IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Dynamic Atomic Storage Without Consensus

Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Alexander Shraer

CCIT Report #731
May 2009

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #731 May 2009

Dynamic Atomic Storage Without Consensus

Marcos K. Aguilera* Idit Keidar† Dahlia Malkhi* Alexander Shraer†

June 2, 2009

Abstract

This paper 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 paper, 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.

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*, i.e., adjust their membership over 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, i.e., 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 [7, 22, 9] (as shown in [27]). 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).

*Microsoft Research Silicon Valley, {aguilera, dalia}@microsoft.com

†Department of Electrical Engineering, Technion. {idish@ee, shralex@tx}.technion.ac.il

Interestingly, consensus is not essential for implementing replicated storage. The ABD algorithm [2] 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 problem 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. Clearly, the progress of such service is conditioned on certain failure restrictions in the deployed system. 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 $t \geq n/2$ [2].

The liveness condition of a dynamic system needs to also capture changes introduced by the user. 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. However, the liveness condition we will formulate will not be in the form of a simple threshold; rather, we require crashes to occur gradually, and adapt to 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.

In order to provide a protocol-independent specification, we must expose in the model the completion of reconfigurations. Our service interface therefore includes explicit *reconfig* operations that allow the user to add and remove processes. These operations return OK when they complete. 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 [19, 11], 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 the 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 which 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 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, e.g., [19, 11, 4, 23, 24].

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 [19, 11] solves a similar problem to the one we have formulated above; other works [21, 23, 24] 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 [17, 10, 19, 11, 21, 24], while others implement fault-tolerant consensus decisions on view changes [4, 23]. 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. In RAMBO, two additional conditions are needed: C must be garbage-collected at every non-faulty 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. Note that if some process $p \in C$ might fail, it might be impossible for other processes to learn whether p garbage-collected C or not. 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 [5]. Maintaining agreement on group membership in itself is impossible in asynchronous systems [3]. 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.

The State Machine Replication (SMR) approach [14, 25] provides a fault tolerant emulation of arbitrary data types by forming agreement on a sequence of operations applied to the data. Paxos [14] 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, e.g., [16, 20, 26], and is formulated in [15]. 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 et al. [8] 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, i.e., 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 [8], namely, to model dynamic reconfiguration in which resilience is adaptive to actions by the processes.

3 Dynamic Problem Definition

The goal of our work is to implement 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. Communication channels between all processes are reliable. Below we revisit the definition of reliable links in a dynamic setting.

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 $\perp \notin \mathcal{V}$. The sequential specification for this service is as follows: In a sequence of operations, a read returns the latest written value or $\perp$ if none was written. Atomicity [13] (also called linearizability [12]) requires that for every execution, there exist a corresponding sequential execution, which conforms with the operation precedence relation, and which satisfies the sequential specification.

In addition to the above API, the service exposes an interface for invoking reconfigurations. We define $\textit{Changes} \stackrel{def}{=} \{\textit{Remove, Add}\} \times \Pi$. 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 ω is *proposed* in the execution if a *reconfig*(c) operation is invoked s.t. $\omega \in c$. 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).

For a set of changes w, the *removal-set of* w, denoted $w.\textit{remove}$, is the set $\{i \,|\, \textit{(Remove, i)} \in w\}$. The *join set of* w, denoted $w.\textit{join}$, is the set $\{i \,|\, \textit{(Add, i)} \in w\}$. Finally, the *membership of* w, denoted $w.\textit{members}$, is the set $w.\textit{join} \backslash w.\textit{remove}$.

At any time t in the execution, we define $V(t)$ to be the union of all sets c s.t. a *reconfig(c)* completes by time t. 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 $V(t).\textit{members}$, for every $t \geq t_0$[1]. We assume a non-empty view $V(0)$ which is initially known to

[1] In practice, one can add back a process by changing its id.

every process in the system and we say, by convention, that a *reconfig*$(V(0))$ completes by time 0. Let $P(t)$ be the set of *pending changes* at time t, i.e., for each element $\omega \in P(t)$ some process invokes a *reconfig*(c) operation s.t. $\omega \in c$ by time t, and no process completes such a *reconfig* operation by time t. Denote by $F(t)$ the set of processes which have crashed by time t.

Intuitively, only processes that are members of the current system configuration should be allowed to initiate actions. To capture this restriction, *read*, *write* and *reconfig* operations at a process p_i are initially disabled, until *enable operations* occurs at p_i. 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 1. [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, then the following holds:

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.

A common definition of reliable links states that if processes p_i and p_j are "correct", then every message sent by p_i is eventually received by p_j. We adapt this definition to a dynamic setting as follows: for a message sent at time t, we require eventual delivery if both processes are active and $j \in V(t).join \cup P(t).join$, i.e., a *reconfig*$(c)$ operation was invoked by time t s.t. $(Add, j) \in c$.

