Self Stabilizing Virtual Synchrony∗

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

Abstract

Virtual synchrony is an important abstraction that is proven to be extremely useful when implemented over asynchronous, typically large, message-passing distributed systems. Fault tolerant design is a key criterion for the success of such implementations. This is because large distributed systems can be highly available as long as they do not depend on the full operational status of every system participant. Namely, they employ redundancy in numbers to overcome non-optimal behavior of participants and to gain global robustness and high availability.

Self-stabilizing systems can tolerate transient faults that drive the system to an arbitrary unpredicted configuration. Such systems automatically regain consistency from any such arbitrary configuration, and then produce the desired system behavior. Practically self-stabilizing systems ensure the desired system behavior for practically infinite number of successive steps e.g., $2^{64}$ steps.

We present the first practically self-stabilizing virtual synchrony algorithm. The algorithm is a combination of several new techniques that may be of independent interest. In particular, we present a new counter algorithm that establishes an efficient practically unbounded counter, that in turn can be directly used to implement a self-stabilizing Multiple-Writer Multiple-Reader (MWMR) register emulation. Other components include self-stabilizing group membership, self-stabilizing multicast, and self-stabilizing emulation of replicated state machine. As we base the replicated state machine implementation on virtual synchrony, rather than consensus, the system progresses in more extreme asynchronous executions in relation to consensus-based replicated state machine.

∗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. Supported by the University of Cyprus.
§Department of Engineering and Computer Science, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden, Email elad@chalmers.se.
1 Introduction

Virtual Synchrony (VS) has been proven to be very important in the scope of fault-tolerant distributed systems [5]. The VS property ensures that two or more processors that participate in two consecutive communicating groups should have delivered the same messages. 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, datacenters 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 useless 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]. To the best of our knowledge, no self-stabilizing virtual synchrony solution exists.

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 [10] 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. We tackle this issue in our work.

Contributions. We present the first self-stabilizing virtual synchrony solution. Specifically:
• We provide a self-stabilizing counter algorithm using bounded memory and communication bandwidth, and yet (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 self-stabilizing MWMR register emulation.

• The self-stabilizing counter algorithm, together with the proposed implementations of a self-stabilizing reliable multicast service and membership service, are composed to yield a self-stabilizing VS-based State Machine Replication (SMR) implementation.

Related Work. Leslie Lamport was the first to introduce SMR, presenting it as an example in [17]. Schneider [20] 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 (practically) self-stabilizing versions of replicated state machines [1, 9, 13, 14] 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 self-stabilizing bounded-size solutions to the never-exhausted counter problem in the restricted case of a single writer. In [9] a self-stabilizing version of Paxos was developed that led to a self-stabilizing consensus-based SMR implementation. To this end, they extend 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 self-stabilizing (bounded-size never-exhausted) counter that also supports many writers, where a single label 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.
Practically-stabilizing VS and self-stabilizing VS are identical when VS is defined by the behaviour of classical VS algorithms that use (bounded) counters. These algorithms preserve the VS requirements as long as the counters do not reach their upper bound. In our setting, if a counter reaches the upper bound due to a transient fault our self-stabilizing/practically-stabilizing solution introduces a new epoch with new sequence numbers. It, thus, converges to act exactly as the non-stabilizing VS (for the same number of steps) as an initialized non-stabilizing VS algorithm.

Next, in Section 2, we overview our construction, describing the core techniques and the way they establish the desired properties. In Section 3 we present the model of computation we consider. Section 4 details the self-stabilizing Labeling and Increment Counter algorithms. In Section 5 we detail the self-stabilizing Virtual Synchrony algorithm and the resulting replicate state machine emulation. We conclude in Section 6.

2 Our Results in a Nutshell

We start with the necessary succinct description of the system settings (more details in Section 3). We consider an asynchronous message passing system consisting of $n$ communicating processors; each with a unique identifier. We assume that up to a minority of the processors might become inactive. The communication network topology is of a fully connected graph. Any message that is sent infinitely often from one active processor to another active processor is eventually received. We often use the term packets for low-level messages, distinguishing packets that are retransmitted to ensure delivery of high-level messages exactly once. Moreover, we assume that the communication links have known bounded capacity, and thus we can use existing self-stabilizing data-link layer algorithms for emulating reliable FIFO communication channel protocols that can even tolerate message omission, duplication as well as transient faults [11, 12].

2.1 Bounded labeling scheme for multiple writers

As mentioned, Alon et al. [1] presented a bounded labeling scheme to implement 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 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.

When all processors (and hence potential writers) are active, the scheme can be viewed as a simple extension of the one of [1]. Informally speaking, the scheme assures 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. Unfortunately, when some processors are not active, finding a global maximal becomes challenging, since these processors will not “clean up” their local labels. So, roughly speaking, the active processors need to do this indirectly without knowing which processors are inactive. To overcome this problem, we have each processor maintaining bounded FIFO histories on labels appearing to have been created by other processors. These histories eventually accumulate the obsolete labels of the inactive processors. 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 the label order relation. 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 \( \ell_1 \) as its global maximal label. Then, it learns about \( \ell_2 \) and hence adopts it, while forgetting about \( \ell_1 \). Then, learning of \( \ell_3 \) it adopts it. Lastly, it learns about \( \ell_1 \), and as it is greater than \( \ell_3 \), it adopts \( \ell_1 \) 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.

### 2.2 Practically infinite counter for multiple writers

Using our labeling scheme, we show how to implement a practically infinite counter supporting multiple writers. The idea is to extend the labeling scheme to handle counters, where a counter consists of a label, as used in the labeling scheme; an integer sequence number, ranging from 0 to \( 2^b \), where \( b \) is large enough, say \( b = 64 \); and a processor id. Conceptually, if the system stabilizes to use a global maximal label, then the pair of the sequence number and the processor id (of this sequence number) can be used as an unbounded counter, as used, for example, in MWMR register implementations [18, 19]. Specifically, we say that counter \( cnt_1 = (\ell_1, seqn_1, wid_1) \) is smaller than counter \( cnt_2 = (\ell_2, seqn_2, wid_2) \) if \( (\ell_1 \prec \ell_2) \) or \( (\ell_1 = \ell_2 \text{ and } (seqn_1 < seqn_2)) \) or \( (\ell_1 = \ell_2 \text{ and } (seqn_1 = seqn_2) \text{ and } (wid_1 < wid_2)) \). Note that when processors have the same label, the above relation forms a total ordering and processors can increment a shared counter also when attempting to do so concurrently. We argue that starting from any initial configuration, eventually the counter
The counter increment algorithm uses the same structures and procedures as the labeling algorithm, but now with counters instead of labels. To increment the counter, a processor $p_i$ first sends a request to all other processors querying the counter they consider as the global maximum and awaits for responses from a majority. Using a similar procedure as the labeling algorithm it (eventually) finds the maximal epoch label and the maximal sequence number it knows for this label. In other words, it collects counters and finds the counter(s) with the largest global label; there can be more than one such counter, in which case it returns the one with the highest sequence number, breaking symmetry with the sequence number processor identifiers. Then it checks whether this maximal sequence number is exhausted, that is, the sequence number is equal or larger than $2^{64}$ (this could be, for example, due to the arbitrary values in the configuration the system starts in). When this is the case, it proceeds to find a new maximal label until it finds one that is not exhausted and uses the maximal sequence number it knows for this epoch label. Then the processor increments the sequence number by one, sets its identifier as the writer of the sequence number and sends the new counter to all processors, and awaits for acknowledgment from a majority (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).

