Practically-self-stabilizing vector clocks without scheduling fairness

Acta Informatica (2026) 63:32
https://doi.org/10.1007/s00236-026-00544-z

RESEARCH

Practically-self-stabilizing vector clocks without scheduling fairness

Iosif Salem[1] · **Elad M. Schiller**[1]

Received: 7 June 2023 / Accepted: 30 June 2026

Abstract
Vector clock algorithms are fundamental wait-free building blocks that enable the causal ordering of events. As wait-free algorithms, they are designed to complete their operations within a finite number of steps. Stabilizing algorithms aid the system in recovering after the occurrence of transient faults, such as soft errors and arbitrary violations of the assumptions according to which the system was designed to behave. To the best of our knowledge, this paper introduces the first stabilizing vector clock algorithm for asynchronous crash-prone message-passing systems that can achieve wait-free recovery after the occurrence of transient faults. In such settings, demonstrating finite and wait-free recovery from transient faults as well as communication and crash failures, bounding the message and storage sizes, handling the removal of stale information without blocking, and addressing concurrent counter overflow events at different network nodes pose significant challenges. We propose an algorithm that ensures safety in the absence of transient faults and offers bounded time recovery during fair executions following the last transient fault. The novelty lies in guaranteeing a bound on the number of safety violations, even in the absence of execution fairness (where existing algorithms may become permanently blocked due to both transient faults and crash failures). Considering the usefulness of vector clocks in facilitating various elementary synchronization building blocks in asynchronous systems without requiring remote replica synchronization, our analytical insights hold promise for designing other systems that cannot guarantee execution fairness.

1 Introduction

Vector clocks have become essential for various applications, including distributed snapshot construction [22], conflict-free replicated data types (CRDTs) [34], distributed debugging [25], and event ordering in distributed systems [24]. These tasks are often challenging because they require capturing causality and maintaining consistency across asynchronous, fault-prone environments without relying on synchronization mechanisms, such as synchro-

✉ Iosif Salem
iosif@chalmers.se

✉ Elad M. Schiller
elad@chalmers.se

[1] Department of Computer Science and Engineering, Chalmers University of Technology, Rännvägen 6, 412 96 Göteborg, Sweden

Published online: 20 August 2026

nized clocks, phase-based commit protocols [6, 35], and quorum systems [30]. In this work, we propose a vector clock algorithm that is more robust than existing solutions.

1.1 The problem

Vector clocks are distributed data structures widely used to capture causal relationships in decentralized systems without relying on synchronization mechanisms, such as synchronized clocks or phase-based commit protocols [6, 35]. They enable reasoning about the order of events, such as message sending and receiving, across processes in the system.

Each process maintains a vector of integers, where each entry corresponds to the number of events that have occurred at each process. When processes exchange messages, they also exchange their vector clocks, allowing each process to track the causal relationships between its own events and the events of others.

Definition 1.1 *(Vector Clocks)* We define a *vector clock* data structure, V, in a system of N processes as an N-dimensional vector of non-negative integers. Each process p_i maintains its own vector V_i, where $V_i[i]$ represents the number of events that have occurred locally at process p_i. We define vector clock operations as follows:

- *Increment:* Before process p_i sends a message, it increments $V_i[i]$ by one to reflect the occurrence of a new event.
- *Merge:* When process p_j receives a message from p_i, it updates its vector clock by taking the element-wise maximum between its local vector and the vector received from p_i. Specifically, for each $k \in \{1, \ldots, N\}$, process p_j sets $V_j[k] \leftarrow \max(V_j[k], V_i[k])$, where V_i is the vector attached to the message.

The partial order relation, $\leq_C$, between two vector clocks V and W is defined as $(V \leq_C W) \iff (\forall i \in \{1, \ldots, N\}, V[i] \leq W[i])$ (see the detailed definition in Property 1 of Sect. 4.3). This relation is used to determine causal precedence: if $V \leq_C W$, the events corresponding to V happened before those corresponding to W.

1.2 Fault model

We study an asynchronous message-passing system with no guarantees of communication delay, and the proposed algorithm cannot explicitly access the (local) clock. We assume this asynchronous system is prone to (up to $N - 1$ undetectable) crash failures, where a crashed node stops taking steps forever. We refer to such failures as *faults specified by the fault model*, as they are well-understood and explicitly considered during the algorithm's design, even though the timing of their occurrence and the identities of the faulty processes are unknown.

In addition, we consider *transient faults*, which represent any temporary violation of the assumptions under which the system was designed to operate, such as memory corruption due to soft errors. These transient faults can arbitrarily change the system state unpredictably while leaving the program code intact. By system state, we refer to the complete set of values of all variables in each process and the status of communication channels. In practice, transient faults are challenging to observe and typically cannot be explicitly specified in the fault model. Since their exact impact is unknown at design time, they are considered *non-specific to the fault model*.

Self-stabilizing systems A system is *self-stabilizing* if, starting from any arbitrary state (including states resulting from transient faults), it is guaranteed to reach a legitimate state within a finite number of steps. Once in a legitimate state, the system remains correct indefinitely. The definition of self-stabilizing systems encompasses two key properties [9] (for details, see Definition 3.1 of Sect. 3.3):

1. **Convergence:** From any arbitrary state, the system run (including faults specified by the fault model) eventually reaches a legitimate state within a bounded number of steps without any external intervention.
2. **Closure:** Once the system reaches a legitimate state, it remains in a legitimate state while tolerating faults specified by the fault model.

Self-stabilizing systems are designed to tolerate both faults specified by the fault model and transient faults, which can arbitrarily corrupt the system's state. They do this without requiring the algorithm to detect faults directly, whether specified by the fault model or not. Instead, they ensure recovery to a correct state within a bounded time. Additionally, the correctness proof of self-stabilizing systems assumes that no further transient faults occur during execution.

Open issues Asynchronous message-passing systems (with bounded memory and channel capacity) can indefinitely hide stale information that transient faults introduce unexpectedly. At any time, this corrupted data can cause the system to violate safety requirements, *e.g.,* data consistency might be lost.

That is, an adversarial timing and order of events and message deliveries can both (i) violate liveness guarantees, *e.g.,* defer termination, and (ii) use opportunities to disrupt the system via a systematic exposure of hidden stale information. The timing of these exposures can aim at prolonging (and, if possible, preventing) recovery from transient faults. For these reasons, self-stabilizing systems often assume fair scheduling to ensure that all stale information is eventually removed [3, 10, 33], *i.e.,* nodes that have applicable steps (infinitely often) are allowed to take any step eventually. Dubois and Tixeuil [17] survey existing scheduling hypotheses in the self-stabilization literature.

As mentioned, any fairness assumption contradicts our fault model, yet without such fairness assumptions, some elementary problems do not have a straightforward answer. For example, any transient fault can cause a bounded counter to reach its maximum value, and yet the system might need to increment the counter an unbounded number of times after that overflow event. This challenge is more significant when there is no elegant way to maintain order among the different counter values, say, by wrapping around to zero upon counter overflow.

Without any fairness assumptions, a system that takes an extraordinary (or even an infinite) number of steps is bound to break any ordering constraint, because an adversarial timing and ordering of events and message deliveries can arbitrarily suspend node operations and defer message arrivals until such violations occur. Having practical systems in mind, we consider this number of (sequential) steps to be no more than practically infinite [13, 14], say, 2^{64}, since sequentially counting from zero to 2^{64} takes longer than the system's practical lifetime. For example, assuming a message is sent or received every nanosecond, counting from zero to 2^{64} takes more than 584 years.

Proposed approaches The design criteria of *practically-self-stabilizing systems* [2, 6, 14] require a bounded number of safety violations during any practically infinite period of the system run. When this bound is low, practically-self-stabilizing systems ensure that

deviations from satisfying the requirements of the studied problem occur only sparsely over practically infinite executions. The studied criteria offer weaker guarantees than the ones of self-stabilizing systems. Still, they do not require fair scheduling when solving, for example, the integer overflow challenge, as shown by Alon et al. [2] and Dolev et al. [14, Algorithm 2]. We note that the concept of practically-self-stabilizing systems is named after the idea of *practically infinite executions* [13].

1.3 Existing solutions

To address the challenge of tracking causality in self-stabilizing systems, Arora, Kulkarni, and Demirbas [5] proposed the use of resettable vector clocks. However, this solution is constrained by its reliance on blocking operations during resets and synchronization mechanisms, which limit performance and scalability, particularly in the presence of transient faults. Specifically, resettable vector clocks use blocking global resets that can disrupt system operations (see Sect. 2 for details). Our work overcomes these limitations by providing a practically-self-stabilizing vector clock algorithm that avoids both blocking operations and synchronization dependencies, thereby ensuring resilience even after arbitrary state corruption.

1.4 Our contributions

We present a fundamental building block for dependable decentralized systems that require reasoning about event causality. To achieve this, we address several scientific challenges associated with vector clocks: (1) capturing causality in asynchronous systems, (2) ensuring bounded resource usage, and (3) regaining and preserving consistency despite crash and transient faults, without relying on synchronized clocks or fairness assumptions. Addressing these challenges necessitates novel solutions.

To tackle these challenges, we propose a practically-self-stabilizing vector clock algorithm, which introduces the following key advancements:

- **Crash failure resilience:** Our algorithm tolerates up to $N - 1$ crash failures in an asynchronous message-passing system, allowing processes to continue correct operations despite significant node failures.
- **Bounded Storage and Communication Overhead:** Our solution considers $3N$ integers and two labels **(author?)** 14 per vector, where N is the number of nodes. Each label has $\mathcal{O}(\mathcal{C}N^3)$ bits, where the constant $\mathcal{C} \in \mathbb{Z}^+$ is a system-dependent upper bound on the channel capacity. The proposed solution stores the most recent message from any node, necessitating a total of $\mathcal{O}(\mathcal{C} \cdot N^4)$ bits per node.
- **Recovery from transient faults without fairness and synchrony assumptions:** We establish that recovery occurs following the last transient fault. As shown in Theorem 8.1, within any practically infinite execution, there are at most $\mathcal{O}(N^8\mathcal{C})$ occurrences where the problem requirements are violated. This bound serves as our complexity measure for practically-self-stabilizing systems.

In summary, our contributions are as follows:

- We formulate the problem of maintaining vector clocks in practically-self-stabilizing systems prone to node crashes and transient faults (Sect. 4.2).
- We propose a practically-self-stabilizing vector clock algorithm (Sect. 6 and 7), the first to recover after the occurrence of transient faults while ensuring resilience in asyn-

chronous, crash-prone systems without relying on synchronized clocks, synchronization mechanisms, or fair scheduling.
- We demonstrate the correctness of our contribution through rigorous proof (Sect. 8). Our proof uses a new enhancement of algorithm composition for practically-self-stabilizing systems (Sect. 5.1).

1.5 Paper organization

We review the related work in Sect. 2. In Sect. 3, we present the system settings and design criteria. In Sect. 4, we provide a detailed version of the problem definition. We present an interface to a labeling scheme in Sect. 5. Then, we present novel techniques (Sect. 6), upon which we base our algorithm (Sect. 7) and proofs (Sect. 8). Our conclusions appear in Sect. 9.

2 Related work

Vector clocks are distributed data structures that enable the analysis of causality among events in distributed systems. According to Shapiro et al. [34], vector clocks are essential building blocks for several conflict-free replicated data types (CRDTs). CRDTs are distributed data structures that can be shared among many replicas in asynchronous networks. Updates to replicas occur independently, ensuring strong eventual consistency and guaranteeing that all replicas will eventually converge to the same state even without synchronization or rollbacks. More specifically, strong eventual consistency in CRDTs is achieved by ensuring that operations on replicas are commutative, meaning that the final state remains consistent regardless of the order in which updates are applied [35]. Strong eventual consistency does not imply the same level of guarantees as linearizability or sequential consistency. Still, it is sufficient in scenarios where immediate consistency is not required, and replicas only need to converge eventually. Bounded non-stabilizing solutions for the vector clocks problem exist in the literature [1, 29]. This short literature review focuses on bounded vector clock algorithms in the context of transient faults.

2.1 Self-stabilizing systems

Arora, Kulkarni, and Demirbas [5] proposed self-stabilizing resettable vector clocks. They consider distributed applications that are structured in phases and track causality merely within a bounded number of successive phases. Whenever the system exceeds the number of clock values that can be used in one phase, resettable vector clocks use reset operations that allow the system to move to the next phase and reuse clock values. In the absence of faults as presented in [5], the system uses non-blocking resets. Nevertheless, the presence of faults can bring the algorithm in [5] to use a *blocking* global reset that requires fair scheduling (and no failing nodes). Our solution does not use blocking operations even after an arbitrary system state corruption.

Arora, Kulkarni, and Demirbas also discuss the possibility of using global snapshots to provide better complexity measures. They rule out this approach because it can change the communication patterns (in addition to using blocking operations during the recovery period). Another concern is how to identify a self-stabilizing snapshot algorithm that can deal with crash failures, *e.g.,* [8, Section 6] declared that this is an open problem.

2.2 Practically-stabilizing systems

There are practically-self-stabilizing algorithms for solving agreement [6, 13], state-machine replication [6, 14], and shared memory emulation [7]. None of them considers the studied problem. They all rely on synchronization mechanisms, *e.g.,* quorum systems. Alon et al. [2] and Dolev et al. [14, Algorithm 2] consider practically-self-stabilizing algorithms that handle counter overflow events using labeling schemes. Both algorithms use these labeling schemes together with synchronization mechanisms to implement shared counters. We solve a different problem and propose a practically-self-stabilizing algorithm for vector clocks that uses a labeling scheme but does not use any synchronization mechanism.

Our work adopts the labeling scheme of Dolev et al. [14], which generalizes the one by Alon et al. [2]. These labeling schemes can be used as epoch numbers of bounded counters. This way, a new label is created whenever a bounded counter reaches its maximum value. In the absence of transient faults, Alon et al. [2] and Dolev et al. [14] guarantee that this newly created label is unique in the entire current system state. Moreover, the labeling schemes in [2, 14] also recycle obsolete labels. Therefore, using the labeling schemes in [2, 14], one can indeed create a bounded size counter that can count to infinity while maintaining a proper change of epoch labels (in the absence of transient faults). The reader can find a comprehensive review of these labeling schemes in [31, Section 3]. We clarify that the creation of a new label does not require blocking, say, until new information arrives.

Dolev, Israeli, and Moran [12] introduced techniques for composing self-stabilizing algorithms under fair scheduling. Using similar assumptions, Gouda and Herman [21], Stomp [36], and Varghese [38] further studied the composition of self-stabilizing algorithms. However, as mentioned, our work does not assume execution fairness; thus, new composition techniques are required. Section 5.1 extends the composition technique presented in [10, 12] using their server-client interface.

2.3 Related self-stabilizing abstractions

The study of self-stabilizing algorithms for distributed systems has produced a rich collection of fault-tolerant communication and coordination abstractions, including group membership services [11], reliable and Byzantine-resilient broadcast [18, 26], total-order broadcast [27], consensus [19, 28], replicated state machines [16], snapshot objects [20], and distributed shared memory [15]. Many of these constructions implicitly rely on notions of logical time, event ordering, or causality preservation to reason about system behavior and correctness. Nevertheless, the problem of maintaining vector-clock information itself in a practically-self-stabilizing manner has received considerably less attention.

The present work addresses this gap by providing a practically-self-stabilizing vector clock mechanism that tolerates transient faults while supporting long-running executions under bounded resources. In this sense, the proposed construction is complementary to existing self-stabilizing abstractions and may serve as a reusable causality-tracking component for future self-stabilizing communication and coordination services.

2.4 Bibliographical note

A more concise version of this work (with proof sketches) appears in [32]. This version of our work includes all the details needed for the proof. We note that Sect. 6 in [32] presents

the key ideas needed for understanding our solution in a way that resembles the presentation in [32].

3 System settings

We study an asynchronous message-passing system that provides no guarantees on communication delay, and in which the algorithm cannot explicitly access the local clock. The system includes a set of processors $P = \{p_1, \ldots, p_N\}$, which are computing and communicating entities that we model as finite state-machines. Processor p_i has an identifier, $ID(i)$, unique in P. We assume that $ID(i) = i$ for a simple presentation. Let p_i and p_j be a pair of processors. We presume that p_i has access to a reliable non-FIFO communication channel, $channel_{i,j}$, for sending messages to p_j.

3.1 The interleaving model

We briefly present the interleaving model; please see Dolev [10, Chapter 2.1] for more details.

The processor's program is a sequence of *(atomic) steps*. Each *step* starts with an internal computation and finishes with a single communication operation, *i.e.,* packet $send$ or $receive$.

The *state*, s_i, of $p_i \in P$ includes all of p_i's variables as well as the set of all messages in p_i's incoming communication channels. Note that p_i's step can change s_i as well as remove a message from $channel_{j,i}$ (upon message arrival) or queue a message in $channel_{i,j}$ (when a message is sent).

The term *system state* refers to a tuple of the form $c = (s_1, s_2, \cdots, s_N)$, where each s_i is p_i's state (including messages in transit to p_i). 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 system state c_{x+1}, except for the initial one, c_0, is obtained from the preceding system state c_x by the execution of a_x.

3.1.1 Abstract tasks, execution length, and number of deviations

We define the system's *abstract task*, $\mathcal{T}$, as a set of executions characterized by a designated set of output variables and constraints, referred to as the task requirements. This definition focuses on essential behaviors that define the desired system behavior without necessarily considering all implementation details. In the context of the studied problem, only the values of $V_i[]$ are relevant, where $p_i \in P$ represents any processor and $V_i[]$ denotes p_i's replica of the vector clock.

We detail the task requirements in Sect. 4.2 and denote by E_{AT} the set of executions in which the abstract task holds, *i.e.,* those that satisfy the requirements of the abstract task. Furthermore, any system state where the requirements in Sect. 4.2 do not hold is counted as a single deviation from the abstract task.

We say that the length of a finite execution $R = c_0, a_0, c_1, a_1, \ldots, c_{x-1}, a_{x-1}$ is equal to x, which we denote by $|R| = x$. We denote with f_R the number of deviations from the abstract task in an execution R, *i.e.,* the number of states in R in which the task requirements do not hold. Hence, $R \in \mathsf{E}_{\mathsf{AT}} \iff f_R = 0$.

3.1.2 Concatenation-of $\circ$ and segment-of $\sqsubseteq$ operators

Suppose that R' is a prefix of an execution R, and R'' is the remaining suffix of R. We use the concatenation operator $\circ$ to write that $R = R' \circ R''$, such that R' is a finite execution that starts with the initial system state of R and ends with a step that is immediately followed by the initial state of R''. We denote by $R' \sqsubseteq R$ the fact that R' is a subexecution (or segment) of R. As in [10, Chapter 2.10], the correctness proof demonstrates its claims by considering whether a given execution segment, rather than the entire execution prefix, belongs to E_{AT}, or not.

3.2 Fault model

As mentioned, we consider asynchronous systems that have no bound on the communication delay and the algorithm cannot explicitly access the local clock.

3.2.1 Nodes failures

At any point and without warning, p_i is prone to a crash failure, which causes p_i to forever stop taking steps (without the possibility of failure detection by any other processor in the system). We say that a processor is active during a finite execution R' if it takes at least one step in R'. We say that a processor is active throughout an infinite execution R, if it takes an infinite number of steps during R.

3.2.2 Transient faults

We also consider (arbitrary) transient faults, which are temporary violations of assumptions according to which the system was designed to operate. These transient faults can change the system state in any way (as long as the program code stays intact). As in [10], we assume that transient faults occur only before the starting system state c_0, and thus c_0 is arbitrary.

3.3 Self-stabilizing systems

Before presenting the design criteria for practically-self-stabilizing systems (Sect. 3.5), we review Dijkstra's design criterion for self-stabilizing systems [9].

Definition 3.1 *(Dijkstra's self-stabilization)* For every infinite execution R, there exists a partition $R = R' \circ R''$, such that $|R'| = z(N) \in \mathbb{N}$ and $f_{R''} = 0$, where $z(N)$ is the complexity measure.

According to Definition 3.1, during execution R'', Dijkstra does not allow the system to deviate from the task requirements. As mentioned, these requirements are the conditions that a system must satisfy to solve a problem or perform a given function or operation successfully. Arora and Gouda [4] presented a definition of Dijkstra's self-stabilization, where R' (Definition 3.1) is referred to as the *Convergence* period, during which the system recovers from the most recent transient fault, and R'' is referred to as the *Closure* period, in which the system satisfies the task requirements.

3.4 Practically-infinite executions

Dolev, Kat, and Schiller [13] considered a way to relax Dijkstra's criterion by requiring no deviation from the abstract task during periods that they consider to be *practically infinite*. These executions correspond to periods in which 2^b sequential system steps, *e.g.,* message send or receive, are longer than a quantity that represents the relevance of the computation. For example, in the case of $b = 64$ (or larger), the execution of 2^b sequential steps will take longer than 580 years (assuming a message is sent or received every nanosecond). This period is clearly longer than the expected system's lifetime.

As in [14], this paper denotes by *MAXINT* an integer that can be considered as a practically infinite quantity for a given system $\mathcal{S}$. *MAXINT* helps us ensure that all variables are bounded. We say that a *finite* execution R is *practically infinite* if $|R| \geq MAXINT$. In the following, we will focus on *MAXINT*-sized executions as they are the shortest practically-infinite executions according to our definition. Defining *MAXINT* to be a system-specific constant allows us to have a clear analysis and should not be interpreted in absolute terms, but rather as a safe upper bound on the system's lifetime (*e.g., MAXINT* $- 1$ is still an enormous quantity with respect to the lifetime of the system $\mathcal{S}$). Also, it is sufficient to study executions of *MAXINT* size, as larger ones exceed the system's lifetime even more.

3.5 Practically-stabilizing systems

This work studies systems that have abandoned the goal of guaranteeing the absence of deviations from the abstract task during unfair executions (that can be infinitely long). Instead, we aim at systems that guarantee to have very long execution fragments during which the system does not deviate from the abstract task. Specifically, Definition 3.2 and Remark 3.3 consider guarantees for sparse occurrences of deviations from the abstract task during practically infinite executions.

Definition 3.2 *(r-Practically-self-stabilizing)* Let $r \in \mathbb{R}$ be a known constant. A system is r-practically-self-stabilizing if for every practically infinite fragment R' of (a possibly infinite) execution R, $\frac{f_{R'}}{|R'|} < r$, where r is the complexity measure and, as mentioned, f_R, is the number of deviations from the abstract task in an execution R.