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:

NV1 Let o be a *scan*$_i()$ operation that returns C. Then for each $c \in C$, an *update*(c) operation is invoked by some process prior to the completion of o.

NV2 Let o be a *scan*$_i()$ operation that is invoked after the completion of an *update*$_j(c)$ operation, and that returns C. Then $C \neq \emptyset$.

NV3 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 $C \subseteq C'$.

NV4 There exists c such that for every *scan*$()$ operation that returns $C \neq \emptyset$, it holds that $c \in C$.

NV5 If some majority M of processes in P keep taking steps then every *scan*$_i()$ and *update*$_i(c)$ invoked by every process $p_i \in M$ eventually completes.

Although these properties bear resemblance to the properties of atomic snapshot objects [1], NV1-NV5 define a weaker abstraction: we do not require that *all update*s are ordered as in atomic snapshot objects, and even in a sequential execution, the set returned by a *scan* does not have to include the value of the most

Algorithm 1 Weak snapshot - code for process p_i.

```
1:  operation update_i(c)
2:      if collect() = ∅ then
3:          Mem[i].Write(c)
4:      return OK
5:  operation scan_i()
6:      C ← collect()
7:      if C = ∅ then return ∅
8:      C ← collect()
9:      return C
10: procedure collect()
11:     C ← ∅;
12:     for each p_k ∈ P
13:         c ← Mem[k].Read()
14:         if c ≠ ⊥ then C ← C ∪ {c}
15:     return C
```

recently completed *update* that precedes it. 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.

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 s.t. 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 [2].

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*$[i]$. Intuitively, if *collect*() returns a non-empty set then 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.

Standard emulation protocols for atomic SWMR registers [2] guarantee integrity (property NV1) and liveness (property NV5). We next explain why Algorithm 1 preserves properties NV2-NV4; the formal proof of correctness appears in Appendix B. First, 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*. This together with atomicity of all registers in *Mem* implies properties NV2-NV3. Property NV4 stems from

the fact that every *scan*() operation that returns $C \neq \emptyset$ executes *collect* twice. Observe such operation o that is the first to complete one *collect*. Any other *scan*() operation o' begins its second *collect* only after o completes its first *collect*. Atomicity of the registers in *Mem* along with the fact that each register is written at-most once, guarantees that any value returned by a *Read* during the first *collect* of o will be read during the second *collect* of o'.

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 31-41) for reading and writing from/to a quorum of members in a given view, used similarly to the *communicate* procedure in ABD [2]. 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 reads the reconfiguration information. If a new view is discovered, the protocol goes back to the read-phase in the new view (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 67) and then invokes *ContactQ* to read the value and timestamp from a quorum of the view (line 68). It returns a non-empty set if a new view was discovered in line 67. 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 73, and then reading configuration information in line 74 (we shall explain lines 71-72 in Section 5.3).

First, for illustration purposes, consider a simple case where only one reconfiguration request is ever invoked, from w_1 to w_2. We shall refer to this reconfiguration operation as *RC*. The main insight into why the above regime preserves read/write atomicity is the following. Say that a *write* operation performs a write-phase W writing in w_1 the value v with timestamp ts. Then there are two possible cases with respect to *RC*. One is that *RC*'s read-phase observes W. Hence, *RC*'s write-phase writes-back a value into w_2, whose timestamp is at least as high as ts. Otherwise, *RC*'s read-phase does not observe W. This means

that W's execution of *ContactQ* writing a quorum of w_1 did not complete before *RC* invoked *ContactQ* during its read-phase, and so W starts to read w_1's configuration information after *RC*'s preliminary phase has completed, updating this information. Hence, W observes w_2 and the *write* operation continues in w_2 (notice that if a value v' with timestamp higher than ts is found in w_2 then the *write* will no longer send v, and instead simply writes back v' to a quorum of w_2).

In our example above, additional measures are needed to preserve atomicity if several processes concurrently propose changes to w_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. Finally, Section 5.4 presents the notion of established views, which is central to the analysis of DynaStore. Proofs are deferred to Appendix C.

5.1 DynaStore Basics

DynaStore uses *operations*, *upon clauses*, and *procedures*. Operations are invoked by the user, whereas upon-clauses are event handlers – they are actions that may be triggered whenever their condition is satisfied. Procedures are called from an operation. In the face of concurrency, operations and upon clauses act like separate monitors: at most one of each kind can be executed at a time. Note that an operation and an upon-clause might execute concurrently. In addition, all accesses to local variables are atomic (even if accessed by an operation and an upon-clause concurrently), and when multiple local variables are assigned as a tuple (e.g., line 72), the entire assignment is atomic. Operations and local variables at process p_i are denoted with subscript i.

Operations and upon-clauses access different variables for storing the value and timestamp: 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 35) when writing a quorum, and a read-request (line 36) 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 45). When a read-request is received, p_i replies with v_i, ts_i, and the received sequence number (line 46).

