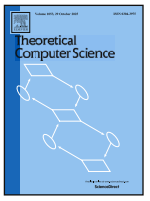
Self-stabilizing snapshot objects for asynchronous failure-prone networked systems

Theoretical Computer Science 1075 (2026) 115929

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs



Self-stabilizing snapshot objects for asynchronous failure-prone networked systems

Chryssis Georgiou [a,*], Oskar Lundström [b], Elad Micheal Schiller [b]

[a] Dept. Computer Science, University of Cyprus, Nicosia, Cyprus
[b] Dept. Computer Science and Engineering, Chalmers Univ. Tech., Gothenburg, Sweden

ARTICLE INFO

Section Editor: Prof. Seth Pettie
Handling Editor: Katie Harris

Keywords:
Snapshot objects
Message-passing systems
Self-stabilization

ABSTRACT

A *snapshot object* simulates the behavior of an array of single-writer/multi-reader shared registers that can be read atomically. In 2018, 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 complete once all write operations have ceased. Their second algorithm allows snapshot operations to *always terminate* independently of write operations.

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, our self-stabilizing algorithms 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. yet they eventually recover from the last occurrence of a transient fault. The main differences are that our proposal considers repeated gossiping of $\mathcal{O}(\nu)$ bit messages, ν being the number of bits for encoding the object, and deals with bounded space (which is a prerequisite for self-stabilization). This facilitates recovery of the registers and sequence numbers after the occurrence of the last transient fault by guaranteeing consistency. We use Lamport's happened-before relation to bound, when possible, the completion time of write and snapshot operations.

1. Introduction

Shared registers are fundamental objects that facilitate synchronization in distributed systems. They provide persistent and consistent distributed storage that can simplify the design and analysis of dependable distributed systems. Snapshot objects extend shared registers; see [1–3]. They allow an algorithm to construct consistent global states of the shared storage without disrupting 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.

* Corresponding author.
E-mail address: chryssis@ucy.ac.cy (C. Georgiou).

https://doi.org/10.1016/j.tcs.2026.115929
Received 1 August 2024; Received in revised form 23 December 2025; Accepted 23 March 2026
Available online 1 April 2026

1.1. 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 [4,5]. 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, which are *linearizable* [6]: the operations write() and snapshot() appear as if they have been executed instantaneously, one after the other (in other words, they appear to preserve real-time ordering).

1.2. Fault model

We consider an asynchronous message-passing system that has no guarantees on the communication delay. Also, we do not assume that the algorithm can explicitly access any clock (or timeout mechanisms). Our fault model includes (i) node crashes, and (ii) communication failures, such as packet omission, duplication, and reordering. 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, or operational assumptions such as that at least half of the system nodes never fail.

We follow the design criteria of *self-stabilization*, which was proposed by Dijkstra [7] and detailed in [8,9]. In particular, when modeling the system, we assume that the above violations bring the system to an arbitrary state from which a *self-stabilizing algorithm* should recover the system within a finite time after the occurrence of the last transient fault.

As transient faults can occur at any point in a system's lifetime, self-stabilizing systems need to keep communicating their state structures to clean any potentially corrupted (stale) information; in this respect, a self-stabilizing system cannot be quiescient [8, Chapter 2.3], *i.e.,* they always need to exchange messages even if no new operations are invocted. We clarify that the rate of this message exchange does not impact the execution time of the write() and snapshot() operations. However, the recovery from transient faults might be delayed as the rate becomes low.

1.3. Related work

Our review does not focus on shared memory systems, such as [10,11]. We also just mention the key design principles of shared memory emulation, see [12,13] for details. Attiya et al. [14] implemented SWMR atomic shared memory in an asynchronous networked system. They assume that the majority of the nodes do not crash or get disconnected. Their work builds on this assumption in the following manner: Any majority subset of the system nodes includes at least one non-faulty node; thus, any two majority subsets of the system nodes have a non-empty intersection. They show that if a majority of the nodes acknowledge an update to the shared register, then that update can safely be considered visible to all non-faulty nodes that retrieve the latest update from a majority of nodes. Attiya et al. also show that this assumption is essential for solvability. Their seminal work has many generalizations and applications [12,15].

Implementing snapshots in message-passing systems. A straightforward way for implementing snapshot objects is to consider a layer of n SWMR atomic registers emulated in a networked system. On top of this layer of these emulated registers, the system can run any algorithm for implementing a snapshot object for a system with shared variables. Delporte-Gallet et al. [16] avoid this composition, obtaining, in this way, a more efficient implementation with respect to the communication costs.

Specifically, Delporte-Gallet et al. claim that when stacking the shared-memory atomic snapshot algorithm of [4] on the shared-memory emulation of [14] (with some improvements), the number of messages per snapshot operation is $8n$, and it takes four round trips. They concurrently scan all emulated SWMR registers using a single message exchange with every writer. Following the end of that scan, a second scan is performed since if the results of both scans are the same, atomicity holds.

Here, *stacking* refers to a two-layer composition in which an existing message-passing algorithm implementing SWMR atomic registers (such as [14]) is used to emulate a shared memory, and then a shared-memory snapshot algorithm (such as [4]) is executed on top of that emulated memory. This indirect construction contrasts with the *direct* approach of Delporte-Gallet et al., which implements the snapshot abstraction on top of the message-passing system without introducing an intermediate SWMR-register layer.

The proposed improvement by Delporte-Gallet et al. takes $2n$ messages per snapshot operation and just one round trip to complete. The algorithms we propose in the present work follow the approach of Delporte-Gallet, which does not consider stacking. Our algorithms have similar communication costs for write and snapshot operations as the ones in [16], while tolerating any failure (in any communication or operation invocation pattern) that the algorithms in [16] can.

The main differences are that our proposal considers repeated gossiping of $\mathcal{O}(\nu)$ bit messages, where ν is the number of bits for encoding the object, and deals with bounded space, which is a prerequisite for self-stabilization. This helps to recover stale values related to the SWMR register and sequence numbers that organize write() and snapshot() operations. At the end of the recovery from the occurrence of the last transient fault, these values are guaranteed to be consistent, and all write() and snapshot() operations perform along the same lines as the solution proposed by Delporte-Gallet et al..

In the context of self-stabilization, there exist algorithms for the propagation of information with feedback, *e.g.,* Delaët et al. [17], which can facilitate the implementation of snapshot objects that can recover after the occurrence of the last transient fault. However,

Delaët et al. do not consider also node failures, as we do in this work. For the sake of clarity, we note that "stacking" of self-stabilizing algorithms for asynchronous systems is not a straightforward process (since the existing "stacking" requires scheduler fairness, see [8, Section 2.7]). Moreover, we are unaware of an attempt in the literature to stack a self-stabilizing shared-memory atomic snapshot algorithm over a self-stabilizing shared-memory emulation.

Recent years have seen significant progress in the design of self-stabilizing abstractions for asynchronous and failure-prone distributed systems. In particular, self-stabilizing consensus has been extensively studied, including indulgent zero-degrading binary consensus [18] and multivalued consensus in asynchronous crash-prone systems [19]. These works establish agreement under transient faults, but focus on decision primitives rather than shared-state abstractions.

Complementary efforts have addressed communication and storage abstractions. Self-stabilizing Byzantine fault-tolerant reliable broadcast has been considered in [20], while [21] studies self-stabilizing shared atomic memory under weak fairness assumptions. At the systems level, Renaissance [22] demonstrates how self-stabilization can be integrated into distributed control planes, and practically-self-stabilizing virtual synchrony [23] explores relaxed stabilization guarantees for group communication.

This work is part of a broader effort to develop self-stabilizing abstractions for asynchronous and failure-prone distributed systems, where different lines of work address complementary problems. Recent results on self-stabilizing consensus provide strong agreement guarantees under demanding settings, including indulgent zero-degrading binary consensus [18] and multivalued consensus in asynchronous crash-prone systems [19,24], but do not offer a shared-memory abstraction for capturing consistent system-wide states.

At the lower levels, self-stabilizing reliable broadcast primitives have been studied, including end-to-end communication [25] and Byzantine fault-tolerant reliable broadcast [20,26]. While these ensure consistent message dissemination, they do not provide the semantics required for coordinated state observation across processes.

At the system level, works such as the *Renaissance* control plane [22] demonstrate the applicability of self-stabilization in complex infrastructures, while practically self-stabilizing virtual synchrony [23] explores relaxations that balance theoretical guarantees with practical constraints.

An earlier version of the two algorithms proposed in this paper appeared in [27,28] without complete proofs, *i.e.,* using proof sketches. This paper includes the complete proofs for the proposed algorithm, and a completely new upper-bound on the operation completion time. Our complementary technical report [13] includes additional details, such as a review of the algorithms in Delporte-Gallet et al. [16] as well as experiments that validate our analysis.

1.4. Our contributions

We present a novel module for dependable networked systems: self-stabilizing algorithms for snapshot objects. To the best of our knowledge, we are the first to provide a broad fault model that includes node failures and transient faults. Specifically, we advance the state of the art as follows:

(1) We offer a self-stabilizing variation on the non-blocking algorithm presented by Delporte-Gallet et al. [16]. As Delporte-Gallet et al. present a non-blocking algorithm, the termination depends on the assumption that the invocation of all write operations cease eventually; yet after termination occurred, write operations may resume.[1] The solution tolerates node failures as well as packet failures. 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 solution can also recover after the occurrence of the last transient fault. We increase the communication costs slightly by resending messages of $\mathcal{O}(\nu)$ bits on every link.

This message exchange facilitates recovery from transient faults independently of the patterns according to which snapshot() operations are invoked. As mentioned, the rate of this message exchange does not impact the execution time of the write() and snapshot() operations; it can be traded with the recovery time from transient faults. Yet after the occurrence of the last transient fault, there is an inherent need to perform one write() operation per writing node to recover in the case of a corrupted writer's internal state.

(2) We also offer a self-stabilizing variation on the always-terminating algorithm presented by Delporte-Gallet et al. [16]. Our algorithm can: (i) recover after the last occurrence of a transient fault, and (ii) both write and snapshot operations always complete (regardless of the invocation patterns of any operation).

To achieve (ii), we use an input parameter, $\delta \in Z^+$. When $\delta = 0$, our self-stabilizing algorithm guarantees an always-termination behavior in a way that resembles the non-self-stabilizing always-terminating algorithm by Delporte-Gallet et al. [16], which incurs $\mathcal{O}(n^2)$ messages per snapshot. When $\delta > 0$, 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 runs concurrently with at least δ writes, it blocks all writes and uses $\mathcal{O}(n^2)$ messages for completing the snapshots.

(3) Using Lamport's happened-before relation [29], we bound to a constant the termination time of the proposed self-stabilizing non-blocking solution and to $\mathcal{O}(n + \delta)$ the self-stabilizing always-termination solution. Specifically, for the case of $\delta > 0$, our solution guarantees that, in the absence of transient faults and the presence of a majority of nodes that never crash, all operations (that were invoked by non-failing nodes) always complete in the presence of asynchrony and any number of node failures (as long as at most a minority of the nodes fail).

[1] Without such an assumption, there are no termination guarantees for Delporte-Gallet et al.'s non-blocking solution. The same holds for our self-stabilizing variation on their solution.

As mentioned, our technical report [13] includes additional details, such as a review of the algorithms in Delporte-Gallet et al. [16] and experiments that validate our analysis. For the sake of consistency with [13], from now on, Algorithms 1 and 2 refer to Delporte-Gallet et al.'s non-blocking and always terminating algorithms, respectively. Also, the proposed algorithms use unbounded counters. In Section 6, we discuss how to bound these counters using existing techniques.

2. System settings

We consider an asynchronous message-passing system that has no guarantees on the communication delay. Also, we do not assume that the algorithm can explicitly access any clock (or timeout mechanisms). The system consists of a set, P, of n failure-prone nodes with unique identifiers. Any pair of nodes $p_i, p_j \in P$ have access to a bidirectional communication channel, $channel_{i,j}$, that, at any time, has at most $\mathsf{capacity} \in N$ packets on transit from p_i to p_j (this assumption is due to a well-known impossibility [8, Chapter 3.2]).

2.1. Execution model

Our analysis considers the *interleaving model* [8], in which the node's program is a sequence of *(atomic) steps*. Each step starts with an internal computation and finishes with a single communication operation, *i.e.,* a message $send$ or $receive$. The *state*, s_i, of node $p_i \in P$ includes all of p_i's variables and $channel_{j,i}$. The term *system state* (or configuration) refers to the tuple $c = (s_1, s_2, \ldots, s_n)$. We define an *execution (or run)* $R = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system states $c[x]$ and steps $a[x]$, such that each $c[x+1]$, except for the starting one, $c[0]$, is obtained from $c[x]$ by $a[x]$'s execution.

2.2. The specifications of the snapshot object task

The set of *legal executions* (LE) refers to all the executions in which the requirements of the task T hold. In this work, T_{snapshot} denotes the task of snapshot object emulation and LE_{snapshot} denotes the set of executions in which the system fulfills T_{snapshot}'s requirements, which consider the operations write() and snapshot(). When node p_i invokes write(v), the system stores value v in the emulated shared memory associated with p_i. When node p_i invokes snapshot(), the system returns an array that includes all the emulated shared memories in a way that fulfills the properties of termination and linearizability. As in Delporte-Gallet et al. [16], termination is defined as the end of operation execution within a finite time. Linearizability is given by the definition of linearizable snapshot histories, which is based on the terms events and event histories. The definitions here are brief versions of the ones that are given by Delporte-Gallet et al. [16].

Events. Let op be a write() or snapshot() operation. The execution of an operation op by a node p_i is modeled by two (atomic) steps: the invocation step, denoted by $invoc(\mathsf{op})$, which calls the op operation, and a response event, denoted by $resp(\mathsf{op})$, which occurs when p_i completes the operation. We use the term *event* to annotate an atomic step that either starts or ends an operation.

Effective operations. Following Delporte-Gallet et al. [16], we provide the following defintion taken from Section 4.1 of [16]. We say that a snapshot() operation is *effective* when the invoking node does not fail during the operation's execution. We say that a write() operation is effective when the invoking node does not fail during its execution, or in case it does fail, the operation's effect is returned by an effective snapshot operation.

Histories. A history is a sequence of operation start and end steps that are totally ordered. We consider histories to compare in an abstract way between two executions of the studied algorithms. Given any two events e and f, $e < f$ if e occurs before f in the corresponding history. A history is denoted by $\widehat{H} = (E, <)$, where E is the set of events. Given an infinite history $\widehat{H}$, we require that: (i) its first event is an invocation and (ii) each invocation is followed by its matching response event. If $\widehat{H}$ is finite, then $\widehat{H}$ might not contain the matching response event of the last invocation event.

Linearizable snapshot history. A snapshot-based history $\widehat{H} = (H, <)$ models a computation at the abstraction level at which the write and snapshot operations are invoked. It is linearizable if there is an equivalent sequential history $\widehat{H}_{seq} = (H, <_{seq})$ in which the sequence of effective write() and snapshot() operations issued by the processes is such that: (1) Each effective operation appears as if executed at a single point of the timeline between its invocation event and its response, and (2) Each effective snapshot() returns an array reg, such that (i) $reg[i] = (v, \bullet)$ if the operation write(v) by p_i appears previously in the sequence and it is the latest one. (ii) Otherwise $reg[i] = \perp$.

When considering a sequential history, it is possible to associate a time instant with each operation. In such a history, all operations are ordered, so no two operations are associated with the same time instant. The following notation is needed to define concurrent operations. Given two arrays reg_1 and reg_2 returned by two snapshot operations, $reg_1 \preceq reg_2$ is a shorthand for $\forall x \in [1..n] : (reg_1[x] \leq reg_2[x])$, and $reg_1 \prec reg_2$ is a shorthand for $(reg_1 \preceq reg_2) \wedge (reg_1 \neq reg_2)$. Given a history $\widehat{H} = (H, <)$ and any two of its operations op_1 and op_2. We say "op_1 precedes op_2", denoted $op_1 \rightarrow op_2$, if $resp(op_1) < invoc(op_2)$. If $\neg(op_1 \rightarrow op_2)$ and $\neg(op_2 \rightarrow op_1)$, we say "$op_1$ and op_2 are concurrent", which is denoted $op_1 \parallel op_2$. It follows that the relation "$\rightarrow$" defined on operations is an irreflexive partial order.