Remark 3.3 In this work, we aim to design r-practically-self-stabilizing systems for which r is almost zero, *i.e.,* f_R is significantly less than $|R|$. In the remainder, we will simply refer to such systems as practically-self-stabilizing. For example, systems where $f_R = poly(N)$ (polynomial on N) for a polynomial of a small degree and $|R| = \max\{MAXINT, 2^N\}$ (practically infinite) are, in the context of this paper, practically-self-stabilizing. Recall that the use of *MAXINT* is necessary for making sure that all variables are bounded. In general, the size of f_R depends on the algorithm, which impacts the complexity measure r, as we show in Theorem 8.1.

As mentioned, our design criteria also require strong eventual consistency (see Sect. 1) without using mechanisms for synchronization or roll-back, as in [2, 14]. In addition, as in all self-stabilizing systems, we require a bounded amount of memory.

4 Problem definition

In this section, we define the problem by specifying the interface our solution uses (Sect. 4.1), the task requirements (Sect. 4.2), and how these requirements can be used for inferring causal precedence (Sect. 4.3).

4.1 Interface between application and protocol layers

This work differentiates between application-level operations, which trigger system events recorded by the vector clock, and protocol-level operations, which manage the vector clock implementation. To log events, such as message sending or receipt, the distributed application invokes a protocol operation called $increment()$. Specifically:

- The protocol-level operation $increment()$ allows a processor p_i to record a local application event by incrementing the i-th entry of its vector clock replica.
- The application can also query its local vector clock replica to observe event occurrences; see Property 1 for details.

Message encapsulation and continuous protocol operation. Application messages are sent with an accompanying vector clock. Specifically, whenever a processor sends an application message, it is encapsulated within a protocol message of the form $\langle V, appPayload \rangle$, where V is the current state of the vector clock, and $appPayload$ is the application payload.

The protocol operates independently of application messages, maintaining consistency of the vector clocks through continuous exchanges of protocol messages. This is necessary for self-stabilizing systems, as they require continuous communication to recover from transient faults, as discussed in [10, Chapter 2.3].

4.2 Task requirements

We present requirements 1 and 2, which define the abstract task (Sect. 3.1.1) of vector clocks for practically-self-stabilizing systems. These requirements serve as a more detailed problem definition than the one presented in Sect. 1.1. They allow the correctness proof (Theorem 8.1) to count the number of deviations from the abstract task. This proof must demonstrate that the algorithm accounts for all events (Requirement 1), ensuring valid state transitions, which is a matter of safety, and disseminates information about their occurrence (Requirement 2), which ensures the eventual satisfaction of task properties, corresponding to liveness. To define the requirements for practically self-stabilizing vector clock algorithms, we assume that the algorithm's unbounded execution, R, includes at least one active processor, which takes at least one computational step during R.

Requirement 1 *(Counting all events)* Let R be an execution, p_i an active processor, and $V_i^k[]$ the value of p_i's vector clock replica at $c_k \in R$. For every active processor p_i, the number of p_i's counter increments, *i.e.,* invocations of the $increment()$ operation, between the states c_k and $c_\ell \in R$ is $V_i^\ell[i] - V_i^k[i]$, where c_k precedes c_ℓ in R.

Requirement 2 states that the information held by the different replicas of the vector clocks indeed propagates in the system (when possible). As explained in Sect. 4.1, upon any invocation of the $increment()$ operation, the protocol operates on the vector-clock data structure, while the remainder of the arriving message, $\langle m_{vc}, \bullet \rangle$, is passed to the application; this remainder is denoted by $\bullet$.

Requirement 2 *(Information dissemination)*
Let R be an execution, $c \in R$ a system state, and $V_i^c[k]$ be the value that p_i records in c for the logical clock of processor p_k. Suppose that processor p_i sends message $m = \langle V, \bullet \rangle$ to an active processor p_j during the atomic step that appears in R immediately after system state $c \in R$, where V is the attached vector clock. Suppose that p_j receives message m during the atomic step that appears in R immediately before system state $c' \in R$. It holds that $V_j^{c'}[k] \geq V_i^c[k]$.

Intuitively, Requirement 2 demands that if processor p_i has recorded $V_i^c[k]$ events at processor p_k until state c and at a later state c' processor p_j records the received vector clock that p_i sent at state c, then the number of events that p_j records for p_k at state c', *i.e.,* $V_j^{c'}[k]$, are at least $V_i^c[k]$.

4.3 Causal precedence

We define causal precedence based on the values recorded in the vector clocks with bounded counter values. The definition of Property 1 relies on the concept of causal precedence between two vector clocks, V and V'. We say that V *causally precedes* V' if, and only if, V' records all the invocations of the $increment()$ operations recorded in V [37]. Additionally, V and V' are *concurrent* when neither V causally precedes V' nor V' causally precedes V.

Property 1 *(Causal precedence)*
Let p_i and p_j be two processors. Let V_i and V_j be their corresponding vector clocks. The query $causalPrecedence(V_i, V_j)$ returns true if, and only if, V_i causally precedes V_j.

In a fault-free system with unbounded counters, requirements 1 and 2 trivially hold since no wrap-around events occur. That is, V causally precedes V', if $V[i] \leq V'[i]$ for every $i \in \{1, \ldots, n\}$ and $\exists_{j \in \{1,\ldots,n\}} V[j] < V'[j]$ hold [37]. However, this is not the case in asynchronous, crash-prone, and bounded-counter settings, where counter overflow events can occur. In Example 4.1, we present cases where requirements 1 and 2, as well as Property 1 do not hold due to a counter overflow event.

Example 4.1 Consider two bounded vector clocks $V_i = \langle v_{i_1}, \ldots, v_{i_N} \rangle$ and $V_j = \langle v_{j_1}, \ldots, v_{j_N} \rangle$ of processors p_i and p_j, such that upon a new event, processor p_k increments $V_k[k]$ by adding 1 (mod $MAXINT$). Assume that $V_i = V_j$ and $V_i[i] = V_j[i] = MAXINT - 1$ hold (*e.g.,* as an effect of a transient fault). In the following step, p_i increments $V_i[i]$ by 1, causing $V_i[i]$ to wrap around to $V_i[i] = 0$, while $V_j[i] = MAXINT - 1$ remains unchanged. Then, $V_i[i]$ mistakenly indicates zero events for p_i ($V_i[i] = 0$) instead of $MAXINT$, *i.e.,* requirements 1 and 2 do not hold. Also, using the definition of causal precedence in fault-free systems and unbounded counters, V_i appears to causally precede V_j, which is wrong, since V_j causally precedes V_i (p_i had one more event than what p_j records). That is, $V_i[k] = V_j[k]$ for $k \neq i$ and $V_i[i] = 0 < MAXINT - 1 = V_j[i]$, which mistakenly indicates that p_j records $MAXINT - 1$ more events than p_i. □

It is essential to consider whether an "order of events" exists within self-stabilizing asynchronous systems prone to crash failures. For example, one may question whether the relation encoded by a given system state meets the criteria for a total order, namely antisymmetry, transitivity, reflexivity, and strong connectivity among system events throughout the execution. In a fault-free environment, where all nodes exchange messages before any integer

overflow occurs, this relation may indeed satisfy these conditions. For example, one may consider the solution proposed by Arora, Kulkarni, and Demirbas [5].

However, the notion of an "order of events" cannot accurately reflect the actual sequence of events occurring in an infinite execution that starts from an arbitrary system state since there is no bound for the number of times any counter may wrap around to the zero value during the period that it takes for all nodes to exchange messages. Therefore, we frame the discussion around causality rather than a total order to avoid confusion.

5 The solution framework

The goal of the solution framework proposed in this section is to leverage the practically-self-stabilizing labeling algorithm of Dolev et al. [14] to construct a compound system that combines labeling functionalities with the ones of the proposed solution to the problem of vector clocks for practically-self-stabilizing systems. The labeling algorithm acts as a 'server' that manages identifiers (or labels) associated with system states, allowing the 'client' algorithm to utilize these labels for tracking and communication. Through the interface, the client algorithm can query, update, and cancel labels, *i.e.,* to mark obsolete labels as disabled, without disrupting the server's recovery guarantees. Importantly, our framework supports the scenarios where the client might initiate state changes—such as canceling outdated labels—to adapt to changes in the system or prevent resource exhaustion while still respecting the stabilization of the server.

In terms of composition, our framework follows the server-client paradigm from [10], where the recovery of the client algorithm depends on the prior recovery of the server. This layered approach enables a separation of concerns: the server manages labeling by creating new labels and canceling obsolete ones. At the same time, the client algorithm uses these labels to perform specific tasks related to the studied problem. Through this section, we detail how the server-client interaction is structured, the specific functionalities provided by the interface, and the mechanisms that preserve the recovery guarantees of both algorithms within the compound system.

5.1 Composition with practically-stabilizing labeling

We follow an approach for algorithm composition in message-passing systems that resembles the one in [10, Section 2.7], which considers a composition of two self-stabilizing algorithms without assuming fair execution. Following the approach of Dolev [10], let us name these two algorithms as the server and client algorithms. The server algorithm provides services and guaranteed properties that the client algorithm uses. In the composition presented in [10, Section 2.7], once the server algorithm stabilizes, the client algorithm can also start to stabilize. This way, the compound algorithm obtains more complex guarantees than the individual algorithms.

We now turn to detail our composition approach. Fig. 1 depicts the interaction between the client and server algorithms, structured to ensure practical self-stabilization in message-passing systems. As in the interface between application and protocol layers (Sect. 4.1), our approach assumes that the client messages piggyback server messages and that the server algorithm can send any message independently.

In the following, we refer to the computations of a step excluding the send or receive operation as the step's *invariant check*, which may include updates of local variables. Fig. 1

emphasizes the two primary types of atomic steps: (1) those with a *send* operation, where the server encapsulates client messages in its own before sending, and (2) those with a *receive* operation, where the server and client each perform invariant checks and updates before determining if label cancelation is required, *e.g.,* to disable obsolete labels. This approach to composition allows the server algorithm to stabilize first, followed by the client, yielding a compound self-stabilizing protocol.

- *Atomic step with send operation:* This sequence highlights how the server and client interact when sending data.
  (1) Server invariant check and updates: The step starts with the server performing its invariant check and any necessary updates to ensure its internal state is consistent.
  (2) Client invariant check and updates: After the server's update, the client performs its own invariant check and updates, ensuring it adheres to the composition's requirements.
  (3) Label cancelation condition check: After the client's updates, the framework checks whether the client algorithm has requested a label cancelation, which may be due to specific conditions like label obsolescence.
    – *If cancelation is required:* The server cancels the label, updates its state, and the process restarts from step (1).
    – *If no cancelation is needed:* The server proceeds to send a composite message.
  (4) Encapsulation of Client Message within Server Message: The server encapsulates the client's message within its own message, $m_{server} = \langle serverPart, m_{client} \rangle$. This composite message maintains the integrity of the client–server state and sends it to the appropriate destination.
- *Atomic step with receive operation:* This part of the diagram illustrates how the server and client handle received messages and maintain their respective states.
  (5) Server message reception and processing: When the server receives a message, it first checks the server portion of the message for validity. The server invariant check and updates are applied to confirm the message consistency with its own state.
  (6) Client message reception event: Following the server check, the message is delivered to the client. The server extracts and passes the client-specific portion of the received message (m_{client}) to the client algorithm.
  (7) Client Invariant Check and Updates: The client performs an invariant check and updates its state as necessary based on the received message.
  (8) Label cancelation condition check: Similar to the send process, the client verifies whether a cancelation is required based on its current state.
    – *If cancelation is required:* The server cancels the specified label, and the invariant checks are repeated starting from step (5).
    – *If no cancelation is needed:* The client completes the process for that received message.
- *Self-stabilizing end-to-end protocol:* The end-to-end protocol supports both client and server algorithms, ensuring that the messages exchanged are valid, encapsulated as needed, and cancelation conditions are properly evaluated, enabling practical self-stabilization even in the presence of crashes or message failures.

We now turn to explain the proposed framework in more detail.

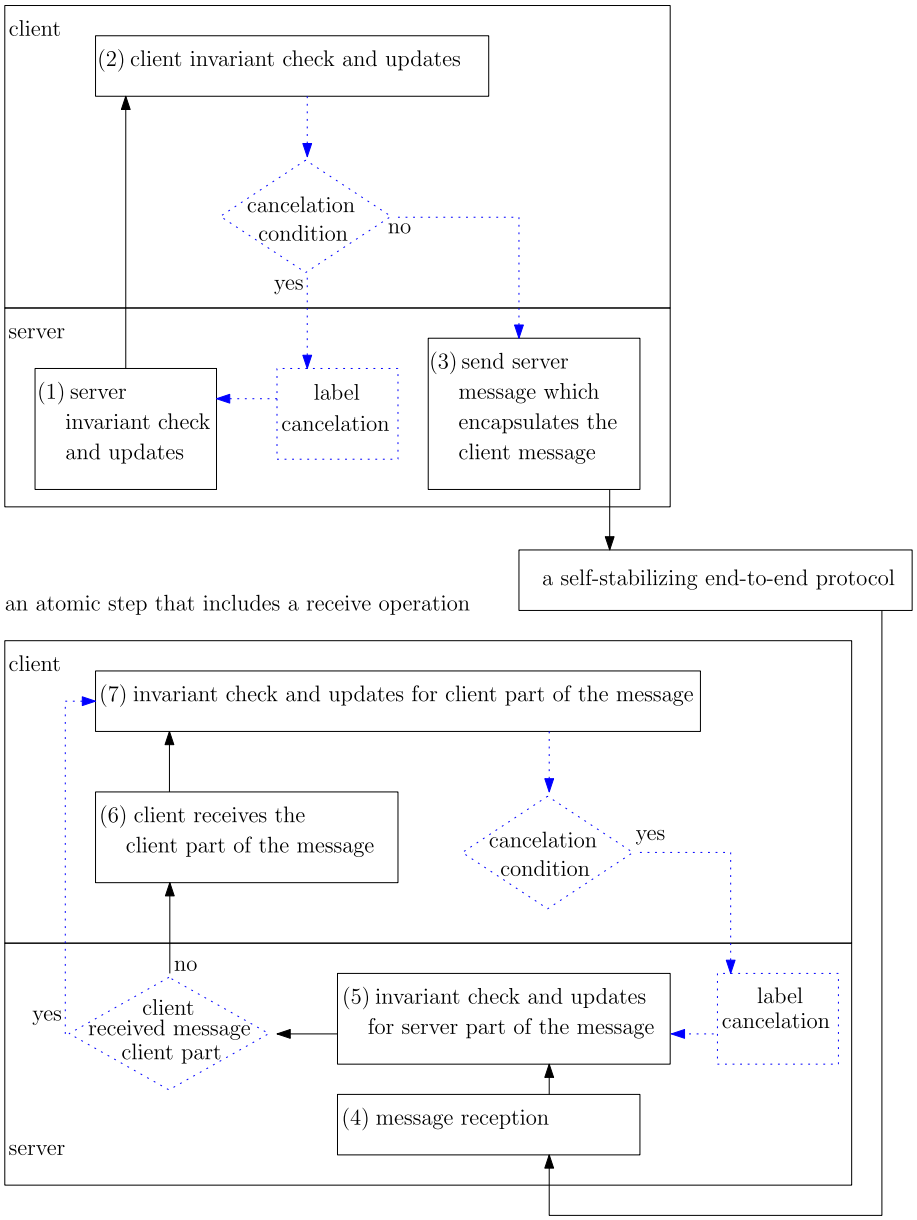


Fig. 1 Composition of the server and the client algorithms following Dolev [10, 12]'s server-client interface. The normal lines denote the composition parts that are common in both strong and practically-self-stabilizing algorithms. The dotted blue lines show the computations in the composition of practically-self-stabilizing algorithms, that are additional to the normal lines. We refer to the computations done in a step excluding the send or receive operation as the (server or client) invariant check and updates.

5.1.1 A step that includes a send

We first explain the computations of the compound algorithm during a step that ends with a send operation. This step starts with the server algorithm's invariant check and updates, which is followed by the client algorithm's invariant check and updates (parts 1 and 2 of Fig. 1, respectively). We assume that the client algorithm can request a change in the labeling (server) algorithm's state, *e.g.,* a label cancelation *i.e.,* disabling obsolete labels, but this change is performed by the labeling algorithm (cf. label cancelation in Fig. 1). In case the client algorithm indeed requires a label to be canceled, the labeling algorithm cancels that label, and then the server and client invariant check and updates repeat (cf. Fig. 1). Otherwise, the server encapsulates the client's message, m_{client}, and transmits the server message $m_{server} = \langle serverPart, m_{client} \rangle$, which encodes the server and client parts of the message.

5.1.2 A step that includes a receive

Upon the arrival of a message m by the labeling (server) algorithm (part 4 in Fig. 1), the server algorithm performs the server invariant check and updates on the server part of the message (part 5 in Fig. 1). Then, the server algorithm raises a message reception event for the client algorithm (part 6 in Fig. 1), which delivers the part of the arriving message that is relevant to the client algorithm, i.e., m_{client}. In the following, the client algorithm performs the client invariant check and updates (part 7 in Fig. 1), which might include a request to change the state of the server algorithm, e.g., by canceling a label. If that is the case, the labeling algorithm cancels that label, and then parts 5 and 7 of Fig. 1 repeat.

5.2 Challenges in the composition of practically-stabilizing algorithms

Composing practically-self-stabilizing algorithms is not always identical to composing strong self-stabilizing algorithms (cf. [10, Section 2.7]). An infinite execution is *fair* [10] if all processors take steps infinitely often (hence no processor crashes).

In this paper we allow processor crashes, i.e., the executions are not fair, in contrast to strong self-stabilization. Moreover, when composing two self-stabilizing algorithms, we assume that the client algorithm does not change the state of the server algorithm. However, labels can become obsolete (canceled), *e.g.,* due to an overflow event of a counter in the client algorithm. Thus, a step of the client algorithm might include requesting the labeling (server) algorithm to change its state by canceling a label (cf. Fig. 1).

Unlike strong self-stabilization (Definition 3.1), which often requires fairness assumptions, the proposed composition tolerates crashes, where fairness is not guaranteed. A key challenge arises from the potential for state dependency between the client and server algorithms, especially when label obsolescence or counter overflows lead to state changes in the server. Despite these challenges, the proposed approach provides an appropriate framework for constructing practically-self-stabilizing systems that can recover from transient faults, making it suitable for asynchronous systems with crash-prone components.

5.3 The interface to Dolev et al.'s labeling [14, Algorithm 2]

The proposed composition uses functions to define the interface between the server and client. They play specific roles in managing Dolev *et al.* 's labeling scheme, where each function interacts to maintain label validity, manage updates, and handle cancellations. The scheme

relies on two main principles: (1) allowing the server algorithm to perform updates and do its invariant testing, and (2) allowing the client algorithm to query the server state without modifying it, except for the function $cancel()$. Recall that the function $cancel(\ell)$ cancels label ℓ and returns no value, *i.e.,* it marks the obsolete label ℓ as disabled.

We also explain how the Dolev *et al.*'s labeling algorithm [14, Algorithm 2] implements the functions of this interface. These functions are also used by the shared counter algorithm in [14, Algorithm 3]. We note that [14, Algorithm 3] relies on synchronization mechanisms, but the labeling algorithm [14, Algorithm 2] does not rely on synchronization mechanisms, and hence it is suitable for our solution.

In the following, we use ℓ and ℓ' to denote Dolev *et al.*'s labels. The relation $\prec_{lb}$ represents the label ordering as defined by the protocol [14]. Processors are indexed by p_i and p_j, with P as the set of all processors. Each processor tracks the largest received label, $sentMax$, which is included in outgoing messages to ensure eventual consistency. As in [14], the variable $max_i[i]$ denotes the largest label stored by processor p_i.

5.3.1 Server updates and invariant testing

- $labelBookkeeping()$*: server invariant check and updates.* This procedure takes no arguments. It allows the labeling algorithm to perform its invariant check and updates, i.e., the step's computations excluding the send or receive operation (part 1 or parts 4 and 5 in Fig. 1). It is intended to be called in every step of the client algorithm, and thus facilitates the composition of the two algorithms.
  In [14, Algorithm 2], when calling $labelBookkeeping()$ without arguments, [14, Algorithm 2, lines 21 to 28] perform the server invariant check and updates (part 1 of Fig. 1). When calling $labelBookkeeping(m, j)$, [14, Algorithm 2, lines 19 to 28] process a message m that arrived from a processor $p_j \in P$ (parts 4 and 5 of Fig. 1). The (mutable) function $labelBookkeeping()$ is an alias to $process()$ [14, Algorithm 3, line 2]. We clarify that, in the context of this work, a *mutable function* may modify the state of the algorithm that executes it, *i.e.,* [14, Algorithm 2]. This contrasts with *immutable functions*, which do not alter the algorithm's state.
- $isStored()$ *and* $isCanceled()$*: querying whether a label is stored or canceled.* Given a label ℓ, the (immutable) predicate $isStored(\ell)$ checks whether ℓ appears in the label storage of the labeling algorithm. We clarify that immutable functions cannot change the state of the algorithm that executes them or the function arguments. The (immutable) predicate $isCanceled(\ell)$ checks whether ℓ is canceled.
  In [14, Algorithm 2], $isStored(\ell)$ returns true, if and only if it holds that $\ell \in storedLabels[j]$, such that $p_j \in P$ is ℓ's creator. Also, $isCanceled()$ is an exact alias to $legit()$ in [14, Algorithm 2, line 6].
- $getLabel()$*: retrieving the largest label.* The (immutable) function $getLabel()$ takes no arguments and returns the largest locally stored label.
  In [14, Algorithm 2], $getLabel()$ returns the largest locally stored label with respect to the labeling relation $\prec_{lb}$, where $\prec_{lb}$ is the relation among labels that Dolev et al. define [14]. In detail, the label returned from $getLabel()$ is stored in $max_i[i]$ (cf. lines 27 and 28 of [14, Algorithm 2]). For executions that belong to E_{AT}, $\ell \prec_{lb} max_i[i]$ holds for any label ℓ that is in the current system state and p_i has ever seen. (See [14] for the guarantees that relation $\prec_{lb}$ provides.)