Every process p_i executing Algorithm 2 maintains a local estimation of the latest view, *curView*$_i$ (line 9), initialized to $V(0)$ when the process starts. 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 $V(0)$) or in line 81 (for processes added later). If p_i discovers that it is being removed from the system, it simply halts (line 53). 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 $j \in w.\mathit{members}$, updating $ws(w)$ to include the change $c \neq \emptyset$ s.t. $w' = w \cup c$; $|c|$ can be viewed as the weight of the edge – the distance between w' and w in changes. Our algorithm maintains the invariant that $c \cap w = \emptyset$, and thus w' always contains more changes than w, i.e., $w \subset w'$. Hence, the graph of views is acyclic.

The main logic of Algorithm 2 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

Algorithm 2 Code for process p_i.

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

10: initially:
11:     if (i ∈ V(0).join) then enable operations

12: operation read_i():
13:     pickNewTS_i ← FALSE
14:     newView ← Traverse(∅, ⊥)
15:     NotifyQ(newView)
16:     return v_i^max

17: operation write_i(v):
18:     pickNewTS_i ← TRUE
19:     newView ← Traverse(∅, v)
20:     NotifyQ(newView)
21:     return OK

22: operation reconfig_i(cng):
23:     pickNewTS_i ← FALSE
24:     newView ← Traverse(cng, ⊥)
25:     NotifyQ(newView)
26:     return OK

27: procedure NotifyQ(w)
28:     if did not receive ⟨NOTIFY, w⟩ then
29:         send ⟨NOTIFY, w⟩ to w.members
30:     wait for ⟨NOTIFY, w⟩ from majority of w.members

31: procedure ContactQ(msgType, D)
32:     M_i ← ∅
33:     msgNum_i ← msgNum_i + 1;
34:     if msgType = W then
35:         send ⟨REQ, W, msgNum_i, v_i^max, ts_i^max⟩ to D
36:     else send ⟨REQ, R, msgNum_i⟩ to D
37:     wait until M_i contains a ⟨REPLY, msgNum_i, ···⟩
            from a majority of D
38:     if msgType = R then
39:         tm ← max{t:⟨REPLY, msgNum_i, v, t⟩ is in M_i}
40:         vm ← value corresponding to tm
41:         if tm > ts_i^max then (v_i^max, ts_i^max)←(vm, tm)

42: upon receiving ⟨REQ, msgType, num, v, ts⟩ from p_j:
43:     if msgType = W then
44:         if (ts > ts_i) then (v_i, ts_i) ← (v, ts)
45:         send ⟨REPLY, num⟩ to p_j
46:     else send message ⟨REPLY, num, v_i, ts_i⟩ to p_j

47: procedure Traverse(cng, v)
48:     desiredView ← curView_i ∪ cng
49:     Front ← {curView_i}
50:     do
51:         s ← min{|ℓ| : ℓ ∈ Front}
52:         w ← any ℓ ∈ Front s.t. |ℓ| = s
53:         if (i ∉ w.members) then halt
54:         if w ≠ desiredView then
55:             update_i(w, desiredView\w)
56:         ChangeSets ← ReadInView(w)
57:         if ChangeSets ≠ ∅ then
58:             Front ← Front \ {w}
59:             for each c ∈ ChangeSets
60:                 desiredView ← desiredView ∪ c
61:                 Front ← Front ∪ {w ∪ c}
62:         else ChangeSets ← WriteInView(w, v)
63:     while ChangeSets ≠ ∅
64:     curView_i ← desiredView
65:     return desiredView

66: procedure ReadInView(w)
67:     ChangeSets ← scan_i(w)
68:     ContactQ(R, w.members)
69:     return ChangeSets

70: procedure WriteInView(w, v)
71:     if pickNewTS_i then
72:         (pickNewTS_i, v_i^max, ts_i^max) ←
                (FALSE, v, (ts_i^max.num + 1, i))
73:     ContactQ(W, w.members)
74:     ChangeSets ← scan_i(w)
75:     return ChangeSets

76: upon receiving ⟨NOTIFY, w⟩ for the first time:
77:     send ⟨NOTIFY, w⟩ to w.members
78:     if (curView_i ⊂ w) then
79:         pause any ongoing Traverse
80:         curView_i ← w
81:         if (i ∈ w.join) then enable operations
82:         if paused in line 79, restart Traverse from line 48

