Dynamic Atomic Snapshots*

Alexander Spiegelman[1] and Idit Keidar[2]

1 **Viterbi Dept. of Electrical Engineering, Technion**
Haifa, 32000, Israel.
sashas@tx.technion.ac.il

2 **Viterbi Dept. of Electrical Engineering, Technion**
Haifa, 32000, Israel.
idish@ee.technion.ac.il

Abstract

Snapshots are useful tools for monitoring big distributed and parallel systems. In this paper, we adapt the well-known atomic snapshot abstraction to dynamic models with an unbounded number of participating processes. Our *dynamic snapshot* specification extends the API to allow changing the set of processes whose values should be returned from a scan operation. We introduce the *ephemeral* memory model, which consists of a dynamically changing set of nodes; when a node is removed, its memory can be immediately reclaimed. In this model, we present an algorithm for wait-free dynamic atomic snapshots.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases snapshots, shared memory, dynamic, ephemeral memory

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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

1 Introduction

Atomic snapshots [2, 12] are essential buildings blocks for distributed computing. For example, systems that perform long-running computations regularly take checkpoints in order to avoid restarting from scratch in case of failures [41, 40, 10, 35, 33, 19, 31]. Other systems use snapshots in order to gather statistics [9, 14] or to detect inconsistent states, (e.g., deadlocks) [32, 34, 17]. A snapshot API supports two operations: *scan* and *update*, where a *scan* returns a mapping from every participant to its last *update* value. Until now, snapshots were mostly considered in static models, where the set of participants cannot be dynamically changed.

Yet it is clear that long-lived reliable systems have to be able to replace old and faulty components with new ones. Indeed, there is a growing interest in *dynamic* distributed systems, in which the set of participating processes can be changed on-the-fly according to application demands and available resources [5, 27, 18, 38, 23, 15]. There is also strong motivation for checkpointing and monitoring dynamic systems, for example, large-scale distributed computations running on platforms like Hadoop [36] and Spark [20]. Another example is distributed block-chains [28, 11], which implement distributed shared memory, (e.g., a ledger); consistent snapshots of this memory can be useful for collecting statistical information and checking whether the system is subject to attacks [13].

Motivated by the above, we define and solve the *dynamic snapshot* problem. While previous work [16, 4] has addressed snapshots with infinitely many participants, (see Section 2), to the best of our knowledge, our snapshot is the first to allow dynamic changes in the set of participants whose values are returned by scan operations. We consider here asynchronous dynamic shared memory consisting of *single-writer, multi-reader (SWMR)* registers, capturing systems in which every process has a private memory space where it publishes its state and all other processes can read from it; this occurs, for example, in map reduce-based computation platforms [36, 20], where each process stores partial computation results for later stages to process, as well as in state-machine replication [25, 21] and blockchain protocols [28], where one may want to monitor consistency across replicas.

We distinguish between *persistent* memory, where registers are available even after the processes that write to them are removed from the system, and *ephemeral* memory, which can be reclaimed. Once a process is removed, any ephemeral register it writes to can immediately become unavailable and thus be garbage collected. (Our model and problem definitions are given in Section 3.)

In order to implement any meaningful service in ephemeral memory we have to assume two essential conditions. First, a slow process may lose track of the active set of processes (ones that were added and not removed). Therefore, we have to equip the model with some *discovery* mechanism, which helps slow processes find new added ones. Second, the number of remove operations must be bounded. Otherwise, there is a scenario in which a slow process always tries to read from reclaimed memory, and is thus unable to complete operations; (see more details in Section 4).

Our main result is an atomic wait-free algorithm for dynamic snapshots in ephemeral memory. The algorithm is an extension of the well-known static snapshot algorithm by Attiya et al. [2]. The main challenges in making it dynamic are (i) tracking the active set of processes, (ii) dealing with a potentially infinite number of processes, which makes the helping mechanism more subtle, and (iii) making sure that no pertinent information is lost when ephemeral memory is reclaimed. For didactic reasons, we first present (in Section 5) an algorithm for the persistent dynamic memory model, overcoming the first two challenges. We then extend

the algorithm (in Section 6) for ephemeral memory, addressing the third challenge. The complexity of every snapshot operation is quadratic in the number of processes that were added before the operation started, denoted m. An interesting question for future work is trying to reduce this complexity to $O(m \cdot log(m))$ as was done for static snapshots [8]; (see Section 7).

Summary of contributions:

- We define the dynamic persistant and ephemeral memory models.
- We define a dynamic atomic snapshot.
- We implement wait-free dynamic atomic snapshots in both dynamic memory models.

2 Related Work

The atomic snapshot abstraction [2] was defined and widely studied in static systems, assuming a fixed set of participating processes. Shared memory models that allow infinitely many participating processes and snapshot implementations therein were previously presented in [16, 4]. As opposed to us, they assume multi-writer, multi-reader (MWMR) registers, which cannot be emulated from SWMR ones in these models (as proven in the full paper [39]). In addition, their implementations require a number of MWMR registers that is linear in the number of participating processes, and they do not allow memory reclamation. We, in contrast, define an ephemeral memory model in which registers pertaining to removed processes can be safely reclaimed.

The snapshot problem was also studied for concurrent data structures [24, 30, 29]. However, these works consider a different memory model than ours, in particular, all their memory objects are shared and are not "owned" by any of the threads. Thus, objects are not ephemeral in the sense of "disappearing" when their owners are removed. These papers more adequately capture shared memory multi-processors, whereas our model captures distributed systems with independent state per process.

Our dynamic shared memory models are inspired by recent dynamic work on dynamic message passing systems [5, 38, 23, 15], from which we adopt the idea that processes must be added via explicit *add* operations before they can invoke operations. Similarly, an explicit *remove* operation allows memory to be reclaimed. This extension allows us emulate snapshots from SWMR registers in the presence of infinitely many potential processes, which is impossible in shared memory models that do not support explicit add and remove [16, 4].

3 Model and Problem Definitions

We consider asynchronous dynamic memory, which extends asynchronous fault-prone memory [3, 22, 1] to allow for a dynamic set of nodes. We begin in Section 3.1 with standard shared memory definition, and continue in Section 3.2 to introduce dynamic memory. For brevity, some of the formal definitions can be found in the full paper [39] In Section 3.3, we define the *dynamic snapshot* abstraction, which we emulate in this paper.

3.1 Preliminaries

A shared memory model consists of an infinite set Π of processes accessing variables that reside at nodes from some set N.

Processes Processes may *fail* by crashing or by invoking an explicit *stop* signal. A correct process is one that never fails. There is no restriction on the number of faulty processes.

Nodes Each of the nodes is some shared memory location, either at a single server, or emulated by a group of servers that use protocols like ABD [6] and SMR [26] via message passing.

Processes access nodes' variables via *low-level operations* (e.g., read, write), and interact with objects emulated on top of the set of nodes via *high-level operations* (e.g., update and scan in a snapshot). Both high-level and low-level operations are *invoked* and subsequently *respond*. A *history* is a (finite or infinite) sequence of invocations and matching responses. We refer to the t^{th} event (invoke or response) in H as *time* t. An operation is *pending* in history H if its invocation occurs in H but its response does not.