5.3.2 Querying the server state without modifying it

- *$legitMsg()$ and $encapsulate()$: Token circulation and message encapsulation.* The $legitMsg()$ function enables a token circulation mechanism for the labeling algorithm, which is part of the self-stabilizing end-to-end protocol (cf. Fig. 1). It takes a (server) message and returns a Boolean. The token circulation mechanism guarantees that for two processors $p_i, p_j \in P$, p_i processes an incoming message from p_j only if p_j has received p_i's local maximal label. To that end, p_j piggybacks the last received maximal label of p_i, $sentMax$, to every message m_{server} that it sends to p_i.
  The function $encapsulate()$ facilitates piggybacking of a message of the labeling algorithm with the one of the composed (client) algorithm (facilitating part 3 of Fig. 1). It takes a (client) message and returns a (server) message. That is, the (immutable) function $encapsulate(m_{client})$ returns a message m_{server}, such that the labeling (server) algorithm's message encapsulates the message m_{client}.
  Let $serverPart = \langle sentMax, \bullet \rangle$ and m_{client} be the server, and respectively, the client part of an outgoing message of the compound algorithm. In [14, Algorithm 2], the server message is $m_{server} = \langle \langle sentMax, \bullet \rangle, m_{client} \rangle$ [14, Algorithm 2] and $encapsulate(m_{client})$ returns the value m_{server}. Moreover, the (immutable) function $legitMsg(m_{server}, \ell)$ tests the consistency of an arriving label ℓ with $serverPart$ of the server message m_{server}. That is, the predicate $legitMsg(m_{server}, \ell)$ returns the value of $\ell = sentMax$.
- *$cancel()$: canceling a label.* This is a function that the client algorithm uses to request the labeling algorithm to cancel a label, e.g., upon an overflow event. As mentioned, it takes a single argument, label ℓ, and returns no value. In contrast to the functions presented above, $cancel()$ is the only function that the client algorithm can use to change the state of the labeling (server) algorithm (cf. Fig. 1). Let ℓ and ℓ' be two labels, such that ℓ' cancels ℓ according to the labeling scheme. Then, when the client algorithm calls the (mutable) function $\ell.cancel(\ell')$, the labeling algorithm marks ℓ as canceled (by ℓ'). In [14, Algorithm 2], in case $\ell' \prec_{lb} \ell$ holds, p_i marks ℓ as canceled by ℓ' by calling $\ell.cancel(\ell')$ (cf. label cancelation definition in [31, Section 3]). In detail, the function $cancel()$ is an alias to $cancelExhausted()$ [14, Algorithm 3, line 10].

5.4 Preserving the guarantees of the labeling algorithm

We note that during a subexecution in which the client algorithm does not call the function $cancel()$, the approach for algorithm composition of this section is along the lines of the one in [10, Section 2.7] (cf. Fig. 1). However, the function $cancel()$ changes the state of the labeling algorithm. Thus, it is necessary for the stabilization proof of the compound algorithm, i.e., the composition of the labeling and client algorithms, to show that the stabilization guarantees of the labeling algorithm are preserved. The (client) algorithm that we propose in Sect. 7 (Algorithm 2) for the vector clock problem is composed with the labeling algorithm of Dolev et al. [14, Algorithm 2] through the interface that we presented in this section. In Sect. 8 we show that the algorithm that we propose for the vector clock problem preserves the labeling algorithm's stabilization guarantees.

6 Vector clock pairs

The data structure of vector clocks is used to capture causal relationships between events. Each processor maintains a vector of counters to record its event counts and those received from others. However, a single transient fault can corrupt the most significant bits of the vector's counters. Once that happens, the vector clock can quickly face counter overflows when the counters reach their maximum value. To address this, we introduce a *vector clock pair* construction, which consists of two components that work together to tolerate counter overflows and maintain accurate causality tracking in practically-self-stabilizing systems.

The key idea behind the vector clock pair is to maintain two versions of the clock: the *current* clock, which tracks the ongoing counts of events, and the *previous* clock, which serves as a reference for handling overflows. Using this dual structure, the system can handle counter overflows, ensuring that causality relationships remain valid even when counters exceed their limits. The main roles of the components are as follows:

- **Current clock** ($curr$): This part holds the vector clock values for the current set of events.
- **Previous clock** ($prev$): This part serves as a reference, allowing the system to track differences between the previous and current states when overflows occur.

The vector clock pair is essential because a single clock would lack the necessary information to recover from counter overflows. The dual structure ensures that even when counters wrap around, the system can eventually recover correctly after a transient fault's last occurrence. Moreover, in the absence of transient faults (and after recovery), we can compute causality by referring to the values stored in $curr$.

In the rest of this section, we define a vector clock pair, which is a construction for emulating a vector clock that can tolerate counter overflows. We define the invariants and conditions that should hold for the vector clock pairs with respect to requirements 1 and 2. We show how to merge two (vector clock) pairs (Sect. 6.6), and use this construction for counting the events of a single processor and computing the query *causalPrecedence*(), which we defined in Sect. 3 (Sect. 6.7). In Sect. 7 we use the vector clock pairs for designing a practically-self-stabilizing algorithm with respect to the abstract task that requirements 1 and 2 define (cf. Section 3).

6.1 The (vector clock) pair

We say that $I = \langle \ell, m, o \rangle$ is a *(vector clock) item*, where ℓ is a label of the Dolev et al. [14] labeling scheme, m (main) is an N-size vector of integers that holds the processor increments, and o (offset) is an N-size vector of integers that the algorithm uses as a reference to m's value upon ℓ's creation. We use $(I.m - I.o) (\text{mod } MAXINT)$ for retrieving I's vector clock value. We define a *(vector clock) pair* as the tuple $Z = \langle curr, prev \rangle$, where both $curr$ and $prev$ are vector clock items, such that $Z.curr.o = Z.prev.m$, *i.e.,* two variable names that refer to the same storage (memory cell). We use $VC(Z) := (Z.curr.m - Z.curr.o) (\text{mod } MAXINT)$ for retrieving Z's vector clock. We assume that each processor p_i stores a vector clock pair $local_i$ and we explain below how p_i uses $local_i$ for counting local events as well as events that it receives from other processors, even when (concurrent) counter overflows occur.

6.2 Starting a vector clock pair

The first value of a pair Z is equal to $\langle\langle \ell, zrs, zrs\rangle, \langle \ell, zrs, zrs\rangle\rangle$, where $\ell := getLabel()$ is the local maximal label and $zrs := (0, \ldots, 0)$ is the zero vector. That is, the vector clock value of Z is an N-sized vector of zeros, *i.e.,* $VC(Z) = zrs$, that we associate with the local maximal label.

6.3 Exhaustion of vector clock pairs

We say that a pair Z is *exhausted* when Condition 1 holds. Condition 1 defines exhaustion when the sum of the elements of the vector clock's value $VC(Z)$ is at least $MAXINT - 1$. Note that defining exhaustion according to the sum of the vector clock's values reduces the exhaustion events, in comparison to defining exhaustion for every vector clock element overflow, *i.e.,* for every $curr.m[i]$, $p_i \in P$. The latter also justifies the use of one label for a vector clock item I, instead of N labels, *i.e.,* one per each element of $I.m$. Since the size of a label $I.\ell$ in the Dolev et al. labeling scheme [14] is in $\mathcal{O}(N^3)$, this linear improvement is significant.

$$\begin{aligned} & exhausted(Z) \\ & \Longleftrightarrow \\ & \Sigma_{k=1}^{N}(Z.curr.m[k] - Z.curr.o[k]) \geq MAXINT - 1 \end{aligned} \tag{1}$$

6.4 Reviving a (vector clock) pair

When the (vector clock) pair Z is exhausted (Condition 1), p_i *revives* Z by (i) canceling the labels of Z, *i.e.,* $Z.curr.\ell$ and $Z.prev.\ell$, and (ii) replacing Z with $Z' = \langle\langle getLabel(), Z.curr.m, Z.curr.m\rangle, Z.curr\rangle$. Hence, the value of the new vector clock, Z', is an N-sized vector of zeros, *i.e.,* $VC(Z') = Z.curr.m - Z.curr.m = (0, \ldots, 0)$ and Z' has the current offset field, $Z'.curr.o$, that refers to the same main values as the ones recorded by $Z.curr.m$ (and $Z'.curr.o$ alias value, which is $Z'.prev.m$). As we show in this section, the fact that $Z'.prev$ stores the value of $Z.curr$ upon exhaustion enables counting local events, as well as, merging (vector clock) pairs even upon concurrent exhaustions in different processors.

6.5 Incrementing vector clock values

Processor $p_i \in P$ increments its (vector clock) pair, Z, by incrementing the i^{th} entry of Z's current item, *i.e.,* it increments $Z.curr.m[i]$ by one. The new value of the vector clock is $VC(Z) = (Z.curr.m + idV(i) - Z.curr.o) \pmod{MAXINT}$, where $idV(i)$ is an N-size vector with zero elements everywhere, except for the i^{th} entry which is one, and $Z.curr.m$ is the value before the increment. In case that increment leads to exhaustion (Condition 1), p_i has to revive the pair Z. We assume that a processor can call $increment()$ only before it starts the computations of a step that ends with a send operation, to ensure that increments are immediately propagated to all other processors.

6.6 Merging two vector clock pairs

We present a set of invariants for a single (vector clock) pair as well as for two pairs. We explain when it is possible to merge two pairs and present the merging procedure. Our approach is based in finding a common label and offset in the items of the two pairs, which works as a common reference.

6.6.1 Pair label relation

Given a pair $Z = \langle curr, prev \rangle$, we say that its elements are *in order* when Condition 2 holds. That is, either the current label of a pair Z, $Z.curr.\ell$, is larger than the previous label, $Z.prev.\ell$, and $Z.prev.\ell$ is canceled, or the labels are equal and not canceled (Condition 2).

$$\begin{aligned} &labelsRelation(Z) \iff \\ &((Z.prev.\ell \prec_{lb} Z.curr.\ell \wedge isCanceled(Z.prev.\ell)) \vee \\ &(Z.prev.\ell = Z.curr.\ell \wedge \neg isCanceled(Z.curr.\ell)) \end{aligned} \tag{2}$$

6.6.2 The $=_{\ell,o}$ and $<_{\ell,o}$ relations

We define the relations $=_{\ell,o}$ and $<_{\ell,o}$ to be able to compare vector clock items (and hence pairs). Let $\ell_1 = \langle ml_1, cl_1 \rangle$ and $\ell_2 = \langle ml_2, cl_2 \rangle$ be two labels of the Dolev et al. labeling scheme [14]. Recall that for $\ell = \langle ml, cl \rangle$, cl indicates if ℓ is canceled; if $cl = \perp$ then ℓ is not canceled and if $cl \neq \perp$, cl is the label that canceled ml, *i.e.,* ℓ is canceled. We say that $\ell_1 =_m \ell_2$, if and only if $ml_1 = ml_2$. In the sequel we will use $=_m$ and $=$ interchangeably when comparing labels, as the cl part is only used for notifying whether a label is canceled or not.

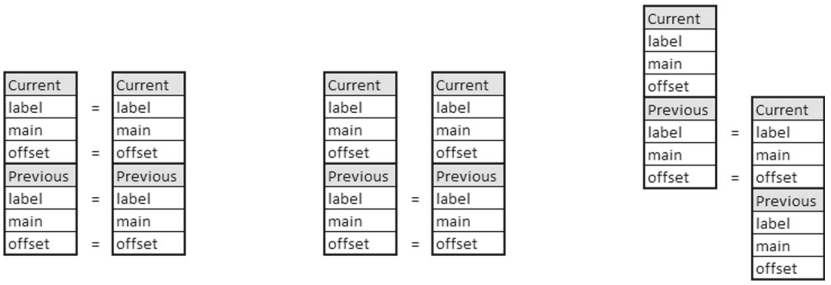
Let $\langle \ell_1, m_1, o_1 \rangle =_{\ell,o} \langle \ell_2, m_2, o_2 \rangle \iff \ell_1 = \ell_2 \wedge o_1 = o_2$. We say that two (vector clock) items z and z' *match (in label and offset)*, if and only if, $z =_{\ell,o} z'$. We use the relation $\langle \ell_1, m_1, o_1 \rangle <_{\ell,o} \langle \ell_2, m_2, o_2 \rangle \iff \ell_1 <_{lb} \ell_2 \vee (\ell_1 = \ell_2 \wedge o_1 <_{lex} o_2)$ for comparing between vector clock items, where $<_{lex}$ is the lexicographic order in $\mathbb{N}$. We define $\max_{\ell,o} \mathcal{X}$ to be the $<_{\ell,o}$-maximum item in a set of items $\mathcal{X}$ in which all labels are comparable with respect to $\prec_{lb}$ and there exists a maximum label among them.

6.6.3 Pivot existence

Condition 3 tests the pair merging feasibility (Fig. 2). It considers two pairs Z and Z' and returns true when one of the following holds:

(a) Z and Z' match (in label and offset) in their $curr$ and $prev$, *i.e.,* $Z.itm =_{\ell,o} Z'.itm$, for $itm \in \{curr, prev\}$ (Fig. 2a), *i.e.,* there was no vector clock exhaustion, or
(b) Z and Z' match in their $prev$, *i.e.,* $Z.prev =_{\ell,o} Z'.prev$ (Fig. 2b), *i.e.,* both vector clocks where exhausted (assuming that case (a) was true before exhaustion), or
(c) the label and offset in the $prev$ of one equals the label and offset in the $curr$ of the other one, *i.e.,* $Z.curr =_{\ell,o} Z'.prev \vee Z.prev =_{\ell,o} Z'.curr$ (Fig. 2c), *i.e.,* one vector clock was exhausted (assuming that case (a) was true before exhaustion).

We clarify that the proposed algorithm considers case (b) after case (a). Thus, via lazy evaluation, the algorithm considers case (a) and then case (b) (which excludes instances in case (a)), before considering case (c) (which excludes instances of cases (a) and (b)).



(a) Condition 3 holds because the two pairs match in their *curr* and *prev* items (no wrap-around, *i.e.,* due to an integer overflow).

(b) Condition 3 holds because the two pairs match in their *prev* item (the pairs had wrapped around concurrently, *i.e.,* due to an integer overflow).

(c) Condition 3 holds because $Z.prev$ and $Z'.curr$ match (Z had wrapped around, ie due to an integer overflow).

Fig. 2 Conditions for merging two given (vector clock) pairs; Z (on the left) and Z' (on the right)

We refer to the common item between Z and Z' as the pivot item (Sect. 6.6.3).

$$\begin{aligned} &existsPivot(Z, Z') \\ &\iff \\ Z.prev =_{\ell,o} Z'.prev \vee Z.curr =_{\ell,o} Z'.prev \\ \vee Z.prev =_{\ell,o} Z'.curr \end{aligned} \tag{3}$$

6.6.4 Merging (vector clock) pairs

Two vector clocks Z and Z' can be merged when there exists a pivot item, *i.e.,* $existsPivot(Z, Z')$ holds (Fig. 2). The $<_{\ell,o}$-maximum pivot item, pvt, in Z and Z', provides a reference point when merging Z and Z', because it is an item that represents a starting point from which both Z and Z' had begun counting their events. In other words, both Z and Z' were created from a common vector clock, which had pvt as its current item. We merge Z and Z' to the pair $output$ in two steps; one for initialization and another for aggregation.

We initialize $output$ to the $<_{\ell,o}$-maximum pair between Z and Z' (Fig. 2), and choose Z (the first input argument) when symmetry exists (Figs. 2a and 2b). In order to distinguish when we treat numbers and operations in $\mathbb{N}$ or in $\mathbb{Z}_{MAXINT}$, we denote by $x +_{\mathbb{N}} y$ the result of adding two numbers $x, y \in \mathbb{Z}_{MAXINT}$ in $\mathbb{N}$ ($x +_{\mathbb{N}} y$ can be possibly larger than $MAXINT$) and $x|_{\mathbb{N}}$ denotes that $x \in \mathbb{Z}_{MAXINT}$ is treated as a number in $\mathbb{N}$.

For every $i \in \{1, \ldots, N\}$, let $newEvents(X, pivot)[i]$ be the number of new events that the pair $X \in \{Z, Z'\}$ counts since the reference item, $pivot$. In Eq. 4, we compute $newEvents(X, pivot)[i]$ depending on whether $pivot$ matches $X.curr$ or $X.prev$. In the former case, we count the number of events in $X.curr.m[i]$ since the offset $X.curr.o[i]$. In the latter case, we also add the number of events in $X.prev.m[i]$ since the offset $X.prev.o[i]$, because $X.prev.o$ is the common offset of Z and Z'. We clarify that the proposed algorithm prioritizes the former case over the latter. The aggregation step sets $output[i] = \max\{newEvents(X, pivot)[i] \mid X \in \{Z, Z'\}\} + pivot[i] (\text{mod } MAXINT)$, for every $i \in \{1, \ldots, N\}$.

$$newEvents(X, pivot)[i] = \begin{cases} (X.curr.m[i] - X.curr.o[i] (\text{mod } MAXINT))|_{\mathbb{N}}, \\ \qquad \text{if } pivot =_{\ell,o} X.curr, \\ \\ (X.curr.m[i] - X.curr.o[i] (\text{mod } MAXINT))|_{\mathbb{N}} +_{\mathbb{N}} \\ \quad (X.prev.m[i] - X.prev.o[i] (\text{mod } MAXINT))|_{\mathbb{N}}, \\ \qquad \text{if } pivot =_{\ell,o} X.prev \end{cases} \tag{4}$$

6.7 Event counting and causal precedence

In this section we present our implementation of the queries about counting the events of a single active processor (requirements 1 and 2) and about causal precedence (Sect. 3), which is based on the vector clock pair construction. We explain the conditions under which we compute the query of how many events occurred in a processor p_i between the states c_x and c_y (Requirement 1) using $local_i$'s value in these two states, and present the query's computation. Then, we describe how we compute the query $causalPrecedence(local_i, local_j)$, for two vector clocks $local_i$ and $local_j$ of active processors p_i and p_j, in possibly different states (cf. Section 3).

Let $V_i^k[i]$ be the i^{th} entry of p_i's vector clock V_i in state c_k, $k \in \{x, y\}$. Requirement 1 implies that in executions that belong to $\mathsf{E_{AT}}$, the query $V_i^y[i] - V_i^x[i]$ returns the number of events that occurred in p_i between the states c_x and c_y, where c_x precedes c_y. Let $local_i^k$ be the value of $local_i$ in state c_k. The result of this query depends on the number of calls to $revive_i()$ between c_x and c_y. That is, in case there were two or more calls to $revive_i()$ between c_x and c_y, then it is not possible to infer the correct response to the query $V_i^y[i] - V_i^x[i]$ from $local_i^x$ and $local_i^y$, since these two pairs have no common pivot item (cf. Section 6.6). Otherwise, in case there was no wrap around (cf. Fig. 2a) or one wrap around (cf. Fig. 2c) between c_x and c_y, we can use $local_i.curr$, or respectively, $local_i.prev$ as pivot items to count the correct number of events in p_i. Thus, we compute the response to the query $V_i^y[i] - V_i^x[i]$ as follows:

$$V_i^y[i] - V_i^x[i] = \begin{cases} VC(local_i^y)[i] - VC(local_i^x)[i], \\ \text{if } local_i^x \text{ and } local_i^y \text{ differ only} \\ \text{on the field } curr.m \text{ (cf. Fig. 2a)} \\ \\ newEvents(local_i^y, local_i^y.prev)[i], \\ \text{if } local_i^x.curr =_{\ell,o} local_i^y.prev \\ \text{(cf. Fig. 2c)} \\ \\ \perp, \text{otherwise} \end{cases} \tag{5}$$

In Sect. 7 we propose Algorithm 2 and in Sect. 8 we show that it is practically-self-stabilizing with respect to requirements 1 and 2 (cf. Section 3). Thus, for executions that belong to $\mathsf{E_{AT}}$, the return value of $V_i^y[i] - V_i^x[i]$ in Eq. 5 is never $\perp$.

In order to compute the query $causalPrecedence(Z, Z')$, which is true if and only if Z causally precedes Z' (Sect. 3), we follow a similar approach to merging pairs (Sect. 6.6). As in the computation of the query $V_i^y[i] - V_i^x[i]$, we require that there exists a pivot item, *pivot*, between two pairs Z and Z' in order to be able to compare them, and we use the $newEvents(X, pivot)$ function to compare these pairs, $X \in \{Z, Z'\}$. We detail the computation of $causalPrecedence(Z, Z')$ in Eq. 6.

Algorithm 1 Definitions of constants, variables, interfaces, and macros that are needed for Algorithm 2

1: **Constants:** $zrs := (0, \ldots, 0)$: the N-size vector of zeros, $idV(i)$: N-size vector, where $idV(i)[i] = 1$ and $idV(i)[j] = 0$, for $j \neq i$
2: **Variables:** $pairs[]$: an N-size vector of pairs, where $pairs[i]$ is the local vector clock pair, *i.e.,* $local$ is an alias to $pairs[i]$. Also, $pairs[j]$ is the latest value of p_j's $local$ that p_i received.
3: **Interfaces:** $isStored()$, $getLabel()$, $legitMsg()$, $encapsulate()$, $cancel()$, $labelBookkeeping()$ (Section 5), $newEvents()$ (Section 6).
4: **Macros:** we use as macros conditions 1 to 3, Equation 4 (Section 6), and the following:
5: $mirroredLocalLabels() := isStored(local.prev.\ell) \wedge local.curr.\ell = getLabel()$
6: $pairInvar(X) := \neg exhausted(X) \wedge (X.prev.\ell \preceq_{lb} X.curr.\ell)$
7: $comparableLabels(\mathcal{X}) := \forall \ell, \ell' \in \{X.curr.\ell, X.prev.\ell \mid X \in \mathcal{X}\}, \ell \preceq_{lb} \ell' \vee \ell' \preceq_{lb} \ell$
8: $legitPairs(X, Y) := comparableLabels(\{X, Y\}) \wedge existsPivot(X, Y)$ (Condition 3, Section 6)
9: $restartLocal() := \{local \leftarrow \langle y, y \rangle\}$, where $y = \langle getLabel(), zrs, zrs \rangle$
10: $equalStatic(X, Y) := (X.curr.\ell = Y.curr.\ell \wedge X.curr.o = Y.curr.o \wedge X.prev = Y.prev)$

$$
\begin{aligned}
&causalPrecedence(Z, Z') \\
&\Leftrightarrow \\
&existsPivot(Z, Z') \wedge \\
&[(\forall_{i \in \{1,\ldots,N\}} newEvents(Z, pivot)[i] \leq \\
&newEvents(Z', pivot)[i]) \wedge \\
&(\exists_{j \in \{1,\ldots,N\}} newEvents(Z, pivot)[j] < \\
&newEvents(Z', pivot)[j])]
\end{aligned}
\tag{6}
$$

Our approach of including the $prev$ item in a vector clock pair allows to count events from a common reference, even when wrap around events occur. Hence, for executions of Algorithm 2 that belong to E_{AT} (Sect. 7), $causalPrecedence(Z, Z')$ as we compute it in Eq. 6 is true if and only if Z causally precedes Z'.