2.3. The fault model and self-stabilization

Node failure. We consider *crash* failures, in which nodes stop taking steps. We assume that the number of failing nodes is bounded by f and that $2f < n$ for the sake of guaranteeing correctness [14].

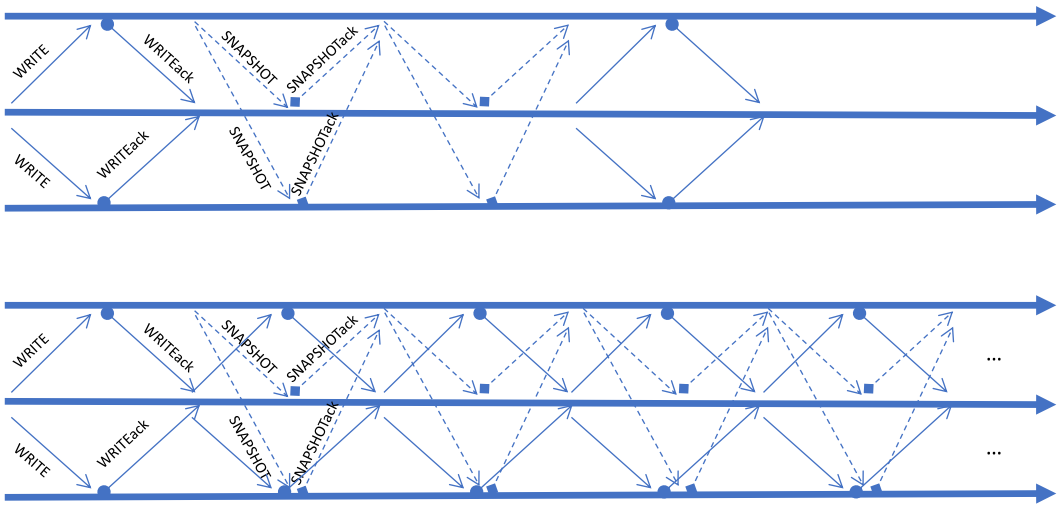


Fig. 1. Examples of Algorithm 1's executions. The upper drawing illustrates a case of a terminating snapshot operation (dashed line arrows) that occurs between two write operations (solid line arrows). The acknowledgments of these messages are arrows that start with circles and squares, respectively. The lower drawing illustrates a case in which every execution of line 14 occurs concurrently with write operations (regardless of whether the algorithm is self-stabilizing or not). Thus, snapshot operations cannot terminate.

Communication failures and fairness. We assume that the communication channels are prone to packet failures, such as omission, duplication, reordering. However, we assume that if p_i sends a message infinitely often to p_j, node p_j receives that message infinitely often. The latter is called the *fair communication* assumption.

Transient faults. We consider any violation of the assumptions according to which the system was designed to operate. We refer to these deviations as *arbitrary transient faults* and assume that they can corrupt the system state arbitrarily (while keeping the program code intact). As in [7–9], we assume that the last transient fault occurs before the system execution starts. Also, it leaves the system in an arbitrary state. An algorithm is *self-stabilizing* with respect to the task of LE, when any system execution R reaches eventually a suffix $R_{legal} \in LE$ that is legal. In other words, self-stabilizing systems recover after the last occurrence of a transient fault.

2.4. Cost measures: The happened-before relation

Lamport [29] defined the happened-before relation as the least strict partial order on events for which: (i) If steps $a, b \in R$ are taken by $p_i \in P$, $a \rightarrow b$ if a appears in R before b. (ii) If a includes sending m that b receives, then $a \rightarrow b$. Using the happened-before definition, one can create a directed acyclic graph $G_R : (V_R, E_R)$, where V_R represents the set of R's states. Also, E_R is given by the happened-before relation. We assume the weight of an edge that is due to cases (i) and (ii) are zero and one, resp. We consider the weight of the heaviest directed path between two state $c, c' \in R$ as the cost measure between c and c' denoted by HBR. As the happened-before relation is a partial order, it is transitive (closure), irreflexive, and asymmetric.

3. Delporte-Gallet et al.'s algorithms

For convenience, we review the Delporte-Gallet et al. [16]'s solutions, which form the basis for our self-stabilizing constructions in Sections 4 to 5. We present Delporte-Gallet et al.'s solutions using our notations.

3.1. The non-blocking algorithm by Delporte-Gallet et al.

We review the non-blocking solution of Delporte-Gallet et al. [16, Algorithm 1], which we refer to as Algorithm 1. The non-blocking solution to snapshot object emulation by Delporte-Gallet *et al.* [16, Algorithm 1] allows all write operations to terminate regardless of the invocation patterns of the other write or snapshot operations (as long as the invoking processors do not fail during the operation). However, for the case of snapshot operations, termination is guaranteed only if eventually the system execution reaches a period in which there are no concurrent write operations.

High-level overview. Algorithm 1 maintains at each process an array `reg[]` of pairs (v, ts), where `ts` is a write index. A write(v) locally increments `ts`, updates `reg[i]`, then completes after a majority acknowledges the broadcast, merging the freshest entries componentwise. A snapshot() repeatedly broadcasts `reg` and returns when two consecutive collects agree, which indicates the absence of concurrent writes and yields an atomic view. Correctness relies on majority intersections to propagate the latest indices and on the max-merge order to ensure monotonicity; non-blocking progress for snapshot() holds once writes quiesce. The message cost is $O(n)$ per write and per collect, and snapshot() needs up to two majority exchanges after the last write to terminate.

Fig. 1 illustrates two representative executions of Algorithm 1. The upper drawing shows a period in which no writes occur between two broadcast rounds, allowing the snapshot operation to stabilize and terminate. In contrast, the lower drawing shows a run in which every collect is interleaved with a concurrent write, preventing the snapshot from observing two identical views and therefore blocking its termination.

Detailed description. As mentioned, Algorithm 1 presents Delporte-Gallet et al. [16, Algorithm 1] using our notation.

Local variables. 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 13). Also, when a message arrives, we use the parameter name xJ to refer to the arriving value for the message field x. Moreover, we denote variable X's value at p_i by X_i.

Algorithm 1: Non-self-stabilizing and non-blocking algorithm by Delporte-Gallet et al. [16], emulating a snapshot object; code for p_i.

```
1  Definitions of ⪯: For integers t and t': (•,t) ⪯ (•,t') ⟺ t ≤ t'; For arrays tab and tab' of (•, integer):
     tab ⪯ tab' ⟺ ∀p_k ∈ P : tab[k] ⪯ tab'[k]; Also, a ≺ b ≡ a ⪯ b ∧ a ≠ b;
2  local variables initialization:
3  ssn := 0; ts := 0;                                   /* snapshot, resp., write operation indices */
4  reg := [⊥, ..., ⊥];          /* shared registers (⊥ is smaller than any possibly written value) */
5  macro merge(Rec) for p_k ∈ P do reg[k] ← max({reg[k]} ∪ {r[k] | r ∈ Rec});
6  operation write(v) begin
7      ts ← ts + 1; reg[i] ← (v, ts); let lReg := reg;
8      repeat broadcast WRITE(lReg); until WRITEack(regJ ⪰ lReg) received from a majority;
9      merge(Rec) where Rec is the set of reg arrays received at line 8;
10     return();
11 operation snapshot() begin
12     repeat
13         let prev := reg; ssn ← ssn + 1;
14         repeat broadcast SNAPSHOT(reg, ssn); until SNAPSHOTack(•, ssnJ = ssn) received from a majority;
15         merge(Rec) where Rec is the set of reg arrays received at line 14;
16     until prev = reg;
17     return(reg);
18 upon message WRITE(regJ) arrival from p_j begin
19     for p_k ∈ P do reg[k] ← max_⪯(reg[k], regJ[k]);
20     send WRITEack(reg) to p_j;
21 upon message SNAPSHOT(regJ, ssn) arrival from p_j begin
22     for p_k ∈ P do reg[k] ← max_⪯{reg[k], regJ[k]};
23     send SNAPSHOTack(reg, ssn) to p_j;
```

Processor p_i stores an array reg of $|P|$ elements (line 4), such that the kth entry stores the most recent information about processor p_k's object value and $reg[i]$ stores p_i's actual object value. Every entry is a pair of the form (v, ts), where the field v is a ν-bits object value and ts is an unbounded integer that stores the object timestamp. The values of ts serve as the index of p_i's write operations. Similarly, p_i maintains an index for the snapshot operations, ssn (sequence number). Algorithm 1 defines also the relation $\preceq$ that compares (v, ts) and (v', ts') according to the write operation indices (line 1).

The write() **operation.** Algorithm 1's write() operation appears in lines 6 to 10 (client-side) and lines 18 to 20 (server-side). The client-side operation write(v) stores the pair (v, ts) in $reg[i]$ (line 7), where p_i is the invoking processor and ts is a unique operation index. The primitive broadcast sends to all the processors in P the message WRITE about p_i's local perception of reg's value.

Upon the arrival of a WRITE message to p_i from p_j (line 18), the server-side code is run. Processor p_i updates reg according to the timestamps of the arriving values (line 19). Then, p_i replies to p_j with the message WRITEack (line 23), 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 processors in P until it receives replies from a majority of them (line 8). Once that happens, it uses the arriving values for keeping reg up-to-date (line 9).

The snapshot() **operation.** Algorithm 1's snapshot() operation appears in lines 11 to 17 (client-side) and lines 21 to 23 (server-side). Recall that Delporte-Gallet et al. [16, Algorithm 1] is non-blocking with respect to the snapshot operations as long as there are no concurrent write operations. Thus, the client-side is written in the form of a repeat-until loop. Processor p_i tries to query the system for the most recent value of the shared registers. The success of such attempts, *i.e.,* satisfying the if-statement condition of line 16, depends on the above assumption. Therefore, before each such broadcast, p_i copies reg's value to $prev$ (line 13) and exits the repeat-

until loop only when the updated value of reg indicates that there are no concurrent write operations. We note that the absence of concurrent writes implies the success of the atomic snapshot since it considers all previous write operations.

Algorithm 2: Non-self-stabilizing and always-terminating algorithm by Delporte-Gallet et al. [16], emulating a snapshot object; p_i's code.

```
24 local variables initialization: ssn, sns, ts := 0;                    /* snapshot and write operation indices */
25 reg := [⊥, ..., ⊥];                  /* shared registers (⊥ is smaller than any possibly written value) */
26 writePending ← ⊥;                                                      /* stores p_i's write task */
27 foreach k, s : repSnap[k, s] := ⊥;                                    /* p_k's snapshot result for index s */
28 macro merge(Rec) for p_k ∈ P do reg[k] ← max({reg[k]} ∪ {r[k] | r ∈ Rec});
29 do forever begin
30     if (writePending ≠ ⊥) then baseWrite(writePending); writePending ← ⊥;
31     if (there are messages SNAP() received and not yet processed) then
32         let SNAP(source, sn) be the oldest of these messages;
33         baseSnapshot(source, sn);
34         wait until (repSnap[source, sn] ≠ ⊥);
35 operation write(v) begin
36     writePending ← v; wait until (writePending = ⊥); return();
37 operation snapshot() begin
38     sns ← sns + 1; reliableBroadcast SNAP(i, sns);
39     wait until (repSnap[i, sns] ≠ ⊥); return(repSnap[i, sns]);
40 function baseWrite(v) begin
41     ts ← ts + 1; reg[i] ← (ts, v); let lReg := reg;
42     repeat broadcast WRITE(lReg); until WRITEack(regJ ≥ lReg) received from a majority;
43     merge(Rec) where Rec is the set of reg arrays received at line 42;
44 function baseSnapshot(s, t) begin
45     while repSnap[s, t] = ⊥ do
46         let prev := reg; ssn ← ssn + 1;
47         repeat
48             broadcast SNAPSHOT(s, t, reg, ssn);
49         until SNAPSHOTack(sJ = s, tJ = t, •, ssnJ = ssn) received from a majority);
50         merge(Rec) where Rec is the set of reg arrays received at line 49;
51         if prev = reg then reliableBroadcast END(s, t, prev);
52 upon message WRITE(regJ) arrival from p_j begin
53     for p_k ∈ P do reg[k] ← max_≺(reg[k], regJ[k]);
54     send WRITEack(reg) to p_j;
55 upon message SNAPSHOT(s, t, regJ, ssnJ) arrival from p_j begin
56     for p_k ∈ P do reg[k] ← max_≺(reg[k], regJ[k]);
57     send SNAPSHOTack(s, t, reg, ssnJ) to p_j;
58 upon message END(s, t, val) arrival from p_j do repSnap[s, t] ← val;
```

3.2. The always-terminating algorithm by Delporte-Gallet et al.

Whereas Algorithm 1 may fail to terminate a snapshot operation in the presence of an unbounded number of concurrent writes, Delporte-Gallet et al.'s always-terminating algorithm (Algorithm 2 in [16]) eliminates this limitation by globally coordinating ongoing snapshot tasks and postponing interfering write operations. Delporte-Gallet et al. [16, Algorithm 2] guarantees termination for any invocation pattern of write and snapshot operations, as long as the invoking processors do not fail during these operations. Its advantage over Delporte-Gallet et al. [16, Algorithm 1] is that it can deal with an infinite number of concurrent write operations. This is because it guarantees the non-blocking progress criterion for the snapshot operations. We present [16, Algorithm 2] in Algorithm 2 using the presentation style of this paper. We review Algorithm 2 while pointing out some key challenges that exist when considering self-stabilization.

High-level overview. Delporte-Gallet et al. [16, Algorithm 2] uses a job-stealing scheme for allowing rapid termination of snapshot operations. Processor $p_i \in P$ starts its snapshot operation by queueing this new task at all processors $p_j \in P$. Once p_j receives p_i's

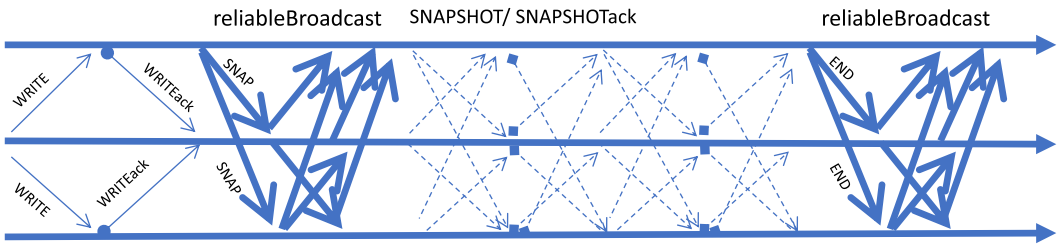


Fig. 2. Algorithm 2's execution for the case depicted by the upper drawing of Fig. 1. The drawing illustrates a case of a terminating snapshot operation (dashed line arrows) that occurs after (a non-concurrent) write operation (thin solid line arrows). The acknowledgments of these messages are arrows that start with circles and squares, respectively.

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 processors are aware of all on-going snapshot operations.

This joint awareness allows the system processors to make sure that no write operation can stand in the way of on-going snapshot operations. To that end, the processors 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.

Detailed description. Algorithm 2 extends Algorithm 1 in the sense that it uses all of Algorithm 1's variables and two additional ones. These are the second operation index, sns, and an array $repSnap$, which snapshot() operations use. The entry $repSnap[x,y]$ holds the outcome of p_x's yth snapshot operation, where no explicit bound on the number of invocations of snapshot operations is given.

In the context of self-stabilization, the use of such unbounded variables is not possible. The reasons are that real-world systems have bounded size memory as well as the fact that a single transient fault can bring any counter to its near overflow value and fill up any finite capacity buffer. We discuss the way around this challenge in Section 6.

The write() *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 $writePending_i$ (line 36). The algorithm then runs the write operation as a background task (line 30) using the baseWrite() function (lines 40 to 43).

The snapshot() *operation.* A call to snapshot() (line 38) causes p_i to reliably broadcast, via the primitive reliableBroadcast, a new sns index in a SNAP message to all processors in P. Processor p_i then waits for the task's completion by placing it as a background task (line 39). We note that for our proposed solutions we do not assume access to a reliable broadcast mechanism such as reliableBroadcast; see Section 5 for details and an alternative approach that uses safe registers instead of the (more involved) reliableBroadcast primitive.