Operation op_i *precedes* operation op_j in a history H, denoted $op_i \prec_H op_j$, if op_i's response occurs before op_j's invoke in H. Operations op_i and op_j are *concurrent* in H if neither precedes the other. A history with no concurrent operations is *sequential*. A history is *well-formed* if every process's subhistory is sequential. We consider only well-formed histories in this paper. We use sequential histories to define objects' correct behavior: an object's set of allowed sequential histories is called its *sequential specification*. The sequential specification of a register is the following: Every read operation returns the value of the last write that precedes it, or some initial value v_0 in case there is no such write.

Two histories of an object are *equivalent* if every process performs the same sequence of operations (with the same return values) in both, where operations that are pending in one can either be included in or excluded from the other. A *linearization* of a history H is an equivalent sequential history that satisfies H's operation precedence relation and the object's sequential specification. An object is *atomic* if each of its histories has a linearization.

3.2 Dynamic memory

In the *dynamic* model, N is infinite, and the memory is actually kept at a finite subset of N, which changes dynamically. Objects in this model, called *dynamic* objects, have to provide a mechanism to reconfigure the system so as to change this subset. This is done via the special *add* and *remove* operations each object exposes. An explicit remove operation is essential for applications in order to be able to safely transfer a node's state before it is removed and becomes unavailable. An explicit add operation helps processes track the participating processes, as discussed in Section 4. Some initial subset $N_0 \subset N$ is known to all processes. We say by convention that for all $n \in N_0$, *add(n)* is invoked and responds at the beginning of every history. We say that a node n_i is *included* (respectively, *excluded*) at time t in history H if the prefix of length t of H includes a response of an $add(n_i)$ (respectively, $remove(n_i)$) operation. A node n_i is *active* in history H if it is included at any time in H and not excluded in H.

In this paper we are interested in what can and cannot be done assuming single writer registers. In this context, each node $n_i \in N$ is associated with a unique process $p_i \in \Pi$, and holds one atomic SWMR register to which only p_i can write and from which all processes can read. We refer to the SWMR register at node n_i, (which is associated with process p_i), as $segment_i$.

A process p_i is *active* if node n_i is active. A *wait-free* implementation of an object (in the dynamic model) is one that guarantees that any operation invoked by a correct (and active) process completes regardless of the actions of other processes.

We define two memory responsiveness models for dynamic memory:

- *Persistent memory:* Every $segment_i$ s.t. n_i is included is wait-free. That is, once a process is added, its segment is forever available.

- *Ephemeral memory:* Segments of active nodes are wait-free. Note that here, once a node is removed, the information it holds is not necessarily available.

Wait-free segments are called *responsive*, whereas other segments are *unresponsive* [22, 3, 1]. We refer to the dynamic model with persistent memory as the *persistent memory* model, and to the dynamic model with ephemeral memory as *ephemeral memory* model.

3.3 Dynamic snapshots

Snapshot objects [2] expose an interface for invoking *scan* and *update* operations. A *dynamic snapshot* object extends the snapshot object with *add* and *remove* operations, and has the following sequential specification:

▶ **Definition 1** (Dynamic snapshots' sequential specification)**.** Update, add, and remove return ok. A *scan* operation invoked at some time t in history H returns a mapping from every node n_i that is included and not excluded at time t in H to a value v_i s.t. v_i is the argument of the last *update* operation invoked by p_i before time t in H, or $\perp$ if no *update* is invoked by p_i before the *scan*.

In this paper we are interested in wait-free implementation of dynamic atomic snapshots in dynamic memory models.

4 Essential Assumptions

In this section we discuss our assumptions.

Explicit add. Wait-free high-level objects cannot be implemented from low-level SWMR registers if infinitely many processes may start to participate, i.e., (invoke high-level operations), at any time without an explicit add. This is actually true in both persistent and ephemeral memory models; (it is stated in [4], and, for completeness, proven in the full paper [39]. Thus, we henceforward assume the following:

▶ **Assumption 1.** *At any time, only processes associated with included nodes can invoke high-level operations.*

Discovery mechanism. Given that in ephemeral memory, removed nodes may be unresponsive, we have to equip processes with some mechanism to locate included nodes. Otherwise, a slow process may be unable to proceed after all nodes it had been aware of have been removed and have become unresponsive. For clarity, we avoid using an additional discovery entity, but instead assume that accesses to unresponsive nodes throw exception messages with segments belonging to responsive nodes. Formally, we assume the following:

▶ **Assumption 2.** *When a process p reads from an unresponsive node n_i, it receives either $segment_i$, or an exception notification with some $segment_j$. Moreover, if p reads n_i infinitely often and never receives $segment_i$, then every segment that belongs to a responsive node is returned at least once.*

Finite number of removals. In addition, it is impossible to implement wait-free dynamic objects in ephemeral memory in the presence of infinitely many remove operations. This can proven similarly to the impossibility proof in [37], and so the formal proof is omitted. Instead we provide the following intuitive justification:

▶ **Claim 1.** *There is no wait-free atomic snapshot implementation in ephemeral memory where infinitely many removes may be invoked.*

Proof sketch. Consider a slow process p_i that invokes a high-level operation at time t and before its low-level operations reach any node, all nodes that were included by time t are removed and become unresponsive. We can construct an infinite history in which the following happens repeatedly: p_i learns from an exception about a node $n \in N$, then some other process p_j adds node $n' \neq n$ and removes node n. Notice that the add and remove operations have to be wait-free and p_j cannot write to the node associated with p_i (single writer), so the operations complete without affecting p_i's node. Then, node n becomes unresponsive, so p_i cannot read from it. By repeating this process infinitely, we get an infinite run where p_i does not read from any node except its own, and thus, its high-level operation cannot complete. A contradiction to wait-freedom.

One way to circumvent the impossibility is by assuming a bound on the rate of remove operations and a corresponding bound on the low-level operation delay [7]. However, since we want to focus on a fully asynchronous model, we instead assume the following:

▶ **Assumption 3.** *The number of remove operations is finite.*

5 Dynamic Snapshots in Persistent Memory

In this section we assume Assumption 1 and present an algorithm for a wait-free dynamic snapshot in the persistent memory model. This algorithm serves as a stepping stone for our ephemeral memory algorithm given in the next section.

Static snapshots. The general idea is based on the well-known snapshot algorithm for static systems [2]: Each process p_i writes only to $segment_i$, which holds the value written by its last *update*, denoted val_i, and some additional information. A process that performs a *scan* operation repeatedly collects all the segments until it gets two identical scans, which is called a *double collect*. The process then stores in its segment the mapping of processes to data read from their segments in this double collect, called *view*. Notice that if other processes perform infinitely many updates concurrently with the *scan*, the scan may fail to ever obtain a double collect. In order to overcome this, the algorithm uses a helping mechanism, whereby a process obtains a scan and stores it in its view before writing a new value to its segment. A process that fails to obtain a successful double collect a certain number of times can "borrow" a view from another process.

