

Practically-Self-Stabilizing Vector Clocks in the Absence of Execution Fairness*

(technical report)

Iosif Salem Elad M. Schiller

Abstract

Vector clock algorithms are basic wait-free building blocks that facilitate causal ordering of events. As wait-free algorithms, they are guaranteed to complete their operations within a finite number of steps. Stabilizing algorithms allow the system to recover 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. We present the first, to the best of our knowledge, stabilizing vector clock algorithm for asynchronous crash-prone message-passing systems that can recover in a *wait-free manner* after the occurrence of transient faults. In these settings, it is challenging to demonstrate a finite and *wait-free* recovery from (communication and crash failures as well as) *transient faults*, bound the message and storage sizes, deal with the removal of all stale information *without blocking*, and deal with counter overflow events (which occur at different network nodes concurrently).

We present an algorithm that never violates safety in the absence of transient faults and provides bounded time recovery during fair executions that follow the last transient fault. The novelty is that in the absence of execution fairness, the algorithm guarantees a bound on the number of times in which the system might violate safety (while existing algorithms might block forever due to the presence of both transient faults and crash failures).

Since vector clocks facilitate a number of elementary synchronization building blocks (without requiring remote replica synchronization) in asynchronous systems, we believe that our analytical insights are useful for the design of other systems that cannot guarantee execution fairness.

*Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden. {iosif,elad}@chalmers.se

1 Introduction

Context and Motivation. Vector clocks allow reasoning about causality among events in distributed systems, for example, when constructing distributed snapshots [17]. Shapiro et al. [24] showed that vector clocks are building blocks of several conflict-free replicated data types (CRDTs). CRDTs are distributed data structures that can be shared among many replicas in asynchronous networks. All replica updates occur independently and achieve *strong eventual consistency* without using mechanisms for synchronization [25] or roll-back.

The industrial use of CRDTs includes globally distributed databases, such as the ones of Redis, Riak, Bet365, SoundCloud, TomTom, Phoenix, and Facebook. Some of these databases have around ten million concurrent users, ten thousand messages per second, store large volumes of data, and offer very low latency. However, while both the literature and the users demonstrate that large-scale decentralized systems can benefit from the use of CRDTs in general and vector clocks in particular, the relationship between fault-tolerance and strong eventual consistency has not received sufficient attention. Providing higher robustness degrees to CRDTs is nevertheless imperative for ensuring the availability and safety of these systems.

Providing robustness in the presence of unexpected failures, i.e., the ones that are not included in the fault model, is challenging, especially in the absence of synchrony, mechanisms for synchronization, or roll-back. In such systems, it is difficult to: (A) provide unbounded storage and message size, (B) model all possible failures, and (C) guarantee periods in which all nodes are up and connected.

The goal of this paper is the design of a highly fault-tolerant distributed algorithm for vector clocks in large-scale asynchronous message passing systems. In particular, we propose the first, to the best of our knowledge, practically-self-stabilizing algorithm for vector clocks that: (I) uses strictly bounded storage and message size, (II) deals with a relevant set of failures (i.e., a fault model) as well as with unexpected failures (i.e., failures that are not considered by the fault model), and (III) the algorithm does not require synchronization guarantees, nor uses mechanisms for synchronization or roll-back *even during the period of recovery from unexpected failures*.

Fault Model. We consider asynchronous message-passing systems that are prone to the following failures [16]: (a) crash failures of nodes (no recovery after crashing), (b) nodes that can crash and then perform an undetectable restart, i.e., resume with the same state as before crashing (without knowing explicitly that a crash has ever occurred), but possibly having lost incoming messages in between, and (c) packet failures, such as omission, duplication, and reordering. In addition to these benign failures, we consider *transient faults*, i.e., any temporary violation of assumptions according to which the system and network were designed to behave, e.g., the corruption of the system state due to soft errors. We assume that these transient faults arbitrarily change the system state in unpredictable manners (while keeping the program code intact).

Moreover, since these transient faults are rare, the system model assumes that all transient faults occurred before the start of the system run.

Design criteria. Dijkstra [8] requires self-stabilizing systems, which may start in an arbitrary state, to return to correct behavior within a bounded period. Asynchronous 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. This is true for any system, and in particular, for Dijkstra’s self-stabilizing systems [8], which are required to remove, within a bounded time, all stale information whenever they appear. Here, the scheduler acts as an adversary that has a bounded number of opportunities to disrupt the system. However, this adversary never reveals *when* it will disrupt the system. Against such unfair adversaries, systems cannot specify when they will be able to remove all stale information and thus they cannot fulfill Dijkstra’s requirements.

Pseudo-self-stabilization [6] deals with the above inability by bounding the number of times in which the system violates safety. We consider the newer criteria of *practically-self-stabilizing systems* [2, 14, 4, 12] that can address additional challenges. 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 for an unbounded number of times after that overflow event. This challenge is greater when there is no elegant way to maintain an order among the different counter values, say, by wrapping around to zero upon counter overflow. Existing attempts to address this challenge use non-blocking resets in the absence of faults, as described in [3]. In case faults occur, the system recovery requires the use of a synchronization mechanism that, at best, blocks the system until the scheduler becomes fair. We note that this assumption contradicts our fault model as well as the key liveness requirement for recovery after the occurrence of transient faults.

Without fair scheduling, a system that takes an extraordinary (or even an infinite) number of steps is bound to break any ordering constraint, because unfair schedulers 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* [14, 12], say, 2^b (where $b = 64$ or an even a larger integer, as long as a constant number of bits can represent it). Practically-self-stabilizing systems [2, 4, 12] require a bounded number of safety violations during any practically infinite period of the system run. For such systems, we propose an algorithm for vector clocks that recovers after the occurrence of transient faults (as well as all other failures considered by our fault model) without assuming synchrony or using synchronization mechanisms. We refer to the latter as a wait-free recovery from transient faults. We note that the concept of *practically-self-stabilizing systems* is named by the concept of *practically infinite executions* [14].

To the end of providing safety (and independently of the practically-self-stabilizing algorithm), the application can use a synchronization mechanism

(similar to [2, 4, 12, 19]). The advantage here is that the application can selectively use synchronization only when needed (without requiring the entire system to be synchronous or blocking after the occurrence of transient faults).

Vector clocks. Logical and vector clocks [20, 15, 22] capture chronological relationships in decentralized systems without accessing synchronization mechanisms, such as synchronized clocks and phase-based commit protocols [25, 4]. A common (non-self-stabilizing and unbounded) way for implementing vector clocks is to let the nodes maintain a local copy of the vector $V[]$, such that each of the N system nodes has a component, e.g., $V[i]$ is the component of node p_i . Upon the occurrence of a local event, p_i increments $V_i[i]$, and sends an update message $m = \langle V[] \rangle$. Upon m 's arrival to node p_j , the latter merges the events counted in $V[]$ and $m.V[]$ by assigning $V[j] \leftarrow \max(V[j], m.V[j])$ for each component $V[j]$. One can define the relation \leq_C as a partial order, where V and W are N -size integer vectors and $(V \leq_C W) \iff (\forall x \in \{1, \dots, N\}, V[x] \leq W[x])$. The relation \leq_C is used to show causality between two events by checking if the corresponding vector clocks are comparable in \leq_C .

We note that there exist approaches for improving the scalability and efficiency of vector clocks that offer bounded size vectors (instead of linear) or approximations [23, Section 7]. These approaches build on, implement, or provide similar semantics to the standard N -size vector definition of a vector clock. Thus, in this paper we focus on the definition of a vector clock as an N -size vector.

The studied question. How can non-failing nodes dependably reason about event causality? We interpret the provable dependability requirement to imply (1) bounded message size and node storage, (2) fault-tolerance independently of synchrony assumptions or synchronization operations, and (3) the system to be practically-self-stabilizing (without fair scheduling).

Related work. Bounded non-stabilizing solutions exist in the literature [1, 21]. Self-stabilizing resettable vector clocks [3] 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 [3], the system uses non-blocking resets. Nevertheless, the presence of faults can bring the algorithm in [3] 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 corruption of the system state.

The authors of [3] also discuss the possibility to use global snapshots for the sake of providing better complexity measures. They rule out this approach because it can change the communication patterns (in addition to the use of 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., [7, Section 6] declared that this is an open problem.

There are practically-self-stabilizing algorithms for solving agreement [14, 4], state-machine replication [4, 12], and shared memory emulation [5]. 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. [12, 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 for implementing 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 Contributions. We present an important building block for dependable large-scale decentralized systems that need to reason about event causality. In particular, we provide a practically-self-stabilizing algorithm for vector clocks that does not require synchrony assumptions or synchronization mechanisms. Concretely, we present, to the best of our knowledge, the first solution that:

(i) Deals with a wide range of failures. The studied asynchronous systems are prone to crash failures (with or without undetectable restarts) and communication failures, such as packet omission, duplication, and reordering failures.

(ii) Uses bounded storage and message size. Our solution considers $3N$ integers and two labels [12] per vector, where N is the number of nodes. Each label has $\mathcal{O}(N^3)$ bits. Since all counters share the same two labels, we propose elegant techniques for dealing with the challenge of concurrent overflows (Section 5). We overcome the difficulties of making sure that no counter increment is ever “lost” even though there is an unbounded period in which these increments are associated with up to N different versions of the vector clock.

(iii) Deals with transient faults and unfair scheduling. Theorem 7.1 proves recovery within $\mathcal{O}(N^8\mathcal{C})$ safety violations in a practically-infinite execution in a wait-free manner after the occurrence of transient faults, which is our complexity measure for practically-self-stabilizing systems, where N is the number of nodes in the system and \mathcal{C} is an upper bound on the channel capacity.

We believe that our approaches for providing items (i)–(iii) are useful for the design of other practically-self-stabilizing systems.

Paper organization. In Section 2 we present the design criteria. In Section 3 we give an overview of relevant labeling schemes and in Section 4 we present an interface to a labeling scheme. Then, we present novel techniques

(Section 5), upon which we base our algorithm (Section 6) and proofs (Section 7).

2 System Settings

The system includes a set of processors $P = \{p_1, \dots, p_N\}$, which are computing and communicating entities that we model as finite state-machines. Processor p_i has an identifier, i , that is unique in P . Any pair of active processors can communicate directly with each other via their bidirectional communication channels (of bounded capacity per direction, $\mathcal{C} \in \mathbb{N}$, which, for example, allows the storage of at most one message). That is, the network's topology is a fully-connected graph and each $p_i \in P$ has a buffer of finite capacity \mathcal{C} that stores incoming messages from p_j , where $p_j \in P \setminus \{p_i\}$. Once a buffer is full, the sending processor overwrites the buffer of the receiving processor. We assume that any $p_i, p_j \in P$ have access to $channel_{i,j}$, which is a self-stabilizing end-to-end message delivery protocol (that is reliable FIFO) that transfers packets from p_i to p_j . Note that [11, 13] present a self-stabilizing reliable FIFO message delivery protocol that tolerates packet omissions, reordering, and duplication over non-FIFO channels.

The interleaving model. 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*. We assume the interleaving model, where steps are executed atomically; one step at a time. Input events refer to packet receptions or a periodic timer that can, for example, trigger the processor to broadcast a message. Note that the system is asynchronous and the algorithm that each processor is running is oblivious to the timer rate. Even though the scheduler can be adversarial, we assume that each processor's local scheduler is fair, i.e., the processor alternates between completing send and receive operations (unless the processor's communication channels are empty). Note that a message that a processor p_i needs to send to its neighbors takes $N - 1$ consecutive steps of p_i (the execution might include steps of other processors in between), since each step can include at most one send (or receive) operation.

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). We assume that if p_i sends a message infinitely often to p_j , processor p_j receives that message infinitely often, i.e., the communication channels are fair. The term *system state* refers to a tuple of the form $c = (s_1, s_2, \dots, s_N)$, where each s_i is p_i 's state (including messages in transit to p_i). We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \dots$ 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 system state c_0 , is obtained from the preceding system state c_x by the execution of step a_x .

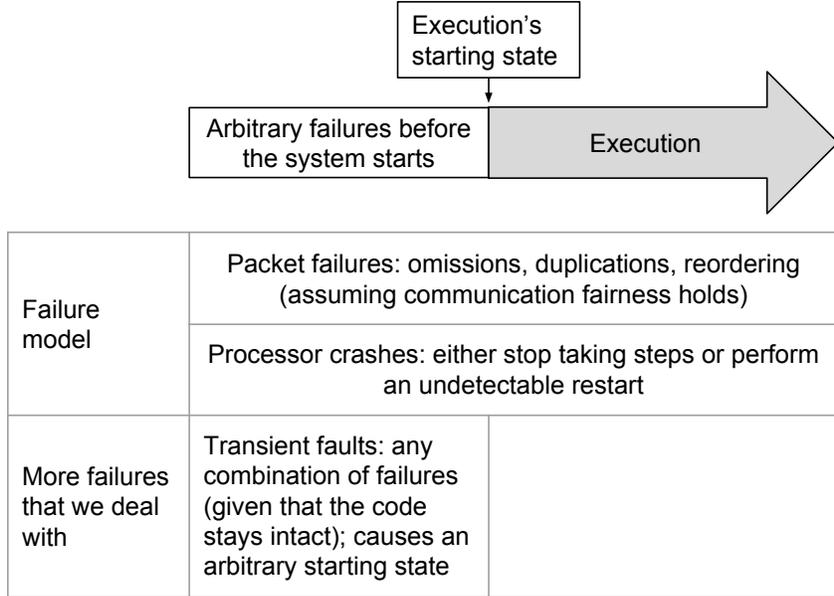


Figure 1: Illustration of the failure model and of transient faults.