7 Practically-self-stabilizing Vector Clock Algorithm

We present Algorithm 2, a practically-self-stabilizing vector clock algorithm designed to meet the requirements outlined in 1 and 2 (Sect. 3). The algorithm leverages a vector clock pair construction (explained in Sect. 6) alongside a practically-self-stabilizing labeling scheme. The resulting algorithm composes this labeling mechanism through an interface (Sect. 5) that provides the necessary stabilization properties.

At a high level, Algorithm 2 operates by repeatedly updating local vector clock pairs, ensuring consistency across the processors by:

1. Incrementing vector clocks when the application layer raises local events;
2. Testing key invariants of the vector clock pair (*e.g.,* to detect clock exhaustion) and broadcasting this local pair to neighboring processors via a continuous update loop;
3. Merging incoming vector clock pairs from other processors with the local one.

Algorithm 2 includes procedures for these three operations:

- Vector clock increments trigger when local events are registered, updating the local state.

- Invariants, such as vector clock exhaustion, are repeatedly checked in each processor's loop to ensure the recovery.
- Merging operations handle the arrival of external clock information, integrating updates from other processors to maintain a consistent state across the system.

As mentioned, Algorithm 2 depends on several functions defined in Sect. 6 to achieve a practically self-stabilizing execution flow, addressing consistency and state recovery requirements. We now turn to review the algorithm details.

7.1 Local variables (line 2)

Processor $p_i \in P$ maintains a local (vector clock) pair, $local_i$, such that for any state, p_i's vector clock value is $VC(local_i)$ (cf. Section 6).

7.2 Restarting *local* via *restartLocal*() (line 9)

The macro $restartLocal()$ allows $local_i$ to revert to its initial value $\langle y, y \rangle$, where $y = \langle getLabel(), zrs, zrs \rangle$ and zrs is the N-size vector of zeros. Processor p_i invokes $restartLocal()$ to reset $local_i$ to its initial value whenever the invariants for $local_i$ are violated during the do-forever loop and in the message arrival procedures of Algorithm 2. This contributes to recovery from transient faults by ensuring that the local state can recover from any inconsistencies that may arise during execution.

7.3 Token passing mechanism for sending and receiving *local*

Algorithm 2 employs a token circulation mechanism for sending and receiving $local$, which operates independently of the algorithm's computations on $local$. This mechanism is crucial for ensuring that for eventually any two processors $p_i, p_j \in P$, processor p_j merges the information received from p_i only if p_i's message confirms that it has received the latest value of $local_j$. This helps to ensure that all processors eventually consolidate the states of their vector clocks.

Without this mechanism, there is a risk that p_i may not receive (and process) the latest value of $local_j$ from p_j for an unbounded number of steps, while p_i continues to send $local_i$ to p_j during this unbounded period. This scenario could lead to repeated calls to $restartLocal()$ at p_j, especially if the pair received from p_i cannot be merged with $local_j$ (refer to Sect. 6.6 and the message arrival procedures).

In Sect. 8, we demonstrate that invoking $restartLocal\ ()$ during an algorithm step (possibly) implies that Requirement 1 is not satisfied in the state immediately following this step. Hence, an unbounded number of calls to $restartLocal\ ()$ implies an unbounded number of states in which Requirement 1 is unmet. The token circulation mechanism is instrumental in preventing this issue.

To facilitate the token circulation mechanism, each processor p_i maintains an N-size vector of pairs, $pairs_i[]$, where $pairs_i[j]$, for $j \neq i$, is the last value of $local_j$ that p_i received (from p_j), and $pairs_i[i]$ contains p_i's own pair, thus allowing $local_i$ to act as an alias for $pairs_i[i]$. We implement the token passing mechanism by augmenting the messages sent by a processor (via $encapsulate()$) in Algorithm 2 as follows: A processor p_i sends $\langle local_i, pairs_i[j] \rangle$ to a processor p_j by invoking $encapsulate(\langle local_i, pairs_i[j] \rangle)$ in line 53. Consequently, a message sent by p_j and received by p_i is structured as

Algorithm 2 Practically-self-stabilizing vector-clock replication, code for p_i. The needed definitions of constants, variables, interfaces, and macros appear in Algorithm 1.

```
11: procedure cancelPairLabels(Z)
12:     for all ℓ ∈ {Z.curr.ℓ, Z.prev.ℓ} do
13:         ℓ.cancel(ℓ); labelBookkeeping();
14:     end for
15: end procedure

16: procedure revive(Z)
17:     cancelPairLabels(Z);
18:     return ⟨⟨getLabel(), Z.curr.m, Z.curr.m⟩, Z.curr⟩
19: end procedure

20: procedure increment()
21:     let local = ⟨⟨local.curr.ℓ, (local.curr.m + idV(i)) (mod MAXINT), local.curr.o⟩, local.prev⟩;
22:     if exhausted(local) then
23:         local ← revive(local)
24:     end if
25: end procedure

26: procedure merge(loc, arr)
27:     if ∃_{x∈{curr, prev}} loc.curr =_{ℓ,o} arr.x then
28:         let pivot := loc.curr.o
29:     else
30:         let pivot := loc.prev.o
31:     end if
32:     if arr.curr ≤_{ℓ,o} loc.curr then
33:         let output := loc
34:     else
35:         let output := arr
36:     end if
37:     for all k ∈ {1, . . . , N} do
38:         output.curr.m[k] ← (pivot[k] + max NewEvents)(mod MAXINT)
39:         where NewEvents = {newEvents(Z, pivot)[k] | Z ∈ {loc, arr}}
40:     end for
41:     return output
42: end procedure

43: procedure DO- FOREVER LOOP
44:     while True do
45:         labelBookkeeping()
46:         if ¬(mirroredLocalLabels() ∧ labelsRelation(local)) then
47:             restartLocal()
48:         end if
49:         if exhausted(local) then
50:             local ← revive(local)
51:         end if
52:         for all p_k ∈ P \ {p_i} do
53:             send encapsulate(⟨local, pairs[i]⟩) to p_k
54:         end for
55:     end while
56: end procedure

57: procedure UPON MESSAGE m = ⟨•, ⟨arriving, rcvdLocal⟩⟩ ARRIVAL FROM p_j
58:     labelBookkeeping(m, j)
59:     pairs[j] ← arriving
60:     if equalStatic(local, rcvdLocal) ∧ legitMsg(m, arriving.curr.ℓ) ∧ pairInvar(arriving) then
61:         if ¬legitPairs(local, arriving) then
62:             restartLocal()
63:         else
64:             local ← merge(local, arriving)
65:             if exhausted(local) then
66:                 local ← revive(local)
67:             end if
68:         end if
69:     end if
70: end procedure
```

$m_j = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ (line 57). Processor p_i stores $arriving_j$ in $pairs_i[j]$ (line 59) to ensure it possesses the latest value of $local_j$. Thus, processor p_i processes the message m_j if the pairs $local_i$ and $rcvdLocal_j$ are equal or differ only in their $curr.m$, as merging conditions (cf. Section 6.6) are independent of $curr.m$. We detail the precise procedures for sending and receiving messages in Algorithm 2 in the last part of this section. In Sect. 8, we illustrate that the token passing mechanism is self-stabilizing, requiring at most $\mathcal{C}N^2$ steps.

7.4 The function *revive*() (lines 16–18)

When the pair Z is exhausted (Condition 1), a call to $revive(Z)$ allows Z to wrap around and return its new version (refer to Sect. 6). Specifically, processor p_i cancels the labels of Z's components, $Z.curr.\ell$ and $Z.prev.\ell$, by executing the labeling algorithm (function *cancelPairLabels*, lines 11–13), and then sets $Z.curr$ to the output pair's $prev$ and $\langle getLabel(), Z.curr.m, Z.curr.m \rangle$ as the new output's $curr$.

7.5 The increment function, *increment*() (lines 20–23)

When p_i invokes $increment()$, it increments the i^{th} entry of p_i's vector clock. Specifically, p_i increments $local_i.curr.m[i]$ by 1 by adding $idV(i)$ to $local_i.curr.m$, where $idV(i)$ is an N-size vector with zero elements everywhere, except for the i^{th} entry which is 1 (line 21). If this increment leads to vector clock exhaustion, it subsequently calls the function $revive()$ (line 23). We assume that a processor can only call $increment()$ at the beginning of a step that concludes with a send operation (refer to the paragraph on Algorithm 2's do-forever loop below), and this call is integral to the step. This constraint guarantees that vector clock increments are promptly sent to all other processors.

7.6 Aggregation of pairs using merge() (lines 26–41)

The function $merge(Z, Z')$ (lines 26–41) aggregates two pairs, Z and Z', including the local one and another received via the network. It produces an output pair $output$ with the $\prec_{\ell,o}$-maximum items, encompassing the aggregated number of events from both Z and Z' (refer to Sect. 6).

The function employs the $\prec_{\ell,o}$-maximum pivot item x from Z and Z', from which it counts the new events in both pairs (line 27). It initializes the output pair, $output$, with the input pair that is $\prec_{\ell,o}$-maximum in both $curr$ and $prev$ (line 35). The algorithm then updates $output.curr.m$ with the maximum number of new events between $Z.curr.m$ and $Z'.curr.m$ since the pivot item (lines 37–38), and returns $output$ (line 41). Specifically, the output is computed as $output.curr.m[i] = \max\{newEvents(X, pivot)[i] \mid X \in \{Z, Z'\}\} + pivot[i] \pmod{MAXINT}$ for every $i \in \{1, \ldots, N\}$.

7.7 The do-forever loop and message arrival

We explain Algorithm 2's do-forever loop (lines 43–53) and message arrival procedure (lines 57–66), which follow the algorithm composition of Fig. 1 (Sect. 5).

7.7.1 The do-forever loop (lines 43–53)

The do-forever loop starts by letting the labeling algorithm take a step in line 45 (part 1 of Fig. 1). Line 47 refers to the invariants of $local$. Algorithm 2 calls $restartLocal()$ in line 47, in case one of the following does not hold: (i) $local.curr.\ell$ is not the local maximal label or $local.prev.\ell$ is not stored in the labeling algorithm's storage, *i.e.,* if *mirroredLocalLabels*() is false (line 5), or (ii) Condition 2 is false, *i.e., labelsRelation*($local$) is false. In line 50, the algorithm checks if $local$ is exhausted and in the positive case, $local$ wraps around to the return value of $revive(local)$ (cf. line 9). Lines 47–50 refer to part 2 of Fig. 1.

In line 53 the processor sends $local$ to every other processor in the system. The processor sends the message $m_{client} = \langle local, pairs[j] \rangle$ to every $p_j \in P \setminus \{p_i\}$, by calling $encapsulate(m_{client})$. The pair $pairs[j]$ is appended due to the token circulation mechanism. Line 53 refers to part 3 of Fig. 1.

7.7.2 Message arrival (lines 57–66)

Upon arrival of a message $m = \langle \bullet, \langle arriving, rcvdLocal \rangle \rangle$ from processor p_j (part 4 of Fig. 1) the labeling algorithm processes its own part of m (part 5 of Fig. 1) by the call to $labelBookkeeping(m, j)$ in line 58. In lines 59–66 of the message arrival procedure, Algorithm 2 processes $\langle arriving, rcvdLocal \rangle$ (parts 6 and 7 of Fig. 1). In line 59 the algorithm stores $arriving$ to $pairs[j]$, *i.e.,* the latest pair that p_i received from p_j, to facilitate the token passing mechanism.

Algorithm 2 proceeds in processing $arriving$ only if $equalStatic(local, rcvdLocal) \wedge legitMsg(m, arriving.curr.\ell) \wedge pairInvar(arriving)$ holds (cf. line 60). Let $rcvdLocal$ be the pair that p_j had received from p_i immediately before the step in which it sent the message m to p_i. The predicate $equalStatic(local, rcvdLocal)$ (line 10) is true, if $rcvdLocal$ either equals $local$ or differs from $local$ only in $curr.m$ (in case until the reception of m, p_i incremented its vector clock pair, without exhausting it).

Recall that the part of m that refers to the labeling algorithm includes p_j's local maximal label, which should be equal to $arriving.curr.\ell$ (cf. predicate *mirroredLocalLabels*() in line 47). The predicate $legitMsg(m, arriving.curr.\ell)$ (cf. Section 5) is true if $arriving.curr.\ell$ is equal to p_j's local maximal label as it appears in the part of m that refers to the labeling algorithm. The predicate $pairInvar(arriving)$ (line 6) is true if $arriving$ is not exhausted (Condition 1) and $arriving.prev.\ell \preceq_{lb} arriving.curr.\ell$ holds. Hence, if $legitMsg(m, arriving.curr.\ell) \wedge pairInvar(arriving)$ is false, m contains stale information and existed in the system in the starting system state.

In case the condition of line 60 holds, the algorithm attempts to merge the arriving pair with the local one. Merging is feasible if $legitPairs(local, arriving)$ holds. The predicate $legitPairs(X, Y)$ (line 8) is true if and only if the value of $comparableLabels(\{X, Y\}) \wedge existsPivot(X, Y)$ is true. That is, all the labels of the pairs X and Y must be comparable with respect to the relation of the labeling scheme and there exist a pivot item between X and Y (Condition 3, Sect. 6).

We clarify that during the period in which the system recovers from the occurrence of transient faults, it might be the case that neither $\ell" \preceq_{lb} \ell'$ nor $\ell' \preceq_{lb} \ell"$ holds. In this case, one must not compare the labels ℓ' and $\ell"$. In fact, the underlying labeling algorithm by Dolev et al. [14, Algorithm 2] will eventually create a new label in order to recover from this state corruption. This new label is guaranteed to be comparable to both ℓ' and $\ell"$.

In case $legitPairs(local, arriving)$ is false, the algorithm calls the function $restartLocal(local)$ (line 62), since merging must be possible in executions that belong to E_{AT}, *i.e.,*

the arriving information must be the result of a transient fault. Otherwise, merging $local$ and $arriving$ is feasible, and thus the algorithm lets $local$ to have the return value of $merge(local, arriving)$ (line 64). In case the new pair value of $local$ is exhausted, $local$ wraps around to the return value of $revive(local)$ in line 66 (cf. line 9).

7.7.3 Algorithms' composition

Note that in case of pair exhaustion Algorithm 2 forces the repetition of parts 1 and 2 of Fig. 1 corresponding to the do-forever loop procedure, as well as, parts 5 and 7 of Fig. 1 corresponding to the message arrival procedure. That is, the algorithm requests the cancelation of $local$'s labels by the labeling algorithm, the labeling algorithm cancels these labels, and the call to $labelBookkeeping()$ provides a new local maximal label (cf. lines 11–18). Then, Algorithm 2 stores the return value of $revive(local)$ in $local$ (line 50 or 66). Thus, if $local$ is not exhausted during a step (line 50 or 66), the composition of the labeling and the vector clock algorithm is along the lines of [10, Section 2.7]. The latter holds, since Algorithm 2 changes the state of the labeling algorithm only when it calls $cancel()$ and this occurs only upon a call to $revive()$ (due to pair exhaustion). Also, this repetition of step parts occurs at most once per step, since the output pair of $revive(local)$ is by definition not exhausted (cf. line 18 and Sect. 6). Moreover, in case the invariants for $local$ do not hold in line 47 or 62, the call to $restartLocal()$ in these lines does not change the state of the labeling algorithm, since it only retrieves the local maximal label through $getLabel()$ (cf. Section 5).

8 Correctness proof

Before presenting the detailed correctness arguments, we briefly recall how the proof obligations of this section relate to the problem formulation in Sects. 1.1 and 4.2. Section 1.1 introduces the classical purpose of vector clocks, namely to count local events and to allow processors to infer causality from the information exchanged in messages. Section 4.2 formalizes these objectives through Requirements 1 and 2, which specify when a system state constitutes a correct vector-clock configuration.

In this section, we show that Algorithm 2 satisfies this formal specification by proving that, after the last transient fault, the execution contains only a bounded number of states that violate the requirements of the abstract task.

Thus, the lemmas in this section collectively bridge the intuitive properties of vector clocks described in Sect. 1.1 with the precise task specification of Sect. 4.2, thereby establishing the event-counting and causality-tracking guarantees expected from vector clocks.

8.1 The proof in a nutshell

We show that Algorithm 2 is practically-self-stabilizing (Definition 3.2). Recall from Sect. 3 that the number of system states in which active processors in an execution R deviate from the abstract task is denoted by f_R. For the vector clock abstract task, f_R denotes the number of system states in R, in which Requirement 1 does not hold, with respect to the active processors in R. Thus, in Theorem 8.1 we show that Algorithm 2 is practically-self-stabilizing due to the fact that for every $MAXINT$-sized execution R, $f_R = poly(N)$, hence $\frac{f_R}{|R|}$ is almost zero (cf. Section 3).

Theorem 8.1 (Algorithm 2 is practically-self-stabilizing) *For any MAXINT-sized execution R of Algorithm 2, $f_R = poly(N)$.*

To the end of proving Theorem 8.1, we first present a set of invariants both for the state of a single active processor and also when considering the states of all active processors in an execution (Sect. 8.4). Given these invariants, we present the conditions for an execution to belong to E_{AT} (Sect. 8.4). More specifically, we show that an execution is in E_{AT} if, (i) there are no steps that include a call to $restartLocal()$, and (ii) for each processor, there is at most one step in which that processor calls the function $revive()$. In Sect. 8.5 we study the functions that *cause* a call to $restartLocal()$ or $revive()$. That is, we define a notion of *function causality*, which bases on the interleaving model (cf. Section 3). Then, in Sect. 8.6, we prove that for every *MAXINT*-sized execution R, the number of steps that include a call to either $restartLocal()$ or $revive()$ is in $poly(N)$, and combine the above to prove Theorem 8.1 (Corollary 8.24).

Our proof also requires to show that the labeling algorithm by Dolev et al. [14] remains practically-self-stabilizing (Sect. 8.3), even if we use a larger, but yet bounded number of labels, by extending the size of the label storage, *i.e.,* $storedLabels_i$, for each $p_i \in P$.

8.1.1 Notation

We refer to the values of variable X at processor p_i as X_i. Similarly, $f_i()$ refers to the returned value of function $f()$ that processor p_i executes. Throughout the proof, any execution is an execution of Algorithm 2. Let $M = \mathcal{C}N(N-1)$ be the maximum number of messages, and hence pairs, that can exist in the communication channels in any system state, *i.e.,* $N(N-1)/2$ links, where each link is a bidirectional communication channel of capacity $\mathcal{C}$ in each direction. Moreover, recall that $P(R) \subseteq P$ is the set of processors that take steps during an execution R. When referring to a value Z_x that a variable takes, *e.g.,* $local_i$, we treat Z_x as an (immutable) literal, *i.e.,* a value that does not change.

8.2 Proof roadmap

This roadmap explains how the lemmas and theorems in Sect. 8 establish that Algorithm 2 satisfies the task requirements of Sect. 4.2 in the sense of practically-self-stabilizing systems. Concretely, we show that Requirement 1 (counting all events) and Requirement 2 (propagation of information) hold in all but a bounded number of states, by proving that, after the last transient fault, the execution belongs to the abstract task set E_{AT} for all but $O(N^8 \cdot C)$ states (Theorem 8.1), which is the core condition required for practical self-stabilization.

We complement the high-level intuition given in Sect. 8.1 by detailing below how each group of lemmas contributes to proving Requirements 1 and 2. Our argument proceeds in four layers that directly correspond to Requirements 1 and 2 and to the composition with the labeling scheme.

1. **Composition and bounded resources.** Sect. 8.3 reuses the bounds on the labeling subroutine from Sect. 5 and lift them to our setting. Corollaries 8.1 and 8.2 bound the number of labels each processor may need to store, and Corollary 8.3 (an extension of Alon et al. [2] and Dolev et al. [14, Algorithm 2]) shows that the labeling layer is practically self-stabilizing under these bounds. This ensures bounded memory usage and guarantees that the labeling service behaves as required by Algorithm 2.

2. **Requirement 1: counting all events.** Sect. 8.4 defines the local and global invariants that characterize when an execution is in E_{AT} with respect to Requirement 1. In particular, Definition 8.4 introduces `localInvariants(i)`, whose violation triggers a local recovery via `restartLocal()` (Algorithm 2, line 47). The lemmas that follow show: (i) when these invariants hold, the value of $VC(\cdot)$ reflects exactly the number of increments since the pivot as formalized in Sect. 6, hence Requirement 1 holds, and (ii) when the invariants are violated, `restartLocal()` restores them within a bounded number of steps, contributing only a bounded number of deviations.
3. **Requirement 2: propagation of information.** Sects. 6.7 to 8.6 define the merge rules for vector-clock pairs and the causalPrecedence predicate (Eq. 6), which, together with the continuous message-exchange loop of Algorithm 2 (Sect. 7), ensure that vector-clock information eventually disseminates throughout the system. The lemmas in Sect. 8.4 show that whenever `revive()` is invoked due to pair exhaustion, the merge-and-propagate mechanism restores pair consistency and resumes information flow. These arguments establish that all information that can flow *does* flow, except for a bounded number of steps in which recovery actions (such as `revive()`) temporarily interrupt normal progress. This proves that Requirement 2 holds for all but a bounded number of states in any *MAXINT*-sized execution.
4. **Bounding the total number of deviations.** Sects. 8.4.3 and 8.5 bound the total number of calls to `revive()` and `restartLocal()` over any *MAXINT*-sized execution, using the label and queue-size bounds from Corollaries 8.1 to 8.3. Combining these bounds yields Theorem 8.1, which states that the number of states that are not in E_{AT} is $O(N^8 \cdot C)$ over any practically-infinite execution. Consequently, Corollary 8.24 concludes that Algorithm 2 implements a practically-self-stabilizing solution to the vector clock task (Requirements 1 and 2).