Dynamic view. In the dynamic model, we need to implement also *add* and *remove*, which change the set of processes that can invoke operations and the set of values that should be returned by a *scan*. The view is thus no longer a static array. Instead, it is a mapping from a dynamic set of nodes to their values. Specifically, the view embedded in the segment holds three fields: The first, denoted *mem*, is the set of all known active nodes, initially N_0. (In the original algorithm, this set is static, thus there is no need to store it in the segment.) The second field, *removed*, tracks excluded nodes. The third field is a map, *snap*, from $mem \cup removed$ to segments, where $snap[i]$ holds the last value val_i read from $segment_i$.

In order for scans to determine which segments' values to return, (i.e., which nodes were included and not excluded), we add to every segment a set *changes* consisting of tuples of the form $\langle add/remove, n_i \rangle$. A process that performs *add* or *remove* adds the operation to *changes*. A scan by process p_i is performed in iterations as follows: It first collects the values

from the segments that belong to processes in its current $mem \cup removed$; then checks their *changes* sets to discover which processes were included or excluded and updates *mem* and *removed* accordingly; and repeats this process if no double collect was obtained. Notice that since we consider a persistent memory model at this point, segments of excluded processes remain responsive. Therefore, information about added and removed processes is never lost, and even slow processes can obtain it.

Helping. The second issue we address is how a process can know which view it can borrow during a *scan* operation. Consider a run, illustrated in Figure 1, in which some process performs a *scan* concurrently with infinitely many *add* operations, s.t. every process performs exactly one *add* and no updates. One way for a scan to complete is by obtaining a successful double collect, but in this case, because of the infinitely many *add* operations, the *scan* can never obtain one despite the fact that there are no updates. Alternatively, a scan can borrow a view from another process, but it needs to make sure that the view is fresh enough.

To this end, we add a version number, denoted *num*, to every segment and include it in the embedded view. Each process increases its *num* at the beginning of every *scan* operation, and in every collect it checks whether some process has a view that contains its own updated *num*. If some process has such a view, then it means that this view is fresh (obtained after the scan began) and can be borrowed. An illustration is presented in Figure 2.

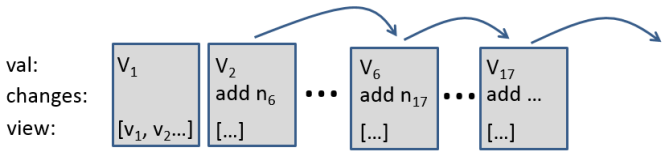

Figure 1 A run with infinitely many process additions; the scanning process cannot detect which view is fresh and may be borrowed.

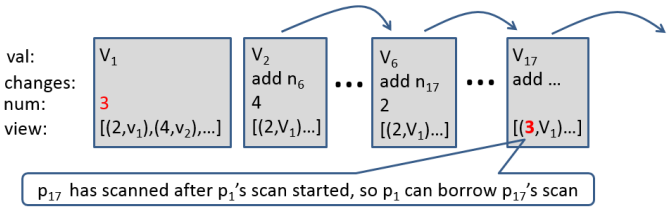

Figure 2 A run with infinitely many process additions; p_1's scan may return the view from $segment_{17}$, since it was obtained by p_{17} during the scan.

Detailed algorithm. The segment structure is defined in Algorithm 1 and illustrated in Figure 3.

In the context of our algorithm, we say that a *node* n_i *is added before time* t if $n_i \in N_0$ or some process performs a low-level write of $\langle add, n_i \rangle$ to its segment's *changes* during an $add(n_i)$ operation before time t. In the same way, we say that a *node* n_i *is removed before time* t if some process performs a low-level write $\langle remove, n_i \rangle$ to its segment during $remove(n_i)$ before time t. These embedded writes are also the linearization points of the *add* and *remove* operations.

At any time t, we define *full-snapshot(t)* to be the states (excluding the embedded views) at time t of the segments of nodes added before time t: each node n_i is mapped to the

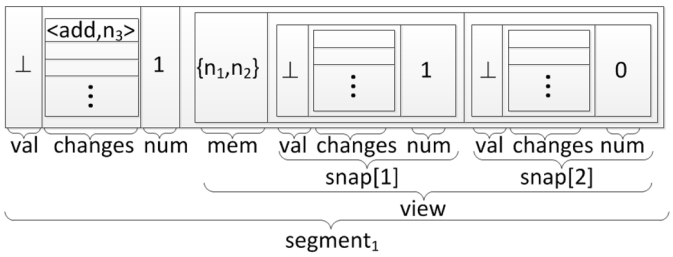

Figure 3 Example of $segment_1$. In this example $N_0 = \{n_1, n_2\}$, process p_2 has not invoked any operation yet, and process p_1 completed *add*(n_3), including writing 1 to $segment_1.num$, performing *embeddedScan* and writing the result to $segment_1.view$, and finally writing $\langle add, n_3\rangle$ to $segment_1.changes$.

tuple $\langle value, changes, num\rangle$ that was last written to $segment_i$ before time t. We define *snapshot(t)* to be the sub-mapping of *full-snapshot(t)* excluding nodes that were removed before time t.

The core of the algorithm lies in the *embeddedScan* procedure, which obtains *full-snapshot(t)* for some t that is later than the time when the procedure is invoked and saves it in the *view* field of the segment. Helping is done by performing *embeddedScan* at the beginning of every operation (*scan*, *update*, *add*, and *remove*).

Algorithm 1 Segment structure.

$segment = \langle val, changes, num, view\rangle$
  where $view = \langle mem, removed, snap\rangle$
    where $mem, removed \subseteq N$, and $snap$ is a mapping from mem to tuples $\langle val, changes, num\rangle$
initially: if $n_i \in N_0$, $segment_i = \langle \perp, \{\}, 0, \langle N_0, \langle \perp, \{\}, 0\rangle^{|N_0|}\rangle\rangle$, else $segment_i = \perp$

Algorithm 2 Dynamic snapshots in persistent memory: operations. Pseudocode for process p_i.

```
1:  procedure scan_i()
2:      embeddedScan()
3:      for each n_j ∈ segment_i.view.mem
4:          V[j] = segment_i.view.snap[j].val
5:      return V

6:  procedure update_i(d)
7:      embeddedScan()
8:      segment_i.val ← d

9:  procedure add_i(n_j)
10:     embeddedScan()
11:     segment_j ← ⟨⊥, {}, 0, segment_i.view⟩          ▷ set segment_j's initial value
12:     segment_i.changes ← segment_i.changes ∪ {⟨add, n_j⟩}

13: procedure remove_i(n_j)
14:     embeddedScan()
15:     segment_i.changes ← segment_i.changes ∪ {⟨remove, n_j⟩}
```

Pseudocode for the algorithm's operations is presented in Algorithm 2. A *scan* first

performs *embeddedScan* in line 2, and then in lines 3-5 it returns a mapping from scanned nodes in *mem* to their segment values. The *update* operation first performs *embeddedScan* and then writes the new value to the segment. Similarly, *add* and *remove* first perform *embeddedScan*, and then add to *changes* the information about the included or excluded node. Additionally, the initial value of a newly added segment is set as part of the *add* operation.

