Practically-Self-Stabilizing Virtual Synchrony[*]

Shlomi Dolev [†] Chryssis Georgiou [‡] Ioannis Marcoullis[‡] Elad M. Schiller [§]

Abstract

The virtual synchrony abstraction was proven to be extremely useful for asynchronous, large-scale, message-passing distributed systems. Self-stabilizing systems can automatically regain consistency after the occurrence of transient faults.

We present the first practically-self-stabilizing virtual synchrony algorithm that uses a new counter algorithm that establishes an efficient practically unbounded counter, which in turn can be directly used for emulating a self-stabilizing Multiple-Writer Multiple-Reader (MWMR). Other self-stabilizing services include membership, multicast, and replicated state machine (RSM) emulation. As we base the latter on virtual synchrony, rather than consensus, the system can progress in more extreme asynchronous executions than consensus-based RSM emulations.

1 Introduction

Virtual Synchrony (VS) is an important property provided by several Group Communication Systems (GCSs) that has proved to be valuable in the scope of fault-tolerant distributed systems where communicating processors are organized in process groups with changing membership [5]. During the computation, groups change allowing an outside observer to track the history (and order) of the groups, as well as the messages exchanged within each group. The VS property guarantees that any two processors that both participate in two consecutive such groups, should deliver the same messages in their respective group. Systems that support the VS abstraction are designed to operate in the presence of fail-stop failures of a minority of the participants. Such a design fits large computer clusters, data-centers and cloud computing, where at any given time some of the processing units are non-operational. Systems that cannot tolerate such failures degrade their functionality and availability to the degree of unuseful systems.

Group communication systems that realize the VS abstraction provide services, such as *group membership* and *reliable group multicast*. The group membership service is responsible for providing the current *group view* of the recently live and connected group members, i.e., a processor set and a unique *view identifier*, which is a sequence number of the view installation. The reliable group multicast allows the service clients to exchange messages with the group members as if it was a single communication endpoint with a single network address and to which messages are delivered in an atomic fashion, thus any message is either delivered to all recently live and connected group members prior to the next message, or is not delivered to any member. The challenges related to VS consist of the need to maintain atomic message delivery in the presence of asynchrony and crash failures. VS facilitates the implementation of a replicated state machine [5] that is more efficient than classical consensus-based implementations that start every multicast round with an agreement on the set of recently live and connected processors. It is also usually easier to implement [5].

[*]A preliminary version of this work has appeared in the proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (*SSS'15*).

[†]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel. Email `dolev@cs.bgu.ac.il`
Partially supported by Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences and Israel Science Foundation (grant number 428/11).

[‡]Department of Computer Science, University of Cyprus, Nicosia, Cyprus. Email `{chryssis, imarco01} @cs.ucy.ac.cy`.
The work of Ioannis Macoullis is supported by a Doctoral Scholarship program of the University of Cyprus.

[§]Department of Engineering and Computer Science, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden, Email `elad@chalmers.se`.

Transient faults Transient violations of design assumptions can lead a system to an arbitrary state. For example, the assumption that error detection ensures the arrival of correct messages and the discarding of corrupted messages, might be violated since error detection is a probabilistic mechanism that may not detect a corrupt message. As a result, the message can be regarded as legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, requiring human intervention. In the presence of transient faults, large multicomputer systems providing VS-based services can prove hard to manage and control. One key problem, not restricted to virtually synchronous systems, is catering for counters (such as view identifiers) reaching an arbitrary value. How can we deal with the fact that transient faults may force counters to wrap around to the zero value and violate important system assumptions and correctness invariants, such as the ordering of events? A self-stabilizing algorithm [13] can automatically recover from such unexpected failures, possibly as part of after-disaster recovery or even after benign temporal violations of the assumptions made in the design of the system. To the best of our knowledge, no *stabilizing virtual synchrony* solution exists. We tackle this issue in our work.

Practically-self-stabilization A relatively new self-stabilization paradigm is *practically-self-stabilization* [1, 7, 16]. Consider an asynchronous system with bounded memory and data link capacity in which corrupt pieces of data (stale information) exist due to a transient fault. (Recall that transient faults can result in the appearance of corrupted information, which the system tends to spread and thus reach an arbitrary state.) These corrupted data may appear *unexpectedly* at any processor as they lie in communication links, or may (indefinitely) remain "hidden" in some processor's local memory until they are added to the communication links as a response to some other processor's input. Whilst these pieces of corrupted data are bounded in number due to the boundedness of the links and local memory, they can eventually force the system to lose its safety guarantees. Such corrupt information may repeatedly drive the system to an undesired state of non-functionality. This is true for all systems and self-stabilizing systems are required to eradicate the presence of all corrupted information. In fact, whenever they appear, the self-stabilizing system is required to regain consistency and in some sense stabilize. One can consider this as an adversary with a limited number of chances to interrupt the system, but only itself knows *when* it will do this.

In this perspective, self-stabilization, as it was proposed by Dijkstra [12], is not the best design criteria for asynchronous systems for which we cannot specifically define *when* stabilization is expected to finish (in some metric like asynchronous cycles, for example). The newer criterion of practically-stabilizing systems is closely related to pseudo-self-stabilizing systems [9], as we explain next. Burns, Gouda and Miller [9] deal with the above challenge by proposing the design criteria of *pseudo-self-stabilization*, which merely bounds the number of possible safety violations. Namely, their approach is to abandon Dijkstra's seminal proposal [12] to bound the period in which such violations occur (using some metric like asynchronous cycles). We consider a variation on the design criteria for pseudo-self-stabilization systems that can address additional challenges that appear when implementing a decentralized shared counter that uses a constant number of bits.

Self-stabilizing systems can face an additional challenge due to the fact that a single transient fault can cause the counter to store its maximum possible value and still (it is often the case that) the system needs to be able to increment the counter for an unbounded number of times. The challenge becomes greater when there is no elegant way to show that the system can always maintain an order among the different values of the counter by, say, wrapping to zero in such integer overflow events. Arora, Kulkarni and Demirbas [2] overcome the challenge of integer overflow by using non-blocking resets in the absence of faults described [2]. In case faults occur, the system recovery requires a blocking operation, which performs a distributed global reset. This work considers a design criteria for message passing systems that perform in a wait-free manner also when recovering from transient faults.

Note that, from the theoretical point of view, systems that take an extraordinary large number of steps (that accedes the counter maximum value, or even an infinite number of steps) are bound to violate any ordering constraints. This is because of the asynchronous nature of the studied system, which could arbitrarily delay a node from taking steps or defer the arrival of a message until such violations occur after, say, a counter wraps around to zero. Having practical systems in mind, we consider systems for which the number of sequential steps that they can take throughout their lifetime is not greater than an integer that can be represented using a constant number of bits. For example, Dolev, Kat and Schiller [16] assume that counting from zero to $2^{64} - 1$ using sequential steps is not possible in any practical system and thus consider only a *practically infinite period*, of 2^{64} sequential steps, that the system takes when demonstrating that safety is not violated. The design criteria of practically-self-stabilizing systems [1, 7, 22] requires that there is a bounded number of possible safety violations during any practically infinite period of the system execution. For such (message passing) systems, we provide a decentralized shared counter algorithm that performs in a wait-free manner also when recovering from transient faults.

Contributions We present the first practically-self-stabilizing (or *practically-stabilizing*) virtual synchrony solution. Specifically:

- We provide a practically-self-stabilizing counter algorithm using bounded memory and communication bandwidth, where many writers can increment the counter for an unbounded number of times in the presence of processor crashes and unbounded communication delays. Our counter algorithm is modular with a simple interface for increasing and reading the counter, as well as providing the identifier of the processor that has incremented it.
- At the heart of our counter algorithm is the underlying labeling algorithm that extends the label scheme of Alon et al. [1] to support *multiple* writers, whilst the algorithm specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to "discover" the greatest label and discard all obsolete ones.
- An immediate application of our counter algorithm is a practically-self-stabilizing MWMR register emulation.
- The practically-self-stabilizing counter algorithm, together with implementations of a practically-self-stabilizing reliable multicast service and membership service that we propose, are composed to yield a practically-self-stabilizing coordinator-based Virtual Synchrony solution.
- Our Virtual Synchrony solution yields a practically-self-stabilizing State Machine Replication (SMR) implementation. As this implementation is based on virtual synchrony rather than consensus, the system can progress in more extreme asynchronous executions than consensus-based SMR implementations.

Related Work Leslie Lamport was the first to introduce SMR, presenting it as an example in [20]. Schneider [23] gave a more generalized approach to the design and implementation of SMR protocols. Group communication services can implement SMR by providing reliable multicast that guarantees VS [4]. Birman et al. were the first to present VS and a series of improvements in the efficiency of ordering protocols [6]. Birman gives a concise account of the evolution of the VS model for SMR in [5].

Research during the last recent decades resulted in an extensive literature on ways to implement VS and SMR, as well as industrial construction of such systems. A recent research line on stabilizing versions of replicated state machines [1, 7, 16, 17] obtains self-stabilizing replicated state machines in shared memory as well as in synchronous and asynchronous message passing systems.

The bounded labeling scheme and the use of practically unbounded sequence numbers proposed in [1], allow the creation of practically-stabilizing bounded-size solutions to the never-exhausted counter problem in the restricted case of a single writer. In [7] a practically-self-stabilizing version of Paxos was developed, which led to a practically-self-stabilizing consensus-based SMR implementation. To this end, they extended the labeling scheme of [1] to allow for multiple counter writers, since unbounded counters are required for ballot numbers. Extracting this scheme for other uses does not seem intuitive. We present a simpler and significantly more communication efficient practically infinite counter that also supports many writers, where only a pair of labels rather than a vector of labels needs to be communicated. Our solution is *highly modular* and can be easily used in any similar setting requiring such counters. We also note that with [1]'s single writer atomic register emulation, a quorum read of the value could return without a value if the writer did not before perform a write to establish a maximal tag. An emulation based on our multiple-writer version guarantees that reads may always terminate with a value, since our labeling algorithm continuously maintains a maximal tag.

In what follows, Section 2 presents the system settings and the necessary definitions. Section 3 details the practically-self-stabilizing Labeling Scheme and Increment Counter algorithms. In Section 4 we present the practically-self-stabilizing Virtual Synchrony algorithm and the resulting replicate state machine emulation. We conclude with Section 5.

2 System Settings and Definitions

We consider an asynchronous message-passing system. The system includes a set P of n communicating processors; we refer to the processor with identifier i, as p_i. At most $n/2 - 1$ processors may fail by crashing and these may sometimes be referred to as *inactive* in contrast to *active* processors that are not crashed. We assume that the system runs on top of a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels as the ones of [14, 15]. The network topology is of a fully connected graph where every two processors exchange (low-level messages called) *packets* to enable a reliable delivery of (high level) messages. When no confusion is possible we use the term messages for packets.

Communication and data link implementation The communication links have bounded capacity, so that the number of messages in every given instance is bounded by a constant *cap*, which is known to the processors. When processor p_i sends a packet, π, to processor p_j, the operation *send* inserts a copy of π to the FIFO queue that represents the communication channel from p_i to p_j, while respecting an upper bound on the number of packets in the channel, possibly omitting the new packet or one of the already sent packets. When p_j receives π from p_j, π is dequeued from the queue representing the channel. We assume that packets can be spontaneously omitted (lost) from the channel, however, a packet that is sent infinitely often is received infinitely often.

One version of a self-stabilizing FIFO data link implementation that we can use, is based on the fact that communication links have bounded capacity. Packets are retransmitted until more than the total capacity acknowledgments arrive; while acknowledgments are sent only when a packet arrives (not spontaneously) [14, 15]. Over this data-link, the two connected processors can constantly exchange a "token". Specifically, the sender (possibly the processor with the highest identifier among the two) constantly sends packet π_1 until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π_2, and so on and so forth. This assures that the receiver has received packet π_1 before the sender starts sending packet π_2. This can be viewed as a token exchange. We use the abstraction of the token carrying messages back and forth between any two communication entities. a *heartbeat* to (imperfectly) detect whether a processor is active or not; when a processor in no longer active, the token will not be returned back to the other processor.

Definitions and complexity measures Every processor, p_i, executes a program that is a sequence of *(atomic) steps*, where a *step* starts with local computations and ends with a single communication operation, which is either *send* or *receive* of a packet. For ease of description, we assume the interleaving model, where steps are executed atomically, a single step at any given time. An input event can be either the receipt of a packet or a periodic timer triggering p_i to (re)send. Note that the system is asynchronous and the rate of the timer is totally unknown.

The *state*, s_i, of a node p_i consists of the value of all the variables of the node including the set of all incoming communication channels. The execution of an algorithm step can change the node's state. The term *(system) configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n)$, where each s_i is the state of node p_i (including messages in transit to p_i). We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ as an alternating sequence of system configurations c_x and steps a_x, such that each configuration c_{x+1}, except

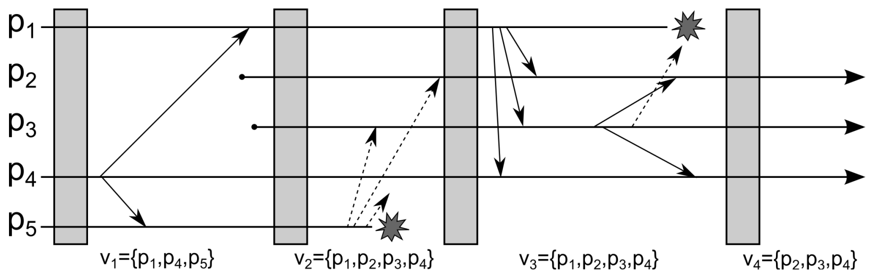

Figure 1: An execution satisfying the VS property. The grey boxes indicate a new view installation, and the example shows four views. View v_1 initially with membership $\{p_1, p_4, p_5\}$. The reliable multicast reaches all members of the group. Two new processors p_2 and p_3 join the group, forming view v_2. In this view, p_5 crashes before completing its multicast which is ignored (dashed lines). The new view v_3 is formed to exclude p_5, and in it, p_1 manages a successful multicast before crashing. The multicast of p_3 is reliable and guaranteed to be delivered to all non-crashed within the view, that is excluding p_1 which might or might not have received it (dotted line). A new view is then formed to encapture the failure of p_1.

the initial configuration c_0, is obtained from the preceding configuration c_x by the execution of the step a_x.

An execution R_p is *practically infinite execution* if it contains a chain of steps ordered according to Lamport's happened-before relation [20] that are longer than 2^τ (τ being, for example, 64), namely they are practically infinite for any given system [16]. Similar to an infinite execution, a processor that fails by crashing stops taking steps, and any processor that does not crash eventually takes a practically infinite number of steps. The code of self-stabilizing algorithms reflects the requirement for non-termination in that it usually consists of a `do`$-$`forever` loop that contains communication operations with the neighbors and validation that the system is in a consistent state as part of the transition decision. An *iteration* of an algorithm formed as a `do`$-$`forever` loop is a complete run of the algorithm starting in the loop's first line and ending at the last line, regardless of whether it enters branches. (Note that an iteration may contain many steps).

We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold, we use the term *safe configuration* for any configuration in any execution in LE. As defined by Dijkstra in [12], an algorithm is *self-stabilizing* with relation to the task LE when every (unbounded) execution of the algorithm reaches a safe configuration with relation to the algorithm and the task. 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 implies the desired system behavior [13]. Note that an execution R can satisfy the abstract task and still not belong to LE, because R considers only a subset of variables, whereas the configurations of executions that are in LE consider every variable in the processor states and message fields. An algorithm is *practically-self-stabilizing* (or just *practically-stabilizing*) with relation to the task $\mathcal{T}$ if in any practically infinite execution has a bounded number of deviations $\mathcal{T}$ [22].

This defines a measure for complexity. The asynchrony of the system makes it hard, if not impossible to infer anything on stabilization *time*, since we cannot predict *when* an element from the corrupt state of the system will reach a processor, (cf. self-stabilizing solutions that give time complexity in asynchronous rounds). Based on the above definition of practically-stabilizing algorithms, a bounded number of corrupt elements that might force the system to *deviate* from its task even if these may or may not (due to asynchrony) appear. Whenever a deviation happens, a number of algorithmic operations are required to satisfy $\mathcal{T}$ once again. As a complexity measure, we bound the total of these operations throughout an execution. These operations differ by algorithm, i.e., it is label creations in the labeling scheme, counter increments for the counter increment algorithm and view creations in the virtual synchrony algorithm.

The *virtual synchrony task* uses the notion of a *view*, a group of processors that perform multicast

within the group and is uniquely identified, to ensure that any two processors that belong to two views that are consecutive according to their identifier, *deliver* identical message sets in these views. The legal execution of virtual synchrony is defined in terms of the input and output sequences of the system with the environment. When a majority of processors are continuously active every external input (and only the external inputs) should be atomically accepted and processed by the majority of the active processors. The system works in the primary component, i.e., it does not deal with partitions and requires that a view contains a majority of the system's processors, i.e., its membership size is always greater than $n/2$. Therefore, there is no delivery and processing guarantee in executions in which there is no majority, still in these executions any delivery and processing is due to a received environment input. Figure 1 is an example of a virtually synchronous execution.
Notation. Throughout the paper we use the following notation. Let y and y' be two objects that both include the field x. We denote $(y =_x y') \equiv (y.x = y'.x)$.

