Self-Stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems

Chryssis Georgiou[1], Oskar Lundström[2], and Elad M. Schiller[2]

[1] Computer Science, University of Cyprus, Cyprus. chryssis@cs.ucy.ac.cy
[2] Computer Science and Engineering, Chalmers Univ. Tech.
{osklunds@student.,elad@}chalmers.se

Abstract. A *snapshot object* simulates the behavior of an array of single-writer/multi-reader shared registers that can be read atomically. Delporte-Gallet *et al.* proposed two fault-tolerant algorithms for snapshot objects in asynchronous crash-prone message-passing systems. Their first algorithm is *non-blocking*; it allows snapshot operations to terminate once all write operations had ceased. It uses $\mathcal{O}(n)$ messages of $\mathcal{O}(n \cdot \nu)$ bits, where n is the number of nodes and ν is the number of bits it takes to represent the object. Their second algorithm allows snapshot operations to *always terminate* independently of write operations. It incurs $\mathcal{O}(n^2)$ messages. The fault model of Delporte-Gallet *et al.* considers node failures (crashes). We aim at the design of even more robust snapshot objects. We do so through the lenses of *self-stabilization*—a very strong notion of fault-tolerance. In addition to Delporte-Gallet *et al.*'s fault model, a self-stabilizing algorithm can recover after the occurrence of *transient faults*; these faults represent arbitrary violations of the assumptions according to which the system was designed to operate (as long as the code stays intact). In particular, in this work, we propose self-stabilizing variations of Delporte-Gallet *et al.*'s non-blocking algorithm and always-terminating algorithm. Our algorithms have similar communication costs to the ones by Delporte-Gallet *et al.* and $\mathcal{O}(1)$ recovery time (in terms of asynchronous cycles) from transient faults. The main differences are that our proposal considers repeated gossiping of $\mathcal{O}(\nu)$ bits messages and deals with bounded space (which is a prerequisite for self-stabilization).

1 Introduction

We propose self-stabilizing implementations of shared memory snapshot objects for asynchronous bounded space networked systems whose nodes may crash.

Context and motivation. Shared registers are fundamental objects that facilitate synchronization in distributed systems. In the context of networked systems, they provide a higher abstraction level than simple end-to-end communication, which provides persistent and consistent distributed storage that can simplify the design and analysis of dependable distributed systems. Snapshot objects extend shared registers. They provide a way to further make the design and analysis of algorithms that base their implementation on shared registers easier. Snapshot objects allow an algorithm to construct consistent global states of the shared storage in a way that does not disrupt the system computation. Their efficient and fault-tolerant implementation is a fundamental problem, as there are many examples of algorithms that are built on top of snapshot objects.

Task description. Consider a fault-tolerant distributed system of n asynchronous nodes that are prone to failures. Their interaction is based on the emulation of Single-Writer/Multi-Reader (SWMR) shared registers over a message-passing communication system. Snapshot objects can read the entire array of system registers [1, 2]. The system lets each node update its own register via write() operations and retrieve the value of all shared registers via snapshot() operations. Note that these snapshot operations may occur concurrently with the write operations that individual nodes perform. We are particularly interested in the study of atomic snapshot objects that are *linearizable*: the operations write() and snapshot() appear as if they have been executed instantaneously, one after the other (*i.e.,* they appear to preserve real-time ordering).

Fault model. We consider an asynchronous message-passing system in which nodes may crash and packets may be lost, duplicated and reordered. In addition to these failures, we also aim to recover from *transient faults*, *i.e.,* any temporary violation of assumptions according to which the system was designed to behave, *e.g.,* the corruption of control variables, such as the program counter and operation indices, which are responsible for the correct operation of the studied system, or operational assumptions, such as that at least half of the system nodes never fail. Since the occurrence of these failures can be combined, we assume that these transient faults alter the system state in unpredictable ways. In particular, when modeling the system, we assume that these violations bring the system to an arbitrary state from which a *self-stabilizing algorithm* should recover the system. Therefore, starting from an arbitrary state, the correctness proof of self-stabilizing systems [3] has to show the return to a “correct behavior” within a bounded period. The complexity measure of self-stabilizing systems is the length of the recovery period.

Related work. We follow the design criteria of self-stabilization, which was proposed by Dijkstra [3] and detailed in [4]. Our overview of the related work focuses on self-stabilizing algorithms for shared-memory objects. Attiya *et al.* [5] implemented SWMR atomic shared-memory in an asynchronous networked system. Delporte-Gallet *et al.* [6] claim that when stacking the shared-memory atomic snapshot algorithm of [1] on the shared-memory emulation of [5] (with some improvements), the number of messages per snapshot operation is $8n$ and it takes 4 round trips. Their proposal, instead, takes $2n$ message per snapshot and just one round trip to complete. Our solution follows the non-stacking approach of Delporte-Gallet and it tolerates any failure (in any communication or operation invocation pattern) that [6] can as well as recover after the occurrence of transient faults that arbitrarily corrupt the system state. The literature on self-stabilization includes a practically-self-stabilizing variation for the work of Attiya *et al.* [5] by Alon *et al.* [7]. Their proposal guarantees wait-free recovery from transient faults. However, there is no bound on the recovery time. Dolev *et al.* [8] consider

MWMR atomic storage that is wait-free in the absence of transient faults. They guarantee a bounded time recovery from transient faults in the presence of a fair scheduler. They demonstrate the algorithm's ability to recover from transient faults using unbounded counters and in the presence of fair scheduling. Then they deal with the event of integer overflow via a consensus-based procedure. Since integer variables can have 64-bits, their algorithm seldom uses this non-wait-free procedure for dealing with integer overflows. In fact, they model integer overflow events as transient faults, which implies bounded recovery time from transient faults in the seldom presence of a fair scheduler (using bounded memory). They call these systems *self-stabilizing systems in the presence of seldom fairness.* Our work adopts these design criteria. We are unaware of self-stabilizing algorithms for snapshot objects that can recover from node failures. We note that "stacking" of self-stabilizing algorithms for asynchronous message-passing systems is not straightforward; the existing "stacking" needs schedule fairness [4, Section 2.7].

Contributions. We propose self-stabilizing algorithms for snapshot objects in networked systems. To the best of our knowledge, we are the first to consider both node failures and transient faults. Specifically, we propose:

(1) *A self-stabilizing variation on the non-blocking algorithm by Delporte-Gallet* et al. (Section 3). As by Delporte-Gallet *et al.*, each snapshot or write operation uses $\mathcal{O}(n)$ messages of $\mathcal{O}(\nu \cdot n)$ bits, where n is the number of nodes and ν is the number of bits for encoding the object. Our communication costs are slightly higher due to $\mathcal{O}(n^2)$ gossip messages of $\mathcal{O}(\nu)$ bits, where ν is the number of bits it takes to represent the object.

