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.

© Alexander Spiegelman and Idit Keidar; licensed under Creative Commons License CC-BY
1 Introduction

Atomic snapshots [2, 12] are essential building 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 persistent 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 \textit{add} operations before they can invoke operations. Similarly, an explicit \textit{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 \textit{dynamic snapshot} abstraction, which we emulate in this paper.

3.1 Preliminaries

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

Processes Processes may \textit{fail} by crashing or by invoking an explicit \textit{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 history $H$ 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 $\bot$ 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 \cite{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 \cite{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 $(add/remove, n_i)$. 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 \( \text{mem} \cup \text{removed} \); then checks their \text{changes} sets to discover which processes were included or excluded and updates \text{mem} \text{ and } \text{removed} \text{ 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 \text{scan} operation. Consider a run, illustrated in Figure 1, in which some process performs a \text{scan} concurrently with infinitely many \text{add} operations, s.t. every process performs exactly one \text{add} and no updates. One way for a \text{scan} to complete is by obtaining a successful double collect, but in this case, because of the infinitely many \text{add} operations, the \text{scan} can never obtain one despite the fact that there are no updates. Alternatively, a \text{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 \( \text{num} \), to every segment and include it in the embedded view. Each process increases its \( \text{num} \) at the beginning of every \text{scan} operation, and in every collect it checks whether some process has a view that contains its own updated \( \text{num} \). 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.

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

At any time \( t \), we define \text{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 segment₁. In this example $N_0 = \{n_1, n_2\}$, process $p_2$ has not invoked any operation yet, and process $p_1$ completed $\text{add}(n_3)$, including writing 1 to $\text{segment}_1$.num, performing embeddedScan and writing the result to $\text{segment}_1$.view, and finally writing $\langle \text{add}, n_3 \rangle$ to $\text{segment}_1$.changes.

Algorithm 1 Segment structure.

$\text{segment} = \langle \text{val}, \text{changes}, \text{num}, \text{view} \rangle$

where $\text{view} = \langle \text{mem}, \text{removed}, \text{snap} \rangle$

where $\text{mem, removed} \subseteq \mathbb{N}$, and $\text{snap}$ is a mapping from $\text{mem}$ to tuples $\langle \text{val}, \text{changes}, \text{num} \rangle$

initially: if $n_i \in N_0$, $\text{segment}_i = \langle \bot, \{\}, 0, \langle N_0, \langle \bot, \{\}, 0 \rangle^{|N_0|} \rangle \rangle$, else $\text{segment}_i = \bot$

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

1: procedure $\text{scan}_i()$
2: \quad $\text{embeddedScan}()$
3: \quad for each $n_j \in \text{segment}_i$.view.mem
4: \quad \quad $V[j] = \text{segment}_i$.view.snap[j].val
5: \quad return $V$
6: procedure $\text{update}_i(d)$
7: \quad $\text{embeddedScan}()$
8: \quad $\text{segment}_i$.val $\leftarrow d$
9: procedure $\text{add}_i(n_j)$
10: \quad $\text{embeddedScan}()$
11: \quad $\text{segment}_i \leftarrow \langle \bot, \{\}, 0, \text{segment}_i$.view $\rangle$ $\triangleright$ set $\text{segment}_i$’s initial value
12: \quad $\text{segment}_i$.changes $\leftarrow \text{segment}_i$.changes $\cup \{\langle \text{add}, n_j \rangle\}$
13: procedure $\text{remove}_i(n_j)$
14: \quad $\text{embeddedScan}()$
15: \quad $\text{segment}_i$.changes $\leftarrow \text{segment}_i$.changes $\cup \{\langle \text{remove}, n_j \rangle\}$