Algorithm 3 Dynamic snapshots in persistent memory: *embeddedScan* function. Pseudocode for process p_i.

```
16: procedure embeddedScan()_i
17:     PrevView ← segment_i.view
18:     segment_i.num ← segment_i.num + 1                          ▷ increase version number

19:     while true                                                 ▷ try to obtain a consistent snapshot
20:         CurView.mem ← PrevView.mem
21:         CurView.removed ← PrevView.removed
22:         for each n_j ∈ CurView.mem ∪ CurView.removed            ▷ collect
23:             CurView.snap[j] ← ⟨segment_j.val, segment_j.changes, segment_j.num⟩
24:         if CurView=PrevView                                    ▷ successful double collect
25:             goto Done
26:         for each n_j ∈ CurView.mem ∪ CurView.removed
27:             if segment_j.view.snap[i].num = segment_i.num      ▷ found a fresh snapshot
28:                 CurView ← segment_j.view
29:                 goto Done
30:             for each ⟨OP, n_l⟩ ∈ CurView.snap[j].changes \ PrevView.snap[j].changes
                                                                   ▷ update view
31:                 if OP = add ∧ n_l ∉ PrevView.removed
32:                     PrevView.mem ← PrevView.mem ∪ {n_l}
33:                     PrevView.snap[l] ← ⟨⊥, {}, 0⟩
34:                 else
35:                     PrevView.mem ← PrevView.mem \ {n_l}
36:                     PrevView.removed ← PrevView.removed ∪ {n_l}
37:             PrevView.snap[j] ← CurView.snap[j]
    Done:
38:     if ∃j s.t. ⟨remove, n_i⟩ ∈ CurView.snap[j].changes then stop   ▷ n_i was excluded
39:     segment_i.view ← CurView
```

The *embeddedScan* procedure (Algorithm 3) first increases the version number (line 18), and then begins repeatedly collecting segments of all known processes. It uses two local variables to track the added nodes and their views, $CurView$ and $PrevView$. Each of them is structured like *view*, consisting of *mem*, *removed*, and *snap*. In every iteration after the first, $PrevView$ stores the view from the previous iteration, and in the first iteration it holds the view from p_i's segment. Lines 22-23 collect a new view into $CurView$. Note that we collect segments not only from nodes in the current *mem*, but also from removed ones. Failing to do so would introduce a subtle problem: it may cause us to miss operations that are successfully completed by processes after their removal, and before they discover the removal and stop; we shall revisit this issue in the next section, where we consider *ephemeral* memory and hence cannot rely on removed nodes to respond.

There are two ways for p_i to complete *embeddedScan*. The first is by obtaining a double collect in line 24. The second is by borrowing the view of another process that contains p_i's up-to-date version number (lines 27–29). It is guaranteed that this view was obtained

after p_i's *embeddedScan* began because version numbers never decrease, and this number is increased at the beginning of the *embeddedScan*.

In lines 30–37, $PrevView$ is updated according to $CurView$. Finally, in line 38, p_i checks if its node was removed, and if so, stops. Otherwise, p_i writes the new view to its segment in line 39.

6 Dynamic Snapshots in Ephemeral Memory

In this section we assume Assumptions 1–3 and extend the algorithm of Section 5 for the ephemeral memory model. We present the algorithm in Section 6.1, and discuss its complexity in Section 6.2. A formal correctness proof is given in the full paper [39].

6.1 Algorithm

Recall that in the ephemeral memory model, nodes can become unresponsive, and thus, information (for example, about added and removed nodes) that is stored in their segments can be lost. Therefore, unlike the algorithm of Section 5, before removing a node, we need to make sure that information about its associated process's completed *add* and *remove* operations will persist after the node is excluded; note that it is possible that such operations are still pending when the node is being removed and complete later. Our algorithm correctness is based on the following claim (see proof in the full paper [39]):

▶ **Claim 2.** *For every time t, for every two processes p_i, p_j, if $segment_j.changes$ includes $\langle remove, n_i, commit\rangle$ at time t, then at time t, $segment_j.changes$ includes every $\langle OP, NODE, commit\rangle$ ever included in $segment_i.changes$.*

Note, in particular, that Claim 2 implies that if p_i completes an operation after p_j removes it, that operation is already reflected in p_j. Given our assumption that the number of removes is finite (Assumption 3), Claim 2 implies that information about every succeeded operation is eventually stored at an active, and therefore responsive, node. Note that once this information is stored at some responsive node, then thanks to our discovery mechanism (Assumption 2), it is reachable by all correct processes. From this point, every correct process can eventually complete its *embeddedScan* as in the algorithm of the previous section.

State transfer. In order to make sure that information about added and removed nodes persists, processes now update their *changes* set with all such information observed during an *embeddedScan*. The new algorithm's *embeddedScan* procedure is presented in Algorithm 4. The segment structure remains as in Algorithm 1. The *embeddedScan* uses a local set $Changes$ to track the information observed during its iterations, and $segment.changes$ is updated according to $Changes$ at the end of the procedure.

When a process p tries to read from a removed node in line 9 during an *embeddedScan*, the discovery service may throw an exception with a value read from another segment. Upon such an exception (line 27), p checks whether the *removed* set in the view returned by the exception contains nodes that p did not know were removed. If so, p updates it local variables $PrevView$ and $Changes$, and jumps to the beginning of the next iteration (Loop) to collect from the new *mem* set. Otherwise, retries the read.

Additional phases in *add* and *remove*. Since removed nodes can be unresponsive, processes should not attempt to collect their segments during *embeddedScan*. However, this

introduces a subtle problem: In the basic algorithm, a process can complete an *add* or *remove* operation long after it is removed. For example, it can complete an *embeddedScan*, then be removed by some other process, and then (without knowing that it has been removed) write to its *segment.changes*; recall that writing to *changes* is the linearization point of the operation. Since processes no longer collect removed segments, we cannot allow removed nodes to complete operations that might be missed by some future *embeddedScan*.

To overcome this problem, we use multiple phases in the *add* and *remove* operations. Pseudocode for the revised operations is given in Algorithm 5. At first, $add(n)$ calls *embeddedScan* and adds $\langle add, n, propose\rangle$ to its *changes* set (lines 34-36). The purpose of this phase is to announce ongoing operations, so that other processes can help complete them if necessary, while still being able to refrain from completing the *add* in case self-removal is observed. Tuples with propose are not taken into account when the sets *mem* and *removed* are updated during *embeddedScan* iterations (line 16). The second phase calls *embeddedScan* again (line 37). Recall that if *embeddedScan* observes its own removal has started by some process, it stops. Otherwise, the operation adds $\langle add, n, commit\rangle$ to its *changes* set (line 38).