Active processors, processor crashes, and undetectable restarts. At any point and without warning, p_i is prone to a crash failure, which causes p_i to either forever stop taking steps (without the possibility of failure detection by any other processor in the system) or to perform an undetectable restart in a subsequent step [16]. In case processor p_i performs an undetectable restart, it continues to take steps by having the same state as immediately before crashing, but possibly having lost the messages that other processors sent to p_i between crashing and restarting. Processors know the set P , but have no knowledge about the number or the identities of the processors that never crash.

We assume that transient faults occur only before the starting system state c_0 , and thus c_0 is arbitrary. Since processors can crash after c_0 , the executions that we consider are not fair [10]. We illustrate the failures that we consider in this paper in Figure 1.

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 . Note that the fact that a processor is active during an infinite execution does not give any guarantee on when or how often it takes steps. Thus, there might be an arbitrarily long (yet finite) subexecution R' of R , such that a processor is active in R but not in R' . Therefore, processors that crash and never restart during an infinite execution R are not active throughout R .

Execution operators: concatenation \circ and segment \sqsubseteq . 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 .

Execution length, practically infinite, and the \ll (significantly less) relation. To the end of defining the stabilization criteria, we need to compare the number of steps that violate safety in a finite execution R with the length of R . In the following, we define how to compare finite executions and sets of states according to their size.

We say that the length of a finite execution $R = c_0, a_0, c_1, a_1, \dots, c_{x-1}, a_{x-1}$ is equal to x , which we denote by $|R| = x$. Let $MAXINT$ be an integer that is considered as a *practically infinite* [14] quantity for a system \mathcal{S} (e.g., the system's lifetime). For example, $MAXINT$ can refer to 2^b (where $b = 64$ or larger) sequential system steps (e.g., single send or receive events). In this paper, we use \ll as a formal way of referring to the comparison of, say, N^c , for a small integer c , and $MAXINT$, such that N^c is an insignificant number when compared to $MAXINT$. Since this comparison of quantities is system-dependent, we give a modular definition of \ll below.

Let $\mathcal{L}_{\mathcal{S}}$ denote a system-dependent quantity that is practically-infinite for a system \mathcal{S} , such that for an integer $z \ll MAXINT$, we have that $\mathcal{L}_{\mathcal{S}} := z \cdot MAXINT$. For a system \mathcal{S} and $x \in \mathbb{N}$, we denote by $x \ll \mathcal{L}_{\mathcal{S}}$ the fact that x is significantly less than (or insignificant with respect to) $\mathcal{L}_{\mathcal{S}}$. We say that an execution R is of $\mathcal{L}_{\mathcal{S}}$ -scale, if there exists an integer $y \ll MAXINT$, such that $|R| = y \cdot MAXINT$ holds.

The design criteria of practically-self-stabilizing systems. We define the system's abstract task \mathcal{T} by a set of variables (of the processor states) and constraints, which we call the *system requirements*, in a way that defines a desired system behavior, but does not consider necessarily all the implementation details. We say that an execution R is a *legal execution* if the requirements of task \mathcal{T} hold for all the processors that take steps during R (which might be a proper subset of P). We denote the set of legal executions with LE . 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 $R \in \text{LE} \iff f_R = 0$). Note that the definition of LE allows executions of very small length, but our focus will be on finding maximal subexecutions $R^* \sqsubseteq R$ for a given $\mathcal{L}_{\mathcal{S}}$ -scale execution R , such that $R^* \in \text{LE}$. Definitions 2.1, 2.2 and 2.3 specify our stabilization criteria.

Definition 2.1 (Strong 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.*

Definition 2.2 (Pseudo Self-stabilization). *For every infinite execution R , $f_R = f(R, N) \in \mathbb{N}$, where f_R is the complexity measure.*

Definition 2.3 (Practically-self-stabilizing System). *For every infinite execution R , and for every \mathcal{L}_S -scale subexecution R' of R , $f_{R'} = f(R', N) \ll |R'|$, where $f_{R'}$ is the complexity measure.*

Problem definition (task requirement). We present a requirement (Requirement 1) which defines the abstract task of vector clocks. This requirement trivially holds for a fault-free system that can store unbounded values (and thus does not need to deal with integer overflow events). The presence of transient faults can violate these assumptions and cause the system to deviate from the abstract task, which LE specifies (through Requirement 1). In the following, we present Requirement 1 and its relation to causal ordering (Property 1).

We assume that each processor p_i is recording the occurrence of a new local event by incrementing the i -th entry of its vector clock. During a legal execution, we require that the processors count all the events occurring in the system, despite the (possibly concurrent) wrap around events. Hence, we require that the vector clock element of each (active) processor records all the increments done by that processor (Requirement 1). As a basic functionality, we assume that each processor can always query the value of its local vector clock. We say that an execution R^* is a legal execution, i.e., $R^* \in \text{LE}$, if Requirement 1 holds for the states of all processors that take steps during R^* .

Requirement 1 (Counting all events). Let R be an execution, p_i be an active processor, and $V_i^k[i]$ be p_i 's value in $c_k \in R$. For every active processor $p_i \in P$, the number of p_i 's counter increments 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 .

Causal precedence. We explain how faults and bounded counter values affect Requirement 1 and causal ordering. Let V and V' be two vector clocks, and $\text{causalPrecedence}(V, V')$ be a query which is true, if and only if, V causally precedes V' , i.e., V' records all the events that appear in V [26]. Then, V and V' are concurrent when $\neg\text{causalPrecedence}(V, V') \wedge \neg\text{causalPrecedence}(V', V)$ holds. We formulate the causal precedence property in Property 1. In a fault-free system with unbounded values, Requirement 1 trivially holds, since no wrap around events occur. That is, V causally precedes V' , if $V[i] \leq V'[i]$ for every $i \in \{1, \dots, n\}$ and $\exists_{j \in \{1, \dots, n\}} V[j] < V'[j]$ hold [26]. However, this is not the case in an asynchronous, crash-prone, and bounded-counter setting, where counter overflow events can occur. We present cases where Requirement 1 and Property 1 do not hold due to a counter overflow event in Example 2.4.

Property 1 (Causal precedence). For any two vector clocks V_i and V_j of two processors $p_i, p_j \in P$, $\text{causalPrecedence}(V_i, V_j)$ is true if and only if V_i causally precedes V_j .

Example 2.4. Consider two bounded vector clocks $V_i = \langle v_{i_1}, \dots, v_{i_N} \rangle$ and $V_j = \langle v_{j_1}, \dots, v_{j_N} \rangle$ of $p_i, p_j \in P$, such that upon a new event $p_k \in P$ 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, thus $V_i[i]$ wraps around to $V_i[i] = 0$, while $V_j[i] = MAXINT - 1$ remains. Then, $V_i[i]$ mistakenly indicates zero events for p_i ($V_i[i] = 0$) instead of $MAXINT$, i.e., Requirement 1 does 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 . \square

We remark that Requirement 1 is a necessary and sufficient condition for Property 1 to hold. Suppose that Requirement 1 does not hold, which means that it is not possible to count the events of a single processor between two states (e.g., as we showed in the previous example). This implies that it is not possible to compare two vector clocks, hence Property 1 cannot hold. Moreover, if Requirement 1 holds, then it is possible to compare how many events occurred in a single processor between two states, and by extension it is possible to compare all vector clock entries for two vector clocks. The latter is a sufficient condition for defining causal precedence (as in the fault-free unbounded-counter setting [26]).

In Section 5 we present our solution for computing $V_i^\ell[i] - V_i^k[i]$ for Requirement 1 (c_ℓ, c_k are states in an execution R and $p_i \in P$) and $causalPrecedence(V_i, V_j)$ for Property 1 in a legal execution. In Section 6 we present an algorithm for replicating vector clocks in the presence of faults and bounded-counters, which we prove to be practically-self-stabilizing in Section 7.

3 Background: Practically-self-stabilizing Labeling Schemes

In this section we give an overview of labeling schemes that can be used for designing an algorithm that guarantees Requirement 1. It is evident from Example 2.4 (Section 2) that a solution for comparing vector clock elements that overflow can be based on associating each vector clock element with a timestamp (or label, or epoch). This way, even if a vector clock element overflows, it is possible to maintain order by comparing the timestamps.

As a first approach for providing these timestamps, one might consider to use an integer counter (or sequence number), cn . We explain why this approach is not suitable in the context of self-stabilization. Any system has memory limitations, thus a single transient fault can cause the counter to quickly reach the memory limit, say $MAXINT$. The event of counter overflow occurs when a processor increments the counter cn , causing cn to encode the maximum value $MAXINT$. In this case, the solution often is that cn wraps around to zero.

Thus, this approach faces the same ordering challenges with the vector clock elements.

Existing solutions associate counters with *epochs* ℓ , which mark the period between two overflow events. A non-stabilizing representation of epochs can simply consider a, say, 64-bit integer. Upon the overflow of cn , the algorithm increments ℓ by one and nullifies cn . The order among the counters is simply the lexicographic order among the pairs $\langle \ell, cn \rangle$. With this approach, it is a challenge to maintain an order within a set of integers during phases of concurrent wrap around events at different processors. In the following we present more elegant solutions for bounded labeling schemes, that tolerate concurrent overflow events, transient faults, and the absence of execution fairness.

Bounded labeling schemes. Bounded labeling schemes (initiated in [18, 9], cf. [17, Section 2]) provide labeling of data and denote temporal relations. Given a bounded set of labels L , a bounded labeling scheme usually includes a partial or total order \prec_L over L and a function for constructing locally a new maximal label from L with respect to \prec_L , given a set of input labels. Labeling algorithms handle these labels such that the processors eventually agree, for example, on a maximal label. Since we consider processor crashes, a suitable labeling scheme should include a garbage collection mechanism that cancels obsolete labels, by possibly using label storage.

Practically-self-stabilizing bounded labeling schemes. Alon et al. [2] and Dolev et al. [12] present practically-self-stabilizing bounded-size labels. Whenever a counter cn reaches $MAXINT$, the algorithm by Alon et al. [2] replaces its current label ℓ with ℓ' , which at the moment of this replacement is greater than any label that appears in the system state. This means that immediately after the counter wraps around, the counter $\langle \ell', cn = 0 \rangle$ is greater than all system counters. In the remainder of this section, we give an overview of the labeling schemes of Alon et al. [2] (Section 3.1) and Dolev et al. [12] (Section 3.2).

3.1 The case of no concurrent overflow events

Alon et al. [2] address the challenge of always being able to introduce a label that is greater than any other previously used one. They present a two-player guessing game, between a finder, representing the algorithm, and a hider, representing an adversary controlling the asynchronous system that starts from an arbitrary state. Let M be the maximum number of labels that can exist in the communication channels, i.e., $M = \mathcal{C}N(N-1)$, where $N(N-1)/2$ is the number of bidirectional communication channels in the system and \mathcal{C} is the capacity in number of messages per channel (and hence labels).

The hider has a bounded size label set, \mathcal{H} , such that $|\mathcal{H}| \leq M \in \mathbb{N}$. The finder, who is oblivious to \mathcal{H} 's content, aims at obtaining a label ℓ that is greater than all of \mathcal{H} 's labels. To that end, the finder generates ℓ in such a way that

whenever the hider exposes a label $\ell' \in \mathcal{H}$, such that ℓ is not greater than ℓ' , \mathcal{H} has one less label that the finder is unaware of its existence. The hider may choose to include ℓ in \mathcal{H} as long as it makes sure that $|\mathcal{H}| \leq M$ by omitting another label from \mathcal{H} (without notifying the finder).

Label construction. A label component $\ell = (\text{sting}, \text{Antistings})$ is a pair, where $\text{sting} \in D$, $D = \{1, \dots, k^2 + 1\}$, $\text{Antistings} \subset D$, $|\text{Antistings}| = k$, and $k > 1$ is an integer. The order among label components is defined by the relation \prec_b , where $\ell_i \prec_b \ell_j \iff (\ell_i.\text{sting} \in \ell_j.\text{Antistings}) \wedge (\ell_j.\text{sting} \notin \ell_i.\text{Antistings})$. The function $\text{Next}_b(L)$ takes a set $L = \{\ell_1, \dots, \ell_\kappa\}$ of (up to) $k \in \mathbb{N}$ label components, and returns a newly created label component, $\ell_j = \langle s, A \rangle$, such that $\forall \ell_i \in L : \ell_i \prec_b \ell_j$, where $s \in D \setminus \cup_{i=1}^\kappa A_i$ and $A = \{s_1, \dots, s_\kappa\}$, possibly augmented by arbitrary elements of $D \setminus A$ when $|A| = \kappa < k$.