3 Practically-Self-Stabilizing Labeling Scheme and Counter Algorithm

Many system like the ones performing replication (e.g. GCSs requiring group identifiers, of Paxos implementations requiring ballot numbers) assume access to an infinite (unbounded) counter. We proceed to give a practically-stabilizing, practically infinite counter based on a bounded labeling scheme. Note that by a *practically infinite (*or *unbounded) counter* we imply that a τ-bit counter (e.g., 64-bit) is not truly infinite (since this is anyway not implementable on hardware), but it is large enough to provide counters for the lifetime of most conceivable systems when started at 0. We refer the reader to the example provided by Blanchard et al. [8], where a 64-bit counter initialized at 0 and incremented per nanosecond is calculated to last for around 500 years, essentially an infinity for most of today's running systems.

The task of a practically-self-stabilizing labeling scheme is for every processor that takes an infinite number of steps to reach to a label that is maximal for all active processors in the system. The task of maintaining a practically infinite counter, is for every processor that takes an infinite yet bounded number of steps, to eventually be able to monotonically increment the counter from 0 to 2^τ. The latter task depends on the former to provide the maximal label in the system to be used as a sequence number epoch, so that within the same epoch, the integer sequence number is incremented as a practically infinite counter. It is implicit that the tasks are performed in the presence of corrupt information that might exist due to transient faults.

Our solutions are *practically infinite*, in the following way. A bounded amount of stale information from the corrupt initial state, may unpredictably corrupt the counter. In such cases, processors are forced to change their labels and restart their counters. A processor cannot predict whether a corrupt piece of information exists, or *when* will it make its appearance as this is essentially the work of asynchrony. Our solutions guarantee that only a bounded number of labels will need to change, or that only a bounded number of counter increments will need to take place before we reach to one that is eligible to last its full 2^τ length, less the fact that this maximal value is practically unattainable.

We first present and prove the correctness of a practically-stabilizing labeling algorithm, and then explain how this can be extended to implement practically stabilizing, practically unbounded counters in Section 3.3.

3.1 Labeling Algorithm for Concurrent Label Creations

3.1.1 Preliminaries

Bounded labeling scheme The bounded labeling scheme of Alon et al. [1] implements an SWMR register emulation in a message-passing system. The *labels* (also called *epochs*) allow the system to stabilize, since once a label is established, the integer counter related to this label is considered to be practically infinite, as a 64-bit integer is practically infinite and sufficient for the lifespan of any reasonable

Algorithm 1: The $nextLabel()$ function; code for p_i

```
1 For any non-empty set $X \subseteq D$, function $pick(d, X)$ returns $d$ arbitrary elements of $X$;
  input  : $S = \langle \ell_1, \ell_2 \ldots, \ell_k \rangle$ set of $k$ labels.
  output : $\langle i, newSting, newAntistings \rangle$
2 let $newAntistings = \{\ell_j.sting : \ell_j \in S\}$;
3 $newAntistings \leftarrow newAntistings \cup pick(k - |newAntistings|, D \setminus newAntistings)$;
4 return $\langle i, pick(1, D \setminus (newAntistings \cup \{\cup_{\ell_j \in S} \ell_j.Antistings\})), newAntistings \rangle$;
```

system. We extend the labeling scheme of [1] to support multiple writers, by including the epoch creator (writer) identity to break symmetry, and decide which epoch is the most recent one, even when two or more creators concurrently create a new label.

Formally defined, we consider the set of integers $D = [1,\ k^2 + 1]$ such that $k \in \mathbb{N}$ a known constant to the processors, which we determine in Corollary 3.2. A *label* (or *epoch*) is a triple $\langle lCreator, sting, Antistings \rangle$, where $lCreator$ is the identity of the processor that established (created) the label, $Antistings \subset D$ with $|Antistings| = k$, and $sting \in D$. Given two labels ℓ_i, ℓ_j, we define the relation $\ell_i \prec_{lb} \ell_j \equiv (\ell_i.lCreator < \ell_j.lCreator) \vee (\ell_i.lCreator = \ell_j.lCreator \wedge ((\ell_i.sting \in \ell_j.Antistings)$ $\wedge\ (\ell_j.sting \notin \ell_i.Antistings)))$; we use $=_{lb}$ to say that the labels are identical. Note that the relation $\prec_{lb}$ does not define a total order. For example, when $\ell_i =_{lCreator} \ell_j$ and $(\ell_i.sting \notin \ell_j.Antistings)$ and $(\ell_j.sting \notin \ell_i.Antisting)$ these labels are *incomparable*.

As an example, consider the situation with $k = 3$, and $D = \{1, 2, \ldots, 10\}$. Assume the existence of three labels $\ell_1 = \langle i, 2, \langle 3, 5, 9 \rangle \rangle$, $\ell_2 = \langle i, 1, \langle 2, 9, 10 \rangle \rangle$, and $\ell_3 = \langle i + 1, 1, \langle 3, 5, 9 \rangle \rangle$. In this case, $\ell_1 \prec_{lb} \ell_3$ and $\ell_2 \prec_{lb} \ell_3$, since the creator of ℓ_3 has a greater identity than the creator of ℓ_1 and ℓ_2. We can also see that $\ell_1 \prec_{lb} \ell_2$, since the sting of ℓ_1, namely 2, belongs to the antistings set of ℓ_2 (which is $\langle 2, 9, 10 \rangle$) while the opposite is not true for the sting of ℓ_2. This makes ℓ_2 "immune" to the sting of ℓ_1.

As in [1], we demonstrate that one can still use this labeling scheme as long as it is ensured that eventually a label greater than all other labels in the system is introduced. We say that a label ℓ ***cancels*** another label ℓ', either if they are incomparable or they have the same $lCreator$ but ℓ is greater than ℓ' (with respect to $sting$ and $Antistings$). A label with creator p_i is said to belong to p_i's domain.

Creating a largest label Function $nextLabel()$, Algorithm 1, gets a set of at most k labels as input and returns a new label that is greater than all of the labels of the input, given that all the input labels have the same creator i.e., the same $lCreator$. This last condition is imposed by the labeling algorithm that calls $nextLabel()$, as we will see further down with a set of labels from the same processor. It has the same functionality as the function called $Next_b()$ in [1], but it additionally appends the label creator to the output. The function essentially composes a new $Antistings$ set from the stings of all the labels that it receives as input, and chooses a $sting$ that is in none of the $Antistings$ of the input labels. In this way it ensures that the new label is greater than any of the input. Note that the function takes k $Antistings$ of k labels that are not necessarily distinct, implying at most k^2 distinct integers and thus the choice of $|D| = k^2 + 1$ allows to always obtain a greater integer as the $sting$. For the needs of our labeling scheme, $k = 4(n^3cap + 2n^2 - 2n) + 1$ (Corollary 3.2).

Scheme idea and challenges When all processors are active, the scheme can be viewed as a simple extension of the one of [1]. Informally speaking, the scheme ensures that each processor p_i eventually "cleans up" the system from obsolete labels of which p_i appears to be the creator (for example, such labels could be present in the system's initial arbitrary state). Specifically, p_i maintains a bounded FIFO history of such labels that it has recently learned, while communicating with the other processors, and creates a label greater than all that are in its history; call this p_i's *local maximal label*. In addition, each processor seeks to learn the *globally maximal label*, that is, the label in the system that is the greatest among the local maximal ones.

We note here that compared to Alon et al. [1], which only had a single writer upon the failure of whom there would be no progress thus stabilization would not be the main concern, we have *multiple* label creators. If these creators were not allowed to crash then the extension of the scheme would be a simple exercise. Nevertheless, when some processors can crash the problem becomes incrementally more difficult as we now explain. The problem lies in cleaning the system of these crashed processors' labels since they will not "clean up" their local labels. Each active processor needs to do this itself, indirectly, without knowing which processor is inactive, i.e., we do not employ any form of failure detection for this algorithm. To overcome this problem, each processor maintains bounded FIFO histories on labels appearing to have been created by other processors. These histories eventually accumulate the obsolete labels of the inactive processors. The reader may already see that maintaining these histories, also creates another source of possible corrupt labels. We show that even in the presence of (a minority of) inactive processors, starting from an arbitrary state, the system eventually converges to use a global maximal label.

Let us explain why obsolete labels from inactive processors can create a problem when no one ever cleans (cancels) them up. Consider a system starting in a state that includes a cycle of labels $\ell_1 \prec \ell_2 \prec \ell_3 \prec \ell_1$, all of the same creator, say p_x, where $\prec$ is a relation between labels. If p_x is active, it will eventually learn about these labels and introduce a label greater than them all. But if p_x is inactive, the system's asynchronous nature may cause a repeated cyclic label adoption, especially when p_x has the greatest processor identifier, as these identifiers are used to break symmetry. Say that an active processor learns and adopts ℓ_1 as its global maximal label. Then, it learns about ℓ_2 and hence adopts it, while forgetting about ℓ_1. Then, learning of ℓ_3 it adopts it. Lastly, it learns about ℓ_1, and as it is greater than ℓ_3, it adopts ℓ_1 once more, as the greatest in the system; this can continue indefinitely. By using the bounded FIFO histories, such labels will be accumulated in the histories and hence will not be adopted again, ending this vicious cycle. We now formally present the algorithm.

3.1.2 The Labeling Algorithm

The labeling algorithm (Algorithm 2) specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to "discover" their greatest label and cancel all obsolete ones. Specifically, we define the abstract task of the algorithm as one that lets every node to maintain a variable that holds the local maximal label. We require that, after the recovery period and as long as there are no calls to $nextLabel()$ (Algorithm 1), these local maximal label actually refer to the same global maximal label.

As we will be using pairs of labels with the *same* label creator, for the ease of presentation, we will be referring to these two variables as the *(label) pair*. The first label in a pair is called ml. The second label is called cl and it is either $\perp$, or equal to a label that cancels ml (i.e., cl indicates whether ml is an obsolete label or not).

The processor state Each processor stores an array of label pairs, $max_i[n]$, where $max_i[i]$ refers to p_i's maximal label pair and $max_i[j]$ considers the most recent value that p_i knows about p_j's pair. Processor p_i also stores the pairs of the most-recently-used labels in the array of queues $storedLabels_i[n]$. The j-th entry refers to the queue with pairs from p_j's domain, i.e., that were created by p_j. The algorithm makes sure that $storedLabels_i[j]$ includes only label pairs with unique ml from p_j's domain and that at most one of them is *legitimate*, i.e., not canceled. Queues $storedLabels_i[j]$ for $i \neq j$, have size $n+m$ whilst $storedLabels_i[i]$ has size $2(mn + 2n^2 - 2n)$ where m is the system's total link capacity in labels. We later show (c.f. Lemmas 3.3 and 3.4) that these queue sizes are sufficient to prevent overflows of useful labels.

High level description Each pair of processors periodically exchange their maximal label pairs and the maximal label pair that they know of the recipient. Upon receipt of such a label pair couple, the receiving processor starts by checking the integrity of its data structures and upon finding a corruption it flushes its label history queues. It then moves to see whether the two labels that it received can cancel any of its non-canceled labels and if the received labels themselves can be canceled by labels that it has

Algorithm 2: Practically-Self-Stabilizing Labeling Algorithm; code for p_i

```
1  **Variables:**
2  $max[n]$ of $\langle ml, cl \rangle$: $max[i]$ is $p_i$'s largest label pair, $max[j]$ refers to $p_j$'s label pair (canceled when $max[j].cl \neq \bot$).
3  $storedLabels[n]$: an array of queues of the most-recently-used label pairs, where $storedLabels[j]$ holds the labels created by
     $p_j \in P$. For $p_j \in (P \setminus \{p_i\})$, $storedLabels[j]$'s queue size is limited to $(n + m)$ w.r.t. label pairs, where $n = |P|$ is the
     number of processors in the system and $m$ is the maximum number of label pairs that can be in transit in the system. The
     $storedLabels[i]$'s queue size is limited to $(n(n^2 + m))$ pairs. The operator $add(\ell)$ adds $lp$ to the front of the queue, and
     $emptyAllQueues()$ clears all $storedLabels[]$ queues. We use $lp.remove()$ for removing the record $lp \in storedLabels[]$. Note
     that an element is brought to the queue front every time this element is accessed in the queue.
4  **Notation:** Let $y$ and $y'$ be two records that include the field $x$. We denote $y =_x y' \equiv (y.x = y'.x)$
5  **Macros:**
6  $legit(lp) = (lp = \langle \bullet, \bot \rangle)$
7  $labels(lp)$ : **return** $(storedLabels[lp.ml.lCreator])$
8  $double(j, lp) = (\exists lp' \in storedLabels[j] : ((lp \neq lp') \wedge ((lp =_{ml} lp') \vee (legit(lp) \wedge legit(lp')))))$
9  $staleInfo() = (\exists p_j \in P, lp \in storedLabels[j] : (lp \neq_{lCreator} j) \vee double(j, lp))$
10 $recordDoesntExist(j) = (\langle max[j].ml, \bullet \rangle \notin labels(max[j]))$
11 $notgeq(j, lp) =$ **if** $(\exists lp' \in storedLabels[j] : (lp'.ml \not\preceq_{lb} lp.ml))$ **then return**$(lp'.ml)$ **else return**$(\bot)$
12 $canceled(lp) =$ **if** $(\exists lp' \in labels(lp) : ((lp' =_{ml} lp) \wedge \neg legit(lp')))$ **then return**$(lp')$ **else return**$(\langle \bot, \bot \rangle)$
13 $needsUpdate(j) = (\neg legit(max[j]) \wedge \langle max[j].ml, \bot \rangle \in labels(max[j]))$
14 $legitLabels() = \{max[j].ml : \exists p_j \in P \wedge legit(max[j])\}$
15 $useOwnLabel() =$ **if** $(\exists lp \in storedLabels[i] : legit(lp))$ **then** $max[i] \leftarrow lp$ **else** $storedLabels[i].add(max[i] \leftarrow$
     $\langle nextLabel(), \bot \rangle)$ // For every $lp \in storedLabels[i]$, we pass in $nextLabel()$ both $lp.ml$ and $lp.cl$.
16 **upon** $transmitReady(p_j \in P \setminus \{p_i\})$ **do transmit**$(\langle max[i], max[j] \rangle)$
17 **upon** $receive(\langle sentMax, lastSent \rangle)$ **from** $p_j$
18 **begin**
19     $max[j] \leftarrow sentMax$;
20     **if** $\neg legit(lastSent) \wedge max[i] =_{ml} lastSent$ **then** $max[i] \leftarrow lastSent$;
21     **if** $staleInfo()$ **then** $storedLabels.emptyAllQueues()$;
22     **foreach** $p_j \in P : recordDoesntExist(j)$ **do** $labels(max[j]).add(max[j])$;
23     **foreach** $p_j \in P, lp \in storedLabels[j] : (legit(lp) \wedge (notgeq(j, lp) \neq \bot))$ **do** $lp.cl \leftarrow notgeq(j, lp)$;
24     **foreach** $p_j \in P, lp \in labels(max[j]) : (\neg legit(max[j]) \wedge (max[j] =_{ml} lp) \wedge legit(lp))$ **do** $lp \leftarrow max[j]$;
25     **foreach** $p_j \in P, lp \in storedLabels[j] : double(j, lp)$ **do** $lp.remove()$;
26     **foreach** $p_j \in P : (legit(max[j]) \wedge (canceled(max[j]) \neq \langle \bot, \bot \rangle))$ **do** $max[j] \leftarrow canceled(max[j])$;
27     **if** $legitLabels() \neq \emptyset$ **then** $max[i] \leftarrow \langle \max_{\prec_{lb}}(legitLabels()), \bot \rangle$;
28     **else** $useOwnLabel()$;
```

in its history. Upon finishing this label housekeeping, it tries to find its local maximal view, first among the non-cancelled labels that other processors report as maximal, and if not such exist among its own labels. In latter case, if no such label exists, it generates a new one with a call to Algorithm 1 and using its own label queue as input. At the end of the iteration the processor is guaranteed to have a maximal label, and continues to receive new label pair couples from other processors.

Information exchange between processors Processor p_i takes a step whenever it receives two pairs $\langle sentMax, lastSent \rangle$ from some other processor. We note that in a legal execution p_j's pair includes both $sentMax$, which refers to p_j's maximal label pair $max_j[j]$, and $lastSent$, which refers to a recent label pair that p_j received from p_i about p_i's maximal label, $max_j[i]$ (line 16).

Whenever a processor p_j sends a pair $\langle sentMax, lastSent \rangle$ to p_i, this processor stores the value of the arriving $sentMax$ field in $max_i[j]$ (line 19). However, p_j may have local knowledge of a label from p_i's domain that cancels p_i's maximal label, ml, of the last received $sentMax$ from p_i to p_j that was stored in $max_j[i]$. Then p_j needs to communicate this canceling label in its next communication to p_i. To this end, p_j assigns this canceling label to $max_j[i].cl$ which stops being $\bot$. Then p_j transmits $max_j[i]$ to p_i as a $lastSent$ label pair, and this satisfies $lastSent.cl \not\preceq_{lb} lastSent.ml$, i.e., $lastSent.cl$ is either greater or incomparable to $lastSent.ml$. This makes $lastSent$ illegitimate and in case this still refers to p_i's current maximal label, p_i must cancel $max_i[i]$ by assigning it with $lastSent$ (and thus $max_i[i].cl = lastSent.cl$) as done in line 20. Processor p_i then processes the two pairs received (lines 21 to 28).