For Requirement 1, see the invariants in Sect. 8.3 and their restoration via `restartLocal()`; for Requirement 2, see the merge and dissemination arguments tied to `revive()` and the continuous protocol loop in Sect. 7. The final bounds appear in Sects. 8.4.3 and 8.5, culminating in Theorem 8.1 and Corollary 8.24.

8.3 Convergence of the labeling algorithm in the absence of wrap around events

We generalize the lemmas of Dolev et al. [14] that bound the number of label creations and adoptions of their labeling algorithm to accommodate for the extra number of labels of Algorithm 2, and show that the labeling algorithm converges when twice as many labels are processed, due to the fact that each pair includes two labels. Recall from Sect. 7 that if there are no calls to $revive()$ (lines 23, 50, and 66) during an execution, then Algorithm 2 does not change the state of the labeling algorithm (cf. function definition in lines 16–18). However, in the starting system state of an execution of Algorithm 2 there exist twice as many labels as in the starting system state of an execution of the labeling algorithm, due to the two labels that each pair consists of.

We extend [14, Lemma 4.3], which bounds the number of labels that were created by p_j and adopted by p_i, after p_j stopped adding labels to the system (Corollary 8.1). In Corollary 8.2, we extend [14, Lemma 4.4], which bounds the number of labels that p_i creates. We clarify that corollaries 8.1 and 8.2 refer to two different bounds. Corollary 8.1 refers to the number, w, of labels that were created by p_j and p_i needs to store (for the sake of the algorithm correctness), whereas Corollary 8.2 refers to the number, x, of labels that were created by p_i and p_i needs to store. Observe that $w < x$.

We then present Corollary 8.3 that is an implication of corollaries 8.1 and 8.2 and states that the labeling algorithm of Dolev et al. [14] remains practically-self-stabilizing given the generalized bounds presented in those corollaries. Corollary 8.3 is an extension of [14, Theorem 4.2], which shows that the labeling algorithm [14, Algorithm 2] is practically-self-stabilizing. Hence, we will use Corollary 8.3 for proving that Algorithm 2 is also practically-self-stabilizing. In Sect. 8.6, we will extend these bounds to accommodate for the extra labels created by Algorithm 2, when a processor calls the function $revive()$.

Corollary 8.1 (extension of [14, Lemma 4.3]) *Let $p_i, p_j \in P$ be two processors. Suppose that p_j has stopped adding labels to the system state, and sending these labels during an execution R. Moreover, suppose that at the system state that immediately follows the last step in which p_j stopped adding labels to the system, the number of labels that have p_j as their creator and that were adopted by any of the N processors in the system is at most $2N$, and the maximum number of labels in transit that were created by p_j is at most $2M$. Processor p_i adopts at most $2N + 2M$ labels ℓ, such that $\ell.creator = j$ and $\ell \notin storedLabels_i[j]$.*

The bound in [14, Lemma 4.3] is $N + M$, but in the setting of Algorithm 2 each pair includes two labels, hence the factor of 2. Thus, setting $|storedLabels_i[j]| = 2N + 2M$, $i \neq j$ allows the labeling algorithm to converge. In the following corollary, we denote with $max_i[i]$ the local maximal label of processor p_i, as in the labeling algorithm of Dolev et al. [14].

Corollary 8.2 (extension of [14, Lemma 4.4]) *Let $p_i \in P$ be a processor and $L_i = \ell_{i_0}, \ell_{i_1}, \ldots$ be the sequence of legitimate (not canceled) labels that p_i stores in $max_i[i]$ over an execution R, such that no counter exhaustions occur during R and $\ell_{i_k}.creator = i$, $k \in \mathbb{N}$. It holds that $|L_i| \leq 4N^2 + 4NM - 4N - 2M$ [14].*

The bound in Corollary 8.2 follows by the proof of [14, Lemma 4.4], which bounds the number of labels existing either in other processors' states or in transit, for which p_i is the label creator. For a given system state, these labels are at most $x = 2(M + \Sigma_{j \neq i}|storedLabels_i[j]|) = 2M + 2(N-1)(2N + 2M)) = 4N^2 + 4NM - 4N - 2M$ (the second equality holds by Corollary 8.1). The proposed algorithm invokes the creation of additional labels. We denote by y the upper bound on the number of labels that the proposed algorithm creates, which the proof calculates (claims 8.18 and 8.19). Thus, due to the arguments above, p_i needs to store in $storedLabels_i[i]$ a total of $L = x + y$ label.

Recall that w (Corollary 8.1) is a bound on the number of labels that p_i needs to store for p_j. Note that y is independent of w. This is because w considers the case in which p_j has stopped adding labels during R, cf. Corollary 8.1. However, y considers labels that processor p_i creates during R.

Corollary 8.3 is a straightforward extension of [14, Theorem 4.2] that also holds for the updated bounds of corollaries 8.1 and 8.2, since the proof is based on the bounds' existence, rather than the actual values of these bounds.

Corollary 8.3 (extension of [14, Theorem 4.2]) *Let R' be an infinite execution of the labeling algorithm [14, Algorithm 2]. Let R be a MAXINT-sized sub-execution of R' in which no wrap around events occur. The labeling algorithm is practically-self-stabilizing in R, given the number of label creations and adoptions in corollaries 8.1 and 8.2 as well as the updated queue lengths in $storedLabels_i$, where $p_i \in P$.*

We remark that in Sect. 8.6 we extend the queue lengths to accommodate for the extra labels that are created due to Algorithm 2, *i.e.,* when wrap-around events occur and a processor calls $revive()$.

8.4 Local and global invariants and their relation to Requirement 1

In this section we study the local and global invariants that determine if an execution is in $\mathsf{E_{AT}}$. We define the predicate $localInvariants(i)$ (Definition 8.4), which gives the local invariants for $local_i$ of a processor p_i. That is, if $localInvariants(i)$ is false in line 47, then processor p_i calls $restartLocal_i()$. We show that for all functions of Algorithm 2 that include $local_i$ of a processor p_i in their input, $localInvariants(i)$ holds (lemmas 8.5–8.8).

We also give the conditions for an execution to be in $\mathsf{E_{AT}}$. To that end, we show that Requirement 1 is possibly violated in a step where a processor calls $restartLocal()$ (Remark 8.9) and definitely violated when a processor calls $revive()$ in two or more steps in an execution (Remark 8.10). Also, we define the predicate $globalInvariants(R, c)$ for an execution R and a state $c \in R$, which gives the invariants that should hold for every active processor in R, so that no step includes a call to $restartLocal()$ in line 62. Finally, in Lemma 8.12, we prove that a subexecution is in $\mathsf{E_{AT}}$ under given conditions. In sections 8.5 and 8.6, we prove that the bounds $B_{restart}(R)$ and $B_{revive}(R)$ on the number of steps that include a call to $restartLocal()$ or $revive()$ exist for every *MAXINT*-sized execution R and show that Algorithm 2 is practically-self-stabilizing.

Definition 8.4 *(The localInvariants() predicate)* Let R be an execution of Algorithm 2, $c \in R$ be a system state, and $p_i \in P$. We say that *the local invariants hold for* p_i *in* $c \in R$, if and only if, $localInvariants(i) := mirroredLocalLabels_i() \wedge labelsRelation_i(local_i)$ (line 47) holds.

Lemma 8.5 *Let a_x be a step in R in which processor p_i calls $revive_i(local_i)$ when executing line 23, 50, or 66 and suppose that localInvariants(i) holds before p_i executes $revive_i(local_i)$. Then, localInvariants(i) also holds in the state that immediately follows a_x.*

Proof Recall that the function $revive_i(local_i)$ cancels $local_i$'s labels and returns $\langle\langle getLabel_i(), local_i.curr.m, local_i.curr.m\rangle, local_i.curr\rangle$ (line 18). Let $local_i = \langle\langle \ell_{old}, m_{old}, o_{old}\rangle, prev\rangle$ be the value of $local_i$ before p_i calls $revive_i()$ and $local_i = \langle\langle \ell_{new}, m_{old}, m_{old}\rangle, \langle \ell_{old}, m_{old}, o_{old}\rangle\rangle$, be the value of $local_i$ after p_i calls $revive_i()$.

Since $localInvariants(i) = mirroredLocalLabels_i() \wedge labelsRelation_i(local_i)$ holds for $local_i = \langle\langle \ell_{old}, m_{old}, o_{old}\rangle, prev\rangle$ (Condition 2 and line 5), the following hold for $local_i = \langle\langle \ell_{new}, m_{old}, m_{old}\rangle, \langle \ell_{old}, m_{old}, o_{old}\rangle\rangle$ (i.e., after p_i calls $revive_i()$):

(i) $isStored_i(local_i.prev.\ell) = isStored_i(\ell_{old})$ holds, since ($mirroredLocalLabels_i()$ holds for $local_i$ before calling $revive_i()$,
(ii) $local_i.curr.\ell = \ell_{new} = getLabel_i()$ holds, by $revive_i()$'s definition (lines 16–18), and
(iii) $(local_i.prev.\ell \prec_{lb} local_i.curr.\ell \wedge isCanceled_i(local_i.prev.\ell) = \ell_{old} \prec_{lb} \ell_{new} \wedge isCanceled_i(\ell_{old})$ holds, again by $revive_i()$'s definition.

□

Lemma 8.6 *Let $m_j = \langle \bullet, \langle arriving_j, rcvdLocal_j\rangle\rangle$ be a message that p_i received from p_j in step $a_i \in R$, and c, c' are system states in R, such that $(c, a_i, c', \bullet) \sqsubseteq R$. Suppose that $mirroredLocalLabels_i() \wedge labelsRelation_i(local_i) \wedge equalStatic_i(local_i, rcvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge pairInvar_i(arriving_j) \wedge legitPairs_i(local_i, arriving_j)$ hold in c. Then, the invariant localInvariants(i) holds in c', i.e., after the execution of line 64 which calls $merge_i(local_i, arriving_j)$ and updates $local_i$.*

Proof Let $m_j = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ be a message that processor p_i receives from processor p_j in step a_i and assume that $mirroredLocalLabels_i() \wedge labelsRelation_i(local_i) \wedge equalStatic_i(local_i, rcvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge pairInvar_i(arriving_j) \wedge legitPairs_i(local_i, arriving_j)$ hold in c with respect to $local_i$ and m_j. For brevity, we denote the predicate $\xi_i := isStored_i(local_i.prev.\ell) \wedge local_i.curr.\ell = getLabel_i() \wedge labelsRelation_i(local_i)$. Observe that $merge_i(local_i, arriving_j)$, initializes $output_i$ either to $local_i$ or to $arriving_j$ (lines 27–35). Then, $output_i.curr.m[k]$ is updated with $newEvents$, for each $k \in \{1, \ldots, N\}$, and the result is returned and saved to $local_i$, hence lines 37 to 38 do not change the value of ξ_i. Therefore, we show that for each of the three different cases in which a pivot exists (Fig. 2), ξ_i holds after $local_i$ is updated with $merge_i(local_i, arriving_j)$.

8.4.1 Case of Fig. 2a

In this case $local_i.itm$ and $arriving_j.itm$ match in label and offset for each $itm \in \{curr, prev\}$, and $output_i$ is initialized to $local_i$, for which ξ_i holds. Also, no new label is processed by the labeling scheme ($labelBookkeeping_i(m_j, j)$ in line 58), hence the return value of $getLabel_i()$ remains the same (in c and c') after the execution of line 58 in step a_i. Therefore, $isStored_i(output_i.prev.\ell) \wedge output_i.curr.\ell = getLabel_i() \wedge labelsRelation_i(output_i)$ holds, since in c and before p_i calls $merge_i()$ in step a_i, $local_i.itm =_{\ell,o} output_i.itm$ holds for each $itm \in \{curr, prev\}$, and $\xi_i = isStored_i(local_i.prev.\ell) \wedge local_i.curr.\ell = getLabel_i() \wedge labelsRelation_i(local_i)$ also holds.

8.4.2 Case of Fig. 2b

Let $\ell_{loc} := local_i.curr.\ell$, $\ell_{arr} := arriving_j.curr.\ell$, $\ell_{prv} := local_i.prev.\ell = arriving_j.prev.\ell$, and $\ell_{max} := \max_{\prec_{lb}}\{\ell_{loc}, \ell_{arr}\}$ in state c. Note that ℓ_{max} exists, since in c (and thus in a_i), $legitPairs_i(local_i, arriving_j)$ holds, which implies that $comparableLabels_i(\mathcal{X})$ holds, where $\mathcal{X}$ includes the labels in $local_i$ and $arriving_j$. Observe that in a_i, $merge_i(local_i, arriving_j)$ initializes $output_i$ to the $\prec_{\ell,o}$-maximum pair in both $curr$ and $prev$ between $local_i$ and $arriving_j$ (lines 27–35), i.e., the pair that stores ℓ_{max} in its $curr.\ell$. Thus, $isStored_i(output_i.prev.\ell)$ holds in c', since $isStored_i(\ell_{prv})$ and $output_i.prev.\ell = \ell_{prv}$ hold in c. Due to line 58, ℓ_{arr} is stored in the variables of the labeling algorithm in c, hence ℓ_{max} is also stored in the variables of the labeling algorithm. Therefore, by ℓ_{max}'s definition, $getLabel_i()$ returns ℓ_{max} in step a_i and after the execution of line 58, i.e., $output_i.curr.\ell = \ell_{max} = getLabel_i()$. Moreover, $mirroredLocalLabels_i()$ holds for $local_i$, after $local_i$ is updated with $merge(local_i, arriving_j)$ in line 64 during step a_i (and hence holds in c').

By the definition of the case of Figure 2b , $local_i.curr.o \neq arriving_j.curr.o$ holds in a_i. If $\ell_{arr} \prec_{lb} \ell_{loc}$, then $labelsRelation_i(local_i)$ holds in c', since $output_i.curr.\ell = \ell_{loc}$, $output_i.prev.\ell = \ell_{prv}$, and $labelsRelation_i(local_i)$ holds in c. If arr ≺lb loc, thenlabelsRelationi (locali) holds i c, since outputi .curr. = loc, outputi . prev. = prv, and labelsRelationi (locali) holds in c. Otherwise, if $\ell_{max} = \ell_{arr}$, then $output_i.prev.\ell = \ell_{prv} \prec_{lb} \ell_{max} = output_i.curr.\ell$ holds in c'. Also $isCanceled_i(\ell_{prv})$ holds in c', since either $isCanceled_i(\ell_{prv})$ holds in c or ℓ_{prv} is canceled in step a_i by the maximal label ℓ_{arr}. Therefore, $labelsRelation_i(local_i)$ holds in c'.

8.4.3 Case of Fig. 2c

In this case we also use the definitions of ℓ_{loc}, ℓ_{arr}, and ℓ_{max} from the previous case (but here ℓ_{prv} is not common for $local_i$ and $arriving_j$). If $local_i.curr.\ell = \ell_{max}$ in c, then $output_i$ is initialized to $local_i$. Thus, in the end of step a_i, $isStored_i(output_i.prev.\ell) \wedge output_i.curr.\ell = getLabel_i() \wedge labelsRelation_i(output)$ holds, since the following hold:

(i) $local_i.itm =_{\ell,o} output_i.itm$, for each $itm \in \{curr, prev\}$,

(ii) $isStored_i(local_i.prev.\ell) \wedge local_i.curr.\ell = getLabel_i() \wedge labelsRelation_i(local_i)$ holds before $merge_i()$ is called, and

(iii) line 58 does not change the return value of $getLabel_i()$. Note that this is the only case where $arriving_j.prev.\ell$ is not processed by the labeling algorithm, since it is a canceled label by p_j that either (a) exists already in the variables of the labeling algorithm (hence, $getLabel_i$ returns a larger label than $arriving_j.prev.\ell$ in p_i), or (b) it can be reused in case all processors that store it as canceled crash before it is introduced in the system by another processor. Hence, $mirroredLocalLabels_i()$ holds in c'.

Otherwise, $output_i$ is initialized to $arriving_j$, since $arriving_j.curr.\ell = \ell_{max}$. In this case, $isStored_i(output_i.prev.\ell) \wedge output_i.curr.\ell = getLabel_i()$ holds, since the following hold:

(i) $output_i.prev.\ell = \ell_{loc}$ and $isStored_i(\ell_{loc})$, and

(ii) $output_i.curr.\ell = \ell_{arr} = \ell_{max} \wedge \ell_{loc} \prec_{lb} \ell_{arr}$.

Hence $getLabel_i()$ returns ℓ_{arr} after the execution of line 58. Also, the predicate $pairInvar_i(arriving_j)$ is true, which implies that $\ell_{loc} = arriving_j.prev.\ell \preceq_{lb} arriving_j.curr.\ell = \ell_{arr} = \ell_{max}$. Thus, it either holds that $arriving_j.prev.\ell = arriving_j.curr.\ell = \ell_{arr}$ and $\neg isCanceled_i(\ell_{arr})$ (since $getLabel_i() = \ell_{arr}$), or that $arriving_j.prev.\ell = arriving_j.curr.\ell \wedge isCanceled_i(arriving_j.prev.\ell)$. Therefore, $labelsRelation_i(arriving_j)$ holds, hence $labelsRelation_i(output_i)$ is true in the end of step a_i, thus $labelsRelation_i(local_i)$ holds in c'. □

Note that during $increment_i()$, p_i changes only in $local_i.curr.m[i]$ (line 21) and the other fields of $local_i$ stay intact. In case $local_i$ is exhausted after that increment, p_i calls $revive_i()$ (line 23), hence by Lemma 8.5 the value of $localInvariants(i)$ does not change whenever p_i calls $increment_i()$. By lemmas 8.5 and 8.6 we have the following.

Corollary 8.7 *Let R be an execution, $p_i, p_j \in P$, and $c_k \in R$ be a system state, which is followed by a step in which p_i calls $revive_i(local_i)$, or $increment()$, or $merge_i(local_i, arriving_j)$. If $localInvariants(i)$ holds in c_k, then $localInvariants(i)$ holds also in c_{k+1}.*

Lemma 8.8 considers the case in which a processor calls $restartLocal_i()$ (line 47 or 62).

Lemma 8.8 *Let $local_i = \langle y, y \rangle$, where $y = \langle getLabel(), zrz, zrz \rangle$, be the value of $local_i$ after a call to $restartLocal_i()$ in line 47 or 62, in a step $a_k \in R$. Then, $localInvariants(i)$ holds for $local_i$ in the state c_{k+1} that immediately follows a_k.*

Proof Since $labelBookkeeping_i()$ is called (lines 45 and 58) before each call of $restartLocal_i()$ (lines 47 and 62) and no other function in the lines between the call to $labelBookkeeping_i()$ and $restartLocal_i()$ changes the variables of the labeling algorithm (lines 59–60), $getLabel_i()$ returns the maximal label ℓ_{max} stored by the labeling algorithm and $local_i.curr.\ell = local_i.prev.\ell = \ell_{max}$. Hence, $isStored_i(local_i.prev.\ell) \wedge local_i.curr.\ell = getLabel_i()$ hold, and therefore $mirroredLocalLabels_i()$ holds (cf. condition 2). Also, $labelsRelation_i(local_i)$ holds (condition 2), since the predicate $(local_i.prev.\ell = local_i.curr.\ell \wedge \neg isCanceled_i(local_i.curr.\ell)$ holds. □

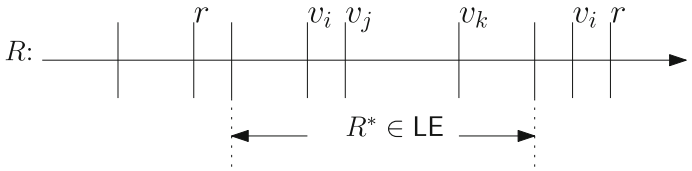


Fig. 3 Illustration of an execution that satisfies the abstract task (cf. Lemma 8.12). The horizontal line denotes an execution R and the vertical lines highlight specific steps of R. The vertical lines that are marked with r, denote a step in which a processor called $restartLocal()$. The vertical lines that are marked with v_x denote a step in which a processor p_x called $revive_x()$. In this example for the fragment of R, marked as R^*, the following hold: (i) no processor called $restartLocal()$, and (ii) p_i (and every other active processor) called $revive()$ at most once. Thus, by Lemma 8.12, R^* is an execution that satisfies the abstract task, i.e., $R^* \in \mathsf{E_{AT}}$

Conditions for an execution to be in $\mathsf{E_{AT}}$ So far in this section, we have shown that $localInvariants(i)$ holds for the output of every function of Algorithm 2 that a processor p_i applies on $local_i$. However, in order to compute queries about the number of events on a single processor between two states (Requirement 1) or to the query $causalPrecedence(local_i, local_j)$, we need to compare two vector clock pairs, i.e., pairs that appear in different processors or different system states. To that end, we present the conditions under which Requirement 1 breaks (remarks 8.9 and 8.10). Then, in Lemma 8.12 we present the conditions under which an execution is in $\mathsf{E_{AT}}$ (specifically, Requirement 1 holds), i.e., we show the conditions under which different vector clock pairs can be compared for computing correctly queries about counting events (and by Property 1 causal precedence).

Remark 8.9 *($restartLocal()$ breaks Requirement 1)* We remark that it is possible that Requirement 1 does not hold immediately after the execution of $restartLocal()$ (lines 47 and 62). Since after executing $restartLocal()$ all values in the main and offset of $local_i.curr$ and $local_i.prev$ are set to zero, it is possible to miscount events when comparing two pairs in the states of active processors. That is, p_i can miscount its own events when its entry $local_i.curr.m[i]$ is set to zero after a call to $restartLocal_i()$, except for the case when $local_i$ remains the same before and after the call to $restartLocal_i()$. □

Consider a message $m_{j,i} = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ that a processor p_i receives from a processor p_j, such that $\chi_{i,j} := equalStatic_i(local_i, rcvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge pairInvar_i(arriving_j)$ does not hold (cf. line 60). Since such messages are not processed by Algorithm 2, there is no call to $restartLocal_i()$ or $revive_i()$ in the step that p_i receives $m_{j,i}$, hence no immediate violation of Requirement 1. However, the fields of $m_{j,i}$ that refer to the labeling algorithm's part of the message are processed by the labeling algorithm in p_i in line 58. Hence, it is possible that the maximal label of p_i has changed in the step where p_i receives $m_{j,i}$, and that p_i calls $restartLocal_i()$ in its next step. In Sect. 8.6 (Lemma 8.17) we show that for every *MAXINT*-sized execution R, there exist bounds in the number of steps that include a call to $restartLocal()$ or to $revive()$ that are in $poly(N)$.