The baseSnapshot() *function.* This function essentially follows the snapshot() operation of Algorithm 1. That is, Algorithm 2's snapshot repeat-until loops iterates until the retrieved reg vector equals to the one that was known prior to the last repeat-until iteration. Algorithm 2's baseSnapshot() procedure returns after at least one snapshot process has terminated. In detail, processor p_i stores in $repSnap[s,t]$, via a reliable broadcast of the END message, the result of the snapshot process (line 51 and 58).

Synchronization between the baseWrite() *and* baseSnapshot() *functions.* Algorithm 2 interleaves the background tasks in a do-forever loop (lines 30 to 34). As long as there is an awaiting write task, processor p_i runs the baseWrite() function (line 30). Also, if there is an awaiting snapshot task, processor p_i selects the oldest task, $(source, sn)$, and uses the baseSnapshot$(source, sn)$ function. Here, Algorithm 2 blocks its execution until $repSnap[source, sn]$ contains the result of that snapshot task.

Note that line 31 implies that Algorithm 2 does not explicitly assume that processor p_i has bounded space for storing SNAP messages. In the context of self-stabilization, there must be an explicit bound on the size of all memory in use. We discuss how to overcome this challenge in Section 6.

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

4. Stabilizing non-blocking snapshot

We propose Algorithm 3 as an extension of Algorithm 1, *i.e.,* Delporte-Gallet et al. [16, Algorithm 1]; the boxed code lines mark our additions to Algorithms 1 (Section 3.1). Algorithms 1 and 3 differ in their ability to deal with stale information, which can appear in the system due to transient faults. The version presented here uses unbounded counters. As mentioned, in Section 6 we discuss how to bound these counters using existing techniques [21].

4.1. High-level overview

Algorithm 3 extends Delporte-Gallet et al. [16, Algorithm 1] (which appears as Algorithm 1 in this paper) with a perpetual gossip and with checks that repair stale local state after transient faults. The boxed lines implement two repairs: (i) keep the local write index `ts` (timestamp) and the local copy $reg[i].ts$ synchronized upward by merging maxima from incoming messages and gossip, and

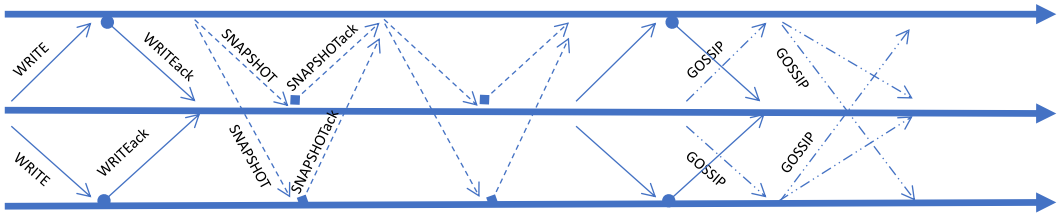


Fig. 3. Example execution of Algorithm 3. A snapshot operation (dashed arrows) occurs between two write operations (solid arrows). Acknowledgments for writes and snapshots are shown as arrows beginning with circles and squares, respectively. Concurrent gossip messages (not all shown explicitly) propagate the maximum timestamp values throughout the system, ensuring that processes recover from transient faults and that both writes and snapshots eventually observe consistent views.

(ii) ignore snapshot acknowledgments whose sequence number does not match the broadcaster's current `ssn` (snapshot sequence number). These repairs guarantee that, after the last transient fault, all per-writer timestamps and per-caller snapshot indices become consistent again, after which the external behaviour simulates Algorithm 1. Hence, linearizability follows from the baseline solution, while termination bounds are preserved modulo a constant number of happened-before steps for the repair phase; the gossip rate only affects recovery time from transient fault, not operation latency.

As mentioned, Algorithm 1 is derived from Algorithm 3, when the boxed code lines of Algorithm 3 highlight our modifications. We now turn to explain the rule of the boxed code lines. Without lines repairing `ts` and `ssn`, a transient fault can leave a process with a stale local index that never catches up, causing it to accept replies for an old snapshot or to publish values dominated by newer timestamps. The repair ensures indices monotonically converge to the current maxima and that acknowledgments are matched to the broadcaster's current query, which together imply recovery to a legal state before resuming the baseline behaviour.

4.2. Detailed description

Algorithm 3 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 in-transit message.

For the case of corrupted ssn, p_i's client ignores arriving messages with ssn not matching ssn_i (line 80).

For the case of corrupted ts, p_i's do-forever loop makes sure ts_i is not smaller than $reg_i[i].ts$ (line 66) before gossiping to every $p_j \in P$ its local copy of p_j's shared register (line 67). Also, upon the arrival of such messages, Algorithm 3 merges the arriving information with the local one (line 85). When the replies arrive to p_i, it merges the arriving ts with ts_i (line 63).

For intuition, Fig. 3 illustrates a typical execution of Algorithm 3, showing how writes, snapshot requests, and gossip messages interact to maintain consistency and to recover from transient faults. Specifically, the figure highlights that gossip messages (lines 67 and 85) disseminate the maximal timestamp (ts) values in the system, allowing each process to repair stale local state after the last transient fault. As a result, both the write and the snapshot operations eventually converge to consistent views, which is essential for proving the correctness of the self-stabilizing algorithm.

4.3. Correctness

Although the extension presented in Algorithm 3 includes only a few changes, proving correctness is not trivial. It requires demonstrating that Algorithm 3 ensures the system reaches a legal execution starting from any arbitrary initial state (Theorem thm:mainConvergence) and that Algorithm 3 satisfies the problem requirements during legal executions (Theorem thm:closure1).

Notation and definitions. Definition 1 refers to p_i's timestamps and sequence numbers, where $p_i \in P$. The set of p_i's timestamps includes ts_i, $reg_i[i].ts$, $reg_j[i].ts$ and the value of $reg_m[i].ts$ in the payload of any in-transit message m. The set of p_i's sequence numbers includes ssn_i and the value of ssn_m in the payload of any SNAPSHOT or SNAPSHOTack message m from, and resp., to p_i that is in-transit in the system.

Definition 1. Consider an execution R of Algorithm 3. Let $c \in R$ be a system state.

(i) Suppose ts_i is greater than or equal to any p_i's timestamps in the variables and fields related to ts. We say that the ts's timestamps are consistent in c.
(ii) Suppose ssn_i is greater than or equal to any p_i's sequence numbers in the variables and fields related to ssn. We say that the ssn's sequence numbers are consistent in c.

Theorem 1 shows recovery after the last occurrence of a transient fault. Theorem 2 shows that the system cannot leave the set of legal executions in the absence of transient faults.

Theorem 1.
Let R be a Algorithm 3's execution. Eventually the system reaches a system state, $c \in R$, in which ts' timestamps and ssn's sequence numbers are consistent in c (Definition 1).

Algorithm 3: Self-stabilizing algorithm for non-blocking snapshot object; code for p_i.

59 **Definitions of** $\preceq$: For integers t and t': $(\bullet, t) \preceq (\bullet, t') \iff t \leq t'$; For arrays tab and tab' of $(\bullet, integer)$: $tab \preceq tab' \iff \forall p_k \in P : tab[k] \preceq tab'[k]$; Also, $a \prec b \equiv a \preceq b \wedge a \neq b$;

60 **local variables:** $ssn := 0; ts := 0;` /* snapshot, resp., write operation indices */`

61 $reg[] := [\perp, \ldots, \perp];$ `/* shared registers (where ⊥ represents a value smaller than any pair of a written value and a timestamp) */`

62 **macro** merge(Rec) **begin**

63 $ts \leftarrow \max(\{ts, reg[i].ts\} \cup \{r[i].ts \mid r \in Rec\})$; `/* select the greatest timestamp */`

64 **for** $p_k \in P$ **do** $reg[k] \leftarrow \max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\})$;

65 **do forever begin**

66 $ts \leftarrow \max\{ts, reg[i].ts\}$; `/* select the greatest timestamp */`

67 **for** $p_k \in P : k \neq i$ **do send** GOSSIP($reg[k]$) to p_k;

68 **operation** write(v) **begin**

69 $ts \leftarrow ts + 1; reg[i] \leftarrow (v, ts)$; **let** $lReg := reg$;

70 **repeat**

71 broadcast WRITE($lReg$);

72 **until** WRITEack($regJ \geq lReg$) *received from a majority* `//test whether more than n/2 write acknowledgments from distinguished senders were received, such that the arriving array, regJ is ≥ lReg`;

73 merge(Rec) **where** Rec is the set of reg arrays received at line 72;

74 **return**();

75 **operation** snapshot() **begin**

76 **repeat**

77 **let** $prev := reg$; $ssn \leftarrow ssn + 1$;

78 **repeat**

79 broadcast SNAPSHOT(reg, ssn);

80 **until** SNAPSHOTack($\bullet, ssnJ = ssn$) *received from a majority*;

81 merge(Rec) **where** Rec is the set of reg arrays received at line 80;

82 **until** $prev = reg$;

83 **return**(reg);

84 **upon** GOSSIP($regJ$) **arrival from** p_j **begin**

85 $reg[i] \leftarrow \max\{reg[i], regJ\}; ts \leftarrow \max\{ts, reg[i].ts\}$;

86 **upon** WRITE($regJ$) **arrival from** p_j **begin**

87 **for** $p_k \in P$ **do** $reg[k] \leftarrow \max_{\preceq}(reg[k], regJ[k])$;

88 **send** WRITEack(reg) **to** p_j;

89 **upon** SNAPSHOT($regJ, ssn$) **arrival from** p_j **begin**

90 **for** $p_k \in P$ **do** $reg[k] \leftarrow \max_{\preceq}\{reg[k], regJ[k]\}$;

91 **send** SNAPSHOTack(reg, ssn) to p_j;

Theorem thm:mainConvergence's proof is implied by Lemmas 1 and 2, which can be read sequentially until Theorem thm:mainConvergence's end of proof symbol "$\square_{Theorem\ 1}$".

Proof. The operation write(v) (line 68) starts by changing the state of the invoking node, *i.e.,* incrementing ts (line 69). Lemma 1 shows the conditions that holds immediately before that increment occurs.

Lemma 1.
Eventually, the system reaches $c \in R$ in which ts_i is greater than or equal to any p_i's timestamp value. Also, suppose p_i takes a step immediately after c that includes line 69. Then in c, $ts_i = reg_i[i].ts = reg_j[i].ts$ and $m \in channel_{i,j} \cup channel_{j,i} : m.reg[i].ts = ts_i$.

Lemma 1's proof is implied by Claims 1–4, which can be read sequentially until Theorem thm:eventuallyAscendingSent's end of proof symbol "$\square_{Lemma\ 1}$".

Proof. Claim 1 denotes by $X_{i,\ell}$ the ℓth value stored in X_i during R, where $\ell \in N$.

Claim 1.
The sequences $ts_{i,\ell}$, $reg_{i,\ell}[i].ts$, $reg_{j,\ell}[i].ts$, $reg_{i,\ell}[i]$ and $reg_{j,\ell}[i]$ are non-decreasing.

Proof. Algorithm 3 does only the following actions on ts and reg fields: increment (line 69) and merge using the max function (lines 63, 64, 66, 81, 85, 87 and 90), *i.e.,* there are no assignments. The claim holds since these fields are never decremented during R. $\square_{Claim\ 1}$

Claim 2.

Eventually $ts_i \geq reg_i[i].ts$.

Proof. Since R is unbounded, $p_j \in P$ calls lines 66, 67, and 85 for an unbounded number of times.

By the proof of Claim 1,

only line 69 changes ts_i, via an increment whereas lines 63, 66, and 85 update ts_i and $reg_i[i].ts$ by taking the maximum of $\{ts_i, reg_i[i].ts\}$. The rest of the proof is implied by Claim 1, and the fact that p_i executes lines 66, 67 and 85 eventually. $\square_{Claim\ 2}$

Algorithm 3 sends GOSSIP messages in line 67, request messages in lines 71 and 79 as well as replies in lines 88 and 91. Claim 3's proof considers lines 71 and 79 in which p_i sends a request message to p_j, whereas Claim 4's proof considers lines 67, 88 and 91 in which p_j replies or gossips to p_i.

Claim 3.

Let $m \in channel_{i,j}$ *and* reg_m *be the value of the* reg *field in* m*, where* $p_i, p_j \in P$ *are non-failing. Eventually,* $reg_i[i].ts \geq reg_m[i].ts$ *and* $reg_i[i].ts \geq regJ[i].ts$ *whenever* p_j *raises the events* GOSSIP(), WRITE() *or* SNAPSHOT().

Proof. Suppose p_i indeed sends m, *i.e.,* m does not appear in R's starting state. Let $a_k \in R$ be the first step in which p_i calls line 67, 71 or 79 and for which there is $a_{depart,k} \in R$, which appears after a_k and in which m is sent.

The value of $reg_m[i].ts$

is defined by

$reg_i[i].ts$ in the state that immediately precedes a_k. The rest of the proof relies on the fact that until m arrives to p_j,

$reg_i[i].ts \geq reg_m[i].ts$ holds (Claim 1).

Let $a_{arrival,k} \in R$ be the first step that appears after $a_{depart,k}$ in R in which p_j delivers m. By the assumption that R is unbounded, $a_{arrival,k} \in R$ eventually. During $a_{arrival,k}$, node p_j raises the message delivery event GOSSIP($regJ$) (when a_k considers line 85), WRITE($regJ$) (when a_k considers line 86) or SNAPSHOT($regJ, ssn$) (when a_k considers line 89), such that $reg_i[i].ts \geq reg_m[i].ts = regJ[i].ts$.

Suppose $a_k \notin R$, *i.e.,* m appears in R's starting state. Eventually, all messages in transit to p_j arrive (or lost). Thus, there is a suffix, R', of R in which all delivered messages during R' were indeed sent during R. Thus, the above proof holds with respect to messages received in R' that were sent in R. $\square_{Claim\ 3}$

Claim 4 shows that the dissemination of timestamp information through gossip, write acknowledgments, and snapshot acknowledgments guarantees that both the receiving process and the writer eventually hold timestamp values at least as large as those carried by any in-transit message. To prepare for the claim statement and proof, we clarify the roles of the processes involved. Process p_k is the *sender* of message m; it performs one of the send operations at lines 67, 88, or 91, and the vector reg_m contained in m reflects p_k's local state immediately before that send step. Process p_j is the *receiver* that eventually delivers m and raises the corresponding event, such as GOSSIP($regJ$), WRITEack($regJ$), or SNAPSHOTack($regJ, \bullet$). Process p_i does not participate in the transmission or reception of m; instead, it is the writer index whose timestamp entry $reg[i].ts$ is being tracked. Accordingly, the proof compares (a) the value $reg_m[i].ts$ carried in m, (b) the timestamp held by p_j upon delivery, and (c) the local timestamp $reg_i[i].ts$ maintained by p_i, where $reg_j[i]$ refers to p_j's local copy of the timestamp associated with writer p_i. All three processes are assumed non-failing, and message delivery is eventual. Claim 4 then establishes that both p_j and p_i eventually maintain timestamp values no smaller than those contained in any message in transit, a property essential to the convergence arguments that follow.

Claim 4.

Let $m \in channel_{j,k}$ *and* reg_m *be the value of the* reg *field in* m*, where* $p_i, p_j, p_k \in P$ *are non-failing nodes and* $i = k$ *may or may not hold. Eventually,* $reg_j[i].ts \geq reg_m[i].ts$ *and* $reg_i[i].ts \geq regJ[i].ts$ *whenever node* p_k *raises the events* GOSSIP($regJ$), WRITEack($regJ$) *or* SNAPSHOTack($regJ, \bullet$).

Proof. Suppose p_k sends m, *i.e.,* m does not appear in R's starting state. Let $a_k \in R$ be the first step in R in which p_k calls line 67, 88 or 91 and for which $\exists a_{depart,k} \in R$ that appears in R after a_k. Note that $reg_m[i].ts$ is defined by $reg_i[i].ts$ in the state that immediately precedes a_k. The rest of the proof relies on the fact that until m arrives to p_j, the invariant $reg_i[i].ts \geq reg_m[i].ts$ holds (due to Claim 1).