Label processing Processor p_i takes a step whenever it receives a new pair message $\langle sentMax, lastSent \rangle$ from processor p_j (line 17). Each such step starts by removing *stale* information, i.e., misplaced or doubly represented labels (line 9). In the case that stale information exists, the algorithm empties the entire label storage. Processor p_i then tests whether the arriving two pairs are already included in the label storage (*storedLabels*[]), otherwise it includes them (line 22). The algorithm continues to see whether, based on the new pairs added to the label storage, it is possible to cancel a non-canceled label pair (which may well be the newly added pair). In this case, the algorithm updates the canceling field of any label pair lp (line 23) with the canceling label of a label pair lp' such that $lp'.ml \not\preceq_{lb} lp.ml$ (line 23). It is implied that since the two pairs belong to the same storage queue, they have the same processor as creator. The algorithm then checks whether any pair of the $max_i[]$ array can cause canceling to a record in the label storage (line 24), and also line 25 removes any canceled records that share the same creator identifier. The test also considers the case in which the above update may cancel any arriving label in $max[j]$ and updates this entry accordingly based on stored pairs (line 26).

After this series of tests and updates, the algorithm is ready to decide upon a maximal label based on its local information. This is the $\preceq_{lb}$-greatest legit label pair among all the ones in $max_i[]$ with respect to their *ml* label (line 27). When no such legit label exists, p_i requests a legit label in its own label storage, $storedLabels_i[i]$, and if one does not exist, will create a new one if needed (line 28). This is done by passing the labels in the $storedLabels_i[i]$ queue to the $nextLabel()$ function. Note that the returned label is coupled with a $\perp$ as the *cl* and the resulting label pair is added to both $max_i[i]$ and $storedLabel_i[i]$.

3.2 Correctness proof

We are now ready to show the correctness of the algorithm. We begin with a proof overview.

Proof overview The proof considers a execution R of Algorithm 2 that may initiate in an arbitrary configuration (and include a processor that takes practically infinite number of steps). It starts by showing some basic facts, such as: (1) stale information is removed, i.e., $storedLabels_i[j]$ includes only unique copies of p_j's labels, and at most one legitimate such label (Corollary 3.1), and (2) p_i either adopts or creates the $\preceq_{lb}$-greatest legitimate local label (Lemma 3.2). The proof then presents bounds on the number adoption steps (Lemmas 3.3 and 3.4), that define the required queue sizes to avoid label overflows.

The proof continues to show that active processors can eventually stop adopting or creating labels, by tackling individual cases where canceled or incomparable label pairs may cause a change of the local maximal label. We show that such labels eventually disappear from the system (Lemma 3.5) and thus no new labels are being adopted or created (Lemma 3.6), which then implies the existence of a global maximal label (Lemma 3.7). Namely, there is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ that is active throughout the execution, it holds that p_i's local maximal (legit) label pair $max_i[i] = \ell_{max}$ is the $\preceq_{lb}$-greatest of all the label pairs in $max_i[]$ and there is no label pair in $storedLabels_i[j]$ that cancels ℓ_{max}, i.e., $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$. We then demonstrate that, when starting from an initial arbitrary configuration, the system eventually reaches a configuration in which there is a global maximal label (Theorem 3.3). Before we present the proof in detail, we provide some helpful definitions and notation.

Definitions We define $\mathcal{H}$ to be the set of all label pairs that can be in transit in the system, with $|\mathcal{H}| = m$. So in an arbitrary configuration, there can be up to m corrupted label pairs in the system's links. We also denote $\mathcal{H}_{i,j}$ as the set of label pairs that are in transit from processor p_i to processor p_j. The number of label pairs in $\mathcal{H}_{i,j}$ obeys the link capacity bound. Recall that the data structures used (e.g., $max_i[]$, $storedLabels_i[]$, etc) store label pairs. For convenience of presentation and when clear from the context, we may refer to the *ml* part of the label pair as "the label". Note that in this algorithm, we consider an *iteration* as the execution of lines 17–28, i.e., the receive action.

3.2.1 No stale information

Lemma 3.1 says that the predicate $staleInfo()$ (line 9) can only hold during the first execution of the $receive()$ event (line 17).

Lemma 3.1 *Let $p_i \in P$ be a processor for which $\neg staleInfo_i()$ (line 9) does not hold during the k-th step in R that includes the complete execution of the $receive()$ event (from line 17 to 28). Then $k = 1$.*

Proof. Since R starts in an arbitrary configuration, there could be a queue in $storedLabels_i[]$ that holds two label records from the same creator, a label that is not stored according to its creator identifier, or more than one legitimate label. Therefore, $staleInfo_i()$ might hold during the first execution of the $receive()$ event. When this is the case, the $storedLabels_i[]$ structure is emptied (line 21). During that $receive()$ event execution (and any event execution after this), p_i adds records to a queue in $storedLabels_i[]$ (according to the creator identifier) only after checking whether $recordDoesntExist()$ holds (line 22).

Any other access to $storedLabels_i[]$ merely updates cancelations or removes duplicates. Namely, canceling labels that are not the $\preceq_{lb}$-greatest among the ones that share the same creating processors (line 23) and canceling records that were canceled by other processors (line 24), as well as removing legitimate records that share the same ml (line 25). It is, therefore, clear that in any subsequent iteration of $receive()$ (after the first), $staleInfo()$ cannot hold. ■

Lemma 3.1 along with the lines 9 and 26 of the Algorithm, imply Corollary 3.1.

Corollary 3.1 *Consider a suffix R' of execution R that starts after the execution of a $receive()$ event. Then the following hold throughout R': (i) $\forall p_i, p_j \in P$, the state of p_i encodes at most one legitimate label, $\ell_j =_{lCreator} j$ and (ii) ℓ_j can only appear in $storedLabels_i[j]$ and $max_i[]$ but not in $storedLabels_i[k]$: $k \neq j$.*

3.2.2 Local $\preceq_{lb}$-greatest legitimate local label

Lemma 3.2 considers processors for which $staleInfo()$ (line 9) does not hold. Note that $\neg staleInfo()$ holds at any time after the first step that includes the $receive()$ event (Lemma 3.1). Lemma 3.2 shows that p_i either adopts or creates the $\preceq_{lb}$-greatest legitimate local label pair and stores it in $max_i[i]$.

Lemma 3.2 *Let $p_i \in P$ be a processor such that $\neg staleInfo_i()$ (line 9), and $L_{pre}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j]) \wedge (\exists \langle max_i[j].ml, x\rangle \in (labels(max_i[j]) \setminus \{max_i[j]\}) \Rightarrow (x = \bot))\}$ be the set of $max_i[]$'s labels that, before p_i executes lines 21 to 28, are legitimate both in $max_i[]$ and in $storedLabels_i[]$'s queues. Let $L_{post}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ and $\langle \ell, \bot\rangle$ be the value of $max_i[i]$ immediately after p_i executes lines 21 to 28. The label $\langle \ell, \bot\rangle$ is the $\preceq_{lb}$-greatest legitimate label in $L_{post}(i)$. Moreover, suppose that $L_{pre}(i)$ has a $\preceq_{lb}$-greatest legitimate label pair, then that label pair is $\langle \ell, \bot\rangle$.*

Proof. **$\langle \ell, \bot\rangle$ is the $\preceq_{\mathbf{lb}}$-greatest legitimate label pair in $\mathbf{L_{post}(i)}$.** Suppose that immediately before line 27, we have that $legitLabels_i() \neq \emptyset$, where $legitLabels_i() = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ (line 14). Note that in this case $L_{post}(i) = legitLabels_i()$. By the definition of $\preceq_{lb}$-greatest legitimate label pair and line 27, $max_i[i] = \langle \ell, \bot\rangle$ is the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$. Suppose that $legitLabels_i() = \emptyset$ immediately before line 27, i.e., there are no legitimate labels in $\{max_i[j] : \exists p_j \in P\}$. By the definition of $\preceq_{lb}$-greatest legitimate label pair and line 15, $max_i[i] = \langle \ell, \bot\rangle$ is the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$.
Suppose that rec $= \langle \ell', \bot\rangle$ is a $\preceq_{\mathbf{lb}}$-greatest legitimate label pair in $\mathbf{L_{pre}(i)}$, then $\ell = \ell'$. We show that the record rec is not modified in $max_i[]$ until the end of the execution of lines 21 to 28. Moreover, the records that are modified in $max_i[]$, are not included in $L_{pre}(i)$ (it is canceled in $storedLabels_i[]$) and no records in $max_i[]$ become legitimate. Therefore, rec is also the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$, and thus, $\ell = \ell'$.

Since we assume that $staleInfo_i()$ does not hold, line 21 does not modify rec. Lines 22, 23 and 25 might add, modify, and respectively, remove $storedLabels_i$'s records, but it does not modify $max_i[]$. Since rec is not canceled in $storedLabels_i[]$ and the $\preceq_{lb}$-greatest legitimate label pair in $max_i[]$, the predicate $(legit(max[j]) \wedge notgeq(j))$ does not hold and line 23 does not modify rec. Moreover, the records in $max_i[]$, for which that predicate holds, become illegitimate. ■

3.2.3 Bounding the number of labels

Lemmas 3.3 and 3.4 present bounds on the number of adoption steps. These are $n + m$ for labels by labels that become inactive in any point in R and $(mn + 2n^2 - 2n)$ for any active processor. Following the above, choosing the queue sizes as $n + m$ for $storedLabels_i[j]$ if $i \neq j$, and $2(nm + 2n^2 - 2n) + 1$ for $storedLabels_i[i]$ is sufficient to prevent overflows given that m is the system's total link capacity in labels.

Maximum number of label adoptions in the absence of creations Suppose that there exists a processor, p_j, that has stopped adding labels to the system (the else part of line 28), say, because it became inactive (crashed), or it names a maximal label that is the $\preceq_{lb}$-greatest label pair among all the ones that the network ever delivers to p_j. Lemma 3.3 bounds the number of labels from p_j's domain that any processor $p_i \in P$ adopts in R.

Lemma 3.3 *Let $p_i, p_j \in P$, be two processors. Suppose that p_j has stopped adding labels to the system configuration (the else part of line 28), and sending (line 16) these labels during R. Processor p_i adopts (line 27) at most $(n+m)$ labels, $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_i(\ell_j)$) where m is the maximum number of label pairs that can be in transit in the system.*

Proof. Let $p_k \in P$. At any time (after the first step in R) processor p_k's state encodes at most one legitimate label, ℓ_j, for which $\ell_j =_{lCreator} j$ (Corollary 3.1). Whenever p_i adopts a new label ℓ_j from p_j's domain (line 27) such that $\ell_j : (\ell_j =_{lCreator} j)$, this implies that ℓ_j is the only legitimate label pair in $storedLabels_i[j]$. Since ℓ_j was not transmitted by p_j before it was adopted, ℓ_j must come from p_k's state delivered by a transmit event (line 16) or delivered via the network as part of the set of labels that existed in the initial arbitrary state. The bound holds since there are n processors, such as p_k, and m bounds the number of labels in transit. Moreover, no other processor can create label pairs from the domain of p_j. ∎

Maximum number of label creations Lemma 3.4 shows a bound on the number of adoption steps that does not depend on whether the labels are from the domain of an active or (eventually) inactive processor.

Lemma 3.4 *Let $p_i \in P$ and $L_i = \ell_{i_0}, \ell_{i_1}, \ldots$ be the sequence of legitimate labels, $\ell_{i_k} =_{lCreator} i$, from p_i's domain, which p_i stores in $max_i[i]$ through the reception (line 17) or creation of labels (line 28), where $k \in \mathbb{N}$. It holds that $|L_i| \leq n(n^2 + m)$.*

Proof. Let $L_{i,j} = \ell_{i_0,j}, \ell_{i_1,j}, \ldots$ be the sequence of legitimate labels that p_i stores in $max_i[j]$ during R and $C_{i,j} = \ell^c_{i_0,j}, \ell^c_{i_1,j}, \ldots$ be the sequence of legitimate labels that p_i receives from processor p_j's domain. We consider the following cases in which p_i stores L's values in $max_i[i]$.
(1) When $\ell_{i_k} = \ell_{j_0,j'}$, where $p_j, p_{j'} \in P$ and $k \in \mathbb{N}$. This case considers the situation in which $max_i[i]$ stores a label that appeared in $max_j[j']$ at the (arbitrary) starting configuration, (i.e. $\ell_{j_0,j'} \in L_{j,j'}$). There are at most $n(n-1)$ such legitimate label values from p_i's domain, namely $n-1$ arrays $max_j[]$ of size n.
(2) When $\ell_{i_k} = \ell_{j_{k'},j'} = \ell^c_{j_0,j'}$, where $p_j, p_{j'} \in P$, $k, k' \in \mathbb{N}$ and $\ell_{j_{k'},j'} \neq \ell_{j_{k'},j}$. This case considers the situation in which $max_i[i]$ stores a label that appeared in the communication channel between p_j and $p_{j'}$ at the (arbitrary) starting configuration, (i.e. $\ell^c_{j_0,j'} \in C_{j,j'}$) and appeared in $max_j[j']$ before p_j communicated this to p_i. There are at most m such values, i.e., as many as the capacity of the communication links in labels, namely $|\mathcal{H}|$.
(3) When ℓ_{i_k} is the return value of nextLabel() (the else part of line 28). Processor p_i aims at adopting the $\preceq_{lb}$-greatest legitimate label pair that is stored in $max_i[]$, whenever such exists (line 27). Otherwise, p_i uses a label from its domain; either one that is the $\preceq_{lb}$-greatest legit label pair among the ones in $storedLabels_i[i]$, whenever such exists, or the returned value of $nextLabel()$ (line 28).

The latter case (the else part of line 28) refers to labels, ℓ_{i_k}, that p_i stores in $max_i[i]$ only after checking that there are no legitimate labels stored in $max_i[]$ or $storedLabels_i[i]$. Note that every time p_i

executes the else part of line 28, p_i stores the returned label, ℓ_{i_k}, in $storedLabels_i[i]$. After that, there are only three events for ℓ_{i_k} not to be stored as a legitimate label in $storedLabels_i[i]$:
(i) execution of line 21, (ii) the network delivers to p_i a label, ℓ', that either cancels ℓ_{i_k} ($\ell'.cl \not\preceq_{lb} \ell_{i_k}.ml$), or for which $\ell'.ml \not\preceq_{lb} \ell_{i_k}.ml$, and ($iii$) ℓ_{i_k} overflows from $storedLabels_i[i]$ after exceeding the $(n(n^2+m)+1)$ limit which is the size of the queue.

Note that Lemma 3.1 says that event (i) can occur only once (during p_i's first step). Moreover, only p_i can generate labels that are associated with its domain (in the else part of line 28). Each such label is $\preceq_{lb}$-greater-equal than all the ones in $storedLabels_i[i]$ (by the definition of $nextLabel()$ in Algorithm 1).

Event (ii) cannot occur after p_i has learned all the labels $\ell \in remoteLabels_i$ for which $\ell \notin storedLabels_i[i]$, where $remoteLabels_i = (((\cup_{p_j \in P} localLabels_{i,j}) \cup \mathcal{H}) \setminus storedLabels_i[i])$ and $localLabels_{i,j} = \{\ell' : \ell' =_{lCreator} i, \exists p_j \in P : ((\ell' \in storedLabels[i]) \vee (\exists\, p_k \in P : \ell' = max_j[k].ml))\}$. During this learning process, p_i cancels or updates the cancellation labels in $storedLabels_i[i]$ before adding a new legitimate label. Thus, this learning process can be seen as moving labels from $remoteLabels_i$ to $storedLabels_i[i]$ and then keeping at most one legitimate label available in $storedLabels_i[i]$. Every time $storedLabels_i[i]$ accumulates a label ℓ that was unknown to p_i, the use of $nextLabel()$ allows it to create a label ℓ_{i_k} that is $\preceq_{lb}$-greater than any label pair in $storedLabels_i[i]$ and eventually from all the ones in $remoteLabels_i$.

Note that $remoteLabels_i$'s labels must come from the (arbitrary) start of the system, because p_i is the only one that can add a label to the system from its domain and therefore this set cannot increase in size. These labels include those that are in transit in the system and all those that are unknown to p_i but exist in the $max_j[\bullet]$ or $storedLabels_j[i]$ structures of some other processor p_j. By Lemma 3.3 we know that $|storedLabels_j[i]| \leq n+m$ for $i \neq j$. From the three cases of L_i labels that we detailed at the beginning of this proof ((1)–(3)), we can bound the size of $remoteLabels_i$ as follows: for $p_j \in P : j \neq i$ we have that $|remoteLabels_i| \leq (n-1)(|max[]|+|storedLabels_j[i]|)+|\mathcal{H}| = (n-1)(n+(n+m))+m = mn+2n^2-2n$. Since p_i may respond to each of these labels with a call to $nextLabel()$, we require that $storedLabels_i[i]$ has size $2|remoteLabels_i|+1$ label pairs in order to be able to accommodate all the labels from $|remoteLabels_i|$ and the ones created in response to these, plus the current greatest. Thus, what is suggested by event (ii) of p_i, i.e., receiving labels from $remoteLabels_i$, stops happening before overflows (event (iii)) occurs, since $storedLabels_i[i]$ has been chosen to have a size that can accommodate all the labels from $remoteLabels_i$ and those created by p_i as a response to these. This size is $2(mn+2n^2-2n)+1 = 2(n^3cap+2n^2-2n)+1$ (since $m = n^2cap$) which is $O(n^3)$. ■

From the end of the proof of Lemma 3.4, we get Corollary 3.2.