Label cancelation. Alon et al. [2] use the order \prec_b for which, during the period of recovery from transient faults, it can happen that ℓ_1, ℓ_2 , and ℓ_3 appear in the system and $\ell_1 \prec_b \ell_2 \prec_b \ell_3 \prec_b \ell_1$ holds. The finder breaks such cycles by *canceling* these label components so that the system (eventually) avoids using them. Alon et al. [2] implement labels (epochs) as pairs (ml, cl) , where ml is always a label component and cl is either \perp when (ml, cl) is legitimate (non-canceled) or a label component for which $cl \not\prec_b ml$ holds. Thus, the finder stores cl as an evidence of ml 's cancelation.

Keeping the number of stored labels bounded. Alon et al. [2] present a finder strategy that queues the most recent labels that the finder is aware of in a FIFO manner. They show a $2M$ bound on the queue size by pointing out that, if the finder queues (1) any label that it generates and (2) the ones that the hider exposes, the hider can surprise the finder at most M times before the queue includes all the labels in \mathcal{H} .

In detail, the algorithm gossips repeatedly its (currently believed) greatest label and stores the received ones in a queue of at most $2M$ labels, where $M = \mathcal{C}N(N - 1)$ is the maximum number of labels that the system can “hide”, i.e., one (currently believed) greatest label that each of the N processors has and \mathcal{C} (capacity) in each communication link. Upon arrival of label ℓ_i to ℓ_j such that $\ell_i \not\prec_b \ell_j$, processor p_j queues the arriving label ℓ_i , uses $\text{Next}_b()$ to create a new (currently believed) greatest legitimate label ℓ'_j and queue it as well. Alon et al. [2] show that, when only p_j may create new labels, the system stabilizes to a state in which p_j believes in a legitimate label that is indeed the greatest in the system. Note that the stabilization period includes at most M arrivals to p_j of labels ℓ_i that “surprise” p_j , i.e., $\ell_i \not\prec_b \ell_j$.

3.2 The case of concurrent overflow events

Alon et al. [2]'s labels allow, once a single label (epoch) ℓ is established, to order the system events using the counter (ℓ, cn) . Dolev et al. [12] extend

Alon et al. [2] to support concurrent *cn* overflow events, by including the label creator identity. This information facilitates symmetry breaking, and decisions about which label is the most recent one, even when more than one creator concurrently constructs a new label. Dolev et al. [12] make sure that active processors p_i remove eventually obsolete labels ℓ that name p_i as their creator (due to the fact that p_i indeed created ℓ , or ℓ was present in the system's arbitrary starting state). Note that the system's arbitrary starting state may include cycles of legitimate (not canceled) labels $\ell_1 \prec_b \ell_2 \prec_b \ell_3 \prec_b \ell_1$ that share the same creator, e.g., p_k . The algorithm by Dolev et al. guarantees cycle breaking by logging all labels that it observes and canceling any label that is not greater than its currently known maximal one.

Label construction. Dolev et al. [12] extend Alon et al.'s label component to (*creator, sting, Antistings*), where *creator* is the identity of the label creating processor, and *sting* as well as *Antistings* are as in [2] (Section 3.1). They use $=_{lb}$ to denote that two labels, ℓ_i and ℓ_j , are identical and define the relation $\ell_i \prec_{lb} \ell_j \iff (\ell_i.\text{creator} < \ell_j.\text{creator}) \vee (\ell_i.\text{creator} = \ell_j.\text{creator} \wedge ((\ell_i.\text{sting} \in \ell_j.\text{Antistings}) \wedge (\ell_j.\text{sting} \notin \ell_i.\text{Antistings})))$. The labels ℓ_i and ℓ_j are *incomparable* when $\ell_i \not\prec_{lb} \ell_j \wedge \ell_j \not\prec_{lb} \ell_i$ (and comparable otherwise).

Label cancellation. Dolev et al. consider label ℓ to be obsolete when there exists another label $\ell' \not\prec_{lb} \ell$ of the same creator. In detail, ℓ_i cancels ℓ_j , if and only if, ℓ_i and ℓ_j are incomparable, or if $\ell_i.\text{creator} = \ell_j.\text{creator} \wedge \ell_i.\text{sting} \in \ell_j.\text{Antistings} \wedge \ell_j.\text{sting} \notin \ell_i.\text{Antistings}$, i.e., ℓ_i and ℓ_j have the same creator but ℓ_j is greater than ℓ_i according to the \prec_b order.

The abstract task of Dolev et al.'s labeling scheme. Each processor presents to the system a label that represents the *locally perceived maximal label*. During a legal execution, as long as there is no explicit request for a new label, all processors refer to the same locally perceived maximal label, which we refer to as the *globally perceived maximal label*. Moreover, it cannot be the case that processor p_i has a locally perceived maximal label ℓ_i and another processor $p_j \in P$ (possibility $i = j$) stores a label ℓ_j that is incomparable to ℓ_i , greater than ℓ_i , or that cancels ℓ_i (where ℓ_j is not necessarily p_j 's locally perceived maximal label).

We note that when the system starts in an arbitrary state, the (active) processors might refer to a globally perceived maximal label that is not the maximal label in the system. This is due to the fact that in practically-self-stabilizing systems there could be a non zero number of deviations from the abstract task during any practically infinite execution (Definition 2.3). In detail, Dolev et al. [12, Algorithm 2] store the locally perceived maximal label of processors p_i at $\text{max}_i[i]$ and demonstrate the satisfaction of the above abstract task in [12, Theorem 4.2].

Keeping the number of stored labels bounded. Whenever p_k is active, it will eventually queue all of the labels that it has created, cancel them, and generate a label that is greater than them all. However, in case that p_k is inactive, the algorithm uses the active processors to prevent the asynchronous system from endlessly using labels that belong to cycles. Dolev et al. [12] show that p_k 's cycle may include at most $M + N$ labels, where M is the number of labels that can appear in the communication channels and N is the number of processors. Therefore, $p_i \in P$ needs to queue $M + N$ labels for any other p_k , so that p_i could remove the label cycles once their creator p_k becomes inactive. Moreover, p_i needs a queue of $2(MN + 2N^2 - 2N) + 1$ labels ℓ (for which $\ell.creator = i$) until it can be sure to have the maximal label. These bounds give the maximum number of labels that p_i can either *adopt* (use as its maximal label) or *create* when it does not store a maximal label, throughout any execution. Next, we provide the algorithm details and use these details when justifying our bounds (Section 4).

Variables. Each processor p_i maintains an N -size vector of labels max_i , where $max_i[i]$ is p_i 's local maximal \prec_{lb} -label and $max_i[j]$ is the latest *legitimate*, i.e., not canceled, label that p_i received most recently from p_j . Also, p_i maintains an N -size vector $storedLabels_i$ of queues that logs the labels that p_i has observed so far, which p_i sorts by their label creator. That is, $storedLabels_i[j]$ queues label ℓ , such that (i) p_i has received ℓ from an arbitrary processor, (ii) ℓ 's creator is $p_j \in P$, i.e., $\ell.creator = j$, (iii) there are no duplicates of ℓ in $storedLabels_i[j]$, and (iv) ℓ is either canceled or every other label in $storedLabels_i[j]$ is canceled.

The algorithm. Processor p_i gossips repeatedly its \prec_{lb} -greatest label, $max_i[i]$, and stores the arriving labels in $storedLabels_i[j]$, where $j = \ell.creator$. The algorithm ensures that $storedLabels_i[j]$ stores at most one legitimate label by canceling any label ℓ' using label ℓ when (1) they are incomparable or (2) they share the same creator and $\ell' \prec_{lb} \ell$. Moreover, p_i makes sure that, for any $j \in [1, N]$, the \prec_{lb} -greater label $max_i[j]$ is indeed greater than any other label in $storedLabels_j[i]$. In case it does not, p_i selects the \prec_{lb} -greatest legitimate label in $storedLabels_i$, and if there is no such legitimate label, p_i creates a new label via $nextLabel()$, which is an extension of $Next_b()$ that also includes the label creator, p_i . Dolev et al. [12] bound the size of $storedLabels_i[j]$, for $j \neq i$ by $N + M$ and the size of $storedLabels_i[i]$ by $2(MN + 2N^2 - 2N) + 1$.

4 Composing practically-self-stabilizing labeling algorithms and the interface to Dolev et al. [12] labeling scheme

In this section we present a framework for composing any practically-self-stabilizing labeling algorithm (server) with any other practically-self-stabilizing algorithm (client). By this composition we obtain a compound algorithm

with combined properties. Then, we discuss the challenges in composing practically-self-stabilizing algorithms, with respect to the composition of strong self-stabilizing algorithms. Moreover, we present an interface to a labeling algorithm that facilitates our composition approach. The interface is also used by the client algorithm to query the state of the labeling algorithm, send messages, or to request the labeling algorithm to cancel a label. We show how this interface is implemented by the practically-self-stabilizing labeling scheme of Dolev et al. [12, Algorithm 2], which we use in our solutions (Section 5) and algorithm (Section 6). We end the section by discussing the stabilization guarantees of the compound algorithm.

Composition with a practically-self-stabilizing labeling algorithm.

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. 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 start to also stabilize. This way, the compound algorithm obtains more complex guarantees than the individual algorithms.

We detail our composition approach which we illustrate in Figure 2. In the following, we refer to the computations of a step excluding the send or receive operation, as the step’s *invariant check*, which possibly includes updates of local variables. Our approach for composing practically-self-stabilizing algorithms assumes that the messages of the client algorithm are piggybacked by the ones of the server, and that the server algorithm can send any message independently. Also, we assume that the communication among processors relies on a self-stabilizing end-to-end protocol, such as the ones in [11, 13].

A step that includes a send operation. 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 Figure 2, respectively). We assume that the client algorithm can request a change in the labeling (server) algorithm’s state, e.g., a label cancellation, but this change is performed by the labeling algorithm (cf. label cancellation in Figure 2). 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. Figure 2). 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.

A step that includes a receive operation. Upon the arrival of a message m by the labeling (server) algorithm (part 4 in Figure 2), the server al-

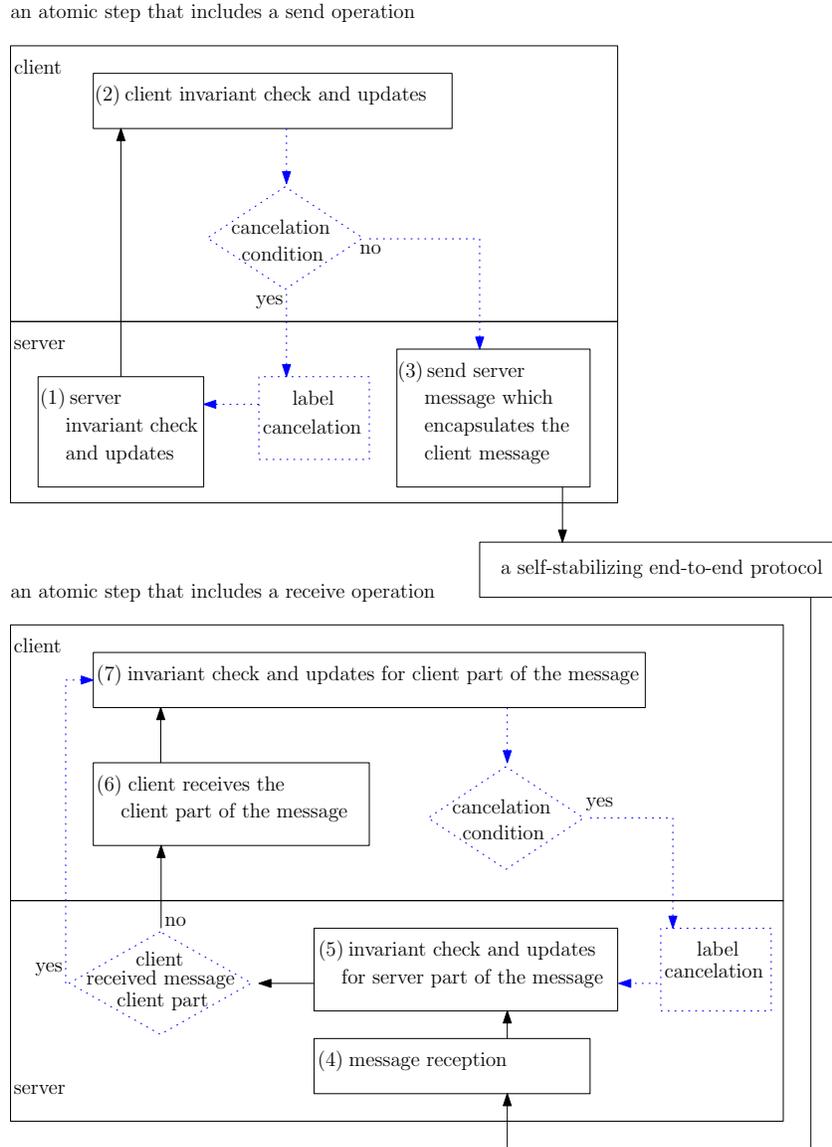


Figure 2: Composition of the server and the client algorithms. The normal lines denote the composition parts that are common in both strong and practically-self-stabilizing algorithms. The dotted blue lines show the computations in the composition of practically-self-stabilizing algorithms, that are 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.

gorithm performs the server invariant check and updates on the server part of the message (part 5 in Figure 2). Then, the server algorithm raises a message reception event for the client algorithm (part 6 in Figure 2), 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 Figure 2), 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 Figure 2 repeat.