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: \textbf{procedure} \( \text{embeddedScan}() \),
17: \hspace{1em} \( \text{PrevView} \leftarrow \text{segment}_{i,\text{view}} \)
18: \hspace{1em} \( \text{segment}_{i,\text{num}} \leftarrow \text{segment}_{i,\text{num}} + 1 \) \hspace{2em} \( \triangleright \) increase version number
19: \hspace{1em} \textbf{while} true \hspace{1em} \( \triangleright \) try to obtain a consistent snapshot
20: \hspace{2em} \( \text{CurView.mem} \leftarrow \text{PrevView.mem} \)
21: \hspace{2em} \( \text{CurView.removed} \leftarrow \text{PrevView.removed} \)
22: \hspace{2em} \textbf{for each} \( n_j \in \text{CurView.mem} \cup \text{CurView.removed} \) \hspace{1em} \( \triangleright \) collect
23: \hspace{2em} \( \text{CurView.snap}[j] \leftarrow (\text{segment}_{j,\text{val}}, \text{segment}_{j,\text{changes}}, \text{segment}_{j,\text{num}}) \)
24: \hspace{2em} \textbf{if} \( \text{CurView} = \text{PrevView} \) \hspace{1em} \( \triangleright \) successful double collect
25: \hspace{2em} \textbf{goto} Done
26: \hspace{2em} \textbf{for each} \( n_j \in \text{CurView.mem} \cup \text{CurView.removed} \)
27: \hspace{2em} \textbf{if} \( \text{segment}_{j,\text{view}.\text{snap}[i].\text{num}} = \text{segment}_{i,\text{num}} \) \hspace{1em} \( \triangleright \) found a fresh snapshot
28: \hspace{2em} \( \text{CurView} \leftarrow \text{segment}_{j,\text{view}} \)
29: \hspace{2em} \textbf{goto} Done
30: \hspace{2em} \textbf{for each} \( (\text{OP}, n_i) \in \text{CurView.snap[j].changes} \setminus \text{PrevView.snap[j].changes} \) \hspace{1em} \( \triangleright \) update view
31: \hspace{3em} \textbf{if} \( \text{OP} = \text{add} \land n_i \notin \text{PrevView.removed} \)
32: \hspace{4em} \( \text{PrevView}.\text{mem} \leftarrow \text{PrevView}.\text{mem} \cup \{n_i\} \)
33: \hspace{4em} \( \text{PrevView}.\text{snap}[i] \leftarrow (\perp, \{\}, 0) \)
34: \hspace{3em} \textbf{else} \)
35: \hspace{4em} \( \text{PrevView}.\text{mem} \leftarrow \text{PrevView}.\text{mem} \setminus \{n_i\} \)
36: \hspace{4em} \( \text{PrevView}.\text{removed} \leftarrow \text{PrevView}.\text{removed} \cup \{n_i\} \)
37: \hspace{4em} \( \text{PrevView}.\text{snap}[j] \leftarrow \text{CurView}.\text{snap}[j] \)
38: \hspace{2em} \textbf{if} \( \exists j \text{ s.t. } (\text{remove}, n_i) \in \text{CurView}.\text{snap[j].changes} \) then stop \hspace{1em} \( \triangleright \) \( n_i \) was excluded
39: \hspace{1em} \( \text{segment}_{i,\text{view}} \leftarrow \text{CurView} \)

```

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, \( \text{CurView} \) and \( \text{PrevView} \). Each of them
is structured like \( \text{view} \), consisting of \( \text{mem} \), \( \text{removed} \), and \( \text{snap} \). In every iteration after the
first, \( \text{PrevView} \) 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 \( \text{CurView} \). Note that
we collect segments not only from nodes in the current \( \text{mem} \), 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, \( \text{PrevView} \) is updated according to \( \text{CurView} \). 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 \(_i\).changes includes \( \langle \text{remove}, n_i, \text{commit} \rangle \) at time \( t \), then at time \( t \), segment \(_j\).changes includes every \( \langle \text{OP, NODE, commit} \rangle \) 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 \( \text{embeddedScan} \) 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 \( \text{changes} \) set with all such information observed during an \( \text{embeddedScan} \). The new algorithm’s \( \text{embeddedScan} \) procedure is presented in Algorithm 4. The segment structure remains as in Algorithm 1. The \( \text{embeddedScan} \) uses a local set \( \text{Changes} \) to track the information observed during its iterations, and segment \(_i\).changes is updated according to \( \text{Changes} \) at the end of the procedure.