Remark 8.10 *(Two calls to $revive()$ by the same processor break Requirement 1)* Let p_i be a processor, and c_x, c_y be two states, such that there exist at least two steps between c_x and c_y in which p_i called $revive_i$. We remark that we cannot compute correctly the events that occurred between c_x and c_y by comparing $local_i^x$ and $local_i^y$, where $local_i^k$ is the value of $local_i$ in state c_k. We explain why this holds in the following.

Let c_u be the first state after (the step in which) p_i calls $revive_i()$ for the first time after c_x, and c_v be the first state after p_i calls $revive_i()$ for the first time after c_u. By the vector

clock pair construction and the definition of the function $revive()$ (Sect. 6), $local_i^u.prev$ is the value of $local_i.curr$ immediately before the first call to $revive_i()$ (after c_x). Hence, the pivot item between $local_i^x$ and $local_i^u$ is $local_i^u.prev$, i.e., $(local_i^x.curr.\ell, local_i^x.curr.o) = (local_i^u.prev.\ell, local_i^u.prev.o)$. Similarly, $local_i^v.prev$ is the value of $local_i.curr$ immediately before the first call to $revive_i()$ after c_u. Hence, the pivot item between $local_i^u$ and $local_i^v$ is $local_i^v.prev$, i.e., $(local_i^u.curr.\ell, local_i^u.curr.o) = (local_i^v.prev.\ell, local_i^v.prev.o)$. Thus, the vector clock items $local_i^x.prev$ and $local_i^x.curr$, and specifically the events that $local_i^x.curr.m$ recorded in state c_x do not appear in state c_v. Therefore, irrespective of the number of calls to $revive_i()$ between c_v and c_y, it is not possible to count the events in p_i (i.e., the calls to $increment_i()$) between the states c_x and c_y by comparing $local_i^x$ and $local_i^y$. □

In Definition 8.11 we describe the conditions under which $restartLocal()$ is never called in an execution. Then, in Lemma 8.12 we give the conditions for an execution to be in $\mathsf{E_{AT}}$. We illustrate Lemma 8.12 in Fig. 3.

Definition 8.11 *(The globalInvariants() predicate)* Let R be an execution of Algorithm 2, $c \in R$ a system state, $\varphi_i \equiv mirroredLocalLabels() \wedge labelsRelation(local_i)$ (line 47), and $\psi_{i,j} \equiv legitPairs(local_i, arriving_j)$ (line 62), where $p_i, p_j \in P$ and $m_j = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ is a message in the communication channel from p_j to p_i. We define $globalInvariants(R, c) := \forall_{p_i \in P(R), p_j \in P}\ \varphi_i \wedge (\neg \chi_{i,j} \vee \psi_{i,j})$. We say that *the global invariants hold during* R, if $globalInvariants(R, c)$ holds for every $c \in R$.

Lemma 8.12 *Let R be a MAXINT-sized execution. (I) For every subexecution R^* of R, such that*

(i) there is no step in R^* in which a processor calls $restartLocal()$, and
(ii) for every processor p_i there exists at most one step $a_x \in R^*$ in which p_i calls $revive()$ in a_x,

$R^* \in \mathsf{E_{AT}}$ holds, i.e., R^* satisfies the abstract task (Sect. 3.1.1).

Proof We prove the lemma using remarks 8.9 and 8.10.

Let R^* be a subexecution of R, such that conditions (i) and (ii) of the lemma hold. We show that Requirement 1 holds throughout R^*. Since no processor calls $restartLocal()$ in R^* (condition (i)) no event is ever lost in R^*. That is, there is no step in which for a processor p_i, $local_i.curr.m \neq zrs$ holds and p_i sets $local_i.curr.m$ to zrs by calling $restartLocal_i()$, where zrs is the zero vector (Remark 8.9). Also, since no processor calls $restartLocal()$ in R^*, $globalInvariants(R^*, c)$ holds for every $c \in R^*$. The latter implies that every message that an active processor receives in R^* is either discarded or contains a pair that is merged with the local one.

Recall that by condition (ii) of the lemma, every processor calls $revive()$ in R^* at most once. Thus, it is always possible to compute the number of events that occurred in each processor between two states in R^*. That is, the query $V_i^y[i] - V_i^x[i]$ (number of events in p_i from state c_x to state c_y), for every p_i that is active in R^*, is computed by the first two cases of Eq. 5 (Sect. 6), since there is always a pivot item between $local_i^x$ and $local_i^y$ in R^* (i.e., the return value is never $\perp$). Moreover, by the fact that $globalInvariants(R^*, c)$ holds for every $c \in R^*$, we have that it is possible to merge every two pairs in R^*. Hence, Property 1 holds (i.e., we can compute correctly the query $causalPrecedence()$), since $existsPivot()$ is true in the computation of the query in Sect. 6. □

By Lemma 8.12, we need to show that for every *MAXINT*-sized execution R the bounds $B_{restart}(R)$ and $B_{revive}(R)$ of the lemma statement indeed exist and also that $B_{restart}(R)$ and $B_{revive}(R)$ are in $poly(N)$. We do that in Sects. 8.5 and 8.6. In the end of Sect. 8.6 we complete the proof by showing that Algorithm 2 is practically-self-stabilizing.

8.5 Pair evolution graph and function causality

In this section we establish that a call to $restartLocal()$ in a step a_x of an execution R is *caused* only due to either (i) stale information that resided in the system in the starting configuration, or (ii) a call to $restartLocal()$ in a step that precedes a_x, or (iii) a call to $revive()$ in a step that precedes a_x. To that end, we define a notion of *function causality* between functions that processors call in R, which bases on a graph that relates pairs when they are either in the input or output set of a function that is called during a step of R. We refer to that graph as the *pair evolution graph* and note that it is an illustration of the interleaving model (Sect. 3).

Our aim is to highlight all the changes that occur to any pair during an execution due to functions that processors apply on these pairs in the steps they take, as well as the relations between these functions. The illustration that we bring resembles Lamport's *happened before* relation [23]. In our work, we study the events that can *cause* a call to the function $restartLocal()$, rather than just the order in which the events occur. In the following paragraphs, we gradually define the pair evolution graph by identifying the functions that can be applied on a pair, the transition from the input to the output pair when applying a function, and the pairs that appear in the system throughout an execution. We then define function causality (Definition 8.15), basing on the pair evolution graph.

Functions called during a step We start by listing the functions that a processor may call during a step. Let R be an arbitrary execution of Algorithm 2 and $(c_x, a_x, c_{x+1}) \sqsubseteq R$ be a subexecution of R, such that processor $p_i \in P$ takes step a_x. During the step a_x and by the definition of the interleaving model (Sect. 3), p_i can either

(1) run lines 43–50 and send one message (out of $N-1$) to another processor due to line 53, or
(2) continue to send messages due to line 53, but this cannot be self-sending or the message considered in item 1. That is, a_x includes the sending of one message (out of at most $N-2$ remaining messages) to another processor due to line 53, or
(3) run the message arrival procedure in lines 57–66.

After giving some insights on the send operation and $labelBookkeeping()$, we detail the functions that p_i can call during a_x.

According to the interleaving model (Sect. 3), each step includes a single send or receive operation. Hence, a complete iteration of Algorithm 2's do-forever loop (lines 43–53) requires $N-1$ steps (not necessarily consecutive), due to the $N-1$ messages to be sent to the processor's neighbors. In detail, we assume that when a processor $p_i \in P$ runs line 53, it calls the function $clone_i(local_i)$, which creates a separate copy of $local_i$ that is then used in every of the $N-1$ calls of $encapsulate_i(local_i)$.

Thus, during a step a processor can call functions from the set $\mathcal{F}_1 := \{increment(), revive(), labelBookkeeping(), restartLocal(), clone(), encapsulate()\}$ in case (1), $\mathcal{F}_2 := \{encapsulate()\}$ in case (2), and $\mathcal{F}_3 := \{labelBookkeeping(), merge(), revive(), restartLocal()\}$ in case (3). We define $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3$ to be the set of functions that a processor can call during a step.

Transitions We define the notion of transitions to denote the application of a single function on a pair during an execution. We say that (Z, f, Z') is a *transition* in R, if there exists a step $a_i \in R$ of a processor $p_i \in P$ and a function $f \in \mathcal{F}$, such that p_i calls $f(Z, \bullet)$ in step a_i with output Z'. In this paragraph, we list all possible transitions for every function $f \in \mathcal{F}$. In the following paragraphs, we define the pair evolution graph of an execution, based on the set of all transitions that occurred during that execution.

Transitions of pairs that stay intact between consecutive steps. We define the transition (Z, λ, Z), which denotes that the pair Z (either in a communication channel or in $local_j$ of a processor $p_j \in P$) remained intact between a step a_x and the beginning of consecutive step, a_{x+1}.

Transitions due to a call to labelBookkeeping(). We define the transition $(Z, labelBookkeeping(), Z')$ to denote a call to $labelBookkeeping_i()$ that does not change the state of the labeling algorithm, and distinguish the following cases. When p_i calls $labelBookkeeping_i()$ in the do-forever loop (line 45), we consider the transition $(local_i, labelBookkeeping(), local_i)$, since $local_i$ stays intact after the call to $labelBookkeeping_i()$ ends. Moreover, when p_i calls $labelBookkeeping_i()$ in the message arrival procedure for a message $m = \langle \bullet, arriving_j \rangle$ (line 58), we consider the transition $(arriving_j, labelBookkeeping(), local_i)$, since information from m is incorporated to the local label storage.

We consider the cases in which p_i possibly changes the state of the labeling algorithm during a call to $labelBookeeping_i()$ by either

(i) canceling a label and creating or recycling another label during a call to $revive_i()$ (in fact $revive()$ calls *cancelPairLabels*() in line 17, which includes a call to $labelBookkeeping_i()$ in line 13), or
(ii) discovering stale information in the label storage in line 45, or
(iii) receiving a new label during the message arrival procedure in line 58.

We denote any of the changes in the state of the labeling algorithm that are stated above with the (abstract) function $newLabel()$ and remark that whenever a processor calls $newLabel()$, the labeling algorithm deviates from its abstract task.

We define transitions for cases (i–iii) as follows:

(i) $(local_i, labelBookkeeping() \circ newLabel_i(), local_i)$ refers to the case where p_i calls $labelBookkeeping_i()$ during a call to $revive_i()$ (lines 16–18)), which includes a call to $newLabel_i()$,
(ii) $(local_i, labelBookkeeping() \circ newLabel(), local_i)$ refers to the case where p_i calls $labelBookkeeping_i()$ in line 17, which includes a call to $newLabel_i()$, and
(iii) $(arriving_j, labelBookkeeping() \circ newLabel(), local_i)$ refers to the case where p_i calls $labelBookkeeping_i()$ in line 58 due to an arriving message $m = \langle \bullet, arriving_j \rangle$, which includes a call to $newLabel_i()$.

Moreover, a call to $revive()$ on $local_i$ is illustrated by the transition $(local_i, labelBookkeeping() \circ newLabel(), local_i)$, denoting the call to the function $cancelPairLabels_i()$, followed by transition $(local_i, revive(), revive_i(local_i))$, to denote the creation of $revive_i()$'s output pair.

Transitions due to a call to increment() or revive(). We illustrate a call to the $increment_i()$ function by p_i through a number of transitions, depending on the value of $exhausted_i(local_i)$ in line 23. In case $exhausted_i(local_i)$ is false, then $increment_i()$ is only changing $local_i$ to a new value in line 21, say $local_i'$. In this case, the transition $(local_i, increment(), local_i')$, captures all the changes that occurred to $local_i$ during the call to $increment_i()$. Otherwise, if

$exhausted_i(local_i)$ is true, then a call to $revive_i()$ (line 23) follows the update from $local_i$ to $local'_i$. In this case, we illustrate the call to $increment_i()$ with the three following transitions. The first is the transition $(local_i, increment(), local'_i)$, which indicates the change that occurs to $local_i$ in line 21. The two following transitions are $(local'_i, labelBookkeeping() \circ newLabel(), local'_i)$ and $(local'_i, revive(), revive_i(local'_i))$, to denote the call to $revive_i()$ in line 23.

All possible transitions. We define the set of all transitions that are possible in a step $a_i \in R$ to be the set $\mathcal{T}_i := \cup_{f \in \mathcal{F}}\{(Z, f, f(Z, \bullet)\} \cup \{(Z, \lambda, Z), (Z, labelBookkeeping_i() \circ newLabel_i(), Z'\}$, where $f(Z, \bullet)$ denotes the output pair of f when Z is part of its input. Moreover, we define the set of all transitions that occur during a step to be the set $E_i(R) \subset \mathcal{T}_i$. Given a transition $e = (Z, f, Z')$, we refer to f as the *tag* of e, and the function $T_R : E(R) \to \mathcal{F}$ returns the tag f of e, i.e., $T_R(e) = f$.

All pair values during an execution Our definitions consider all the pair values that appear in the system during R. These values can appear in the data field of a message that resides in the communication channel, or in the state of a processor. Additionally, we consider values of pairs that appear temporarily during a step, because we are interested in the exact values that the algorithm functions compute, and thus we unseal the encapsulation of the step atomicity (Sect. 3). For the sake of providing intuition for the definition of pair evolution graphs, we define the collection $V_i(R)$. Specifically, for a step $a_i \in R$ that follows the state c_i, we define $V_i(R)$ to be the collection (with duplicates) of pairs that includes: (i) all pairs that appear in the state of every processor in c_i, and (ii) all the pairs that are outputs of functions that are called during the step a_i.

Consider two pairs that are identical but appear either (a) in different states or (b) in the same state but one appears in the communication channel, and the other one appears either in a different message or in the processor's local pair. Then these two pairs appear as different elements in $V_i(R)$. Moreover, since the output of $clone()$, $encapsulate()$, and $labelBookkeeping()$ equals the input these pairs appear twice in $V_i(R)$. At any time, the size of $V_i(R)$ is bounded by $N + M$ plus the number of pairs that are outputs of the functions that a processor calls during step a_i. This bound holds due to the fact that we have N processors and $M = \mathcal{C}N(N-1)$ is the maximum capacity of pairs in the communication channels.

Pair evolution graph We define a graph that illustrates the evolution of all the pair values that appear in the system during R, according to the interleaving model. In this graph, the vertices are all the pair values that appear in the data field of every message and the states of every processor (including the intermediate stages that steps use for their computation) of an execution R. The graph's edges are all the transitions between couples of pairs that occur during R.

We define the *pair evolution graph* of an execution R to be the directed and layered graph with tagged edges $\mathcal{G}(R) = (V(R), E(R))$, where $V(R) = \cup_{a_i \in R} V_i(R)$, $E(R) = \cup_{a_i \in R} E_i(R)$, and an edge $(Z, f, Z') \in E(R)$ is a directed graph's edge (Z, Z') tagged with f (and hence denoted with a triple). We say that $V_i(R) \subseteq V(R)$ is a *layer* of $\mathcal{G}(R)$, for every step a_i of R. We illustrate an example of a pair evolution graph (and hence the transitions) in Fig. 4, and give some insights below.

We do some observations for pair evolution graphs. Let $=_{V(R)}$ be the relation that denotes the fact that two pairs are the same node in $\mathcal{G}(R)$. For example, it might be the case that $Z_1 = Z_2$ but $Z_1 \neq_{V(R)} Z_2$, due to the multiple copies of a single pair that are created in a send operation. By the definition of $\mathcal{G}(R)$, all edges that are tagged with λ, connect only pairs of consecutive layers, i.e., for every $e = (Z, \lambda, Z') \in E_i(R)$, $Z' \in V_{i+1}(R) \wedge$

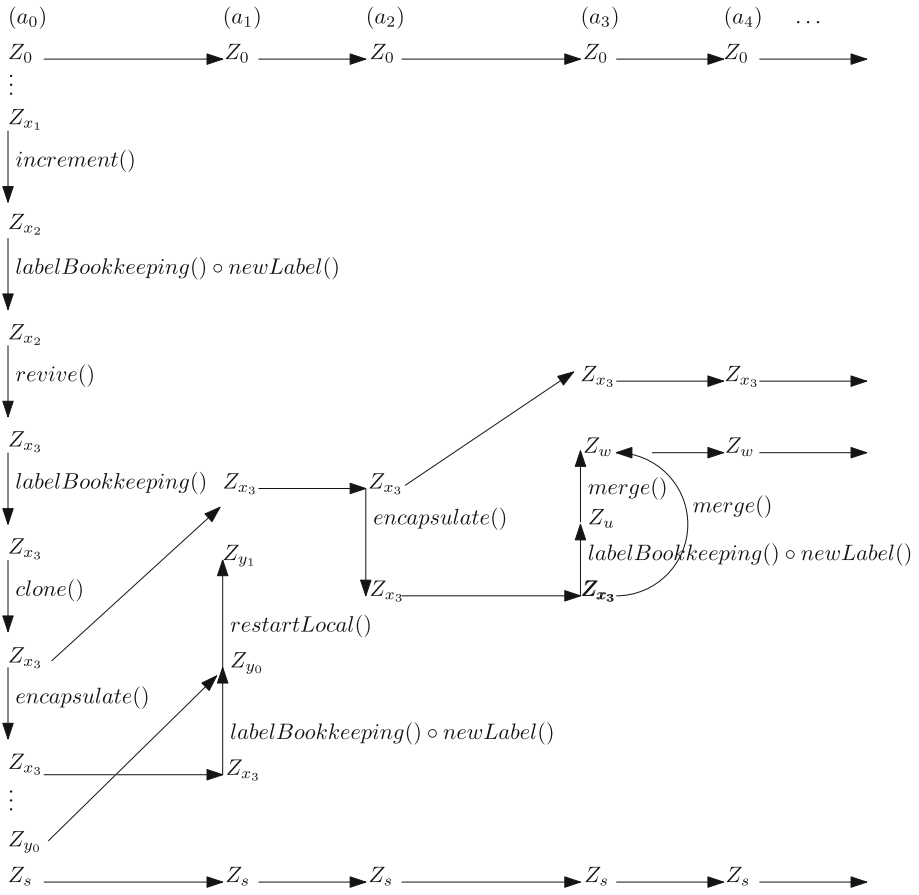


Fig. 4 An example of the pair evolution graph $\mathcal{G}(R)$ of an arbitrary execution R. For simplicity, we illustrate any edge $(Z, \lambda, Z') \in E(R)$ without its tag λ. In step a_0, processor p_i calls $increment$ on $Z_{x_1} = local_i$, which corresponds to the following three transitions: $(Z_{x_1}, increment(), Z_{x_2})$ refers to line 21 and $(Z_{x_2}, labelBookkeeping() \circ newLabel(), Z_{x_2})$ together with $(Z_{x_2}, revive(), Z_{x_3})$ refer to the call to $revive()$ in line 23. Then, p_i does an iteration of its do-forever loop (lines 43–53), which ends with one send operation, say to processor p_j. The latter corresponds to the transitions $(Z_{x_3}, labelBookkeeping(), Z_{x_3})$, $(Z_{x_3}, clone(), Z_{x_3})$, and $(Z_{x_3}, encapsulate(), Z_{x_3})$. In step a_1, p_j receives p_i's message (lines 57–66), which cannot be merged with $Z_{y_0} = local_j$ due to label incomparability, and hence p_j calls $restartLocal_j()$ (line 62). This step corresponds to the transitions $(Z_{x_3}, labelBookkeeping() \circ newLabel(), Z_{y_0})$ and $(Z_{y_0}, restartLocal(), Z_{y_1})$. In step a_2, p_i does one more send operation (line 53), say to processor p_k, which corresponds to the transition $(Z_{x_3}, encapsulate(), Z_{x_3})$. Then, in step a_3, p_k receives p_i's message and merges Z_{x_3} to $Z_u = local_k$ (lines 57–66). This step corresponds to the transitions $(Z_{x_3}, labelBookkeeping() \circ newLabel()(), Z_u)$, $(Z_u, merge(), Z_w)$, and $(Z_{x_3}, merge(), Z_w)$

$Z = Z' \wedge Z \neq_{V(R)} Z'$ holds and also Z is not further processed in step a_i. Also, all edges that are not tagged with λ, include pairs from the same layer, i.e., $\forall e = (Z, t, Z') \in E(R)$ such that $t \neq \lambda$, there exists a step $a_i \in R$, such that $Z \in V_i(R) \wedge Z' \in V_i(R)$ holds. Moreover, there is no edge in $\mathcal{G}(R)$, that includes pairs from non-consecutive layers, i.e., $\forall e = (Z, t, Z') \in E_i(R),\ Z' \in V_i(R) \cup V_{i+1}(R)$ holds. We note that given a subexecution $R' \sqsubseteq R$, the pair evolution graph of R' is the subgraph of $\mathcal{G}(R)$ that includes the layers corresponding to the steps of R', i.e., $\mathcal{G}(R')$.

Function causality for $restartLocal()$ In Sect. 8.4 we showed that $restartLocal()$ and $revive()$ are the only two functions of Algorithm 2 that can possibly violate the conditions for an execution to be in $\mathsf{E_{AT}}$. In this paragraph, we define a notion of function causality with respect to $restartLocal()$, and note that we deal with the case of $revive()$ in the following paragraph. Our definition determines when a call to a function $f \in \mathcal{F}$ in a step of an execution R *causes* a function call to $restartLocal_i()$ in a subsequent step.

We focus on a subset of $\mathcal{F}$, $\mathcal{F}_{focused} := \{restartLocal(), revive()\}$, since as we will show in Sect. 8.6, the number of calls to functions in $\mathcal{F}_{focused}$ has an effect on the number of subsequent calls to $restartLocal()$. In order to define function causality, we first define when two functions are adjacent or connected (i.e., there is a path that connects them in $\mathcal{G}(R)$) in Definition 8.13.