Challenges in composition of practically-self-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. Figure 2).

An interface to a labeling algorithm and its implementation by the labeling algorithm of Dolev et al. [12, Algorithm 2]. We detail an interface to a labeling (server) algorithm in order to facilitate composition with a client algorithm. The functions of the interface allow the labeling algorithm to do its invariant check and updates. They also allow the client algorithm to query the state of the labeling algorithm without changing the labeling algorithm’s state, except for the function *cancel()*. The function *cancel()* changes the labeling algorithm’s state by canceling a label. Moreover, we explain how the Dolev et al. labeling algorithm [12, Algorithm 2] (cf. Section 3.2) implements the functions of this interface. These functions are also used by the shared counter algorithm in [12, Algorithm 3]. We note that [12, Algorithm 3] relies on synchronization mechanisms, but the labeling algorithm [12, Algorithm 2] does not rely on synchronization mechanisms, and hence it is suitable for our solution.

- *labelBookkeeping()*: **server invariant check and updates.** This function 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 Figure 2). It is intended to be called in every step of the client algorithm, and thus facilitates the composition of the two algorithms.

In [12, Algorithm 2], when calling *labelBookkeeping()* without arguments, [12, Algorithm 2, lines 21 to 28] perform the server invariant check and updates (part 1 of Figure 2). When calling *labelBookkeeping(m, j)*, [12, Algorithm 2, lines 19 to 28] process a message m that arrived from a processor $p_j \in P$ (parts

4 and 5 of Figure 2). The (mutable) function *labelBookkeeping()* is an alias to *process()* [12, Algorithm 3, line 2].

- ***isStored()* and *isCanceled()*: querying whether a label is stored or canceled.** Given a label ℓ , the (immutable, i.e., its value cannot change) predicate *isStored*(ℓ) checks whether ℓ appears in the label storage of the labeling algorithm. The (immutable) predicate *isCanceled*(ℓ) checks whether ℓ is canceled.

In [12, Algorithm 2], *isStored*(ℓ) returns true, if and only if $\ell \in \text{storedLabels}[j]$, such that $p_j \in P$ is ℓ 's creator. Also, *isCanceled*() is an exact alias to *legit*() in [12, Algorithm 2, line 6].

- ***getLabel()*: retrieving the largest label.** The (immutable) function *getLabel*() returns the largest locally stored label.

In [12, Algorithm 2], *getLabel*() returns the largest locally stored label with respect to the partial order of labels \prec_{lb} (cf. Section 3.2). In detail, that label is stored in *max_i[i]* (cf. lines 27 and 28 of [12, Algorithm 2]).

- ***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. Figure 2). 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 . Moreover, the function *encapsulate*() facilitates piggybacking of a message of the labeling algorithm with the one of the composed (client) algorithm (facilitating part 3 of Figure 2). 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* = $\langle \text{sentMax}, \bullet \rangle$ and m_{client} be the server, and respectively, the client part of an outgoing message of the compound algorithm. In [12, Algorithm 2], the server message is $m_{server} = \langle \langle \text{sentMax}, \bullet \rangle, m_{client} \rangle$ [12, Algorithm 2] and *encapsulate*(m_{client}) returns the value m_{server} . Moreover, the (immutable) function *legitMsg*(m_{server}, ℓ) tests the consistency of an arriving label ℓ with *serverPart* of the server message m_{server} . That is, the predicate *legitMsg*(m_{server}, ℓ) returns the value of $\ell = \text{sentMax}$.

- ***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. 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. Figure 2). Let ℓ and ℓ' be two labels, such that ℓ' cancels ℓ according to the scheme's label order. Then, when the client algorithm calls the

(mutable) function $\ell.cancel(\ell')$, the labeling algorithm marks ℓ as canceled (by ℓ').

In [12, Algorithm 2], in case $\ell' \not\prec_{lb} \ell$ holds, p_i marks ℓ as canceled by ℓ' by calling $\ell.cancel(\ell')$ (cf. label cancelation definition in Section 3.2). In detail, the function $cancel()$ is an alias to $cancelExhausted()$ [12, Algorithm 3, line 10].

Preserving the stabilization 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. Figure 2). 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 Section 6 (Algorithm 1) for the vector clock problem is composed with the labeling algorithm of Dolev et al. [12, Algorithm 2] through the interface that we presented in this section. In Section 7 we show that the algorithm that we propose for the vector clock problem preserves the labeling algorithm’s stabilization guarantees.

5 Vector Clock Pairs: operations, invariants, and event counting

In 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 Requirement 1. We show how to merge two (vector clock) pairs (Section 5.1), and use this construction for counting the events of a single processor and computing the query $causalPrecedence()$, which we defined in Section 2 (Section 5.2). In Section 6 we use the vector clock pairs for designing a practically-self-stabilizing algorithm with respect to the abstract task that Requirement 1 defines (cf. Section 2).

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. [12] labeling scheme (Section 3.2), 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 $(I.m - I.o) \pmod{MAXINT}$ 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 $VC(Z) := (Z.curr.m - Z.curr.o) \pmod{MAXINT}$ 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.

Starting a vector clock pair. The first value of a pair Z is $\langle\langle\ell, zrs, zrs\rangle, \langle\ell, zrs, zrs\rangle\rangle$, where $\ell := getLabel()$ is the local maximal label and $zrs := (0, \dots, 0)$ 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.

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 [12] is in $\mathcal{O}(N^3)$, this linear improvement is significant.

$$exhausted(Z) \iff \sum_{k=1}^N (Z.curr.m[k] - Z.curr.o[k]) \geq MAXINT - 1 \quad (1)$$

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., $Z.curr.l$ and $Z.prev.l$, 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., $VC(Z') = Z.curr.m - Z.curr.m = (0, \dots, 0)$ 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.

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.

5.1 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.

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

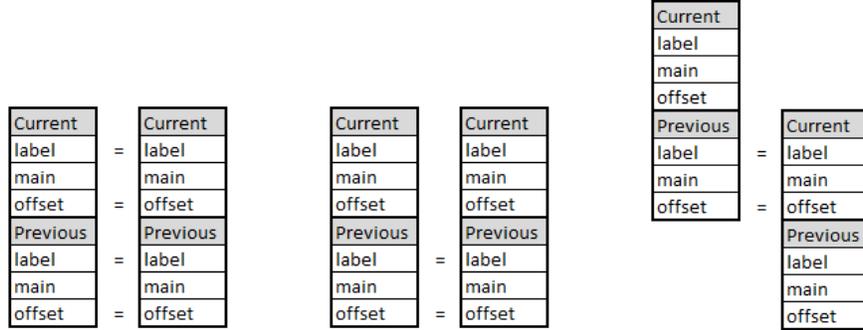
$$\begin{aligned} labelsOrdered(Z) \iff & ((Z.prev.l \prec_{lb} Z.curr.l \wedge isCanceled(Z.prev.l)) \vee \\ & (Z.prev.l = Z.curr.l \wedge \neg isCanceled(Z.curr.l))) \end{aligned} \quad (2)$$

The $=_{\ell,o}$ and $<_{\ell,o}$ relations. We define the relations $=_{\ell,o}$ and $<_{\ell,o}$ to be able to compare and order 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 [12]. 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 order $\langle \ell_1, m_1, o_1 \rangle <_{\ell,o} \langle \ell_2, m_2, o_2 \rangle \iff \ell_1 <_{lb} \ell_2 \vee (\ell_1 = \ell_2 \wedge o_1 <_{lex} o_2)$ 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 $<_{lb}$ and there exists a maximum label among them.

Pivot existence. Condition 3 tests the pair merging feasibility (Figure 3). 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 $itm \in \{curr, prev\}$ (Figure 3a), i.e., there was no vector clock exhaustion, or
- (b) Z and Z' match in their *prev*, i.e., $Z.prev =_{\ell,o} Z'.prev$ (figure 3b), 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., $Z.curr =_{\ell,o} Z'.prev \vee Z.prev =_{\ell,o} Z'.curr$ (Figure 3c),



(a) Condition 3 holds because the two pairs differ only by their *curr.main* fields (no wrap-around).

(b) Condition 3 holds because the two pairs match in their *prev* item (the pairs had wrapped around concurrently).

(c) Condition 3 holds because $Z.prev$ and $Z'.curr$ differ only by their main filed (Z has wrapped around).

Figure 3: Conditions for merging two given (vector clock) pairs; Z (on the left) and Z' (on the right).

i.e., one vector clock was exhausted (assuming that case (a) was true before exhaustion).

We refer to the common item between Z and Z' as the *pivot* item.

$$existsPivot(Z, Z') \iff Z.prev =_{\ell, o} Z'.prev \vee Z.curr =_{\ell, o} Z'.prev \vee Z.prev =_{\ell, o} Z'.curr \quad (3)$$

Merging two (vector clock) pairs. Two vector clocks Z and Z' can be merged when there exists a pivot item, i.e., $existsPivot(Z, Z')$ holds (Figure 3). The $<_{\ell, o}$ -maximum pivot item, pvt , in Z and Z' , provides a reference point when merging Z and Z' , because it refers to a point in time from which both Z and Z' had started counting their events. 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' (Figure 3), and choose Z (the first input argument) when symmetry exists (figures 3a and 3b). 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 $i \in \{1, \dots, N\}$, 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 Equation 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' . The aggregation step sets $output[i] = \max\{newEvents(X, pivot)[i] \mid X \in \{Z, Z'\}\} + pivot[i] \pmod{MAXINT}$, for every $i \in \{1, \dots, N\}$.

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

5.2 Event counting and causal precedence

In this section we present our implementation of the queries about counting the events of a single active processor (Requirement 1) and about causal precedence (Section 2), 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 2).

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 a legal execution, 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 5.1). Otherwise, in case there was no wrap around (cf. Figure 3a) or one wrap around (cf. Figure 3c) 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:

$$V_i^y[i] - V_i^x[i] = \begin{cases} VC(local_i^y)[i] - VC(local_i^x)[i], & \text{if } local_i^x \text{ and } local_i^y \text{ differ only on} \\ & \text{the field } curr.m \text{ (cf. Figure 3a)} \\ newEvents(local_i^y, local_i^y.prev)[i], & \text{if } local_i^x.curr =_{\ell, o} local_i^y.prev \\ & \text{(cf. Figure 3c)} \\ \perp, & \text{otherwise} \end{cases} \quad (5)$$

In Section 6 we propose Algorithm 1 and in Section 7 we show that it is practically-self-stabilizing with respect to Requirement 1 (cf. Section 2). Thus, during a legal execution the return value of $V_i^y[i] - V_i^x[i]$ in Equation 5 is never \perp .

In order to compute the query $causalPrecedence(Z, Z')$, which is true if and only if Z causally precedes Z' (Section 2), we follow a similar approach to merging pairs (Section 5.1). 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 Equation 6.

$$causalPrecedence(Z, Z') \Leftrightarrow existsPivot(Z, Z') \wedge (\forall_{i \in \{1, \dots, N\}} newEvents(Z, pivot)[i] \leq newEvents(Z', pivot)[i] \wedge \exists_{j \in \{1, \dots, N\}} newEvents(Z, pivot)[j] < newEvents(Z', pivot)[j]) \quad (6)$$

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, in a legal execution of Algorithm 1 (Section 6), $causalPrecedence(Z, Z')$ as we compute it in Equation 6 is true if and only if Z causally precedes Z' .

6 Practically-self-stabilizing Vector Clock Algorithm

We propose Algorithm 1 as a practically-self-stabilizing vector clock algorithm that fulfills Requirement 1 (Section 2). Algorithm 1 builds on the vector clock pair construction (Section 5), which uses a practically-self-stabilizing labeling scheme (cf. Section 3). Thus, Algorithm 1 is composed with a labeling algorithm using our composition approach and the interface in Section 4. In a nutshell, Algorithm 1 includes the procedures for (i) vector clock increments, (ii) checking the invariants of the local (vector clock) pair, e.g., vector clock exhaustion, and sending the local pair of a processor (*local*) to its neighbors (do-forever loop procedure), and (iii) merging an incoming vector clock pair with the local one. To that end, Algorithm 1 relies on the functions that we defined in Section 5.