When a process \( p \) tries to read from a removed node in line 9 during an \( \text{embeddedScan} \), 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 \( \text{PrevView} \) and \( \text{Changes} \), and jumps to the beginning of the next iteration (Loop) to collect from the new \( \text{mem} \) 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 \( \text{embeddedScan} \). 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 ⟨add, n, propose⟩ 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 ⟨add, n, commit⟩ 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:   PreView ← segment.$i$.view
3:   Changes ← segment.$i$.changes
4:   segment.$i$.num ← segment.$i$.num + 1
5:   while true
6:     CurView.mem ← PreView.mem
7:     CurView.removed ← PreView.removed
8:     for each $n_j \in$ CurView.mem s.t. $x \neq y$
9:       CurView.snap[$j$] ← ⟨segment.$j$.value, segment.$j$.changes, segment.$j$.num⟩
10:      if CurView=PreView
11:         goto Done
12:      for each $n_j \in$ CurView mem s.t. $x \neq y$
13:        if segment.$j$.view.snap[$i$].num = segment.$i$.num
14:           CurView ← segment.$j$.view
15:         goto Done
16:      for each ⟨OP, n, l⟩ \in CurView.snap[$j$].changes \ Changes
17:        if OP = add \ n \notin PreView.removed
18:           PreView.mem ← PreView.mem \ {n}
19:           PreView.snap[$i$] ← ⟨⊥, {n}, 0⟩
20:        else
21:           PreView.mem ← PreView.mem \ {n}
22:           PreView.removed ← PreView.removed \ {n}
23:           Changes ← Changes \ CurView.snap[$j$].changes
24:           PreView.snap[$j$] ← CurView.snap[$j$]
25:         segment.$i$.num ← segment.$i$.num + 1
26:       if ⟨remove, n, l⟩ \in segment.$i$.changes then stop
27:   upon exception(Seg)
28:     if Seg.removed \ PreView.removed ≠ {} \ found new removed node, jump forward
29:       PreView ← Seg.view
30:       Changes ← Seg.changes
31:       goto Loop
32:     else retry read

OPODIS 2016
A remove operation consists of three phases. A process \( p_i \) that performs \( \text{remove}(n_j) \) first calls \( \text{embeddedScan} \), then adds \( \langle \text{remove}, n_j, \text{prepare} \rangle \) 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 \( \text{embeddedScan} \) 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 \( \langle \text{OP}, \text{NODE}, \text{propose} \rangle \) to its changes set for every \( \langle \text{OP}, \text{NODE}, \text{propose} \rangle \) it has observed in segment\(_i\).\text{changes} during its last \( \text{embeddedScan} \) together with \( \langle \text{remove}, p_j, \text{propose} \rangle \). 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 \( \text{add}(n) \) or \( \text{remove}(n) \) concurrently with a \( \text{remove}(n_1) \) operation by another process \( p_2 \), then either (1) \( p_1 \) observes \( \langle \text{remove}, n_1, \text{prepare} \rangle \) before committing its operation and stops, or (2) \( p_2 \) observes \( p_1 \)'s \( \langle \text{OP}, n, \text{propose} \rangle \) and proposes it together with \( \text{remove}(n_1) \).

In the third phase \( p_i \) calls \( \text{embeddedScan} \) again, but this time it serves two different purposes: First, as in add, it checks (at the end of the \( \text{embeddedScan} \)) 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 \( \text{remove}(p_j) \), in which case it completes the operation without committing \( p_j \)'s proposals. Otherwise, \( p_i \) commits all its proposals, i.e., it adds \( \langle \text{OP}, \text{NODE}, \text{commit} \rangle \) to its changes set for every \( \langle \text{OP}, \text{NODE}, \text{propose} \rangle \) 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 \( p_j \)'s proposals and committed \( p_j \)'s removal without them. Hence, committing them know violates Claim 2.

The linearization point of an \( \text{add}(n) \) or \( \text{remove}(n) \) operation is when \( \langle \text{add}, n, \text{commit} \rangle \) or \( \langle \text{remove}, n, \text{commit} \rangle \) 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 \( \text{embeddedScan} \) 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 \( \text{embeddedScan} \) 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 \( \text{embeddedScan} \) 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\(_j\).\text{view}.\text{snap}[i].\text{num} \lt segment\(_i\).\text{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,