Let $a_{arrival,k} \in R$ be the first step that appears after $a_{depart,k}$, if there is any such step, in which the p_j delivers m that $a_{depart,k}$ transmits. Eventually, $a_{arrival,k} \in R$, during which p_j raises GOSSIP($regJ$) (when a_k considers line 67), WRITEack($regJ$) (when a_k considers line 88), or SNAPSHOTack($regJ, \bullet$) (when a_k considers line 91), such that $reg_i[i].ts \geq reg_m[i].ts = regJ[i].ts$. For the $a_k \notin R$ case, the proof follows Claim 3's arguments. $\square_{Claim\ 4}$ $\square_{Lemma\ 1}$

Lemma 2.

Let R *be an Algorithm 3's unbounded execution. Eventually, the system reaches* $c_x \in R$ *in which* ssn_i *is greater than or equal to any* p_i*'s sequence number.*

Proof. Claims 5–7 prove the lemma.

Claim 5.

The sequence $ssn_{i,\ell}$ *is non-decreasing.*

Proof. Algorithm 3 only increments (line 77), and assigns (lines 79 and 91) ssn values. Thus, the claim is true, because ssn is never decremented. $\square_{Claim\ 5}$

The proofs of Claims 6 and 7 follow by similar arguments to the ones of Claims 3 and 4.

Claim 6.

Let $m \in channel_{i,j}$ be a SNAPSHOT *message on transit from p_i to p_j that includes the field $ssn = ssn_m$. Eventually, $ssn_i \geq ssn_m$; also when p_j raises the* SNAPSHOT() *event.*

Claim 7.

Let $m \in channel_{j,i}$ be a SNAPSHOTack *message on transit from p_j to p_i and ssn_m the value of the reg field in m. Eventually, $ssn_i \geq ssn_m$; also when p_j raises the* SNAPSHOTack() *event.* $\square_{Lemma\ 2}$ $\square_{Theorem\ 1}$

Theorem 2 shows Algorithm 3's termination and linearization properties. It also bounds the cost of write() and snapshot() operations. Since termination is not always guaranteed, the latter bound considers executions eventually without write() invocations.

Theorem 2.

Let c be a system state with consistent timestamps and sequence numbers. Let R be an Algorithm 3's execution that starts in state c. (i) Execution R is legal w.r.t. atomic snapshot objects. (ii) write() *cost is two HBRs. (iii) After the last occurrence of* write(), *the* snapshot() *cost is four HBRs.*

Proof.

Any step that includes line 85 does not change the state of the calling node, because every timestamp uniquely couples an object value (line 66) and that timestamp is consistent (Lemma 1).

The rest of the proof considers $Alg_{noGOSSIP}$ that is obtained from the code of Algorithm 3 by the removal of lines 67 and 85, in which the gossips are sent and received, resp. We use this definition to show that $Alg_{noGOSSIP}$ simulates Algorithm 1. This means that from the perspective of its external behavior (*i.e.,* its requests, replies and failure events), any trace of $Alg_{noGOSSIP}$ has a trace of Algorithm 1 since c is consistent. Since Algorithm 1 satisfies the task of emulating snapshots, it holds that $Alg_{noGOSSIP}$ and Algorithm 3 satisfy it as well.

Recall that every timestamp uniquely couples an object value (line 66) as well as that timestamps and sequence numbers are consistent in every state throughout R (Lemma 1). These facts imply that also lines 63 and 66 do not change the state of the calling node.

Observe that each write() requires a single round of majority communication. Thus, write() cost is two HBRs. Also, after the last write(), it takes at most two rounds of majority communication for any snapshot() to end, *i.e.,* snapshot() cost is four HBRs.

$\square_{Theorem\ 2}$

5. Stabilizing always-terminating snapshot

We propose Algorithm 4 as a variation on Delporte-Gallet et al. [16, Algorithm 2]. (Note that due to its length, Algorithm 4 appears in two parts.) These algorithms differ mainly in their ability to recover after the last occurrence of a transient fault, which imposes constraints on how pending snapshot tasks are managed.

Specifically, Algorithm 4 must have a clear bound on the number of pending and completed snapshot tasks. Thus, Algorithm 4 restricts each process to at most one outstanding snapshot task. As a consequence, the algorithm distinguishes between two operational modes:

(i) the *normal mode*, in which fewer than δ concurrent writes have occurred, and the snapshot may terminate normally, and
(i) the *helping mode*, which is entered once at least δ concurrent writes have been observed, meaning that the current snapshot cannot complete without external assistance. Thus, the initiating process (and potentially other processes) cooperatively complete the snapshot by disseminating partial results and finalizing them through reliable storage using SAVE and SAVEack messages.

We clarify that the two conditions do not specify conflicting criteria for deferring tasks; rather, they determine whether the helping scheme must be invoked or whether the task can continue without it. Moreover, we use vector clocks to detect the number of concurrent writes, and the helping mechanism guarantees termination even under unbounded concurrency. This high-level structure explains the purpose of the macros $\Delta(k)$, Δ, and safeReg, and clarifies how Algorithm 4 achieves always-termination in a self-stabilizing setting.

The version presented here uses unbounded counters; as discussed in Section 6, these can be bounded using existing techniques [21]. We clarify that this transformation is self-stabilizing, *i.e.,* it does not leave the set of legal executions, unlike earlier approaches such as practically self-stabilizing systems [30,31] and pseudo-self-stabilization [32].

Algorithm 4 lets every node disseminate its (at most one) pending snapshot task and uses techniques for emulating safe registers for delivering the result. In other words, once a node finishes a snapshot task, it broadcasts the result and waits for replies from a majority, which may possibly include the initiator of the snapshot task (using the macro safeReg(), line 98). This way, if p_j notices that it has the result of an ongoing snapshot task, it sends that result to the requesting node.

Algorithm 4: Part I of self-stabilizing always-terminating snapshot; p_i's code

92 **input:** δ a number of observed concurrent writes after which write operations block temporarily;

93 **local variables initialization (optional in the context of self-stabilization):** $ts := 0$ is p_i's write operation index; $ssn, sns := 0$ are p_i's snapshot operation indices; $reg[n] := [\bot, \dots, \bot]$ buffers all shared registers; $writePending \leftarrow \bot$ stores p_i's write task; $pndTsk[n] := [(0, \bot, \bot), \dots, (0, \bot, \bot)]$ control variables of snapshot operations; each entry form is (sns, vc, fnl), where sns is an index, vc is a vector clock that timestamps the snapshot operation sns, and fnl is the operation's returned value;

94 **macro** $\text{VC} := [ts_k]_{p_k \in P}$ **where** $ts_k := 0$ **when** $reg[k] = \bot$ **otherwise** $ts_k = x$ where $reg[k] = (\bullet, x)$;

95 **macro** $exceeds(k) := (\delta \leq \sum_{\ell \in \{1,\dots,n\}} \text{VC}[\ell] - pndTsk[k].vc[\ell])$;

96 **macro** $\Delta(k) := \{(k, pndTsk[k].sns, pndTsk[k].vc) : pndTsk[k].fnl = \bot \wedge ((\delta = 0 \wedge pndTsk[k].sns > 0) \vee (pndTsk[k].vc \neq \bot \wedge exceeds(k)))\}$;

97 **macro** $\Delta := (\bigcup_{p_k \in P} \Delta(k)) \cup \{(i, pndTsk[i].sns, pndTsk[i].vc) : pndTsk[i].sns > 0 \wedge pndTsk[i].fnl = \bot\}$;

98 **macro** safeReg(A) **repeat** broadcast SAVE(A) **until** majority of SAVEack(AJ=$\{(k, s) : (k,s,\bullet) \in A\}$) arrived;

99 **macro** merge(Rec) $\{ts \leftarrow \max(\{ts, reg[i].ts\} \cup \{r[i].ts \mid r \in Rec\})$; **for** $p_k \in P$ **do** $reg[k] \leftarrow \max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\})\}$;

100 **do forever begin**

`/* Delete out-of-sync SNAPSHOTack messages */`

101 **foreach** $ssn' \neq ssn$ **do delete** SNAPSHOTack($-, ssn'$);

`/* Update local timestamp and snapshot number */`

102 $(ts, sns) \leftarrow (\max\{ts, reg[i].ts\}, \max\{sns, pndTsk[i].sns\})$;

103 **for** $k \in \{1, \dots, n\} : \neg(pndTsk[k].vc \preceq \text{VC})$, **where** *line 59 defines the relation* $\preceq$ **do** $pndTsk[k].vc \leftarrow \bot$ `/* Fix illogical vector clocks;`

104 **if** $sns \neq pndTsk[i].sns$ **then**

105 $pndTsk[i] \leftarrow (sns, \bot, \bot)$ `/* Reset corrupted snapshot entries`

`/* Send gossip messages to other processes */`

106 **for** $p_k \in P : k \neq i$ **do send** GOSSIP($reg[k], pndTsk[k].sns$) to p_k;

`/* Perform pending writes */`

107 **if** $writePending \neq \bot$ **then**

108 {baseWrite($writePending$); $writePending \leftarrow \bot$;}

109 **if** $\Delta \neq \emptyset$ **then** baseSnapshot(Δ) `/* Perform pending snapshots;`

5.1. High-level overview

Algorithm 4 augments the baseline with a helping scheme that disseminates at most one pending snapshot task per process and writes results back using a safe-register style pattern. A vector-clock summary VC timestamps the currently known $reg[]$ entries; when a snapshot observes at least δ concurrent writes, the algorithm temporarily prioritizes completion of the in-flight snapshots, then resumes writes. This balances write and snapshot latencies: $\delta = 0$ yields unconditional completion akin to the always-terminating baseline, while $\delta > 0$ keeps typical snapshot costs at $O(n)$ messages and falls back to $O(n^2)$ only when necessary. As in Algorithm 3, gossip and max-merge restore consistency of indices after transient faults, thereby ensuring convergence, termination, and linearizability under the same failure model.

Fig. 4 provides two examples of Algorithm 4. In the upper drawing, a write is followed by a single snapshot, highlighting that significantly fewer messages are exchanged compared to the non-blocking execution example in Fig. 2. The lower drawing shows the case where all processes invoke snapshot operations concurrently. Both drawings emphasize how Algorithm 4 improves efficiency: unlike Algorithm 2, which uses $\mathcal{O}(n^2)$ messages per snapshot and processes one snapshot task at a time, Algorithm 4 reduces the message cost under low contention and achieves higher throughput when multiple snapshots occur concurrently.

5.2. Detailed description

We review Algorithm 4's do-forever loop (lines 101 to 109), the baseSnapshot() function, the dealing with message SNAPSHOT (lines 141 to 142), and the macro safeReg(s, r) (line 98) together with the dealing of message SAVE (lines 131 to 135).

5.3. Macros

For the sake of brevity, the code of Algorithm 4 uses several macros (lines 94 to 99). We use the symbol $\bullet$ (lines 125, 128, 129, 130, 133, 135, and 145) to denote a finite sequence of arbitrary values whose specific contents are immaterial to the discussion. For example, the triplet (k, s, vc) can be represented by $(k, \bullet)$, $(k, \bullet, vc)$, and $(\bullet, vc)$.

Algorithm 4: Part II of self-stabilizing always-terminating snapshot; p_i's code.

110 **operation** write(v) **begin**
111 $writePending \leftarrow v$; **wait until** $(writePending = \perp)$; **return**();

112 **operation** snapshot() **begin**
113 $(sns, pndTsk[i]) \leftarrow (sns + 1, (sns, \perp, \perp))$;
114 **wait until** $(pndTsk[i].fnl \neq \perp)$; **return**$(pndTsk[i].fnl)$;

115 **function** baseWrite(v) **begin**
116 $\{ts \leftarrow ts + 1; reg[i] \leftarrow (ts, v)$; **let** $lReg := reg$;
117 **repeat**
118 broadcast WRITE($lReg$); merge(Rec) **where** Rec is the received reg arrays$\}$
119 **until** WRITEack($regJ \geq lReg$) *received from a majority*;

120 **function** baseSnapshot(S) **begin**
121 **repeat**
122 $ssn \leftarrow ssn + 1$; **let** $prev := reg$;
123 **repeat**
124 broadcast SNAPSHOT($(S \cap \Delta), reg, ssn$);
125 **until** $(S \cap \Delta) = \emptyset$ or majority of (SNAPSHOTack($\bullet, ssnJ = ssn$) arrived);
126 merge(Rec) **where** Rec is the set of reg arrays received at line 125;
127 **if** $prev = reg \wedge (S \cap \Delta) \neq \emptyset$ **then**
128 safeReg($\{(k, pndTsk[k].sns, prev) : (k, s, \bullet) \in (S \cap \Delta)\}$)
129 **else if** $((i, \bullet) \in (S \cap \Delta)) \wedge (pndTsk[i].vc = \perp)$ **then** $pndTsk[i].vc \leftarrow$ VC;
130 **until** $(S \cap \Delta) = \emptyset \vee ((S \cap \Delta) = (i, \bullet) \wedge pndTsk[i].sns > 0 \wedge pndTsk[i].fnl = \perp \wedge \neg exceeds(i))$;

131 **upon** SAVE(AJ) **arrival from** p_j **begin**
132 **foreach** $(k, s, r) \in AJ$ **do**
133 **if** $pndTsk[k] = (s, \bullet, \perp)$ **then** $pndTsk[k].fnl \leftarrow r$;
134 **else if** $pndTsk[k].sns < s$ **then** $pndTsk[k] \leftarrow (s, \perp, r)$;
135 **send** SAVEack($\{(k, s) : (k, s, \bullet) \in AJ\}$) **to** p_j;

136 **upon** GOSSIP($regJ, snsJ$) **arrival from** p_j **begin**
137 $reg[i] \leftarrow \max\{reg[i], regJ\}; (ts, sns) \leftarrow (\max\{ts, reg[i].ts\}, \max\{sns, snsJ\})$;

138 **upon** WRITE($regJ$) **arrival from** p_j **begin**
139 **for** $p_k \in P$ **do** $reg[k] \leftarrow \max_{\prec}(reg[k], regJ[k])$;
140 **send** WRITEack(reg) **to** p_j (*concurent baseSnapshot() calls need piggybacking with line 124's message*);

141 **upon** SNAPSHOT($SJ, regJ, ssnJ$) **arrival from** p_j **begin**
142 **for** $p_k \in P$ **do** $reg[k] \leftarrow \max_{\prec}(reg[k], regJ[k])$;
143 **foreach** $(sInd, sn, vc) \in SJ : pndTsk[sInd].sns < sn \vee pndTsk[sInd] = (sn, \perp, \perp)$ **do**
144 $pndTsk[sInd] \leftarrow (sn, vc, \perp)$
145 **let** $A := \{(k, pndTsk[k].sns, pndTsk[k].fnl) : (k, sInd, \bullet) \in SJ : (pndTsk[k].fnl \neq \perp \vee sInd < pndTsk[k].sns)\}$ /* prepare the results, before sending (line 147);
146 **send** SNAPSHOTack($reg, ssnJ$) **to** p_j;
147 **if** $A \neq \emptyset$ **then send** SAVE(A) **to** p_j (* piggyback these messages *);

The macro VC (line 94) aggregates the timestamp values in $reg[k]$ numerically. It returns a vector $[ts_k]_{p_k \in P}$ where the kth entry holds zero whenever $reg[k] = \perp$. Otherwise, $ts_k = x$ is returned where x is the 2nd element in the pair $reg[k] = (\bullet, x)$. The macro $exceeds(k)$ lets baseSnapshot(S) decide whether the algorithm should be in the helping or the non-helping mode by using the vector clocks to enforce a bound on concurrent writes.

For a given $p_k \in P$, the macro $exceeds(k)$ (line 95) returns true whenever the number of write operations that have occurred since p_k initiated its snapshot reaches the bound δ. Formally, the quantity $\sum_{\ell=1}^{n}(\text{VC}[\ell] - pndTsk[k].vc[\ell])$ counts how many timestamps have advanced across all writers since the snapshot baseline was recorded in $pndTsk[k].vc$ (cf. line 129). Thus, $exceeds(k)$ detects when too many concurrent writes have taken place for the snapshot to complete without assistance from the proposed helping scheme.