(2) *A self-stabilizing variation on the always-terminating algorithm by Delporte-Gallet* et al. (Section 4). Our algorithm can: (i) recover from of transient faults, and (ii) both write and snapshot operations always terminate (regardless of the invocation patterns of any operation). We achieve (*ii*) by choosing to use *safe registers* for storing the result of recent snapshot operations, rather than a *reliable broadcast* mechanism, which often has higher communication costs. Moreover, instead of dealing with one snapshot task at a time, we take care of several at a time. We also consider an input parameter, δ. For the case of $\delta = 0$, our self-stabilizing algorithm guarantees an always-termination behavior (as in the non-self-stabilizing algorithm by Delporte-Gallet *et al.*) that blocks all write operation upon the invocation of any snapshot operation at the cost of $\mathcal{O}(n^2)$ messages. For the case of $\delta > 0$, our solution aims at using $\mathcal{O}(n)$ messages per snapshot operation while monitoring the number of concurrent write operations. Once our algorithm notices that a snapshot operation runs concurrently with at least δ write operations, it blocks all write operations and uses $\mathcal{O}(n^2)$ messages for completing the snapshot operations. Thus, the proposed algorithm can trade communication costs with an $\mathcal{O}(\delta)$ bound on snapshot operation latency. Moreover, between any two consecutive periods in which snapshot operations block the system for write operations, the algorithm guarantees that at least δ write operations can occur.

The proposed algorithms use unbounded counters. In Section 5 we explain how to bound these counters. Due to the page limit, omitted details and proofs appear in [9], together with an explanation on how to extend our solutions to reconfigurable ones.

2 System Settings

We consider an asynchronous message-passing system. The system includes the set $\mathcal{P}$ of n failure-prone nodes whose identifiers are unique and totally ordered in $\mathcal{P}$. Any pair of nodes have access to a bidirectional bounded capacity communication channel that has no guarantees on the communication delays.

Each node runs a program, which we model as a sequence of *(atomic) steps*. Each step starts with an internal computation and finishes with a single communication operation, *i.e.,* message *send* or *receive*. The *state*, s_i, of $p_i \in \mathcal{P}$ includes all of p_i's variables and the set of all incoming communication channels. Note that p_i's step can change s_i and remove a message from $channel_{j,i}$ (upon message arrival) or add a message in $channel_{i,j}$ (when a message is sent). The term *system state* refers to a tuple, $c = (s_1, s_2, \cdots, s_n)$, where each s_i is p_i's state. An *execution* $R = c_0, a_0, c_1, a_1, \ldots$ is an alternating sequence of system states c_x and steps a_x, such that each c_{x+1}, except, c_0, is obtained from the preceding state c_x by the execution of step a_x. Let R' and R'' be a prefix, and resp., a suffix of R, such that R' is a finite sequence, which starts with a system state and ends with a step $a_x \in R'$, and R'' is an unbounded sequence, which starts in the system state that immediately follows step $a_x \in R$. The proof of the algorithms considers the number of *(asynchronous) cycles* of a fair execution, *i.e.,* every step that is applicable infinitely often is executed infinitely often and fair communication is kept. The first (asynchronous) cycle (with round-trips) of a fair execution $R = R'' \circ R'''$ is the shortest prefix R'' of R, such that each non-failing node executes in R'' at least one complete iteration of its do forever loop (and completes the round trips associated with the messages sent during that iteration), where $\circ$ denotes the concatenation operator. The second cycle in execution R is the first cycle in suffix R'' of execution R, and so on.

Fault model. We assume communication fairness, *i.e.,* if p_i sends a message infinitely often to p_j, node p_j receives that message infinitely often. We note that without this assumption, the communication channel between any two correct nodes eventually becomes non-functional. We consider standard terms for characterizing node failures [10]. A *crash failure* considers the case in which a node stops taking steps forever and there is no way to detect this failure. We say that a failing node resumes when it returns to take steps without restarting its program — the literature sometimes refer to this as an *undetectable restart*. The case of a detectable restart allows the node to restart all of its variables. We assume that each node has access to a quorum service, *e.g.,* [8, Section 13], that deals with packet loss, reordering, and duplication. A failure of node $p_i \in \mathcal{P}$ implies that it stops executing any step without any warning. The number of failing nodes is at most f and $2f < n$ for the sake of guaranteeing correctness [11]. In the absence of transient faults, failing nodes can simply crash, as in Delporte-Gallet *et al.* [6]. In the presence of transient faults, we assume that failing nodes resume within some unknown finite time and restart their program after initializing all of their variables (including the control variables). The latter assumption is needed *only* for recovering from transient faults; in [9] we explain how to remove this assumption. As already mentioned, we consider arbitrary violations of the assumptions according to which the system and the communication network were designed to operate. We refer to these violations as *transient faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). The occurrence of a transient fault is rare. Thus, we assume that transient faults occur before the system execution starts [4]. Moreover, it leaves the system to start in an arbitrary state.

Dijkstra's self-stabilization criterion. The set of *legal executions* (LE) refers to all the executions in which the requirements of the task T hold. We say that a system state c is *legitimate* when every execution R that starts from c is in LE. An algorithm is *self-stabilizing* with respect to the task of LE, when every (unbounded) execution R of the algorithm reaches within a bounded period a suffix $R_{legal} \in LE$ that is legal. That is, Dijkstra [3] requires that $\forall R : \exists R' : R = R' \circ R_{legal} \wedge R_{legal} \in LE \wedge |R'| \in \mathbb{N}$, where the length of R' is the complexity measure, which we refer to as the *recovery time*.

Self-stabilization in the presence of seldom fairness. As a variation of Dijkstra's self-stabilization criterion, Dolev *et al.* [8] proposed design criteria in which (i) any execution $R = R_{recoveryPeriod} \circ R' : R' \in LE$, which starts in an arbitrary system state and has a prefix ($R_{recoveryPeriod}$) that is fair, reaches a legitimate system state within a bounded prefix $R_{recoveryPeriod}$. (Note that the legal suffix R' is not required to be fair.) Moreover, (ii) any execution $R = R'' \circ R_{globalReset} \circ R''' \circ R_{globalReset} \circ \ldots : R'', R''', \ldots \in LE$ in which the prefix of R is legal, and not necessarily fair but includes at most $\mathcal{O}(n \cdot z_{\max})$ write or snapshot operations, has a suffix, $R_{globalReset} \circ R''' \circ R_{globalReset} \circ \ldots$, such that $R_{globalReset}$ is required to be fair and bounded in length, but it might permit the violation of liveness requirements, *i.e.,* a bounded number of operations might be aborted (as long as the safety requirement holds). Furthermore, R''' is legal and not necessarily fair, but includes at least $z_{\max}$ write or snapshot operations before the system reaches another $R_{globalReset}$. Since we can choose $z_{\max} \in \mathbb{Z}^+$ to be a very large value, say 2^{64}, and the occurrence of transient faults is rare, we refer to the proposed criteria as one for self-stabilizing systems that their execution fairness is unrequited except for seldom periods. We note that self-stabilizing algorithms (that follows Dijkstra's criterion) often assume fairness *throughout* R.

3 The Non-blocking Algorithm

The non-blocking solution to snapshot object emulation by [6, Algorithm 1] allows writes to terminate regardless of the invocation patterns of any other operation (as long as the invoking nodes do not fail during the operation). However, snapshot operation termination is guaranteed only after the last write operation. We discuss Delporte-Gallet *et al.* [6, Algorithm 1]'s solution before proposing our self-stabilizing variation.

Delporte-Gallet *et al.*'s non-blocking algorithm. Algorithm 1 presents [6, Algorithm 1] using our presentation style; the boxed code lines are irrelevant to [6, Algorithm 1]. The node state appears in lines 2 to 4 and automatic variables (which are allocated and deallocated automatically when program flow enters and leaves the variable's scope) are defined using the let keyword, *e.g.,* the variable *prev* (line 19). Also, when a message arrives, we use the parameter name xJ to refer to the arriving value for the message field x. Node p_i stores the array reg (line 4), such that the k-th entry stores the most recent information about node p_k's object and $reg[i]$ stores p_i's actual object. Every entry is a pair of the form (v, ts), where the field v is an object value and ts is an unbounded object index. The relation $\preceq$ can compare (v, ts) and (v', ts') according to the write operation indices (line 1). Node p_i also has an index for the snapshot operations, *i.e., ssn*.