Corollary 3.2 *The number k of antistings needed by Algorithm 1 is $2\cdot(2(n^3cap+2n^2-2n)+1)$ (twice the queue size).*

3.2.4 Pair diffusion

The proof continues and shows that active processors can eventually stop adopting or creating labels. We are particularly interested in looking into cases in which there are canceled label pairs and incomparable ones. We show that they eventually disappear from the system (Lemma 3.5) and thus no new labels are being adopted or created (Lemma 3.6), which then implies the existence of a global maximal label (Lemma 3.7).

Lemmas 3.5 and 3.6, as well as Lemma 3.7 and Theorem 3.3 assume the existence of at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Suppose that processor $p_i \in P$ takes a bounded number of steps in R during a period in which $p_{unknown}$ takes a practically infinite number of steps. We say that p_i has become inactive (crashed) during that period and assume that it does not resume to take steps at any later stage of R (in the manner of fail-stop failures, as in Section 2).

Consider a processor $p_i \in P$ that takes any number of (bounded or practically infinite) steps in R and two processors $p_j, p_k \in P$ that take a practically infinite number of steps in R. Given that p_j has a label pair ℓ as its local maximal, and there exists another label pair ℓ' such that $(\ell'.ml \not\preceq_{lb} \ell.ml) \vee \ell'.cl \not\preceq_{lb} \ell.ml$

Notation	**Definition**	**Remark**
$hName_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as $sentMax$ feedback about $max_k[k]$
$hAck_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as $lastSent$ feedback about $max_k[j]$
$max_{i,j,k}$	$\{(max_j[j], max_k[k])\}$	Local maximal labels of p_j and p_k
$ack_{i,j,k}$	$\{(max_j[j], max_k[j])\}$	ℓ_j is p_j's local maximal label and $\ell_k = max_k[j]$
$stored_{i,j,k}$	$\{\{max_j[j]\} \times storedLabels_k[i]\}$	A label ℓ_k in $storedLabels_k[i]$ that can cancel $\ell_j = max_j[j]$

Table 1: The notation used to identify the possible positions of label pairs ℓ_j and ℓ_k that can cause canceling as used in Lemmas 3.5 to 3.7 and in Theorem 3.3.

and they have the same creator p_i. Algorithm 2 suggests only two possible routes for some label pair ℓ' to find its way in the system through p_j. Either by p_j adopting ℓ' (line 27), or by creating it as a new label (the else part of line 28). Note, however, that p_j is not allowed to create a label in the name of p_i and since $\ell' =_{lCreator} i$, the only way for ℓ' to disturb the system is if this is adopted by p_j as in line 27. We use the following definitions for estimating whether there are such label pairs as ℓ and ℓ' in the system.

There is a *risk* for two label pairs from p_i's domain, ℓ_j and ℓ_k, to cause such a disturbance when either they cancel one another or when it can be found that one is not greater than the other. Thus, we use the predicate $risk_{i,j,k}(\ell_j, \ell_k) = (\ell_j =_i \ell_k) \wedge legit(\ell_j) \wedge (notGreater(\ell_j, \ell_k) \vee canceled(\ell_j, \ell_k))$ to estimate whether p_j's state encodes a label pair, $\ell_j =_{lCreator} i$, from p_i's domain that may disturb the system due to another label, ℓ_k, from p_i's domain that p_k's state encodes, where $canceled(\ell_j, \ell_k) = (legit(\ell_j) \wedge \neg legit(\ell_k) \wedge \ell_j =_{ml} \ell_k)$ refers to a case in which label ℓ_j is canceled by label ℓ_k, $notGreater(\ell_j, \ell_k) = (legit(\ell_j) \wedge legit(\ell_k) \wedge \ell_k.ml \not\preceq_{lb} \ell_j.ml)$ that refers to a case in which label ℓ_k is not $\preceq_{lb}$-greater than ℓ_j and $(\ell_j =_i \ell_k) \equiv (\ell_j =_{lCreator} \ell_k =_{lCreator} i)$.

These two label pairs, ℓ_j and ℓ_k, can be the ones that processors p_j and p_k name as their local maximal label, as in $max_{i,j,k} = \{(max_j[j], max_k[k])\}$, or recently received from one another, as in $ack_{i,j,k} = \{(max_j[j], max_k[j])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j, as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $stored_{i,j,k} = \{\{max_j[j]\} \times storedLabels_k[i]\}$ We define the union of these cases to be the set $risk = \{(\ell_j, \ell_k) \in max_{i,j,k} \cup ack_{i,j,k} \cup hName_{i,j,k} \cup hAck_{i,j,k} \cup stored_{i,j,k} : \exists p_i, p_j, p_k \in P \wedge stopped_j \wedge stopped_k \wedge risk_{i,j,k}(\ell_j, \ell_k)\}$, where $stopped_i = true$ when processor p_i is inactive (crashed) and $false$ otherwise. The above notation can also be found in Table 1.

Lemma 3.5 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R during a period where p_j never adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$). Then eventually $risk = \emptyset$.*

Proof. Suppose this Lemma is false, i.e., the assumptions of this Lemma hold and yet in any configuration $c \in R$, it holds that $(\ell_j, \ell_k) \in risk \neq \emptyset$. We use $risk$'s definition to study the different cases. By the definition of $risk$, we can assume, without the loss of generality, that p_j and p_k are alive throughout R.

Claim: If p_j and p_k are alive throughout R, i.e. $stopped_j = stopped_k = \texttt{False}$, then $risk \neq \emptyset \iff risk_{i,j,k} = \texttt{True}$. This means that there exist two label pairs (ℓ_j, ℓ_k) where ℓ_k can force a cancellation to occur. Then the only way for this two labels to force $risk \neq \emptyset$ is if, throughout the execution, ℓ_k never reaches p_j.

The above claim is verified by a simple observation of the algorithm. If ℓ_k reaches p_j then lines 20, 24 and 26 guarantee a canceling and lines 22 and 23 ensure that these labels are kept canceled inside $storedLabels_j[]$. The latter is also ensured by the bounds on the labels given in Lemmas 3.3 and 3.4 that do not allow queue overflows. Thus to include these two labels to $risk$, is to keep ℓ_k hidden from p_j throughout R. We perform a case-by-case analysis to show that it is impossible for label ℓ_k to be "hidden" from p_j for an infinite number of steps in R.

The case of $(\ell_j, \ell_k) \in hName_{i,j,k}$**.** This is the case where $\ell_j = max_j[j]$ and ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $max_k[k]$. This may also contain such labels from the corrupt state. We note that p_j and p_k are alive throughout R. The stabilizing implementation of the data-link ensures that a message cannot reside in the communication channel during an infinite number of $transmit() - receive()$ events of the two ends. Thus ℓ_k, which may well have only a single instance in the link coming from the initial corrupt state, will either eventually reach p_j or it become lost. In the both cases (the first by the Claim for the second trivially) the two clashing labels are removed from $risk$ and the result follows.

The case of $(\ell_j, \ell_k) \in hAck_{i,j,k}$**.** This is the case where $\ell_j = max_j[j]$ and ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $max_k[j]$. The proof line is exactly the same as the previous case.

This case follows by the same arguments to the case of $(\ell_j, \ell_k) \in ack_{i,j,k}$.

The case of $(\ell_j, \ell_k) \in max_{i,j,k}$**.** Here the label pairs ℓ_j and ℓ_k are named by p_j and p_k as their local maximal label. We note that p_j and p_k are alive throughout R. By our self-stabilizing data-links and by the assumption on the communication that a message sent infinitely often is received infinitely often, then p_k transmits its $max_k[k]$ label infinitely often when executing line 16. This implies that p_j receives ℓ_k infinitely often. By the Claim the canceling takes place, and the two labels are eventually removed from the global observer's $risk$ set, giving a contradiction.

The case of $(\ell_\mathbf{j}, \ell_\mathbf{k}) \in \mathbf{ack_{i,j,k}}$. This is the case case where the labels (ℓ_j, ℓ_k) belong to $\{(max_j[j], max_k[j])\}$. Since processor p_k continuously transmits its label pair in $max_k[j]$ (line 16) the proof is almost identical to the previous case.

The case of $(\ell_\mathbf{j}, \ell_\mathbf{k}) \in \mathbf{stored_{i,j,k}}$. This case's proof, follows by similar arguments to the case of $(\ell_j, \ell_k) \in max_{i,j,k}$. Namely, p_k eventually receives the label pair $\ell_j = max_j[j]$. The assumption that $risk_{i,j,k}(\ell_j, \ell_k)$ holds implies that one of the tests in lines 23 and 26 will either update $storedLabels_k[i]$, and respectively, $max_k[j]$ with canceling values. We note that for the latter case we argue that p_j eventually received the canceled label pair in $max_k[j]$, because we assume that p_j does not change the value of $max_j[j]$ throughout R.

By careful and exhaustive examination of all the cases, we have proved that there is no way to to keep ℓ_k hidden from p_j throughout R. This is a contradiction to our initial assumption, and thus eventually $risk = \emptyset$. ■

These two label pairs, ℓ_j and ℓ_k, can be the ones that processors p_j and p_k name as their local maximal label, as in $max_{i,j,k} = \{(max_j[j], max_k[k])\}$, or recently received from one another, as in $ack_{i,j,k} = \{(max_j[j], max_k[j])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j, as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $stored_{i,j,k} = \{\{max_j[j]\} \times storedLabels_k[i]\}$.

Lemma 3.6 *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Then p_j* never *adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$).*

Proof. Note that the definition of $risk$ considers almost every possible combination of two label pairs ℓ_j and ℓ_k from p_i's domain that are stored by processor p_j, and respectively, p_k (or in the channels to them). The only combination that is not considered is $(\ell_j, \ell_k) \in storedLabels_j[i] \times storedLabels_k[i]$. However, this combination can indeed reside in the system during a legal execution and it cannot lead to a disruption for the case of $risk = \emptyset$ in every configuration throughout R because before that could happen, either p_j or p_k would have to adopt ℓ_j, and respectively, ℓ_k, which means a contradiction with the assumption that $risk = \emptyset$.

The only way that a label in $storedLabels[]$ can cause a change of the local maximum label and be communicated to also disrupt the system, is to find its way to $max[]$. Note that p_j cannot create a label under p_i's domain (line 28) since the algorithm does not allow this, nor can it adopt a label from $storedLabels_j[i]$ (by the definition of $legitLabels()$, line 14). So there is no way for ℓ_j to be added to $max_j[j]$ and thus make $risk \neq \emptyset$ through creation or adoption.

On the other hand, we note that there is only one case where p_k extracts a label from $storedLabels_k[i] : i \neq k$ and adds it to $max_k[j]$. This is when it finds a legit label $\ell_j \in max_k[j]$ that can be canceled by some other label ℓ_k in $storedLabels_k[i]$), line 26. But this is the case of having the label pair (ℓ_j, ℓ_k) in $stored_{i,j,k}$. Our assumption that $risk = \emptyset$ implies that $stored_{i,j,k} = \emptyset$. This is a contradiction. Thus a label ℓ_k cannot reach $max_k[]$ in order for it to be communicated to p_j.

In the same way we can argue for the case of two messages in transit, $\mathcal{H}_{j,k} \times \mathcal{H}_{k,j}$ and that $risk = \emptyset$ throughout R. ■

Lemma 3.7 *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. There is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$.*

Proof. We initially note that the two processors p_i, p_j that take an infinite number of steps in R will exchange their local maximal label $max_i[i]$ and $max_j[j]$ an infinite number of times. By the assumption that $risk = \emptyset$, there are no two label pairs in the system that can cause canceling to each other that are unknown to p_i or p_j and are still part of $max_i[i]$ or $max_i[j]$. Hence, any differences in the local maximal label of the processors must be due to the labels' $lCreator$ difference.

Since $max_i[i]$ and $max_j[j]$ are continuously exchanged and received, assuming $max_i[i].ml \prec_{lb} max_j[j].ml$ where the labels are of different label creators, then p_i will be led to a $receive()$ event of $\langle sentMax_j, lastSent_j \rangle$ where $max_i[i].ml \prec_{lb} sentMax_j.ml$. By line 19, $sentMax_j$ is added to $max_i[j]$ and since $risk = \emptyset$ no action from line 20 to line 26 takes place. Line 27 will then indicate that the greatest label in $max_i[\bullet]$ is that in $max_i[j]$ which is then adopted by p_i as $max_i[i]$, i.e., p_i's local maximal. The above is true for every pair of processors taking an infinite number of steps in R and so we reach to the conclusion that eventually all such processors converge to the same ℓ_{max} label, i.e., it holds that $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$. ■

3.2.5 Convergence

Theorem 3.3 combines all the previous lemmas to demonstrate that when starting from an arbitrary starting configuration, the system eventually reaches a configuration in which there is a global maximal label.

Theorem 3.3 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Within a bounded number of steps, there is a legitimate label pair $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that p_i has $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$.*

Proof. For any processor in the system, which may take any (bounded or practically infinite) number of steps in R, we know that there is a bounded number of label pairs, $L_i = \ell_{i_0}, \ell_{i_1}, \ldots$, that processor $p_i \in P$ adds to the system configuration (the else part of line 28), where $\ell_{i_k} =_{lCreator} i$ (Lemma 3.4). Thus, by the pigeonhole principle we know that, within a bounded number of steps in R, there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst (all processors) p_i do not add any label pair, $\ell_{i_k} =_{lCreator} i$, to the system configuration (the *else* part of line 28).

During this practically infinite period (with respect to $p_{unknown}$), in which no label pairs are added to the system configuration due to the *else* part of line 28, we know that for any processor $p_j \in P$ that

Variables: A label *lbl* is extended to the triple $\langle lbl, seqn, wid \rangle$ called a *counter* where *seqn*, is the sequence number related to *lbl*, and *wid* is the identifier of the creator of this *seqn*. A counter pair $\langle mct, cct \rangle$ extends a label pair. *cct* is a canceling counter for *mct*, such that $cct.lbl \not\prec_{lb} mct.lbl$ or $cct.lbl = \bot$. We rename structures $max[]$ and $storedLabels[]$ of Alg. 2 to $maxC[]$ and $storedCnts[]$ that hold counter pairs instead of label pairs. Variable $status \in \{\mathsf{MAX\_REQUEST}, \mathsf{MAX\_WRITE}, \mathsf{COMPLETE}\}$.
Operators: $add(ctp)$ - places a counter pair *ctp* at the front of a queue. If *ctp.mct.lbl* already exists in the queue, it only maintains the instance with the greatest counter w.r.t. $\prec_{ct}$, placing it at the front of the queue. If one counter pair is canceled then the canceled copy is retained. We consider an array field as a single sized queue and use $add()$.

Figure 2: Variables and Operators for Counter Increment; code for p_i.

takes any number of (bounded or practically infinite) steps in R, and processor $p_k \in P$ that adopts labels in R (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin storedLabels_k(j)$) it holds that p_k adopts such labels (line 27) only a bounded number times in R (Lemma 3.3). Therefore, we can again follow the pigeonhole principle and say that there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst neither p_i adds a label, $\ell_{i_k} =_{lCreator} i$, to the system (the else part of line 28), nor p_k adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_k(\ell_j)$).

We deduce that, when the above is true, then we have reached a configuration in R where $risk = \emptyset$ (Lemma 3.5) and remains so throughout R (Lemma 3.6). Lemma 3.7 concludes by proving that, whilst $p_{unknown}$ takes a practically infinite number of steps, all processors (that take practically infinite number of steps in R) name the same $\preceq_{lb}$-greatest legitimate label pair which the theorem statement specifies. Thus no label $\ell =_{lCreator} j$ in $max_i[\bullet]$ or in $storedLabels_i[j]$ may satisfy $\ell.ml \not\preceq_{lb} \ell_{max}.ml$. ■

3.2.6 Algorithm complexity

The required local memory of a processor comprises of a queue of size (in labels) $2(n^3cap + 2n^2 - 2n)$ that hosts the labels with the processor as a creator (Corollary 3.2). The local state also includes $n - 1$ queues of size $n + n^2cap$ to store labels by other processors, and a single label for the maximal label of every processor. We conclude that the *space complexity* is of order $O(n^3)$ in labels. Given the number of possible labels in the system by the same processor is $\beta = n^3cap + 2n^2 - 2n$, as shown in the proof of Lemma 3.4, we deduce that the size of a label in bits is $O(\beta \log \beta)$.

By Theorem 3.3 we can bound the *stabilization time* based on the number of label creations. Namely, in an execution with $O(n \cdot \beta)$ label creations (e.g., up to n processors can create $O(\beta)$ labels), there is a practically infinite execution suffix (of size 2^τ iterations) where the receipt of a label which starts an iteration never changes the maximal label of any processor in the system.

3.3 Increment Counter Algorithm

We adjust the labeling algorithm to work with counters, so that our counter increment algorithm is a stand-alone algorithm. In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection to obtain a practically (infinite) self-stabilizing counter increment algorithm.

