and a read-request (line 74) when reading a quorum ($msgNum_i$, a local sequence number, is also included in such messages). When p_i receives a write-request, it updates v_i and ts_i if the received timestamp is bigger than ts_i, and sends back a REPLY message containing the sequence number of the request (line 86). When a read-request is received, p_i replies with v_i, ts_i, and the received sequence number (line 88).

Every process p_i executing DynaStore maintains a local estimation of the latest view, $curView_i$ (line 9), initialized to *Init* when the process starts. If the number of changes proposed in the execution is finite, such estimates will eventually become the same at all active processes, however the estimates may otherwise diverge as we shall see below. Although p_i is able to execute all event-handlers immediately when it starts, recall that invocations of *read*, *write* or *reconfig* operations at p_i are only allowed once they are enabled for the first time; this occurs in line 11 (for processes in *Init.join*) or in line 96 (for processes added later). If p_i discovers that it is being removed from the system, it simply halts (line 52). In this section, we denote changes of the form (Add, i) by $(+, i)$ and changes of the form $(Remove, i)$ by $(-, i)$.

5.2. Traversing the Graph of Views

Weak snapshots organize all views into a DAG, where views are the vertices and there is an edge from a view w to a view w' whenever an $update_j(w, c)$ has been made in the execution by some process $j \in w.members$, updating $ws(w)$ to include the change $c \neq \emptyset$ such that $w' = w \cup c$; $|c|$ can be viewed as the weight of the edge – the distance between w' and w in changes. Our algorithm maintains the invariant that $c \cap w = \emptyset$ (Lemma 5.3 in Section 5.5), and thus w' always contains more changes than w, that is, $w \subset w'$. Hence, the graph of views is acyclic.

The main logic of DynaStore lies in procedure *Traverse*, which is invoked by all operations. This procedure traverses the DAG of views, and transfers the state of the emulated register from view to view along the way. *Traverse* starts from the view $curView_i$. Then, the DAG is traversed in an effort to find all membership changes in the system; these are collected in the set *desiredView*. After finding all changes, *desiredView* is added to the DAG by updating the appropriate *ws* object, so that other processes can find it in future traversals.

The traversal resembles the well-known Dijkstra algorithm for finding shortest paths from some single source [Cormen et al. 1990], with the important difference that our traversal modifies the graph. A set of views, *Front*, contains the vertices reached by the traversal and whose outgoing edges were not yet inspected. Initially, $Front = \{curView_i\}$ (line 48). Each iteration processes the vertex w in *Front* closest to $curView_i$ (lines 50 and 51).

During an iteration of the loop in lines 49–64, it might be that p_i already knows that w does not contain all proposed membership changes. This is the case when *desiredView*, the set of changes found in the traversal, is different from w. In this case, p_i installs an edge from w to *desiredView* using $update_i$ (line 54). As explained in Section 4, in case another update to $ws(w)$ has already completed, *update* does not install an additional edge from w; the only case when multiple outgoing edges exist is if they were installed concurrently by different processes.

[6]This allows for a practical optimization, whereby operations and upon clauses act like separate monitors: an operation can execute concurrently with an upon-clause, and at most one of each kind can be executed at a time.

ALGORITHM 2: Code for process p_i, part 1

```
state
  v_i ∈ V ∪ {⊥}, initially ⊥                              // latest value received in a WRITE message
  ts_i ∈ ℕ_0 × (Π ∪ {⊥}), initially (0, ⊥)                // timestamp corresponding to v_i (timestamps have
    selectors num and pid)
  v_i^max ∈ V ∪ {⊥}, initially ⊥                          // latest value observed in Traverse
  ts_i^max ∈ ℕ_0 × (Π ∪ {⊥}), initially (0, ⊥)            // timestamp corresponding to v_i^max
  pickNewTS_i ∈ {FALSE, TRUE}, initially FALSE            // should Traverse pick a new timestamp?
  M_i: set of messages, initially ∅
  msgNum_i ∈ ℕ_0, initially 0                             // counter for sent messages
  curView_i ∈ Views, initially Init                       // latest view

initially:
  if (i ∈ Init.join) then enable operations

operation read_i():
  pickNewTS_i ← FALSE
  newView ← Traverse(∅, ⊥)
  NotifyQ(newView)
  return v_i^max

operation write_i(v):
  pickNewTS_i ← TRUE
  newView ← Traverse(∅, v)
  NotifyQ(newView)
  return OK

operation reconfig_i(cng):
  pickNewTS_i ← FALSE
  newView ← Traverse(cng, ⊥)
  NotifyQ(newView)
  return OK

procedure NotifyQ(w)
  if did not receive ⟨NOTIFY, w⟩ then
    send ⟨NOTIFY, w⟩ to w.members
  end if
  wait for ⟨NOTIFY, w⟩ from
    a majority of w.members
end

procedure ReadInView(w)
  ChangeSets ← scan_i(w)
  ContactQ(R, w.members)
  return ChangeSets
end

procedure WriteInView(w, v)
  if pickNewTS_i then
    (pickNewTS_i, v_i^max, ts_i^max) ←
        (FALSE, v, (ts_i^max.num + 1, i))
  end if
  ContactQ(W, w.members)
  ChangeSets ← scan_i(w)
  return ChangeSets
end

procedure Traverse(cng, v)
  desiredView ← curView_i ∪ cng
  Front ← {curView_i}
  do
    s ← min{|ℓ| : ℓ ∈ Front}
    w ← any ℓ ∈ Front s.t. |ℓ| = s
    if (i ∉ w.members) then halt_i
    if w ≠ desiredView then
      update_i(w, desiredView \ w)
    end if
    ChangeSets ← ReadInView(w)
    if ChangeSets ≠ ∅ then
      Front ← Front \ {w}
      for each c ∈ ChangeSets
        desiredView ← desiredView ∪ c
        Front ← Front ∪ {w ∪ c}
    end if
    else ChangeSets ← WriteInView(w, v)
  while ChangeSets ≠ ∅
  curView_i ← desiredView
  return desiredView
end

procedure ContactQ(msgType, D)
  M_i ← ∅
  msgNum_i ← msgNum_i + 1
  if msgType = W then send
    ⟨REQ, W, msgNum_i, v_i^max, ts_i^max⟩ to D
  else send
    ⟨REQ, R, msgNum_i, ⊥, (0, ⊥)⟩ to D
  wait until ⟨REPLY, msgNum_i, ···⟩ is
    in M_i from a majority of D
  if msgType = R then
    tm ← maximal timestamp t s.t.
        ⟨REPLY, msgNum_i, v, t⟩ is in M_i
    vm ← value corresponding to tm
    if tm > ts_i^max then
      (v_i^max, ts_i^max) ← (vm, tm)
    end if
end
```

ALGORITHM 3: Code for process p_i, part 2

```
83: upon receiving ⟨REQ, msgType, num, v, ts⟩ from p_j:
84:    if msgType = W then
85:       if (ts > ts_i) then (v_i, ts_i) ← (v, ts)
86:       send ⟨REPLY, num⟩ to p_j
87:    end if
88:    else send message ⟨REPLY, num, v_i, ts_i⟩ to p_j

89: upon receiving ⟨REPLY, ···⟩:
90:    add the message and its sender-id to M_i

91: upon receiving ⟨NOTIFY, w⟩ for the first time:
92:    send ⟨NOTIFY, w⟩ to w.members
93:    if (curView_i ⊂ w) then
94:       pause any ongoing Traverse
95:       curView_i ← w
96:       if (i ∈ w.join) then enable operations
97:       if paused in line 94, restart Traverse from line 47
98:    end if
```

Next, p_i invokes *ReadInView*(w) (line 56), which reads the state and configuration information in this view, returning all edges outgoing from w found when scanning $ws(w)$ in line 34. By validity (property PR2), if p_i or another process had already installed an edge from w, a non-empty set of edges is returned from *ReadInView*. If one or more outgoing edges are found, w is removed from *Front*, the next views are added to *Front*, and the changes are added to *desiredView* (lines 59–61). If p_i does not find outgoing edges from w, it invokes *WriteInView*(w) (line 63), which writes the latest known value to this view and again scans $ws(w)$ in line 43, returning any outgoing edges that are found. If here too no edges are found, the traversal completes.

Notice that *desiredView* is chosen in line 51 only when there are no other views in *Front*, since it contains the union of all views observed during the traversal (Lemma 5.2), and thus any other view in *Front* must be of smaller size (i.e., contain fewer changes). Moreover, when $w \neq desiredView$ is processed, the condition in line 53 evaluates to true, and *ReadInView* returns a non-empty set of changes (outgoing edges) by validity (property PR2). Thus, *WriteInView*$(w, *)$ is invoked only when *desiredView* is the only view in *Front*, that is, $w = desiredView$ (this transfers the state found during the traversal to *desiredView*, the latest-known view). For the same reason, when the traversal completes, $Front = \{desiredView\}$ (Lemma 5.6). Then, *desiredView* is assigned to $curView_i$ in line 65 and returned from Traverse.