83: upon receiving ⟨REPLY, ···⟩:
84:     add the message and its sender-id to M_i
```

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 [6], 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 49). Each iteration processes the vertex w in *Front* closest to *curView*$_i$ (lines 51 and 52).

During an iteration of the loop in lines 50–63, 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 55). 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.

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 67. By property NV2, 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 62), which writes the latest known value to this view and again scans *ws*(w) in line 74, returning any outgoing edges that are found. If here too no edges are found, the traversal completes.

Notice that *desiredView* is chosen in line 52 only when there are no other views in *Front*, since it contains the union of all views observed during the traversal, 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 54 evaluates to true, and *ReadInView* returns a non-empty set of changes (outgoing edges) by property NV2. Thus, *WriteInView*$(w, *)$ is invoked only when *desiredView* is the only view in *Front*, i.e., $w =$ *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*$\}$. Then, *desiredView* is assigned to *curView*$_i$ in line 64 and returned from Traverse.

To illustrate such traversals, consider the example in Figure 1. 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 1, includes only *initView*, and *desiredView* $=$ *initView*. Then, the condition in line 54 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 1. 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 50–63, 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 $\{\{(+,5),(-,1),(+,4)\}\}$ (properties NV1 and NV2). Then, V_1 is removed from *Front* and $V_4 = V_1 \cup \{(+,5),(-,1),(+,4)\}$ 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 55), however since an outgoing edge already exists, a new

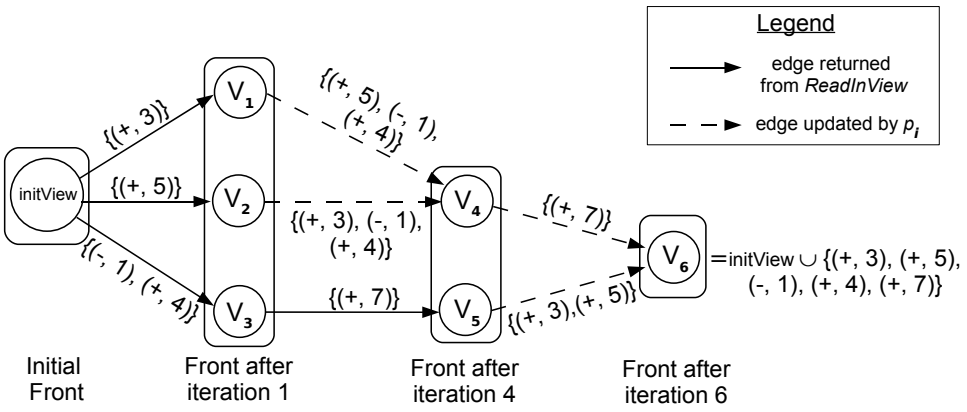

Figure 1: Example DAG of views.

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 63 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 48). Since *desiredView* only grows during a traversal, this ensures that the view returned from *Traverse* includes the changes in *cng*. As explained earlier, *Front* $=$ $\{$*desiredView*$\}$ when *Traverse* completes, which means that *desiredView* appears in the DAG of views.

When a process p_i completes *WriteInView* in line 62 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 77), 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 48. Restarting *Traverse* is necessary when p_j waits for responses from a majority of some view w' where less than a majority of members are active. Intuitively, Definition 1 implies that w' must be an old view, i.e., some *reconfig* operation completes proposing new changes to system membership. We prove in Appendix C.3 that in this case p_j will receive a $\langle$NOTIFY, $w\rangle$ message s.t. *curView*$_j \subset w$ and restart its traversal.

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. Later, it is possible that p_i needs to invoke another operation on *ws*(w). In that case, we require that 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.

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 72). The condition in line 71 prevents *Traverse* from re-assigning v to v_i^{max} or incorrect ts_i^{max}, even if a restart occurs.

In Appendix C.3 we prove that DynaStore preserves Dynamic Service Liveness (Definition 1). Thus, liveness is conditioned on the number of different changes proposed in the execution being finite. In reality, only the number of such changes proposed concurrently with every operation has to be finite. Then, the number of times Traverse can be restarted during that operation is finite and so is the number of views encountered during the traversal, implying termination.

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, property NV4 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 2. [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 $V(0)$;
- 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 2 is a set of changes, and that property NV4 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 $w \stackrel{\cdot}{\leq} w'$ if w appears in $\mathcal{E}$ no later than w'; if in addition $w \neq w'$ then $w \stackrel{\cdot}{<} w'$. Notice that for two established views w and w', $w \stackrel{\cdot}{<} w'$ if an only if $w \subset w'$.

Notice that the first graph traversal in the system starts from *curView*$_i = V(0)$, 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 64 and returned from *Traverse* is also established. Thus, all views sent in NOTIFY messages or stored in *curView*$_i$ are established. Note that while a process encounters all established views in its traversal, it only recognizes a subset of established views as such (whenever *Front* contains a single view, that view must be in $\mathcal{E}$).

It is easy to see that each traversal performs a *ReadInView* on every established view in $\mathcal{E}$ between *curView*$_i$ and the returned view *desiredView*. Notice that *WriteInView* (line 62) is always performed in an

established view. 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 $\mathcal{E}$. Then, performing the *scan* before *ContactQ* in *ReadInView* and after the *ContactQ* in *WriteInView* guarantees that in this intersection, indeed the state is transferred correctly, as explained in the beginning of this section. A formal correctness proof of our protocol appears in Appendix C.

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 paper 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

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[3] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, 1996.

[4] G. Chockler, S. Gilbert, V. C. Gramoli, P. M. . Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[5] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, 2001.

[6] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[7] D. Davcev and W. Burkhard. Consistency and recovery control for replicated files. In *10th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 87–96, 1985.