3.3.1 From labels to counters and to a counter version of Algorithm 2

Counters. To achieve this task, we now need to work with practically unbounded *counters*. A counter *cnt* is a triplet $\langle lbl, seqn, wid \rangle$, where *lbl* is an epoch label as defined in the previous subsection, *seqn* is a τ-bit integer sequence number and *wid* is the identifier of the processor that last incremented the counter's sequence number, i.e., *wid* is the counter *writer*. Then, given two counters cnt_i, cnt_j we define the relation $cnt_i \prec_{ct} cnt_j \equiv (cnt_i.lbl \prec_{lb} cnt_j.lbl) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn < cnt_j.seqn)) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn = cnt_j.seqn) \wedge (cnt_i.wid < cnt_j.wid))$. Observe that when the labels of the two counters are incomparable, the counters are also incomparable.

```
// Where macros coincide with Algorithm 2 we do not restate them.
1  Macros:
2  $exhausted(ctp) = (ctp.mct.seqn \geq 2^{\tau})$
3  $cancelExh(ctp) : ctp.cct \leftarrow ctp.mct$
4  $cancelExhMaxC()$ : foreach $p_j \in P,\ c \in maxC[j] : exhausted(c)$ do $cancelExh(maxC[j])$;
5  $legit(ctp) = (ctp.cct = \bot\rangle)$
6  $staleCntrInfo() = staleInfo() \vee (\exists p_j \in P, x \in storedCnts[j] : exhausted(x) \wedge legit(x))$
7  $retCntrQ(ct)$ : return $(storedCnts[ct.lbl.lCreator])$
8  $retMaxCnt(ct) =$ return $(max_{\prec_{ct}}(ct, ct'))$ where $ct' \in retCntrQ(ct) \wedge (ct =_{lbl} ct')$
9  $legitCnts() = \{maxC[j].mct : \exists p_j \in P \wedge legit(maxC[j])\}$
10 $useOwnCntr() =$ if $(\exists cp \in storedCnts[i] : legit(lp))$ then $maxC[i] \leftarrow cp$ else $storedCnts[i].add(maxC[i] \leftarrow$
       $\langle\langle nextLabel(), 0, i\rangle, \bot\rangle)$ // For every $cp \in storedCnts[i]$, we pass to $nextLabel()$ both $cp.mct.lbl$ and
       $cp.cct.lbl$.
11 $getMaxSeq()$ : return $max_{wid}(\{max_{seqn}(\{ctp : ctp.mct \in legitCnts() \wedge maxC[i] =_{mct.lbl} ctp\})\})$
12 $initWrite = \{\langle maxC[i], responseSet, status\rangle \leftarrow \langle maxC[i](), \emptyset, \mathsf{MAX\_WRITE}\rangle;\}$
13 $increment() = \{maxC[i] \leftarrow \langle maxC[i].mct.lbl, maxC[i].mct.seqn + 1, i\rangle;\}$
14 $correctResponse(A, B) =$ return $((status = \mathsf{MAX\_REQUEST} \wedge (A, B \notin \{\bot\})) \vee ((status = \mathsf{MAX\_WRITE}) \wedge$
       $(\langle A, B\rangle = \langle \bot, maxC_i[i]\rangle)))$
```

Figure 3: Macros for Algorithm 3.

The relation $\prec_{ct}$ defines a total order (as required by practically unbounded counters) for counters with the same label, thus, only when processors share a globally maximal label. Conceptually, if the system stabilizes to use a global maximal label, then the pair of the sequence number and the processor identifier (of this sequence number) can be used as an unbounded counter, as used, for example, in MWMR register implementations [18, 21].

Structures. We convert the label structures $max[]$ and $storedLabels[]$ of the labeling algorithm into the structures $maxC[]$ and $storedCnts[]$ that hold counters rather than labels (see Figure 2). Each label can yield many different counters with different $\langle seqn, wid\rangle$. Therefore, in order to avoid increasing the size of the queues of $storedCnts$ (with respect to the number of elements stored), we only keep the highest sequence number observed for each label (breaking ties with the wid).

This is encapsulated in the definition of the $add()$ operator (Figure 2 – Operators). In particular, we define the operator $add(ctp)$ (Fig. 2) to enqueue a counter pair ctp to a queue of $storedCnts[n]$, where in case a counter with the same label already exists, the following two rules apply: (1) if at least one of the two counters is canceled we keep a canceled instance, and (2) if both counters are legitimate, we keep the greatest counter with respect to $\langle seqn, wid\rangle$. The counter is placed at the front of the queue. In this way we allow for labels for which the counters have not been exhausted to be reused. We denote a counter pair by $\langle mct, cct\rangle$, with this being the extension of a label pair $\langle ml, cl\rangle$, where cct is a canceling counter for mct, such that either $cct.lbl \not\prec_{lb} mct.lbl$ (i.e., the counter is canceled), or $cct.lbl = \bot$.

Exhausted counters. These are the ones satisfying $seqn \geq 2^{\tau}$, and they are treated in a way similar to the canceled labels in the labeling algorithm; an exhausted counter mct in a counter pair $\langle mct, cct\rangle$ is canceled, by setting $mct.lbl = cct.lbl$ (i.e., the counter's own label cancels it) and hence cannot be used as a local maximal counter in $maxC_i[i]$. This cannot increase the number of labels that are created, since the initial set of corrupt counters remains the same as the one for labels, for which we have already produced a proof in Section 3.1.

The enhanced labeling algorithm. Figure 4 presents a standalone version of the labeling algorithm adjusted for counters. Each processor p_i uses the token-based communication to transmit to every other processor p_j its own maximal counter and the one it currently holds for p_j in $maxC_i[j]$ (line 1). Upon receipt of such an update from p_j, p_i first performs canceling of any exhausted counters in $storedCnts[]$ (line 4), in $maxC[]$ (line 6) and in the received couple of counter pairs (line 5). Having catered for exhaustion, it then calls $maintainCntrs(\langle\bullet, \bullet\rangle)$ with the received two counter pairs as arguments. This is essentially a counter version of Algorithm 2. Macros that require some minor adjustments to handle counters are seen in Figure 3 lines 5 to 10. We also address the need to update counters of $maxC[]$ w.r.t. $seqn$ and wid based on counters from the $storedCnts[]$ structure and vice versa in lines 19 and 10.

```
   // Lines 1 and 2 run in the background.
1  upon $transmitReady(p_j \in P \setminus \{p_i\})$ do transmit$(\langle maxC[i], maxC[j] \rangle)$;
2  upon $receive(\langle sentMax, lastSent \rangle)$ from $p_k$ do $processCntr(sentMax, lastSent, j)$
3  procedure $processCntr($counter pair $sentMax$, counter pair $lastSent$, int $k)$ begin
4      foreach $p_j \in P, ctp \in storedCnts[j] : legit(ctp) \wedge exhausted(ctp)$ do $cancelExh(ctp)$;
5      if $(\exists ctp' \in \langle sentMax, lastSent \rangle : exhausted(ctp'))$ then $cancelExh(ctp')$;
6      $cancelExhMaxC()$; $maintainCntrs(sentMax, lastSent)$;
7  operator $maintainCntrs($counter pair $sentMax$, counter pair $lastSent$, int $k)$
8  begin
9      if $sentMax \neq$ NULL then $maxC[k] \leftarrow sentMax$;
10     if $lastSent \neq$ NULL $\wedge \neg legit(lastSent) \wedge maxC[i] =_{mct.lbl} lastSent$ then $maxC[i].add(lastSent)$;
11     if $staleCntrInfo()$ then $storedCnts.emptyAllQueues()$;
12     foreach $p_j \in P : recordDoesntExist(j)$ do $retCntrQ(maxC[j]).add(maxC[j])$;
13     foreach $p_j \in P, cp \in storedCnts[j] : (legit(cp) \wedge (notgeq(j, cp) \neq \perp))$ do
14         $cp.cct \leftarrow notgeq(j, cp)$
15     foreach $p_j \in P : ((\neg legit(maxC[j]) \vee (cp <_{mct.seqn} maxC[j])) \wedge (maxC[j] =_{ml} cp) \wedge legit(cp)$ where
       $cp \in retCntrQ(maxC[j])$ do $cp \leftarrow maxC[j]$;
16     foreach $p_j \in P, cp \in storedCnts[j] : double(j, cp)$ do $cp.remove()$;
17     foreach $p_j \in P : (legit(maxC[j]) \wedge (canceled(maxC[j]) \neq \langle \perp, \perp \rangle))$ do
18         $maxC[j] \leftarrow canceled(maxC[j])$
19     foreach $p_j \in P, cp \in)$ do $maxC[j] \leftarrow getMaxCnt(maxC[j])$;
20     if $legitCnts() \neq \emptyset$ then $maxC[i] \leftarrow \langle \max_{\prec_{ct}}(legitCnts()), \perp \rangle$;
21     else $useOwnCntr()$;
```

Figure 4: The $maintainCntrs()$ operator (code for p_i).

We define the operator $add(ctp)$ (Fig. 2) to enqueue a counter pair ctp to a queue of $storedCnts[n]$, where in case a counter with the same label already exists the following two rules apply: (1) if at least one of the two counters is cancelled we keep a canceled instance, and (2) if both counters are legitimate, we keep the greatest counter with respect to $\langle seqn, wid \rangle$. The counter is placed at the front of the queue.

3.3.2 Counter Increment Algorithm

Algorithm 3 shows a self-stabilizing counter increment algorithm where multiple processors can increment the counter. We start with some useful definitions and proceed to describe the algorithm.

Quorums We define a *quorum set* $\mathbb{Q}$ based on processors in P, as a set of processor subsets of P (named *quorums*), that ensure a non-empty intersection of every pair of quorums. Namely, for all quorum pairs $Q_i, Q_j \in \mathbb{Q}$ such that $Q_i, Q_j \subset P$, it must hold that $Q_i \cap Q_j \neq \emptyset$. This *intersection property* is useful to propagate information among servers and exploiting the common intersection without having to write a value v to all the servers in a system, but only to a single quorum, say Q. If one wants to retrieve this value, then a call to *any* of the quorums (not necessarily Q), is expected to return v because there is least one processor in every quorum that also belongs to Q. In the counter algorithm we exploit the intersection property to retrieve the currently greatest counter in the system, increment it, and write it back to the system, i.e., to a quorum therein. Note that majorities form a special case of a quorum system.

Algorithm description To increment the counter, a processor p_i enters status MAX_REQUEST (line 2) and starts sending a request to all other processors, waiting for their maximal counter (via line 8). Processors receiving this request respond with their current maximal counter and the last sent by p_i (line 12). When such a response is received (line 15), p_i adds this to the local counter structures via the counter bookkeeping algorithm of Figure 4. Once a quorum of responses (line 4) have been processed, $maxC[i]$ holds the maximal counter that has come to the knowledge of p_i about the system's maximal

Algorithm 3: Increment Counter; code for p_i

```
1  interface function $incrementCounter()$ begin
2      let $\langle responseSet, status\rangle \leftarrow \langle\emptyset, \mathsf{MAX\_REQUEST}\rangle$;
3      repeat
4          if $status = \mathsf{MAX\_REQUEST} \wedge (\exists Q \in \mathbb{Q} : Q \subseteq \{responseSet\})$ then
5              $initWrite()$; $increment()$
6          else if $status = \mathsf{MAX\_WRITE} \wedge (\exists Q \in \mathbb{Q} : Q \subseteq \{responseSet\})$ then
7              $\langle status \leftarrow \mathsf{COMPLETE}\rangle$
8          foreach $p_j \in P$ do send $\langle status, maxC[i], maxC[j]\rangle$;
9      until $status = \mathsf{COMPLETE}$;
10     return $maxC[i]$
11 upon receive of $m = \langle subj, sentMax, lastSent\rangle$ from $p_j$ begin
12     if $(m.subj = \mathsf{MAX\_REQUEST})$ then send $\langle \mathsf{ACK}, maxC_i[i], maxC_i[j]\rangle$ to $p_j$;
13     else if $(m.subj = \mathsf{MAX\_WRITE})$ then
14         $processCntr(sentMax, lastSent, j)$; send $\langle \mathsf{ACK}, \perp, \mathsf{lastSent}\rangle$ to $p_j$;
15     else if $(m.subj = \mathsf{ACK} \wedge correctResponse(sentMax, lastSent))$ then
16         $processCntr(sentMax, lastSent, j)$; $responseSet \leftarrow j$
```

counter. This counter is then incremented locally and p_i enters status $\mathsf{MAX\_WRITE}$ by initiating the propagation of the incremented counter (line 5), and waiting to gather acknowledgments from a quorum (the condition of line 6). When the latter condition is satisfied, the function returns the new counter. This is, in spirit, similar to the two-phase write operation of MWMR register implementations, focusing on the sequence number rather than on an associated value.

3.3.3 Proof of correctness

Proof outline Initially we prove, by extending the proof of the labeling algorithm, that starting from an arbitrary configuration the system eventually reaches to a global maximal label (as given in Theorem 3.3), even in the presence of exhausted counters (Lemma 3.4). By using the intersection property of quorums we establish that a counter that was written is known by at least one processor in every quorum (Lemma 3.5. We then combine the two previous lemmas to prove that counters increment monotonically.

Lemma 3.4 *In a bounded number of steps of Algorithm 4 every processor p_i has counter $maxC_i[i] = ct$ with $ct.lbl = \ell_{max}$ the globally maximal non-exhausted label.*

Proof. For this lemma we refer to the enhanced labeling algorithm for counters (Figure 4). The lemma proof can be mapped on the arguments proving lemmas Lemma 3.1 to Lemma 3.4 of Algorithm 2. Specifically, consider a processor p_i that has performed a full execution of $processCntr()$ (Fig. 4 line 3) at least once due to a receive event. This implies a call to $maintainCntrs$ and thus to $staleCntrInfo()$ (Fig. 4 line 11) which will empty all queues if exhausted non-canceled counters exist. Also there is a call to $cancelExhaustedMaxC()$ which cancels all counters that are exhausted in $maxC[]$. By observation of the code, after a single iteration, there is no local exhausted counter that is not canceled.

Since every counter that is received and is exhausted becomes canceled, and since the arbitrary counters in transit are bounded, we know that there is no differentiation between exhausted labels that may cause a counter's label to be canceled. Namely, the size of the queues of $storedCnts[]$ remain the same while at the same time provide the guarantees provided by the proof of the labeling algorithm. It follows from the labeling algorithm correctness and by our cancellation policy on the exhausted counters, that Theorem 3.3 can be extended to also include the use of counters without any need to locally keep more counters than there are labels.

We proceed to deduce that, eventually, any processor taking practically infinite number of steps in R obtains a counter with globally maximal label ℓ_{max}.

■

For the rest of the proof we refer to line numbers in Algorithm 3.

Lemma 3.5 *In an execution where Lemma 3.4 holds, it also holds that* $\forall Q \in \mathbb{Q}, \exists p_j \in Q : maxC_j[j] = ct \wedge (ct' \prec_{ct} ct)$, *where* $ct' \in \{storedCnts_k[k'] \cup maxC_k[k'] : ct' =_{lbl} ct\}_{p_k, p_{k'} \in P} \setminus \{ct\}$, *i.e.,* ct' *is every counter in the system with identical label but less than ct w.r.t. seqn or wid and ct is the last counter increment.*

Proof. Observe that upon a quorum write, the new incremented counter *ct* with the maximal label *lb* is propagated (lines 6 and 8) until a quorum of acknowledgments have been received. Upon receiving such a counter by p_i, a processor p_j will first add *ct* to its structures via *processCntr*() and will then acknowledge the write. If this is the maximal counter that it has received (there could be concurrent ones) then the call to *processCntr*() will also have the following effects: (i) the counter's *seqn* and *wid* will be updated in the $storedCnts_j[]$ structure in the queue of the creator of *lb*, (ii) $maxC_j[j] \leftarrow ct$.

Since p_i waits for responses by a quorum before it returns, it follows that by the intersection property of the quorums, the lemma must hold when p_i reaches status COMPLETE. ■

Theorem 3.6 *Given an execution R of the counter increment algorithm in which at least a majority of processors take a practically infinite number of steps, the algorithm ensures that counters eventually increment monotonically.*

Proof. Consider a configuration $c \in R'$ where R' is a suffix of R in which Lemma 3.4 holds, and in which ct_{max} is the counter which is maximal with respect to $\prec_{ct}$. There are two cases that the counter may be incremented.

In the first case, a legal execution, the counter ct_{max} is only incremented by a call to *incrementCounter*(), By Lemma 3.5 any call to *incrementCounter*() will return the last written maximal counter (namely ct_{max}). When this is incremented giving ct'_{max} then $ct'_{max}.seqn = ct_{max}.seqn + 1$ which is monotonically greater than ct'_{max} and in case of concurrent writes the *wid* is unique and can break symmetry enforcing the monotonicity.

The second case arises when ct_{max} comes from the arbitrary initial state, is not known by a quorum, and resides in either a local state or is in transit. When ct_{max} eventually reaches a processor, it becomes the local maximal and it is propagated either via counter maintenance or in the first stage of a counter increment when the maximal counters are requested by the writer. In this case the use of ct_{max} is also a monotonic increment, and from this point onwards any increment in R' proceeds monotonically from ct_{max}, as described in the previous paragraph. ■

3.3.4 Algorithm Complexity

The local memory of a processor implementing the counter increment is not different in order to the labeling algorithm's, since converting to the counter structures only adds an integer (the sequence number). Hence the *space complexity* of the algorithm is $O(n^3)$ in counters. The upper bound on *stabilization time* in the number of counter increments that are required to reach a period of practically infinite counter increments can be deduced by Theorem 3.6. For some t such that $0 \leq t \leq 2^\tau$ in an execution with $O(n \cdot \beta \cdot t)$ counter increments (recall that $\beta = n^3 cap + 2n^2 - 2n$), there is a practically infinite period of (2^τ) monotonically increasing counter increments in which the label does not change.