Note that when a processor $p_i$ establishes a new label $\ell$ as the global maximum, it sets the corresponding counter $cnt = (\ell, 0, i)$; in this case, the label creator identifier and the sequence number writer identifier is $i$. When there is an already established maximal label $\ell$ in the system and processor $p_i$ wants to increment the counter, it increases the corresponding (to $\ell$) maximal sequence number found ($\text{maxseqn}$) by one, and sets the counter $cnt = (\ell, \text{maxseqn} + 1, i)$; in this case, it is possible that the label creator identifier and the sequence number writer identifier are not the same, i.e., if $p_i$ was not the creator of label $\ell$. Also, note that some extra care is needed with respect to counter bookkeeping so as not to increase the size of the bounded histories used in the labeling algorithm. Having a counter increment algorithm, it is not difficult to obtain a practically self-stabilizing MWMR register implementation; counters are associated with values and the counter increment algorithm is run with this small amendment (more details in Sect. 4.3).

2.3 Practically self-stabilizing virtual synchrony and Replicated state machine

Our self-stabilizing Virtual Synchrony implementation combines the implementation of the our counter algorithm and a self-stabilizing FIFO data link between any two participants; the latter is used to implement a self-stabilizing reliable multicast service and a self-stabilizing failure detector (used for the membership service).
Data link implementation. 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) [11, 12]. 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 π₁ until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π₂, and so on and so forth. This assures that the receiver has received packet π₁ before the sender starts sending packet π₂. 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. We use this token exchange technique when implementing a reliable multicast procedure, as well as the basis for a heartbeat for detecting whether a processor is active or not; when a processor is no longer active, the token will not be returned back to the other processor.

Reliable multicast implementation. As we will see next, we use a coordinator to exchange messages (by multicasting) within the group. 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 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 ⊥, 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. 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, when needed.

Failure detector implementation. Every processor p maintains a heartbeat integer counter for every other processor q. 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. This is essentially the failure detector mentioned in [9]. Note that for the correctness of our virtual synchrony algorithm, we require a weaker failure detector. Specifically, we require that at least one processor is not suspected, for sufficiently long time, only by a majority of the processors, as opposed to an eventually perfect failure detector that ensures that after a
certain time, no active processor suspects any other active processor.

Self-stabilizing virtual synchrony implementation. The algorithm is coordinator-based and we consider a primary-group implementation [6]. To assign view identifiers, we use our counter increment algorithm. Specifically, the view identifier is a triple that includes an epoch (label), the currently highest counter, $cnt$, which the counter algorithm obtains, and the processor that has created this counter, $cnt.wid$ (writer), which is also the view coordinator. Note that this defines a simple interface with the counter algorithm, which provides an identical output. Furthermore, the view membership uses the output of the coordinator’s failure detector for defining the set of view members; this helps to maintain a consistent membership among the group members, despite inaccuracies between the various failure detectors; as we show, this does not break the virtual synchrony property, as long as the majority-based failure detector property is preserved. Recall that the coordinator is responsible for the consistency of the multicast mechanism within the group.

It may happen that the system reaches a configuration with no coordinator. For example, this could be the case in the arbitrary configuration that the system starts in, or in the case that the coordinator of an installed view is no longer active. Each participant that detects that it has no 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 higher counter among the candidates. If there is one, then $p$ considers it to be the coordinator and waits to hear from it (or learn that it is not active). If there is none, and based on its 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).

The virtual synchrony property essentially requires that any two processors that participate in two consecutive groups should have delivered the same messages. Roughly speaking, our algorithm preserves this property 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. We show that this, together with the intersection property of majorities, (and after taking care of some subtle issues,) provides the virtual synchrony property. Starting from an arbitrary configuration, we show that if there is no valid coordinator, eventually a processor proposes a new view and, therefore, a valid coordinator is eventually elected. To assure this, processors continuously exchange through the failure detector’s token their coordinator’s identifier (or ⊥ if there’s no such). This helps to detect initially corrupted states when, say a processor \( p_i \) might consider \( p_j \) as its coordinator, but \( p_j \) does not consider itself to be the coordinator. Combining the above with the self-stabilization of the counter increment algorithm, the data links, the failure detector and multicast, we are able to guarantee reaching a legal execution in which the virtual synchrony property is always satisfied.

**Self-stabilizing replicate state machine implementation.** 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 elected, the coordinator 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 group 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. Then the coordinator, as part of the multicast procedure, collects 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 ⊥) 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.

Perhaps some of the above ideas appear conceptually clear, however, there are low-level critical details that are essential to realizing them and prove them correct, as we are ready to describe.
3 System Settings

We consider an asynchronous message passing system as the one used in [1]. The system includes a set $P$ of $n$ communicating processors; we refer to the processor with identifier $i$, as $p_i$. We assume that up to a minority of processors may become inactive. We assume that the system runs on top of a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels [11, 12]. 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. The communication links have bounded capacity, so that the number of packets in every given instance is bounded by a constant. When processor $p_i$ sends a packet, $pkt$, to processor $p_j$, the operation send inserts a copy of $pkt$ 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 $pkt$ from $p_j$, $pkt$ 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.

The code of self-stabilizing algorithms 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 is said to be complete if it starts in the loop’s first line and ends at the last (regardless of whether it enters branches).

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 $(s_1, s_2, \ldots, s_n)$, where each $s_i$ is the state of node $p_i$ (including messages in transit for $p_i$). 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 the initial configuration $c_0$, is obtained from the preceding configuration $c_x$ by the execution of the step $a_x$. A practically infinite execution is an execution with many steps (and iterations), where many is defined to be proportional to the time it takes to execute a step and the life-span time of a system.

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 LE. An algorithm is self-stabilizing with relation to
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 task $LE$ when every (unbounded) execution of the algorithm reaches a safe configuration with relation to the algorithm and the task. An algorithm is practically stabilizing with relation to the task $LE$ if in any practically infinite execution a safe configuration is reached.

The virtual synchrony task requires that any two processors that share the same sequence of views, ought to deliver the same 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. Note that 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. An exemplar virtually synchronous execution can be found in Figure 1.

Notation. Throughout the paper we use the following notation. Let $y$ and $y'$ be two objects that both include the field $x$. We denote $(y =_x y') \equiv (y.x = y'.x)$.

4 Self-stabilizing Labeling Scheme and Counter Algorithm

In this section, we first present and prove correct of the proposed self-stabilizing labeling algorithm and then explain how this can be extended to implement self-stabilizing practically unbounded counters in Section 4.3.
Algorithm 1: The \texttt{nextLabel()} function; code for $p_i$

1. For any non-empty set $X \subseteq D$, function \texttt{pick}(d, X) returns $d$ arbitrary elements of $X$;

   \textbf{input} : $S = \langle \ell_1, \ell_2, \ldots, \ell_k \rangle$ set of $k$ labels.