To illustrate such traversals, consider the example in Figure 2. Process p_i invokes *Traverse* and let *initView* be the value of $curView_i$ when *Traverse* is invoked. Assume that *initView.members* includes at least p_1 and p_i, and that $cng = \emptyset$ (this parameter of *Traverse* will be explained in Section 5.3). Initially, its *Front*, marked by a rectangle in Figure 2, includes only *initView*, and $desiredView = initView$. Then, the condition in line 53 evaluates to false and p_i invokes *ReadInView*(*initView*), which returns $\{\{(+, 3)\}, \{(+, 5)\}, \{(-, 1), (+, 4)\}\}$. Next, p_i removes *initView* from *Front* and adds vertices V_1, V_2 and V_3 to *Front* as shown in Figure 2. For example, V_3 results from adding the changes in $\{(-, 1), (+, 4)\}$ to *initView*. At this point, $desiredView = initView \cup \{(+, 3), (+, 5), (-, 1), (+, 4)\}$. In the next iteration of the loop in lines 49–64, one of the smallest views in *Front* is processed. In our scenario, V_1 is chosen. Since $V_1 \neq desiredView$, p_i installs an edge from V_1 to *desiredView*. Suppose that no other updates were made to $ws(V_1)$ before p_i's update completes. Then, *ReadInView*(V_1) returns $\{\{(+, 5), (-, 1), (+, 4)\}\}$ (integrity and validity properties of weak snapshots). Then, V_1 is removed from *Front* and

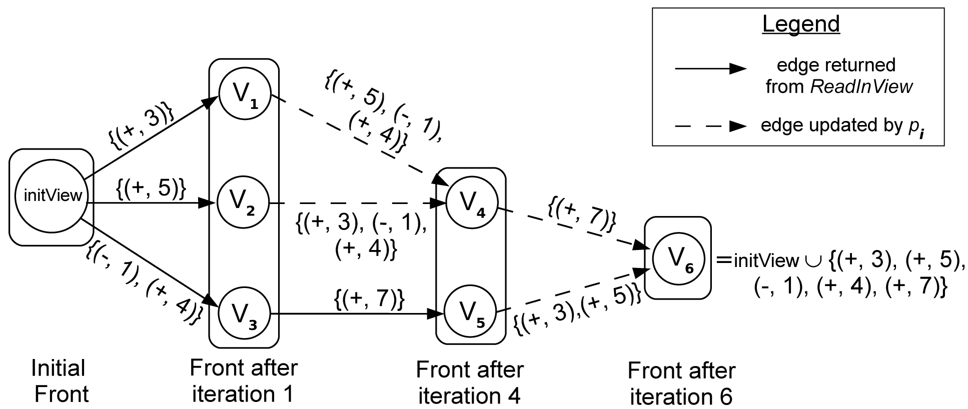

Fig. 2. Example DAG of views.

$V_4 = V_1 \cup \{(+, 5), (-, 1), (+, 4)\}$ is added to *Front*. In the next iteration, an edge is installed from V_2 to V_4 and V_2 is removed from *Front*.

Now, the size of V_3 is smallest in *Front*, and suppose that another process p_j has already completed $update_j(V_3, \{(+, 7)\})$. p_i executes *update* (line 54), however since an outgoing edge already exists, a new edge is not installed. Then, $ReadInView(V_3)$ is invoked and returns $\{\{(+, 7)\}\}$. Next, V_3 is removed from *Front*, $V_5 = V_3 \cup \{(+, 7)\}$ is added to *Front*, and $(+, 7)$ is added to *desiredView*. Now, $Front = \{V_4, V_5\}$, and we denote the new *desiredView* by V_6. When V_4 and V_5 are processed, p_i installs edges from V_4 and V_5 to V_6. Suppose that *ReadInView* of V_4 and V_5 in line 56 return only the edge installed in the preceding line. Thus, V_4 and V_5 are removed from *Front*, and V_6 is added to *Front*, resulting in $Front = \{V_6\}$. During the next iteration $ReadInView(V_6)$ and $WriteInView(V_6)$ execute and both return $\emptyset$ in our execution. The condition in line 64 terminates the loop, V_6 is assigned to $curView_i$ and Traverse completes returning V_6.

5.3. Reconfigurations (Liveness)

A *reconfig(cng)* operation is similar to a *read*, with the only difference that *desiredView* initially contains the changes in *cng* in addition to those in $curView_i$ (line 47). Since *desiredView* only grows during a traversal, this ensures that the view returned from *Traverse* includes the changes in *cng* (Lemma 5.7 in Section 5.5). As explained earlier, $Front = \{desiredView\}$ when *Traverse* completes (Lemma 5.6), which means that *desiredView* appears in the DAG of views.

When a process p_i completes *WriteInView* in line 63 of *Traverse*, the latest state of the register has been transfered to *desiredView*, and thus it is no longer necessary for other processes to start traversals from earlier views. Thus, after Traverse completes returning *desiredView*, p_i invokes *NotifyQ* with this view as its parameter (lines 15, 20 and 25), to let other processes know about the new view. $NotifyQ(w)$ sends a NOTIFY message (line 29) to *w.members*. A process receiving such a message for the first time forwards it to all processes in *w.members* (line 92), and after a NOTIFY message containing the same view was received from a majority of *w.members*, *NotifyQ* returns. In addition to forwarding the message, a process p_j receiving $\langle$NOTIFY, $w\rangle$ checks whether $curView_j \subset w$ (i.e., w is more up-to-date than $curView_j$), and if so it pauses any ongoing *Traverse*, assigns w to $curView_j$, and restarts *Traverse* from line 47. As the execution of *Traverse* between *wait* statements is atomic, *Traverse* executed by p_j can be restarted only when it blocks waiting for messages from a majority of some view w'. Restarting *Traverse* in such case can be necessary if less than a majority of members in w' are active. Intuitively, Definition 3.2 implies that in such case w' must be an old view, that is, some *reconfig* operation completes proposing new changes to system membership. Lemma 5.26 proves that in this case p_j will receive a ⟨NOTIFY, w⟩ message such that $curView_j \subset w$ and restart its traversal (provided, of course, that p_j has not been removed, that is, that it belongs to *w.members*). We show in Theorem 5.28(a) that such NOTIFY messages also ensure that *enable operations* event occurs at every active process that was added by a complete *reconfig* operation, as required by Definition 3.2.

Note that when a process p_i restarts *Traverse*, p_i may have an outstanding $scan_i$ or $update_i$ operation on a weak snapshot $ws(w)$ for some view w, in which case p_i restarts *Traverse* without completing the operation. It is possible that p_i might be unable to complete such outstanding operations because w is an old view, that is, more than a majority of its members were removed. After *Traverse* is restarted, it is possible that p_i encounters w again in the traversal and needs to invoke another operation on $ws(w)$, in which case w is not known to be old. We require that in this case p_i first terminates previous outstanding operations on $ws(w)$ before it invokes the new operation. The mechanism to achieve this is a simple queue, and it is not illustrated in the code. Note that started snapshot operations on old views do not need to be completed.

Restarts of *Traverse* introduce an additional potential complication for *write* operations: suppose that during its execution of $write(v)$, p_i sends a WRITE message with v and a timestamp *ts*. It is important that if *Traverse* is restarted, v is not sent with a different timestamp (unless it belongs to some other write operation). Before the first message with v is sent, we set the $pickNewTS_i$ flag to *false* (line 40). The condition in line 39 prevents *Traverse* from re-assigning v to v_i^{max} or incorrect ts_i^{max}, even if a restart occurs.

In Section 5.5.3, we prove that DynaStore preserves Dynamic Service Liveness (Definition 3.2). Thus, liveness is conditioned on the number of different changes proposed in the execution being finite (in reality, liveness would still hold even with an infinite number of reconfigurations, provided that each operation is concurrent with a finite number of reconfigurations). Using this assumption, we prove in Theorem 5.28(b) that from some point of the execution onward no more ⟨NOTIFY, *newView*⟩ messages can be received by a process p_i that can cause the restart of *Traverse*, that is, such that $curView_i \subset newView$. Lemma 5.27 proves that, if p_i is active, *Traverse* and the operation during which it was invoked will then terminate.