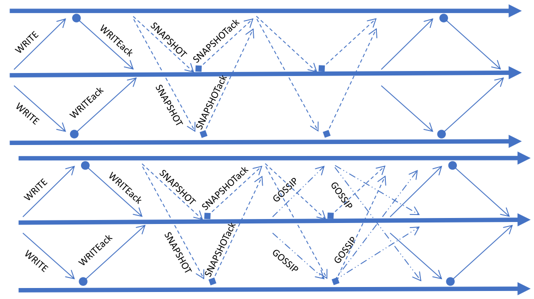

Fig. 1. Examples of Algorithm 1's executions. The upper drawing illustrates a case of a terminating snapshot operation (dashed line arrows) that occurs between two write operations (solid line arrows). The acknowledgments of these messages are arrows that start with circles and squares, respectively. The lower drawing depicts the execution of Algorithm 1's self-stabilizing version for the same case illustrated in the upper drawing. Note that the gossip messages do not interfere with other messages.

The write(v) *operation.* Algorithm 1's write(v) operation appears in lines 12 to 15 (client-side) and lines 17 to 23 (server-side). The client-side operation write(v) stores the pair (v, ts) in $reg[i]$ (line 13), where p_i is the calling node and ts is a unique operation index. Upon the arrival of a WRITE message to p_i from p_j (line 26), the server-side code is ran. Node p_i updates reg according to the timestamps of the arriving values (line 27). Then, p_i replies to p_j with the message WRITEack (line 31), which includes p_i's local perception of the system shared registers. Getting back to the client-side, p_i repeatedly broadcasts the message WRITE to all nodes until it receives replies from a majority of them (line 14). Once that happens, it uses the arriving values for keeping reg up-to-date (line 15).

The snapshot(v) *operation.* Algorithm 1's snapshot() operation appears in lines 17 to 23 (client-side) and lines 29 to 31 (server-side). Delporte-Gallet *et al.* [6, Algorithm 1] is non-blocking w.r.t. snapshot operations (in the absence of writes). Thus, the client-side is written as a repeat-until loop. Node p_i tries to query the system for the most recent value of the shared registrars. As said, the success of such attempts depends on the absence of writes. Thus, before each

Algorithm 1: Self-stabilizing algorithm for non-blocking snapshot object; code for p_i. The boxed code lines mark our additions to Delporte-Gallet *et al.* [1, Algorithm 1].

1 **Definitions of $\preceq$:** For integers t and t': $(\bullet, t) \preceq (\bullet, t') \iff t \leq t'$; For arrays tab and tab' of $(\bullet, integer)$: $tab \preceq tab' \iff \forall p_k \in \mathcal{P} : tab[k] \preceq tab'[k]$; Also, $a \prec b \equiv a \preceq b \wedge a \neq b$;
2 **local variables initialization (optional in the context of self-stabilization):**
3 $ssn := 0; ts := 0;$ `/* indices of the snapshout, resp., write operations */`
4 $reg := [\perp, \ldots, \perp];$ `/* shared registers (⊥ is smaller than any other written value) */`
5 **macro** merge(Rec) **begin**
6     $\boxed{ts \leftarrow \max(\{ts, reg[i].ts\} \cup \{r[i].ts \mid r \in Rec\});}$
7   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\});$
8 **do forever begin**
9     $\boxed{\textbf{foreach } ssn' \neq ssn \textbf{ do delete } \mathrm{SNAPSHOTack}(-, ssn');}$
10     $\boxed{ts \leftarrow \max\{ts, reg[i].ts\};}$
11     $\boxed{\textbf{for } p_k \in \mathcal{P} : k \neq i \textbf{ do } \text{send } \mathrm{GOSSIP}(reg[k]) \text{ to } p_k;}$
12 **operation** write(v) **begin**
13     $ts \leftarrow ts + 1; reg[i] \leftarrow (v, ts);$ **let** $lReg := reg;$
14   **repeat** broadcast WRITE($lReg$); **until** WRITEack($regJ \succeq lReg$) *received from a majority*;
15   merge(Rec) **where** Rec is the set of reg arrays received at line 14;
16   **return**();
17 **operation** snapshot() **begin**
18   **repeat**
19   **let** $prev := reg; ssn \leftarrow ssn + 1;$
20   **repeat** broadcast SNAPSHOT(reg, ssn); **until** SNAPSHOTack($\bullet, ssnJ = ssn$) *received from a majority*;
21   merge(Rec) **where** Rec is the set of reg arrays received at line 20;
22   **until** $prev = reg$;
23   **return**(reg);
24 **upon** message GOSSIP($regJ$) **arrival from** p_j **begin**
25     $\boxed{reg[i] \leftarrow \max\{reg[i], regJ\}; ts \leftarrow \max\{ts, reg[i].ts\};}$
26 **upon** message WRITE($regJ$) **arrival from** p_j **begin**
27   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\preceq}(reg[k], regJ[k]);$
28   **send** WRITEack(reg) **to** p_j;
29 **upon** message SNAPSHOT($regJ, ssn$) **arrival from** p_j **begin**
30   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\preceq}\{reg[k], regJ[k]\};$
31   **send** SNAPSHOTack(reg, ssn) to p_j;

such broadcast, p_i copies reg's value to $prev$ (line 19) and exits the repeat-until loop once the updated value of reg indicates the obscene of concurrent writes.

The proposed unbounded self-stabilizing variation. We propose Algorithm 1 as an extension of Delporte-Gallet *et al.* [6, Algorithm 1]. The boxed code lines mark our additions. We denote variable X's value at node p_i by X_i. Algorithm 1 considers the case in which any of p_i's operation indices, ssn_i and ts_i, is smaller than some other ssn or ts value, say, ssn_m, $reg_i[i].ts$, $reg_j[i].ts$ or $reg_m[i].ts$, where X_m appears in the X field of some on transit message. For the case of corrupted ssn values, p_i's client-side ignores arriving messages with ssn values that do not match ssn_i (line 20). The do-forever loop removes any stored snapshot reply whose ssn field is not ssn_i. For the case of corrupted

ts values, p_i's do-forever loop makes sure that ts_i is not smaller than $reg_i[i].ts$ (line 10) before gossiping to every node $p_j \in \mathcal{P}$ its local copy of the shared register (line 11). Also, upon the arrival of such gossip messages, Algorithm 1 merges the arriving information with the local one (line 25). Moreover, when replies from write or snapshot messages arrive to p_i, it merges the arriving ts value with the one in ts_i (line 6). Figure 1's upper and lower drawings depict executions of the non-self-stabilizing algorithm [6], and respectively, our self-stabilizing version (Algorithm 1). The drawings illustrate a write operation that is followed by a snapshot operation and then a second write. We use this example for comparing algorithms 1, 2 and 3 (the latter two are presented in Section 4). The complete discussion for Algorithm 1 and proof details appear in [9].