   \textbf{output} : $\langle i, \text{newSting}, \text{newAntistings} \rangle$.

2. Let $\text{newAntistings} = \{ \ell_j.\text{sting} : \ell_j \in S \}$;

3. $\text{newAntistings} \leftarrow \text{newAntistings} \cup \text{pick}(k - |\text{newAntistings}|, D \setminus \text{newAntistings})$;

4. Return $\langle i, \text{pick}(1, D \setminus (\text{newAntistings} \cup \bigcup_{\ell_j \in S} \ell_j.\text{Antistings})) \rangle, \text{newAntistings} \rangle$.

4.1 Labeling Algorithm for Concurrent Label Creations

4.1.1 Bounded Labeling Scheme

We extend the labeling scheme of \cite{1} to support wait-free multi-writer systems. We do so, by extending the label with a \textit{label creator’s} identifier, so as to break symmetry and decide about the most recent epoch even when two or more writers concurrently attempt to create a new label.

Specifically, we consider the set of integers $D = [1, k^2 + 1]$. A label (or \textit{epoch}) is a triple $\langle l\text{Creator}, \text{sting}, \text{Antistings} \rangle$, where $l\text{Creator}$ is the identity of the processor that established (created) the label, $\text{Antistings} \subset D$ with $|\text{Antistings}| = k$, and $\text{sting} \in D$. Given two labels $\ell_i, \ell_j$, we define the relation $\ell_i \prec_{lb} \ell_j \equiv (\ell_i.l\text{Creator} < \ell_j.l\text{Creator}) \lor (\ell_i.l\text{Creator} = \ell_j.l\text{Creator} \land ((\ell_i.\text{sting} \in \ell_j.\text{Antistings}) \land (\ell_j.\text{sting} \not\in \ell_i.\text{Antistings})))$; we use $\equiv_{lb}$ to say that the labels are identical. Note that the relation $\prec_{lb}$ does not define a total order. For example, when $\ell_i =_{l\text{Creator}} \ell_j$ and $(\ell_i.\text{sting} \not\in \ell_j.\text{Antistings})$ and $(\ell_j.\text{sting} \not\in \ell_i.\text{Antistings})$ these labels are incomparable. As in \cite{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 $\ell$ \textit{cancels} another label $\ell'$, either if they are incomparable or they have the same $l\text{Creator}$ but $\ell$ is greater than $\ell'$ (with respect to $\text{sting}$ and $\text{Antistings}$). A label with creator $p_i$ is said to belong to $p_i$’s domain.

Function \texttt{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. It has the same functionality as the function called \texttt{Nextb()} in \cite{1}, but it additionally considers the label creator. The function essentially composes a new $\text{Antistings}$ set from the stings of all the labels it has as input, and chooses a $\text{sting}$ that is in none of the $\text{Antistings}$ 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 $k$ $\text{Antistings}$ 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 $\text{sting}$. 
4.1.2 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. 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 ⊥, 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 \( 2(n^2 + 2n^2 - 2m) \) where \( m \) is the system’s total link capacity in labels. We later show (c.f. Lemmas 4.3 and 4.4) that these queue sizes are sufficient to prevent overflows of useful labels.

**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, \( max_i[i] \) (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 \( max_j[i] \) (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 ⊥. 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...
Algorithm 2: Self-Stabilizing Labeling Algorithm; code for \( p_i \)

1. Variables:
   1. \( \text{max}[n] \) of (\( m, c_l \)): \( \text{max}[i] \) is \( p_i \)'s largest label pair, \( \text{max}[j] \) refers to \( p_j \)'s label pair (canceled when \( \text{max}[j]:c_l \neq \perp \)).
   2. \( \text{storedLabels}[\sum] \): an array of queues of the most-recently-used label pairs, where \( \text{storedLabels}[\sum] \) holds the labels created by \( p_j \in P \). For \( p_j \in (P \setminus \{p_i\}) \), \( \text{storedLabels}[\sum] \)'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 \( \text{storedLabels}[\sum] \)'s queue size is limited to \( (n(\sqrt{n} + m)) \) pairs. The operator \( \text{add}() \) adds \( lp \) to the front of the queue, and \( \text{emptyAllQueues()} \) clears all \( \text{storedLabels}[\sum] \) queues. We use \( \text{lp.remove()} \) for removing the record \( lp \in \text{storedLabels}[\sum] \). Note that an element is brought to the queue front every time this element is accessed in the queue.

2. Notation: Let \( y \) and \( y' \) be two records that include the field \( x \). We denote \( y =_x y' \equiv (y.x = y'.x) \).

3. Macros:
   - \( \text{legit}(lp) = (lp = (\perp, \perp)) \)
   - \( \text{labels}(lp) = \text{storedLabels}[lp.ml.Creator] \)
   - \( \text{double}(j, lp) = (3lp' \in \text{storedLabels}[j] : ((lp \neq lp') \land (lp.ml = ml) \lor (\text{legit}(lp') \land \text{legit}(lp)))) \)
   - \( \text{stateInfor}(j) = (3lp \in P, lp \in \text{storedLabels}[j] : (lp.ml \neq \perp) \lor \text{double}(j, lp)) \)
   - \( \text{recordDoesntExist}(j) = (3p, ml, P, ml) \notin \text{labels}(max[j])) \)
   - \( \text{notgeq}(j, lp) = \text{if } (3lp' \in \text{storedLabels}[j] : (lp.ml \neq ml) \text{ then } \text{return}(lp.ml) \text{ else return}(\perp) \) \)

4. \( \text{canceled}(lp) = \text{if } (3lp' \in \text{labels}(lp) : ((lp' = ml) \lor \text{notgeq}(lp')) \text{ then return}(lp') \) \)

5. \( \text{return}(\perp) \)

6. \( \text{needUpdate}(j) = (\text{~notgeq}(\text{max}[j]) \land (\text{max}[j].ml, \perp) \in \text{labels}() \text{)} \)

7. \( \text{legitLabels}() = (\text{max}[j].ml : 3p \in P \land \text{legit}(\text{max}[j])) \)

8. \( \text{useOwnLabel}() = \text{if } (3lp \in \text{storedLabels}[i] : \text{legit}(lp) \text{ then max}[i] \leftarrow lp \text{ else } \text{storedLabels}[i].\text{add}(\text{max}[i] \leftarrow (\text{nextLabel}(), \perp)) \text{ for every } lp \in \text{storedLabels}[i], \text{ we pass in } \text{nextLabel}() \text{ both } lp.ml \text{ and } lp.cl. \) \)

9. \( \text{upon receive}(\text{sentMax}, \text{lastSent}) \text{ from } p_j \)

10. \( \text{begin} \)

11. \( \text{max}[j] \leftarrow \text{sentMax}; \)

12. \( \text{if } \text{~notgeq}(\text{lastSent}) \land \text{max}[i] = ml \text{ lastSent then max}[i] \leftarrow \text{lastSent} \)

13. \( \text{if stateInfor() then storedLabels.emptyAllQueues()} \)

14. \( \text{foreach } p_j \in P : \text{recordDoesntExist}(j) \text{ do labels(max[j]).add(max[j]))} \)