3.3.5 MWMR Register Emulation

Having a practically-self-stabilizing counter increment algorithm, it is not hard to implement a *practically-self-stabilizing MWMR register emulation.* Each counter is associated with a value and the counter increment procedure essentially becomes a write operation: once the maximal counter is found, it is increased and associated with the new value to be written, which is then communicated to a majority of processors. The read operation is similar: a processor first queries all processors about the maximum counter they are aware of. It collects responses from a majority and if there is no maximal counter, it returns $\perp$ so the processor needs to attempt to read again (i.e., the system hasn't converged to a maximal label yet). If a maximal counter exists, it sends this together with the associated value to all the processors, and once it collects a majority of responses, it returns the counter with the associated value (the second phase is a required to preserve the consistency of the register (c.f. [3, 18]).

4 Virtually Synchronous Stabilizing Replicated State Machine

Group communication systems (GCSs) that guarantee the virtual synchrony property, essentially suggest that processes that remain together in consecutive groups (called *views*) will deliver the same messages in the desired order [5]. This is particularly suited to maintain a replicated state machine service, where replicas need to remain consistent, by applying the same changes suggested by the environment's requests. A key advantage of multicast services (with virtual synchrony) is the ability to reuse the same view during many multicast rounds, and which allows every automaton step to require just a single multicast round, as compared to other more expensive solutions.

GCSs provide the VS property by implementing two main services: a reliable multicast service, and a membership service to provide the membership set of the view, whilst they also assume access to unbounded counters to use as unique view identifiers. We combine existing self-stabilizing versions of the two services (with adaptations where needed), and we use the counter from the previous section to build the first (to our knowledge) practically self-stabilizing virtually synchronous state machine replication. While the ideas appear simple, combining the services is not always intuitive, so we first proceed to a high-level description of the algorithm and the services, and then follow the algorithm with a more technical description and the correctness proof.

4.1 Preliminaries

The algorithm progresses in state replication by performing multicast rounds, when a *view*, a tuple composed of a members *set* taken from P, and of a unique identifier (ID) that is a counter as defined in the previous section, is installed. This view must include a *primary component* (defined formally in Definition 4.1), namely it must contain a majority of the processors in P, i.e., $n/2+1$. In our version, a processor, the *coordinator*, is responsible: (1) to progress the multicast service which we detail later, (2) to change the view when its failure detector suggests changes to the composition of the view membership. Therefore, the output of the coordinator's failure detector defines the set of view members; this helps to maintain a consistent membership among the group members, despite inaccuracies between the various failure detectors.

On the other hand, the counter increment algorithm that runs in the background allows the coordinator to draw a counter for use as a view identifier and in this case, the counter's writer identifier (wid) is that of the view's coordinator. This defines a simple interface with the counter algorithm, which provides an identical output. Pairing the coordinator's member set with a counter as view identity we obtain a *view*. Of course as we will describe later, reaching to a unique coordinator may require issuing several such view proposals, of which one will prevail. We first suggest a possible implementation of a failure detector (to provide membership) and of a reliable multicast service over the self-stabilizing FIFO data link given in Section 2, and then proceed to an algorithm overview.

Definition 4.1 *We say that the output of the (local) failure detectors in execution R includes a* primary partition *when it includes a supporting majority of processors $P_{maj} : P_{maj} \subseteq P$, that (mutually) never suspect at least one processor, i.e., $\exists p_\ell \in P$ for which $|P_{maj}| > \lfloor n/2 \rfloor$ and $(p_i \in (P_{maj} \cap FD_\ell)) \iff (p_\ell \in (P_{maj} \cap FD_i))$ in every $c \in R$, where FD_x returns the set of processors that according to p_x's failure detector are active.*

Failure detector We employ the self-stabilizing failure detector of [7] which is implemented as follows. Every processor p uses the token-based mechanism to implement a heartbeat (see Section 2) with every other processor, and maintain a heartbeat integer counter for every other processor q in the system. Whenever processor p receives the token from processor q over their data link, processor p resets the counter's value to zero and increments all the integer counters associated with the other processors by one, up to a predefined threshold value W. Once the heartbeat counter value of a processor q reaches W, the failure detector of processor p considers q as inactive. In other words, the failure detector at processor p considers processor q to be active, if and only if the heartbeat associated with q is strictly less than W.

As an example, consider a processor p which holds an array of heartbeat counters for processors p_i, p_j, p_k such that their corresponding values are $\langle 2, 5, W-1 \rangle$. If p_j sends its heartbeat, then p's array will be changed to $\langle 3, 0, W \rangle$. In this case, p_k will be suspected as crashed, and the failure detector reading will return the set p_i, p_j as the set of processors considered correct by p.

Note that our virtual synchrony algorithm, employs the same implementation but has weaker requirements than [7] that solve consensus, and thus they resort to a failure detector at least as strong as Ω [10]. Specifically, in Definition 4.1 we pose the assumption that *just a majority of the processors* do not suspect at least one processor of P for sufficiently long time, in order to be able to obtain a long-lived coordinator. This is different, as we said before, to an eventually perfect failure detector that ensures that after a certain time, no active processor suspects any other active processor.

Our requirements, on the other hand, are stronger than the weakest failure detector required to implement atomic registers (when more than a majority of failures are assumed), namely the Σ failure detector [11], since virtual synchrony is a more difficult task. In particular, whilst the Σ failure detector eventually outputs a set of only correct processors to correct processors, we require that this set in at least half of the processors, will contain at least one common processor. In this perspective our failure detector seems to implement a *self-stabilizing* version of a slightly stronger failure detector than Σ. It would certainly be of interest for someone to study what is the weakest failure detector required to achieve practically-self-stabilizing virtually synchronous state replication, and whether this coincides with our suggestion.

Reliable multicast implementation The coordinator of the view controls the exchange of messages (by multicasting) within the view. The coordinator requests, collects and combines input from the group members, and then it multicasts the updated information. Specifically, when the coordinator decides to collect inputs, it waits for the token (see Section 2) to arrive from each group participant. Whenever a token arrives from a participant, the coordinator uses the token to send the request for input to that participant, and waits the token to return with some input (possibly $\perp$, when the participant does not have a new input). Once the coordinator receives an input from a certain participant with respect to this multicast invocation, the corresponding token will not carry any new requests to receive input from the same participant; of course, the tokens continue to move back and forth to sustain the heartbeat-based failure detector. When all inputs are received, the processor combines them and again uses the token to carry the updated information. Once this is done, the coordinator can proceed to the next input collection.

We provide the pseudocode for the practically-stabilizing replicate state machine implementation as Algorithm 4, a high level description, and proceed to a line-by-line description and correctness analysis.

4.2 Virtual Synchrony Algorithm

4.2.1 High-level algorithm description

Each participant maintains a replica of the state machine and the last processed (composite) message. Note that we bound the memory used to store the history of the replicated state machine by deciding to have the (encapsulated influence of the history represented by the) current state of the replicated state machine. In addition, each participant maintains the last delivered (composite) message to ensure common reliable multicast, as the coordinator may stop being active prior to ensuring that all members received a copy of the last multicast message. Whenever a new coordinator is installed, the coordinator

Algorithm 4: A practically-self-stabilizing automaton replication using virtual synchrony, code for processor p_i

```
1  Constants: $PCE$ (periodic consistency enforcement) number of rounds between global state check;
2  Interfaces: $fetch()$ next multicast message, $apply(state, msg)$ applies the step $msg$ to $state$ (while producing side effects),
   $synchState(replica)$ returns a replica consolidated state, $synchMsgs(replica)$ returns a consolidated array of last
   delivered messages, $failureDetector()$ returns a vector of processor pairs $\langle pid, crdID\rangle$, $inc()$ returns a counter from the
   increment counter algorithm;
3  Variables: $rep[n] = \langle view = \langle ID,\ set\rangle,\ status \in \{\mathsf{Multicast}, \mathsf{Propose}, \mathsf{Install}\},\ (multicast\ round\ number)\ rnd,\ (replica)$
   $state,\ (last\ delivered\ messages)\ msg[n]\ (to\ the\ state\ machine),\ (last\ fetched)\ input\ (to\ the\ state\ machine),\ propV =$
   $\langle ID,\ set\rangle,\ (no\ coordinator\ alive)\ noCrd,\ (recently\ live\ and\ connected\ component)\ FD\rangle$ : an array of state replica of the
   state machine, where $rep[i]$ refers to the one that processor $p_i$ maintains. A local variable $FDin$ stores the
   $failureDetector()$ output. $FD$ is an alias for $\{FDin.pid\}$, i.e. the set of processors that the failure detector considers as
   active. Let $crd(j) = \{FDin.crdID : FDin.pid = j\}$, i.e. the id of $p_j$'s local coordinator, or $\perp$ if none.
4  Do forever begin
5      let $FDin = failureDetector()$;
6      let $seemCrd = \{p_\ell = rep[\ell].propV.ID.wid \in FD : (|rep[\ell].propV.set| > \lfloor n/2\rfloor) \wedge (|rep[\ell].FD| > \lfloor n/2\rfloor) \wedge (p_\ell \in$
       $rep[\ell].propV.set) \wedge (p_k \in rep[\ell].propV.set \leftrightarrow p_\ell \in rep[k].FD) \wedge ((rep[\ell].status = \mathsf{Multicast}) \rightarrow (rep[\ell].(view =$
       $propV) \wedge crd(\ell) = \ell)) \wedge ((rep[\ell].status = \mathsf{Install}) \rightarrow crd(\ell) = \ell)\}$;
7      let $valCrd = \{p_\ell \in seemCrd : (\forall p_k \in seemCrd : rep[k].propV.ID \preceq_{ct} rep[\ell].propV.ID)\}$;
8      $noCrd \leftarrow (|valCrd| \neq 1)$; $crdID \leftarrow valCrd$;
9      if $(|FD| > \lfloor n/2\rfloor) \wedge (((|valCrd| \neq 1) \wedge (|\{p_k \in FD : p_i \in rep[k].FD \wedge rep[k].noCrd\}| > \lfloor n/2\rfloor)) \vee ((valCrd =$
       $\{p_i\}) \wedge (FD \neq propV.set) \wedge (|\{p_k \in FD : rep[k].propV = propV\}| > \lfloor n/2\rfloor)))$ then $(status, propV) \leftarrow (\mathsf{Propose},$
       $\langle inc(),\ FD\rangle)$;
10     else if $(valCrd = \{p_i\}) \wedge (\forall\ p_j \in view.set : rep[j].(view,\ status,\ rnd) = (view,\ status,\ rnd)) \vee ((status \neq$
       $\mathsf{Multicast}) \wedge (\forall\ p_j \in propV.set : rep[j].(propV, status) = (propV, \mathsf{Propose}))$ then
11         if $status = \mathsf{Multicast}$ then
12             $apply(state, msg)$; $input \leftarrow fetch()$;
13             foreach $p_j \in P$ do if $p_j \in view.set$ then $msg[j] \leftarrow rep[j].input$ else $msg[j] \leftarrow \perp$;
14             $rnd \leftarrow rnd + 1$;
15         else if $status = \mathsf{Propose}$ then $(state, status, msg) \leftarrow (synchState(rep), \mathsf{Install}, synchMsgs(rep))$;
16         else if $status = \mathsf{Install}$ then $(view, status, rnd) \leftarrow (propV, \mathsf{Multicast}, 0)$;
17     else if $valCrd = \{p_\ell\} \wedge \ell \neq i \wedge ((rep[\ell].rnd = 0 \vee rnd < rep[\ell].rnd \vee rep[\ell].(view \neq propV))$ then
18         if $rep[\ell].status = \mathsf{Multicast}$ then
19             if $rep[\ell].state = \perp$ then $rep[\ell].state \leftarrow state$ /* PCE optimization, line 25 */;
20             $rep[i] \leftarrow rep[\ell]$; $apply(state, rep[\ell].msg)$; /* for the sake of side-effects */
21             $input \leftarrow fetch()$;
22         else if $rep[\ell].status = \mathsf{Install}$ then $rep[i] \leftarrow rep[\ell]$;
23         else if $rep[\ell].status = \mathsf{Propose}$ then $(status, propV) \leftarrow rep[\ell].(status, propV)$;
24     let $m = rep[i]$ /* sending messages: all to coordinator and coordinator to all */ ;
25     if $status = \mathsf{Multicast} \wedge rnd (\bmod\ PCE) \neq 0$ then $m.state \leftarrow \perp$ /* PCE optimization, line 19 */ ;
26     let $sendSet = (seemCrd \cup \{p_k \in propV.set : valCrd = \{p_i\}\} \cup \{p_k \in FD : noCrd \vee (status = \mathsf{Propose})\})$
27     foreach $p_j \in sendSet$ do $send(m)$;
28 Upon message arrival $m$ from $p_j$ do $rep[j] \leftarrow m$;
```

inquires all members (forming a majority) for the most updated state and delivered message. Since at least one of the members, say p_i, participated in the view in which the last completed state machine transition took place, p_i's information will be recognized as associated with the largest counter, adopted by the coordinator that will in turn assign the most updated state and available delivered message to all the current group members, in essence satisfying the virtual synchrony property.

After this, the coordinator, as part of the multicast procedure, will collect inputs received from the environment before ensuring that all group members apply these inputs to the replica state machine. Note that the received multicast message consists of input (possibly $\perp$) from each of the processors, thus, the processors need to apply one input at a time, the processors may apply them in an agreed upon sequential order, say from the input of the first processor to the last. Alternatively, the coordinator may request one input at a time in a round-robin fashion and multicast it. Finally, to ensure that the system stabilizes when started in an arbitrary configuration, every so often, the coordinator assigns the state of its replica to the other members.

If the system reaches a configuration with no coordinator, e.g., due to an arbitrary configuration that the system starts in, or due to the coordinator's crash. Each participant detecting the absence if a coordinator, seeks for potential candidates based on the exchanged information. A processor p regards a processor q as a candidate, if q is active according to p's failure detector, and there is a majority of processors that also think so (all these are based on p's knowledge, which due to asynchrony might not be up to date). When there is more than one such candidate, processor p checks whether there is a candidate that has proposed a view with a highest identifier (i.e., counter) among the candidates. If there is one, then p considers this to be the coordinator and waits to hear from it (or learn that it is not active).

If there is none such, and if based on its local knowledge there is a majority of processors that also do not have a coordinator, then processor p acquires a counter from the counter increment algorithm and proposes a new view, with view ID, the counter, and group membership, the set of processors that appear active according to its failure detector. As we show, if p receives an "accept" message from *all* the processors in the view, then it proceeds to install the view, unless another processor who has obtained a higher counter does so. In a transition from one view to the next, there can be several processors attempting to become the coordinator (namely, those who according to their knowledge have a supporting majority). Still, by exploiting the intersection property of the supporting majorities we prove that each of these processors will propose a view at most once, and out of these, one view will be installed (i.e., we do not have never-ending attempts for new views to be installed).

As an aside, we note that GCSs that provide VS often leverage on the system's ability to preserve (when possible) the coordinator during view transitions rather than venturing to install a new one, a certainly more expensive procedure. Our solution naturally follows this approach through our assumption of a supportive majority (Definition 4.1), where coordinators enjoy the support of a majority of processors by never being supported throughout a very long period. During such a period, our algorithm persists on using the same coordinator, even when views change.

4.2.2 Detailed algorithm description

The existence of coordinator p_ℓ is in the heart of Algorithm 4. Processors that belong to and accept p_ℓ's view proposal are called the *followers* of p_ℓ. The algorithm determines the availability of a coordinator and acts towards the election of a new one when no valid such exists (lines 5 to 9). The pseudocode details the coordinator-side (lines 10 to 16) and the follower-side (lines 17 to 22) actions. At the end of each iteration the algorithm, defines how p_ℓ and its followers exchange messages (lines 25 to 28).

The processor state and interfaces The state of each processor includes its current *view*, and *status* = {Propose, Install, Multicast}, which refers to usual message multicast operation when in Multicast, or view establishment rounds in which the coordinator can Propose a new view and proceed to Install it once all preparations are done (line 3). During multicast rounds, *rnd* denotes the round number, *state* stores the replica, $msg[n]$ is an array that includes the last delivered messages to the state machine, which is the *input* fetched by each group member and then aggregated by the coordinator during the previous multicast round. During multicast rounds, it holds that $propV = view$. However, whenever $propV \neq view$ we consider $propV$ as the newly proposed view and $view$ as the last installed one. Each processor also uses $noCrd$ and FD to indicate whether it is aware of the absence of a recently active and connected valid coordinator, and respectively, of the set of processor present in the connected component, as indicated by its local failure detector. The processors exchange their state via message passing and store the arriving messages in the replica's array, $rep[n]$ (line 28), where $rep[i].(view, \ldots, noCrd)$ is an alias to the aforementioned variables and $rep[j]$ refers to the last arriving message from processor p_j containing p_j's $rep[j]$. Our presentation also uses subscript $_k$ to refer to the content of a variable at processor p_k, e.g., $rep_k[j].view$, when referring to the last installed view that processor p_k last received from p_j.