Theorem 1 (Recovery). *Within $\mathcal{O}(1)$ cycles, a fair execution of Algorithm 1 reaches a state c in which (i) ts_i's value is not smaller than any p_i's timestamp value. Also, if node p_i takes a step immediately after c that includes line 13, then in c it holds that $ts_i = reg_i[i].ts = reg_j[i].ts$ and for every messages m that is in transit from p_i to p_j or p_j to p_i it holds that $m.reg[i].ts = ts_i$. Moreover, (ii) ssn_i is not smaller than any p_i's snapshot sequence number.*

Proof Sketch. Arguments (1) to (3) show invariant (i). (1) *The values installed in ts_i, $reg_i[i].ts$, $reg_j[i].ts$, $reg_i[i]$ and $reg_j[i]$ are non-decreasing,* since their values are never decremented. (2) *Within $\mathcal{O}(1)$ cycles, $ts_i \geq reg_i[i].ts$,* since p_i executes line 10 at least once in every cycle. (3) Within $\mathcal{O}(1)$ cycles, $reg_i[i].ts \geq reg_m[i].ts$ and $reg_i[i].ts \geq regJ[i].ts$ whenever p_j raises SNAPSHOTack$(regJ, ssn)$ or WRITE$(regJ)$, where m' is a message on transit from p_j to p_k and denote $reg_{m'}$ as values of the reg filed in m', and $p_i, p_j, p_k \in \mathcal{P}$ are non-failing nodes (and $i = k$ possibly holds). Moreover, $reg_j[i].ts \geq reg_{m'}[i].ts$ and $reg_i[i].ts \geq regJ[i].ts$ whenever p_k raises GOSSIP$(regJ)$, WRITEack$(regJ)$ or SNAPSHOTack$(regJ, \bullet)$. The proof follows by the nodes' message exchange. Invariant (ii) follows by arguments similar to (1) to (3). ■

4 The Always-terminating Algorithm

Delporte-Gallet *et al.* [6, Algorithm 2] guarantee termination for any invocation pattern of write and snapshot operations, as long as the invoking nodes do not fail during these operations. Its advantage over Delporte-Gallet *et al.* [6, Algorithm 1] is that it can deal with an infinite number of concurrent write operations. Before proposing our self-stabilizing always-terminating solution, we bring [6, Algorithm 2] in Algorithm 2 using the presentation style of this paper.

Delporte-Gallet *et al.*'s always-terminating algorithm. Delporte-Gallet *et al.* [6, Algorithm 2] use a job-stealing scheme for allowing rapid termination of snapshot operations. Node $p_i \in \mathcal{P}$ starts its snapshot operation by queueing this new task at all nodes $p_j \in \mathcal{P}$. Once p_j receives p_i's new task and when that task reaches the queue front, p_j starts the baseSnapshot(s, t) procedure, which is similar to Algorithm 1's snapshot() operation. This joint participation in all snapshot operations makes sure that all nodes are aware of all on-going snapshot operations. Moreover, it allows the nodes to make sure that no write() can stand in the way of on-going snapshot operations. To that end, the nodes wait until the oldest snapshot operation terminates before proceeding with later operations. Specifically, they defer write operations that run concurrently with snapshot operations. This guarantees termination of snapshot operations via the interleaving and synchronization of snapshot and write operations.

Algorithm 2 extends Algorithm 1 (non-self-stabilizing version, which does not include the boxed code lines) in the sense that it uses all of Algorithm 1's variables and an additional one, array *repSnap*, which snapshot() operations use. The entry $repSnap[x, y]$ holds the outcome of p_x's y-th snapshot operation, where no explicit bound on the number of invocations of snapshot operations is given. Note that bounded space is a prerequisite for self-stabilization.

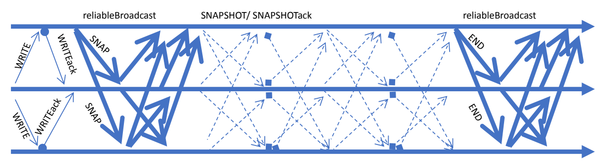

Fig. 2. Algorithm 2's run for the case of Figure 1's upper drawing.

The write(v) *operation and the* baseWrite() *function.* Since write(v) operations are preemptible, p_i cannot always start immediately to write. Instead, p_i stores v in $writePend_i$ together with a unique operation index (line 44). It then runs the operation as a background task (line 38) using baseWrite() (lines 48 to 51).

The snapshot() *operation.* A call to snapshot() (line 46) causes p_i to reliably broadcast, via the primitive reliableBroadcast, a new *ssn* index in a SNAP to all nodes in $\mathcal{P}$. Node p_i then places it as a background task (line 47).

The baseSnapshot() *function.* As in Algorithm 1's snapshot, the repeat-until loop iterates until the retrieved *reg* vector equals to the one that was known prior to the last repeat-until iteration. Then, p_i stores in $repSnap[s, t]$, via a reliable broadcast of the END message, the snapshot result (line 59 and 66).

Synchronization between the baseWrite() *and* baseSnapshot() *functions.* Algorithm 2 interleaves the background tasks in a do forever loop (lines 38 to 42). As long as there is an awaiting write task, node p_i runs the baseWrite() function (line 38). Also, if there is an awaiting snapshot task, node p_i selects the oldest task, $(source, sn)$, and uses the baseSnapshot($source, sn$) function. Here, Algorithm 2 blocks until $repSnap[source, sn]$ contains the result of that snapshot task.

Figure 2 depicts an example of Algorithm 2's execution where a write operation is followed by a snapshot operation. Each snapshot is handled separately and the communications of each such operation requires $\mathcal{O}(n^2)$ messages.

An unbounded self-stabilizing always-terminating algorithm. We propose Algorithm 3 as a variation of Delporte-Gallet *et al.* [6, Algorithm 2]. Algorithms 2 and 3 differ mainly in their ability to recover from transient faults. This implies some constraints. *E.g.,* Algorithm 3 must have a clear bound on the number of pending snapshot tasks. For the sake of simple presentation, Algorithm 3

Algorithm 2: The non-self-stabilizing and always-terminating algorithm by Delporte-Gallet *et al.* [6] that emulates snapshot object; code for p_i

32 **local variables initialization:** $ssn := 0; ts := 0;$ /* snapshout, resp., write indices */
33 $writePending \leftarrow \bot;$ /* stores p_i's write task */
34 $reg := [\bot, \ldots, \bot];$ /* shared registers ($\bot$ is smaller than any other written value) */
35 **foreach** $k, s : repSnap[k, s] := \bot;$ /* stores p_k's snapshot task result for index s */
36 **macro** merge(Rec) **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\});$
37 **do forever begin**
38   **if** $(writePending \neq \bot)$ **then** baseWrite($writePending$); $writePending \leftarrow \bot;$
39   **if** (there are messages SNAP() received and not yet processed) **then**
40   **let** SNAP($source, sn$) **be** the oldest of these messages;
41   baseSnapshot($source, sn$);
42   **wait until** $(repSnap[source, sn] \neq \bot);$

43 **operation** write(v) **begin**
44   $writePending \leftarrow v$; **wait until** $(writePending = \bot)$; **return**();

45 **operation** snapshot() **begin**
46   $sns \leftarrow sns + 1$; reliableBroadcast SNAP(i, sns);
47   **wait until** $(repSnap[i, sns] \neq \bot)$; **return**($repSnap[i, sns]$);

48 **function** baseWrite(v) **begin**
49   $ts \leftarrow ts + 1; reg[i] \leftarrow (ts, v)$; **let** $lReg := reg$;
50   **repeat** broadcast WRITE($lReg$); **until** WRITEack($regJ \succeq lReg$) *received from a majority*;
51   merge(Rec) **where** Rec is the set of reg arrays received at line 50;