For a given $p_k \in P$, the macro $\Delta(k)$ (line 96) returns a set of triples of the form $(k, pndTsk[k].sns, pndTsk[k].vc)$ representing the snapshot task initiated by p_k. An entry is included when the task has not yet produced a final value ($pndTsk[k].fnl = \perp$) and either (i) $\delta = 0$ and the snapshot index is positive, or (ii) the corresponding vector clock, $pndTsk[k].vc$, is defined and $exceeds(k)$ holds.

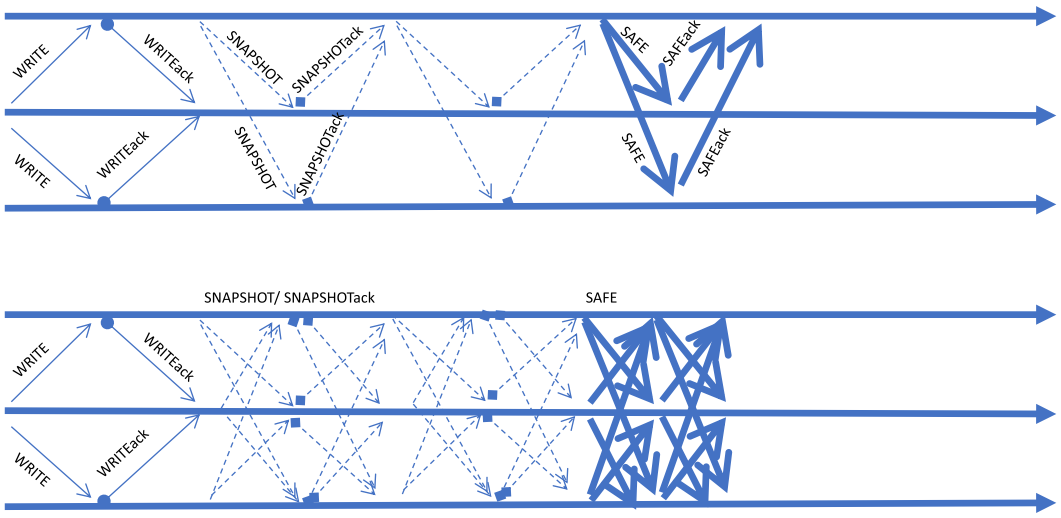


Fig. 4. The upper drawing depicts an example execution of Algorithm 4 in a setting equivalent to the one shown in the upper drawing of Fig. 2, where a single snapshot operation occurs. The lower drawing illustrates the case where all nodes invoke snapshot operations concurrently.

The macro Δ (line 97) is the union over all $\Delta(k)$ for $p_k \in P$, together with the analogous triple for p_i itself when its snapshot task is pending. Hence, $\Delta(k)$ and Δ return aggregated information regarding pending snapshot tasks invoked by p_k any node, respectively. Note that Δ always includes the most recent pending snapshot task of p_i, if any.

As explained in line 145, the set A holds a collection of triplets (k, s, vc) that identify the snapshot tasks whose results the process is preparing to finalize. Each triplet specifies the task owner k, the snapshot index s, and the vector-clock baseline vc associated with that task. Algorithm 4 uses A to assemble the values required for generating the final snapshot reply. Specifically, the macro safeReg(A) (line 98) repeatedly broadcasts SAVE(A) until a majority of matching SAVEack(AJ) messages is received, where $AJ = \{(k, s) : (k, s, \bullet) \in A\}$.

The macro merge(Rec) (line 99) aggregates all timestamps fields by updating ts with $\max(\{ts, reg[i].ts\} \cup \{r[i].ts \mid r \in Rec\})$. Then, for each $p_k \in P$, it assgins $reg[k]$ with $\max(\{reg[k]\} \cup \{r[k] \mid r \in Rec\})$.

5.4. The do-forever loop

The lines 101 to 109 are repeatedly executed to maintain the system state free of stale information.

- *Delete Out-of-Sync SNAPSHOTack Messages* (line 101): For every snapshot sequence number (ssn') that is not the current one (ssn), delete the SNAPSHOTack messages to clean up outdated acknowledgments.
- *Update Timestamps and Snapshot Numbers* (line 102): Update the local timestamp (ts) and snapshot number (sns) to be the maximum of their current values and the values from the local registry ($reg[i].ts$), and respectivly, pending snapshot entries ($pndTask[i].sns$).
- *Fix Illogical Vector Clocks* (line 103): For each node p_k, if its pending snapshot vector clock ($pndTsk[k].vc$) does not logically precede (line 59) the current vector clock (VC), reset it to the default state ($\perp$).
- *Reset Corrupted Snapshot Entries* (line 105): If the local snapshot number (sns) is different from the pending snapshot's number ($pndTsk[i].sns$), reset the local pending snapshot entry to default values.
- *Send Gossip Messages* (line 106): Send GOSSIP messages containing the local registry and snapshot numbers to all other processes (p_k), helping to synchronize and update their states.
- *Perform Pending Writes* (line 108): If there is a pending write operation ($writePending \neq \perp$), execute it using $baseWrite(writePending)$ and then clear the $writePending$ variable.
- *Perform Pending Snapshots* (line 109): If there are any snapshots pending due to at least δ concurrent writes ($\Delta \neq \emptyset$), execute the procedure $baseSnapshot(\Delta)$.

We clarify that the synchronization between writes and snapshots (lines 108 and 109) starts with a write, if there is any pending task (line 108), before running its own snapshot if there is any pending. Additionally, it runs any snapshot initiated by others for which p_i observed at least δ writes occurring concurrently (line 109).

5.5. The write() and baseWrite() functions

Node p_i cannot start immediately a write; it permits concurrent writes by storing v and a unique index in $writePending_i$ (line 110). The write runs as a background task (line 108) using baseWrite() (line 115 to 118).

5.6. The baseSnapshot() function and the SNAPSHOT message

Algorithm 4 maintains the state of every snapshot task in the array $pndTsk$. The entry $pndTsk_i[k] = (sns, vc, fnl)$ includes: (i) the index sns of the most recent snapshot that $p_k \in P$ 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 94) and (iii) the final result fnl of the snapshot (or $\perp$, in case it is still running).

We clarify that the vector clock, $pndTsk_i[k].vc$, serves as a timestamp baseline for detecting how many concurrent write operations have occurred since process p_k initiated its snapshot task. When p_k starts a snapshot, the vector clock records the timestamps of all registers at that moment (line 129). Later, the test *exceeds*(k) (line 95) compares this baseline against the current vector clock VC (line 94), and the difference VC $-$ $pndTsk[k].vc$ counts how many writers have advanced their timestamps. If this increase reaches δ, the system concludes that the snapshot cannot complete without help, and therefore invokes the helping mechanism (line 109). Thus, the vector clock is essential for bounding the number of concurrent writes that may delay a snapshot and for ensuring always-termination.

The baseSnapshot() function takes as a parameter the set S, which is initialized to $\Delta \neq \emptyset$ (line 97). It includes an outer loop part (lines 122 and 130), an inner loop part (lines 124 to 125), and a result update part (lines 127 to 129). The outer loop increments the snapshot index, ssn (line 122), so that it can consider a new query attempt by the inner loop.

The outer loop ends when (i) there are no more pending snapshots that this call to baseSnapshot(S) needs to handle, *i.e.,* $(S \cap \Delta) = \emptyset$ since Δ represents the current set of pending snapshots, or (ii) the only pending snapshot for the current invocation of baseSnapshot() is the one of p_i and p_i has not observed at least δ concurrent writes. The inner loop broadcasts SNAPSHOT messages, which includes all the pending 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. Recall that S is initialed to Δ and as the outer loop part (lines 122 and 130) progresses, the set Δ might lose elements due to the completion of tasks. Thus, if $\Delta = \emptyset$, also $(S \cap \Delta) = \emptyset$. In other words, the inner loop ends when acknowledgments are received from a majority and the received values are merged (line 126). The results are updated by writing to a variable that is replicated via patterns similar to the ones used to emulate a safe register (line 127) whenever $prev = reg$. In case the results do not allow p_i to complete its snapshot task (line 129), Algorithm 4 uses the query results for storing the timestamps in the field vs. As we explain next, this allows us to balance a trade-off between snapshot write latencies.

5.7. Using δ for balancing the trade-off between snapshot and write latencies

For the case $\delta = 0$, the set Δ (line 97) includes all the nodes for which there is no stored result, *i.e.,* $pndTsk[k].fnl = \perp$. Thus, no snapshot tasks are ever deferred, as in [16, Algorithm 2]. The case of $\delta > 0$ uses the fact that Algorithm 4 reads the vector clock value of reg_i and stores it in $pndTsk[i].vc$ (line 129) once it had completed at least one iteration of the repeat-until loop (lines 124 and 125), *i.e.,* the sampling of the vector clock is an event that occurred not before the start of p_i's snapshot that has the index of $pndTsk[i].sns$.

5.8. The helping scheme

As long as $pndTsk[k].fnl \neq \perp$, p_k's task is considered active. For helping all active tasks, p_i samples the set of pending tasks (Δ) (line 109) before starting the inner repeat-until loop (lines 124 and 125). Node p_i broadcasts from the client-side the SNAPSHOT message, which includes the task information. The reception of this SNAPSHOT message on the server-side (lines 141 to 142), updates the local information (line 144) and prepares the reply (line 145) before sending it to the client-side (line 147). If the receiver notices that it has the result of an ongoing task, it sends that result to the requestor (line 147).

5.9. The safeReg() function

This function stores the snapshot result r in a safe register. It does so by broadcasting the client-side message SAVE (line 98). Upon SAVE's arrival, the receiver update its state and replies with a SAVEack message to the client-side, who is waiting for a majority of such replies (line 98).

5.10. Correctness

We prove the recovery (Theorem thm:terminatingAlgConvergence), termination (Theorem thm:livenessAll), and linearization (Lemma 5) of Algorithm 4.

Definition 2.

Towards the definition of Algorithm 4's consistent system states and executions, we bring the following consistency invariants.

(i) Let c be a state in which ts_i is greater than or equal to any p_i's timestamps in the variables and fields related to ts. We say that the ts' timestamps are consistent in c.

(ii) Let c be a state in which ssn_i is greater than or equal to any p_i's sequence numbers in the variables and fields related to ssn. We say that the ssn's sequence numbers are consistent in c.

(iii) Let c be a state in which sns_i is greater than or equal to any p_i's operation index in the variables and fields related to sns. Also, $\forall p_i \in P : sns_i = pndTsk_i[i].sns$ and $\forall p_i, p_j \in P : pndTsk_j[i].sns \leq pndTsk_i[i].sns$. We say that the sns's operation indexes are consistent in c.

(iv) Let c be a state in which $\forall p_i, p_k \in P : pndTsk_i[k].vc \preceq \text{VC}_i$ holds, where VC_i is defined in line 94. We say that the vector clock values are consistent in c.

We say that state c is consistent if it is consistent with respect to invariants (i) to (iv). Let R be an Algorithm 4's execution in which all 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 3 (Algorithm 4's convergence).
Let R be an Algorithm 4's execution. Eventually the system reaches a consistent suffix, R', of R.

Proof. Note that Lemmas 1 and 2 imply invariants (i), and resp., (ii) of Definition 2 also for the case of Algorithm 4, because they use the same code lines for asserting these invariants. For the convenience of the reader, Lemmas 7 and 8 in Section A of the Appendix present the proof details.

Invariant (iv) is implied by the fact that $p_i \in P$ executes line 103 at least once and the fact that VC_i (defined in line 94) is assigned to $pndTsk_i[k].vc$ in line 129. Note that these are the only lines of code that assign values to $pndTsk_i[k].vc$ and that the value of every entry in VC_i is not decreasing (cf. Claim 1).

Eventually, R reaches a suffix R', such that every received message during R' was sent during R. We can also require that for every received reply message in R', we have that its associated request message was sent during R. Thus, R' is consistent. $\square_{Theorem\ 3}$

The proof of Theorem 4 considers both complete and not complete snapshots. We say that a snapshot() operation is *complete* if it starts due to a step a_i in which p_i calls snapshot() (line 112) and its index, s, is greater than any of p_i's snapshot indices in the state that appears immediately before a_i, where s is the value of sns_i stored in $pndTsk_i[i].sns$ (line 113).

We clarify that the recovery from transient faults of the proposed solution does not depend on write operations, since some value, perhaps $\perp$, is always returned. This is mainly because gossip messages help to recover corrupted values and replace them with $reg_i[i]$. Nevertheless, the update (and recovery) of the value stored in $reg_i[i]$ requires invocating a write operation after the last occurrence of a transient fault.

In detail, the proof of Theorem thm:livenessAll considers the case in which R's first step includes a complete call to snapshot() in line 112 as well as the case in which R's first step does not include such a call, but p_i's state partially encodes such a call, say, due to a transient fault. Therefore, Theorem thm:livenessAll's refers to the R's second system state in which the conditions $pndTsk_i[i] = (s, \bullet, \perp)$ and $s > 0$ hold. Note that when $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$ holds, the system has reached a state in which p_i can locally retrieve a value that is associated with a snapshot operation under the sequence number s.

The proof of Theorem thm:livenessAll considers the term *message round trip exchange* (RTE), which refers to the period that starts in a state that appears immediately before a step in which a request message is sent from node p_i to p_j and ends immediately after the step in which p_i receives p_j's response to its request. We note that each RTE has the cost of two in Lamport's happened-before relation (HBR) [29].

Theorem 4 (Algorithm 4's termination).
Let R be an Algorithm 4's consistent execution. Suppose $\exists p_i \in P$, such that in R's second system state, it holds that $pndTsk_i[i] = (s, \bullet, \perp)$ and $s > 0$.

Eventually the system reaches $c \in R$ in which $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$. Also, Algorithm 4's costs are in $\mathcal{O}(n)$ RTEs per write() *and in $\mathcal{O}(n + \delta)$ RTEs per* snapshot().

Theorem thm:livenessAll's proof is implied by Lemmas 3–6, which can be read sequentially until Theorem thm:livenessAll's end of proof symbol "$\square_{Theorem\ 4}$".

Proof. Lemmas 3 and 4 use $\mathcal{S}_i()$, which we define next, as well as Definition 3. Whenever p_i's program counter is outside of baseSnapshot(), the $\mathcal{S}_i()$ function returns Δ_i. Otherwise, the function returns $(\mathcal{S}_i \cap \Delta_i)$.

Definition 3.
Let R be an Algorithm 4's execution and $p_i \in P$ a node that (does not crash and) invokes snapshot task, $T(i, s)$, such that $pndTsk_i[i] = (s, \bullet, \perp)$ and $s > 0$ in R's starting state. Also, $D \subseteq P$ is the *diffusion set* that includes at least one majority $M \subseteq P : |M| > |P|/2$ and any node that invokes write concurrently with $T(i, s)$.

(i) Suppose there is a diffusion set, D, such that eventually, the system reaches $c \in R$ in which $(i, \bullet) \in \mathcal{S}_i()$ and $pndTsk_j[i] = (s, \bullet, \perp)$, for any $p_j \in D$. In this case, we say that R has diffused task $T(i, s)$ by c.

(ii) Suppose there is a diffusion set, D, such that eventually the system reaches $c' \in R$ in which for every $p_j \in D$, it holds that $(i, \bullet) \in \mathcal{S}_i()$, $pndTsk_j[i] = (s, y, \bullet) : y \neq \perp$, and $y \preceq VC_j$. Here, VC_j refers to the vector-clock array defined in line 94 and used in line 129 of Algorithm 4, which stores p_j's local view of the system-wide timestamp maxima. In this case, we say that R triggers the helping-scheme for task $T(i, s)$ in c'.

(iii) Suppose there is a diffusion set, D, such that eventually, the system reaches $c'' \in R$ in which for any $p_j \in D$, it holds that $(i, \bullet) \in \mathcal{S}_i()$ and $pndTsk_j[i] \in \Delta_j$. Then we say that R does not interrupt task $T(i, s)$ in c''.

Note that if execution R does not interrupt task $T(i, s)$ in c'' (Definition 3, item (iii)), then within a finite time after c'' the task $T(i, s)$ is completed. The high-level proof idea can be taken from Algorithm 3; we provide the proof details in Lemma 6, see arguments (5) to (8). Towards this proof of operation completion, Lemmas 3 and 4 consider three completion cases, see invariants (i) to (iii) in

Lemma 3. Invariant (i) considers the case of non-interruption, which we discussed above. Invariant (ii) considers a case that Lemma 4 shows to eventually result in Invariant (iii). Invariant (iii) considers a case in which the processor who invoked the snapshot task received the result, *i.e.,* task completion.