Definition 8.13 *(Adjacent and connected functions)* For two functions $f, g \in \mathcal{F}$, we say that f *is adjacent to* g in R, if and only if, $\exists i \in \mathbb{N}, e_1 = (Z_1, f, Z_1') \in E(R), e_2 = (Z_2, g, Z_2') \in E(R) : T_R(e_1) = f \wedge T_R(e_2) = g \wedge Z_1' =_{V(R)} Z_2$ holds, i.e., (e_1, e_2) is a path in $\mathcal{G}(R)$. For $f, g \in \mathcal{F}_{focused}$, we say that f *is connected to* g in R, if and only if, there exist $e_1, e_2, \ldots, e_x \in E(R)$, such that $T_R(e_1) = f \wedge T_R(e_x) = g \wedge (\wedge_{i=1,\ldots,x-1}(e_i$ is adjacent to $e_{i+1}))$.

Recall that we refer to all the variables of Z except for $Z.curr.m$ as the static part of Z, since increments to Z affect only $Z.curr.m$. We say that a function f leaves the static part of a pair Z intact, if the static part of $f(Z, \bullet)$ equals the static part of Z. Lemma 8.14 shows which functions leave the static part of their input pairs intact and which don't.

Lemma 8.14 *The functions in* $\mathcal{F} \setminus \mathcal{F}_{focused}$ *leave the static part of at least one of their input pairs intact. For each function in* $\mathcal{F}_{focused}$*, the input and output pairs may differ in their static parts.*

Proof The lemma statement holds for all functions in $\mathcal{F} \setminus (\mathcal{F}_{focused} \cup \{merge(), increment()\}) = \{clone(), encapsulate(), labelBookkeeping()\}$, since these functions leave their input intact (recall that $labelBookkeeping()$ does not operate on a pair). Since the output of $merge()$ equals, in its static part, to one of the two input pairs, the claim holds also for $merge()$ (cf. Section 6). Note that a call to $increment()$ can include a call to $revive()$ (we study the case of $revive()$ below), however line 21 does not change the static part of the input pair.

The functions in $\mathcal{F}_{focused} = \{revive(), restartLocal()\}$, by their definitions, can possibly output pairs that have different static part from their input (cf. lines 9 and 16–18). For the case of $revive()$, let p_i be a processor, $Z_1 = local_i$, and $Z_2 = revive_i(Z_1)$. Since p_i cancels the labels of Z_1 and uses the new maximal label that the labeling algorithm returns as the label in $Z_2.curr.\ell$ (lines 16–18), the static parts of Z_1 and Z_2 are different.

We now study the case of $restartLocal()$. Recall from the definition of $restartLocal()$ (line 9) that immediately after a processor p_i calls the function $restartLocal_i()$, $local_i$ has the form $\langle y, y\rangle$, where $y = \langle getLabel_i(), zrs, zrs\rangle$ and zrs is the N-size zero vector. Thus, the only case where $restartLocal_i()$ leaves the static part of $local_i$ intact, is when $local_i = \langle\langle \ell_x, \bullet, zrs\rangle, \langle \ell_x, zrs, zrs\rangle\rangle$ holds immediately before p_i calls $restartLocal_i()$, where ℓ_x is p_i's local maximal label. The latter holds because $local_i$ equals $\langle\langle \ell_x, zrs, zrs\rangle, \langle \ell_x, zrs, zrs\rangle\rangle$ after p_i calls $restartLocal_i()$. This is the case when p_i receives a pair $arriving_j$ from a processor p_j, such that $existsPivot_i(local_i, arriving_j)$ does not hold and $local_i.curr.\ell$ remains p_i's maximal label even after p_i processes the labels in $arriving_j$.

For any other case except for the one described above one of the following is true for $local_i$ immediately before p_i calls $restartLocal_i()$: $local_i.curr.\ell \neq local_i.prev.\ell$ or $s \neq zrs$,

for at least one vector s in $\{local_i.curr.m, local_i.prev.m, local_i.prev.o\}$. For any of these cases and by the definition of $restartLocal()$, immediately after p_i calls $restartLocal_i()$, $local_i$ has a different static part than immediately before p_i called $restartLocal_i()$. □

In Definition 8.15 we define function causality between a call to a function in $\{revive(), restartLocal()\}$ and a subsequent call to $restartLocal()$. Let $p_{restartLocal}(i,k) := \neg(mirroredLocalLabels() \wedge labelsRelation(local_i))$ (line 47) and $q_{restartLocal}(i,j,k) := \neg legitPairs(local_i, arriving_j)$ (line 62) be the predicates such that whenever either of them is true p_i calls $restartLocal_i()$ in a step $a_k \in R'$, where $arriving_j$ is the pair received by p_i in a message from p_j in step a_k.

Definition 8.15 *(f causes $restartLocal()$, $f \in \{revive(), restartLocal()\}$)* Let $|R'| \leq MAXINT$ be an execution, $p_i, p_j \in P$, $a_k \in R'$, and $\mathcal{P}(i,j,k) := p_{restartLocal}(i,k) \vee q_{restartLocal}(i,j,k)$. Also, let e, e' be two edges in $\mathcal{G}(R')$, such that $T_{R'}(e) \in \{revive(), restartLocal()\}$, $T_{R'}(e') = restartLocal()$, $e \in E_r(R')$, $e' \in E_k(R')$, $r \leq k$, and $\mathcal{P}(i,j,k)$ is true (in step a_k). We say that $f := T_{R'}(e)$ *causes* $restartLocal()$ in R', if and only if, the following hold:

(a) f is connected to $restartLocal()$ in $\mathcal{G}(R')$ through a path $P = (e_1, \ldots, e_x)$, such that $e_1 = e$ and $e_x = e'$,
(b) the value of a predicate π in $\mathcal{P}(i,j,k)$ depends on a vector clock item I_f in f's output, and
(c) for every edge $e_t \in P$, such that $e_t \notin \{e, e'\}$, it holds that the function $T_{R'}(e_t)$ does not change the label and the offset of I_f.

For example, in Fig. 4 $revive()$ in step a_0 causes $restartLocal()$ in step a_1, since the pair (and hence the vector clock items) that p_i sent to p_j, was incomparable (no pivot existed) with p_j's local pair, was created when p_i called $revive()$ in a_0.

Function causality for $revive()$ In the following lemma, we show which functions in $\mathcal{F}$ can change the value of the predicate $exhausted(local)$ (Eq. 1 and lines 23, 50, and 66), and thus cause a call to $revive()$ (Lemma 8.16). We note that the analysis for $revive()$ (Lemma 8.16) is much simpler than the one for $restartLocal()$, because the condition for calling $revive()$, i.e., $exhausted(local)$, depends only on vector clock increments. On the contrary, the conditions for calling $restartLocal()$ (lines 47 and 62) depend on every field of $local$, as well as on whether an arriving pair can be merged with the local one.

Lemma 8.16 *Let $p_i \in P$. (i) The value of the predicate $exhausted_i(local_i)$ (cf. Section 6) can change to true only due to a call to $increment_i()$ or $merge_i()$, or it can be true in the starting system state due to stale information in p_i's state. (ii) The value of $exhausted_i(local_i)$ does not change after p_i calls a function in $\{labelBookkeeping_i(), clone_i(), encapsulate_i()\}$. (iii) The value of $exhausted_i(local_i)$ is false after p_i calls a function in $\mathcal{F}_{focused} = \{restartLocal(), revive()\}$.*

Proof Recall that $exhausted(Z) \Leftrightarrow \Sigma_{k=1}^{N}(Z.curr.m[k] - Z.curr.o[k]) \geq MAXINT - 1$ (Eq. 1). For part (i) of the claim, first note that the starting system state, c_0, of any execution, R, is arbitrary. Hence, it can be the case that $exhausted_i(local_i)$ is true for $local_i$ in c_0. The lemma statement holds for $increment()$, since by its definition (lines 20–23) it increases $local_i.curr.m$. Similarly, $merge_i(local_i, arriving_j)$ outputs a pair that possibly includes more events than $local_i$ and $arriving_j$ (cf. lines 26–41 and Sect. 6). Hence, it might be the

case that $exhausted_i(local_i)$ is false before a call to $merge_i(local_i, arriving_j)$, but true for the new value of $local_i.curr.m$, when p_i stores in $local_i$ the output of $merge_i()$.

For part (ii) of the claim, note that $labelBookkeeping_i()$, $clone_i()$, and $encapsulate_i()$ do not change $local_i.curr.m$. Finally, part (iii) of the claim is true since by the definitions of $restartLocal()$ (line 9) and $revive()$ (lines 16–18), $local_i.curr.m = local_i.curr.o$ holds for their outputs, hence the predicate $exhausted_i(local_i)$ is false. □

8.6 Bounding the number of deviations from the abstract task in a *MAXINT*-sized execution

In Lemma 8.17, we show that the number of steps in which a processor calls $revive()$ or $restartLocal_i()$ during an execution R', such that $|R'| \leq MAXINT$, is in $poly(N)$. We focus on these two functions, because due to Sect. 8.5, only these two functions can cause a call to $restartLocal()$ (cf. Lemma 8.14 and Definition 8.15). Then, in Corollary 8.24 we show that Algorithm 2 is practically-self-stabilizing (i.e., Theorem 8.1 holds).

Lemma 8.17 *Let R be an execution of Algorithm 2 and R' be a subexecution of R, such that $|R'| \leq MAXINT$. Then, the number of steps in which a processor calls either $revive()$ or $restartLocal()$ in R' is in $poly(N)$.*

Proof The proof focuses on giving a bound on the number of steps in which a processor calls $restartLocal()$ in R' and showing that the bound is significantly less than $MAXINT$. As a by-product of this goal, Claim 8.18 shows that the number of steps in R' in which a processor calls $revive()$ is in $poly(N)$.

A processor can call $restartLocal()$ either in line 47 or in line 62. The proof considers both cases. We first show that there can be at most one call per processor to $restartLocal()$ during any execution due to line 47. To prove this statement, first observe that the condition in line 47 can be false due to stale information that resided in the processor's state in the starting state. However, by Corollary 8.7, for any function that changes $local$, it holds that the condition in line 47 is false for the updated value of $local$.

In the remainder of this proof, we show that the number of steps in R' that include a call to $restartLocal()$ due to line 62 is in $poly(N)$. We first bound the maximum number of steps that include a call to $revive()$ (Claim 8.18), as well as the maximum number of labels that can exist during R' (Claim 8.19). In claims 8.20 and 8.21, we bound the number of steps that include a call to $restartLocal()$ in line 62 due to stale information in the communication channels, and respectively, the token-passing mechanism (Sect. 7). Moreover, in Claim 8.22 we bound the number of calls to $restartLocal()$ in line 62 that occur due to a single pair static part that appears in R'. Finally, in Claim 8.23 we show that these bounds imply that the number of steps that include a call to $restartLocal()$ in line 62 during R' is in $poly(N)$, by showing that the number of pair static parts that appear in R' is in poly(N) and combining Claims 8.18–8.22.

Claim 8.18 The number of steps during R' that include a call to $revive()$ is at most $2N^2 + N^3 \cdot \mathcal{C}$.

Proof of Claim 8.18 The proof considers the three causes for pair exhaustion during R (cf. Lemma 8.16). That is, due to calls to $increment()$ and $merge()$, as well as due to stale information that appeared in the starting system state.

Since $|R'| \leq MAXINT$, the maximum number of increments that can occur in R is less than $MAXINT$. Note that for a single vector clock pair exhaustion, at most all processors

can wrap around concurrently. This can occur when processor p_j holds a pair value Z_j in $local_j$ that is close to be exhausted, say, just one increment away (lines 20–23). Then, p_j sends $local_j$'s value Z_j to all other processors $p_k \in P$ in the system. Every processor p_k that receives Z_j, merges it with $local_k$, and in the following step calls $increment_k()$, which leads to exhausting $local_k$. Hence, there can be at most N^2 steps in R' that processors p_k take, that include a call to $revive_k()$ due to the exhaustion of a pair that was merged with Z_j (at most N choices for p_j and at most N choices for p_k). Indeed, p_k can exhaust the output of $merge_k(local_k, Z_j)$ at most once, because the call to $revive_k()$ produces a pair with a static part (and hence the labels $curr.\ell$ and $prev.\ell$) that is different than the one of Z_j and $local_k$.

The remaining pair exhaustions can be only due to arbitrary values that resided in the starting system state (cf. Lemma 8.16). At most N such vector clocks have resided in the states of the processors, and at most $M \leq N^2 \cdot \mathcal{C}$ resided in the communication channels. Since for each of these (at most) $N + N^2 \cdot \mathcal{C}$ pair values can lead to at most N concurrent exhaustions, there can be $N^2 + N^3 \cdot \mathcal{C}$ exhaustions due to pairs that come from the arbitrary starting state.

Note that we have counted the number of steps that include a call to $revive()$ in two ways; (i) calls to $increment()$ or $merge()$, and (ii) stale information that appeared in the starting system state. Of course, a pair can become exhausted due to a combination of these two causes. The arguments above hold for such combinations and the counting is correct because each pair exhausted is counted at least once. Therefore, in total, there can be at most $2N^2 + N^3 \cdot \mathcal{C}$ pair exhaustions that can occur during R'. Hence, at most that many calls to $revive()$ in R'. □

Claim 8.19 The maximum number of labels that can exist in R' (and hence the number of steps that include a call to the $newLabel()$ function) is in $\mathcal{O}(\mathcal{C}N^3)$.

Proof of Claim 8.19 Recall that from the proofs of corollaries 8.1, 8.2 and 8.3, proving that the labeling algorithm of Dolev et al. [14] is practically-self-stabilizing depends on the existence of a bound on the maximum number of labels, rather than the actual value of the bound. We give a (polynomial) bound on the number of labels that exist during R', which implies that the number of steps that include a call to $newLabel()$ (cf. Section 8.5) has the same bound.

By corollaries 8.1 and 8.2 there can be at most $x = 4N^2 + 4NM - 4N - 2M$ labels in a given system state, where $M = \mathcal{C}N(N-1)$ is the maximum capacity of pairs in the communication channels. Note that, during R, there can be at most $y = 2N^2 + N^3 \cdot \mathcal{C}$ additional labels creations, due to calls to the $revive()$ function (cf. Claim 8.18). Thus, there can be at most $L = x + y = (2N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ labels in any system state that appear in R', and hence at most that many calls to $newLabel()$. Also see the two comments after corollary 8.2. □

In the labeling algorithm of Dolev et al. [14], each processor p_i uses an N-size array of bounded FIFO queues, $storedLabels_i[]$, for keeping a label history. The queue $storedLabels_i[j]$ stores the labels that p_i has received that show p_j as their creator, i.e., $\ell.creator = j$ holds for every $\ell \in storedLabels_i[j]$. Recall that in Sect. 8.3 we extended the queue lengths for an execution of Algorithm 2 in which no processor calls $revive()$, to accommodate for the two labels that each pair includes. By Claim 8.19, we are able to extend the size of the label storage of the labeling algorithm, in order to accommodate for the extra label creations due to calls to the function $revive()$ in Algorithm 2. Thus, by Sect. 8.3, we know that it is sufficient to let $storedLabels_i[j]$ hold a queue of length of at most L, for every $p_i, p_j \in P$, where $L \in \mathcal{O}(\mathcal{C}N^3)$ is calculated in the proof of Claim 8.19.

Claim 8.20 There can be at most $\mathcal{C} \cdot N^2$ steps with a call to $restartLocal()$ in line 62 due to stale information at the communication channels.

Proof of Claim 8.20 Recall that there are $N(N-1)/2$ links in the system, where each of them is a bidirectional communication channel and assumed to hold at most $\mathcal{C}$ messages per direction. Therefore, there can be at most $2\mathcal{C} \cdot N(N-1)/2 \leq \mathcal{C} \cdot N^2$ steps that include a call to $restartLocal()$ due to stale information that, at the starting system state, resides in the channels. □

Claim 8.21 Let $m_{j,i} = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ be a message that p_i receives from p_j, via their communication channel, $channel_{j,i}$. There can be at most $\mathcal{C}$ steps that include a call to $restartLocal_i()$ in line 60, due to stale information that appears in the field $rcvdLocal_j$ of $m_{j,i}$, where $m_{j,i}$ appears in $channel_{j,i}$ in the starting system state and $rcvdLocal_j$ sets $equalStatic_i(local_i, rcvdLocal_j)$ to true. Hence, there can be at most $M \leq \mathcal{C}N^2$ such calls to $restartLocal()$ in any execution.

Proof of Claim 8.21 Notice that there can be at most $\mathcal{C}$ messages in the communication channel from p_j to p_i, $channel_{j,i}$, at any time, and specifically in the starting system state. Each of these $\mathcal{C}$ messages in transit from p_j to p_i can possibly store a value of $rcvdLocal_j$, such that $equalStatic_i(local_i, rcvdLocal_j)$ is true in line 60, which leads to a call to $restartLocal_i()$ in line 62.

We provide details about how this can occur. Consider a step $a_x \in R$ of p_i that includes a call to $restartLocal_i()$ due to a message arrival (line 62). Hence, $equalStatic_i(local_i, rcvdLocal_j)$ was true during a_x. The fact that a_x includes a call to $restartLocal_i()$ does not cause the other $\mathcal{C}-1$ stale messages in the channel from p_j to p_i to be omitted. Therefore, there could be a subsequent step during which $equalStatic_i(local_i, rcvdLocal_j)$ is true due to the other $\mathcal{C}-1$ messages in $channel_{j,i}$ that have stale information, since those messages appeared in the starting system state.

Such steps can be repeated at most $\mathcal{C}$ times for $channel_{j,i}$ and at most M in total during R', where M is the number of messages in transit at any given time and hence in starting system state, c_0. □

As mentioned, we refer to all variables of the pair Z except for $Z.curr.m$ as the static part of Z, see the text before Lemma 8.14 for details.

Claim 8.22 Let $L = (2N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ be the maximum number of labels that can appear in the system in R' (Claim 8.19). During R', there can be at most $2N \cdot L$ calls to $restartLocal_i()$ in line 60 for every pair static part that appears in $local_i$ of a processor p_i. Hence, for each pair static part that appears in the state ($local$) of a processor in R', there can be at most $2N^2L$ calls to $restartLocal_i()$ in line 60.

Proof of Claim 8.22 Let $p_i, p_j \in P$ be two processors and $m_{j,i} = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ be a message that p_j sends to p_i by adding it to $channel_{j,i}$. Consider the case where the pair static part of $arriving_j$ sent by p_j to p_i causes p_i in step a_x to call $restartLocal_i()$ and obtain $local_i = Z$. Note that when referring to a value Z or Z_x that a variable takes, e.g., $local_i$, we treat Z and Z_x as (immutable) literals, i.e., pair values that do not change. In Part I of the proof, we show that there can be at most one more call to $restartLocal_i()$ (i.e., a total of at most two) due to receiving the same pair static part from p_j, before p_j stores a pair with a different static part in $local_j$. The proof relies on the token-passing mechanism (lines 53, 59, and 60). In the proof of this claim, we assume that

the token passing mechanism has stabilized (since Claim 8.21 has already showed that the token passing mechanism of lines 53, 59, and 60 can cause an additive (bounded) number of calls to $restartLocal()$). Then, in Part II of the proof, we show that each pair static part can be created by a processor at most L times in R'. We combine Part I and II to obtain the claim's bound. Throughout the claim's proof, we denote with $\mathcal{S}(Z) = \langle\langle \ell_1, \perp, o_1\rangle, \langle \ell_2, m_2, o_2\rangle\rangle$ the static part of a pair $Z = \langle\langle \ell_1, m_1, o_1\rangle, \langle \ell_2, m_2, o_2\rangle\rangle$.

Part I Recall from line 60 that for any message $m_{j,i} = \langle \bullet, \langle arriving_j, rcvdLocal_j\rangle\rangle$ that p_j sends to p_i, $equalStatic_i(local_i, rcvdLocal_j)$ has to be true for p_i to process $arriving_j$. Let $local_i = Z_{i_1}$ in state c_x and suppose in the step a_x that immediately follows c_x, processor p_i calls $restartLocal_i()$ in line 60 after receiving $m_{j,i} = \langle \bullet, \langle Z_{j_1}, rcvdLocal_j\rangle\rangle$, which produces $local_i = Z_{i_2}$. Due to the token passing mechanism (lines 53, 59, and 60), p_i can process a new message from p_j in a step that follows a_x, only after p_j receives Z_{i_2} or a subsequent pair that appeared in $local_i$ after a_x (possibly after receiving other pairs). Let $a_{x'}$ be the first step after a_x in which p_j stores in $local_j$ a pair with static part different than $\mathcal{S}(Z_{j_1})$. We show that there can be at most one step between a_x and $a_{x'}$ (different than a_x and $a_{x'}$), in which p_i calls $restartLocal_i()$ due to receiving a pair with static part equal to the one of Z_{j_1}. Hence, there can be at most two such calls to $restartLocal_i()$, until a state in which p_j stores a pair in $local_j$ with static part different than $\mathcal{S}(Z_{j_1})$.

Recall that $local_i = Z_{i_2}$ is the value of $local_i$ in the state that immediately follows step a_x, in which p_i calls $restartLocal_i()$, and let $\ell_{i_2} = Z_{i_2}.curr.\ell = Z_{i_2}.prev.\ell$ for brevity. Observe from the definition of $restartLocal()$ (line 9), that $getLabel_i()$ returns ℓ_{i_2} according to a possible update of the local maximal label in line 58. Then ℓ_{i_2} is equal to either $Z_{i_1}.curr.\ell$ (Case a), or $Z_{j_1}.curr.\ell$ (Case b), or ℓ_{i_3} is different than both $Z_{i_1}.curr.\ell$ and $Z_{j_1}.curr.\ell$ (Case c). The latter case refers to a situation in which $Z_{i_1}.curr.\ell$ and $Z_{j_1}.curr.\ell$ cancel each other and p_i produces a new label in line 58 (which is then used in line 62).