52 **function** baseSnapshot(s, t) **begin**
53   **while** $repSnap[s, t] = \bot$ **do**
54   **let** $prev := reg$; $ssn \leftarrow ssn + 1$;
55   **repeat**
56   broadcast SNAPSHOT(s, t, reg, ssn);
57   **until** $(sJ = s, tJ = t, \bullet, ssnJ = ssn)$ received from a majority);
58   merge(Rec) **where** Rec is the set of reg arrays received at line 56;
59   **if** $prev = reg$ **then** reliableBroadcast END($source, sn, prev$);

60 **upon** message WRITE($regJ$) **arrival from** p_j **begin**
61   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\prec_{sn}}(reg[k], regJ[k]);$
62   **send** WRITEack(reg) **to** p_j;

63 **upon** message SNAPSHOT($s, t, regJ, ssnJ$) **arrival from** p_j **begin**
64   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\prec_{sn}}(reg[k], regJ[k]);$
65   **send** SNAPSHOTack($s, t, reg, ssnJ$) **to** p_j;

66 **upon** message END(s, t, val) **arrival from** p_j **do** $repSnap[s, t] \leftarrow val$;

assumes that the system needs, for each node, to cater for at most one pending snapshot task. We avoid the use of a reliable broadcast, which Delporte-Gallet *et al.* use, and instead, we use a simpler mechanism for safe registers.

Algorithm 3 can defer snapshot tasks until either (i) at least one node was able to observe at least δ concurrent write operations, where δ is an input parameter, or (ii) there are no concurrent write operations. The tunable parameter δ balances between the latency (with respect to snapshot operations) and communication costs. *I.e.,* for the case of δ being a very high (finite) value, Algorithm 3 guarantees termination in a way that resembles [6, Algorithm 1], which uses $\mathcal{O}(n)$ messages per snapshot operation, and for the case of $\delta = 0$, Algorithm 3 behaves in a way that resembles [6, Algorithm 2], which uses $\mathcal{O}(n^2)$ messages per snapshot.

Algorithm details. Algorithm 3 lets every node disseminate its (at most one) pending snapshot task and use a safe register for facilitating the delivery of the task result to its initiator. *I.e.,* once a node finishes a snapshot task, it broadcasts the result to all nodes and waits for replies from a majority of nodes, which may possibly include the initiator of the snapshot task (see safeReg(), line 71). This way, if node p_j notices that it has the result of an ongoing snapshot task, it sends that result to the node who initiated the task.

The do forever loop. Algorithm 3's do forever loop (lines 74 to 80), includes a number of lines for cleaning stale information, *e.g.,* out-of-synch SNAPSHOTack messages (line 74), out-dated operation indices (line 75), illogical vector-clocks (line 76) or corrupted pndTsk entries (line 77). The gossiping of operation indices (lines 78 and 98) also helps to remove stale information (as in Algorithm 1 but only with the addition of sns values). The synchronization between write and snapshot operations (lines 79 and 80) starts with a write, if there is any such pending task (line 79), before running its own snapshot task, if there is any such pending, as well as any snapshot task (initiated by others) for which p_i observed that at least δ write operations occur concurrently with it (line 80).

The baseSnapshot() *function and the* SNAPSHOT *message.* Algorithm 3 maintains the state of every snapshot task in the array pndTsk. The entry $\mathrm{pndTsk}_i[k] = (sns, vc, \mathrm{fnl})$ includes: (i) the index sns of the most recent snapshot operation that $p_k \in \mathcal{P}$ has initiated and p_i is aware of, (ii) the vector clock representation of reg_k (*i.e.,* just the timestamps of reg_k, cf. line 69) and (iii) the final result fnl of the snapshot operation (or $\bot$, in case it is still running).

The baseSnapshot() function includes an outer loop part (lines 87 and 94), an inner loop part (lines 87 to 90), and a result update part (lines 91 to 93). The outer loop increments the snapshot index, ssn (line 87), so that it can consider a new query attempt by the inner loop. The outer loop ends when there are no more pending snapshot tasks that this call to baseSnapshot() needs to handle. The inner loop broadcasts SNAPSHOT messages, which includes all the pending snapshot tasks, $(S \cap \Delta)$, that are relevant to this call to baseSnapshot() together with the local current value of reg and the snapshot query index ssn. The inner loop ends when acknowledgments are received from a majority of processors and the received values are merged (line 90). The results are updated by writing to an emulated safe shared register (line 91) whenever $prev = reg$. In case the results do not allow p_i to terminate its snapshot task (line 93), Algorithm 3 uses the query results for storing the timestamps in the field vs. This allows to balance a trade-off between snapshot operation latency and communication costs, as we explain next.

The use of the input parameter δ for balancing the trade-off between snapshot operation latency and communication costs. For the case of $\delta = 0$, since no snapshot task is to be deferred, the set Δ (line 70) includes all the nodes for which there is no stored result, *i.e.,* $\mathrm{pndTsk}[k].\mathrm{fnl} = \bot$. The case of $\delta > 0$ uses the fact that Algorithm 3 samples the vector clock value of reg_k and stores it in $\mathrm{pndTsk}[k].vc$ (line 93) once it had completed at least one iteration of the repeat-until loop (line 89 and 90). *I.e.,* the sampling of the vector clock is an event that occurs not before the start of p_k's snapshot (that has the index pndTsk$[k].sns$).

Many-jobs-stealing scheme for reduced blocking periods. Whenever pndTsk$[k]$.fnl $\neq \perp$ and $sns > 0$, we consider p_k's task as active. To the end of helping all actives tasks, p_i samples the set of currently pending task ($S_i \cap \Delta_i$) (line 87) before starting the inner repeat-until loop (lines 89 to 90) and broadcasting the client-side message SNAPSHOT, which includes the most recent snapshot task information. The server-side reception of this message (lines 103 to 104), updates the local information (line 105) and sends the reply to the client-side (lines 106 to 107). Note that if the receiver notices that it has the result of an ongoing snapshot task, then it sends that result to the requesting processor (line 107).

The safeReg() function and the SAVE message. The safeReg() function considers a snapshot task that was initiated by node $p_k \in \mathcal{P}$. This function is responsible for storing the results of snapshot tasks in a safe register. It does so by broadcasting the client-side message SAVE to all nodes in the system (line 71). Upon the arrival of the SAVE message to the server-side, the receiver stores the arriving information, as long as the arriving information is more recent than the local one. Then, the server-side replies with a SAVEack message to the client-side, who is waiting for a majority of such replies (line 71).

Figure 3 depicts two examples of Algorithm 3's execution. In the upper drawing, a write operation is followed by a snapshot operation. Note that fewer messages are considered when comparing to Figure 2's example. The lower drawing illustrates the case of concurrent invocations of snapshot operations by all nodes. Observe the potential improvement with respect to number of messages (in the upper drawing) and throughput (in the lower drawing) since Algorithm 2 uses $\mathcal{O}(n^2)$ messages for each snapshot task and handles only one snapshot task at a time.