[8] C. Delporte, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 338–346, 2004.

[9] A. El Abbadi and S. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, 6:319–332, 1991.

[10] B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 454, Washington, DC, USA, 2000. IEEE Computer Society.

[11] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the 17th Intl. Symp. on Distributed Computing (DISC)*, pages 259–268, June 2003.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] L. Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[15] L. Lamport, D. Malkhi, and L. Zhou. Brief announcement: Vertical paxos and primary-backup replication. In 28th ACM Symposium on Principles of Distributed Computing (PODC), August 2009. Full version appears as Microsoft Technical Report MSR-TR-2009-63, May 2009.

[16] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[17] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *In Symposium on Fault-Tolerant Computing*, pages 272–281. IEEE, 1997.

[18] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[19] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *5th International Symposium on Distributed Computing (DISC)*, 2002.

[20] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, pages 105–120, 2004.

[21] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[22] J. Paris and D. Long. Efficient dynamic voting algorithms. In *13th International Conference on Very Large Data Bases (VLDB)*, pages 268–275, 1988.

[23] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, 2003.

[24] R. Rodrigues and B. Liskov. Reconfigurable byzantine-fault-tolerant atomic memory. In *Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, St. John's, Newfoundland, Canada, July 2004. Brief Announcement.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[26] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI 04)*, December 2004.

[27] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 63–71, August 1997.

A Additional Definitions and Assumptions

Our algorithm implements a multi-writer/multi-reader (MWMR) register, from which any client may read or write. Such register is defined via a *sequential specification*, which indicates its behavior in sequential executions. The sequential specification 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. We assume that different *write* operations are invoked with different values, and that no *write* operation is invoked writing the initial value $\perp$. 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 identity of the writer and a local sequence number.

As operations take time, they are represented by two events – an *invocation* and a *response*. 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. We sometimes consider 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, and denote this subsequence of σ by σ_{RW}.

Definition 3 (linearizability [12]). A history σ is *linearizable* w.r.t. a MWMR read/write register X, 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 of X.

We assume that executions of our algorithm are *well-formed*, i.e., 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 [18]).

B Analysis of Weak Snapshot Objects

First, 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 53. Thus, it is allowed to perform these operations on w. The following lemmas prove correctness of a single weak snapshot object accessible by a set of processes P. We assume that all registers in *Mem* are initialized to $\perp$ and that no process invokes $update(\perp)$, which is indeed preserved by DynaStore. The first lemma shows that each register $Mem[i]$ can be assigned at-most one non-initial value.

Lemma 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. □

Properties NV1 (integrity) and NV5 (liveness) can be guaranteed by using standard emulation protocols for atomic SWMR registers [2]. The following lemmas prove that Algorithm 1 preserves properties NV2-NV4.

Lemma 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 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 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 $C \neq \emptyset$. □

Lemma 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 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*, i.e, $c \in C'$. □

Lemma 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 6 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 1 and atomicity of *Mem*$[k]$, a *Read* from the same register performed during the second *collect* of o' returns c. Thus, $C \subseteq C'$. □

C Analysis Of DynaStore

C.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 48.

Lemma 5. *At the beginning and end of each iteration of the loop in lines 50-63, it holds that* $\bigcup_{w \in \textit{Front}} w \subseteq$ *desiredView.*

Proof. We prove that if an iteration begins with $\bigcup_{w \in \textit{Front}} w \subseteq$ *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* $= \{\textit{curView}_i\}$ (line 49) and *desiredView* $=$ *curView*$_i \cup$ *cng* (line 48).

Suppose that at the beginning of an iteration $\bigcup_{w \in \textit{Front}} w \subseteq$ *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 6. *Whenever* $\textit{update}_i(w, c)$ *is executed,* $c \neq \emptyset$ *and* $c \cap w = \emptyset$.

Proof. $\textit{update}_i(w, c)$ is executed only in line 55 when $w \neq$ *desiredView* and $c =$ *desiredView*$\backslash w$, which means that $c \cap w = \emptyset$. By Lemma 5, since $w \neq$ *desiredView*, it holds that $w \subset$ *desiredView*. Thus, $c =$ *desiredView*$\backslash w \neq \emptyset$. □

Lemma 7. *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 50-63 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* $= \{\textit{curView}_i\}$ (line 49).