Algorithm 4 Dynamic snapshots in ephemeral memory: *embeddedScan* function. Pseudocode for process p_i.

```
1:  procedure embeddedScan_i()
2:      PrevView ← segment_i.view
3:      Changes ← segment_i.changes
4:      segment_i.num ← segment_i.num + 1                    ▷ increase version number
5:      while true                                           ▷ try to obtain consistent snapshot
6:          CurView.mem ← PrevView.mem
7:          CurView.removed ← PrevView.removed
8:          for each n_j ∈ CurView.mem                       ▷ the following line may through an exception
9:              CurView.snap[j] ← ⟨segment_j.value, segment_j.changes, segment_j.num⟩
10:         if CurView=PrevView                              ▷ successful double collect
11:             goto Done
12:         for each n_j ∈ CurView.mem s.t. PrevView.snap[j] ≠ CurView.snap[j]
13:             if segment_j.view.snap[i].num = segment_i.num    ▷ found a fresh snapshot
14:                 CurView ← segment_j.view                 ▷ may through an exception
15:                 goto Done
16:             for each ⟨OP, n_l, commit⟩ ∈ CurView.snap[j].changes \ Changes   ▷ update view
17:                 if OP = add ∧ n_l ∉ PrevView.removed
18:                     PrevView.mem ← PrevView.mem ∪ {n_l}
19:                     PrevView.snap[l] ← ⟨⊥, {}, 0⟩
20:                 else
21:                     PrevView.mem ← PrevView.mem \ {n_l}
22:                     PrevView.removed ← PrevView.removed ⋃ {n_l}
23:             Changes ← Changes ∪ CurView.snap[j].changes
24:             PrevView.snap[j] ← CurView.snap[j]
    Done:                                                    ▷ no exceptions from here
25:     segment_i ← ⟨segment_i.value, Changes, segment_i.num, CurView⟩
26:     if ⟨remove, n_i, ∗⟩ ∈ segment_i.changes then stop

27: upon exception(Seg)
28:     if Seg.removed \ PrevView.removed ≠ {}           ▷ found new removed node, jump forward
29:         PrevView ← Seg.view
30:         Changes ← Seg.changes
31:         goto Loop
32:     else retry read
```

A *remove* operation consists of three phases. A process p_i that performs $remove(n_j)$ first calls *embeddedScan*, then adds $\langle remove, n_j, prepare\rangle$ to its *changes* set. The purpose of this phase is to announce ongoing remove operations so that removed processes will observe them and stop before committing new operations. In the second phase p_i calls *embeddedScan* again in order to check what operations p_j concurrently performs, i.e., what operations p_j has already proposed but has not yet committed, and then it proposes them together with its proposal by adding $\langle OP, NODE, propose\rangle$ to its *changes* set for every $\langle OP, NODE, propose\rangle$ it has observed in $segment_j.changes$ during its last *embeddedScan* together with $\langle remove, p_j, propose\rangle$. This phase enforces a "flag principle": if the removed node doesn't see its own remove and stop, then its proposal is seen and proposed together with the proposal to remove it. For example, if a process p_1 performs $add(n)$ or $remove(n)$ concurrently with a $remove(n_1)$ operation by another process p_2, then either (1) p_1 observes $\langle remove, n_1, prepare\rangle$ before committing its operation and stops, or (2) p_2 observes p_1's $\langle OP, n, propose\rangle$ and proposes it together with $remove(n_1)$.

In the third phase p_i calls *embeddedScan* again, but this time it serves two different purposes: First, as in *add*, it checks (at the end of the *embeddedScan*) if some other process already initiated removal, in which case it stops before committing its proposals. Second, it checks if some other process has already committed a $remove(p_j)$, in which case it completes the operation without committing p_j's proposals. Otherwise, p_i commits all its proposals, i.e., it adds $\langle OP, NODE, commit\rangle$ to its *changes* set for every $\langle OP, NODE, propose\rangle$ it proposed in the second phase. The second check is essential because in case p_i observes that some other process p_k had removed p_j, it may be the case that p_k had missed some of p_i's proposals and committed p_i's removal without them. Hence, committing them know violates Claim 2.

The linearization point of an $add(n)$ or $remove(n)$ operation is when $\langle add, n, commit\rangle$ or $\langle remove, n, commit\rangle$ is added to a *changes* set of one of the segments for the first time (not necessarily by the process that invoked the operation).

6.2 Complexity

In this section we analyze the complexity of our algorithm. We measure complexity of an operation as the total number of memory accesses it performs, including ones that result in exceptions. Note that all the operations (*update*, *scan*, *add*, and *remove*) perform *embeddedScan* at most three times in addition to a constant number of low-level writes. Thus, the asymptotic complexity of all operations is equal to the complexity of the *embeddedScan* procedure. We assume that the discovery service does not return the same segment twice during the same while iteration (collect).

▶ **Claim 3.** *Let op be an embeddedScan invoked at time t by process p_i, and let m be the number of included nodes at time t. Then op's complexity is $O(m^2)$.*

Proof sketch. We start by showing that *op* performs at most $O(m)$ collects. Note that after two iterations, *op* performs an additional collect only if there exists a $segment_j$ that is different in the current and in the previous collects, and $segment_j.view.snap[i].num < segment_i.num$. This can only happen if there is an operation by process p_j that is invoked before *op*, during which p_j writes to $segment_j$ after p_i reads $segment_j$ in the previous collect, and before p_i reads $segment_j$ in the current collect. By Assumption 1 and since we assume well-formed histories, the number of such operations is bounded by m. Thus, *op* performs $O(m)$ collects.

We now show that *op* successfully reads at most $O(m)$ segments in every collect. Assume in a way of contradiction that *op* reads more than $2m$ segments in some collect *col*. Therefore,

op observes, before *col* begins, more than m nodes that were added after *op* was invoked. Thus, *op* observes in some segment $segment_j$, before *col* begins, at least one node whose addition was invoked after *op*. Therefore, *op* reads $segment_j.view.snap[i].num = segment_i.num$ before *col* begins, and thus completes without performing *col*. A contradiction.

By a similar argument and by the assumption that the discovery service does not return the same segment twice in the same collect, the number of exceptions *op* handles in every collect is $O(m)$. All in all, we conclude that the complexity of our algorithm is $O(m^2)$.

◀

In our analysis above m denotes the number of nodes that are included before *op* is invoked. However, we do not need to count in m excluded nodes that become unresponsive before *op* is invoked and the discovery service no longer returns them. Therefore, the complexity of the algorithm depends on the quality of the discovery service: the faster it is notified about excluded nodes, the less excluded nodes affect complexity. For example, if the discovery service is perfect and excluded nodes immediately become unresponsive, then the complexity of an *embeddedScan* does not depend on nodes that were excluded before it was invoked.

Algorithm 5 Dynamic snapshots in ephemeral memory: *add* and *remove* operations. The *update* and *scan* operations remain the same as in Algorithm 2. Pseudocode for process p_i.

```
33: procedure add_i(n_j)
34:     embeddedScan()                                              ▷ phase 1: propose
35:     segment_j ← ⟨⊥, segment_i.changes, 0, segment_i.view⟩        ▷ set segment_j's initial value
36:     segment_i.changes ← segment_i.changes ∪ {⟨add, n_j, propose⟩}
37:     embeddedScan()                                              ▷ phase 2: commit
38:     segment_i.changes ← segment_i.changes ∪ {⟨add, n_j, commit⟩}

39: procedure remove_i(n_j)
40:     embeddedScan()                                              ▷ phase 1: prepare
41:     segment_i.changes ← segment_i.changes ∪ {⟨remove, n_j, prepare⟩}
42:     embeddedScan()                                              ▷ phase 2: propose
43:     ProposeSet = {⟨∗, ∗, propose⟩ ∈ segment_i.snap[j].changes} ∪ {⟨remove, n_j, propose⟩}
44:     segment_i.changes ← segment_i.changes ∪ ProposeSet
45:     embeddedScan()                                              ▷ phase 3: commit
46:     if ⟨remove, p_j, commits⟩ ∉ segment_i.changes
47:       CommitSet = {⟨OP, NODE, commit⟩ | ⟨OP, NODE, propose⟩ ∈ ProposedSet}
48:       segment_i.changes ← segment_i.changes ∪ CommitSet
```

7 Discussion

Atomic snapshots are essential building blocks in distributed systems. Clearly, any long-lived distributed system must support dynamism to replace old entities with new ones. In this paper, we addressed dynamic atomic snapshots for the first time. We defined asynchronous dynamic shared memory models consisting of a changing active set of nodes, each of which contains SWMR registers. We distinguished between the case in which nodes that are no longer part of the set can be reclaimed and become unresponsive (ephemeral memory), and the case in which nodes are always responsive (persistent memory). We then defined a dynamic snapshot object that allows users to change the set of processes whose values should be returned by a scan operation, and presented implementations of this object in the persistent and ephemeral memory models.