15. \( \text{foreach } p_j \in P, lp \in \text{storedLabels}[j] : (\text{legit}(lp) \land \text{notgeq}(j, lp) \neq \perp) \text{ do } \)

16. \( \text{foreach } p_j \in P, lp \in \text{storedLabels}[max[j]] : (\text{~notgeq}(max[j]) \land (max[j].ml = ml) \lor \text{legit}(lp)) \text{ do } \)

17. \( \text{lp} \leftarrow \text{max}[j] \)

18. \( \text{foreach } p_j \in P, lp \in \text{storedLabels}[max[j]] : \text{double}(j, lp) \text{ do } \text{lp.remove()}; \)

19. \( \text{foreach } p_j \in P : (\text{legit}(\text{max}[j]) \land \text{canceled}(\text{max}[j]) \neq \perp) \text{ do } \)

20. \( \text{max}[j] \leftarrow \text{canceled}(\text{max}[j]); \)

21. \( \text{if } \text{legitLabels()} \neq \perp \text{ then max}[i] \leftarrow \text{max}_{\neq}(\text{legitLabels}()); \perp) \)

22. \( \text{else useOwnLabel}() \)

are already included in the label storage (\( \text{storedLabels}[\sum] \)), 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 \( lp'.ml \neq ml \) (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 \( \text{max}[i] \) 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 \( \text{max}[j] \) 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 $\max_i[i]$ (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 $\perp$ and the resulting label pair is added to both $\max_i[i]$ and $storedLabel_i[i]$.

4.2 Correctness proof

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

**Overview of the proof.** 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 4.1), and (2) $p_i$ either adopts or creates the $\preceq_{lb}$-greatest legitimate local label (Lemma 4.2). The proof then presents bounds on the number adoption steps (Lemmas 4.3 and 4.4), 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 4.5) and thus no new labels are being adopted or created (Lemma 4.6), which then implies the existence of a global maximal label (Lemma 4.7). Namely, there is a legitimate label $\ell_{\text{max}}$ such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in $R$), it holds that $\max_i[i] = \ell_{\text{max}}$. Moreover, for any processor $p_j \in P$ that is active throughout the execution, it holds that $p_i$’s local maximal label $\max_i[i] = \ell_{\text{max}}$ is the $\preceq_{lb}$-greatest of all the labels in $\max_i[i]$ and there is no label pair in $storedLabels_i[j]$ that cancels $\ell_{\text{max}}$, i.e., $((\max_i[j] \preceq_{lb} \ell_{\text{max}}) \land (\forall \ell \in storedLabels_i[j] : \text{legit}(\ell) \Rightarrow (\ell \preceq_{lb} \ell_{\text{max}})))$. 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 4.2).

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., \( \text{max}[i] \), \( \text{storedLabels}[i] \), 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”.

### 4.2.1 No stale information

Lemma 4.1 says that the predicate \( \text{staleInfo}() \) (line 9) can only hold during the first execution of the \( \text{receive}() \) event (line 17).

**Lemma 4.1** Let \( p_i \in P \) be a processor for which \( \neg \text{staleInfo}_i() \) (line 9) does not hold during the \( k \)-th step in \( R \) that includes the complete execution of the \( \text{receive}() \) event (from line 17 to 28). Then \( k = 1 \).

**Proof.** Since \( R \) starts in an arbitrary configuration, there could be a queue in \( \text{storedLabels}[i] \) 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, \( \text{staleInfo}_i() \) might hold during the first execution of the \( \text{receive}() \) event. When this is the case, the \( \text{storedLabels}[i] \) structure is emptied (line 21). During that \( \text{receive}() \) event execution (and any event execution after this), \( p_i \) adds records to a queue in \( \text{storedLabels}[i] \) (according to the creator identifier) only after checking whether \( \text{recordDoesntExist}() \) holds (line 22).

Any other access to \( \text{storedLabels}[i] \) 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 \( \text{receive}() \) (after the first), \( \text{staleInfo}_i() \) cannot hold.

Lemma 4.1 along with the lines 9 and 26 of the Algorithm, imply Corollary 4.1.

**Corollary 4.1** Consider a suffix \( R' \) of execution \( R \) that starts after the execution of a \( \text{receive}() \) 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, \( \ell_j =_{\text{creator}} j \) and (ii) \( \ell_j \) can only appear in \( \text{storedLabels}[i][j] \) and \( \text{max}[i][j] \) but not in \( \text{storedLabels}[i][k] : k \neq j \).  

### 4.2.2 Local \( \preceq_{lb} \)-greatest legitimate local label

Lemma 4.2 considers processors for which \( \text{staleInfo}() \) (line 9) does not hold. Note that \( \neg \text{staleInfo}_i() \) holds at any time after the first step that includes the \( \text{receive}() \) event (Lemma 4.1). Lemma 4.2 shows that \( p_i \) either adopts or creates the \( \preceq_{lb} \)-greatest legitimate local label and stores it in \( \text{max}[i] \).

**Lemma 4.2** Let \( p_i \in P \) be a processor such that \( \neg \text{staleInfo}_i() \) (line 9), and \( L_{\text{pre}}(i) = \{ \text{max}[i][j].ml : \exists p_j \in P \land \text{legit}(\text{max}[i][j]) \land (\exists (\text{max}[i][j].ml,x) \in (\text{labels}(\text{max}[i][j]) \setminus \{\text{max}[i][j]\}) \Rightarrow (x = \bot)) \} \) be the set of \( \text{max}[i][j] \)'s labels that, before \( p_i \) executes lines 21 to 28, are legitimate both in \( \text{max}[i][j] \) and in
storedLabels[\ell]'s queues. Let \( L_{post}(i) = \{ \max_i[j].ml : \exists p_j \in P \land \text{legit}(\max_i[j]) \} \) and \( (\ell, \bot) \) be the value of \( \max_{i}[i] \) immediately after \( p_i \) executes lines 21 to 28. The label \( (\ell, \bot) \) is the \( \leq_{lb} \)-greatest legitimate label in \( L_{post}(i) \). Moreover, suppose that \( L_{pre}(i) \) has a \( \leq_{lb} \)-greatest legitimate label, then that label is \( (\ell, \bot) \).

**Proof.** \( (\ell, \bot) \) is the \( \leq_{lb} \)-greatest legitimate label in \( L_{post}(i) \). Suppose that immediately before line 27, we have that \( \text{legitLabels}_i() \neq \emptyset \), where \( \text{legitLabels}_i() = \{ \max_i[j].ml : \exists p_j \in P \land \text{legit}(\max_i[j]) \} \) (line 14). Note that in this case \( L_{post}(i) = \text{legitLabels}_i() \). By the definition of \( \leq_{lb} \)-greatest legitimate label and line 27, \( \max_i[i] = (\ell, \bot) \) is the \( \leq_{lb} \)-greatest legitimate label in \( L_{post}(i) \). Suppose that \( \text{legitLabels}_i() = \emptyset \) immediately before line 27, i.e., there are no legitimate labels in \( \{ \max_i[j] : \exists p_j \in P \} \). By the definition of \( \leq_{lb} \)-greatest legitimate label and line 15, \( \max_i[i] = (\ell, \bot) \) is the \( \leq_{lb} \)-greatest legitimate label in \( L_{post}(i) \).