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, i.e., 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 8. *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 50-63, 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 51 and 52) 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 $\textit{scan}_i$ in line 67. By property NV1, some $\textit{update}(w, c)$ operation is invoked in the execution, and by Lemma 6, $c \neq \perp$ 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 9. *If at some iteration of the loop in lines 50-63 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, $\textit{update}_i(w,$ *desiredView* $\setminus w)$ completes, and then by Lemma 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 s.t. $w' \neq$ *desiredView*. By Lemma 5, $w' \subseteq$ *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 52 in the iteration. □

Lemma 10. *desiredView returned from Traverse contains cng.*

Proof. At the beginning of Traverse, *desiredView* is set to *curView*$_i \cup$*cng* in line 48, and during the execution of Traverse, no element is removed from *desiredView*. Thus, *cng* $\subseteq$ *desiredView* when Traverse completes. □

Lemma 11. *curView*$_i$ *is an established view. Moreover, desiredView in line 64 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 11.1. *If curView*$_i$ *from which a traversal starts is an established view, then Front at the beginning and end of the loop in lines 50-63 contains an established view, and the view desiredView assigned to curView*$_i$ *in line 64 in Traverse is established. Moreover, whenever WriteInView*$(w, *)$ *is invoked,* w *is an established view.*

Proof. Initially, *Front* contains *curView*$_i$ (line 48), 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 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 50-63 still contains an established view.

By Lemma 9, when the loop in lines 50-63 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 64 is established, and so is any view w passed to *WriteInView*. □

We next show that the precondition of the claim above holds, i.e., that *curView*$_i$ is an established view, by induction on $|$*curView*$_i|$. The base is *curView*$_i = V(0)$, 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 $k > |V(0)|$. Consider how *curView*$_i$ got its current value – it was assigned either by some earlier execution of Traverse at p_i in line 64, or in line 80 when a NOTIFY message is received, which implies that some process completes a traversal returning this view. In either case, since *curView*$_i \neq V(0)$, some process p_j has *desiredView* $=$ *curView*$_i$ in line 64, 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 64 of a Traverse which started with an established view, it is also established by Claim 11.1. □

Lemma 12. *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' *s.t. initView* $\dot{\leq} w' \dot{<} w$*, and a ReadInView*(w) *which returns* $\emptyset$*; and (b) if* T *reaches line 64 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 51 and 52, 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 and the definition of $\mathcal{E}$). The arguments above hold for every established view w' s.t. *initView* $\dot{\leq} w' \dot{<} w$, 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 64 with *desiredView* $= w''$. By Lemma 9, w during the last iteration of the loop equals to w''. Observe the condition in line 63, which requires that *ChangeSets* $= \emptyset$ for the loop to end. Notice that *ChangeSets* is assigned either in line 56 or line 62. If it was assigned in line 62, then *WriteInView*$(w, *)$ was executed which completes the proof of (b). Otherwise, *ReadInView*(w) returns *ChangeSets* $= \emptyset$ in line 56, which causes line 62 to be executed. Then, since this is the last iteration, *WriteInView*$(w, *)$ returns $\emptyset$. □

C.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 13. *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, i.e., p's answer counts towards the necessary majority of replies for both W and R. If p receives the $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W with timestamp ts before the $\langle \text{REQ}, \text{R}, \cdots \rangle$ message from R, then by lines 44 and 46 it responds to the message from R with a timestamp at least as big as ts. By lines 39-41, 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 $\langle \text{REQ}, \text{R}, \cdots \rangle$ message from R with timestamp ts before the $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W, then $C' \subseteq C$.

Suppose that p receives the $\langle \text{REQ}, \text{R}, \cdots \rangle$ 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 74. On the other hand, since R has started *ContactQ*(R, *w.members*), it has already completed its *scan*(w) in line 67, which returned C'. When W completes its ContactQ it invokes *scan*(w), which then returns C. By Lemma 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 $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W first, and thus, in this case, $ts' \geq ts$. □

Lemma 14. *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 64 with ts_i^{max} equal to ts' and its desiredView equal to w'. If $w \dot{<} w'$ then $ts \leq ts'$.*

Proof. Consider the prefix of $\mathcal{E}$ up to w': $V_0, V_1, \cdots, V_l$ s.t. $V_0 = V(0)$, $V_l = w'$, 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 11).

First, consider the case that $w'' \dot{\leq} w$. By Lemma 12, since T returns w, it completes *WriteInView*$(w, *)$ which returns $C = \emptyset$. Since T' starts from $w'' \dot{\leq} w$ and reaches line 64 with *desiredView* $= w'$ s.t. $w \dot{<} w'$, by Lemma 12 it completes a *ReadInView*(w) which returns $C' \neq \emptyset$ (notice that *ReadInView*(w) might be

executed in two consecutive iterations of T', in which case during the first iteration *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 13 we have that ts_i^{max} upon the completion of the *ReadInView*(w) by T' is at least as big as ts_i^{max} upon the completion of *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 64, we have that $ts' \geq ts$.