Our algorithm has quadratic time complexity, and since it is based on a quadratic-complexity static algorithm [2], we cannot expect any better from our algorithm. An interesting question for future research is to determine whether more efficient algorithms exist, given that for static snapshots, $O(m \cdot log(m))$ algorithms are known [8].

Our notion of ephemeral memory is interesting in its own right because of its generality. It can be applied to message-passing models: Each node can be emulated on top of a number of servers (e.g., using ABD [6]), and our responsiveness definition abstracts away the need to deal explicitly with the failure model of the emulation algorithm. Therefore, another interesting future direction is to try to implement dynamic reliable storage [5, 38, 23, 27] in the ephemeral memory model.

References

1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

3 Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. Benign failure models for shared memory. In *Distributed Algorithms*. Springer, 1993.

4 Marcos K Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2):36–59, 2004.

5 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011.

6 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1), 1995.

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

8 Hagit Attiya and Ophir Rachman. Atomic snapshots in o (n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.

9 Jim Basney and Miron Livny. Managing network resources in condor. In *hpdc*, pages 298–299, 2000.

10 Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *ACM SIGARCH Computer Architecture News*, 32(5):235–247, 2004.

11 Christian Cachin. Architecture of the hyperledger blockchain fabric. 2016.

12 K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

13 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

14 Naser Ezzati-Jivan and Michel R Dagenais. A framework to compute statistics of system parameters from very large trace files. *ACM SIGOPS Operating Systems Review*, 47(1):43–54, 2013.

15 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 140–153. Springer, 2015.

16 Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 161–169. ACM, 2001.

17 MD Gan, ZJ Ding, SG Wang, WH Wu, and MC Zhou. Deadlock control of multithreaded software based on petri nets: A brief review. In *2016 IEEE 13th International Conference on Networking, Sensing, and Control (ICNSC)*, pages 1–5. IEEE, 2016.

18 Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

19 Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9. IEEE Computer Society, 2005.

20 Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

21 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

22 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3), 1998.

23 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

24 Nikolaos D Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

25 Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

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

28 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

29 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 239–248. IEEE, 2015.

30 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks, and Games*, pages 358–369. Springer, 2015.

31 Rolf Riesen, Kurt Ferreira, Duma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G Bridges. Alleviating scalability issues of checkpointing protocols. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

32 Alexander Borisovitch Romanovsky and Yi-Min Wang. Method for deadlock recovery using consistent global checkpoints, September 2 1997. US Patent 5,664,088.

33 Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.

34 Peter Scheuermann and Hsiang-Lung Tung. A deadlock checkpointing scheme for multidatabase systems. In *Research Issues on Data Engineering, 1992: Transaction and Query Processing, Second International Workshop on*, pages 184–191. IEEE, 1992.

35 Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

36 Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

37 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. *arXiv preprint arXiv:1507.07086*, 2015.

38 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *OPODIS*, pages 259–259, 2015.

39 Alexender Spiegelman and Idit Keidar. Dynamic atomic snapshots. Technical Report CCIT 907, EE, Technion, November 2016. `http://webee.technion.ac.il/publication-link/index/id/703`.

40 Nathan Stone, John Kochmar, Raghurama Reddy, J Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. *Pittsburgh Supercomputing Center, Tech. Rep*, 2001.

41 Zheng Zhang. Checkpoint computer system utilizing a fifo buffer to re-synchronize the memory systems on the detection of an error, May 8 2001. US Patent 6,230,282.

A Definitions: Runs, Global States, and Algorithms

An *algorithm* defines the behavior of processes as deterministic state machines, where a high-level operation performs a series of low-level invoke and respond *actions* on variables, starting with the high-level operation's invocation and ending with its response; where a process p_i's action may change p_i's local state as well as $segment_i$. A *global state* is a mapping to states from system components, i.e., processes and nodes. An *initial global state* is one where all components are in initial states specified by the algorithm. A *run* of algorithm A is a (finite or infinite) alternating sequence of global states and actions, beginning with some initial global state, such that global state transitions occur according to A. A *run fragment* is a contiguous subsequence of a run.

B Mutable Object Impossibility

Here we prove that without an explicit *add* operation no meaningful object can be emulated from SWMR registers. An illustration of the proof of the following theorem can be found in Figure 4. We start with a definition of mutable objects.

Mutable objects. For a global state c and operation a, we denote by $c.a_i$ the sequential run fragment of a by process p_i from the global state c. Intuitively, a mutable object is one that can be changed by a process, in the sense that a mutating operation can change the value returned by another operation. Formally, a *mutable object* is an object that has operations a, b, possibly $a = b$, s.t. there exists a global state c, s.t. for every pair of processes p_i, p_j, $c.a_i$ returns a different value than a_i in $c.b_j.a_i$. In this case we say that b *mutates* the object.

For example, an MWMR register is a mutable object, where a is a *read* operation and b is a $write(v)$ operation s.t. $v \neq v'$, where $write(v')$ is the last write to complete before c. Another example is an atomic snapshot [2] object, where a is a *scan* and b is an *update*.

▶ **Theorem 2.** *A wait-free mutable object cannot be emulated from SWMR registers if infinitely many processes are allowed to invoke high-level operations at any time.*

Proof. Assume to the contrary that there is some object with operations a and b as in the definition of mutable objects. Consider an initial global state c. We construct a sequential run in which the mutability of the object is contradicted. Consider a solo run of a_i from c, $c.a_i$. By wait-freedom, the operation completes and returns some value v after reading only finitely many registers (reading infinitely many registers cannot be wait-free). Therefore, there is a process p_j that none of its registers are read through the run. Now consider another run in which p_j executes b solo from c and completes in some global state c'. In global states c and c', all registers except ones written by p_j hold the same value, so a solo run of a_i from c' also completes, and the steps of a_i are the same in both runs. Since a does not read any of p_j's registers, the value returned from $c.b_j.a_i$ is also v. A contradiction to the assumption that b mutates the object.

◀

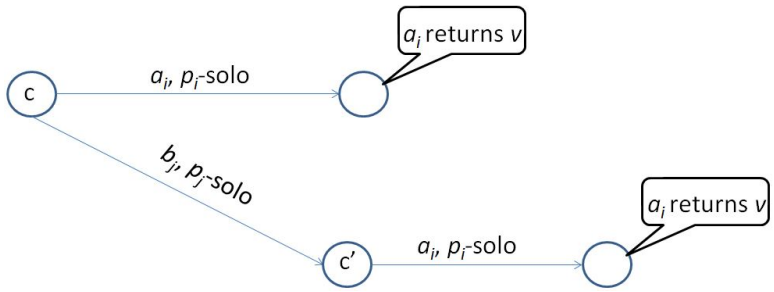

Figure 4 Theorem 2 proof illustration: impossibility of wait-free mutable object emulation.