5.4. Sequence of Established Views (Safety)

Our traversal algorithm performs a $scan(w)$ to discover outgoing edges from w. However, different processes might invoke $update(w)$ concurrently, and different *scan*s might see different sets of outgoing edges. In such cases, it is necessary to prevent processes from working with views on different branches of the DAG. Specifically, we would like to ensure an intersection between views accessed in reads and writes. Fortunately, non-empty intersection (property PR4) guarantees that all $scan(w)$ operations that return non-empty sets (i.e., return some outgoing edges from w), have at least one element (edge) in common. Note that a process cannot distinguish such an edge from others and therefore traverses all returned edges. This property of the algorithm enables us to define a totally ordered subset of the views, which we call *established*, as follows:

Definition 5.1 (*Sequence of Established Views*). The unique sequence of established views $\mathcal{E}$ is constructed as follows:

—the first view in $\mathcal{E}$ is the initial view *Init*;
—if w is in $\mathcal{E}$, then the next view after w in $\mathcal{E}$ is $w' = w \cup c$, where c is an element chosen arbitrarily from the intersection of all sets $C \neq \emptyset$ returned by some $scan(w)$ operation in the execution.

Note that each element in the intersection mentioned in Definition 5.1 is a set of changes, and that property PR4 guarantees a non-empty intersection. In order to find such a set of changes c in the intersection, one can take an arbitrary element from the set C returned by the first *collect*(w) that returns a non-empty set in the execution. This unique sequence $\mathcal{E}$ allows us to define a total order relation on established views. For two established views w and w' we write $w \dot{\leq} w'$ if w appears in $\mathcal{E}$ no later than w'; if in addition $w \neq w'$ then $w \dot{<} w'$. Notice that for two established views w and w', $w \dot{<} w'$ if an only if $w \subset w'$.

Notice that the first graph traversal in the system starts from $curView_i = Init$, which is established by definition. When *Traverse* is invoked with an established view $curView_i$, every time a vertex w is removed from *Front* and its children are added, one of the children is an established view, by definition. Thus, *Front* always includes at least one established view, and since it ultimately contains only one view, *desiredView*, we conclude that *desiredView* assigned to $curView_i$ in line 65 and returned from *Traverse* is also established (Lemma 5.8). Thus, all views sent in NOTIFY messages or stored in $curView_i$ are established. Note that while a process p_i encounters all established views between $curView_i$ and the returned *desiredView* in an uninterrupted traversal, it only recognizes a subset of established views as such (whenever *Front* contains a single view, that view must be in $\mathcal{E}$).

We show that *WriteInView* (line 63) is always performed in an established view (Lemma 5.8). Moreover, we prove that each traversal performs a *ReadInView* on every established view in $\mathcal{E}$ between $curView_i$ and the returned view *desiredView* (Lemma 5.9(a)). Thus, intuitively, by reading each view encountered in a traversal, we are guaranteed to intersect any write completed on some established view in the traversed segment of $\mathcal{E}$.

By performing the *scan* before *ContactQ* in *ReadInView* and after the *ContactQ* in *WriteInView* we guarantee that in this intersection the state is transferred correctly as we now explain. This interleaving of snapshot and data operations guarantees that for any $W = WriteInView$ and $R = ReadInView$ that operate on the same view w, either the maximal timestamp found by R is at least as high as the one written by W, that is, R reads the data written by W or some newer data, or the set of changes returned by R is contained in the set returned by W, that is, W sees at least all those outgoing edges from the view w that R sees. Moreover, if R is invoked after W then the former necessarily holds. This property is proven in Lemma 5.10 of Section 5.5 and we refer the reader to the beginning of Section 5 for examples that illustrate its usefulness. Our proof uses this property in Lemma 5.11 to show that data read from a view w' will be at least as new as data read from a view w if both views are established and $w \dot{<} w'$, which means, intuitively, that state is transfered correctly along the sequence of established views.

To facilitate the proof of atomicity we associate a timestamp *ats(o)* with each read or write operation o. If o is a *read*, then $ats(o)$ is ts_i^{max} upon the completion of Traverse during o; if o is a *write*, then $ats(o)$ equals to ts_i^{max} when its assignment completes in line 40. Although this definition does not associate a timestamp with every *read* or *write* operation, Lemma 5.16 shows that timestamps are well defined for all complete *read*s and *write*s. It also proves that for every *read* there is a corresponding *write* of the returned value that has the same associated timestamp as the *read*, and that the timestamps associated with different *write*s are different. Lemma 5.17 proves that if o and o' are two complete *read* or *write* operations such that o completes before o' is invoked, then $ats(o) \leq ats(o')$ and if o' is a *write* operation, then $ats(o) < ats(o')$. Theorem 5.18 completes the proof of linearizability (atomicity) by using associated timestamps to construct for every execution a serial history equivalent to the history of the execution.

5.5. Correctness of DynaStore

5.5.1. Traverse. We use the convention whereby each time Traverse is restarted, a new execution of Traverse begins; this allows us to define one view from which a traversal starts – this is the value $curView_i$ when the execution of Traverse begins in line 47.

We note that whenever a process p_i performs $scan_i(w)$ or $update_i(w, c)$, it holds that $i \in w.members$ because of the check in line 52. Thus, it is allowed to perform these operations on w.

LEMMA 5.2. *At the beginning and end of each iteration of the loop in lines* 49-64*, it holds that* $\bigcup_{w \in Front} w \subseteq desiredView$.

PROOF. We prove that if an iteration begins with $\bigcup_{w \in Front} w \subseteq desiredView$ then this invariant is preserved also when the iteration ends. The lemma then follows from the fact that at the beginning of the first iteration $Front = \{curView_i\}$ (line 48) and $desiredView = curView_i \cup cng$ (line 47).

Suppose that at the beginning of an iteration $\bigcup_{w \in Front} w \subseteq desiredView$. If the loop in lines 59-61 does not execute, then Front and $desiredView$ do not change, and the condition is preserved at the end of the iteration. If the loop in lines 59-61 does execute, then $w \subseteq desiredView$ is removed from Front, $w \cup c$ is added to Front and c is added to $desiredView$, thus the condition is again preserved. □

LEMMA 5.3. *Whenever* $update_i(w, c)$ *is executed,* $c \neq \emptyset$ *and* $c \cap w = \emptyset$.

PROOF. $update_i(w, c)$ is executed only in line 54 when $w \neq desiredView$ and $c = desiredView \setminus w$, which means that $c \cap w = \emptyset$. By Lemma 5.2, since $w \neq desiredView$, it holds that $w \subset desiredView$. Thus, $c = desiredView \setminus w \neq \emptyset$. □

LEMMA 5.4. *Let* T *be an execution of Traverse that starts from* $curView_i = initView$. *For every view* w *that appears in Front at some point during the execution of* T*, it holds that* $initView \subseteq w$.

PROOF. We prove that if an iteration of the loop in lines 49-64 begins such that each view in *Front* contains *initView*, then this invariant is preserved also when the iteration ends. The lemma then follows from the fact that at the beginning of the first iteration $Front = \{curView_i\}$ (line 48).

Suppose that at the beginning of an iteration each view in *Front* contains *initView*. *Front* can only change during this iteration if the condition in line 57 evaluates to true, that is, if $ChangeSets \neq \emptyset$. In this case, the loop in lines 59-61 executes at least once, and $w \cup c$ is added to *Front* in line 61 for some c. Since w was in *Front* in the beginning of this iteration, by our assumption it holds that $initView \subseteq w$, and therefore $w \cup c$ also contains $initView$. □

LEMMA 5.5. *Let* $w \in Front$ *be a view. During the execution of Traverse, if* w *is removed from Front in some iteration of the loop in lines* 49-64*, then the size of any view* w' *added to Front in the same or a later iteration, is bigger than* $|w|$.

PROOF. Suppose that w is removed from *Front* during an iteration. Then its size, $|w|$, is minimal among the views in *Front* (lines 50 and 51) at the beginning of this iteration. By line 61, whenever a view is inserted to *Front*, it has the form $w \cup c$ where $c \in ChangeSets$ returned by $scan_i$ in line 34. By property PR1, some $update(w, c)$ operation is invoked in the execution, and by Lemma 5.3, $c \neq \perp$ and $c \cap w = \emptyset$. Thus, the view $w \cup c$ is strictly bigger than w removed from *Front* in the same iteration. It follows that any view w' added to *Front* in this or in a later iteration has size bigger than $|w|$. □

LEMMA 5.6. *If at some iteration of the loop in lines* 49-64 *ReadInView returns ChangeSets* $= \emptyset$*, then* $w =$ *desiredView and Front* = {*desiredView*}.

PROOF. Suppose for the sake of contradiction that $w \neq$ *desiredView*. Before *ReadInView* is invoked, $update_i(w, desiredView \setminus w)$ completes, and then, by Lemma 4.2, when *ReadInView* completes it returns a non-empty set, a contradiction.