The second case to consider is $w \dot{<} w''$, which implies that $w'' \neq V(0)$. In this case, there exists a traversal T'' which starts from a view $w''' \dot{<} w''$ and reaches line 64 before T begins, with *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 64. Notice that T'' completes *WriteInView*$(w'', *)$ before T' starts *ReadInView*(w''), and by Lemma 13 when *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 $w''' \dot{<} w''$, i.e., 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 $V(0)$ in each recursive step. Therefore, the recursion will eventually reach a traversal which starts from an established view α and reaches line 64 with *desiredView* equal to an established view β s.t. $\alpha \dot{\leq} w$ 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 15. *Let T be an execution of Traverse that reaches line 64 at time t with desiredView equal to w s.t. $w \neq V(0)$, and consider the prefix of $\mathcal{E}$ up to w: $V_0, V_1, \cdots, V_l$ s.t. $V_0 = V(0)$ 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 $w \neq V(0)$ there exists a traversal T' that starts from $V_i \dot{<} w$ and reaches line 64 with *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 12, a *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, $V_i \neq V(0)$ 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 16. *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 11 both w and w' are established. Suppose for the purpose of contradiction that $w' \dot{<} w$. By Lemma 15, 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 3 since T' starts after T completes. □

Corollary 17. *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 11, *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 15 some *scan*(w') completes returning a non-empty set before T completes. Since T' invokes *WriteInView*$(w', *)$, by Lemma 12 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 3 since T' starts after T completes. □

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

Definition 4 (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 72.

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 18. *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*$_i$ = TRUE until it is set to FALSE in line 72, and therefore the condition in line 71 is TRUE until such time. Thus, for a *write* operation, line 72 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 72 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 41 and 72. 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 41. 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 $\langle \text{REQ}, \text{W}, \cdots \rangle$ messages. Thus, some process sends a $\langle \text{REQ}, \text{W}, \cdots \rangle$ message with v before o' sets its v_i^{max} to v. Since a $\langle \text{REQ}, \text{W}, \cdots \rangle$ 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 72. We conclude that o' is a *write*(v) operation which executes line 72. 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 44). This timestamp is $ats(o')$, determined when o' executes line 72. 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 41. 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 19. *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 13, 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 72. By Corollary 17, 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 18, $ats(o')$ is well-defined and therefore exactly one traversal during o' executes line 72. As explained, since *ReadInView*(w) has already completed when line 72 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 11, 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 16, $w \dot{\leq} w''$ and by Lemma 12, 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 17, 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 12, 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 $w \dot{<} w'$, $w' \neq V(0)$. Moreover, since *curView*$_i = w'$ when T' starts, there exists a traversal T'' which reaches line 64 with *desiredView* equal to w' before T' begins. Let ts'' be the ts_i^{max} when T'' reaches line 64. By Lemma 14, 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 13 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 72 executes. Since this occurs only after *ReadInView*(w') completes, ts_i^{max} is already at least as high as ts''. Then, line 72 sets $ats(o')$ to be $(ts_i^{max}.num + 1, i)$ and therefore $ats(o') > ts'' \geq ats(o)$, which completes the proof. □

Theorem 20. *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 18 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 18 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' s.t. o' is invoked after o completes. By Lemma 19, $ats(o') \geq ats(o)$. If $ats(o') > ats(o)$ then o' appears after o in π by construction. Otherwise $ats(o') = ats(o)$ and by Lemma 19 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 41 and 72, we have that $ats(o_r) = (0, \perp)$, in which case o_r is ordered before any *write* in π by Lemma 18 part (c). Otherwise, $v \neq \perp$ and by part (b) of Lemma 18 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 18, other *write* operations in π have a different associated timestamp and thus appear in π either before *write*(v) or after o_r. □

C.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 $|V(t).\textit{members}|/2$ processes out of $V(t).\textit{members} \cup P(t).join$ are in $F(t) \cup P(t).\textit{remove}$.

A2 The number of different changes proposed in the execution is finite.

Lemma 21. *Let ω be any change s.t. $\omega \in$ desiredView at time t. Then a reconfig(c) operation was invoked before t such that $\omega \in c$.*

Proof. If $\omega \in V(0)$, the lemma follows from our assumption that a *reconfig*$(V(0))$ completes by time 0. In the remainder of the proof we assume that $\omega \notin V(0)$. Let T' be a traversal that adds ω to its *desiredView* at time t' s.t. 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 $\omega \in c$, s.t. c is in the set returned by a *scan* in line 67. By property NV1, an *update* completes before this time with c as parameter. By line 55, $\omega \in$ *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 48 during T'. Since no traversal includes ω in *desiredView* before t', and since $\omega \notin V(0)$, we conclude that $\omega \notin \textit{curView}_i$. 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 22. *(a) If w is an established view, then for every change $\omega \in w$, a reconfig(c) operation is invoked in the execution s.t. $\omega \in c$; (b) If w is a view s.t. $w \in$ 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 $w = V(0)$, then the claim holds by our assumption that a *reconfig*$(V(0))$ completes by time 0. Assume that the claim holds until some position $k \geq 0$ in $\mathcal{E}$. Let w be the k-th 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 property NV1, some *update*(w, c) operation is invoked. By line 55, $c \subseteq$ *desiredView* at the traversal that executes the *update*. (a) then follows from Lemma 21. (b) follows from Lemma 21 since by Lemma 5 we have that $w \subseteq$ *desiredView* and therefore $\omega \in$ *desiredView* at time t. □