Suppose that \( \text{rec} = (\ell', \bot) \) is a \( \leq_{lb} \)-greatest legitimate label in \( L_{pre}(i) \), then \( \ell = \ell' \). We show that the record \( \text{rec} \) 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 \( \text{storedLabels}_i() \) and no records in \( \max_i[] \) become legitimate. Therefore, \( \text{rec} \) is also the \( \leq_{lb} \)-greatest legitimate label in \( L_{post}(i) \), and thus, \( \ell = \ell' \).

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

### 4.2.3 Bounding the number of labels

Lemmas 4.3 and 4.4 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 \( \text{storedLabels}_i[j] \) if \( i \neq j \), and \( 2(mn + 2n^2 - 2n) + 1 \) for \( \text{storedLabels}_i[i] \) 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 \( \leq_{lb} \)-greatest label among all the ones that the network ever delivers to \( p_j \). Lemma 4.3 bounds the number of labels from \( p_j \)'s domain that any processor \( p_i \in P \) adopts in \( R \).

**Lemma 4.3** 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
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, \( \ell_j \), for which \( \ell_j = \text{Creator } j \) (Corollary 4.1). Whenever \( p_i \) adopts a new label \( \ell_j \) from \( p_j \)'s domain (line 27) such that \( \ell_j : (\ell_j = \text{Creator } j) \), this implies that \( \ell_j \) is the only legitimate label pair in \( \text{storedLabels}_i[j] \). Since \( \ell_j \) was not transmitted by \( p_i \) before it was adopted, \( \ell_j \) 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 4.4 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 4.4** Let \( p_i \in P \) and \( L_i = \ell_{i0}, \ell_{i1}, \ldots \) be the sequence of legitimate labels, \( \ell_{ik} = \text{Creator } i \), from \( p_i \)'s domain, which \( p_i \) stores in \( \text{max}_i[i] \) 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 \( L_{i,j} = \ell_{i,j0}, \ell_{i,j1}, \ldots \) be the sequence of legitimate labels that \( p_i \) stores in \( \text{max}_i[j] \) during \( R \) and \( C_{i,j} = \ell_{i,j0}, \ell_{i,j1}, \ldots \) 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 \( \text{max}_i[i] \).

1. **When** \( \ell_{ik} = \ell_{j0,j'} \), **where** \( p_j, p_j' \in P \) **and** \( k \in \mathbb{N} \). This case considers the situation in which \( \text{max}_i[i] \) stores a label that appeared in \( \text{max}_j[j'] \) at the (arbitrary) starting configuration, (i.e. \( \ell_{j0,j'} \in L_{j,j'} \)). There are at most \( n(n-1) \) such legitimate label values from \( p_i \)'s domain, namely \( n - 1 \) arrays \( \text{max}_j[j] \) of size \( n \).

2. **When** \( \ell_{ik} = \ell_{j0,j'} = \ell_{j0,j''} \), **where** \( p_j, p_j' \in P, k, k' \in \mathbb{N} \) **and** \( \ell_{jk,j'} \neq \ell_{jk,j''} \). This case considers the situation in which \( \text{max}_i[i] \) stores a label that appeared in the communication channel between \( p_j \) and \( p_j' \) at the (arbitrary) starting configuration, (i.e. \( \ell_{j0,j'} \in C_{j,j'} \)) and appeared in \( \text{max}_j[j'] \) 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 \( |H| \).

3. **When** \( \ell_{ik} \) **is the return value of nextLabel()** (the else part of line 28). Processor \( p_i \) aims at adopting the \( \preceq_{\ell0} \)-greatest legitimate label that is stored in \( \text{max}_i[i] \), whenever such exists (line 27). Otherwise, \( p_i \) uses a label from its domain; either one that is the \( \preceq_{\ell0} \)-greatest legit label among the ones in \( \text{storedLabels}_i[i] \), whenever such exists, or the returned value of nextLabel() (line 28).

The latter case (the else part of line 28) refers to labels, \( \ell_{ik} \), that \( p_i \) stores in \( \text{max}_i[i] \) only after checking that there are no legitimate labels stored in \( \text{max}_i[i] \)
or \text{storedLabels}_i[i]. Note that every time \( p_i \) executes the else part of line 28, \( p_i \) stores the returned label, \( \ell_{ik} \), in \text{storedLabels}_i[i]. After that, there are only three events for \( \ell_{ik} \) not to be stored as a legitimate label in \text{storedLabels}_i[i]: (i) execution of line 21, (ii) the network delivers to \( p_i \) a label, \( \ell' \), that either cancels \( \ell_{ik} \) or for which \( \ell' \not\preceq \ell_{ik} \), and (iii) \( \ell_{ik} \) overflows from \text{storedLabels}_i[i] after exceeding the \((n(n^2 + m) + 1)\) limit which is the size of the queue.

Note that Lemma 4.1 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 \( \preceq \) greater-equal than all the ones in \text{storedLabels}_i[i] (by the definition of \text{nextLabel}() in Algorithm 1).

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

Note that \text{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 \text{storedLabels}_j[i] structures of some other processor \( p_j \). By Lemma 4.3 we know that \(|\text{storedLabels}_i[i]| \leq n + m\) 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 \text{remoteLabels}_i as follows: for \( p_j \in P : j \neq i \) we have that \(|\text{remoteLabels}_i[i]| = (n - 1)(|\text{max}[\bullet]| + |\text{storedLabels}_j[i]|) + |\mathcal{H}| = (n - 1)(n + (n + m)) + m = mn + 2n^2 - 2n. \) Since \( p_i \) may respond to each of these labels with a call to \text{nextLabel}(), we require that \text{storedLabels}_i[i] has size \( 2|\text{remoteLabels}_i[i]| + 1 \) label pairs in order to be able to accommodate all the labels from \text{remoteLabels}_i[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 \text{remoteLabels}_i, stops happening before overflows (event (iii)) occurs, since \text{storedLabels}_i[i] has been chosen to have a size that can accommodate all the labels from \text{remoteLabels}_i and those created by \( p_i \) as a response to these. This size is \( 2(mn + 2n^2 - 2n) + 1 \) which is \( O(n^3) \).

### 4.2.4 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 4.5) and thus no new labels are
being adopted or created (Lemma 4.6), which then implies the existence of a global maximal label (Lemma 4.7).

Lemmas 4.5 and 4.6, as well as Lemma 4.7 and Theorem 4.2 assume the existence of at least one processor, $p_{\text{unknown}} \in P$ 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 $p_{\text{unknown}}$ 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 3).

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 $\ell$ as its local maximal, and there exists another label pair $\ell'$ such that $\ell' \not\preceq_{lb} \ell$ and they have the same creator $p_i$. Algorithm 2 suggests only two possible routes for some label pair $\ell'$ to find its way in the system through $p_j$. Either by $p_j$ adopting $\ell'$ (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 $\ell'$ 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 $\ell$ and $\ell'$ in the system.