Lemma 3.

Eventually the system reaches $c \in R$ in which either: (i) R does not interrupt task $T(i,s)$ in c (Definition 3),
(ii) any $M \subseteq P : |M| > |P|/2$ includes at least one $p_j \in M : pndTsk_j[i] = (s, \bullet, x) : x \neq \perp$, or (iii) $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$.

Proof.

Lemma 3's proof is by contradiction of the above assumption. The contradiction is shown in Claim 10, which uses Claims 8 and 9. The proof can be read sequentially until Lemma 3's end of proof symbol "$\square_{Lemma\ 3}$".

Towards a proof by contradiction, Claim 8 considers the suffix R', to denote the maximal suffix of R in which none of the invariants (i)-(iii) of Lemma 3 hold. Although R is assumed to be consistent (Theorem 4), the purpose of defining R' is to isolate the longest portion of the run in which all three invariants fail simultaneously. The subsequent arguments in the proof of Claim 8 derive a contradiction to the existence of such a suffix.

Claim 8.

R' cannot contain any step in which p_i evaluates the condition of the if-statement in line 127, because such an evaluation would immediately satisfy invariant (ii), contradicting the assumption that none of the invariants hold throughout R'.

Proof. Although the definition R' implies that it cannot contain a step in which the condition of line 127 is evaluated, Arguments (1)—(3) establish that p_i must eventually reach such a step. This ensures that the assumption that none of the invariants hold throughout R' leads to a contradiction, which is demonstrated in Argument (4). Arguments (1) to (3) show p_i calls $\mathsf{safeReg}(\{(k, pndTsk[k].sns, prev) : (k, s, \bullet) \in S\})$ in $a_i \in R'$. Argument (4) shows the needed contradiction.

Argument (1): baseWrite() *ends eventually.*

A call to baseWrite(v) starts with p_i incrementing ts_i and storing it in $reg_i[i]$ (line 116). The value of ts_i is unique (in the state that immediately follows) w.r.t. ts's variables and fields related to p_i (Theorem 3). The repeat-until loop (line 119) ends eventually since R is consistent and the existence of a non-faulty majority.

Argument (2): *the repeat-until loop (lines 122 to 125) ends eventually.*

The call to baseSnapshot(S_i) : $(i, \bullet) \in S_i$ starts with p_i incrementing ssn_i (line 122) to a value that is unique w.r.t. ssn's variables and fields related to p_i (Theorem 3). The repeat-until loop (line 125) ends eventually since the existence of a non-faulty majority, the consistency of R, and the uniqueness of ssn_i (or the fact that $S_i() = \emptyset$, which implies the lemma since then invariant (iii) holds).

Argument (3): *eventually, $p_i \in P$ executes* baseSnapshot$_i(S_i) : (i, \bullet) \in S_i$ *and p_i evaluates the if-statement in line 127.*

Since invariant (iii) of Lemma 3 does not hold, we know that $(i, \bullet) \in S_i$ whenever p_i executes baseSnapshot$_i(S_i)$ or line 109. The latter occurs eventually (Argument (1)) and calls to baseSnapshot$_i()$. Thus, the execution of line 127 is implied since the repeat-until loop (lines 122 to 126) ends (Argument (2)).

Argument (4): *invariant (ii) of Lemma 3 holds.*

The function $\mathsf{safeReg}_i()$ (line 128) repeatedly broadcasts SAVE until p_i receives matching SAVEack messages from a majority. Theorem 3 and the assumption that R' is consistent imply that every received SAVEack message can be associated with a matching SAVE message that was indeed sent during R. We show that this majority of acknowledgments from $p_j \in P$ implies that invariant (ii) holds (intersection of majority groups). By lines 131 to 135, SAVE's arrival to $p_j \in P$ assures $pndTsk_j[i].fnl \neq \perp$ before sending SAVEack back to p_i (Theorem 3 and R's consistency). $\square_{Claim\ 8}$

Claim 9.

Invariant (ii) of Definition 3 holds.

Proof. We show the claim holds for a majority via the cases of $j = i$ and $j \neq i$. Then, we show that the claim holds for any node that performs an unbounded number of writes concurrently with p_i's snapshot.

The j = i case.

Eventually, p_i calls baseSnapshot(S_i) : $(i, \bullet) \in S_i$ (line 109) (Argument (3), Claim 8). This, Claim 8 and its Argument (2) imply the execution of line 129 in every call for baseSnapshot(S_i). Thus, the case of $j = i$ is true.

The j ≠ i case.

By the arguments of the case of $j = i$, p_i executes lines 122 and 124 in which p_i broadcasts $(i, pndTsk_i[i].sns, pndTsk_i[i].vc) \in S'$ via SNAPSHOT($S', \bullet$) messages. This repeats until a majority acknowledges its reception, which we know to occur within a finite time due to the theorem assumption that R is consistent and that a non-failing majority exists. Note, $pndTsk_i[i].vc \neq \perp$ (by the above case of $j = i$). Also, once p_j receives the SNAPSHOT() message, $pndTsk_j[i].vc \neq \perp$ holds (line 144). And, $y \preceq VC_j$ holds since otherwise, R is not consistent.

The previous two cases fix the writer index j. We next consider not another index case, but the concurrency scenario in which some process p_k performs an unbounded number of WRITE operations concurrently with p_i's snapshot. This scenario must be addressed for Invariant (ii) of Definition 3

The case of $p_k \in P$ that performs any number of writes concurrently with p_i's snapshot. The above arguments for the case of $j \neq i$ can be repeated as long as invariant (iii) of Lemma 3 does not hold. This is true also for the case of p_k since the acknowledgments

to its WRITE messages are piggybacked with SNAPSHOT messages, cf. the comment in line 140. Thus, the arrival of such a SNAPSHOT message to all $p_j \in P$ occurs eventually (or one of the lemma invariants holds). $\square_{Claim\ 9}$

Claims 8 to 10 establish that, under the assumption that none of the invariants of Lemma 3 ever holds in the suffix R', process p_i cannot execute the outer loop of baseSnapshot() more than δ times. However, by definition, R' is an infinite suffix of the execution in which p_i continues to invoke baseSnapshot() indefinitely. This contradiction shows that our assumption about R' is impossible; hence some of the invariants (i)–(iii) of Lemma 3 must eventually hold.

Claim 10.

Let $c' \in R'$ be the state that appears in Definition 3, which Claim 9 showed to exist. Let x denote the number of iterations of Algorithm 4's outer loop in baseSnapshot() *(lines 122–130) that process p_i performs in R' starting from c'. Assuming that none of the invariants (i)–(iii) of Lemma 3 holds throughout R', the value of x is finite and furthermore satisfies $x \leq \delta$.*

Proof. Arguments (1) to (3) show $x \leq \delta$ and between c'' and c''' there are a finite number of steps.

Argument (1): *whenever p_i iterates over the outer loop in* baseSnapshot() *(lines 122 and 130), p_i takes a step in which it tests the if-statement condition at line 127 and that condition does* not *hold.*

Eventually, p_i calls baseSnapshot(S_i) : $(i, \bullet) \in S_i$ (line 109) at least once (Argument (3) in Claim 8). By Claim 8, that call includes the execution of line 127 in which the if-statement condition does not hold (because then Argument (4) in Claim 8 implies that invariant (ii) holds).

Argument (2): *suppose there are at least x consecutive and complete iterations of p_i's outer loop in* baseSnapshot() *(lines 122 and 130) between c' and c'' in which the if-statement condition at line 127 does not hold. There are at least x writes that run concurrently with the snapshot that has the index of s.*

The only way that the if-statement condition in line 127 does not hold in a repeated manner is by repeated changes of ts field values in reg_i during the different runs of lines 122 to 126. Such changes are due to increments of $ts_j : p_j \in P$ (line 110) at write().

Argument (3): *$\exists x' \leq \delta$ for which $(i, \bullet) \in S_i()$, where x' is the number of consecutive and complete iterations of the outer loop in* baseSnapshot() *(lines 122 and 130) between c' and c'' in which the if-statement condition at line 127 does not hold.*

Argument (2) implies that the number of iterations continues to grow. Argument (2) and Claim 1 imply that during every such iteration there are increments of at least one of the summation $\sum_{\ell \in \{1,\ldots,n\}} \mathrm{VC}_i[\ell] - pndTsk_i[i].vc[\ell]$ until that summation is at least δ. Recall that $pndTsk[i].vc \neq \perp$ (Claim 9) and $pndTsk_i[i].fnl = \perp$ (lemma invariants do hold). Thus, $(i, \bullet) \in S_i()$ holds (line 97, for $k = i$).

Argument (4): *suppose that p_i has taken at least x' iterations of the outer loop in* baseSnapshot() *(lines 122 and 130) after state c' (which is defined in Claim 9). After these x' iterations, suppose the system has reached c'' in which $(i, \bullet) \in S_i()$, as in Argument (3). Eventually after c'', one of the lemma invariants holds in state c'''.*

In order to show that there is a diffusion set, D, that does not interrupt $(i, \bullet) \in S_i()$ in $c''' \in R$, we first show that a majority set $M \subseteq P : |M| > |P|/2$ exists before considering any p_j that executes an unbounded number of writes concurrently with task $(i, \bullet) \in S_i()$.

Eventually after c'' (which Argument (3) defines), it holds that reg_j's ts fields are not smaller than the ones of reg_i's ts fields in c''. This is because in every iteration of the outer loop in baseSnapshot() (lines 122 and 130), p_i broadcasts reg_i (line 124). These SNAPSHOT messages arrive eventually at least a majority, M, of non-failing nodes $p_j \in M$ and upon their arrival p_j updates reg_j (lines 141 to 142). The rest of the proof shows that $(i, \bullet) \in S_j()$ holds (line 97 for $k = i$); the reasons for that are similar to the ones that appear in Argument (3)'s proof.

For the case of p_j that executes any number of writes concurrently with task $(i, \bullet) \in S_i()$, one can repeat the above reasoning due to the fact that the acknowledgments of WRITE are piggybacked with SNAPSHOT, cf. the comment in line 140. $\square_{Claim\ 10}$

We end the lemma's proof noting that, other than in Claim 10, all interaction between p_i and the other diffusion set members $p_j \in D$ are based on a constant number of round-trip interactions.

$\square_{Lemma\ 3}$

Lemma 4 shows that starting from a system state in which Invariant (ii) of Lemma 3 holds means that the system eventually reaches a state in which Invariant (iii) of Lemma 3 holds.

Lemma 4.

Let R be an Algorithm 4's consistent execution (Definition 2) and $p_i \in P$. Also, suppose
in any state of R, it holds that $pndTsk_i[i] = (s, \bullet, \perp)$, $s > 0$.
Eventually, the system reaches $c \in R$ in which $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$.

Proof. The proof is implied by Claims 11 and 12. The proof can be read sequentially until Lemma 4's end of proof symbol "$\square_{Lemma\ 4}$".

Claim 11 relies on SNAPSHOTack messages, which carry acknowledgment information (line 146), while the snapshot results are propagated via separate SAVE(A) messages (line 147) generated in the same handler. In particular, whenever a process sends a SNAPSHOTack message, it also sends a corresponding SAVE(A) message if the set A is non-empty. In the proof of Claim 11, we explicitly distinguish between acknowledgment messages and result propagation, which are handled separately via SNAPSHOTack and SAVE(A), respectively.

Claim 11.

Suppose $pndTsk_i[i].sns > 0$ holds in any state of R and that every majority $M \subseteq P$ with $|M| > |P|/2$ includes at least one $p_j \in M$ such that $pndTsk_j[i] = (s, \bullet, x)$ for some $x \neq \perp$. Eventually, the system reaches a state $c \in R$ in which $pndTsk_i[i] = (s, \bullet, x)$ for some $x \neq \perp$.

Proof. Towards a contradiction, suppose R has no suffix R' such that $pndTsk_i[i] = (s, \bullet, x)$ for some $x \neq \perp$ holds in the starting state of R'. Arguments (1) and (2) show the needed contradiction. Recall that by Argument (3) in Claim 8, every iteration of the do-forever loop during R' includes a call to baseSnapshot(S_i) such that $(i, \bullet) \in S_i$ (line 109).

Argument (1): *eventually, a majority of matching* SNAPSHOTack$(reg, ssnJ)$ *messages arrives, and for at least one such acknowledging node, say* p_j*, the handling of the* SNAPSHOT *request also triggers a* SAVE(A) *message with* $(i, s, x) \in A$ *for some* $x \neq \perp$.

By Argument (2) in Claim 8, the repeat-until loop in lines 122 to 125 ends. Therefore, eventually a majority of processes respond with matching SNAPSHOTack$(reg, ssnJ)$ messages (line 146) to the repeated broadcasts of SNAPSHOT$(\bullet, ssn_i)$ (line 124).

By the claim assumption and the intersection property of majority sets, at least one of these acknowledging processes, say p_j, satisfies $pndTsk_j[i] = (s, \bullet, x)$ for some $x \neq \perp$. When p_j handles one of the corresponding SNAPSHOT request, it constructs the set A in line 145. Since $pndTsk_j[i] = (s, \bullet, x)$ with $x \neq \perp$, it follows from the definition of A that $(i, s, x) \in A$. Hence $A \neq \emptyset$, and so p_j sends SAVE(A) to p_i in line 147.

Argument (2): *eventually,* $pndTsk_i[i].fnl \neq \perp$ *holds.* By Argument (1), there exists a process p_k that acknowledges the SNAPSHOT request and sends a corresponding SAVE(A) message with $(i, s, x) \in A$ for some $x \neq \perp$. By the communication assumptions and fairness, this SAVE(A) message eventually arrives at p_i (line 131). Upon arrival of SAVE(A) at p_i, line 133 applies to the triple $(i, s, x) \in A$, and therefore updates $pndTsk_i[i].fnl$ to x. Thus, eventually $pndTsk_i[i].fnl \neq \perp$, *i.e.,* $pndTsk_i[i] = (s, \bullet, x)$ for some $x \neq \perp$, contradicting the assumption on R'. $\square_{Claim\ 11}$

Claim 12.

Suppose R *does not interrupt task* $(i, \bullet) \in S_i()$ *in* R*'s starting state (Definition 3). Eventually,* $c \in R$ *in which* $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$.

Proof. The proof is by arguments (1) to (3).

Argument (1): *a call to* safeReg$_k()$ *implies the claim eventually.*

Suppose the if-statement condition in line 127 holds after the execution of lines 124 to 126. For $i = k$, the call to safeReg$_i(\{(\bullet, r_i) : r_i \neq \perp\})$ causes p_i to send SAVE$(\{(\bullet, r_i) : r_i \neq \perp\})$ to itself and the reception of this message assigns $r_i \neq \perp$ to $pndTsk_i[i].fnl$ (line 133). For $k \neq i$, Claim 8's Argument (4) implies that the call to safeReg$_k()$ creates a majority set, $M \subseteq P$, that stores in $pndTsk_j[i].fnl : p_j \in M$ the result of $(i, \bullet) \in S_j()$. Also, Claim 11's Argument (2) implies the claim.

Argument (2): *eventually, there are no active writes or neither* $(i, \bullet) \in S_j()$ *nor* $(i, \bullet) \in S_i()$ *hold.*

During R, any node that calls write(), returns eventually (Claim 8's Argument (1) and R is non-interruptive). Specifically, by Definition 3, for any p_j that executes any number of writes concurrently with task $(i, \bullet) \in S_j()$, eventually $pndTsk_j[i] \in \Delta_j$. Then, p_j calls baseSnapshot$_j(S_j) : S_j = \Delta_j$. The exit condition of the outer repeat-until loop (line 130) implies that p_j does not return from baseSnapshot$_j(S_j) : S_j = \Delta_j$ as long as $(i, \bullet) \in S_j()$. If the latter does not hold, then neither $(i, \bullet) \in S_i()$ holds (due to Claims 9 and 10).

Argument (3): p_i*'s snapshot,* $(i, \bullet) \in S_i()$*, ends.*

Towards a contradiction, let R' be a suffix of R, in which p_i's snapshot does not end. *I.e.,* in every state of R', $pndTsk_i[i] = (s, \bullet, \perp)$ holds.