C Correctness Proof

We prove here the correctness of our algorithm for dynamic snapshots in the ephemeral memory model (Section 6).

Notation. We denote the view of local variable v at process p_i as v_i, e.g., $PrevView_i$ is p_i's $PrevView$. We say that a process p commits $add(n)$ ($remove(n)$) when it writes $\langle add, n, commit\rangle$ (respectively, $\langle remove, n, commit\rangle$) to its segment. Recall that we say that node n is *added* (*removed*) when $add(n)$ (respectively, $remove(n)$) is committed for the first time. We will show that the linearization point of an $add(n)$ ($remove(n)$) operation is when n is added (respectively, removed). Recall also that a *snapshot(t)* is a mapping from every added and not removed node at time t to its value at time t. In order to prove correctness we show that every scan operation that is invoked at time t_1 and completes at time t_2 returns *snapshot(t')* for some t' s.t. $t_1 < t' < t_2$.

We begin with the following observation:

▶ **Observation 1.** *Consider some time t in a run of the algorithm when some process p_i is executing embeddedScan. Then $N_0 \subseteq PrevView_i.mem \cup PrevView_i.removed$ at time t.*

Our proof is based on the following key property:

▶ **Property 1** (Remove propagation up to time t)**.** *For every two processes p_i, p_j, if $segment_j.changes$ includes $\langle remove, n_i, commit\rangle$ at time t, then at time t, $segment_j.changes$ includes every $\langle OP, NODE, commit\rangle$ ever included in $segment_i.changes$.*

We first show that Property 1 implies certain pertinent properties about segment views (Lemma 3 to Corollary 6) and proceed to prove it by induction for all t.

▶ **Lemma 3.** *Consider some time t in a run of the algorithm when some process p_i is at the beginning of some collect in an execution of embeddedScan (line 5 in Algorithm 4). Assume that Property 1 is true for time t. Assume also that there exists either a node not in $PrevView_i.mem$ at time t that has been added and not removed before time t, or a node in $PrevView_i.mem$ that has been removed before time t. Then there exists $p_k \in PrevView_i.mem$ s.t. $PrevView_i.snap[k].changes \neq segment_k.changes$ at time t.*

Proof. By the second assumption, there is an operation (*add* or *remove*) *op*, committed before time t, that p_i did not see. Now pick a process that commits *op* before time t and denote it by p_{j_1}. Node n_{j_1} is either in N_0 or it has been added by another process p_{j_2} before time t. Node n_{j_2} is also either in N_0 or it has been added by another process p_{j_3} before time t, and so

on. The number of added nodes before time t is finite. Thus, there is a sequence $p_{j_1}, \ldots, p_{j_n}$ s.t. for every $0 \leq l < n$ node n_{j_l} has been added by $p_{j_{l+1}}$ before time t, and $n_{j_n} \in N_0$. Now let p_{j_k} be the process with the lowest k s.t. $n_{j_k} \in PrevView_i.mem \cup PrevView_i.removed$ at time t. It is guaranteed that there is such process because $n_{j_n} \in N_0$ and by Observation 1, $N_0 \subseteq PrevView_i.mem \cup PrevView_i.removed$ at time t.

Note that if $k = 1$, process p_{j_k} commits *op* before time t. Otherwise, p_{j_k} commits $add(n_{j_{k-1}})$ before time t. Now consider two cases:

- First, $n_{j_k} \in PrevView_i.mem$. In this case, $segment_{j_k}.changes$ contains $\langle add, n_{j_{k-1}}, commit \rangle$ (if $k > 1$, or *op*'s commit otherwise) at time t, whereas $PrevView_i.snap[j_k].changes$ does not. Otherwise, p_i would have added $n_{j_{k-1}}$ to $PrevView_i.mem$ earlier (if $k > 1$, or see *op* otherwise), and we are done.
- Second, $n_{j_k} \in PrevView_i.removed$. In this case, $\langle remove, p_{j_k}, commit \rangle \in segment_i.changes$. Thus, by the first assumption (Property 1), $\langle add, n_{j_{k-1}}, commit \rangle \in segment_i.changes$, and thus $n_{j_{k-1}} \in PrevView_i.mem \cup PrevView_i.removed$ at time t (if $k > 1$, or p_i sees *op*'s commit otherwise before time t). A contradiction.

◀

▶ **Lemma 4.** *Assume that Property 1 is true for time t. Assume also that some process p_i completes a successful double collect in a run of the algorithm during an execution of embeddedScan, and let t be the time when the second collect begins. Then at time t, $PrevView_i.snap = snapshot(t)$, and the value written to $segment_i.view$ at the end of the embeddedScan is snapshot(t).*

Proof. At time t, $PrevView_i.snap$ holds the values returned from the first collect. We first show that $\forall n_j \in PrevView_i.mem$, $PrevView_i.snap[j]$ is equal to $segment_j$ at time t. Assume the contrary, then p_j wrote to $segment_j$ in the interval between the read of $segment_j$ in the first collect and time t. The values of $segment_j$ cannot repeat themselves because $segment_j.num$ is increased at the beginning of every operation. Therefore, the value read from $segment_j$ in the second collect is different from the one read from $segment_j$ in the first collect. A contradiction to the successful double collect.

We get, in particular, that $PrevView_i.snap[j].changes = segment_j.changes$ for all $n_j \in PrevView.mem$. By the contrapositive of Lemma 3, $PrevView_i.mem$ contains at time t all the processes that have been added and not removed before time t. Therefore, at time t, $PrevView_i.snap$ is *snapshot(t)*. After a successful double collect, p_i stops the iterations and writes $PrevView_i$ to its *segment.view*.

◀

▶ **Lemma 5.** *Consider some process p_i that begins an embeddedScan at some time t_s and completes at time t_e. Assume that Property 1 is true for every time $t \leq t_1$, $t_e > t_1 > t_s$. If at time t_1 p_i reads some $segment_j$ s.t. $segment_j.view.snap[i].num = segment_i.num$, then the value of $segment_j.view.snap$ at time t_1 is a snapshot(t_2) for some time $t_e > t_2 > t_s$.*

Proof. Let V be the value of $segment_j.view$ at time t_1. First note that every segment value is either the initial value, or obtained by a successful double collect, or borrowed from another process' segment. Since the number of *embeddedScans* that complete before time t_1 is finite and initial values are not borrowed, it follows by induction that every non-initial segment value is the result of a successful double collect by some process. Since $segment_j.num > 0$, $segment_j$ is the result of a successful double collect D obtained by some process p_l, possibly $j = l$, not after time t_1. Now recall that process p_i increases $segment_i.num$ at the beginning

of its *embeddedScans*, and since $segment_j.view.snap[i].num = segment_i.num$, we get that p_l reads p_i's segment during the first collect of D, after p_i increases its version number, i.e., after time t_s. Let t_2 be the time at the beginning of the second collect of D, and notice that $t_e > t_1 > t_2 > t_s$. By Lemma 4, the value of $PrevView_l.snap$ at t_1 is *snapshot(*t_2*)*. Therefore V is *snapshot(*t_2*)*. The lemma follows.