There is a risk for two label pairs from $p_i$’s domain, $\ell_j$ and $\ell_k$, 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 $\text{risk}_{i,j,k}(\ell_j, \ell_k) = (\ell_j = i, \ell_k) \land \text{legit}(\ell_j) \land (\text{notGreater}(\ell_j, \ell_k) \lor \text{canceled}(\ell_j, \ell_k))$ 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, $\ell_k$, from $p_i$’s domain that $p_k$’s state encodes, where $\text{canceled}(\ell_j, \ell_k) = (\text{legit}(\ell_j) \land \neg \text{legit}(\ell_k) \land \ell_j =_{ml} \ell_k)$ refers to a case in which label $\ell_j$ is canceled by label $\ell_k$, $\text{notGreater}(\ell_j, \ell_k) = (\text{legit}(\ell_j) \land \text{legit}(\ell_k) \land \ell_j \not\preceq_{lb} \ell_k)$ that refers to a case in which label $\ell_k$ is not $\preceq_{lb}$-greater than $\ell_j$ and $(\ell_j = i, \ell_k) \equiv (\ell_j =_{lcreator} \ell_k =_{lcreator} i)$.

These two label pairs, $\ell_j$ and $\ell_k$, can be the ones that processors $p_j$ and $p_k$
name as their local maximal label, as in $\text{max}_{i,j,k} = \{ (\text{max}_j[j], \text{max}_k[k]) \}$, or recently received from one another, as in $\text{ack}_{i,j,k} = \{ (\text{max}_j[j], \text{max}_k[j]) \}$. These two cases also appear when considering the communication channel (or buffers) from $p_k$ to $p_j$, as in $h\text{Name}_{i,j,k} = \{ (\ell_j, \ell_k) : \ell_j = \text{max}_j[j] \land (\exists (\ell_k, \bullet) \in \mathcal{H}_{k,j}) \}$ and $h\text{Ack}_{i,j,k} = \{ (\ell_j, \ell_k) : \ell_j = \text{max}_j[k] \land (\exists (\bullet, \ell_k) \in \mathcal{H}_{k,j}) \}$. 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 $\text{stored}_{i,j,k} = \{ (\text{max}_j[j]) \times \text{storedLabels}_k[i] \}$ We define the union of these cases to be the set $\text{risk} = \{ (\ell_j, \ell_k) \in \text{max}_{i,j,k} \cup \text{ack}_{i,j,k} \cup \text{Name}_{i,j,k} \cup \text{Ack}_{i,j,k} \cup \text{stored}_{i,j,k} : \exists p_i, p_j, p_k \in P \land \text{stopped}_j \land \text{stopped}_k \land \text{risk}_{i,j,k}(\ell_j, \ell_k) \}$, where $\text{stopped}_i = \text{true}$ when processor $p_i$ is inactive (crashed) and $\text{false}$ otherwise. The above notation can also be found in Table 1.

**Lemma 4.5** Suppose that there exists at least one processor, $p_{\text{unknown}} \in P$ whose identity is unknown, that takes practically infinite number of steps in $R$ during a period where $p_j$ never adopts labels (line 27), $\ell_j : (\ell_j = !\text{creator} i)$, from $p_i$’s unknown domain ($\ell_j \notin \text{labels}_j(\ell_j)$). Then eventually $\text{risk} = \emptyset$.

**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 $(\ell_j, \ell_k) \in \text{risk} \neq \emptyset$. We use $\text{risk}$’s definition to study the different cases. By the definition of $\text{risk}$, 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. $\text{stopped}_j = \text{stopped}_k = \text{False}$, then $\text{risk} \neq \emptyset \iff \text{risk}_{i,j,k} = \text{True}$. This means that there exist two label pairs $(\ell_j, \ell_k)$ where $\ell_k$ can force a cancellation to occur. Then the only way for this two labels to force $\text{risk} \neq \emptyset$ is if, throughout the execution, $\ell_k$ never reaches $p_j$.

The above claim is verified by a simple observation of the algorithm. If $\ell_k$ 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 $\text{storedLabels}_j[i]$. The latter is also ensured by the bounds on the labels given in Lemmas 4.3 and 4.4 that do not allow queue overflows. Thus to include these two labels to $\text{risk}$, is to keep $\ell_k$ hidden from $p_j$ throughout $R$. We perform a case-by-case analysis to show that it is impossible for label $\ell_k$ to be “hidden” from $p_j$ for an infinite number of steps in $R$.

**The case of $(\ell_j, \ell_k) \in \text{Name}_{i,j,k}$.** This is the case where $\ell_j = \text{max}_j[j]$ and $\ell_k$ is a label in $\mathcal{H}_{k,j}$ that appears to be $\text{max}_k[k]$. 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 $\text{transmit()} – \text{receive()}$ events of the two ends. Thus $\ell_k$, 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 $\text{risk}$ and the result follows.

**The case of $(\ell_j, \ell_k) \in \text{Ack}_{i,j,k}$.** This is the case where $\ell_j = \text{max}_j[j]$ and
\(\ell_k\) 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 \((\ell_j, \ell_k) \in \text{ack}_{i,j,k}\).

**The case of \((\ell_j, \ell_k) \in \max_{i,j,k}\).** Here the label pairs \(\ell_j\) and \(\ell_k\) 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 \(\ell_k\) 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 \((\ell_j, \ell_k) \in \text{ack}_{i,j,k}\).** This is the case where the labels \((\ell_j, \ell_k)\) 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 \((\ell_j, \ell_k) \in \text{stored}_{i,j,k}\).** 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 \(\text{risk}_{i,j,k}(\ell_j, \ell_k)\) holds implies that one of the tests in lines 23 and 26 will either update \(\text{storedLabels}_{k[i]}\), and respectively, \(\max_k[j]\) with canceling values. We note that for the latter case we argue that \(p_j\) receives \(\ell_k\) infinitely often. By the Claim the canceling takes place, and 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 \(\ell_k\) hidden from \(p_j\) throughout \(R\). This is a contradiction to our initial assumption, and thus eventually risk = \emptyset.

These two label pairs, \(\ell_j\) and \(\ell_k\), 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 \(\text{ack}_{i,j,k} = \{(\max_j[j], \max_k[j])\}\). These two cases also appear when considering the communication channel (or buffers) from \(p_k\) to \(p_j\), as in \(h\text{Name}_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = \max_j[j] \land (\exists (\ell_k, \bullet) \in \mathcal{H}_{k,j})\}\) and \(h\text{Ack}_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = \max_j[j] \land (\exists (\bullet, \ell_k) \in \mathcal{H}_{k,j})\}\). 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 \(\text{stored}_{i,j,k} = \{(\max_j[j]) \times \text{storedLabels}_{k[i]}\}\).

**Lemma 4.6** Suppose that risk = \emptyset in every configuration throughout \(R\) and that there exists at least one processor, \(p_{\text{unknown}} \in P\) whose identity is unknown, that takes practically infinite number of steps in \(R\). Then \(p_j\) never adopts labels (line 27), \(\ell_j : (\ell_j = \text{Creator} i)\), from \(p_i\)'s unknown domain \((\ell_j \notin \text{labels}_{j}(\ell_j))\).