Algorithm 4 assumes access to the application's message queue via $fetch()$, which returns the next multicast message, or $\perp$ when no such message is available (line 2). It also assumes the availability of the automaton state transition function, $apply(state, msg)$, which applies the aggregated input array, msg, to the replica's $state$ and produces the local side effects. The algorithm also collects the followers' replica states and uses $synchState(replica)$ to return the new state. The function $failureDetector()$ provides access to p_i's failure detector, and the function $inc()$ (counter increment) fetches a new and unique (view) identifier, ID, that can be totally ordered by $\preceq_{ct}$ and $ID.wid$ is the identity of the processor that incremented the counter, resulting to the counter value ID (hence view IDs are counters as defined in Section 3.3). Note that when two processors attempt to concurrently increment the counter, due to symmetry breaking, one of the two counters is the largest. Each processor will continue to propose a new view based on the counter written, but then (as described below) the one will the highest counter will succeed (line 7).

Determining coordinator availability The algorithm takes an agile approach to multicasting with atomic delivery guarantees. Namely, a new view is installed whenever the coordinator sees a change to its local failure detector, $failureDetector()$, which p_i stores in FD_i (line 5). Nevertheless, we might reach a configuration without a view coordinator as a result of an arbitrary initial configuration, or of a coordinator becoming inactive. Using the failure detector heartbeat exchange, processors can detect such initially corrupted states. Each participant that detects that it has no coordinator, seeks for potential candidates based on the exchanged information.

Processor p_i can see the set of processors, $seemCrd_i$, that each *seems* to be the view coordinator, because p_i stored a message from $p_\ell \in FD_i$ in which $p_\ell = rep[\ell].propV.ID.wid$. Note that p_i cannot consider p_ℓ as a (seemly) coordinator unless the conditions in line 6 hold. Intuitively, such a processor must be active according to p_i's failure detector, and there is a majority of processors that also think so. Note that all these are based on local knowledge, which due to asynchrony might not be up to date. The next step is for p_i to consider the processor in $seemCrd_i$ with the $\preceq_{ct}$-greatest view identifier (line 7) as the valid coordinator. Here, set $valCrd_i$ is either a singleton or empty (line 8). If p_i considers some processor p_ℓ as a valid coordinator, it waits to hear from p_ℓ (or learn that it is not active). We call p_i a *follower* of p_ℓ. If there is no such processor, p_i will only propose a new view if its failure detector indicates that there exists a supportive majority of active processors that are also without a valid coordinator (line 9). If such a majority exists, p_i acquires a counter from the counter increment algorithm and proposes a new view, with the counter as the view ID, and the set of processors that appear active according to its failure detector as the group membership.

As we show, if p_i's view is accepted from *all* the processors in the view, then it proceeds to install the view, unless another processor who has obtained a higher counter does so. In a transition from one view to the next, there can be several processors attempting to become the coordinator (namely, those who according to their knowledge have a supporting majority). Still, by exploiting the intersection property of the supporting majorities we prove that each of these processors will propose a view at most once, and out of these, one view will be installed (i.e., we do not have never-ending attempts for new views to be installed). To satisfy the VS property, no new multicast message is delivered to a new view, before the coordinator of this new view has collected all the participants' last delivered messages (of their prior views) and has resent the messages appearing not to have been delivered uniformly.

The coordinator-side Processor p_i is aware of its valid coordinatorship when $(valCrd_i = \{p_i\})$ (line 10). It takes action related to its role as a coordinator when it detects the round end, based on input from other processors. During a normal Multicast round, p_i observes the round end once for every view member p_j it holds that $(rep_i[j].(view, status, rnd) = (view_i, status_i, rnd_i))$ in line 10. In the cases of Propose and Install rounds, the algorithm does not need to consider the round number, rnd.

Depending on its $status$, the coordinator p_i proceeds once it observes the successful round conclusion. At the end of a normal Multicast round (line 11), the coordinator increments the round number (line 14) after applying the changes to its local replica (line 12) and aggregating the followers' input (line 13). The coordinator continues from the end of a Propose round to an Install round after using the most recently received replicas to install a synchronized state of the emulated automaton (line 15). At the end of a successful Install round, the coordinator proceeds to a Multicast round after installing the proposed view and the first round number (line 16). (Note that implicitly the coordinator creates a new view if it detects that the round number is exhausted $(rnd > 2^{\tau})$, or if there is another member of its view that has a greater round number than the one this coordinator has. This can only be due to corruption in the initial arbitrary state which affected rnd part of the state.)

The follower-side Processor p_i is aware of its coordinator's identity when $(valCrd_i = \{p_\ell\})$ and $i \neq \ell$ (line 17). Being a follower, p_i only enters this block of the pseudocode when it receives a new message, i.e., the first message round when installing a new view $(rep[\ell].rnd = 0)$, the first time a message arrives $(rnd < rep[\ell].rnd)$ or a new view is proposed $(rep[\ell].(view \neq propV))$.

During normal Multicast rounds (lines 18–21) the follower p_i adopts the coordinator's replica, applies the aggregated message of this round to its current automaton state so that it produces the needed side-effects, and then fetches new messages from the environment. Note that, in the case of a Propose round (line 23), the algorithm design stops p_i from overwriting its round number, thus allowing the coordinator to know what was the last round number that it delivered during the last installed view. This is only overwritten on upon the installation of the new view (line 22).

The exchanging message and PCE optimization Each processor periodically sends its current replica (line 27) and stores the received ones (line 28). As an optimization, we propose to avoid sending the entire replica state in every Multicast round. Instead, we consider a predefined constant, PCE (periodic consistency enforcement), that determines the maximum number of Multicast rounds during which the followers do not transmit their replica state to the coordinator and the coordinator does not send its state to them (lines 19 and 25). Note that the greater the PCE's size, the longer it takes to recover from transient faults. Therefore, one has to take this into consideration when extending the approach of periodic consistency enforcement to other elements of replica, e.g., in $view$ and $propV$, one might want to reduce the communication costs that are associated with the set field and the epoch part of the ID field.

4.3 Correctness Proof of Algorithm 4

The correctness proof shows that starting from an arbitrary state in an execution R of Algorithm 4 and once the primary partition property (Definition 4.1) holds throughout R, we reach a configuration $c \in R$ in which some processor with supporting majority p_ℓ will propose a view including its supporting majority. This view is either accepted by all its member processors or in the case where p_ℓ experiences a failure detection change, it can repropose a view.

We conclude by proving that any execution suffix of R that begins from such a configuration c will preserve the virtual synchrony property and implement state machine replication. We begin with some definitions. Intuitively, the latter part of the proof is deduced as follows: once a processor does not have a coordinator, it stops participating in group multicasting, and prior to delivering a new multicast message in a new view, the algorithm assures that the coordinator of this new view has collected all the participants' last delivered messages (in their prior views) and resends the messages appearing not to have been delivered uniformly. To do so, each participant keeps the last delivered message and the view identifier that delivered this message. This, together with the intersection property of majorities, provides the virtual synchrony property. We begin with some definitions.

Once the system considers processor p_ℓ as the view coordinator (Definition 4.1) its supporting majority can extend the support throughout R and thus p_ℓ continues to emulate the automaton with them. Furthermore, there is no clear guarantee for a view coordinator to continue to coordinate for an unbounded period when it does not meet the criteria of Definition 4.1 throughout R. Therefore, for the sake of presentation simplicity, the proof considers any execution R with only *definitive suspicions*, i.e., once processor p_i suspects processor p_j, it does not stop suspecting p_j throughout R. The correctness proof implies that eventually, once all of R's suspicions appear in the respective local failure detectors, the system elects a coordinator that has a supporting majority throughout R.

Consider a configuration c in an execution R of Algorithm 4 and a processor $p_i \in P$. We define the *local (view) coordinator* of p_i, say p_j, to be the only processor that, based on p_i's local information, has a proposed view satisfying the conditions of lines 6 and 7 such that $valCrd = \{p_j\}$. p_j is also considered the *global (view) coordinator* if for all p_k in p_j's proposed view ($propV_j$), it holds that $valCrd_k = \{p_j\}$. When p_i has a (local) coordinator then p_i's local variable $noCrd = \mathsf{False}$, whilst when it has no local coordinator, $noCrd = \mathsf{True}$. Moving to the proof, we consider the following useful remark on Definition 4.1 of page 22.

Remark 4.1 *Definition 4.1 suggests that we can have more than one processor that has supporting majority. In this case, it is not necessary to have* the same *supporting majority for all such processors. Thus for two such processors p_i, p_j with respective supporting majorities $P_{maj}(i)$ and $P_{maj}(j)$ we do not require that $P_{maj}(i) = P_{maj}(j)$, but $P_{maj}(i) \cap P_{maj}(j) \neq \emptyset$ trivially holds.*

Lemma 4.1 *Let R be an execution with an arbitrary initial configuration, of Algorithm 4 such that Definition 4.1 holds. Consider a processor $p_i \in P_{maj}$ which has a local coordinator p_k, such that p_k is either inactive or it does not have a supporting majority throughout R. There is a configuration $c \in R$, after which p_i does not consider p_k to be its local coordinator.*

Proof. There are the two possibilities regarding processor p_k.
Case 1: We first consider the case where p_k is inactive throughout R. By the design of our failure detector, p_i is informed of p_k's inactivity such that line 5 will return an FD_i to p_i where $p_k \notin FD_i$. The threshold W that we set for our failure detector determines how soon p_k is suspected. By the first condition of line 6 we have that $p_k \notin FD_i \Rightarrow p_k \notin seemCrd \Rightarrow p_k \notin valCrd_i$, i.e., p_i stops considering p_k as its local coordinator. By definitive suspicions, that p_i does not stop suspecting p_k throughout R.

We now turn to the case where p_k is active, however it does not have a supporting majority throughout R, but p_i still considers p_k as its local coordinator, i.e. $valCrd_i = \{p_k\}$. Two subcases exist:
Case 2(a): p_k considers itself to have a supporting majority, and $p_i \in propV_k$. Note that the latter assumption implies that p_k is forced by lines 24 - 27 to propagate $rep_k[k]$ to p_i in every iteration. By the failure detector, there exists an iteration where p_k will have $|FD_k = n/2 + 1|$ and is informed that some $p_j \in propV_k$ has $p_k \notin FD_j$ and so the condition of line 6 ($FD > \lfloor n/2 \rfloor$) fails for p_k, which stops being the coordinator of itself. If p_k does not find a new coordinator, hence $noCrd_k = \mathsf{True}$, then p_k propagates its $rep_k[k]$ to p_i. But this implies that p_i receives $rep_k[k]$ and stores it in $rep_i[k]$. Upon the next iteration of this reception, p_i will remove p_k from its $seemCrd$ set because p_k does not satisfy the condition $|rep_i[k].FD| < \lfloor n/2 \rfloor$ of line 6. We conclude that p_i stops considering p_k as its local coordinator if p_k does not find a new coordinator. Nevertheless, p_k may find a new coordinator before propagating $rep_k[k]$. If p_k has a coordinator other than itself, then it only propagates $rep_k[k]$ to its coordinator and thus p_i does not receive this information. We thus refer to the next case:
Case 2(b): p_k has a different local coordinator than itself. This can occur either as described in Case 2(a) or as a result of an arbitrary initial state in which p_i believes that p_k is its local coordinator but p_k has a different local coordinator. We note that the difficulty of this case is that p_k only sends $rep_k[k]$

to its coordinator, and thus the proof of Case 2(a) is not useful here. As explained in Algorithm 4, the failure detector returns a set with the identities (*pid*) of all the processors it regards as active, as well as the identity of the local coordinator of each of these processors. As per the algorithm's notation, the coordinator of processor p_k is given by $crd(k)$. Since p_i's failure detector regards p_k as active, then $crd(k)$ is indeed updated (remember that p_i receives the token with p_k's $crd(k)$ infinitely often from p_k), otherwise p_k is removed from FD and is not a valid coordinator for p_i. But p_k does not consider itself as the coordinator (by the assumption of Case 2(b)), and thus it holds that $crd(k) \neq k$. Therefore, in the first iteration after p_i receives $crd(k) \neq k$, one of the last two conditions of line 6 fails (depending on what is the view status that p_i has in $rep_i[k]$) so $p_k \notin seemCrd_i$ and thus $valCrd_i \neq \{p_k\}$. We conclude that any such p_k stops being p_i's coordinator and by the assumption of definitive suspicions we reach to the result. It is also important to note that p_k never again satisfies all the conditions of line 9 to create a new view. ■

We now define the notion of "propose" more rigorously to be used in the sequel.

Definition 4.2 *Processor $p_\ell \in P$ with $status = \mathsf{Propose}$, is said to* propose *a view $propV_\ell$, if in a complete iteration of Algorithm 4, p_ℓ either satisfies $valCrd_\ell = \{p_\ell\}$ or satisfies all the conditions of line 9 to create $propV_\ell$. A proposal is completed when $propV_\ell$ is propagated through lines 24–27 to all the members of FD_ℓ.*

The above definition does not imply that p_ℓ will continue proposing the view $propV$, since the replicas received from other processors may force p_ℓ to either exclude itself from $valCrd_\ell$ or create a new view (see Lemma 4.3). If the view is installed, then the proposal procedure will stop, although $propV_\ell$ will still be sent as part of the replica propagation at the end of each iteration. Also note that the origins of such a proposed view are not defined. Indeed it is possible for a view that was not created by p_ℓ but bears p_ℓ's creator identity to come from an arbitrary state and be proposed, as long as all the conditions of lines 6 and 7 are met.

Lemma 4.2 *If the conditions of Definition 4.1 hold throughout an execution R of Algorithm 4, then starting from an arbitrary configuration in which there is no global coordinator, the system reaches a configuration in which at least one processor with a supporting majority will propose a view (with "propose" defined as in Definition 4.2).*

Proof. By Definition 4.1, at least one processor with supporting majority exists. Denote one such processor as p_ℓ. Assume for contradiction that throughout R, no processor p_ℓ with supporting majority proposes a view. p_ℓ either has a local coordinator (that is not global) or does not have a coordinator.
Case 1: $\mathbf{p_\ell}$ does not have a coordinator ($\mathbf{noCrd_\ell} = \mathsf{True}$). If p_ℓ does not propose a view (as per the "propose" Definition 4.2), this is because it does not hold a proposal that is suitable and it does not satisfy some condition of line 9 which would allow it to create a new view. The first condition of line 9, $(|FD| > \lfloor n/2 \rfloor)$ is always satisfied by our assumption that p_ℓ is not suspected by a majority throughout R. In the second condition, both (i) $((|valCrd_\ell| \neq 1) \wedge (|\{p_i \in FD_\ell : p_\ell \in rep_\ell[i].FD_\ell \wedge rep_\ell[i].noCrd\}| > \lfloor n/2 \rfloor))$ and (ii) $((valCrd_\ell = \{p_\ell\}) \wedge (FD_\ell \neq propV_\ell.set) \wedge (|\{p_i \in FD : rep[i].propV = propV\}| > \lfloor n/2 \rfloor))$ must fail due to our assumption that p_ℓ never proposes. Indeed (ii) fails since $noCrd_\ell = \mathsf{True} \Rightarrow valCrd_\ell \neq \{p_\ell\}$. If the first expression also fails, this implies that throughout R, p_ℓ does not know of a majority of processors with $noCrd = \mathsf{True}$ and so it cannot propose a new view.

Let's assume that only one processor $p_j \in P_{maj}(\ell) \subseteq FD_\ell$ is required to switch from $noCrd_j = \mathsf{False}$ to True in order for p_ℓ to gain a majority of processors without a coordinator. But if $noCrd_j = \mathsf{False}$ then p_j must already have a coordinator, say p_k. We have the following two subcases:
Case 1(a): p_k does not have a supporting majority. Lemma 4.1 guarantees that p_j stops considering p_k as its local coordinator. Thus p_j eventually goes to $noCrd = \mathsf{True}$ and by the propagation of its replica, p_ℓ receives the required majority to go into proposing a view. But this contradicts our initial assumption, so we are lead to the following case.
Case 1(b): p_k has a supporting majority and a view proposal $propV_k$ from the initial arbitrary configuration but is not the global coordinator. But this implies that the Lemma trivially holds, and so the

following case must be true.
Case 2: $\mathbf{p}_\ell$ has a coordinator, say $\mathbf{p}_{\mathbf{k}'}$. The two subcases of whether $p_{k'}$ has a supporting majority or not, are identical to the two subcases 1(a) and 1(b) concerning p_k that we studied above. Thus, it must be that either p_ℓ will eventually propose a label, or that $p_{k'}$ has a proposed view, thus contradicting our assumption and so the lemma follows. ■

Lemma 4.2 establishes that at least one processor with supporting majority will propose a view in the absence of a valid coordinator. We now move to prove that such a processor will only propose one view, unless it experiences changes in its FD that render the view proposal's membership obsolete. The lemma also proves that any two processors with supporting majority will not create views in order to compete for the coordinatorship.

Lemma 4.3 (Closure and Convergence) *If the conditions of Definition 4.1 hold throughout an execution R of Algorithm 4, then starting from an arbitrary configuration, the system reaches a configuration in which any processor p_ℓ with a supporting majority proposes a view $propv_\ell$, and cannot create a new proposed view in R unless $FD_\ell \neq propV_\ell.set$ and a majority of processors has adopted $propV_\ell$. As a consequence, the system reaches a configuration in which one processor with supporting majority is the global coordinator until the end of the execution.*

Proof. We distinguish the following cases:
Case 1: Only one processor with supporting majority exists. Assume there is only a single processor p_ℓ that has a supporting majority throughout R. According to Lemma 4.2, p_ℓ must eventually propose a view $propV_\ell$, based on the current FD_ℓ reading (line 5) which becomes the $propV_\ell.set$. By Lemma 4.1, any other processor without a supporting majority will eventually stop being the local coordinator of any $p_j \in propV_\ell.set$ and since such processors do not have a supporting majority, the first condition of line 9 will prevent them from proposing.