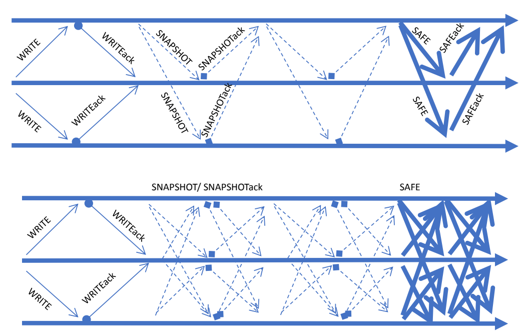

Fig. 3. The upper drawing depicts an example of Algorithm 3's execution for a case that is equivalent to the one depicted in the upper drawing of Figure 2, i.e., only one snapshot operation. The lower drawing illustrates the case of concurrent invocations of snapshot operations by all nodes.

Correctness. The complete discussion and proof details appear in [9].

Definition 1 (Consistent system states and executions). *(i) Let c be a system state in which ts_i is greater than or equal to any p_i's timestamp values in the variables and fields related to ts. We say that the ts' timestamps are consistent in c. (ii) Let c be a system state in which ssn_i is greater than or equal to any p_i's snapshot sequence numbers in the variables and fields related to ssn. We*

Algorithm 3: Self-stabilizing always-terminating snapshot; code for p_i

67 **input:** δ a number of observed concurrent writes after which writes block temporarily;
68 **variables:** $ts := 0$ is p_i's write operation index; $ssn, sns := 0$ are p_i's snapshot operation indices; $reg[n] := [\bot, \ldots, \bot]$ buffers all shared registers; $writePending \leftarrow \bot$ stores p_i's write task; $\mathrm{pndTsk}[n] := [(0, \bot, \bot), \ldots, (0, \bot, \bot)]$ control variables of snapshot operations; each entry form is (sns, vc, fnl), where sns is an index, vc is a vector clock that time stamps the snapshot operation sns, and fnl is the operation's returned value; (In the context of self-stabilization, variable initialization is optional.)
69 **macro** $\mathrm{VC} := [ts_k]_{p_k \in \mathcal{P}}$ **where** $ts_k := 0$ **when** $reg[k] = \bot$ **otherwise** $reg[k] = (\bullet, ts_k)$;
70 **macro** $\Delta := \{(k, \mathrm{pndTsk}[k].sns, \mathrm{pndTsk}[k].vc) | p_k \in \mathcal{P} \land \mathrm{pndTsk}[k].fnl = \bot \land ((\delta = 0 \land \mathrm{pndTsk}[k].sns > 0) \lor (\mathrm{pndTsk}[k].vc \neq \bot \land \delta \leq \sum_{\ell \in \{1,\ldots,n\}} \mathrm{VC}[\ell] - \mathrm{pndTsk}[k].vc[\ell]))\} \cup \{(i, \mathrm{pndTsk}[i].sns, \mathrm{pndTsk}[i].vc) : \mathrm{pndTsk}[i].sns > 0 \land \mathrm{pndTsk}[i].fnl = \bot\}$;
71 **macro** safeReg(A) **repeat** broadcast SAVE(A) **until** majority of SAVEack($AJ = \{(k, s) : (k, s, \bullet) \in A\}$) arrived;
72 **macro** merge(Rec) $\{ts \leftarrow \max(\{ts, reg[i].ts\} \cup \{r[i].ts \mid r \in Rec\})$; **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\})\}$;
73 **do forever begin**
74   **foreach** $ssn' \neq ssn$ **do delete** SNAPSHOTack($-, ssn'$);
75     $(ts, sns) \leftarrow (\max\{ts, reg[i].ts\}, \max\{sns, \mathrm{pndTsk}[i].sns\})$;
76   **for** $k \in \{1, \ldots, n\} : \mathrm{pndTsk}[k].vc \not\preceq \mathrm{VC}$, ***where*** *line 1 defines the relation* $\preceq$ **do** $\mathrm{pndTsk}[k].vc \leftarrow \bot$;
77   **if** $sns \neq \mathrm{pndTsk}[i].sns$ **then** $\mathrm{pndTsk}[i] \leftarrow (sns, \bot, \bot)$;
78   **for** $p_k \in \mathcal{P} : k \neq i$ **do** send GOSSIP($reg[k], \mathrm{pndTsk}[k].sns$) to p_k;
79   **if** $writePending \neq \bot$ **then** {baseWrite($writePending$); $writePending \leftarrow \bot$; };
80   **if** $\Delta \neq \emptyset$ **then** baseSnapshot(Δ);

81 **operation** write(v) $\{writePending \leftarrow v$; **wait until** $(writePending = \bot)$; **return**();}
82 **operation** snapshot() **begin**
83     $(sns, \mathrm{pndTsk}[i]) \leftarrow (sns + 1, (sns, \bot, \bot))$; **wait until** $(\mathrm{pndTsk}[i].fnl \neq \bot)$; **return**($\mathrm{pndTsk}[i].fnl$);

84 **function** baseWrite(v) $\{ts \leftarrow ts + 1; reg[i] \leftarrow (ts, v)$; **let** $lReg := reg$; **repeat** broadcast WRITE($lReg$); merge(Rec) **where** Rec is the received reg arrays} **until** WRITEack($regJ \succeq lReg$) *received from a majority*;
85 **function** baseSnapshot(S) **begin**
86   **repeat**
87         $ssn \leftarrow ssn + 1$; **let** $prev := reg$; **repeat**
88   broadcast SNAPSHOT($(S \cap \Delta), reg, ssn$);
89   **until** $(S \cap \Delta) = \emptyset$ or majority of (SNAPSHOTack($\bullet, ssnJ = ssn$) arrived);
90   merge(Rec) **where** Rec is the set of reg arrays received at line 89;
91   **if** $prev = reg \land (S \cap \Delta) \neq \emptyset$ **then**
92   safeReg($\{(k, \mathrm{pndTsk}[k].sns, prev) : (k, s, \bullet) \in (S \cap \Delta)\}$)
93   **else if** $((i, \bullet) \in (S \cap \Delta)) \land (\mathrm{pndTsk}[i].vc = \bot)$ **then** $\mathrm{pndTsk}[i].vc \leftarrow \mathrm{VC}$;
94   **until** $(S \cap \Delta) = \emptyset \lor ((S \cap \Delta) = (i, \bullet) \land \mathrm{pndTsk}[i].sns > 0 \land \mathrm{pndTsk}[i].fnl = \bot \land \delta \leq \sum_{\ell \in \{1,\ldots,n\}} (\mathrm{VC}[\ell] - \mathrm{pndTsk}[i].vc[\ell]))$;

95 **upon** message SAVE(AJ) **arrival from** p_j **begin**
96   **foreach** $(k, s, r) \in AJ : \mathrm{pndTsk}[k].sns < s \lor \mathrm{pndTsk}[k] = (s, \bullet, \bot)$ **do** $(\mathrm{pndTsk}[k].sns, \mathrm{pndTsk}[k].fnl) \leftarrow (s, r)$;
97   **send** SAVEack($\{(k, s) : (k, s, \bullet) \in AJ\}$) **to** p_j;

98 **upon** message GOSSIP($regJ, snsJ$) **arrival from** p_j **begin**
99     $reg[i] \leftarrow \max\{reg[i], regJ\}$; $(ts, sns) \leftarrow (\max\{ts, reg[i].ts\}, \max\{sns, snsJ\})$;

100 **upon** message WRITE($regJ$) **arrival from** p_j **begin**
101   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\prec_{sn}}(reg[k], regJ[k])$;
102   **send** WRITEack(reg) **to** p_j;