Suppose for the sake of contradiction that there exists a view $w' \in$ Front such that $w' \neq$ *desiredView*. By Lemma 5.2, $w' \subseteq$ *desiredView*. Since $w' \neq$ *desiredView*, we get that $w' \subset$ *desiredView* and thus $|w'| < |desiredView|$, contradicting the fact that $w =$ *desiredView*, and not w', is chosen in line 51 in the iteration. □

LEMMA 5.7. *desiredView returned from Traverse contains cng.*

PROOF. At the beginning of Traverse, *desiredView* is set to $curView_i \cup cng$ in line 47, and during the execution of Traverse, no element is removed from *desiredView*. Thus, $cng \subseteq desiredView$ when Traverse completes. □

LEMMA 5.8. *$curView_i$ is an established view. Moreover, desiredView in line 65 of Traverse is established and whenever WriteInView(w, $*$) is invoked, w is an established view.*

PROOF. We prove the lemma using the following claim:

CLAIM 5.8.1. *If $curView_i$ from which a traversal starts is an established view, then Front at the beginning and end of the loop in lines* 49-64 *contains an established view, and the view desiredView assigned to $curView_i$ in line* 65 *in Traverse is established. Moreover, whenever WriteInView(w, $*$) is invoked, w is an established view.*

PROOF. Initially, *Front* contains $curView_i$ (line 47), which is established by assumption, and therefore *Front* indeed contains an established view when the first iteration of the loop begins. If a view w is removed from *Front* in line 58, then *ChangeSets* $\neq \emptyset$. We distinguish between two cases: (1) if w is not an established view, then Front at the end of the iteration still contains an established view; (2) if w is an established view, then, by Lemma 4.4 and the definition of $\mathcal{E}$, since *ChangeSets* is a non-empty set returned by $scan_i(w)$, there exists $c \in$ *ChangeSets* such that $w \cup c$ is established. Since for every $c \in$ *ChangeSets*, $w \cup c$ is added to *Front* in line 61, the established view succeeding w in the sequence is added to *Front*, and thus *Front* at the end of this iteration of the loop in lines 49-64 still contains an established view.

By Lemma 5.6, when the loop in lines 49-64 completes, as well as when *WriteInView*(w, $*$) is invoked, Front = {*desiredView*}. Since during such iterations, *ReadInView* returns $\emptyset$, *Front* does not change from the beginning of the iteration. We have just shown that *Front* contains an established view at the beginning of the do-while loop, and thus, *desiredView* in line 65 is established, and so is any view w passed to *WriteInView*. □

We next show that the precondition of the claim above holds, that is, that $curView_i$ is an established view, by induction on $|curView_i|$. The base is $curView_i = Init$, in which case it is established by definition. Assuming that $curView_i$ is established if its size is less than k, observe such view of size $k > |Init|$. Consider how $curView_i$ got its current value – it was assigned either by some earlier execution of Traverse at p_i in line 65, or in line 95 when a NOTIFY message is received, which implies that some process completes a traversal returning this view. In either case, since $curView_i \neq Init$, some process p_j has $desiredView = curView_i$ in line 65, while starting the traversal with a smaller view $curView_j$. Notice that $curView_j$ is established by our induction assumption, and since $curView_i$ is the value of *desiredView* in line 65 of a Traverse that started with an established view, it is also established by Claim 5.8.1.

LEMMA 5.9. *Let T be an execution of Traverse and initView be the value of curView$_i$ when p_i starts this execution, then (a) if T invokes WriteInView$(w, *)$ then T completes a ReadInView(w') which returns a non-empty set for every established view w' such that initView $\leq w' \lessdot w$, and a ReadInView(w) that returns $\emptyset$; and (b) if T reaches line 65 with desiredView $= w''$, then it completes WriteInView$(w'', *)$ which returns $\emptyset$.*

PROOF. When T begins, the established view $w' = initView$ is the only view in Front. Since some iteration during T chooses w in lines 50 and 51, which has bigger size than w', it must be that w' is removed from Front. This happens only if some *ReadInView*(w') during T returns *ChangeSets* $\neq \emptyset$. After w' is removed from Front, for every $c \in$ *ChangeSets*, $w' \cup c$ is added to Front, and thus, the established view succeeding w' in $\mathcal{E}$ is added to *Front* (by Lemma 4.4 and the definition of $\mathcal{E}$). The arguments above hold for every established view w' such that *initView* $\leq w' \lessdot w$, since a bigger view w is chosen from Front during T. During the iteration when *WriteInView*$(w, *)$ is invoked, ReadInView(w) completes in line 56 and returns $\emptyset$, which completes the proof of (a).

Suppose that T reaches line 65 with *desiredView* $= w''$. By Lemma 5.6, w during the last iteration of the loop equals to w''. Observe the condition in line 64, which requires that *ChangeSets* $= \emptyset$ for the loop to end. Notice that *ChangeSets* is assigned either in line 56 or line 63. If it was assigned in line 63, then *WriteInView*$(w, *)$ was executed which completes the proof of (b). Otherwise, *ReadInView*(w) returns *ChangeSets* $= \emptyset$ in line 56, which causes line 63 to be executed. Then, since this is the last iteration, *WriteInView*$(w, *)$ returns $\emptyset$. □

5.5.2. Atomicity. We say that *WriteInView* writes a timestamp *ts* if ts_i^{max} sent in the REQ message by ContactQ(W, *) equals *ts*. Similarly, a *ReadInView* reads timestamp *ts* if at the end of ContactQ(R, *) invoked by the *ReadInView*, ts_i^{max} is equal to *ts*.

LEMMA 5.10. *Let W be a WriteInView(w, *) that writes timestamp ts and returns C, and R be a ReadInView(w) that reads timestamp ts′ and returns C′. Then, either $ts' \geq ts$ or $C' \subseteq C$. Moreover, if R is invoked after W completes, then $ts' \geq ts$.*

PROOF. Because both operation invoke ContactQ in w, there exists a process p in *w.members* from which both W and R get a REPLY message before completing their ContactQ, that is, p's answer counts towards the necessary majority of replies for both W and R. If p receives the $\langle$REQ, W, ...$\rangle$ message from W with timestamp *ts* before the $\langle$REQ, R, ...$\rangle$ message from R, then by lines 85 and 88 it responds to the message from R with a timestamp at least as big as *ts*. By lines 77-80, when R completes *ContactQ*(R, *w.members*), ts_i^{max} is set to be at least as high as *ts*, and thus $ts' \geq ts$. It is left to show that if p receives the $\langle$REQ, R, ...$\rangle$ message from R before the $\langle$REQ, W, ...$\rangle$ message from W, then $C' \subseteq C$.

Suppose that p receives the $\langle$REQ, R, ...$\rangle$ message from R first. Then, when this message is received by p, *ContactQ*(W, *w.members*) has not yet completed at W, and thus W has not yet invoked *scan*(w) in line 43. On the other hand, since R has started *ContactQ*(R, *w.members*), it has already completed its *scan*(w) in line 34, which returned C'. When W completes its ContactQ it invokes *scan*(w), which then returns C. By Lemma 4.3, it holds that $C' \subseteq C$.

Notice that if R is invoked after W completes then it must be the case that p receives the $\langle$REQ, W, ...$\rangle$ message from W first, and thus, in this case, $ts' \geq ts$. □

LEMMA 5.11. *Let T be an execution of Traverse that completes returning w and upon completion its ts_i^{max} is equal to ts, and T' be an execution of Traverse that reaches line 65 with ts_i^{max} equal to ts′ and its desiredView equal to w'. If $w \lessdot w'$, then $ts \leq ts'$.*

PROOF. Consider the prefix of $\mathcal{E}$ up to w': $V_0, V_1, \ldots, V_l$ such that $V_0 = \mathit{Init}$, $V_l = w'$, and $w = V_i$ where $i \in \{0, \ldots, l-1\}$. Moreover, let w'' be the view from which T' starts the traversal (w'' is established by Lemma 5.8).

First, consider the case that $w'' \dot{\leq} w$. By Lemma 5.9, since T returns w, it completes $\mathit{WriteInView}(w, *)$ which returns $C = \emptyset$. Since T' starts from $w'' \dot{\leq} w$ and reaches line 65 with $\mathit{desiredView} = w'$ such that $w \dot{<} w'$, by Lemma 5.9 it completes a $\mathit{ReadInView}(w)$ that returns $C' \neq \emptyset$ (notice that $\mathit{ReadInView}(w)$ might be executed in two consecutive iterations of T', in which case during the first iteration $\mathit{ReadInView}(w)$ returns $\emptyset$; we then look on the next iteration, where a non-empty set is necessarily returned). Since $C' \not\subseteq C$, by Lemma 5.10, we have that ts_i^{max} upon the completion of the $\mathit{ReadInView}(w)$ by T' is at least as big as ts_i^{max} upon the completion of $\mathit{WriteInView}(w, *)$ by T, which equals to ts. Since ts_i^{max} does not decrease during T' and ts' is the value of ts_i^{max} when T' reaches line 65, we have that $ts' \geq ts$.