Processor p_ℓ continuously proposes $propV_\ell$ until all processors in $propV_\ell.set$ have sent a replica showing that they have adopted $propV_\ell$ as their $propV$. Every processor that is alive throughout R and in FD_ℓ should receive this replica through the self-stabilizing reliable communication. The only condition that may prevent p_j to adopt $propV_\ell$ is if for some $p_r \in rep_j[\ell].propV_\ell.set$ it holds that $p_\ell \notin rep_j[r].FD$ (line 6). Plainly put, p_j believes that p_r suspects p_ℓ.

Case 1(a): If p_j's information is correct about p_r, then $p_r \notin P_{maj}(\ell)$. Thus at some point p_ℓ will suspect p_r and exclude p_r from FD_ℓ.

Case 1(b): If p_j's information is false –remnant of some arbitrary state– then $p_\ell \in FD_r$ and since p_r, by the last condition of line 26, sends $rep_r[r]$ infinitely often to p_j, then $rep_j[r]$ will be corrected and p_j will accept $propV_\ell$.

Since p_ℓ has a majority $P_{maj}(\ell) \subseteq propV_\ell.set$, then at least a majority of processors have received $propV_\ell$ and eventually accept it. If some processor $p_j \in propV_\ell$ does not adopt p_ℓ's proposal in R, it is eventually removed from FD_ℓ and thus does not belong to the supporting majority of p_ℓ (as detailed in Case 1(a) above). By the above we note that p_ℓ is able to get at least the supporting majority $P_{maj}(\ell)$ to accept its view if not all of the members in $propV_\ell.set$. In the last case it can proceed to the installation of the view. If there is any change in the failure detector of p_ℓ before it installs a view, p_ℓ can satisfy the second case of line 9, to create a new updated view. Note that in the mean time no processor other than p_ℓ can satisfy the conditions of that line, and thus it is the only processor that can propose and become the coordinator.
Thus p_ℓ eventually becomes the coordinator if it is the single majority-supported processor.

Case 2: More than one processor with supporting majority. Consider two processors $p_\ell, p_{\ell'}$ that have a supporting majority such that each creates a view (line 9). By the correctness of our counter algorithm, $inc()$ returns two distinct and ordered counters to use as view identifiers. Without loss of generality, we assume that $propV_\ell$ proposed by p_ℓ has the greatest identifier of all the counters created by calls to $inc()$. We identify the following four subcases:

Case 2(a): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. In this case $p_{\ell'}$ will propose its view $propV_{\ell'}$ and wait for all $p_i \in propV_{\ell'}.set$ to adopt it (line 10). Whenever p_ℓ receives $propV_{\ell'}$, it will store it but will not adopt it, since $propV_{\ell'}.ID \preceq_{ct} propV_\ell.ID$ (line 7). The proposal $propV_\ell$ is also propagated to every $p_i \in propV_\ell.set$. Since there is no greater proposed view identifier than $propV_\ell.ID$, this is adopted by all $p_i \in propV_\ell$ which also includes $p_{\ell'}$ as well. Thus any processor with supporting majority that belonged to the proposed set of p_ℓ will propose at most once, and p_ℓ will become the sole coordinator. Note that if $p_{\ell'}$ is prevented from adopting $propV_\ell$ for some time, this is due to reasons detailed and solved in Case 1 of the previous lemma. The case where the failure detection reading changes for p_ℓ is also tackled as in Case 1 of this lemma, by noticing that if p_ℓ manages to get a majority of processors of $propV.set$ then p_ℓ will change its proposed view without losing this majority.

Case 2(b): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Since both processors were able to propose, this implies that a majority of processors that belonged to each of p_ℓ's and $p_{\ell'}$'s supporting majority had informed that they had no coordinator (line 9). Each of p_ℓ and $p_{\ell'}$, proposes its view to its $propV.set$, and waits for acknowledgments from *all* the processors in $propV.set$ (line 10), in order to install the view. Since $p_\ell \notin FD_{\ell'}$, $p_{\ell'}$ does not consider $propV_\ell$ a valid proposal (line 6) and retains its own proposal that it propagates. The same is done by p_ℓ. Since p_ℓ has the greatest label, any $p_i \in propV_\ell.set \cap propV_{\ell'}.set$ might initially adopt $propV_{\ell'}$ but it will eventually choose the greatest $propV_\ell$. If $p_{\ell'}$'s proposal was accepted by all members of $propV_{\ell'}$ then this means that $p_{\ell'}$ became the global coordinator but will then lose the coordinatorship to p_ℓ because $propV_\ell$ has a greater view identifier.

What is more crucial, is that $p_{\ell'}$ cannot make another proposal, since it will not have a majority of processors that do not have a coordinator. This is deduced from the intersection property of the two majorities ($propV_\ell.set$ and $propV_{\ell'}.set$). Since any processor p_k in the intersection $propV_\ell.set \cap propV_{\ell'}.set$ has p_ℓ as its coordinator, $p_{\ell'}$ does not satisfy the condition $|\{p_k \in FD_{\ell'} : p_{\ell'} \in rep[k].FD \wedge rep[k].noCrd\}| > \lfloor n/2 \rfloor$ of line 9, and thus cannot propose a new view. Processor p_ℓ will install its view and remains the sole coordinator. Also, p_ℓ is the only one that can change its view due to failure detector change since it manages to get a majority of processors in $propV_\ell.set$ as opposed to $p_{\ell'}$.

Case 2(c): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Here we note that since p_ℓ has the greatest counter but has not included $p_{\ell'}$ to its $propV_\ell.set$, it should eventually be able to get all the processors in $propV_\ell.set$ to follow $propV_\ell$ by using the arguments of Case 2(a). In the mean time $p_{\ell'}$ will, in vain, be waiting for a response from p_ℓ accepting $propV_{\ell'}$. We note that $p_{\ell'}$ will not be able to initiate a new view once $propV_\ell$ is accepted, since it will not be able to gather a majority of processors with either $noCrd = \mathsf{True}$ or proposed view $propV_{\ell'}$.

Case 2(d): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. This case is not symmetric to the above due to our assumption that p_ℓ is the one that has drawn the greatest view identifier from $inc()$. Here $propV_\ell.set$ includes $p_{\ell'}$ so p_ℓ waits for a response from $p_{\ell'}$ to proceed to the installation of $propV_\ell$. On the other hand, $p_{\ell'}$ will be waiting for responses from the processors in $propV_{\ell'}.set$. Any $p_i \in propV_\ell.set \cap propV_{\ell'}.set$ cannot keep $propV_\ell$ (even if initially it has accepted it, since it does not satisfy condition $p_{\ell'} \in rep[\ell].propV.set \Leftrightarrow p_\ell \in rep[\ell'].FD$ of line 6. Thus p_i accepts $propV_{\ell'}$ instead of $propV_\ell$, p_ℓ cannot propose a different view since it will not be able to get a majority of processors that have $propV_\ell$.

By the above exhaustive examination of cases, we reach to the result. Note that the above proof guarantees both convergence and closure of the algorithm to a legal execution, since p_ℓ remains the coordinator as long as it has a supporting majority. ■

Theorem 4.4 *Starting from an arbitrary configuration, any execution R of Algorithm 4 satisfying Definition 4.1, simulates automaton replication preserving the virtual synchrony property.*

Proof. Consider a finite prefix R' of R. Assume that in this prefix Lemma 4.3 holds, i.e., we reach a configuration in which a processor p_ℓ has a supporting majority and is the global coordinator with view v. We define a *multicast round* to be a sequence of ordered events: (i) $fetch()$ input and propagate to coordinator, (ii) coordinator propagates the collected messages of this round, (iii) messages are delivered and (iv) all view members $apply()$ side effects. The VS property is preserved between two consecutive rounds r, r' that may belong to different views v, v' (with possibly identical coordinators $p_\ell, p_{\ell'}$) respectively, if and only if $\forall p_i \in v.set \cap v'.set$ it holds that every $rep_i[i].input$ at round r is in $rep_[\ell'].msg[i]$ of round r'. Our proof is progressive: Claim 4.5 proves that VS is preserved between any two consecutive multicast rounds, Claim 4.6 that VS is preserved in two consecutive views with the same coordinator and Claim 4.7 preservation in two consecutive view installations where the coordinator changes.

Claim 4.5 *VS is preserved between r and r' where $v = v'$.*

Proof. Suppose that there exists an input and a related message m in round r that is not delivered within r. We follow the multicast round r. First observe the following.
Remark: Within any multicast round, the coordinator executes lines 12 to 14 only once and a follower executes lines 18 to 21 only once, because the conditions are only satisfied the first time that the coordinator's local copy of the replica changes the round number.

By our Remark we notice that $fetch()$ is called only once per round to collect input from the environment. This cannot be changed/overwritten since followers can never access $rep[i] \leftarrow rep[\ell]$ of line 20 that is the only line modifying the *input* field, unless they receive a new round number greater than the one they currently hold. We notice that the followers have produced side effects for the previous round (using $apply()$) based on the messages and state of the previous round. Similarly, the coordinator executes $fetch()$ exactly once and only before it populates the *msg* array and after it has produced the side effects for the environment that were based on the previous messages (line 12). Line 13 populates the *msg* array with messages and including m. The coordinator $p_\ell$then continuously propagates its current replica but cannot change it by the Remark and until condition ($\forall\, p_i \in v.set : rep_\ell[i].(view, status, rnd) = (view_\ell, status_\ell, rnd_\ell)$) (line 10) holds again. This ensures that the coordinator will change its *msg* array only when every follower has executed line 20 which allows the aforementioned condition to hold.

Any follower that keeps a previous round number does not allow the coordinator to move to the next round. If the coordinator moves to a new round, it is implied that $rep[i] \leftarrow rep[\ell]$ and thus message m was received by any follower p_i, by our assumptions that the replica is propagated infinitely often and the data links are stabilizing. Thus, by the assumptions, any message m is certainly delivered within the view and round it was sent in, and thus the virtual synchrony property is preserved, whilst at the same time common state replication is achieved. ■

Claim 4.6 *VS is preserved between r and r' where $v \neq v'$ and $p_\ell = p_{\ell'}$.*

Proof. We now turn to the case where from one configuration c_{safe} we move to a new c'_{safe} that has a different view v' but has the same coordinator p_ℓ. Once p_ℓ is in an iteration where the condition $FD \neq propV.set$ of line 9 holds, a view change is required. Since p_ℓ is the global coordinator holds, no other processor can satisfy the condition ($|\{p_k \in FD_\ell : rep[k]_\ell.propV = propV_\ell\}| > \lfloor n/2 \rfloor$) of line 9, and so only p_ℓ. For more on why this holds one can prefer to Lemma 4.2. Processor p_ℓ creates a new *propV* with a new view ID taken from the increment counter algorithm, which is greater than the previous established view ID in $v.ID$. The last condition of line 10 guarantees that p_ℓ will not execute lines 12 to 16 and thus will not change its $rep.(state, input, msg)$ fields, until all the expected followers of the proposed view have sent their replicas. Followers that receive the proposal will accept it, since none of the conditions that existed change and so the new view proposal enforces that $valCrd = \{p_\ell\}$. Moreover, the proposal satisfies the condition of line 17 and the followers of the view enter status Propose leading to the installation of the view. What is important is that virtual synchrony is preserved since no follower is changing $rep.(state, input, msg)$ during this procedure, and moreover each sends its replica to p_ℓ by line 26. Once the replicas of all the followers have been collected, the coordinator creates a consolidated *state* and *msg* array of all messages that were either delivered or pending. p_ℓ's new replica is communicated to the followers who adopt this state as their own (line 22). Thus virtual synchrony is preserved and once all the processors have replicated the state of the coordinator, a new series of multicast rounds can begin by producing the side effects required by the input collected before the view change.
■

Claim 4.7 *VS is preserved between r and r' where $v \neq v'$ and $p_\ell \neq p_{\ell'}$.*

Proof. We assume that p_ℓ had a supporting majority throughout R'. We define a matching suffix R'' to prefix R', such that R'' results from the loss of supporting majority by p_ℓ. Notice that since Definition 4.1 is required to hold, then some other processor with supporting majority $p_{\ell'}$, will by Lemma 4.2 propose the view v' with the highest view ID. We note that by the intersection property and the fact that a view set can only be formed by a majority set, $\exists p_i \in v \cap v'$. Thus, the "knowledge" of the system, $(state, input, msg)$ is retained within the majority.

As detailed in step 2, if a processor p_i had $noCrd = \mathsf{True}$ for some time or was in status Propose it did not incur any changes to its replica. If it entered the Install phase, then this implies that the proposing processor has created a consolidated state that p_i has replicated. What is noteworthy is that whether in status Propose or Install, if the proposer collapses (becomes inactive or suspected), the virtual synchrony property is preserved. It follows that, once status Multicast is reached by all followers, the system can start a practically infinite number of multicast rounds.

Thus, by the self-stabilization property of all the components of the system (counter increment algorithm, the data links, the failure detector and multicast) a legal execution is reached in which the virtual synchrony property is guaranteed and common state replication is preserved. content... ■ ■

4.4 Algorithm Complexity

The local memory for this algorithm consists of n copies of two labels, of the encapsulated state (say of size $|S|$ bits) and of other lesser size variables. These give a *space complexity* of order $\mathrm{O}(n\beta \log \beta + n|S|)$; recall that $\beta = n^3cap + 2n^2 - 2n$. *Stabilization time* can be provided by a bound on view creations. It is, therefore, implicit that stabilization is dependent upon the stabilization of the counter algorithm, i.e., $\mathrm{O}(n \cdot \beta \cdot t)$, before processors can issue views with identifiers that can be totally ordered. When this is satisfied, then Lemma 4.3 suggests that $O(n)$ view creations are required to acquire a coordinator, namely, in the worse case where every processor is a proposer. Once a coordinator is established then Theorem 4.4 guarantees that there can be practically infinite multicast rounds (0 to 2^τ).

5 Conclusion

State-machine replication (SMR) is a service that simulates finite automata by letting the participating processors to periodically exchange messages about their current state as well as the last input that has led to this shared state. Thus, the processors can verify that they are in sync with each other. A well-known way to emulate SMRs is to use reliable multicast algorithms that guarantee *virtual synchrony* [4, 19]. To this respect, we have presented the first practically-self-stabilizing algorithm that guarantees virtual synchrony, and used it to obtain a practically-self-stabilizing SMR emulation; within this emulation, the system progresses in more extreme asynchronous executions in contrast to consensus-based SMRs, like the one in [7]. One of the key components of the virtual synchrony algorithm is a novel practically-self-stabilizing counter algorithm, that establishes an efficient practically unbounded counter, which in turn can be directly used to implement a practically-self-stabilizing MWMR register emulation; this extends the work in [1] that implements SWMR registers and can also be considered simpler and more communication efficient than the MWMR register implementation presented in [7].

Acknowledgements

We thank Iosif Salem for providing comments to improve the readability of this paper. We also thank the anonymous reviewers whose constructive feedback has helped us to significantly improve the presentation of our results and of the manuscript in general.

References

[1] 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.

[2] Anish Arora, Sandeep S. Kulkarni, and Murat Demirbas. Resettable vector clocks. *J. Parallel Distrib. Comput.*, 66(2):221–237, 2006.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[4] Alberto Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.

[5] Ken Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 91–120. Springer, 2010.

[6] Keneth Birman and Robbert Van Renesse. *Reliable distributed computing with the Isis toolkit.* Wiley-IEEE Computer society press Los Alamitos, 1994.

[7] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing paxos replicated state-machine. In *Networked Systems - Second International Conference, NETYS 2014, Marrakech, Morocco, May 15-17, 2014. Revised Selected Papers*, pages 99–121, 2014.

[8] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing paxos replicated state-machine. In *In Revised Selected Papers of the Second International Conference on Networked Systems, NETYS 2014*, volume 8593 of *Lecture Notes in Computer Science*, pages 99–121. Springer, 2014.

[9] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[10] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[11] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 338–346, 2004.

[12] Edsger W Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[13] Shlomi Dolev. *Self-stabilization.* The MIT press, 2000.

[14] 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.

[15] 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 - (extended abstract). In *Proceedings of the 14th International Symposium Stabilization, Safety, and Security of Distributed Systems, SSS 2012*, pages 133–147, 2012.

[16] Shlomi Dolev, Ronen I. Kat, and Elad M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.

[17] Shlomi Dolev, Limor Lahiani, Nancy A. Lynch, and Tina Nolte. Self-stabilizing mobile node location management and message routing. In *Self-Stabilizing Systems*, pages 96–112, 2005.

[18] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

[19] Roger Khazan, Alan Fekete, and Nancy A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In Shay Kutten, editor, *Distributed Computing, 12th International Symposium, DISC '98, Andros, Greece, September 24-26, 1998, Proceedings*, volume 1499 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 1998.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *The 27th Annual International Symposium on Fault-Tolerant Computing (FTC 1997)*, pages 272–281, 1997.

[22] Iosif Salem and Elad M. Schiller. Practically-stabilizing vector clocks. Technical Report 05, Dept. of Computer Science and Engineering, Chalmers Univ. of Technology, Rannvagen 6B, 412 96 (Goteborg) Sweden, 2017. ISSN 1652-926X.

[23] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.