**Proof.** Note that the definition of risk considers almost every possible combination of two label pairs \(\ell_j\) and \(\ell_k\) 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 \((\ell_j, \ell_k) \in \text{storedLabels}_{j[i]} \times \text{storedLabels}_{k[i]}\). 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 \(\ell_j\), and respectively, \(\ell_k\), 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 \( \ell_j \) to be added to \( \text{max}_j[j] \) and thus make \( \text{risk} \neq \emptyset \) 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 \( \text{max}_k[j] \). This is when it finds a legit label \( \ell_j \in \text{max}_k[j] \) that can be canceled by some other label \( \ell_k \) in storedLabels\(_k[i] \), line 26. But this is the case of having the label pair \((\ell_j, \ell_k)\) in stored\(_i,j,k\). Our assumption that \( \text{risk} = \emptyset \) implies that stored\(_i,j,k = \emptyset \). This is a contradiction. Thus a label \( \ell_k \) cannot reach \( \text{max}_k[] \) 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, \( H_{i,k} \times H_{k,j} \) and that \( \text{risk} = \emptyset \) throughout \( R \).

**Lemma 4.7** Suppose that \( \text{risk} = \emptyset \) in every configuration throughout \( R \) and that there exists at least one processor, \( p_{\text{unknown}} \in P \) whose identity is unknown, that takes practically infinite number of steps in \( R \). There is a legitimate label \( \ell_{\text{max}} \), such that for any processor \( p_i \in P \) (that takes a practically infinite number of steps in \( R \)), it holds that \( \text{max}_{i}[i] = \ell_{\text{max}} \). Moreover, for any processor \( p_j \in P \) (that takes a practically infinite number of steps in \( R \)), it holds that \((\forall \ell \in \text{storedLabels}_i[j] : \text{legit}(\ell)) \Rightarrow (\ell \preceq \ell_{\text{max}})\).

**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 \( \text{max}_{i}[i] \) and \( \text{max}_{j}[j] \) an infinite number of times. By the assumption that \( \text{risk} = \emptyset \), 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 \( \text{max}_{i}[i] \) or \( \text{max}_{j}[j] \). Hence, any differences in the local maximal label of the processors must be due to the labels’ iCreator difference.

Since \( \text{max}_{i}[i] \) and \( \text{max}_{j}[j] \) are continuously exchanged and received, assuming \( \text{max}_{i}[i] \prec_{ib} \text{max}_{j}[j] \) where the labels are of different label creators, then \( p_i \) will be led to a receive() event of \( \text{sentMax}_{j}, \text{lastSent}_{j} \) where \( \text{max}_{i}[i] \prec_{ib} \text{sentMax}_{j} \). By line 19, \( \text{sentMax}_{j} \) is added to \( \text{max}_{i}[j] \) and since \( \text{risk} = \emptyset \) no action from line 20 to line 26 takes place. Line 27 will then indicate that the greatest label in \( \text{max}_{i}[i] \) is that in \( \text{max}_{j}[j] \) which is then adopted by \( p_i \) as \( \text{max}_{i}[i] \), 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 \( \ell_{\text{max}} \) label, i.e., it holds that \((\forall \ell \in \text{storedLabels}_i[j] : \text{legit}(\ell)) \Rightarrow (\ell \preceq \ell_{\text{max}})\).

**4.2.5 Convergence**

Theorem 4.2 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 4.2** Suppose that there exists at least one processor, \( p_{\text{unknown}} \in P \) 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 \( \ell_{\text{max}} \), 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 \( \max_i[\ell_{\text{max}}] = \ell_{\text{max}} \). Moreover, for any processor \( p_j \in P \) (that takes a practically infinite number of steps in \( R \)), it holds that

\[
((\max_i[j] \preceq_{lb} \ell_{\text{max}}) \wedge (\forall \ell \in \text{storedLabels}_i[j] : \text{legit}(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\text{max}})).
\]

**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 \( \ell_{i_k} = \text{creator}_i \) (Lemma 4.4). Thus, by the pigeonhole principle we know that, within a bounded number of steps in \( R \), there is a period during which \( p_{\text{unknown}} \) takes a practically infinite number of steps in \( R \) whilst (all processors) \( p_i \) do not add any label pair, \( \ell_{i_k} = \text{creator}_i \), to the system configuration (the else part of line 28).

During this practically infinite period (with respect to \( p_{\text{unknown}} \)), 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 takes any number of (bounded or practically infinite) steps in \( R \), and processor \( p_k \in P \) that adopts labels in \( R \) (line 27), \( \ell_j : (\ell_j = \text{creator}_j) \), from \( p_j \)'s unknown domain (\( \ell_j /\in \text{storedLabels}_k(j) \)) it holds that \( p_k \) adopts such labels (line 27) only a bounded number times in \( R \) (Lemma 4.3). Therefore, we can again follow the pigeonhole principle and say that there is a period during which \( p_{\text{unknown}} \) takes a practically infinite number of steps in \( R \) whilst neither \( p_i \) adds a label, \( \ell_{i_k} = \text{creator}_i \), to the system (the else part of line 28), nor \( p_k \) adopts labels (line 27), \( \ell_j : (\ell_j = \text{creator}_j) \), from \( p_j \)'s unknown domain (\( \ell_j /\in \text{labels}_k(\ell_j) \)).

We deduce that, when the above is true, then we have reached a configuration in \( R \) where \( \text{risk} = \emptyset \) (Lemma 4.5) and remains so throughout \( R \) (Lemma 4.6). Lemma 4.7 concludes by proving that, whilst \( p_{\text{unknown}} \) 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 which the theorem statement specifies. Thus no label \( \ell = \text{creator}_j \) in \( \max_i[\bullet] \) or in \( \text{storedLabels}_i[j] \) may satisfy \( \ell \preceq_{lb} \ell_{\text{max}} \).

4.3 Increment Counter Algorithm

In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection to obtain a practically self-stabilizing counter increment algorithm.

**Counters.** To achieve this task, we now need to work with practically unbounded counters. As already mentioned in Section 2, a counter \( \text{cnt} \) is a triplet \((\text{lbl}, \text{seqn}, \text{wid})\), where \( \text{lbl} \) is an epoch label as defined in the previous subsection,
seqn is a 64-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 \(\text{cnt}_i, \text{cnt}_j\) we define the relation \(\text{cnt}_i \prec_{ct} \text{cnt}_j\) as \([\text{cnt}_i, \text{lbl}] \prec_{lb} [\text{cnt}_j, \text{lbl}] \lor ([\text{cnt}_i, \text{seqn}] = [\text{cnt}_j, \text{seqn}] \land [\text{cnt}_i, \text{seqn}] < [\text{cnt}_j, \text{seqn}] \lor ([\text{cnt}_i, \text{seqn}] = [\text{cnt}_j, \text{seqn}] \land [\text{cnt}_i, \text{wid}] < [\text{cnt}_j, \text{wid}])\). Observe that when the labels of the two counters are incomparable, the counters are also incomparable.

Therefore, the relation \(\prec_{ct}\) defines a total order (as required by practically unbounded counters) only when processors share a globally maximal label, (i.e., the system runs within a “stable” epoch). As we have shown in Theorem 4.2, starting from an arbitrary configuration, we eventually reach a configuration where the active processors have adopted the same maximal label. Essentially, the counter increment algorithm enhances the labeling algorithm to take care of the counter increment once such a maximal label exists in the system.