Alexander Spiegelman and Idit Keidar 23:13

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_j.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: \textit{embeddedScan}() \quad \triangleright \text{phase 1: propose}
35: \textit{segment}_i \leftarrow \langle \bot, \textit{segment}_i\.changes, 0, \textit{segment}_i\.view \rangle \quad \triangleright \text{set segment}_i\.s initial value
36: \textit{segment}_i\.changes \leftarrow \textit{segment}_i\.changes \cup \{\langle \textit{add}, n_j, \textit{propose} \rangle\}
37: \textit{embeddedScan}() \quad \triangleright \text{phase 2: commit}
38: \textit{segment}_i\.changes \leftarrow \textit{segment}_i\.changes \cup \{\langle \textit{add}, n_j, \textit{commit} \rangle\}

39: procedure remove_i(n_j)
40: \textit{embeddedScan}() \quad \triangleright \text{phase 1: prepare}
41: \textit{segment}_i\.changes \leftarrow \textit{segment}_i\.changes \cup \{\langle \textit{remove}, n_j, \textit{prepare} \rangle\}
42: \textit{embeddedScan}() \quad \triangleright \text{phase 2: propose}
43: \textit{ProposeSet} = \{\langle *, *, \textit{propose} \rangle \in \textit{segment}_i\.snap[j].changes \cup \{\langle \textit{remove}, n_j, \textit{propose} \rangle\}\}
44: \textit{segment}_i\.changes \leftarrow \textit{segment}_i\.changes \cup \textit{ProposeSet}
45: \textit{embeddedScan}() \quad \triangleright \text{phase 3: commit}
46: \textbf{if} (\langle \textit{remove}, p_j, \textit{commits} \rangle \notin \textit{segment}_i\.changes)
47: \quad \textit{CommitSet} = \{\langle \textit{OP}, \textit{NODE}, \textit{commit} \rangle | \langle \textit{OP}, \textit{NODE}, \textit{propose} \rangle \in \textit{ProposedSet}\}
48: \quad \textit{segment}_i\.changes \leftarrow \textit{segment}_i\.changes \cup \textit{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


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.
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., \( \text{PrevView}_i \) is \( p_i \)'s \( \text{PrevView} \). We say that a process \( p \) commits \( \langle \text{add}, n, \text{commit} \rangle \) (respectively, \( \langle \text{remove}, n, \text{commit} \rangle \)) when it writes \( \langle \text{add}, n, \text{commit} \rangle \) (respectively, \( \langle \text{remove}, n, \text{commit} \rangle \)) to its segment. Recall that we say that node \( n \) is added (removed) when \( \langle \text{add}, n \rangle \) (respectively, \( \langle \text{remove}, n \rangle \)) is committed for the first time. We will show that the linearization point of an \( \langle \text{add}, n \rangle \) (respectively, \( \langle \text{remove}, n \rangle \)) 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:

\begin{itemize}
  \item \textbf{Observation 1.} Consider some time \( t \) in a run of the algorithm when some process \( p_i \) is executing \texttt{embeddedScan}. Then \( N_0 \subseteq \text{PrevView}_i.\text{mem} \cup \text{PrevView}_i.\text{removed} \) at time \( t \).
\end{itemize}

Our proof is based on the following key property:

\begin{itemize}
  \item \textbf{Property 1 (Remove propagation up to time \( t \)).} For every two processes \( p_i, p_j \), if \( \text{segment}_j.\text{changes} \) includes \( \langle \text{remove}, n, \text{commit} \rangle \) at time \( t \), then at time \( t \), \( \text{segment}_j.\text{changes} \) includes every \( \langle \text{OP, NODE, commit} \rangle \) ever included in \( \text{segment}_i.\text{changes} \).
\end{itemize}

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 \( t \).

\begin{itemize}
  \item \textbf{Lemma 3.} Consider some time \( t \) in a run of the algorithm when some process \( p_k \) is at the beginning of some collect in an execution of \texttt{embeddedScan} (line 5 in Algorithm 4). Assume that Property 1 is true for time \( t \). Assume also that there exists a node not in \( \text{PrevView}_i.\text{mem} \) at time \( t \) that has been added and not removed before time \( t \), or a node in \( \text{PrevView}_i.\text{mem} \) that has been removed before time \( t \). Then there exists \( p_k \in \text{PrevView}_i.\text{mem} \) s.t. \( \text{PrevView}_i.\text{snap}[k].\text{changes} \neq \text{segment}_k.\text{changes} \) at time \( t \).