103 **upon** message SNAPSHOT($SJ, regJ, ssnJ$) **arrival from** p_j **begin**
104   **for** $p_k \in \mathcal{P}$ **do** $reg[k] \leftarrow \max_{\prec_{sn}}(reg[k], regJ[k])$;
105   **foreach** $(s, sn, vc) \in SJ : \mathrm{pndTsk}[s].sns < sn \lor \mathrm{pndTsk}[s] = (sn, \bot, \bot)$ **do** $\mathrm{pndTsk}[s] \leftarrow (sn, vc, \bot)$;
106   **let** $A := \{(k, \mathrm{pndTsk}[k].sns, \mathrm{pndTsk}[k].fnl) : (k, \bullet) \in SJ \land \mathrm{pndTsk}[k].fnl \neq \bot\}$;
107   **send** SNAPSHOTack($reg, ssnJ$) **to** p_j; **if** $A \neq \emptyset$ **then send** SAVE(A) **to** p_j ;

say that the ssn's snapshot sequence numbers are consistent in c. (iii) Let c be a system state in which sns_i *is not smaller than any* p_i*'s snapshot index sns. Moreover,* $\forall p_i \in \mathcal{P} : sns_i = \mathrm{pndTsk}_i[i].sns$ *and* $\forall p_i, p_j \in \mathcal{P} : \mathrm{pndTsk}_j[i].sns \leq \mathrm{pndTsk}_i[i].sns$*. We say that the sns's snapshot indices are consistent in c. (iv) Let c be a system state in which* $\forall p_i, p_k \in \mathcal{P} : \mathrm{pndTsk}_i[k].vc \preceq \mathrm{VC}_i$ *holds, where* VC_i *is the returned value from* VC() *(line 69). We say that the vector clock values are consistent in c. We say that system state c is consistent if it is consistent with respect to invariants (i) to (iv). Let R be an execution of Algorithm 3 that all of its system states are consistent and* R' *be a suffix of R. We say that execution* R' *is consistent (with respect to R) if any message arriving in* R' *was indeed sent in R and any reply arriving in* R' *has a matching request in R.*

Theorem 2 (Recovery). *Let R be Algorithm 3's fair execution. Within* $\mathcal{O}(1)$ *cycles in R, the system reaches a consistent state* $c \in R$ *(Definition 1). Within* $\mathcal{O}(1)$ *cycles after c, the system starts a consistent execution* R'*.*

Proof Sketch. Note that Theorem 1 implies invariants (i) and (ii) of Definition 1 also for the case of Algorithm 3, because they use the similar lines of code for asserting these invariants. For invariant (iii), *sns* and pndTsk in Algorithm 3 follow the same propagation patterns as *ts* and *reg* in Algorithm 1. Moreover, within a cycle, every $p_i \in \mathcal{P}$ executes line 77. Thus, invariant (iii)'s proof follows similar arguments to the ones in Theorem 1's proof. Invariant (iv)'s proof is implied by the fact that within a cycle, $p_i \in \mathcal{P}$ executes line 76. By the definition of cycles (Section 2), within a cycle, R reaches a suffix R', such that every received message during R' was sent during R. By repeating the previous argument, it holds that within $\mathcal{O}(1)$ cycles, R reaches a suffix R' in which for every received reply has an associated request that was sent during R. ■

Theorem 3 (Algorithm 3's termination and linearization). *Let R be Algorithm 3's consistent execution (Definition 1). Suppose that there exists* $p_i \in \mathcal{P}$*, such that in R's second system state, it holds that* $\mathrm{pndTsk}_i[i] = (s, \bullet, \bot)$ *and* $s > 0$*. Within* $\mathcal{O}(\delta)$ *cycles, the system reaches* $c \in R : \mathrm{pndTsk}_i[i] = (s, \bullet, x) : x \neq \bot$*.*

Proof Sketch. Lemma 1 sketches the key arguments of the termination proof.

Lemma 1 (Algorithm 3's termination). *Within* $\mathcal{O}(\delta)$ *cycles, the system reaches a state* $c \in R$ *in which either: (i) for any non-failing node* $p_j \in \mathcal{P}$ *it holds that* $i \in \Delta_j$ *(line 70) and* $\mathrm{pndTsk}_j[i] = (s, \bullet, \bot)$*, (ii)* $\forall M \subseteq \mathcal{P} : |M| > |\mathcal{P}|/2 : \exists_{p_j \in M} : \mathrm{pndTsk}_j[i] = (s, \bullet, x) : x \neq \bot$ *or (iii)* $\mathrm{pndTsk}_i[i] = (s, \bullet, x) : x \neq \bot$*.*

Proof Sketch. We show that R has a prefix R' that includes $\mathcal{O}(\delta)$ cycles, such that none of the lemma invariants hold during R'.

Claim (a). There is no step $a_i \in R'$ in which p_i evaluate the if-statement condition in line 91 to be true (or one of the lemma invariants holds).

Proof of claim. Towards a contradiction, suppose that $a_i \in R$ calls $\mathsf{safeReg}_i()$. Arguments (1) and (2) show that this happens for the case of $k = i$, and that invariant (ii) holds. *Argument (1): a_i includes the execution of line 91.* This is because, once in $\mathcal{O}(1)$ cycles, p_i calls baseSnapshot$_i(S_i)$ (line 80), which does not change the value of S_i. *Argument (2): invariant (ii) holds.* The function $\mathsf{safeReg}_i(\{(\bullet, r) : r \neq \bot\})$ (line 71) repeatedly broadcasts SAVE$(\{(\bullet, r) : r \neq \bot\})$ until p_i receives SAVEack$(\{(\bullet, r) : r \neq \bot\})$ from a majority. Theorem 2 and R's consistency imply that every received SAVEack is associated with a SAVE that was sent in R. Invariant (ii) holds due to the majority intersection property. □

Claim (b). Within $\mathcal{O}(1)$ asynchronous cycles, the system reaches a state $c' \in R'$ in which for any non-faulty node $p_j \in \mathcal{P}$ it holds that $\text{pndTsk}_j[i] = (s, y, \bullet) : y \neq \bot$.

Proof of claim. For the case of $j = i$, we note that claim (a) implies that $(i, \bullet) \in S_i$ holds and the execution of line 93 in every call for baseSnapshot(S_i). For the $j \neq i$ case, we note that within $\mathcal{O}(1)$ cycles, p_i executes lines 87 and 88 in which p_i broadcasts SNAPSHOT$(\{(\bullet, \text{pndTsk}_i[i].vc), \bullet\})$, such that $\text{pndTsk}_i[i].vc \neq \bot$ holds by the case of $j = i$. Once p_j receives this message, $\text{pndTsk}_j[i].vc \neq \bot$ holds (line 105). The above arguments for the case of $j \neq i$ can be repeated as long as invariant (iii) does not hold. Thus, the arrival of such a SNAPSHOT message to all $p_j \in \mathcal{P}$ occurs within $\mathcal{O}(1)$ asynchronous cycles. □

Claim (c). Let $c' \in R'$ be a system state in which for any non-faulty node $p_j \in \mathcal{P}$ it holds that $\text{pndTsk}_j[i] = (s, y, \bullet) : y \neq \bot$. Let x be the number of iterations of the outer loop in baseSnapshot() (lines 87 and 94) that node p_i takes between c' and $c'' \in R'$, where c'' is a system state after which it takes at most $\mathcal{O}(\delta)$ asynchronous cycles until the system reach the state c''' in which at least one of the lemma invariants holds. The value of x is actually finite and $x \leq \delta$.