Algorithm 1: Practically-self-stabilizing vector-clock replication, code for p_i

```
1 Constants:  $zrs := (0, \dots, 0)$ : 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 Interface:  $isStored()$ ,  $getLabel()$ ,  $legitMsg()$ ,  $encapsulate()$ ,  $cancel()$ ,
    $labelBookkeeping()$  (Section 4),  $newEvents()$  (Section 5).
4 Macros: we use as macros conditions 1 to 3, Equation 4 (Section 5), 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 5);
9  $restartLocal() := \{local \leftarrow (y, y)\}$ , where  $y = \langle getLabel(), zrs, zrs \rangle$ ;
10  $equalStatic(X, Y) := X.curr.\ell = Y.curr.\ell \wedge X.curr.o = Y.curr.o \wedge X.prev = Y.prev$ ;
11 procedure  $cancelPairLabels(Z)$  begin
12   foreach  $\ell \in \{Z.curr.\ell, Z.prev.\ell\}$  do  $\ell.cancel(\ell)$ ;  $labelBookkeeping()$ ;
13 function  $revive(Z)$  begin
14    $cancelPairLabels(Z)$ ; return  $\langle (getLabel(), Z.curr.m, Z.curr.m), Z.curr \rangle$ ;
15 procedure  $increment()$  begin
16   let  $local = \langle (local.curr.\ell, (local.curr.m +$ 
17      $idV(i))(\bmod MAXINT), local.curr.o), local.prev \rangle$ ;
18   if  $exhausted(local)$  then  $local \leftarrow revive(local)$ ;
19 function  $merge(loc, arr)$  begin
20   if  $\exists x \in \{curr, prev\} loc.curr =_{\ell, o} arr.x$  then let  $pivot := loc.curr.o$  else let
21      $pivot := loc.prev.o$ ;
22   let  $initToLoc := arr.curr <_{\ell, o} loc.curr \vee (arr.curr =_{\ell, o} loc.curr \wedge arr.prev \leq_{\ell, o}$ 
23      $loc.prev)$ ;
24   if  $initToLoc$  then let  $output := loc$  else let  $output := arr$ ;
25   foreach  $k \in \{1, \dots, N\}$  do
26     let  $maxNewEvents = \max\{newEvents(Z, pivot)[k] \mid Z \in \{loc, arr\}\}$ ;
27      $output.curr.m[k] \leftarrow (pivot[k] + maxNewEvents)(\bmod MAXINT)$ ;
28   return  $output$ ;
29 do forever begin
30    $labelBookkeeping()$ ;
31   if  $\neg(mirroredLocalLabels() \wedge labelsOrdered(local))$  then  $restartLocal()$ ;
32   if  $exhausted(local)$  then  $local \leftarrow revive(local)$ ;
33   foreach  $p_k \in P \setminus \{p_i\}$  do send  $encapsulate(\langle local, pairs[j] \rangle)$  to  $p_k$ ;
34 upon message  $m = \langle \bullet, (arriving, rcvdLocal) \rangle$  arrival from  $p_j$  begin
35    $labelBookkeeping(m, j)$ ;
36    $pairs[j] \leftarrow arriving$ ;
37   if  $equalStatic(local, rcvdLocal) \wedge legitMsg(m, arriving.curr.\ell) \wedge$ 
38      $pairInvar(arriving)$  then
39     if  $\neg legitPairs(local, arriving)$  then  $restartLocal()$ ;
40     else
41        $local \leftarrow merge(local, arriving)$ ;
42       if  $exhausted(local)$  then  $local \leftarrow revive(local)$ ;
```

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 5).

Restarting local via $restartLocal()$ (line 9). The macro $restartLocal()$ lets $local_i$ have its starting value $\langle y, y \rangle$, where $y = \langle getLabel(), zrs, zrs \rangle$ and zrs is the N -size vector of zeros. Processor p_i can use $restartLocal()$ for setting $local_i$ to its initial value, whenever the invariants for $local_i$ do not hold in the do-forever loop and in the message arrival procedures of Algorithm 1.

Token passing mechanism for sending and receiving local. Algorithm 1 uses a token circulation mechanism for sending and receiving $local$, which is independent of the algorithm's computations on $local$. This mechanism is necessary for ensuring that (after a constant number of steps) for every two processors $p_i, p_j \in P$, p_j processes a message from p_i only if p_i has received the latest value of $local_j$.

We remark that without this mechanism, it is possible that p_i does not receive (and process) p_j 's latest value of $local_j$ for an unbounded number of steps, and yet p_i keeps sending $local_i$ to p_j for an unbounded number of steps. The latter case can cause an unbounded number of steps that include a call to $restartLocal()$ at p_j , if the pair that p_j received from p_i cannot be merged with $local_j$ (cf. Section 5.1 and message arrival procedure in this section). In Section 7, we show that a call to $restartLocal()$ in a step of the algorithm (possibly) implies that Requirement 1 does not hold for the state that immediately follows this step. Hence, an unbounded number of calls to $restartLocal()$, imply an unbounded number of states in which Requirement 1 does not hold. The token circulation mechanism helps the proposed algorithm to avoid this problem.

To implement 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 $pairs_i[i]$ stores p_i 's pair, i.e., $local_i$ is an alias for $pairs_i[i]$. We implement the token passing mechanism by augmenting the messages that a processor sends (via $encapsulate()$) in Algorithm 1 as follows. A processor p_i sends $\langle local_i, pairs_i[j] \rangle$ to a processor p_j by calling $encapsulate(\langle local_i, pairs_i[j] \rangle)$ in line 30. Hence, a message sent by p_j and received by p_i has the form $m_j = \langle \bullet, \langle arriving_j, rcvdLocal_j \rangle \rangle$ (line 31). Processor p_i stores $arriving_j$ in $pairs_i[j]$ (line 33), in order to ensure that p_i has received 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 on their $curr.m$, since the merging conditions (cf. Section 5.1) don't depend on $curr.m$. We detail the exact procedures of sending and receiving messages in Algorithm 1 in the last part of this section. In Section 7 we show that the token passing mechanism is self-stabilizing (in at most CN^2 steps).

The function $revive()$ (lines 13–14). When the pair Z is exhausted (Condition 1), a call to $revive(Z)$ lets Z to wrap around and return its new version

(Section 5). That is, p_i cancels Z 's labels, $Z.curr.l$ and $Z.prev.l$, by calling the labeling algorithm (function *cancelPairLabels*, lines 11–12), and then sets $Z.curr$ to be the output pair's *prev* and $\langle getLabel(), Z.curr.m, Z.curr.m \rangle$ as the output's *curr*.

The vector clock increment function, *increment()* (lines 15–17).

When p_i calls *increment()*, it increments the i^{th} entry of p_i 's vector clock. That is, 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 16). In case that increment leads to a vector clock exhaustion, it calls the function *revive()* (line 17). We assume that a processor can only call *increment()* in the beginning of a step that ends with a send operation (see paragraph on Algorithm 1's do-forever loop below), and this call is part of the step. This restriction ensures that vector clock increments are immediately sent to all other processors.

Aggregation of vector clock pairs with the *merge()* function (lines 18–25). The function *merge*(Z, Z') (lines 18–25) aggregates two pairs, Z and Z' , such as the local one and another one arriving via the network. It outputs a pair *output* with the $<_{\ell,o}$ -maximum items that includes the aggregated number of events of Z and Z' (Section 5).

The function uses the $<_{\ell,o}$ -maximum pivot item x in Z and Z' , from which it counts the new events in Z and Z' (line 19). It initializes the output pair, *output*, with the input pair that is $<_{\ell,o}$ -maximum both in *curr* and *prev* (lines 20–21). 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 22–24), and returns *output* (line 25). That is, $output.curr.m[i] = \max\{newEvents(X, pivot)[i] \mid X \in \{Z, Z'\}\} + pivot[i] \pmod{MAXINT}$, for every $i \in \{1, \dots, N\}$.

The procedures of the do-forever loop and the message arrival event.

We explain Algorithm 1's do-forever loop (lines 26–30) and message arrival procedure (lines 31–38), which follow the algorithm composition of Figure 2 (Section 4).

The do-forever loop procedure (lines 26–30). The do-forever loop starts by letting the labeling algorithm take a step in line 27 (part 1 of Figure 2). Line 28 refers to the invariants of *local*. Algorithm 1 calls *restartLocal()* in line 28, in case one of the following does not hold: (i) $local.curr.l$ is not the local maximal label or $local.prev.l$ 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., *labelsOrdered(local)* is false. In line 29, 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 28–29 refer to part 2 of Figure 2.

In line 30 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 30 refers to part 3 of Figure 2.

The message arrival procedure (lines 31–38). Upon arrival of a message $m = \langle \bullet, \langle arriving, rcvdLocal \rangle \rangle$ from processor p_j (part 4 of Figure 2) the labeling algorithm processes its own part of m (part 5 of Figure 2) by the call to $labelBookkeeping(m, j)$ in line 32. In lines 33–38 of the message arrival procedure, Algorithm 1 processes $\langle arriving, rcvdLocal \rangle$ (parts 6 and 7 of Figure 2). In line 33 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 1 proceeds in processing *arriving* only if $equalStatic(local, rcvdLocal) \wedge legitMsg(m, arriving.curr.l) \wedge pairInvar(arriving)$ holds (line 34). 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 $arriving.curr.l$ (cf. $mirroredLocalLabels()$ predicate in line 28). The predicate $legitMsg(m, arriving.curr.l)$ (cf. Section 4) is true if $arriving.curr.l$ 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 $arriving.prev.l \preceq_{ib} arriving.curr.l$ holds. Hence, if $legitMsg(m, arriving.curr.l) \wedge pairInvar(arriving)$ is false, m contains stale information and existed in the system in the starting system state.

In case the condition of line 34 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 $comparableLabels(\{X, Y\}) \wedge existsPivot(X, Y)$ holds. That is, all the labels of the pairs X and Y must be comparable with respect to the order of the labeling scheme and there exist a pivot item between X and Y (Condition 3, Section 5). In case $legitPairs(local, arriving)$ is false, the algorithm calls $restartLocal(local)$ (line 35), since merging must be possible in a legal execution. Otherwise, merging *local* and *arriving* is feasible, and thus the algorithm lets *local* to have the return value of $merge(local, arriving)$ (line 37). In case the new pair value of *local* is exhausted, *local* wraps around to the return value of $revive(local)$ in line 38 (cf. line 9).

Remarks on algorithm composition. Note that in case of pair exhaustion Algorithm 1 forces the repetition of parts 1 and 2 of Figure 2 corresponding to the do-forever loop procedure, as well as, parts 5 and 7 of Figure 2 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 11–14). Then, Algorithm 1 stores the return value of *revive(local)* in *local* (line 29 or 38). Thus, if *local* is not exhausted during a step (line 29 or 38), the composition of the labeling and the vector clock algorithm is along the lines of [10, Section 2.7]. The latter holds, since Algorithm 1 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 14 and Section 5). Moreover, in case the invariants for *local* do not hold in line 28 or 35, 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 4).

7 Correctness Proof

7.1 The proof in a nutshell

We show that Algorithm 1 is practically-self-stabilizing (Definition 2.3). Recall from Section 2 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 7.1 we show that for any \mathcal{L}_S -scale execution R , $f_R \ll |R|$ holds (cf. Section 2).

Theorem 7.1 (Algorithm 1 is practically-self-stabilizing). For every infinite execution R of Algorithm 1, and for every \mathcal{L}_S -scale subexecution $R' \sqsubseteq R$, $f_{R'} = f(R', N) \ll |R'|$ holds.

To the end of proving Theorem 7.1, 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 (Section 7.3). Given these invariants we present the conditions for an execution to be legal (Section 7.3). More specifically, we show that an execution is legal 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 Section 7.4 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 2). Then, in Section 7.5, we prove that for every \mathcal{L}_S -scale execution R' , the number of steps that include a call to either *restartLocal()* or *revive()* is significantly less than $|R'|$, and combine the above to prove Theorem 7.1 (Corollary 7.24).

Our proof also requires to show that the labeling algorithm by Dolev et al. [12] remains practically-self-stabilizing (Section 7.2), 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 $p_i \in P$ (cf. Section 3.2).

7.1.1 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 1. 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.

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

We generalize the lemmas of Dolev et al. [12] (Section 3.2) that bound the number of label creations and adoptions of their labeling algorithm (cf. Section 3.2) to accommodate for the extra number of labels of Algorithm 1, 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 Section 6 that if there are no calls to *revive()* (lines 17, 29, and 38) during an execution, then Algorithm 1 does not change the state of the labeling algorithm (cf. function definition in lines 13–14). However, in the starting system state of an execution of Algorithm 1 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 [12, 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 7.1). In Corollary 7.2, we extend [12, Lemma 4.4], which bounds the number of labels that p_i creates (Corollary 7.2). We then present Corollary 7.3 that is an implication of corollaries 7.1 and 7.2 and states that the labeling algorithm of Dolev et al. [12] remains practically-self-stabilizing given the generalized bounds presented in those corollaries. Corollary 7.3 is an extension of [12, Theorem 4.2], which shows that the labeling algorithm [12, Algorithm 2] is practically-self-stabilizing. Hence, we will use Corollary 7.3 for proving that Algorithm 1 is also practically-self-stabilizing. In Section 7.5, we will extend these bounds to accommodate for the extra labels created by Algorithm 1, when a processor calls the function *revive()*.