By Claim 8's Argument (2), the loop in lines 124 to 126 ends. By line 125, this happens when (i) $(S_i \cap \Delta_i) \neq \emptyset$ or (ii) majority of matching SNAPSHOTack arrived. For case (i), the proof is done since $(i, \bullet)$ leaves the set $(S_i \cap \Delta_i)$ only when $pndTsk_i[i] = (s, \bullet, x) : x \neq \perp$ (line 133). For case (ii), Argument (2) implies that the if-statement condition in line 127 holds, and the contradiction is shown. $\square_{Claim\ 12}$

We complete the lemma proof by saying that all interaction between p_i and $p_j \in D$ are based on a constant number of round-trip interactions.

$\square_{Lemma\ 4}$

Lemma 5 (Algorithm 4's linearization).

Algorithm 4 satisfies the linearizable history requirement.

Proof. We note that the baseWrite() functions in Algorithms 2 and 4 are identical. Moreover, Algorithm 4's lines 122 to 126 differ only in the following manner: (i) the dissemination of the operation tasks is done outside of Algorithm 2's broadcasting of SNAPSHOT() appears inside of Algorithm 4's lines 122, and (ii) Algorithm 2 considers one snapshot at a time whereas Algorithm 4 considers many snapshots.

The proof is based on observing that the definition of linearizability (Section 2.2) allows concurrent snapshots to have the same result (as long as they each individually respect all the other constraints that appear in the definition of linearizability). Also, the linearizability property does not depend on the way in which the snapshot tasks (and their results) are disseminated.

(Indeed, the linearizability proof of Delporte-Gallet et al. [16, Lemma 7] does not consider the way in which the snapshot tasks, and their results, are disseminated when selecting linearization points. These linearization points are selected according to some partition, defined in [16, Lemma 7]. The proof there explicitly allows the same partition to include more than one snapshot result. $\square_{Lemma\ 5}$

Lemma 6. *Algorithm 4's costs are* $4n + 18$ *HBRs per* write() *and* $8n + 2\delta + 34$ *HBRs per* snapshot().

Proof. The lemma's proof is given in Claims 14 and 15. They consider a legal execution of Algorithm 4 and use claim 13. The proof can be read sequentially until Lemma 6's end of proof symbol "$\square_{Lemma\ 6}$".

Claim 13. *Suppose* $c_{calls} \in R$ *appears immediately before* p_i*'s calls to* $baseSnapshot_i(S_i)$*. Within* $(2n + 8)$ *RTEs, the system reaches* $c_{returns} \in R$ *that appears immediately after* p_i *returns from this call.*

Proof. We sum up the RTEs in the longest happened-before relation between c_{calls} and $c_{returns}$.

Argument (1) *p_i observes the tasks in $S_i \cap \Delta_i$ to be concurrent with at least δ writes.* Suppose p_i has observed all tasks in $S_i \cap \Delta_i$ to be concurrent with at least δ writes. In order to not lose generality, Argument (10) considers the complementary case. Thus, p_i does not return until all tasks in $S_i \cap \Delta_i$ have a result due to the exit condition in line 130.

Argument (2) *Only fresh tasks are stored in $pndTsk[]$.* Until Argument (9) in the proof, we assume $pndTsk[]$ does not store a result for any task in $S_i \cap \Delta_i$. There we also explain why generality is not lost.

Argument (3) *A majority observes the tasks in $S_i \cap \Delta_i$ to be concurrent with at least δ writes.* Since p_i called $\text{baseSnapshot}_i(S_i)$, it broadcast $\text{SNAPSHOT}(S_i \cap \Delta_i, reg_i, \bullet)$ (line 124). Within an RTE from c_{calls}, p_i's first majority access ends and the system reaches $c_{S_i \cap \Delta_i \text{ majority}} \in R$, in which a majority stores the tasks in $S_i \cap \Delta_i$ in their local Δ-set. The reasons for this are as follows. Recall that p_i has observed that all tasks in S_i have been concurrent with at least δ writes (Argument (1)). Also, incoming reg values are merged with the local reg (line 142). Thus, all nodes in this majority merge the incoming reg with their local ones and they must include the tasks in $S_i \cap \Delta_i$ in their local Δ-sets.

Argument (4) *Node p_i performs as many snapshot majority accesses as possible.* After c_{calls}, p_i performs accesses (line 124). If $reg_i = prev_i$ (line 127), $baseSnapshot_i(S_i)$ returns. Otherwise, a concurrent write was observed by p_i. Since the proof aims to find the longest HBR, assume, whenever possible, $reg_i \neq prev_i$ after every access. Similarly, assume, whenever possible, no task in $S_i \cap \Delta_i$ ends, because that also allows $\text{baseSnapshot}_i(S_i)$ to return.

Argument (5) *All nodes observe the tasks in $S_i \cap \Delta_i$ to be concurrent with at least δ writes.* In order for $reg \neq prev$ to hold at the end of every access, at least one write has to be performed by another node concurrently with every access of p_i (line 124). Starting from $c_{S_i \cap \Delta_i \text{ majority}}$, this repeated interruption by a concurrent write can happen at most $n-1$ times, because each time p_j writes, at least one node in the majority that p_j receives replies for its WRITEack has that reply piggybacked with a $\text{SNAPSHOT}(S_i \cap \Delta_i, \bullet)$, cf. the comment in line 140.

Thus, when p_j returns from $\text{baseWrite}_j()$, node p_j immediately calls $\text{baseSnapshot}_j(S_j) : (S_i \cap \Delta_i) \subseteq S_j$. The reasons for this are as follows. Recall that Argument (3) showed that for a majority, the property that the node has observed the tasks in $S_i \cap \Delta_i$ to be concurrent with at least δ writes, holds. Due to that property, and because that property propagates due to the piggybacked $\text{SNAPSHOT}(S_i \cap \Delta_i, \bullet)$ (by a similar reason as in Argument (3)), $(S_i \cap \Delta_i) \subseteq S_j$ holds. Also, the call to $\text{baseSnapshot}_j(S_j)$ is indeed immediate, because the call appears in line 109 which is listed right after line 108 where baseWrite() returns. And, recall the assumption from Argument (4) that, whenever possible, no task in $S_i \cap \Delta_i$ completes.

Argument (6) *All non-failing and writing nodes call* baseSnapshot(). Let $c_{all\ called} \in R$ be a state where all non-failing and writing nodes p_j have called $\text{baseSnapshot}_j(S_j) : (S_i \cap \Delta_i) \subseteq S_j$. From Argument (5), during each majority access (line 124) by p_i, one other node might perform a write and then immediately calls $\text{baseSnapshot}_j(S_j) : (S_i \cap \Delta_i) \subseteq S_j$. Since each such majority access takes one RTE and since by Argument (5) there are at most $(n-1)$ such other nodes, the system reaches $c_{all\ called}$ within $(n-1)$ RTEs from $c_{S_i \cap \Delta_i \text{ majority}}$. It is significant that $c_{all\ called}$ is reached, and not only $c_{S_i \cap \Delta_i \text{ majority}}$, since the argument in Argument (7) requires that all nodes and not only a majority have called $\text{baseSnapshot}_j(S_j) : (S_i \cap \Delta_i) \subseteq S_j$.

Argument (7) *First call to* safeReg(). Within an RTE from $c_{all\ called}$, at least one p_k notices that $reg_k = prev_k$ (line 127) after its majority access (line 124) ends. This is because no node performs a write majority access at this point (they are only done in line 118). *I.e.,* no write can interrupt the snapshot. Since $reg_k = prev_k$, p_k calls $\text{safeReg}_k(S_k) : (S_i \cap \Delta_i) \subseteq S_k$ (line 128) and within one more RTE, a majority stores the results in $S_i \cap \Delta_i$ immediately before $c_{majority\ result} \in R$.

The arguments above can also consider the cases in which many nodes call safeReg() concurrently and even crash during safeReg(). This is since the system includes a majority of non-faulty nodes and the fact that any non-$\perp$ result stored in $pndTsk[].fnl$ is legitimate.

Argument (8) *p_i returns from* baseSnapshot(). Within one RTE from $c_{majority\ result}$, p_i receives a SAVE that is piggybacked with a SNAPSHOTack (cf. line 147) from a majority when p_i performs a majority access (line 125). Thus, for each task $T \in S_i \cap \Delta_i$, either p_i gets the result for T (fnl stores a non-$\perp$) or finds that sns is increased (meaning the node which created T has finished it and then begun a new snapshot), see line 145. In both cases, $S_i \cap \Delta_i$ becomes empty and p_i returns from $baseSnapshot_i(S_i)$ immediately before $c_{returns}$.

Argument (9) *Removing the assumption made in Argument (2).* Consider the case in which there is a non-failing $p_k \in P$ that stores in $pndTsk_k[j]$ a result for any task in $S_i \cap \Delta_i$ (without ever having the result stored in $pndTsk[j]$ by the task initiator or a majority). During legal executions, all transient faults are absent (Section 2), and thus the only way that this can happen is when the node that executes safeReg() crashes. By the assumption that there is a majority of non-faulty nodes and the last paragraph of Argument (7), there is a non-faulty node that will complete the task correctly and within the bounds stated by the lemma.

Argument (10) *Argument (1)'s complementary cases.* We consider the two complementary cases: (i) p_i does not observe any task in S_i to run concurrently with at least δ writes, and (ii) p_i observes some, but not all, tasks in S_i of these δ concurrent writes.

Case (i): the only task that can be in S_i without been observed to run concurrently with at least δ writes is p_i's own task T_i (line 97). When $\text{baseSnapshot}_i(S_i)$ is called, p_i accesses the majority once. Then, if $reg_i \neq prev_i$, p_i might observe that T_i has been running concurrently with at least δ writes. Then, we are back to the case in Argument (1) and the total time is one RTE plus all RTEs from Argument (1) to Argument (9). Otherwise, $\text{baseSnapshot}_i(S_i)$ returns and the proof is done, since only one RTE was spent. If $reg_i = prev_i$, p_i calls $\text{safeReg}_i()$ and then $\text{baseSnapshot}_i(S_i)$ returns. In total, two RTEs were spent.

Case (ii): S_i contains p_i's own task T_i and some non-empty set of tasks, E, that p_i has observed to be concurrent with at least δ writes, *i.e.,* $S_i = \{T_i\} \cup E$. Ignoring T_i, the same execution as in Argument (1) to Argument (9) happens, and the tasks in E ends within the sum of the RTEs in those arguments. During this execution, p_i also tries to get a result for T_i, but p_i is the only node which tries to do that. When the tasks in E get a result, T_i also gets a result if it is p_i that reads $reg_i = prev_i$. If another node gets a result for the tasks in E, task T_i might not get a result.

During the execution described above, it may happen that some nodes observe that T_i suddenly *has* been concurrent with at least δ writes. For all nodes $p_j : i \neq j$, this does not necessarily cause T_i to be added to $S_j \cap \Delta_j$, since we have to consider the case in which when p_j calls baseSnapshot$_j(S_j)$, $T_i \in S_j$ did not hold. For p_i, if this happens just after p_i received results for all other tasks in $S_i \cap \Delta_i$, *i.e.,* E, p_i essentially has to start all over again in baseSnapshot$_i(S_i)$, since S_i contains a task, T_i, that only p_i has, but no other node, p_j, has called baseSnapshot$_j(S_j) : T_i \in S_j$.

Therefore, all RTEs spent in Argument (1) to Argument (9) have to be spent again, and thus the total time for this case is twice the RTEs from those items. This "start over" behavior cannot happen more than once per T_i, because no tasks are added to $S_i \cap \Delta_i$ during baseSnapshot$_i(S_i)$, and T_i is the only p_i's task that can go from not having δ observed concurrent writes, to having δ observed concurrent writes.

Argument (3) is 1 RTE, Argument (6) is $n-1$ RTEs, Argument (7) is 2 RTEs, and Argument (8) is 1 RTE. Argument (10) tells to multiply the sum by 2. Thus, the longest happened-before relation between c_{calls} and $c_{returns}$ has $2n+8$ RTEs. $\square_{Claim\ 13}$

Claim 14. *The* write()*'s cost is* $4n+18$ *HBRs.*

Proof. Let p_i be the node that invokes write() during $a_i \in R$. The proof considers the cases in which (i) a_i does not include the execution of any line in baseSnapshot() and (ii) otherwise.

In case (i), baseWrite(), and hence write(), return within one RTE due to similar arguments to the ones that appear in the end of the proof of Theorem 2 about the number of rounds it takes to perform a write() operation. In case (ii), write() returns within $2n+9$ RTEs due to Claim 13 and case (i). $\square_{Claim\ 14}$

Claim 15. *The* snapshot()*'s cost is* $8n+2\delta+34$ *HBRs.*

Proof. Let p_i be the node that invokes snapshot$_i$(), which creates task T. Let $c_{calls} \in R$ be the state that appears immediately before, $a_i \in R$, which includes snapshot$_i$()'s innovation and let $c_{returns} \in R$ be the state that appears immediately after the step in which p_i returns from that snapshot$_i$() invocation.

The proof sums up the RTEs found in the longest possible happened-before relation between c_{calls} and $c_{returns}$.

Argument (1) p_i *returns from* baseSnapshot(). Suppose a_i includes the execution of some line in baseSnapshot$_i(S_i) : T \notin S_i$. The case of $T \in S_i$ reduces the proof to only Argument (3). Also, if a_i does not include the run of baseSnapshot$_i$(), then the proof reduces to arguments (2) and (3).

Due to Claim 13, p_i returns from baseSnapshot$_i(S_i)$ in $2n+8$ RTEs, and the system reaches $c_{do\ forever} \in R$.

Argument (2) p_i *observes* T *to be concurrent with* δ *writes.* Starting from $c_{do\ forever}$, p_i iterates in the do-forever loop.

We follow similar assumptions as in Claim 13's arguments 1 to 3.

From $c_{do\ forever}$, there can be δ RTEs before p_i has observed T to be concurrent with at least δ writes. The reason is that each RTE p_i spends in the do-forever loop corresponds to one write that T is concurrent with. One write-RTE-correspondence happens in baseWrite(), where one write occurs and p_i spends one RTE. One write-RTE-correspondence happens in baseSnapshot(), since we assume $reg_i \neq prev_i$, meaning at least one write occurs during p_i's call to baseSnapshot().

Argument (3) p_i *returns from* baseSnapshot$(S) : T \in S$. If $c_{T \in \Delta}$ (Argument (2)) is reached after baseSnapshot$_i$()'s returns, then p_i may call baseWrite$_i$(), which takes 1 RTE. Then, p_i has to call baseSnapshot$_i(S_i) : T \in S$ and returns within $2n+8$ (Claim 13). By then, p_i has a result for T, *i.e.,* $snapshot_i()$ returns and $c_{returns}$ is reached.

Arguments (1) to (3) consider $2n+8$, δ, and $2n+9$ RTEs, resp. Thus, the longest happened-before relation between c_{calls} and $c_{returns}$ has $4n+\delta+17$ RTEs. $\square_{Claim\ 15}$

$\square_{Lemma\ 6}$

$\square_{Theorem\ 4}$

6. Bounded variations on Algorithms 3 and 4

This section discusses how we can obtain bounded variations of our two unbounded self-stabilization algorithms. Dolev et al. [21, Section 10] present a solution to a similar transformation: They show how to take a self-stabilizing atomic MWMR register algorithm for message passing systems that uses unbounded operation indices and transform it to an algorithm that uses bounded indices. We review the techniques that Dolev et al. use and explain how a similar transformation can also be used for Algorithms 3 and 4 concerning their different operation indices.

The procedure by Dolev et al. [21, Section 10] considers operation indices, which they call tags, whereas the proposed algorithms refer to operation indices as (i) ts values in the variables and message fields, as well as (ii) ssn and sns values in the variables and message fields. That is, Dolev et al. consider just one type of operation index whereas we consider several types. For simple presentation, when describing next the procedure by Dolev et al., we refer to the case of many kinds of operation indices.