Corollary 23. *The sequence of established view $\mathcal{E}$ is finite.*

Proof. By Lemma 22, 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. We define t_{fix} to be any time s.t. $\forall t \geq t_{fix}$ the following conditions hold:

1. $V(t) = V(t_{fix})$
2. $P(t) = P(t_{fix})$
3. $(V(t).\textit{join} \cup P(t).\textit{join}) \cap F(t) = (V(t_{fix}).\textit{join} \cup P(t_{fix}).\textit{join}) \cap F(t_{fix})$
   (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 24. *There exists t_{fix} as required by Definition 5.*

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 $t_{vp} = \textit{max}(t_v, t_p)$ onward, V and P do not change. By assumption A2, $V(t_{vp}).\textit{join} \cup P(t_{vp}).\textit{join}$ is a finite set of processes. Thus, we can take t_{fix} to be any time after t_{vp} s.t. 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 25. *If w is a view in Front s.t. $V(t_{fix}) \subseteq w$, then at least a majority of w.members are active.*

Proof. By Lemma 22, 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 $M = V(t_{fix}).\textit{members}$ 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 26. *Let p_i be an active process and w be an established view s.t. $i \in$ w.members. Then $i \in w'$.members for every established view w' s.t. $w \dot{\leq} w'$.*

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 s.t. $(-, i) \in c$, and by Lemma 22 we have that $(-, i) \notin w'$. Thus, $i \in$ *w'.members*. □

Lemma 27. *If p_i and p_j are active processes and p_i sends a message to p_j during the execution of DynaStore, then p_j eventually receives this message.*

Proof. Recall that the link between p_i and p_j is reliable. Since p_i and p_j are active, it remains to show that if the message is sent at time t then $j \in V(t)$.*join* $\cup P(t)$.*join*. Note that p_i sends messages only to processes in w.*members*, where w is a view in Front during Traverse, and therefore $(+, j) \in w$ at time t. By Lemma 22, a *reconfig*(c) was invoked before time t s.t. $(+, j) \in c$. If such operation completes by time t, then $j \in V(t)$.*join*, and otherwise $j \in P(t)$.*join*. □

Lemma 28. *If a reconfig operation o completes such that Traverse returns the view w, then every active process p_j s.t. $j \in$ 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 11), and $\mathcal{E}$ is finite by Corollary 23. 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 10, Traverse during this operation returns a view w' containing ω. By Lemma 11 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 25.

Since a *reconfig* operation completes where Traverse returns w_{max}, a $\langle$NOTIFY, $w_{max}\rangle$ 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 77. 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 26, since $w \dot{\leq} w_{max}$, every active process p_j s.t. $j \in$ *w.members* is also in w_{max}.*members*. By Lemma 27, every such p_j eventually receives this message. □

Lemma 29. *Consider an operation executed by an active process p_i that invokes Traverse at time t_0 starting from curView$_i$ = initView. If no $\langle$*NOTIFY*, $newView\rangle$ messages are received by p_i from time t_0 onward s.t. 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 81) and *curView$_i$* only grows during the execution, $i \in$ *initView.join*. By Lemma 7, 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 22 we have that $(-, i) \notin w$ and therefore $i \in$ *w.members*. This means that p_i does not halt in line 53, and by Lemma 27 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 10 this operation returns a view w' s.t. $\omega \in w'$. By Lemma 11 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 26 we have that $i \in$ *w'.members*. By Lemma 28, a $\langle$NOTIFY, $w''\rangle$ 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 25 there exists an active majority Q of *w.members*. By Lemma 27, 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 NV5. 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 63 eventually holds. After Traverse completes, *NotifyQ*(w) is invoked where w is a view returned from Traverse. By Lemma 9, *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 22, 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 k-th iteration (notice that by Lemma 8, 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 52 in this iteration. By Lemma 5, 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 55 executes, and by Lemma 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 k-th 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 52 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 8, 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 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 30. *DynaStore preserves Dynamic Service Liveness (Definition 1). 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 $i \in V(0)$.*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 10, $(+, i) \in w$. Since p_i is active, no *reconfig*(c) operation is invoked s.t. $(-, i) \in c$. By Lemma 22 we get that $(-, i) \notin w$, which means that $i \in w$.*members*. By Lemma 28, p_i eventually receives a $\langle$NOTIFY, $w'\rangle$ message such that $w \dot{\leq} w'$. By Lemma 26, $(+, i) \in w'$, i.e., $i \in w'$.*join*. This causes operations at p_i to be enabled in line 81 (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 s.t. *curView*$_i \subset$ *newView*, *curView*$_i$ becomes *newView* in line 80, and *Traverse* is restarted. By Corollary 23, $\mathcal{E}$ is finite. By Lemma 11, 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 29, Traverse eventually returns and the operation completes. □