Corollary 7.1 (extension of [12, 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 [12, Lemma 4.3] is $N + M$, but in the setting of Algorithm 1 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. [12] (cf. Section 3.2).

Corollary 7.2 (extension of [12, Lemma 4.4]). *Let $p_i \in P$ be a processor and $L_i = \ell_{i_0}, \ell_{i_1}, \dots$ 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 $|L_i| \leq 4N^2 + 4NM - 4N - 2M$ [12].*

The bound in Corollary 7.2 follows by the proof of [12, 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. These labels are at most $2(M + \sum_{j \neq i} |storedLabels_i[j]|) = 2M + 2(N - 1)(2N + 2M) = 4N^2 + 4NM - 4N - 2M$ (the second equality holds by Corollary 7.1).

Corollary 7.3 is a straightforward extension of [12, Theorem 4.2] that also holds for the updated bounds of corollaries 7.1 and 7.2, since the proof is based on the bounds' existence, rather than the actual bounds.

Corollary 7.3 (extension of [12, Theorem 4.2]). *Let R be an \mathcal{L}_S -scale execution of the labeling algorithm [12, Algorithm 2], 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 7.1 and 7.2 as well as the updated queue lengths in $storedLabels_i$, where $p_i \in P$.*

We remark that in Section 7.5 we extend the queue lengths to accommodate for the extra labels that are created due to Algorithm 1, i.e., when wrap-around events occur and a processor calls *revive()*.

7.3 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 legal. We define the predicate *localInvariants(i)* (Definition 7.4), which gives the local invariants for *local_i* of a processor p_i . That is, if *localInvariants(i)* is false in line 28, then processor p_i calls *restartLocal_i()*. We show that for all functions of Algorithm 1 that include *local_i* of a processor p_i in their input, *localInvariants(i)* holds (lemmas 7.5–7.8).

We also give the conditions for an execution to be legal. To that end, we show that Requirement 1 is possibly violated in a step where a processor calls *restartLocal()* (Remark 7.9) and definitely violated when a processor calls *revive()* in two or more steps in an execution (Remark 7.10). 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 35. Finally, in Lemma 7.12, we prove that given the bounds $B_{restart}(R)$ and $B_{revive}(R)$ on the number of steps

that include a call to $restartLocal()$ or $revive()$ in an \mathcal{L}_S -scale execution R , there exists at least one legal subexecution R^* of R , such that $|R^*| \ll |R|$ holds under the condition that $B_{restart}(R) \ll |R| \wedge B_{revive}(R) \ll |R|$ holds. In sections 7.4 and 7.5, we prove that the bounds $B_{restart}(R)$ and $B_{revive}(R)$ indeed exist for every \mathcal{L}_S -scale execution R (and show that Algorithm 1 is practically-self-stabilizing).

Definition 7.4 (The $localInvariants()$ predicate). *Let R be an execution of Algorithm 1, $c \in R$ 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, $localInvariants(i) := mirroredLocalLabels_i() \wedge labelsOrdered_i(local_i)$ (line 28) holds.*

Lemma 7.5. *Let a_x be a step in R in which processor p_i calls $revive_i(local_i)$ when executing line 17, 29, or 38 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 $\langle getLabel_i(), local_i.curr.m, local_i.curr.m \rangle$ (line 14). 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 $localInvariants(i) = mirroredLocalLabels_i() \wedge labelsOrdered_i(local_i)$ 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 13–14), 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 7.6. *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 $mirroredLocalLabels_i() \wedge labelsOrdered_i(local_i) \wedge equalStatic_i(local_i, rcvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge pairInvar_i(arriving_j) \wedge legitPairs_i(local_i, arriving_j)$ hold in c . Then, $localInvariants(i)$ holds in c' , i.e., after the execution of line 37 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 p_i receives from p_j in step a_i and assume that $mirroredLocalLabels_i() \wedge labelsOrdered_i(local_i) \wedge equalStatic_i(local_i, rcvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge$

$pairInvar_i(arriving_j) \wedge legitPairs_i(local_i, arriving_j)$ hold in c with respect to $local_i$ and m_j . For brevity, we denote $\xi_i := isStored_i(local_i.prev.l) \wedge local_i.curr.l = getLabel_i() \wedge labelsOrdered_i(local_i)$. Observe that $merge_i(local_i, arriving_j)$, initializes $output_i$ either to $local_i$ or to $arriving_j$ (lines 19–21). Then, $output_i.curr.m[k]$ is updated with $newEvents$, for each $k \in \{1, \dots, N\}$, and the result is returned and saved to $local_i$, hence lines 22 to 24 do not change the value of ξ_i . Therefore, we show that for each of the three different cases in which a pivot exists (Figure 3), ξ_i holds after $local_i$ is updated with $merge_i(local_i, arriving_j)$.

Case of Figure 3a. 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 32), hence the return value of $getLabel_i()$ remains the same (in c and c') after the execution of line 32 in step a_i . Therefore, $isStored_i(output_i.prev.l) \wedge output_i.curr.l = getLabel_i() \wedge labelsOrdered_i(output_i)$ 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 $\xi_i = isStored_i(local_i.prev.l) \wedge local_i.curr.l = getLabel_i() \wedge labelsOrdered_i(local_i)$ also holds.

Case of Figure 3b. Let $\ell_{loc} := local_i.curr.l$, $\ell_{arr} := arriving_j.curr.l$, $\ell_{prv} := local_i.prev.l = arriving_j.prev.l$, and $\ell_{max} := \max_{\prec_{ib}} \{\ell_{loc}, \ell_{arr}\}$ 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 19–21), i.e., the pair that stores ℓ_{max} in its $curr.l$. Thus, $isStored_i(output_i.prev.l)$ holds in c' , since $isStored_i(\ell_{prv})$ and $output_i.prev.l = \ell_{prv}$ hold in c . Due to line 32, 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 32, i.e., $output_i.curr.l = \ell_{max} = getLabel_i()$. Moreover, $mirroredLocalLabels_i()$ holds for $local_i$, after $local_i$ is updated with $merge(local_i, arriving_j)$ in line 37 during step a_i (and hence holds in c').

By the definition of the case of Figure 3b, $local_i.curr.o \neq arriving_j.curr.o$ holds in a_i . If $\ell_{arr} \prec_{ib} \ell_{loc}$, then $labelsOrdered_i(local_i)$ holds in c' , since $output_i.curr.l = \ell_{loc}$, $output_i.prev.l = \ell_{prv}$, and $labelsOrdered_i(local_i)$ holds in c . Otherwise, if $\ell_{max} = \ell_{arr}$, then $output_i.prev.l = \ell_{prv} \prec_{ib} \ell_{max} = output_i.curr.l$ 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, $labelsOrdered_i(local_i)$ holds in c' .

Case of Figure 3c. 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 $local_i.curr.l = \ell_{max}$ in c , then $output_i$ is initialized to $local_i$. Thus, $isStored_i(output_i.prev.l) \wedge output_i.curr.l = getLabel_i() \wedge labelsOrdered_i(output)$ holds in the end of step a_i , since (i) $local_i.itm =_{\ell,o} output_i.itm$, for each $itm \in \{curr, prev\}$, (ii) $isStored_i(local_i.prev.l) \wedge local_i.curr.l = getLabel_i() \wedge labelsOrdered_i(local_i)$ holds before $merge_i()$ is called, and (iii) line 32 does not change the return value of $getLabel_i()$. Note that this is the only case where $arriving_j.prev.l$ 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 $arriving_j.prev.l$ 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 $arriving_j.curr.l = \ell_{max}$. In this case, $isStored_i(output_i.prev.l) \wedge output_i.curr.l = getLabel_i()$ holds, since (i) $output_i.prev.l = \ell_{loc}$ and $isStored_i(\ell_{loc})$ holds, and (ii) $output_i.curr.l = \ell_{arr} = \ell_{max} \wedge \ell_{loc} \prec_{lb} \ell_{arr}$, hence $getLabel_i()$ returns ℓ_{arr} after the execution of line 32. Also, $pairInvar_i(arriving_j)$ holds, which implies that $\ell_{loc} = arriving_j.prev.l \preceq_{lb} arriving_j.curr.l = \ell_{arr} = \ell_{max}$. Thus, it either holds that $arriving_j.prev.l = arriving_j.curr.l = \ell_{arr}$ and $\neg isCanceled_i(\ell_{arr})$ (since $getLabel_i() = \ell_{arr}$), or that $arriving_j.prev.l = arriving_j.curr.l \wedge isCanceled_i(arriving_j.prev.l)$. Therefore, $labelsOrdered_i(arriving_j)$ holds, hence $labelsOrdered_i(output_i)$ holds in the end of step a_i , thus $labelsOrdered_i(local_i)$ holds in c' . \square

Note that during $increment_i()$, p_i changes only in $local_i.curr.m[i]$ (line 16) and the other fields of $local_i$ stay intact. In case $local_i$ is exhausted after that increment, p_i calls $revive_i()$ (line 17), hence by Lemma 7.5 the value of $localInvariants(i)$ does not change whenever p_i calls $increment_i()$. By lemmas 7.5 and 7.6 we have the following.

Corollary 7.7. *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 7.8 considers the case in which a processor calls $restartLocal_i()$ (line 28 or 35).

Lemma 7.8. *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 28 or 35, 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 27 and 32) before each call of $restartLocal_i()$ (lines 28 and 35) and no other function in the lines between the call to $labelBookkeeping_i()$ and $restartLocal_i()$ changes the variables of the labeling algorithm (lines 33–34), $getLabel_i()$ 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, $labelsOrdered_i(local_i)$ holds (condition 2), since $(local_i.prev.\ell = local_i.curr.\ell \wedge \neg isCanceled_i(local_i.curr.\ell))$ holds. \square

Conditions for an execution to be legal So far in this section, we have shown that $localInvariants(i)$ holds for the output of every function of Algorithm 1 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 7.9 and 7.10). Then, in Lemma 7.12 we present the conditions under which an execution is legal (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 7.9 ($restartLocal()$ breaks Requirement 1). We remark that it is possible that Requirement 1 does not hold immediately after the execution of $restartLocal()$ (lines 28 and 35). 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 miscounting 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 $restartLocal_i()$. \square

Consider a message $m_{j,i} = \langle \bullet, \langle arriving_j, rvdLocal_j \rangle \rangle$ that a processor p_i receives from a processor p_j , such that $\chi_{i,j} := equalStatic_i(local_i, rvdLocal_j) \wedge legitMsg_i(m_j, arriving_j.curr.\ell) \wedge pairInvar_i(arriving_j)$ does not hold (line 34). Since such messages are not processed by Algorithm 1, 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 32. 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 Section 7.5 (Lemma 7.17) we show that for every \mathcal{L}_S -scale execution R , there exist bounds in the number of steps that include a call to $restartLocal()$ or to $revive()$ that are significantly less than $|R|$.

Remark 7.10 (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.

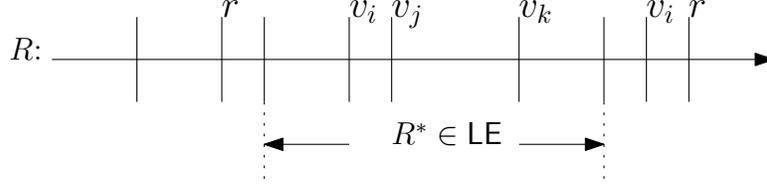


Figure 4: Illustration of a legal execution (cf. Lemma 7.12). The horizontal line denotes an execution R and the vertical lines highlight specific steps of R . The vertical lines that are marked with r , denote a step in which a processor called $\text{restartLocal}()$. The vertical lines that are marked with v_x denote a step in which a processor p_x called $\text{revive}_x()$. In this example for the segment of R , marked as R^* , the following hold: (i) no processor called $\text{restartLocal}()$, and (ii) p_i (and every other active processor) called $\text{revive}()$ at most once. Thus, by Lemma 7.12, R^* is a legal execution, i.e., $R^* \in \text{LE}$.