Proof of claim. Argument (1): during the outer loop in baseSnapshot() *(lines 87 and 94), p_i tests the if-statement condition at line 91 and that condition does* not *hold,* due to Claim (a). *Argument (2): suppose that there are at least x consecutive and complete iterations of p_i's outer loop in* baseSnapshot() *(lines 87 and 94) between c' and c'' in which the if-statement condition at line 91 does not hold. Then, there are at least x write operations that run concurrently with the snapshot operation that has the index of s, since* the only way that the if-statement condition in line 91 does not hold in a repeated manner is by repeated changes of ts fields in reg_i during the different executions of lines 87 to 90 (due to line 81 of $\mathsf{write}()$). We define the function $\mathcal{S}_i()$ so that whenever p_i's program counter is outside of the function baseSnapshot(), $\mathcal{S}_i()$ returns Δ_i. Otherwise, it returns $(S_i \cap \Delta_i)$. *Argument (3): there exists $x' \leq \delta$ for which $(i, \bullet) \in \mathcal{S}_i()$, where x' is the number of consecutive and complete iterations of p_i's outer loop in* baseSnapshot() *between c' and c'' in which the if-statement condition at line 91 does not hold.* This is because Argument (2) implies that the number of iterations continues to grow. During every such iteration there are increments of the summation $\sum_{\ell \in \{1,\ldots,n\}} \text{VC}_i[\ell] - \text{pndTsk}_i[i].vc[\ell]$ until it is at least δ, and thus, $(i, \bullet) \in \mathcal{S}_i()$ holds (line 70, for the case of $k = i$). *Argument (4): suppose that p_i has taken at least x' iterations of the outer loop in* baseSnapshot() *(lines 87 and 94) after system state c'. After this, suppose that the system has reached a state c'' in*

which $i \in \Delta_i$, where c'' is defined in Argument (3). Within $\mathcal{O}(1)$ cycles after c'', the system reaches c''' in which $i \in \Delta_j$ holds for any non-failing $p_j \in \mathcal{P}$. Within $\mathcal{O}(1)$ asynchronous cycles after c'', it holds that reg_j's ts fields are not smaller than the ones of reg_i's ts fields in c'' (because in every iteration of the outer loop in baseSnapshot(), p_i broadcasts reg_i and these boradcasts arrive within one cycle to p_j, who updates reg_j). The rest of the proof shows that $i \in \Delta_j$ holds (line 70, case of $k = i$), as in Argument (3). □
This completes the proof of the lemma. ■

The rest of the theorem's proof considers the case in which (i) in any system state of R, it holds that $\text{pndTsk}_i[i] = (s, \bullet, \bot)$, $s > 0$ and any majority $M \subseteq \mathcal{P} : |M| > |\mathcal{P}|/2$ include at least one $p_j \in M$, such that $\text{pndTsk}_j[i] = (s, \bullet, x) : x \neq \bot$, or (ii) in any system state of R, it holds that $\text{pndTsk}_i[i] = (s, \bullet, \bot)$, $s > 0$ and for any non-failing node $p_j \in \mathcal{P}$ it holds that $i \in \Delta_j$ (line 70) and $\text{pndTsk}_j[i] = (s, \bullet, \bot)$. The idea is to show that within $\mathcal{O}(1)$ cycles, the system is in state $c \in R$ in which $\text{pndTsk}_i[i] = (s, \bullet, x) : x \neq \bot$. For the case (i), the proof shows that p_i recives a SNAPSHOTack message that matches the first condition in line 89 due to a reply to an SNAPSHOT message in line 106. The proof of case (ii) follows by the fact that all non-failing nodes participate in a helping scheme that solves p_i's task and then write the result to a safe register by calling safeReg() in line 91.

Linearizability. We note that the baseWrite(wp) functions in Algorithms 2 and 3 are identical. Moreover, Algorithm 2's lines 54 to 56 are similar to Algorithm 3's lines 87 to 90, but differ in the following manner: (i) the dissemination of the operation tasks is done outside of Algorithm 2's lines 54 to 56 but inside of Algorithm 3's lines 87, and (ii) Algorithm 2 considers one snapshot operation at a time whereas Algorithm 3 considers many snapshot operations. The linearizability proof of Delporte-Gallet *et al.* [6, Lemma 7] is independent of the task dissemination and result propagation. Moreover, it shows a way to select linearization points according to some partition. The proof there explicitly allows the same partition to include more than one snapshot result. ■

5 Bounded Variations on Algorithms 1 and 3

There is a technique for transforming a self-stabilizing atomic register algorithm that uses unbounded operation indices into one with bounded indices, see [8, Section 10]: [Step-1] once p_i notices an index that is at least $\text{MAXINT} = 2^{64} - 1$, it disables new operations and starts gossiping of the maximal indices (while merging the arriving information with the local one). [Step-2] once all nodes share the same maximal indices, the procedure uses a consensus-based global reset procedure for replacing, per operation type, the highest operation index with its initial value, 0, while keeping the values of all shared registers unchanged. After the end of the global reset procedure, all operations are enabled.

Self-stabilizing global reset procedure. The implementation of the self-stabilizing procedure for global reset can be based on existing mechanisms, such as the one by Awerbuch *et al.* [12]. We note that the system settings of Awerbuch *et al.* [12] assume execution fairness. This assumption is allowed by our system settings (Section 2). This is because we assume that reaching MAXINT can only occur due to a transient fault. Thus, execution fairness, which implies all nodes are eventually alive, is seldom required (only for recovering from transient faults).

6 Discussion

We showed how to transform the two non-self-stabilizing algorithms of Delporte-Gallet *et al.* [6] into ones that can recover after the occurrence of transient faults. This requires some non-trivial considerations that are imperative for self-stabilizing systems, such as the explicit use of bounded memory and the reoccurring clean-up of stale information. Interestingly, these considerations are not restrictive for the case of Delporte-Gallet *et al.* [6]. As a future direction, we propose to consider the techniques presented here for providing self-stabilizing versions of more advanced algorithms, *e.g.,* [13].

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40** (1993) 873–890
2. Anderson, J.H.: Multi-writer composite registers. Distributed Computing **7** (1994) 175–195
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17** (1974) 643–644
4. Dolev, S.: Self-Stabilization. MIT Press (2000)
5. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42** (1995) 124–142
6. Delporte-Gallet, C., Fauconnier, H., Rajsbaum, S., Raynal, M.: Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. IEEE Trans. Parallel Distrib. Syst. **29** (2018) 2033–2045
7. Alon, N., Attiya, H., Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Practically stabilizing SWMR atomic memory in message-passing systems. J. Comput. Syst. Sci. **81** (2015) 692–701
8. Dolev, S., Petig, T., Schiller, E.M.: Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. CoRR **abs/1806.03498** (2018)
9. Georgiou, C., Lundström, O., Schiller, E.M.: Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. CoRR (2019)
10. Georgiou, C., Shvartsman, A.A.: Cooperative Task-Oriented Computing: Algorithms and Complexity. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2011)
11. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
12. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset. In: WDAG. Volume 857 of LNCS., Springer (1994) 326–339
13. Imbs, D., Mostéfaoui, A., Perrin, M., Raynal, M.: Set-constrained delivery broadcast: Definition, abstraction power, and computability limits. In: 19th Distributed Computing and Networking, ICDCN, ACM (2018) 7:1–7:10