Case a In this case $Z_{j_1}.curr.\ell \prec_{lb} Z_{i_1}.curr.\ell$ holds (cf. [31, Section 3] regarding the $\prec_{lb}$ relation). That is, $Z_{i_1}.curr.\ell$ was the value of $local_i.curr.\ell$ in the system state before a_x, $Z_{i_1}.curr.\ell$ remains as the maximal label of p_i even after p_i receives Z_{j_1} in a_x, and hence p_i uses $Z_{i_1}.curr.\ell$ in the return pair of $restartLocal_i()$ in a_x, $Z_{i_2} = \langle\langle Z_{i_1}.curr.\ell, zrs, zrs\rangle, \langle Z_{i_1}.curr.\ell, zrs, zrs\rangle\rangle$. We show that in a system state that immediately follows a step a_{y_a} in which p_j receives Z_{i_2} from p_i (hence after a_x), $\mathcal{S}(Z_{j_1}) \neq \mathcal{S}(local_j)$ holds. This is true due to the fact that p_j receives the message $m_{i,j} = \langle \bullet, \langle Z_{i_2}, \bullet\rangle\rangle$ from p_i, such that $Z_{i_2}.curr.\ell = Z_{i_1}.curr.\ell$, or a message from p_i with a pair which has a label larger than $Z_{i_2}.curr.\ell$ (due to the token passing mechanism in lines 53, 59, and 60). Since $Z_{j_1}.curr.\ell \prec_{lb} Z_{i_1}.curr.\ell$ (this case's assumption), we have that $Z_{j_1}.curr.\ell$ cannot be the label that appears in $local_j.curr.\ell$ after a_{y_a}, because during a_{y_a} line 58 causes p_j to adopt the label $Z_{i_2}.curr.\ell = Z_{i_1}.curr.\ell$, since we have $Z_{j_1}.curr.\ell \prec_{lb} Z_{i_1}.curr.\ell$ (or a label larger than $Z_{i_1}.curr.\ell$).

Case b In this case $Z_{i_1}.curr.\ell \prec_{lb} Z_{j_1}.curr.\ell$ holds, due to the fact that p_i sets $local_i = Z_{i_2} = \langle\langle Z_{j_1}.curr.\ell, zrs, zrs\rangle, \langle Z_{j_1}.curr.\ell, zrs, zrs\rangle\rangle$ when calling $restartLocal_i()$ in step a_x. Let a_{y_b} be the step (that follows a_x) when p_j receives Z_{i_2} and c_{y_b} be the system state that immediately precedes a_{y_b}. Note that the next pair after a_x that p_j will receive from p_i can possibly have a larger label than $\ell_{i_2} = Z_{i_2}.curr.\ell = Z_{j_1}.curr.\ell$, but this event falls in Case c, which we study below. Thus, in case p_j indeed receives Z_{i_2} in a_{y_b}, either (b-i) $\mathcal{S}(local_j) = \mathcal{S}(Z_{i_2})$ or (b-ii) $\mathcal{S}(local_j) \neq \mathcal{S}(Z_{i_2})$ holds (in c_{y_b}).

In case (b-i) $\mathcal{S}(local_j) = \mathcal{S}(Z_{i_2})$, the two pairs can be merged to $local_j = Z_{j_2}$, such that $\mathcal{S}(Z_{j_2}) = \mathcal{S}(Z_{i_2}) = \langle\langle \ell_{i_2}, \perp, zrs\rangle, \langle \ell_{i_2}, zrs, zrs\rangle\rangle$ is the representation of Z_{j_2}'s static part.

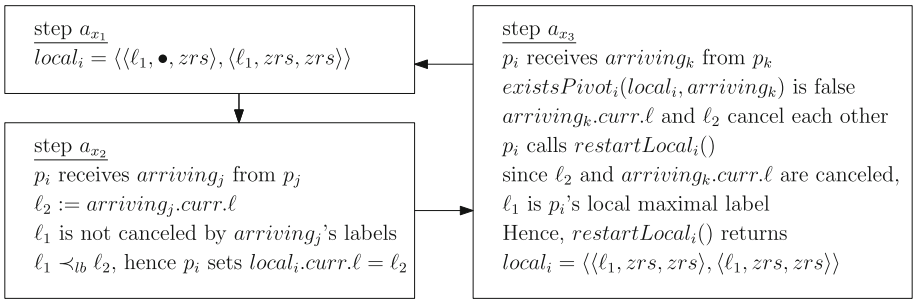


Fig. 5 Recycling of a pair static part by a processor p_i. In the end of step a_{x_1}, processor p_i stores $local_i = \langle\langle \ell_1, \bullet, zrs\rangle, \langle \ell_1, zrs, zrs\rangle\rangle$. In step a_{x_2}, p_i receives $arriving_j$ from p_j, such that $\ell_2 := arriving_j.curr.\ell$ becomes p_i's maximal label without canceling ℓ_1 (the labels were created by different processors). In step a_{x_3}, p_i receives $arriving_k$ from p_k, such that $arriving_k.curr.\ell$ and ℓ_2 cancel each other. Hence, ℓ_1, becomes again p_i's local maximal label and after p_i calls $restartLocal_i()$, $local_i = \langle\langle \ell_1, zrs, zrs\rangle, \langle \ell_1, zrs, zrs\rangle\rangle$ holds. That is, p_i *recycled* the same pair static part. Steps a_{x_1}, a_{x_2}, and a_{x_3} can be repeated (possibly with other steps in between) at most L times (Claim 8.22, Part II)

Thus, in a step that follows a_{y_b}, processor p_j sends Z_{j_2} to p_i and in step a_{z_b}, processor p_i receives Z_{j_2}. Consider the case where p_i merges Z_{j_2} with $local_i$ in a_{z_b} to $local_i = Z_{i_3}$.

- If $\mathcal{S}(Z_{i_3}) = \mathcal{S}(Z_{j_2})$ (due to merge or a $restartLocal_i()$ that used ℓ_{i_2} as p_i's maximal label), then we loop back to the beginning of Case b (without having an additional call to $restartLocal_i()$).
- Otherwise, if $\mathcal{S}(Z_{i_3}) \neq \mathcal{S}(Z_{j_2})$, then ℓ_{i_2} is not the maximal label in p_i (due to a call to $merge_i()$ (line 26) or a call to $restartLocal_i()$ in a_{z_b}). Hence, $Z_{i_3}.curr.\ell$ is either larger than ℓ_{i_2} or cancels ℓ_{i_2}. Subsequently, once p_j receives $Z_{i_3}.curr.\ell$ from p_i (due to the token passing mechanism in lines 53, 59, and 60), $local_j.curr.\ell$ will change to either $Z_{i_3}.curr.\ell$ or a larger label that resides in p_j.

Hence, in this subcase (b-i), a single pair static part that was stored in $local_j$, can cause p_i to call $restartLocal_i()$ at most twice due to the static part of Z_{j_1} before p_j changes the static part of $local_j$ to another one.

In case (b-ii), the fact that $\mathcal{S}(local_j) \neq \mathcal{S}(Z_{i_2})$ holds in the system state immediately before a_{y_b} implies that $Z_{i_2}.curr.\ell = Z_{j_1}.curr.\ell$ is not the maximal label in p_j immediately before a_{y_b}. The latter holds, because $local_j$ used to hold the value Z_{j_1} before a_{y_b} and p_j can only substitute the value of $local_j.curr.\ell$ for a label with a larger label than $Z_{i_2}.curr.\ell = Z_{j_1}.curr.\ell$. Hence, for the value of $local_j$ in the system state that immediately follows a_{y_b}, it holds that $\mathcal{S}(local_j) \neq \mathcal{S}(Z_{j_1})$, since $local_j.curr.\ell$ cannot be equal to $Z_{j_1}.curr.\ell$.

Case c In this case both $Z_{i_1}.curr.\ell$ and $Z_{j_1}.curr.\ell$ are canceled in a_x and p_i creates a larger label ℓ_{i_3} to use in Z_{i_2}. Thus, in the system state that immediately follows step a_{y_c}, in which p_j receives Z_{i_2} (or a pair with a $curr.\ell$ that is larger than $Z_{i_2}.curr.\ell$), it holds that $\mathcal{S}(local_j) \neq \mathcal{S}(Z_{j_1})$, since $Z_{j_1}.curr.\ell$ will not be the largest label in p_j's state that immediately precedes a_{y_c}.

By the case analysis above, we conclude that a single pair static part in p_j can cause p_i to call $restartLocal_i()$ either once (cases a, b-ii, and c) or twice (case b-i), before p_j changes the static part of $local_j$ to another one (different from the static part of Z_{j_1}).

Part II We show that there is a bound on the number of times a processor can create the same pair static part. Let us consider a scenario in which processor p_j stores the pair Z_1 in $local_j$ at some system state c, and then stores the pair Z_2, such that $\mathcal{S}(Z_1) \neq \mathcal{S}(Z_2)$, at a

system state c' that follows c, before creating (via $restartLocal_j()$) the pair Z_3, such that $\mathcal{S}(Z_1) = \mathcal{S}(Z_3)$, which results in a subsequent system state c'' that follows c'. We illustrate this scenario in Fig. 5.

We argue that $Z_3.curr.\ell = Z_3.prev.\ell$ holds in c''. Assume, towards a contradiction, that $Z_3.curr.\ell \neq Z_3.prev.\ell$ holds in c''. Since $\mathcal{S}(Z_1) = \mathcal{S}(Z_3)$, we have that $Z_1.curr.\ell \neq Z_1.prev.\ell$ holds in c. Moreover, since line 47 makes sure that either $Z_1.curr.\ell = Z_1.prev.\ell$ or $Z_1.prev.\ell$ is canceled, it holds that $Z_1.prev.\ell$ is canceled, and hence $Z_3.prev.\ell = Z_1.prev.\ell$ is canceled. Thus, p_j in a step that follows c used the canceled label $Z_1.prev.\ell$ to create a new pair and store it in $local_j$, which is a contradiction, because $getLabel_j()$ by its definition never returns a canceled label. Thus, it can only be the case that $Z_3.curr.\ell = Z_3.prev.\ell$ in c'', which means that p_j created Z_3 via a call to a $restartLocal_j()$. The latter implies that $\mathcal{S}(Z_1) = \mathcal{S}(Z_3) = \langle\langle \ell_x, \perp, zrs\rangle, \langle \ell_x, zrs, zrs\rangle\rangle$, where $\ell_x := Z_k.curr.\ell = Z_k.prev.\ell$, for $k \in \{1, 3\}$.

In fact, for this scenario to occur, ℓ_x should remain non-canceled in the label storage of p_j between c (where $local_j = Z_1$) and c'' (where $local_j = Z_3$), so that p_j can recycle ℓ_x via the labeling algorithm and have ℓ_x returned through $getLabel_j()$, as part of a $restartLocal_j()$. For that to happen, $Z_2.curr.\ell$ must be canceled by another label (so then p_j recycles ℓ_x). Such cancelation scenarios can occur at most L times in R', since there exist at most L labels in R'.

We remark that the number of pairs with static part different than $\mathcal{S}(Z_1)$ that p_j stores in $local_j$ between the states c and c'' does not change the fact that p_j can (create and thus) store Z_3 in c''. That is, the recycling scenario that we describe above with Z_1 and Z_3, is possible to occur even if p_j stores the pairs Z_k, for every k in a set of indices K, such that $\mathcal{S}(Z_k) \neq \mathcal{S}(Z_1), k \in K$. However, for the recycling scenario to occur we require that all the labels of the pairs $Z_k, k \in K$, cancel each other and make ℓ_x the maximal label in p_j's state in a system state between c and c'', which results to p_j using ℓ_x to create Z_3.

We now show that by combining Part I and II we obtain the claim's bounds. By Part I, a single pair static part of a pair Z that processor p_j stores can cause another processor p_i to call $restartLocal_i()$ at most twice before p_j stores a pair with a different static part than Z. By Part II, after p_j stores a pair in $local_j$ that has a static part different than Z, it can create again a pair with the same static part as Z at most L times in R'. Hence, a single pair static part can cause p_i to call $restartLocal_i()$ in at most $2L$ steps in R', hence $2(N-1) \cdot L \leq 2N \cdot L$ for all other processors (since there are $N-1$ choices for p_i). Since there are N choices for p_j there can be at most $2N^2L$ steps that include a call to $restartLocal()$ for each pair static part. □

In the proof of Claim 8.23, we use the claims of this lemma as well as Lemma 8.14 to show that the number of steps in R' that include a call to $restartLocal()$ due to line 62 is in $poly(N)$.

Claim 8.23 The number of steps that include a call to $restartLocal()$ due to line 62 in R' is in $\mathcal{O}(N^8\mathcal{C})$.

Proof of Claim 8.23 First, we show that the claim is true when there are no calls to $revive()$ in R'. Then, we extend our arguments to show that the claim holds even when there are calls to $revive()$ during R'. In the following, we denote with $V = 2N^2 + N^3\mathcal{C}$ the maximum number of steps that include a call to $revive()$ (Claim 8.18) and $L = (2N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ the maximum number of labels that can be created during R' (Claim 8.19).

First, suppose that there are no calls to $revive()$ during R'. Recall that there are at most $N+M$ distinct pairs in the starting system state of R', c_x. Since there are no calls to $revive()$ during R' and due to Lemma 8.14, any pair that appears in R' and differs to the ones that appear in c_x with respect to their pair static part (i.e., all pair variables except for $curr.m$), can only be created in a step of R' in which a processor called $restartLocal()$. A pair that is an output of $restartLocal()$ has the form $\langle\langle \ell, zrs, zrs\rangle, \langle \ell, zrs, zrs\rangle\rangle$, where $\ell = getLabel()$ is the local maximal label and zrs is an N-size vector of zeros. Thus, during R' there can be at most $N+M+L$ pairs with respect to their static part. The latter holds, since (i) each of the $N+M$ pairs from the starting state can be the input of a call to $restartLocal()$ (line 62), and (ii) any further call to $restartLocal()$ produces a pair of the form $\langle\langle \ell, zrs, zrs\rangle, \langle \ell, zrs, zrs\rangle\rangle$, and there can be at most L such pairs during R' due to Claim 8.19 (since their static part only differs on ℓ).

Hence, if there are no calls to $revive()$ during R', there can be at most $2\mathcal{C}N^2 + 2N^2L(N+M+L)$ calls to $restartLocal()$. This bound holds, since at most $2\mathcal{C}+1$ calls to $restartLocal()$ can be caused due to Claim 8.20, $\mathcal{C}N^2$ due to Claim 8.21, there can be at most $N+M+L$ pair static parts in R', and each of them can cause at most $2N^2L$ calls to $restartLocal()$ due to Claim 8.22 (including concurrent calls).

In case there exist steps in R' that include calls to $revive()$, then at most $2V$ more pair static parts are added in the system. The latter holds, because each of the V pair static parts are added in the system by the output of $revive()$, can be the input to $restartLocal()$, which in turn creates a new pair static part (hence at most V more pairs with different static parts). Thus, we update the calculation of the bound as follows: $2\mathcal{C}N^2+2N^2L(N+M+L+2V) \in \mathcal{O}(N^8\mathcal{C})$. □

We are now ready to combine the claims of this proof to prove the lemma statement. In the beginning of the proof we showed that each processor calls $restartLocal()$ in line 47 at most once for any execution and in Claim 8.23 we showed that during R' each processor calls $restartLocal()$ in line 62 in a number of steps that is in poly(N). Thus, the number of steps in which a processor calls $revive()$ (Claim 8.18) or $restartLocal()$ during R' is in $poly(N)$. □

By the definition of practically-self-stabilizing systems and by the fact that the number of deviations from the vector clock task (requirements 1 and 2) is in $poly(N)$, we get the following.

Corollary 8.24 *Algorithm 2 implements a practically-self-stabilizing solution for the vector clock task.*

9 Conclusion

Dijkstra's self-stabilization requires, within a bounded recovery period, the complete absence of stale information (that is due to transient faults). In this paper, we study stabilization criteria that are less restrictive than Dijkstra's self-stabilization. The design criteria that we consider allow recovery after the occurrence of transient faults (without considering fair execution) and still tolerate crash failures, which we do not model as transient faults. We show the composition of two practically-self-stabilizing systems (Sect. 5) and present an elegant technique for dealing with concurrent overflow events (Sect. 6). We believe that the

proposed algorithm (Sect. 7) and its proof techniques (*e.g.,* the counting arguments) can be the basis of other practically-self-stabilizing algorithms.

Author Contributions All authors contributed equally to all parts of the paper preparation and review.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

References

1. Almeida, J.B., Almeida, P.S., Baquero, C.: Bounded version vectors. In: Distributed Computing, 18th International Conference, pp 102–116 (2004)
2. Alon, N., Attiya, H., Dolev, S., et al.: Practically stabilizing SWMR atomic memory in message-passing systems. J. Comput. Syst. Sci. **81**(4), 692–701 (2015)
3. Altisen, K., Devismes, S., Dubois, S., et al.: Introduction to distributed self-stabilizing algorithms. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2019)
4. Arora, A., Gouda, M.G.: Closure and convergence: a foundation of fault-tolerant computing. IEEE Trans. Software Eng. **19**(11), 1015–1027 (1993)
5. Arora, A., Kulkarni, S.S., Demirbas, M.: Resettable vector clocks. J Parallel Distrib Comput **66**(2), 221–237 (2006)
6. Blanchard, P., Dolev, S., Beauquier, J.: Practically self-stabilizing paxos replicated state-machine. In: Networked Systems NETYS, pp. 99–121. (2014)
7. Bonomi, S., Dolev, S., Potop-Butucaru, M.: Stabilizing server-based storage in byzantine asynchronous message-passing systems: extended abstract. In: ACM Symposium on Principles of Distributed Computing, PODC, pp. 471–479. (2015)
8. Delaët, S., Devismes, S., Nesterenko, M., et al.: Snap-stabilization in message-passing systems. J Parallel Distrib Comput **70**(12), 1220–1230 (2010)
9. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
10. Dolev, S.: Self-stabilization. MIT Press (2000)
11. Dolev, S., Schiller, E.: Communication adaptive self-stabilizing group membership service. IEEE Trans. Parallel Distrib. Syst. **14**(7), 709–720 (2003)
12. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Comput **7**(1), 3–16 (1993)
13. Dolev, S., Kat, R.I., Schiller, E.M.: When consensus meets self-stabilization. J. Comput. Syst. Sci. **76**(8), 884–900 (2010)
14. Dolev, S., Georgiou, C., Marcoullis, I., et al.: Practically-self-stabilizing virtual synchrony. J. Comput. Syst. Sci. **96**, 50–73 (2018)
15. Dolev, S., Petig, T., Schiller, E.M.: Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. Algorithmica **85**(1), 216–276 (2023)
16. Dolev, S., Hendin, A., Herlihy, M., et al.: Self-stabilizing replicated state machine coping with byzantine and recurring transient faults. CoRR abs/2506.12900 (2025)
17. Dubois, S., Tixeuil, S.: A taxonomy of daemons in self-stabilization. CoRR abs/1110.0334 (2011)
18. Duvignau, R., Raynal, M., Schiller, E.M.: Self-stabilizing byzantine fault-tolerant repeated reliable broadcast. Theor. Comput. Sci. **972**(114), 070 (2023)

19. Duvignau, R., Raynal, M., Schiller, E.M.: Self-stabilizing multivalued consensus in the presence of byzantine faults and asynchrony. Theor. Comput. Sci. **1039**(115), 184 (2025)
20. Georgiou, C., Lundström, O., Schiller, E.M.: Self-stabilizing snapshot objects for asynchronous failure-prone networked systems. Theor. Comput. Sci. **1075**(115), 929 (2026)
21. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Trans. Software Eng. **17**(9), 911–921 (1991)
22. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
24. Landes, T.: Dynamic vector clocks for consistent ordering of events in dynamic distributed applications. In: Arabnia, H.R. (ed.) Parallel and distributed processing techniques and applications & conference on real-time computing systems and applications, pp. 31–37. CSREA Press, PDPTA (2006)
25. Landsiedel, O., Schiller, E.M., Tomaselli, S.: Libreplay: deterministic replay for bug hunting in sensor networks. In: Abdelzaher, T.F., Pereira, N., Tovar, E. (eds.) Wireless Sensor Networks-12th European Conference, EWSN. Lecture notes in computer science, vol. 8965, pp. 258–265. Springer (2015)
26. Lundström, O., Raynal, M., Schiller, E.M.: Self-stabilizing uniform reliable broadcast. In: Georgiou, C., Majumdar, R. (eds.) Networked Systems-8th International Conference, NETYS. Lecture notes in computer science, vol. 12129, pp. 296–313. Springer (2020)
27. Lundström, O., Raynal, M., Schiller, E.M.: Brief announcement: self-stabilizing total-order broadcast. In: Devismes, S., Petit, F., Altisen, K., et al. (eds.) Stabilization, safety, and security of distributed systems-24th International Symposium, SSS. Lecture notes in computer science, vol. 13751, pp. 358–363. Springer (2022)
28. Lundström, O., Raynal, M., Schiller, E.M.: Self-stabilizing multivalued consensus in asynchronous crash-prone systems. Theor. Comput. Sci. **1022**(114), 886 (2024)
29. Malkhi, D., Terry, D.B.: Concise version vectors in WinFS. Distrib. Comput. **20**(3), 209–219 (2007)
30. Peleg, D., Wool, A.: The availability of quorum systems. Inf. Comput. **123**(2), 210–223 (1995)
31. Salem, I., Schiller, E.M.: Practically-self-stabilizing vector clocks in the absence of execution fairness. CoRR abs/1712.08205 (2017)
32. Salem, I., Schiller, E.M.: Practically-self-stabilizing vector clocks in the absence of execution fairness. In: Networked Systems NETYS, pp. 318–333. (2018)
33. Schneider, M.: Self-stabilization. ACM Comput. Surv. **25**(1), 45–67 (1993)
34. Shapiro, M., Preguiça, N.M., Baquero, C.: Conflict-free replicated data types. In: Stabilization, Safety, and Security of Distributed Systems SSS, pp. 386–400. (2011)
35. Skeen, D.: Nonblocking commit protocols. In: ACM SIGMOD International Conference on Management of Data, pp. 133–142. (1981)
36. Stomp, F.A.: Structured design of self-stabilizing programs. In: Second Israel Symposium on Theory of Computing Systems, pp. 167–176. ISTCS. IEEE Computer Society (1993)
37. Tanenbaum, A.S., van Steen, M.: Distributed systems - principles and paradigms, 2nd Edition. Pearson Education (2007)
38. Varghese, G.: Compositional proofs of self-stabilizing protocols. In: Ghosh, S., Herman, T. (eds.) 3rd Workshop on Self-stabilizing Systems, pp. 80–94. Carleton University Press (1997)