Let c_u be the first state after (the step in which) p_i calls $\text{revive}_i()$ for the first time after c_x , and c_v be the first state after p_i calls $\text{revive}_i()$ for the first time after c_u . By the vector clock pair construction and the definition of the function $\text{revive}()$ (Section 5), $\text{local}_i^u.\text{prev}$ is the value of $\text{local}_i.\text{curr}$ immediately before the first call to $\text{revive}_i()$ (after c_x). Hence, the pivot item between local_i^x and local_i^u is $\text{local}_i^u.\text{prev}$, i.e., $(\text{local}_i^x.\text{curr}.\ell, \text{local}_i^x.\text{curr}.\text{o}) = (\text{local}_i^u.\text{prev}.\ell, \text{local}_i^u.\text{prev}.\text{o})$. Similarly, $\text{local}_i^v.\text{prev}$ is the value of $\text{local}_i.\text{curr}$ immediately before the first call to $\text{revive}_i()$ after c_u . Hence, the pivot item between local_i^u and local_i^v is $\text{local}_i^v.\text{prev}$, i.e., $(\text{local}_i^u.\text{curr}.\ell, \text{local}_i^u.\text{curr}.\text{o}) = (\text{local}_i^v.\text{prev}.\ell, \text{local}_i^v.\text{prev}.\text{o})$. Thus, the vector clock items $\text{local}_i^u.\text{prev}$ and $\text{local}_i^x.\text{curr}$, and specifically the events that $\text{local}_i^x.\text{curr}.\text{m}$ recorded in state c_x do not appear in state c_v . Therefore, irrespective of the number of calls to $\text{revive}_i()$ between c_v and c_y , it is not possible to count the events in p_i (i.e., the calls to $\text{increment}_i()$) between the states c_x and c_y by comparing local_i^x and local_i^y . \square

In Definition 7.11 we describe the conditions under which $\text{restartLocal}()$ is never called in an execution. Then, in Lemma 7.12 we give the conditions for an execution to be legal. We also prove that for every \mathcal{L}_S -scale execution R , there exists a legal subexecution R^* of R , such that $|R^*| \ll |R|$, under the conditions that there exist bounds $B_{\text{restart}}(R)$ and $B_{\text{revive}}(R)$ on the number of steps that include a call to $\text{restartLocal}()$, and respectively, $\text{revive}()$ in R , and $B_{\text{restart}}(R) \ll |R| \wedge B_{\text{revive}}(R) \ll |R|$ holds. We illustrate Lemma 7.12 in Figure 4.

Definition 7.11 (The *globalInvariants()* predicate). Let R be an execution of Algorithm 1, $c \in R$ a system state, $\varphi_i \equiv \text{mirroredLocalLabels}() \wedge \text{labelsOrdered}(\text{local}_i)$ (line 28), and $\psi_{i,j} \equiv \text{legitPairs}(\text{local}_i, \text{arriving}_j)$ (line 35), where $p_i, p_j \in P$ and $m_j = \langle \bullet, \langle \text{arriving}_j, \text{rcvdLocal}_j \rangle \rangle$ is a message in the communication channel from p_j to p_i . We define $\text{globalInvariants}(R, c) :=$

$\forall_{p_i \in P(R), p_j \in P} \varphi_i \wedge (\neg \chi_{i,j} \vee \psi_{i,j})$. We say that the global invariants hold during R , if $globalInvariants(R, c)$ holds for every $c \in R$.

Lemma 7.12. *Let R be an \mathcal{L}_S -scale 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 ,*

$R^* \in \text{LE}$ holds, i.e., R^* is a legal execution.

(II) *Moreover, let $B_{restart}(R)$ and $B_{revive}(R)$ be bounds on the number of steps in R that include a call to $restartLocal()$ and $revive()$, respectively, such that $B_{restart}(R) \ll |R| \wedge B_{revive}(R) \ll |R|$ holds. Then, there exists at least one subexecution R^* of R such that $R^* \in \text{LE} \wedge |R^*| \ll |R|$ holds.*

Proof. We prove Part I of the lemma using remarks 7.9 and 7.10. For Part II we use the definitions of \mathcal{L}_S -scale executions and the \ll relation (Section 2), as well as, the pigeonhole principle.

Proof of Part I. 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 7.9). 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 Equation 5 (Section 5), 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 Section 5.

Proof of Part II. Since R is of \mathcal{L}_S -scale, and $B_{restart}(R) \ll |R| \wedge B_{revive}(R) \ll |R|$ holds, (by the pigeonhole principle) there exists at least one segment R^* of R in which conditions (i) and (ii) of the lemma hold, such that $|R^*| \ll |R|$. Thus, by Part I of the lemma, $R^* \in \text{LE}$. In fact, the maximal such R^* is of size at least $|R| / (B_{restart}(R) + B_{revive}(R))$, hence $|R^*| \ll \mathcal{L}_S$ indeed holds (cf. Section 2). \square

By Lemma 7.12, we need to show that for every \mathcal{L}_S -scale execution R the bounds $B_{restart}(R)$ and $B_{revive}(R)$ of the lemma statement indeed exist and also that $B_{restart}(R) \ll |R| \wedge B_{revive}(R) \ll |R|$ holds. We do that in Sections 7.4 and 7.5. In the end of Section 7.5 we complete the proof by showing that Algorithm 1 is practically-self-stabilizing.

7.4 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 (Section 2).

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 [20]. 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 7.15), 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 1 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 (Section 2), p_i can either

- (1) run lines 26–29 and send one message (out of $N - 1$) to another processor due to line 30, or
- (2) send one message (out of at most $N - 2$ remaining messages) to another processor due to line 30, or
- (3) run the message arrival procedure in lines 31–38.

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 (Section 2), each step includes a single send or receive operation. Hence, a complete iteration of Algorithm 1’s do-forever loop (lines 26–30) requires $N - 1$ steps (not necessarily consecutive),

due to the $N - 1$ messages to be send to the processor's neighbors. In detail, we assume that when a processor $p_i \in P$, runs line 30, 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 $encapsulate_i(local_i)$ and remains intact during those calls, regardless of the changes that occur to $local_i$ after the first (out of $N - 1$) send operation. We assume p_i automatically discards the output pair of $clone_i(local_i)$ in the last of the $N - 1$ steps of that send operation. These $N - 1$ steps can be interleaved with steps of other processors or with steps in which p_i runs the message arrival procedure (lines 32–38) and possibly changes $local_i$, but not the copy of $local_i$ that is used to complete the send operation.

Thus, during a step a processor can call functions from $\mathcal{F}_1 := \{increment(), revive(), labelBookkeeping(), restartLocal(), clone(), encapsulate()\}$ 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, basing 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 27), 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 32), 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 $labelBookkeeping_i()$ 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 14, which includes a call to $labelBookkeeping_i()$ in line 12), or

- (ii) discovering stale information in the label storage in line 27, or
- (iii) receiving a new label during the message arrival procedure in line 32.

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 (cf. Section 3.2).

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 13–14), 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 14, 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 32 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 $cancelPairLabels_i()$, followed by the 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 17. In case $exhausted_i(local_i)$ is false, then $increment_i()$ is only changing $local_i$ to a new value in line 16, 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 17), 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 16. 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 17.

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 $T_R : E(R) \rightarrow \mathcal{F}$ 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

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 $M = \mathcal{CN}(N - 1)$ is the maximum capacity of pairs in the communication channels. Note that this definition of $V_i(R)$ makes the definition of pair evolution graphs more intuitive.

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 Figure 5, 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 $e = (Z, \lambda, Z') \in E_i(R)$, $Z' \in V_{i+1}(R) \wedge Z = Z' \wedge Z \neq_{V(R)} Z'$ 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()*. As we showed in Section 7.3, *restartLocal()* and *revive()* are the only two functions of Algorithm 1 that

can possibly violate the conditions for an execution to be legal. 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 in a subset of \mathcal{F} , $\mathcal{F}_{focused} := \{restartLocal(), revive()\}$, since as we will show in Section 7.5, 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 7.13.

Definition 7.13 (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 $e_1, e_2, \dots, e_x \in E(R)$, such that $T_R(e_1) = f \wedge T_R(e_x) = g \wedge (\wedge_{i=1, \dots, x-1} (e_i \text{ is adjacent to } e_{i+1}))$.*

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 7.14 shows which functions leave the static part of their input pairs intact and which don't.

Lemma 7.14. *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 5). Note that a call to $increment()$ can include a call to $revive()$ (we study the case of $revive()$ below), however line 16 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 13–14). 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 $Z_2.curr.l$ (lines 13–14), 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 $restartLocal_i()$, $local_i$ 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 $local_i.curr.l$ 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 $restartLocal_i()$: $local_i.curr.l \neq local_i.prev.l$ 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 $restartLocal_i()$. \square

In Definition 7.15 we define function causality between a call to a function in $\{revive(), restartLocal()\}$ and a subsequent call to $restartLocal()$. Let $p_{restartLocal}(i, k) := \neg(mirroredLocalLabels() \wedge labelsOrdered(local_i))$ (line 28) and $q_{restartLocal}(i, j, k) := \neg legitPairs(local_i, arriving_j)$ (line 35) 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 7.15 (f causes $restartLocal()$, for $f \in \{revive(), restartLocal()\}$). Let $|R'| \leq MAXINT$ be an execution, $p_i, p_j \in P$, $a_k \in R'$, and $\mathcal{P}(i, j, k) := p_{restartLocal}(i, k) \vee q_{restartLocal}(i, j, k)$. Moreover, 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 $P = (e_1, \dots, e_x)$, 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 Figure 5 $revive()$ 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 $exhausted(local)$ (Equation 1 and lines 17, 29, and 38), and thus cause a call to $revive()$ (Lemma 7.16). We note that the analysis for $revive()$ (Lemma 7.16) is much simpler than the one for $restartLocal()$, because the condition for calling $revive()$, $exhausted(local)$, depends only on vector clock increments. On the contrary,

the conditions for calling $restartLocal()$ (lines 28 and 35) depend on every field of $local$, as well as on whether an arriving pair can be merged with the local one.

Lemma 7.16. *Let $p_i \in P$. (i) The value of the predicate $exhausted_i(local_i)$ (cf. Section 5) 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 $exhausted(Z) \Leftrightarrow \sum_{k=1}^N (Z.curr.m[k] - Z.curr.o[k]) \geq MAXINT - 1$ (Equation 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 15–17) 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 18–25 and Section 5). 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 13–14), $local_i.curr.m = local_i.curr.o$ holds for their outputs, hence $exhausted_i(local_i)$ is false. \square

7.5 Bounding the number of deviations from the abstract task in an \mathcal{L}_S -scale execution

In Lemma 7.17, 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 significantly less than $|R'|$. We focus in these two functions, because due to Section 7.4, only these two functions can cause a call to $restartLocal()$ (cf. Lemma 7.14 and Definition 7.15). Then, in Corollary 7.24 we show that Algorithm 1 is practically-self-stabilizing (i.e., Theorem 7.1 holds).

Lemma 7.17. *Let R be an execution of Algorithm 1 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 significantly less than $MAXINT$.*

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 7.18 shows that the number of steps in R' in which a processor calls $revive()$ is significantly less than $MAXINT$.

A processor can call *restartLocal()* either in line 28 or in line 35. 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 28. To prove this statement, first observe that the condition in line 28 can be false due to stale information that resided in the processor's state in the starting state. However, by Corollary 7.7, for any function that changes *local*, it holds that the condition in line 28 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 35 is significantly less than $MAXINT$. We first bound the maximum number of steps that include a call to *revive()* (Claim 7.18), as well as the maximum number of labels that can exist during R' (Claim 7.19). In claims 7.20 and 7.21 we bound the number of steps that include a call to *restartLocal()* in line 35 due to the recovery of the link-layer algorithm [13], and respectively, the token-passing mechanism (Section 6). Moreover, in Claim 7.22 we bound the number of calls to *restartLocal()* in line 35 that occur due to a single pair static part that appears in R' . Finally, in Claim 7.23 we show that these bounds imply that the number of steps that include a call to *restartLocal()* in line 35 during R' is significantly less than $|R'|$, by showing that the number of pair static parts that appear in R' is significantly less than $MAXINT$ and combining Claims 7.18–7.22.

Claim 7.18. *The number of steps during R' that include a call to *revive()* is at most $N + N^2 + N^3 \cdot C$.*

Proof of Claim 7.18. The proof considers the three causes for pair exhaustion during R (cf. Lemma 7.16). 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 15–17). 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 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 . 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.l* and *prev.l*) 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 7.16). At most N such vector clocks have resided in the states of the processors, and at most $M \leq N^2 \cdot C$ resided in the communication channels. Since for each of these $N + N^2 \cdot C$ pair values can lead to at most N concurrent exhaustions, there can be $N^2 + N^3 \cdot C$ 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 $N + N^2 + N^3 \cdot \mathcal{C}$ pair exhaustions that can occur during R' . Hence, at most that many calls to *revive()* in R' . \square

Claim 7.19. *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 7.19. Recall that from the proofs of corollaries 7.1, 7.2 and 7.3, proving that the labeling algorithm of Dolev et al. [12] 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 7.4) has the same bound.

By corollaries 7.1 and 7.2 there can be at most $4N^2 + 4NM - 4N - 2M$ labels in the system, where $M = \mathcal{C}N(N - 1)$ is the maximum capacity of pairs in the communication channels. Note that there can be at most $N + N^2 + N^3 \cdot \mathcal{C}$ additional labels creations, due to calls to the *revive()* function. Thus, there can be at most $L := (N + N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ labels in the system during R' , and hence at most that many calls to *newLabel()*. \square

In the labeling algorithm of Dolev et al. [12], 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 $\ell \in storedLabels_i[j]$ (cf. Section 3.2). Recall that in Section 7.2 we extended the queue lengths for an execution of Algorithm 1 in which no processor calls *revive()*, to accommodate for the two labels that each pair includes. By Claim 7.19, 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 1. Thus, by Claim 7.19 and Section 7.2 we set $|storedLabels_i[j]| = L$, for every $p_i, p_j \in P$, where $L = (N + N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$.