The second case to consider is $w \dot{<} w''$, which implies that $w'' \neq \mathit{Init}$. In this case, there exists a traversal T'' that starts from a view $w''' \dot{<} w''$ and reaches line 65 before T begins, with $\mathit{desiredView} = w''$ (T'' is either an earlier execution of Traverse by the same process that executes T', or by another process, in which case T'' completes and sends a NOTIFY message with w'' which is then received by the process executing T' before T' starts). Let ts'' be the ts_i^{max} when T'' reaches line 65. Notice that T'' completes $\mathit{WriteInView}(w'', *)$ before T' starts $\mathit{ReadInView}(w'')$, and by Lemma 5.10 when $\mathit{ReadInView}(w'')$ completes at T' its ts_i^{max} is at least ts''. Since ts_i^{max} at T' can only increase from that point on, we get that $ts' \geq ts''$. It is therefore enough to show that $ts'' \geq ts$ in order to complete the proof. In order to do this, we apply the arguments above recursively, considering T'' instead of T', w'' instead of w' and ts'' instead of ts' accordingly (recall that $w \dot{<} w''$). Notice that since the prefix of $\mathcal{E}$ up to w' is finite, and since $w''' \dot{<} w''$, that is, the starting point of T'' is before that of T' in $\mathcal{E}$, the recursion is finite and the starting point of the traversal we consider gets closer to Init in each recursive step. Therefore, the recursion will eventually reach a traversal that starts from an established view α and reaches line 65 with $\mathit{desiredView}$ equal to an established view β such that $\alpha \dot{\leq} w$ and $w \dot{<} \beta$, which is the base case we consider. □

By definition of $\mathcal{E}$, if w is an established view then for every established view w' in the prefix of $\mathcal{E}$ before w (not including), some $scan_i(w')$ returns a non-empty set. However, the definition only says that such a $scan_i(w')$ exists, and not when it occurs. The following lemma shows that if w is returned by a Traverse T at time t, then some scan on w' returning a non-empty set must complete before time t. Notice that this scan might be performed by a different process than the one executing T.

LEMMA 5.12. *Let T be an execution of Traverse that reaches line* 65 *at time t with desiredView equal to w such that $w \neq \mathit{Init}$, and consider the prefix of $\mathcal{E}$ up to w: $V_0, V_1, \ldots, V_l$ such that $V_0 = \mathit{Init}$ and $V_l = w$. Then for every $k = 0, \ldots, l-1$, some $scan(V_k)$ returns a non-empty set before time t.*

PROOF. Since $w \neq \mathit{Init}$ there exists a traversal T' that starts from $V_i \dot{<} w$ and reaches line 65 with $\mathit{desiredView} = w$ no later than t. Notice that T' can be T if T starts from a view different than w, or alternatively T' can be a traversal executed earlier by the same process, or finally, a traversal at another process that completes before T begins. By Lemma 5.9, a $\mathit{ReadInView}(V_j)$ performed during T' returns a non-empty set for every $j = i, \ldots, l-1$. If $i = 0$, we are done. Otherwise, $V_i \neq \mathit{Init}$ and we continue the same argument recursively, now substituting V_l with V_i. Since the considered prefix of $\mathcal{E}$ is finite and since each time we recurse we consider a subsequence starting at least one place earlier than the previous starting point, the recursion is finite. □

Corollary 5.13. *Let T be an execution of Traverse that returns a view w and let T' be an execution of Traverse invoked after the completion of T, returning a view w'. Then, $w \dot{\leq} w'$.*

Proof. First, note that by Lemma 5.8 both w and w' are established. Suppose for the purpose of contradiction that $w' \dot{<} w$. By Lemma 5.12, some $scan(w')$ completes returning a non-empty set before T completes. Since T' returns w', its last iteration performs a $scan(w')$ that returns an empty set. This contradicts Lemma 4.3 since T' starts after T completes. □

Corollary 5.14. *Let T be an execution of Traverse that returns a view w and let T' be an execution of Traverse invoked after the completion of T. Then, T' does not invoke $WriteInView(w', *)$ for any view $w' \dot{<} w$.*

Proof. First, by Lemma 5.8, *WriteInView* is always invoked with an established view as a parameter. Suppose for the sake of contradiction that $WriteInView(w', *)$ is invoked during T' for some view $w' \dot{<} w$. Since T returns w and $w' \dot{<} w$, by Lemma 5.12, some $scan(w')$ completes returning a non-empty set before T completes. Since T' invokes $WriteInView(w', *)$, by Lemma 5.9 a $ReadInView(w')$ returned $\emptyset$ during T'. Thus, during the execution of this $ReadInView(w')$, a $scan(w')$ returned $\emptyset$ during T'. This contradicts Lemma 4.3 since T' starts after T completes. □

We associate a timestamp with *read* and *write* operations as follows:

Definition 5.15 (*Associated Timestamp*). Let o be a *read* or *write* operation. We define $ats(o)$, the timestamp *associated* with o, as follows: if o is a *read* operation, then $ats(o)$ is ts_i^{max} upon the completion of Traverse during o; if o is a *write* operation, then $ats(o)$ equals to ts_i^{max} when its assignment completes in line 40.

Notice that not all operations have associated timestamps. The following lemma shows that all complete operations as well as writes that are read-from by some complete read operation have an associated timestamp.

Lemma 5.16. *We show three properties of associated timestamps: (a) for every complete operation o, $ats(o)$ is well defined; (b) if o is a read operation that returns $v \neq \perp$, then there exists $o' = write(v)$ operation, $ats(o')$ is well defined, and it holds that $ats(o) = ats(o')$; (c) if o and o' are write operations with associated timestamps, then $ats(o) \neq ats(o')$ and both are greater than $(0, \perp)$.*

Proof. There might be several executions of Traverse during a complete operation, but only one of these executions completes. Therefore, $ats(o)$ is well defined for every complete *read* operation o. If o is a complete *write*, then notice that $pickNewTS_i = \text{TRUE}$ until it is set to FALSE in line 40, and therefore the condition in line 39 is TRUE until such time. Thus, for a *write* operation, line 40 executes at least once – in *WriteInView* which completes right before the completion of a Traverse during o (notice that *WriteInView* might be executed earlier as well). Once line 40 executes for the first time, $pickNewTS_i$ becomes FALSE. Thus, this line executes at-most once in every *write* operation and exactly once during a complete *write* operation, which completes the proof of (a).

To show (b), notice that v_i^{max} equals to v upon the completion of o. Moreover, since $v \neq \perp$, v is not the initial value of v_i^{max}. Observe the first operation o' that sets v_i^{max} to v during its execution, and notice that v_i^{max} is assigned only in lines 80 and 40. Suppose for the purpose of contradiction that the process executing o' receives v in a REPLY message from another process and sets v_i^{max} to v in line 80. A process p_i sending a REPLY message always includes v_i in this message, and v_i is set only to values received by p_i in $\langle$REQ, W, ...$\rangle$ messages. Thus, some process sends a $\langle$REQ, W, ...$\rangle$ message with v before o' sets its v_i^{max} to v. Since a $\langle$REQ, W, ...$\rangle$ message contains the v_i^{max} of the sender, we conclude that some process must have $v_i^{max} = v$ before o' sets its v_i^{max} to v, contradiction to our choice of o'. Thus, it must be that o' sets v_i^{max} to v in line 40. We conclude that o' is a *write*(v) operation which executes line 40. As mentioned above, this line is not executed more than once during o' and therefore $ats(o')$ is well-defined.

Recall our assumption that only one *write* operation can be invoked with v. Thus, o' is the operation that determines the timestamp with which v later appears in the system (any process that sets v_i to v, also sets ts_i to the timestamp sent with v by o', as the timestamp and value are assigned atomically together in line 85). This timestamp is $ats(o')$, determined when o' executes line 40. When o sets v_i^{max} to v, it also sets ts_i^{max} to $ats(o')$, as the timestamp and value are always assigned atomically together in line 80. Thus, $ats(o) = ats(o')$.

Finally, notice that the associated timestamp of a *write* operation is always of the form $(ts_i^{max}.num + 1, i)$, which is strictly bigger than $(0, \perp)$. Since i is a unique process identifier, if o and o' are two *write* operations executed by different processes, $ats(o) \neq ats(o')$. If they are executed by the same process, since ts_i^{max} pertains its value between operation invocations, increasing the first component of the timestamp by one makes sure that $ats(o) \neq ats(o')$, which completes the proof of (c). □

LEMMA 5.17. *Let o and o' be two complete read or write operations such that o completes before o' is invoked, Then, $ats(o) \leq ats(o')$ and if o' is a write operation, then $ats(o) < ats(o')$.*

PROOF. Denote the complete execution of Traverse during o by T, and let w be the view returned by T and ts be the value of ts_i^{max} when T returns. Note that $ats(o) \leq ts$, since ts_i^{max} only grows during the execution of o, and if o is a *read* operation then $ats(o) = ts$. Notice that there might be several incomplete traversals during o' which are restarted, and there is exactly one traversal that completes.