◀

▶ **Corollary 6.** *Consider time t in a run of the algorithm. Assume that Property 1 is true for every time* $t' \leq t$*. Then every* embeddedScan *that completes before t returns* snapshot(t') *for some time* t' *in the* embeddedScan *interval.*

We are now ready to prove our key claim:

Claim 2 (restated). *For every time t, Property 1 holds.*

Proof. We prove by induction on t.

Base: $t = 0$. Since no remove operations have been committed yet, the claim trivially holds.

Step: Assume that the claim holds for every time $0 \leq t' \leq t$, we prove that the claim holds for $t + 1$. By Corollary 6, every *embeddedScan* that completes before $t + 1$ returns *snapshot(t')* for some time t' in the *embeddedScan* interval. Let p_j be a process that writes $\langle remove, p_i, commit\rangle$ to $segment_j.changes$ at time $t+1$, and let $\langle OP, NODE, commit\rangle$ be a commit ever written by p_i to $segment_i.changes$. We need to show that $\langle OP, NODE, commit\rangle \in segment_j.changes$ at time $t+1$. Since p_j writes $\langle remove, p_i, commit\rangle$ to $segment_j.changes$ at time $t + 1$, it writes $\langle remove, p_i, propose\rangle$ to $segment_j.changes$ at time $t^j_{propose} < t + 1$ and some process p_l (possibly p_j) writes $\langle remove, p_i, prepare\rangle$ to $segment_l.changes$ at time $t^l_{prepare} < t_{ES^j_2} < t^j_{propose}$ s.t. p_j's second *embeddedScan* returns *snapshot(*$t_{ES^j_2}$*)*. Denote the third *embeddedScan* performed by p_i during the operation that commits $OP(NODE)$ by ES^i_3. Now consider three cases according to ES^i_3.

- First, ES^i_3 returns *snapshot(*$t_{ES^i_3}$*)* for some $t_{ES^i_3} < t^l_{prepare}$. Thus, p_i writes $\langle OP, NODE, propose\rangle$ to $segment_i.changes$ before time $t^l_{prepare}$. We now show that p_j sees it and writes $\langle OP, NODE, commit\rangle$ to $segment_j.changes$ at time $t + 1$. Consider two cases:
  - First, p_j reads $segment_i$ during its second *embeddedScan*. In this case, p_j sees $\langle OP, NODE, propose\rangle$ in $segment_i$ its second *embeddedScan*, and thus p_j writes $\langle OP, NODE, commit\rangle$ to $segment_j.changes$ at time $t + 1$.
  - Second, p_j skips $segment_i$ because it sees $\langle remove, p_i, commit\rangle$ at some $segment_k$ during its second *embeddedScan*. By the induction assumption, p_j also sees $\langle OP, NODE, commit\rangle$ in $segment_k$, and thus p_j writes $\langle OP, NODE, commit\rangle$ to $segment_j.changes$ at time $t + 1$.
- Second, ES^i_3 returns *snapshot(*$t_{ES^i_3}$*)* for some $t_{ES^i_3}$, $t^l_{prepare} < t_{ES^i_3} < t^j_{propose}$. Note that $segment_l$ includes $\langle remove, p_i, prepare\rangle$ throughout ES^i_3. Consider two cases:
  - First, p_i sees $\langle remove, p_i, prepare\rangle$ in $segment_l.changes$ during ES^i_3, and thus stops before writing $\langle OP, NODE, commit\rangle$ to $segment_i.changes$. A contradiction.
  - Second, p_i does not see $segment_l$ in ES^i_3. Since p_l is added before ES^i_3 begins, it must be the case that p_i sees $\langle remove, p_l, commit\rangle$ during ES^i_3. Now note that p_j's third *embeddedScan* starts after $t^j_{propose}$, and thus after $t_{ES^i_3}$. Therefore, p_j sees $\langle remove, p_l, commit\rangle$ during its third *embeddedScan*. Therefore, it does not write $\langle remove, p_i, commit\rangle$ to $segment_j$ at time $t + 1$. A contradiction.

- Third, ES_3^i does not return *snapshot*$(t_{ES_3^i})$ s.t. $t_{ES_3^i} < t^j_{propose}$. Since p_i does not stop after ES_3^i, p_i does not read $segment_j$ during its ES_3^i. Therefore, p_i sees $\langle remove, p_j, commit\rangle$ in some $segment_{k_1}$. Note that p_i does not see $\langle remove, p_i, propose\rangle$ in $segment_{k_1}$. Otherwise it would have stopped. Therefore, there is a sequence $p_i, p_{k_1}, \ldots, p_{k_n}$ (possibly, $n = 1$) s.t. each process in the sequence except p_{k_n} reads $\langle remove, p_j, commit\rangle$ and not $\langle remove, p_i, propose\rangle$ from the segment of the consecutive process during its third *embeddedScan*, and p_{k_n} writes $\langle remove, p_j, commit\rangle$ to its segment and sees neither $\langle remove, p_j, commit\rangle$ nor $\langle remove, p_i, propose\rangle$ during its third *embeddedScan*, $ES_3^{k_n}$. Therefore, $ES_3^{k_n}$ reads from $segment_j$, and returns *snapshot*$(t_{ES_3^{k_n}})$ s.t. $t_{ES_3^{k_n}} < t^j_{propose}$. Thus, p_{k_n} writes $\langle remove, p_j, propose\rangle$ to $segment_{k_n}$ before $t^j_{propose}$. Now note that p_j's third *embeddedScan*, ES_3^j, begins after $t^j_{propose}$, and consider two options:
  - First, p_j reads $segment_{k_n}$ during ES_3^j. In this case p_j sees $\langle remove, p_j, propose\rangle$, and stops before writing $\langle remove, p_i, commit\rangle$ to its segment. A contradiction.
  - Second, p_j reads $\langle remove, p_{k_n}, commit\rangle$ from some $segment_r$ during or before ES_3^j. Since ES_3^j completes before time $t + 1$, by the induction assumption, p_j reads $\langle remove, p_j, commit\rangle$ in $segment_r$ as well. Therefore, p_j stops before writing $\langle remove, p_i, commit\rangle$ to its segment. A contradiction.

◀

▶ **Theorem 7.** *The algorithm presented in Algorithms 1,* update *and* scan *operations in Algorithm 2, Algorithm 4, and Algorithm 5 implements dynamic atomic snapshot.*

Proof. Let r be a run of the algorithm and let r_s be a sequential run s.t. the operation in r_s are ordered according to their linearization points in r. Now consider some process p_i that invokes a *scan* operation at time t_s in r. Assume that the *scan* operation completes at some time $t_e > t_s$ in r and returns V. By Claim 2 and Corollary 6, V is a *snapshot(t)* for some $t_s < t < t_e$, and thus r_s satisfies the dynamic snapshot's sequential specification. Therefore, r_s is a linearization of r.

◀

Note that the wait-freedom of our algorithm follows from Claim 3 (Section 6.2).