\end{itemize}

\begin{itemize}
  \item \textbf{Proof.} By the second assumption, there is an operation \( \langle \text{add or remove} \rangle \text{op} \), committed before time \( t \), that \( p_i \) did not see. Now pick a process that commits \( \text{op} \) 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
\end{itemize}
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. $n_{j_k} \in \text{PrevView}.\text{mem} \cup \text{PrevView}.\text{removed}$ at time $t$. It is guaranteed that there is such process because $n_{j_n} \in N_0$ and by Observation 1, $N_0 \subseteq \text{PrevView}.\text{mem} \cup \text{PrevView}.\text{removed}$ at time $t$.

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

- First, $n_{j_k} \in \text{PrevView}.\text{mem}$. In this case, segment$_{j_k}.\text{changes}$ contains $(\text{add}, n_{j_k-1}, \text{commit})$ (if $k > 1$, or $\text{op}$'s commit otherwise) at time $t$, whereas $\text{PrevView}.\text{snap}[j_k].\text{changes}$ does not. Otherwise, $p_i$ would have added $n_{j_k-1}$ to $\text{PrevView}.\text{mem}$ earlier (if $k > 1$, or see $\text{op}$ otherwise), and we are done.
- Second, $n_{j_k} \in \text{PrevView}.\text{removed}$. In this case, $(\text{remove}, p_{j_k}, \text{commit}) \in \text{segment}.\text{changes}$.

Thus, by the first assumption (Property 1), $(\text{add}, n_{j_k-1}, \text{commit}) \in \text{segment}.\text{changes}$, and thus $n_{j_k-1} \in \text{PrevView}.\text{mem} \cup \text{PrevView}.\text{removed}$ at time $t$ (if $k > 1$, or $p_i$ sees $\text{op}$'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 $t$, $\text{PrevView}.\text{snap} = \text{snapshot}(t)$, and the value written to segment$_i$.view at the end of the embeddedScan is snapshot$(t)$.

**Proof.** At time $t$, $\text{PrevView}.\text{snap}$ holds the values returned from the first collect. We first show that $\forall n_j \in \text{PrevView}.\text{mem}$, $\text{PrevView}.\text{snap}[j]$ 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 $\text{PrevView}.\text{snap}[j].\text{changes} = \text{segment}.\text{changes}$ for all $n_j \in \text{PrevView}.\text{mem}$. By the contrapositive of Lemma 3, $\text{PrevView}.\text{mem}$ contains at time $t$ all the processes that have been added and not removed before time $t$. Therefore, at time $t$, $\text{PrevView}.\text{snap}$ is snapshot$(t)$. After a successful double collect, $p_i$ stops the iterations and writes $\text{PrevView}_i$ to its segment$_i$.view.

**Lemma 5.** Consider some process $p_i$ that begins an embeddedScan at some time $t_s$ and completes at time $t$. Assume that Property 1 is true for every time $t \leq t_1$, $t_c > t_1 > t_s$. 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 $t_2 > t_1 > t_s$.

**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 $p_l$ increases segment$_l$.num at the beginning
of its `embeddedScans`, and since `segment_j.view.snap[i].num = segment_i.num`, we get that `p_i` 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_s > t_2 > t_2 > t_s`. By Lemma 4, the value of `PrevView_j.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' ≤ 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 ≤ t' ≤ 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 `{remove, p_i, commit}` to `segment_j.changes` at time `t + 1`, and let `{OP, NODE, commit}` be a commit ever written by `p_i` to `segment_i.changes`. We need to show that `{OP, NODE, commit} ∈ segment_j.changes` at time `t + 1`. Since `p_j` writes `{remove, p_i, commit}` to `segment_j.changes` at time `t + 1`, it writes `{remove, p_i, propose}` to `segment_j.changes` at time `t'_propose < t + 1` and some process `p_j` (possibly `p_i`) writes `{remove, p_i, prepare}` to `segment_j.changes` at time `t'_prepare < t'_propose` s.t. `p_j`'s second `embeddedScan` returns `snapshot(t'_ES_j)`. Denote the third `embeddedScan` performed by `p_i` during the operation that commits `OP(NODE)` by `ES_j^3`. Now consider three cases according to `ES_j^3`.