There are two cases to consider. The first is that o' executes a *ReadInView*(w) that returns. Before this *ReadInView*(w) is invoked, T completes a *WriteInView*$(w, *)$, writing a value with timestamp ts. By Lemma 5.10, after the *ReadInView*(w) completes during o', $ts_i^{max} \geq ts \geq ats(o)$ and thus, when o' completes $ts_i^{max} \geq ats(o)$. If o' is a *read* operation then $ats(o')$ is equal to this ts_i^{max}, which proves the lemma. Suppose now that o' is a *write* operation. Then during o', $pickNewTS_i =$ TRUE until it is set to FALSE in line 40. By Corollary 5.14, no traversal during o' invokes *WriteInView* for any established view $\alpha \dot{<} w$. Thus, *ReadInView*(w) completes during o' before any *WriteInView* is invoked. By Lemma 5.16, $ats(o')$ is well defined and therefore exactly one traversal during o' executes line 40. As explained, since *ReadInView*(w) has already completed when line 40 executes, $ts_i^{max} \geq ats(o)$ and then, ts_i^{max} is assigned $(ts_i^{max}.num + 1, i)$, implying that $ats(o') > ats(o)$.

The second case is that no *ReadInView*(w) completes during o'. Let T' be the traversal which determines $ats(o')$. Let w' be the view from which T' starts, and notice that since T' sets $ats(o')$, it completes *ReadInView*(w'). By Lemma 5.8, w' is an established view. We claim that $w \dot{<} w'$. First, if o' is a *read*, then T' completes and returns some view w''. By Corollary 5.13, $w \leq w''$ and by Lemma 5.9, T' performs a *ReadInView* on all established views between w' and w''. Since o' does not complete *ReadInView*(w), it must be that $w \dot{<} w'$, which shows the claim. Now suppose that o' is a *write*. By Corollary 5.14, T' does not invoke *WriteInView*$(\alpha, *)$ for any view $\alpha \dot{<} w$. It is also impossible that T' invokes *WriteInView*$(w, *)$ as it does not complete *ReadInView*(w). Thus, it must be that T' sets $ats(o')$ when it invokes *WriteInView*$(\alpha, *)$ where $w \dot{<} \alpha$. By Lemma 5.9, T' performs a *ReadInView* on all established views between w' and α. Since it does not complete *ReadInView*(w), it must be that $w \dot{<} w'$, which shows the claim.

Since $w \dot{<} w'$, $w' \neq Init$. Moreover, since $curView_i = w'$ when T' starts, there exists a traversal T'', which reaches line 65 with *desiredView* equal to w' before T' begins. Let ts'' be the ts_i^{max} when T'' reaches line 65. By Lemma 5.11, since $w \dot{<} w'$, it holds that $ts \leq ts''$ and thus $ats(o) \leq ts''$. Since T'' performs $WriteInView(w', *)$ and after it completes, T' invokes and completes $ReadInView(w')$, by Lemma 5.10 we get that ts_i^{max} when $ReadInView(w')$ completes is at least as high as ts''. If o' is a *read*, then $ats(o')$ equals to ts_i^{max} when T' completes, and since ts_i^{max} only grows during the execution of T', we have that $ats(o') \geq ts'' \geq ats(o)$. If o' is a *write*, then $ats(o')$ is determined when line 40 executes. Since this occurs only after $ReadInView(w')$ completes, ts_i^{max} is already at least as high as ts''. Then, line 40 sets $ats(o')$ to be $(ts_i^{max}.num + 1, i)$ and therefore $ats(o') > ts'' \geq ats(o)$, which completes the proof. □

THEOREM 5.18. *Every history σ corresponding to an execution of DynaStore is linearizable.*

PROOF. We create σ' from σ_{RW} by completing operations of the form *write*(v) where v is returned by some complete *read* operation in σ_{RW}. By Lemma 5.16 parts (a) and (b), each operation which is now complete in σ' has an associated timestamp. We next construct π by ordering all complete *read* and *write* operations in σ' according to their associated timestamps, such that a *write* with some associated timestamp ts appears before all *reads* with the same associated timestamp, and reads with the same associated timestamp are ordered by their invocation times. Lemma 5.16 part (c) implies that all *write* operations in π can be totally ordered according to their associated timestamps.

First, we show that π preserves real-time order. Consider two complete operations o and o' in σ' such that o' is invoked after o completes. By Lemma 5.17, $ats(o') \geq ats(o)$. If $ats(o') > ats(o)$ then o' appears after o in π by construction. Otherwise, $ats(o') = ats(o)$ and by Lemma 5.17 this means that o' is a *read* operation. If o is a *write* operation, then it appears before o' since we placed each *write* before all *reads* having the same associated timestamp. Finally, if o is a *read*, then it appears before o' since we ordered reads having the same associated timestamps according to their invocation times.

To prove that π preserves the sequential specification of a MWMR register we must show that a *read* always returns the value written by the closest *write* which appears before it in π, or the initial value of the register if there is no preceding write in π. Let o_r be a *read* operation returning a value v. If $v = \perp$, then since v_i^{max} and ts_i^{max} are always assigned atomically together in lines 80 and 40, we have that $ats(o_r) = (0, \perp)$, in which case o_r is ordered before any *write* in π by Lemma 5.16 part (c). Otherwise, $v \neq \perp$ and by part (b) of Lemma 5.16 there exists a *write*(v) operation, which has the same associated timestamp, $ats(o_r)$. In this case, this *write* is placed in π before o_r, by construction. By part (c) of Lemma 5.16, other *write* operations in π have a different associated timestamp and thus appear in π either before *write*(v) or after o_r. □

5.5.3. Liveness. Recall that all active processes take infinitely many steps. As explained in Section 2, termination has to be guaranteed only when certain conditions hold. Thus, in our proof we make the following assumptions:

A1 At any time t, fewer than $|V(t).members|/2$ processes out of $V(t).members \cup P(t).join$ are in $F(t) \cup P(t).remove$.
A2 The number of different changes proposed in the execution is finite.

LEMMA 5.19. *Let ω be any change such that $\omega \in$ desiredView at time t. Then, a reconfig(c) operation was invoked before t such that $\omega \in c$.*

PROOF. If $\omega \in Init$, the lemma follows from our assumption that a *reconfig*(*Init*) completes by time 0. In the remainder of the proof we assume that $\omega \notin Init$. Let T'

be a traversal that adds ω to its *desiredView* at time t' such that t' is the earliest time when $\omega \in$ *desiredView* for any traversal in the execution. Thus, $t' \leq t$. Suppose for the purpose of contradiction that ω is added to *desiredView* in line 60 during T'. Then $\omega \in c$, such that c is in the set returned by a *scan* in line 34. By property PR1, an *update* completes before this time with c as parameter. By line 54, $\omega \in$ *desiredView* at the traversal that executes the *update*, which contradicts our choice of T' as the first traversal that includes ω in *desiredView*. The remaining option is that ω is added to *desiredView* in line 47 during T'. Since no traversal includes ω in *desiredView* before t', and since $\omega \notin Init$, we conclude that $\omega \notin curView_i$. Thus, $\omega \in cng$. This means that T' is executed during a *reconfig*(c) operation invoked before time t, such that $\omega \in c$, which is what we needed to show. □

LEMMA 5.20. *(a) If w is an established view, then for every change $\omega \in w$, a reconfig(c) operation is invoked in the execution such that $\omega \in c$;*

(b) If w is a view such that $w \in$ Front at time t then for every change $\omega \in w$, a reconfig(c) operation is invoked before t such that $\omega \in c$.

PROOF. We prove the claim by induction on the position of w in $\mathcal{E}$. If $w = Init$, then the claim holds by our assumption that a *reconfig*$(Init)$ completes by time 0. Assume that the claim holds until some position $k \geq 0$ in $\mathcal{E}$. Let w be the kth view in $\mathcal{E}$ and observe w', the $k+1$th established view. By definition of $\mathcal{E}$, there exists a set of changes c such that $w' = w \cup c$, where c was returned by some *scan*(w) operation in the execution. By integrity (property PR1), some *update*(w, c) operation is invoked. By line 54, $c \subseteq$ *desiredView* at the traversal that executes the *update*. (a) then follows from Lemma 5.19. (b) follows from Lemma 5.19, since by Lemma 5.2, we have that $w \subseteq$ *desiredView* and therefore $\omega \in$ *desiredView* at time t. □

COROLLARY 5.21. *The sequence of established view $\mathcal{E}$ is finite.*

PROOF. By Lemma 5.20, established views contain only changes proposed in the execution. Since all views in $\mathcal{E}$ are totally ordered by the "$\subset$" relation, and by assumption A2, $\mathcal{E}$ is finite. □

Definition 5.22. We define t_{fix} to be any time such that $\forall t \geq t_{fix}$ the following conditions hold:

(1) $V(t) = V(t_{fix})$
(2) $P(t) = P(t_{fix})$
(3) $(V(t).join \cup P(t).join) \cap F(t) = (V(t_{fix}).join \cup P(t_{fix}).join) \cap F(t_{fix})$
(i.e., all processes in the system that crash in the execution have already crashed by t_{fix}).