Claim 7.20. *There can be at most $(2\mathcal{C} + 1)N^2$ steps that include a call to *restartLocal()* in line 35 due to the recovery of the link-layer algorithm [13].*

Proof of Claim 7.20. Recall that the self-stabilizing link-layer algorithm of [13], which we rely on, requires at most $2\mathcal{C} + 1$ message arrivals per direction of a communication channel to stabilize. Therefore, since there are $N(N - 1)/2$ links in the system, where each of them is a bidirectional communication channel, there can be at most $2 \cdot (2\mathcal{C} + 1) \cdot N(N - 1)/2 \leq (2\mathcal{C} + 1)N^2$ steps that include a call to *restartLocal()* due to stale information that, at the starting system state, resides in the communication channels. \square

Claim 7.21. Let $m_{j,i} = \langle \bullet, \langle \text{arriving}_j, \text{rcvdLocal}_j \rangle \rangle$ be a message that p_i receives from p_j , via their communication channel, $\text{channel}_{j,i}$. There can be at most \mathcal{C} steps that include a call to $\text{restartLocal}_i()$ in line 34, due to stale information that appears in the field rcvdLocal_j of $m_{j,i}$, where $m_{j,i}$ appears in $\text{channel}_{j,i}$ in the starting system state and rcvdLocal_j sets $\text{equalStatic}_i(\text{local}_i, \text{rcvdLocal}_j)$ to true. Hence, there can be at most $M \leq \mathcal{C}N^2$ such calls to $\text{restartLocal}()$ in any execution.

Proof of Claim 7.21. Notice that there can be at most \mathcal{C} messages in the communication channel from p_j to p_i , $\text{channel}_{j,i}$, 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 $\text{equalStatic}_i(\text{local}_i, \text{rcvdLocal}_j)$ is true in line 34, which leads to a call to $\text{restartLocal}_i()$ in line 35.

We provide details about how this can occur. Consider a step $a_x \in R$ of p_i that includes a call to $\text{restartLocal}_i()$ due to a message arrival (line 35). Hence, $\text{equalStatic}_i(\text{local}_i, \text{rcvdLocal}_j)$ was true during a_x . The fact that a_x includes a call to $\text{restartLocal}_i()$ 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 $\text{equalStatic}_i(\text{local}_i, \text{rcvdLocal}_j)$ is true due to the other $\mathcal{C} - 1$ messages in $\text{channel}_{j,i}$ that have stale information, since those messages appeared in the starting system state.

Such steps can be repeated at most \mathcal{C} times for $\text{channel}_{j,i}$ 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 . \square

Claim 7.22. Let $L = (N + N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ be the maximum number of labels that can appear in the system in R' (Claim 7.19). During R' , there can be at most $2N \cdot L$ calls to $\text{restartLocal}_i()$ in line 34 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 $\text{restartLocal}_i()$ in line 34.

Proof of Claim 7.22. Let $p_i, p_j \in P$ be two processors and $m_{j,i} = \langle \bullet, \langle \text{arriving}_j, \text{rcvdLocal}_j \rangle \rangle$ be a message that p_j sends to p_i by adding it to $\text{channel}_{j,i}$. 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 $\text{restartLocal}_i()$ and obtain $\text{local}_i = Z$. 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 $\text{restartLocal}_i()$ (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 30, 33, and 34). In the proof of this claim, we assume that the token passing mechanism has stabilized (since Claim 7.21 has already showed that the token passing mechanism of lines 30, 33, and 34 can cause an additive (bounded) number of calls to $\text{restartLocal}()$). 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 34 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 34 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 30, 33, and 34), 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 $\ell_{i_2} = Z_{i_2}.curr.l = Z_{i_2}.prev.l$ 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 32. Then ℓ_{i_2} is equal to either $Z_{i_1}.curr.l$ (Case a), or $Z_{j_1}.curr.l$ (Case b), or ℓ_{i_3} is different than both $Z_{i_1}.curr.l$ and $Z_{j_1}.curr.l$ (Case c). The latter case refers to a situation in which $Z_{i_1}.curr.l$ and $Z_{j_1}.curr.l$ cancel each other and p_i produces a new label in line 32 (which is then used in line 35).

Case a In this case $Z_{j_1}.curr.l \prec_{lb} Z_{i_1}.curr.l$ holds (cf. Section 3.2 regarding the \prec_{lb} relation). That is, $Z_{i_1}.curr.l$ was the value of $local_i.curr.l$ in the system state before a_x , $Z_{i_1}.curr.l$ remains as the maximal label of p_i even after p_i receives Z_{j_1} in a_x , and hence p_i uses $Z_{i_1}.curr.l$ in the return pair of $restartLocal_i()$ in a_x , $Z_{i_2} = \langle \langle Z_{i_1}.curr.l, zrs, zrs \rangle, \langle Z_{i_1}.curr.l, zrs, zrs \rangle \rangle$. 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 $Z_{i_2}.curr.l = Z_{i_1}.curr.l$, or a message from p_i with a pair which has a label larger than $Z_{i_2}.curr.l$ (due to the token passing mechanism in lines 30, 33, and 34). Since $Z_{j_1}.curr.l \prec_{lb} Z_{i_1}.curr.l$ (this case's assumption), we have that $Z_{j_1}.curr.l$ cannot be the label that appears in $local_j.curr.l$ after a_{y_a} , because during a_{y_a} line 32 causes p_j to adopt the label $Z_{i_2}.curr.l = Z_{i_1}.curr.l$, since we have $Z_{j_1}.curr.l \prec_{lb} Z_{i_1}.curr.l$ (or a label larger than $Z_{i_1}.curr.l$).

Case b In this case $Z_{i_1}.curr.l \prec_{lb} Z_{j_1}.curr.l$ holds, due to the fact that p_i sets $local_i = Z_{i_2} = \langle \langle Z_{j_1}.curr.l, zrs, zrs \rangle, \langle Z_{j_1}.curr.l, zrs, zrs \rangle \rangle$ 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 $\ell_{i_2} = Z_{i_2}.curr.l = Z_{j_1}.curr.l$, 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. 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 18) or a call to $restartLocal_i()$ in a_{z_b}). Hence, $Z_{i_3}.curr.l$ is either larger than ℓ_{i_2} or cancels ℓ_{i_2} . Subsequently, once p_j receives $Z_{i_3}.curr.l$ from p_i (due to the token passing mechanism in lines 30, 33, and 34), $local_j.curr.l$ will change to either $Z_{i_3}.curr.l$ 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 $Z_{i_2}.curr.l = Z_{j_1}.curr.l$ 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 $local_j.curr.l$ for a label with a larger label than $Z_{i_2}.curr.l = Z_{j_1}.curr.l$. 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 $local_j.curr.l$ cannot be equal to $Z_{j_1}.curr.l$.

Case c In this case both $Z_{i_1}.curr.l$ and $Z_{j_1}.curr.l$ 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 $curr.l$ that is larger than $Z_{i_2}.curr.l$), it holds that $\mathcal{S}(local_j) \neq \mathcal{S}(Z_{j_1})$, since $Z_{j_1}.curr.l$ 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

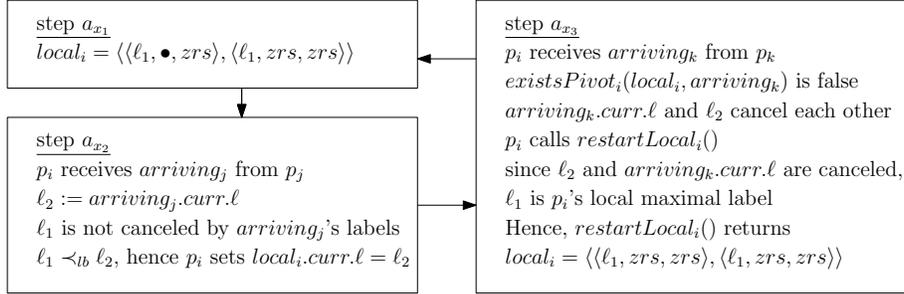


Figure 6: Recycling of a pair static part by a processor p_i . In the end of step a_{x_1} , processor p_i stores $local_i = \langle \langle \ell_1, \bullet, zrs \rangle, \langle \ell_1, zrs, zrs \rangle \rangle$. In step a_{x_2} , p_i receives $arriving_j$ from p_j , such that $\ell_2 := arriving_j.curr.\ell$ 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 7.22, Part II).

(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 Figure 6.

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 28 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. \square

In the proof of Claim 7.23, we use the claims of this lemma and Lemma 7.14 to show that the number of steps in R' that include a call to $restartLocal()$ due to line 35 is significantly less than $|R'|$.

Claim 7.23. *The number of steps that include a call to $restartLocal()$ due to line 35 in R' is in $\mathcal{O}(N^8\mathcal{C})$.*

Proof of Claim 7.23. 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 $V = N + N^2 + N^3\mathcal{C}$ the maximum number of steps that include a call to $revive()$ (Claim 7.18) and $L = (N + N^2 + N^3 \cdot \mathcal{C}) + (4N^2 + 4NM - 4N - 2M) \in \mathcal{O}(\mathcal{C}N^3)$ the maximum number of labels that can be created during R' (Claim 7.19).

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 7.14, 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 35), 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 7.19 (since their static part only differs on ℓ).

Hence, if there are no calls to $revive()$ during R' , there can be at most $(2\mathcal{C} + 1)N^2 + \mathcal{C}N^2 + N^2L(N + M + L)$ calls to $restartLocal()$. This bound holds, since at most $2\mathcal{C} + 1$ calls to $restartLocal()$ can be caused due Claim 7.20, $\mathcal{C}N^2$ due to Claim 7.21, 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 7.22 (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: $(2C+1)N^2 + CN^2 + 2N^2L(N+M+L+2V) \in \mathcal{O}(N^8C)$. \square

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 28 at most once for any execution and in Claim 7.23 we showed that during R' each processor calls $restartLocal()$ in line 35 in a number of steps that is significantly less than $MAXINT$. Thus, the number of steps in which a processor calls $revive()$ (Claim 7.18) or $restartLocal()$ during R' is significantly less than $MAXINT$. \square

Corollary 7.24. *Let R be an \mathcal{L}_S -scale execution of Algorithm 1. By the definition of \mathcal{L}_S -scale (Section 2), there exists an integer $x \ll MAXINT$, such that $|R| = x \cdot MAXINT$ holds. By Lemma 7.17 the number of steps in which a processor calls $restartLocal()$ or $revive()$ in every $MAXINT$ -segment R' of R is significantly less than $|R'| = MAXINT$. Hence, since $x \ll MAXINT$, the number of steps in which a processor calls $restartLocal()$ or $revive()$ in R is also significantly less than $|R|$. Therefore, by Lemma 7.12 the number of states in R in which Requirement 1 does not hold is significantly less than $|R|$, and thus (by Definition 2.3) Algorithm 1 is practically-self-stabilizing.*

8 Conclusion

Self-stabilization often requires, within a bounded recovery period, the complete absence of stale information (that is due to transient faults). This paper studies stabilization criteria that are less restrictive than 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 (Section 4) and present an elegant technique for dealing with concurrent overflow events (Section 5). We believe that the proposed algorithm (Section 6) and its techniques can be the basis of other practically-self-stabilizing algorithms.

References

- [1] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In *Distributed Computing, 18th International Conference*,

- DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 102–116, 2004.
- [2] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.
- [3] Anish Arora, Sandeep S. Kulkarni, and Murat Demirbas. Resettable vector clocks. *J. Parallel Distrib. Comput.*, 66(2):221–237, 2006.
- [4] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing paxos replicated state-machine. In Guevara Noubir and Michel Raynal, editors, *Networked Systems - Second International Conference, NETYS 2014, Marrakech, Morocco, May 15-17, 2014. Revised Selected Papers*, volume 8593 of *Lecture Notes in Computer Science*, pages 99–121. Springer, 2014.
- [5] Silvia Bonomi, Shlomi Dolev, Maria Potop-Butucaru, and Michel Raynal. Stabilizing server-based storage in byzantine asynchronous message-passing systems: Extended abstract. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 471–479. ACM, 2015.
- [6] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.
- [7] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *J. Parallel Distrib. Comput.*, 70(12):1220–1230, 2010.
- [8] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
- [9] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.
- [10] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
- [11] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-FIFO channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.
- [12] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Practically stabilizing virtual synchrony. *CoRR*, abs/1502.05183, 2015. An earlier version appeared in the Proceedings of the 17th International Symposium Stabilization, Safety, and Security of Distributed Systems, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015.

- [13] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-FIFO) dynamic networks. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, pages 133–147, 2012.
- [14] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8):884–900, 2010.
- [15] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1987. <http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf>.
- [16] Chryssis Georgiou and Alexander A Shvartsman. Cooperative task-oriented computing: Algorithms and complexity. *Synthesis Lectures on Distributed Computing Theory*, 2(2):1–167, 2011.
- [17] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
- [18] Amos Israeli and Ming Li. Bounded time-stamps. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 371–382. IEEE, 1987.
- [19] Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2015.
- [20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [21] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, 2007.
- [22] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: IEEE, 1994, pp. 123–133.) <https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf>.
- [23] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.
- [24] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400, 2011.

- [25] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 133–142, 1981.
- [26] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.