1. First, `ES_j^3` returns `snapshot(t'_ES_j) for some t'_ES_j < t'_prepare`. Thus, `p_i` writes `{OP, NODE, propose}` to `segment_j.changes` before time `t'_prepare`. We now show that `p_j` sees it and writes `{OP, NODE, commit}` 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 `{OP, NODE, propose}` in `segment_i` its second `embeddedScan`, and thus `p_j` writes `{OP, NODE, commit}` to `segment_j.changes` at time `t + 1`.
   - Second, `p_j` skips `segment_i` because it sees `{remove, p_i, commit}` at some `segment_k` during its second `embeddedScan`. By the induction assumption, `p_j` also sees `{OP, NODE, commit}` in `segment_k`, and thus `p_j` writes `{OP, NODE, commit}` to `segment_j.changes` at time `t + 1`.

2. Second, `ES_j^3` returns `snapshot(t'_ES_j) for some t'_ES_j < t'_prepare < t'_propose`. Note that `segment_j` includes `{remove, p_i, prepare}` throughout `ES_j^3`. Consider two cases:
   - First, `p_i` sees `{remove, p_i, prepare}` in `segment_j.changes` during `ES_j^3`, and thus stops before writing `{OP, NODE, commit}` to `segment_j.changes`. A contradiction.
   - Second, `p_i` does not see `segment_i` in `ES_j^3`. Since `p_i` is added before `ES_j^3` begins, it must be the case that `p_i` sees `{remove, p_i, commit}` during `ES_j^3`. Now note that `p_j`’s third `embeddedScan` starts after `t'_propose`, and thus after `t'_ES_j`. Therefore, `p_j` sees `{remove, p_i, commit}` during its third `embeddedScan`. Therefore, it does not write `{remove, p_i, commit}` to `segment_j` at time `t + 1`. A contradiction.
Third, $\text{ES}_3^i$ does not return $\text{snapshot}(t_{\text{ES}_3^i})$ s.t. $t_{\text{ES}_3^i} < t_{\text{propose}}^i$. Since $p_i$ does not stop after $\text{ES}_3^i$, $p_i$ does not read $\text{segment}_j$ during its $\text{ES}_3^i$. Therefore, $p_i$ sees $\langle \text{remove}, p_j, \text{commit} \rangle$ in some $\text{segment}_{k_i}$. Note that $p_i$ does not see $\langle \text{remove}, p_i, \text{propose} \rangle$ in $\text{segment}_{k_i}$. 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 $\langle \text{remove}, p_j, \text{commit} \rangle$ and not $\langle \text{remove}, p_i, \text{propose} \rangle$ from the segment of the consecutive process during its third $\text{embeddedScan}$, and $p_{k_n}$ writes $\langle \text{remove}, p_j, \text{commit} \rangle$ to its segment and sees neither $\langle \text{remove}, p_j, \text{commit} \rangle$ nor $\langle \text{remove}, p_i, \text{propose} \rangle$ during its third $\text{embeddedScan}$, $\text{ES}_3^{k_n}$. Therefore, $\text{ES}_3^{k_n}$ reads from $\text{segment}_j$, and returns $\text{snapshot}(t_{\text{ES}_3^{k_n}})$ s.t. $t_{\text{ES}_3^{k_n}} < t_{\text{propose}}^i$. Thus, $p_{k_n}$ writes $\langle \text{remove}, p_j, \text{propose} \rangle$ to $\text{segment}_{k_n}$ before $t_{\text{propose}}^i$. Now note that $p_j$’s third $\text{embeddedScan}$, $\text{ES}_3^j$, begins after $t_{\text{propose}}^i$, and consider two options:

- First, $p_j$ reads $\text{segment}_{k_n}$ during $\text{ES}_3^j$. In this case $p_j$ sees $\langle \text{remove}, p_j, \text{propose} \rangle$, and stops before writing $\langle \text{remove}, p_i, \text{commit} \rangle$ to its segment. A contradiction.
- Second, $p_j$ reads $\langle \text{remove}, p_{k_n}, \text{commit} \rangle$ from some $\text{segment}_r$ during or before $\text{ES}_3^j$. Since $\text{ES}_3^j$ completes before time $t + 1$, by the induction assumption, $p_j$ reads $\langle \text{remove}, p_j, \text{commit} \rangle$ in $\text{segment}_r$ as well. Therefore, $p_j$ stops before writing $\langle \text{remove}, p_i, \text{commit} \rangle$ to its segment. A contradiction.

\[\square\]

**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 $\text{scan}$ operation at time $t_s$ in $r$. Assume that the $\text{scan}$ operation completes at some time $t_e > t_s$ in $r$ and returns $V$. By Claim 2 and Corollary 6, $V$ is a $\text{snapshot}(t)$ 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 $r$.

\[\square\]

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