The next lemma proves that t_{fix} is well defined.

LEMMA 5.23. *There exists t_{fix} as required by Definition* 5.22.

PROOF. $V(t)$ contains only changes that were proposed in the execution (for which there is a reconfiguration proposing them that completes). Since no element can leave $V(t)$ once it is in this set, $V(t)$ only grows during the execution, and from assumption A2 there exists a time t_v starting from which $V(t)$ does not change. No *reconfig* operation proposing a change $\omega \notin V(t)$ can complete from t_v onward, and therefore no element leaves the set P from that time and P can only grow. From assumption A2 there exists a time t_p starting from which $P(t)$ does not change. Thus, from time $t_{vp} = max(t_v, t_p)$ onward, V and P do not change. By assumption A2, $V(t_{vp}).join \cup P(t_{vp}).join$ is a finite set of processes. Thus, we can take t_{fix} to be any time after t_{vp} such that all processes from this set that crash in the execution have already crashed by t_{fix}. □

Recall that an active process is one that did not fail in the execution, whose Add was proposed and whose Remove was never proposed.

LEMMA 5.24. *If w is a view in Front such that $V(t_{fix}) \subseteq w$, then at least a majority of w.members are active.*

PROOF. By Lemma 5.20, all changes in w were proposed in the execution. Since all changes proposed in the execution are proposed by time t_{fix}, $w \subseteq V(t_{fix}) \cup P(t_{fix})$. Denote the set of changes $w \setminus V(t_{fix})$ by AC. Notice that $AC \subseteq P(t_{fix})$. Each element in AC either adds or removes one process. Observe the set of members in w, and let us build this set starting with $M = V(t_{fix}).members$ and see how this set changes as we add elements from AC. First, consider changes of the form $(+, j)$ in AC. Each change of this form adds a member to M, unless $j \in V(t_{fix}).remove$, in which case it has no effect on M. A change of the form $(-, k)$ removes p_k from M. According to this, we can write *w.members* as follows: $w.members = (V(t_{fix}).members \cup J_w) \setminus R_w$, where $J_w \subseteq P(t_{fix}).join \setminus V(t_{fix}).remove$ and $R_w \subseteq P(t_{fix}).remove$. We denote $V(t_{fix}).members \cup J_w$ by L and we will show that a majority of L is active. Since R_w contains only processes that are not active, when removing them from L (in order to get *w.members*), it is still the case that a majority of the remaining processes are active, which proves the lemma.

We next prove that a majority of L are active. By definition of t_{fix}, all processes proposed for removal in the execution have been proposed by time t_{fix}. Notice that no process in $V(t_{fix}).members \cup J_w$ is also in $V(t_{fix}).remove$ by definition of this set, and thus, if the removal of a process in L was proposed by time t_{fix}, this process is in $P(t_{fix}).remove$. Since $L \subseteq V(t_{fix}).join \cup P(t_{fix}).join$, by definition of t_{fix} every process in L that crashes in the execution does so by time t_{fix}. Thus, $F(t_{fix}) \cup P(t_{fix}).remove$ includes all processes in L that are not active. Assumption A1 says that fewer than $|V(t_{fix}).members|/2$ out of $V(t_{fix}).members \cup P(t_{fix}).join$ are in $F(t_{fix}) \cup P(t_{fix}).remove$. Thus, fewer than $|V(t_{fix}).members|/2$ out of $V(t_{fix}).members \cup J_w$, which equals to L, are in $F(t_{fix}) \cup P(t_{fix}).remove$. This means that a majority of the processes in L are active. □

LEMMA 5.25. *Let p_i be an active process and w be an established view such that $i \in w.members$. Then $i \in w'.members$ for every established view w' such that $w \dot{\leq} w'$.*

PROOF. Since $w \subseteq w'$ and $i \in w.members$, we have that $(+, i) \in w'$. Since p_i is active, no *reconfig*(c) is invoked such that $(-, i) \in c$, and by Lemma 5.20, we have that $(-, i) \notin w'$. Thus, $i \in w'.members$. □

LEMMA 5.26. *If a reconfig operation o completes such that Traverse returns the view w, then every active process p_j such that $j \in w.members$ eventually receives a message $\langle$NOTIFY, $\tilde{w}\rangle$ where $w \dot{\leq} \tilde{w}$.*

PROOF. Since o completes, there is at least one complete *reconfig* operation in the execution. Let w_{max} be a view returned by a *Traverse* during some complete *reconfig* operation, such that no *reconfig* operation completes in the execution during which *Traverse* returns a view w' where $w_{max} \dot{<} w'$. w_{max} is well defined since every view returned from *Traverse* is established (Lemma 5.8), and $\mathcal{E}$ is finite by Corollary 5.21. Notice that $w \dot{\leq} w_{max}$. We next prove that $V(t_{fix}) \subseteq w_{max}$. Suppose for the purpose of contradiction that there exists a change $\omega \in V(t_{fix}) \setminus w_{max}$. Since $\omega \in V(t_{fix})$, a *reconfig*(c) operation completes where $\omega \in c$. By Lemma 5.7, Traverse during this operation returns a view w' containing ω. By Lemma 5.8 w' is established, and recall that all established views are totally ordered by the "$\subset$" relation. Since $\omega \in w' \setminus w_{max}$ it must be that $w_{max} \dot{<} w'$. This contradicts the definition of w_{max}. We have shown that $V(t_{fix}) \subseteq w_{max}$, which implies that a majority of w_{max} are active, by Lemma 5.24.

Since a *reconfig* operation completes where Traverse returns w_{max}, a ⟨NOTIFY, w_{max}⟩ message is sent in line 29, and it is received by a majority of w_{max}.*members*. Each process receiving this message forwards it in line 92. Since a majority of w_{max} are active, and every two majority sets intersect, one of the processes that forwards this message is active. By Lemma 5.25, since $w \preceq w_{max}$, every active process p_j such that $j \in w.members$ is also in $w_{max}.members$. Since links are reliable and, by definition, an active process does not crash in the execution, every such p_j eventually receives this message. □

LEMMA 5.27. *Consider an operation executed by an active process p_i that invokes Traverse at time t_0 starting from $curView_i = initView$. If no ⟨NOTIFY, $newView$⟩ messages are received by p_i from time t_0 onward such that $initView \subset newView$ then Traverse eventually returns and the operation completes.*

PROOF. Since operations are enabled at p_i only once $i \in curView_i.join$ (lines 11 and 96) and $curView_i$ only grows during the execution, $i \in initView.join$. By Lemma 5.4, for every view w which appears in *Front* during the traversal it holds that $initView \subseteq w$ and therefore $i \in w.join$. Since p_i is active, no *reconfig*(c) is invoked such that $(-, i) \in c$. By Lemma 5.20 we have that $(-, i) \notin w$ and therefore $i \in w.members$. This means that p_i does not halt in line 52, and since links are reliable p_i receives every message sent to it by active processes in w.

Let w be any view that appears in *Front* during the execution of *Traverse*. Notice that w is not necessarily established, however we show that $V(t_{fix}) \subseteq w$. Suppose for the purpose of contradiction that there exists $\omega \in V(t_{fix}) \setminus w$. Since $initView \subseteq w$, $\omega \in V(t_{fix}) \setminus initView$. Since $\omega \in V(t_{fix})$, a *reconfig*(c) operation completes where $\omega \in c$, and by Lemma 5.7 this operation returns a view w' such that $\omega \in w'$. By Lemma 5.8, both *initView* and w' are established, and since $\omega \in w' \setminus initView$, we get that $initView \dot{<} w'$. Since $i \in initView.members$ and p_i is active, by Lemma 5.25, we have that $i \in w'.members$. By Lemma 5.26, a ⟨NOTIFY, w''⟩ message where $w' \dot{\leq} w''$ is eventually received by p_i. Since $initView \dot{<} w''$, this contradicts the assumption of our lemma.

We have shown that $V(t_{fix}) \subseteq w$, and from Lemma 5.24 there exists an active majority Q of *w.members*. Since links are reliable, all messages sent by p_i to *w.members* are eventually received by every process in Q, and every message sent to p_i by a process in Q is eventually received by p_i. Thus, all invocations of *ContactQ*$(*, w.members)$, which involves communicating with a majority of *w.members*, eventually complete, and so do invocations of $scan_i$ and $update_i$ by property PR5. Given that all such procedures complete during a *Traverse* and it is not restarted (this follows from the statement of the lemma since no NOTIFY messages that can restart *Traverse* are received at p_i starting from t_0), it is left to prove that the termination condition in line 64 eventually holds. After Traverse completes, *NotifyQ*(w) is invoked where w is a view returned from Traverse. By Lemma 5.6, $Front = \{w\}$ when Traverse returns, and therefore *NotifyQ*(w) completes as well since there is an active majority in *w.members*, as explained above.