1. Once node $p_i \in P$ stores an operation index that is at least MAXINT, node p_i disables the invocation of all operations (of all types) while allowing the completion of the existing ones (until all nodes agree on the highest index for each type of operation, cf. item 2), where MAXINT is a huge constant, say, MAXINT $= 2^{64}-1$.
2. While the invocation of new operations (of all types) is disabled (by item 1), the gossip procedure keeps on propagating the maximal operation indices (and merges the arriving information with the local one). Eventually, all nodes share the same operation indices (for all types). At that point in time, the procedure for dealing with integer overflow events 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.

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. [33] or Arora and Gouda [34]. We note that the system settings of Awerbuch et al. [33] assume execution fairness. The same can be said for Arora and Gouda [34]. This is because it is assumed 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).

7. Conclusion

We showed how to transform the non-self-stabilizing algorithms of Delporte-Gallet et al. [16] into ones that can recover after the last occurrence of a transient fault. 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 [16].

One future direction emanating from this work is to consider the problem of lattice agreement [35]. We believe the techniques developed here can be used when solving this generalized problem more tailored for message-passing systems.

CRediT authorship contribution statement

Chryssis Georgiou: Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Conceptualization; **Oskar Lundström:** Writing – original draft, Methodology, Investigation, Conceptualization; **Elad Micheal Schiller:** Writing – original draft, Validation, Resources, Project administration, Methodology, Investigation, Conceptualization.

Data availability

No data was used for the research described in the article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was partially supported by the MAGIC project (Meeting Automotive Liability Challenges Through Forensics Soundness), funded by Vinnova within the FFI program under grant number 2024–03687.

Appendix A. Additional proof details

Lemma 7.
Let R be an unbounded Algorithm 4's execution. Eventually, the system reaches $c \in R$ in which ts_i is greater than or equal to any p_i's timestamp value. Also, suppose p_i takes a step immediately after c that includes line 116. Then in c, $ts_i = reg_i[i].ts = reg_j[i].ts$ and $m \in channel_{i,j} \cup channel_{j,i} : m.reg[i].ts = ts_i$.

Lemma 7's proof is implied by Claims 16, 17, 20, and 19, which can be read sequentially until Lemma 7's end of proof symbol "$\square_{Lemma\ 7}$".

Proof. Claim 16 denotes by $X_{i,\ell}$ the ℓth value stored in X_i during R, where $\ell \in N$.

Claim 16.
The sequences $ts_{i,\ell}$, $reg_{i,\ell}[i].ts$, $reg_{j,\ell}[i].ts$, $reg_{i,\ell}[i]$ and $reg_{j,\ell}[i]$ are non-decreasing.

Proof. Algorithm 4 does only the following actions on ts and reg fields: increment (line 116) and merge using the max function (lines 99, 102, 137, 139, and 142), *i.e.,* there are no assignments. The claim holds since these fields are never decremented during R. $\square_{Claim\ 16}$

Claim 17.
Eventually $ts_i \geq reg_i[i].ts$.

Proof. Since R is unbounded, $p_j \in P$ calls lines 101 to 109 and line 141 for an unbounded number of times. By the proof of Claim 16, only line 116 changes ts_i, via an increment whereas lines 99, 102, 137, 139, and 142 update ts_i and $reg_i[i].ts$ by taking the maximum of $\{ts_i, reg_i[i].ts\}$. The rest of the proof is implied by Claim 16, and the fact that p_i executes lines 99, 102, 137, 139, and 142 eventually. $\square_{Claim\ 17}$

Algorithm 4 sends GOSSIP messages in line 106, request messages in lines 118, 124, and 128 as well as replies in lines 140, 146 and 135, respectively. Claim 20's proof considers lines 106, 118, 124, and 128 in which p_i sends a request message to p_j, whereas Claim 19's proof considers lines 106, 140, 146 and 135 in which p_j replies or gossips to p_i.

Claim 18.

Let $m \in channel_{i,j}$ and reg_m be the value of the reg field in m, where $p_i, p_j \in P$ are non-failing. Eventually, $reg_i[i].ts \geq reg_m[i].ts$ and $reg_i[i].ts \geq regJ[i].ts$ whenever p_j raises the events GOSSIP(), WRITE(), SNAPSHOT(), *or* SAFE().

Proof. Suppose p_i indeed sends m, *i.e.,* m does not appear in R's starting state. Let $a_k \in R$ be the first step in which p_i calls lines 106, 118, 124, and 128 and for which there is $a_{depart,k} \in R$, which appears after a_k and in which m is sent. The value of $reg_m[i].ts$ is defined by $reg_i[i].ts$ in the state that immediately precedes a_k. The rest of the proof relies on the fact that until m arrives to p_j, $reg_i[i].ts \geq reg_m[i].ts$ holds (Claim 16).

Let $a_{arrival,k} \in R$ be the first step that appears after $a_{depart,k}$ in R in which p_j delivers m. By the assumption that R is unbounded, $a_{arrival,k} \in R$ eventually. During $a_{arrival,k}$, node p_j raises the message delivery event GOSSIP($regJ$) (when a_k considers line 106), WRITE($regJ$) (when a_k considers line 118), SNAPSHOT($regJ, ssn$) (when a_k considers line 124), or the message SAFE(Aj) (when a_k considers line 128 and $Aj = \{k, pndTsk[k].sns, prev\}$), such that $reg_i[i].ts \geq reg_m[i].ts = regJ[i].ts$.

Suppose $a_k \notin R$, *i.e., m* appears in R's starting state. Eventually, all messages in transit to p_j arrive (or lost). Thus, there is a suffix, R', of R in which all delivered messages during R' were indeed sent during R. Thus, the above proof holds with respect to messages received in R' that were sent in R. $\square_{Claim\ 20}$

Claim 19.

Let $m \in channel_{j,k}$ and reg_m be the value of the reg field in m, where $p_i, p_j, p_k \in P$ are non-failing nodes and $i = k$ may or may not hold. Eventually, $reg_j[i].ts \geq reg_m[i].ts$ and $reg_i[i].ts \geq regJ[i].ts$ whenever node p_k raises the events GOSSIP($regJ$), WRITEack($regJ$), SNAPSHOTack($regJ, \bullet$), *or* SAFEack(Aj).

Proof. Suppose p_k sends m, *i.e.,* m does not appear in R's starting state. Let $a_k \in R$ be the first step in R in which p_k calls line 106, 118, 124, or 128 and for which $\exists a_{depart,k} \in R$ that appears in R after a_k. Note that $reg_m[i].ts$ is defined by $reg_i[i].ts$ in the state that immediately precedes a_k. The rest of the proof relies on the fact that until m arrives to p_j, the invariant $reg_i[i].ts \geq reg_m[i].ts$ holds (due to Claim 16).

Let $a_{arrival,k} \in R$ be the first step that appears after $a_{depart,k}$, if there is any such step, in which the p_j delivers m that $a_{depart,k}$ transmits. Eventually, $a_{arrival,k} \in R$, during which p_j raises GOSSIP($regJ$) (when a_k considers line 106), WRITEack($regJ$) (when a_k considers line 140), SNAPSHOTack($regJ, \bullet$) (when a_k considers line 146), or SAFEack(AJ) (when a_k considers line 135), such that $reg_i[i].ts \geq reg_m[i].ts = regJ[i].ts$. For the $a_k \notin R$ case, the proof follows Claim 20's arguments. $\square_{Claim\ 19}$ $\square_{Lemma\ 7}$

Claim 20.

Let $m \in channel_{i,j}$ and reg_m be the value of the reg field in m, where $p_i, p_j \in P$ are non-failing. Eventually, $reg_i[i].ts \geq reg_m[i].ts$ and $reg_i[i].ts \geq regJ[i].ts$ whenever p_j raises the events GOSSIP(), WRITE(), SNAPSHOT(), *or* SAFE(Aj).

Proof. Suppose p_i indeed sends m, *i.e.,* m does not appear in R's starting state. Let $a_k \in R$ be the first step in which p_i calls line 106, 118, 124, or 128 and for which there is $a_{depart,k} \in R$, which appears after a_k and in which m is sent. The value of $reg_m[i].ts$ is defined by $reg_i[i].ts$ in the state that immediately precedes a_k. The rest of the proof relies on the fact that until m arrives to p_j, $reg_i[i].ts \geq reg_m[i].ts$ holds (Claim 16).

Let $a_{arrival,k} \in R$ be the first step that appears after $a_{depart,k}$ in R in which p_j delivers m. By the assumption that R is unbounded, $a_{arrival,k} \in R$ eventually. During $a_{arrival,k}$, node p_j raises the message delivery event GOSSIP($regJ$) (when a_k considers line 136), WRITE($regJ$) (when a_k considers line 138), SNAPSHOT($regJ, ssn$) (when a_k considers line 141), or SAVE(AJ) (when a_k considers line 131), such that $reg_i[i].ts \geq reg_m[i].ts = regJ[i].ts$.

Suppose $a_k \notin R$, *i.e., m* appears in R's starting state. Eventually, all messages in transit to p_j arrive (or lost). Thus, there is a suffix, R', of R in which all delivered messages during R' were indeed sent during R. Thus, the above proof holds with respect to messages received in R' that were sent in R. $\square_{Claim\ 20}$

Lemma 8.

Let R be an Algorithm 4's unbounded execution. Eventually, the system reaches $c_x \in R$ in which ssn_i is greater than or equal to any p_i's sequence number.

Proof. Claims 21, 22 and 23 prove the lemma.

Claim 21.

The sequence $ssn_{i,\ell}$ is non-decreasing.

Proof. Algorithm 4 only increments (line 122), but never assigns ssn values. Thus, the claim is true, because ssn is never decremented. $\square_{Claim\ 21}$

The proofs of Claims 22 and 23 follow by similar arguments to the ones of Claims 20 and 19.

Claim 22.

Let $m \in channel_{i,j}$ be a SNAPSHOT *message on transit from p_i to p_j that includes the field $ssn = ssn_m$. Eventually, $ssn_i \geq ssn_m$; also when p_j raises the* SNAPSHOT() *event.*

Claim 23.

Let $m \in channel_{j,i}$ be a SNAPSHOTack *message on transit from p_j to p_i and ssn_m the value of the reg field in m. Eventually, $ssn_i \geq ssn_m$; also when p_j raises the* SNAPSHOTack() *event.*

□*Lemma 8*

The variables and fields of sns and the data structure $pndTsk$ in Algorithm 4 follow the same propagation pattern as the variables and fields of ts and reg in Algorithm 3. For completeness, we provide Corollary 1, which restates the propagation argument in the context of sns and $pndTsk$. The proof of Corollary 1 follows the same structure as Claim 3, replacing reg by $pndTsk$ and ts by sns, and observing that the same gossip and acknowledgment messages disseminate the values monotonically.

Corollary 1. *For Algorithm 4, the dissemination of the fields $pndTsk[][i].sns$ satisfies the same monotonicity and eventual dominance properties as the dissemination of $reg[][i].ts$ in Algorithm 3. In particular, for any non-failing $p_i, p_j \in P$, eventually $pndTsk_j[i].sns \leq pndTsk_i[i].sns$.*

References

[1] J.L. Welch, J.E. Walter, Link Reversal Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2011.
[2] G. Taubenfeld, Distributed Computing Pearls, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2018.
[3] E. Ruppert, Implementing shared registers in asynchronous message-passing systems, in: Encyclopedia of Algorithms, Springer, 2016, pp. 954–958.
[4] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, J. ACM 40 (4) (1993) 873–890.
[5] J.H. Anderson, Multi-writer composite registers, Distrib. Comput. 7 (4) (1994) 175–195.
[6] M. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[7] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[8] S. Dolev, Self-Stabilization, MIT Press, 2000.
[9] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2019.
[10] L.M. Kirousis, P.G. Spirakis, P. Tsigas, Simple atomic snapshots: a linear complexity solution with unbounded time-stamps, Inf. Process. Lett. 58 (1) (1996) 47–53.
[11] L.M. Kirousis, P.G. Spirakis, P. Tsigas, Reading many variables in one atomic operation: solutions with linear or sublinear complexity, IEEE Trans. Parallel Distrib. Syst. 5 (7) (1994) 688–696.
[12] H. Attiya, Robust simulation of shared memory: 20 years after, Bull. EATCS 100 (2010) 99–113.
[13] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous fail-prone network systems, (2019). arXiv:1906.06420
[14] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (1995) 124–142.
[15] V. Gramoli, N. Nicolaou, A.A. Schwarzmann, Consistent Distributed Storage, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2021.
[16] C. Delporte-Gallet, H. Fauconnier, S. Rajsbaum, M. Raynal, Implementing snapshot objects on top of crash-prone asynchronous message-passing systems, IEEE Trans. Parallel Distrib. Syst. 29 (9) (2018) 2033–2045.
[17] S. Delaët, S. Devismes, M. Nesterenko, S. Tixeuil, Snap-stabilization in message-passing systems, J. Parallel Distrib. Comput. 70 (12) (2010) 1220–1230.
[18] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing indulgent zero-degrading binary consensus, Theor. Comput. Sci. 989 (2024) 114387.
[19] O. Lundström, M. Raynal, E.M. Schiller, Self-stabilizing multivalued consensus in asynchronous crash-prone systems, Theor. Comput. Sci. 1022 (2024) 114886.
[20] R. Duvignau, M. Raynal, E.M. Schiller, Self-stabilizing Byzantine fault-tolerant repeated reliable broadcast, Theor. Comput. Sci. 972 (2023) 114070.
[21] S. Dolev, T. Petig, E.M. Schiller, Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks, Algorithmica 85 (1) (2023) 216–276.
[22] M. Canini, I. Salem, L. Schiff, E.M. Schiller, S. Schmid, *Renaissance*: a self-stabilizing distributed SDN control plane using in-band communications, J. Comput. Syst. Sci. 127 (2022) 91–121.
[23] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Practically-self-stabilizing virtual synchrony, J. Comput. Syst. Sci. 96 (2018) 50–73.
[24] S. Dolev, A. Hendin, M. Herlihy, M. Potop-Butucaru, E.M. Schiller, Self-stabilizing replicated state machine coping with Byzantine and recurring transient faults, (2025). arXiv:2506.12900
[25] S. Dolev, A. Hanemann, E.M. Schiller, S. Sharma, Self-stabilizing end-to-end communication in (Bounded capacity, omitting, duplicating and non-FIFO) dynamic networks - (Extended abstract), in: SSS, 7596 of *LNCS*, Springer, 2012, pp. 133–147.
[26] T. Albouy, D. Frey, R. Gelles, C. Hazay, M. Raynal, E.M. Schiller, F. Taïani, V. Zikas, Near-optimal communication Byzantine reliable broadcast under a message adversary, in: S. Bonomi, L. Galletta, E. Rivière, V. Schiavoni (Eds.), 28th International Conference on Principles of Distributed Systems, OPODIS 2024, Lucca, Italy, December 11–13, 2024, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:29.
[27] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous failure-prone networked systems, in: Networked Systems - 7th International Conference, NETYS, 2019, pp. 113–130.
[28] C. Georgiou, O. Lundström, E.M. Schiller, Self-stabilizing snapshot objects for asynchronous failure-prone networked systems, in: ACM Symposium on Principles of Distributed Computing, PODC, 2019, pp. 209–211.
[29] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[30] S. Dolev, C. Georgiou, I. Marcoullis, E.M. Schiller, Self-stabilizing reconfiguration, in: NETYS, 10299 of *LNCS*, 2017, pp. 51–68. https://doi.org/10.1007/978-3-319-59647-1_5
[31] S. Dolev, S. Dubois, M. Potop-Butucaru, S. Tixeuil, Stabilizing data-link over non-FIFO channels with optimal fault-resilience, Inf. Process. Lett. 111 (18) (2011) 912–920.
[32] J.E. Burns, M.G. Gouda, R.E. Miller, Stabilization and pseudo-stabilization, Distrib. Comput. 7 (1) (1993) 35–42.
[33] B. Awerbuch, B. Patt-Shamir, G. Varghese, S. Dolev, Self-stabilization by local checking and global reset, in: WDAG, 857 of *LNCS*, Springer, 1994, pp. 326–339.
[34] A. Arora, M.G. Gouda, Distributed reset, IEEE Trans. Comput. 43 (9) (1994) 1026–1038.
[35] J.M. Faleiro, S.K. Rajamani, K. Rajan, G. Ramalingam, K. Vaswani, Generalized lattice agreement, in: PODC, ACM, 2012, pp. 125–134.