By assumption A2 and Lemma 5.20, the number of different views added to *Front* in the execution is finite. Suppose for the purpose of contradiction that *Traverse* does not terminate and consider iteration k of the loop starting from which views are not added to *Front* unless they have been already added before the kth iteration (notice that by Lemma 5.5, when a view is removed from *Front*, it can never be added again to *Front*; thus, from iteration k onward views can only be removed from Front and the additions have no affect in the sense that they can add views that are already present in Front but not new views or views that have been removed from Front). We first show that in some iteration $k' \geq k$, $|Front| = 1$. Consider any iteration where $|Front| > 1$, and let w be the view chosen from Front in line 51 in this iteration. By Lemma 5.2, in this case $w \neq desiredView$, as *desiredView* contains the changes of all views in Front, and $|Front| > 1$ means that there is at least one view in Front which contains changes that are not in w. Then, line 54 executes, and by Lemma 4.2, *ReadInView* returns a non-empty set. Next, the condition in line 57 evaluates to true and w is removed from Front in line 58. Since no new additions are made to Front starting with the kth iteration (i.e., only a view that is already in Front can be added in line 61), the number of views in Front decreases by 1 in this iteration. Thus, there exists an iteration $k' \geq k$ where only a single view remains in Front.

Observe iteration k', where $|Front| = 1$, and let w be the view chosen from Front in line 51 in this iteration. Suppose for the purpose of contradiction that the condition on line 57 evaluates to true. Then, w is removed from *Front*, and the loop on lines 59–61 executes at least once, adding views to *Front*. By Lemma 5.5, the size of these views is bigger than w, and therefore every such view is different than w, contradicting the fact that starting from iteration k only views that are already in *Front* can be added to *Front* (recall that $k' \geq k$). Thus, starting from iteration k' the condition on line 57 evaluates to false, and *WriteInView* is invoked in iteration k'. Assume for the sake of contradiction that *WriteInView* does not return $\emptyset$. In this case, the loop would continue and w (the only view in Front) is chosen again from Front in iteration $k'+1$. Then, $ReadInView(w)$ returns a non-empty set by Lemma 4.3 and the condition in line 57 evaluates to *true*, which cannot happen, as explained above. Thus, in iteration k', the condition in line 57 evaluates to false, $WriteInView(w, *)$ returns $\emptyset$, and the loop terminates. □

THEOREM 5.28. *DynaStore preserves Dynamic Service Liveness (Definition* 3.2*). Specifically: (a) Eventually, the enable operations event occurs at every active process that was added by a complete reconfig operation. (b) Every operation o invoked by an active process p_i eventually completes.*

PROOF

(a) Let p_i be an active process that is added to the system by a complete *reconfig* operation. If $i \in Init.join$ then the operations at p_i are enabled from the time it starts taking steps (line 11). Otherwise, a *reconfig* adding p_i completes, and let w be the view returned by Traverse during this operation. By Lemma 5.7, $(+, i) \in w$. Since p_i is active, no $reconfig(c)$ operation is invoked such that $(-, i) \in c$. By Lemma 5.20, we get that $(-, i) \notin w$, which means that $i \in w.members$. By Lemma 5.26, p_i eventually receives a $\langle$NOTIFY, $w'\rangle$ message such that $w \dot{\leq} w'$. By Lemma 5.25, $(+, i) \in w'$, that is, $i \in w'.join$. This causes operations at p_i to be enabled in line 96 (if they were not already enabled by that time).

(b) Every operation o invokes Traverse and during its execution, whenever a $\langle$NOTIFY, $newView\rangle$ message is received by p_i such that $curView_i \subset newView$, $curView_i$ becomes *newView* in line 95, and *Traverse* is restarted. By Corollary 5.21, $\mathcal{E}$ is finite. By Lemma 5.8, only established views are sent in NOTIFY messages. Thus, the number of times a Traverse can be restarted is finite and at some point in the execution, no more $\langle$NOTIFY, $newView\rangle$ messages can be received s.t. $curView_i \subset newView$. By Lemma 5.27, Traverse eventually returns and the operation completes.

6. CONCLUSIONS

We defined a dynamic R/W storage problem, including an explicit liveness condition stated in terms of user interface and independent of a particular solution. The definition captures a dynamically changing resilience requirement, corresponding to reconfiguration operations invoked by users. Our approach easily carries to other problems, and allows for cleanly extending static problems to the dynamic setting.

We presented DynaStore, which is the first algorithm we are aware of to solve the atomic R/W storage problem in a dynamic setting without consensus or stronger primitives. In fact, we assumed a completely asynchronous model where fault-tolerant consensus is impossible even if no reconfigurations occur. This implies that atomic R/W storage is weaker than consensus, not only in static settings as was previously known, but also in dynamic ones. Our result thus refutes a common belief, manifested in the design of all previous dynamic storage systems, which used agreement to handle configuration changes. Our main goal in this article was to prove feasibility; future work may study the performance tradeoffs between consensus-based solutions and consensus-free ones.

ACKNOWLEDGMENTS

We thank Ittai Abraham, Eli Gafni, Leslie Lamport and Lidong Zhou for early discussions of this work.

REFERENCES

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *J. ACM 40,* 4, 873–890.

ATTIYA, H., BAR-NOY, A., AND DOLEV, D. 1995. Sharing memory robustly in message-passing systems. *J. ACM 42,* 1, 124–142.

BIRMAN, K., MALKHI, D., AND VAN RENESSE, R. 2010. Virtually synchronous methodology for dynamic service replication. Tech. rep. MSR-TR-2010-151.

CHANDRA, T. D., HADZILACOS, V., TOUEG, S., AND CHARRON-BOST, B. 1996. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*. 322–330.

CHOCKLER, G., GILBERT, S., GRAMOLI, V. C., MUSIAL, P. M., AND SHVARTSMAN, A. A. 2005. Reconfigurable distributed storage for dynamic networks. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*.

CHOCKLER, G., KEIDAR, I., AND VITENBERG, R. 2001. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33,* 4, 1–43.

CORMEN, T. T., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DAVCEV, D., AND BURKHARD, W. 1985. Consistency and recovery control for replicated files. In *Proceedings of the 10th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 87–96.

DELPORTE-GALLET, C., FAUCONNIER, H., AND GUERRAOUI, R. 2010. Tight failure detection bounds on atomic object implementations. *J. ACM 57*(4), 22:1–22:32.

EL ABBADI, A., AND DANI, S. 1991. A dynamic accessibility protocol for replicated databases. *Data and Knowl. Eng. 6*, 319–332.

ENGLERT, B., AND SHVARTSMAN, A. A. 2000. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the the 20th International Conference on Distributed Computing Systems (ICDCS'00)*. IEEE Computer Society, 454.

GILBERT, S., LYNCH, N., AND SHVARTSMAN, A. 2003. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*. 259–268.

HERLIHY, M. P., AND WING, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3, 463–492.

LAMPORT, L. 1986. On interprocess communication – part II: Algorithms. *Distrib. Comput. 1,* 2, 86–101.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16,* 2, 133–169.

LAMPORT, L., MALKHI, D., AND ZHOU, L. 2009. Brief announcement: Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, (Full version appears as Microsoft Technical Report MSR-TR-2009-63.)

LEE, E. K., AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, 84–92.

LYNCH, N., AND SHVARTSMAN, A. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the Symposium on Fault-Tolerant Computing*. IEEE, 272–281.

LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann, San Francisco.

LYNCH, N. A., AND SHVARTSMAN, A. A. 2002. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 5th International Symposium on Distributed Computing (DISC)*.

MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04)*. 105–120.

MARTIN, J.-P., AND ALVISI, L. 2004. A framework for dynamic byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*.

PARIS, J., AND LONG, D. 1988. Efficient dynamic voting algorithms. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*. 268–275.

RODRIGUES, R., AND LISKOV, B. 2003. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Tech. rep. TR/932, MIT LCS.

RODRIGUES, R., AND LISKOV, B. 2004. Reconfigurable byzantine-fault-tolerant atomic memory. In *Proceedings of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22,* 4, 299–319.

VAN RENESSE, R., AND SCHNEIDER, F. B. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04)*.

YEGER LOTEM, E., KEIDAR, I., AND DOLEV, D. 1997. Dynamic voting for consistent primary components. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*. 63–71.

Received December 2009; revised December 2010; accepted January 2011