Enhancing the labeling algorithm to handle counters. Recall that in the labeling algorithm each processor \(p_i\) was maintaining two main structures of pairs of labels: array \(\max[]\) that stored the local maximal labels of each other processor (based on the message exchange) and \(\text{storedLabels}[]\), an array of queues of label pairs that each processor maintains in an attempt to clean up obsolete labels created by itself or other processors. These structures now need to contain counters instead of just labels and are renamed to \(\maxC[]\) and \(\text{storedCnts}[]\) (see line 1 of Algorithm 3). Each label can yield many different counters with different \(\langle \text{seqn}, \text{wid} \rangle\). Therefore, in order to avoid increasing the size of these queues (with respect to the number of elements stored), we only keep the highest sequence number observed for each label (breaking ties with wid). We denote a counter pair by \(\langle \text{mct}, \text{cct} \rangle\), with this being the extension of a label pair \(\langle \text{ml}, \text{cl} \rangle\), where \(\text{cct}\) is a canceling counter for \(\text{mct}\), such that either \(\text{cct}.\text{lbl} \not\prec \text{lb} \text{mct}.\text{lbl}\) (i.e., the counter is canceled), or \(\text{cct}.\text{lbl} = \perp\).

Also, note that if there are counters in the system that are corrupt (being in the initial arbitrary configuration), then they can only force a change of label if their sequence number is exhausted (i.e., \(\text{seqn} \geq 2^{64}\)). Exhausted counters are treated by the counter algorithm in a way similar to the canceled labels in the labeling algorithm; an exhausted counter \(\text{mct}\) in a counter pair \(\langle \text{mct}, \text{cct} \rangle\) is canceled, by setting \(\text{mct}.\text{lbl} = \text{cct}.\text{lbl}\) (i.e., the counter’s own label cancels it) and hence making the counter non-legit (thus it cannot be used as a local maximal counter in \(\maxC_i[]\)). This cannot increase the number of labels that are created due to the initially corrupted ones, as shown in the correctness proof that follows.

Another issue worth mentioning, is that the system is allowed to revert back to a previous legit label \(x\), in case the current maximal label \(y\) becomes canceled. Label \(x\) might have been used before to create counters, so it is required to store the last sequence number written. If \(x\) is legit the system should not propose a new label and instead revert to \(x\). Otherwise, the queues might grow with no bound. We enable reverting to such an \(x\), by imposing
that each processor only stores a single instance of counters with the same label inside \( \text{storedCnts}[] \), namely the one with the maximal sequence number \( (\text{seqn}, \text{wid}) \). This is performed by storing the highest value of a counter that we hear about, as performed in line 19 upon a successful quorum write of a new sequence value, upon a receipt of any write request (line 31) and in every receipt of a counter through \( \text{receive}() \) by the definition of \( \text{process}() \). Namely, in every possible appearance of a counter to the local state of a processor.

Quorums. We define a quorum set \( Q \) 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 \( Q_i, Q_j \in Q \) such that \( Q_i, Q_j \subseteq P \), 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.

Description of the Counter Algorithm. A pseudocode of the counter increment algorithm appears in Algorithm 3. The algorithm shows periodic counter operations (lines 12–14) –extending those of the labeling algorithm–and the counter increment operations (lines 15–31). The algorithm uses the enhanced counter structures \( \text{maxC}[n] \) and \( \text{storedCnts}[n] \) which are maintained in the same way as in the labeling algorithm with some additional operations. We define the operator \( \text{enqueue}(ctp) \) (line 3) to add a counter pair \( ctp \) to a queue of these structures if a corresponding counter with the same \( \text{lbl} \) doesn’t exist, or to keep only one of the two instances if it exists. There are two enqueuing rules: (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 \( \langle \text{seqn}, \text{wid} \rangle \). The counter is placed at the front of the queue.

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 \( \text{maxC}[j] \) (line 12). Upon receipt of such an update from \( p_j \), \( p_i \) first performs canceling of any exhausted counters in \( \text{storedCnts}[] \) (line 14), in \( \text{maxC}[] \) (line 14) and in the received couple of counter pairs (line 14). Having catered for exhaustion, it then calls \( \text{process}(\bullet, \bullet) \) with the received two counter pairs as arguments.

The \( \text{process}() \) operator calls lines 19 to 28 of Algorithm 2 adjusted for counter structures and handling counters. Thus, mentions to either labels or label structures in the labeling algorithm now refer to counters and counter structures. When adding to the counter queues the two enqueuing rules mentioned for \( \text{enqueue}() \) (above) hold. For ease of presentation we assume that
Algorithm 3: Counter Increment; code for $p_i$

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 $\langle cct, ct \rangle$ extends a label pair. $ct$ is a canceling counter for $cct$, such that $ct.lbl \neq lbl.mct.lbl$ or $ct.lbl = \perp$. We rename structures $max[]$ and $storedLabels[]$ of Alg. 2 to $maxC[]$ and $storedCnts[]$ that hold counter pairs instead of label pairs.

Operators: $process\langle \bullet, \bullet \rangle$ - executes the lines 19 to 28 of Algorithm 2 adjusted for counter structures and handling counters. For counter pairs with the same $mct$ label, only the instance with the greatest counter w.r.t. $\prec_c$ is retained. In the case where one counter is cancelled we keep the cancelled. For ease of presentation we assume that a counter with a label created by $p_i$ in line 28 of Algorithm 2, is initiated with a $seqn = 0$ and $wid = i$. A call of $process()$ (without arguments) essentially ignores lines 19 and 20 of Alg. 2.

$enqueue(cpt)$ - places a counter pair $cpt$ at the front of a queue. If $cpt.mct.lbl$ already exists in the queue, it only maintains the instance with the greatest counter w.r.t. $\prec_c$, placing it at the front of the queue. If one counter pair is canceled then the canceled copy is the one retained.

Notation: Let $y$ and $y'$ be two records that include the field $x$. We denote $y =_x y' \equiv (y.x = y'.x)$.

Macros:

1. $exhausted(cpt) = (cpt.mct.seqn \geq 2^{64})$
2. $legit(cpt) = (cpt.cct = \perp)$
3. $retCntrQ(ct) : return\ (storedCnts[ct.lbl.\text{Creator}])$
4. $legitCounters() = \{maxC[i].mct: 3p_i \in P \land legitim(maxC[j])\}$
5. $cancelExhausted(cpt) : ctp.cct \leftarrow ctp.mct$
6. $cancelExhaustedMaxC() : \text{foreach}\ p_j \in P, c \in maxC[j] : exhausted(c)\ \text{do}\$ $cancelExhausted(cpt)\ \text{with}\ (ctp : ctp.mct ; \text{in\} legitimCounters() \land maxC[i] = mct.lbl\ ctp))$;
7. $\langle seqn, \perp \rangle$ that hold counter pairs instead of label pairs.

Operators:

1. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
2. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
3. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
4. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
5. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
6. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
7. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
8. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
9. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
10. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
11. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
12. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
13. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
14. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
15. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
16. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
17. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
18. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
19. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
20. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
21. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
22. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
23. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
24. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
25. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
26. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
27. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
28. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
29. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
30. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$
31. $\text{begin}\$ $\langle seqn, \perp \rangle = \langle seqn, \perp \rangle$

26
a counter with a label created by \( p_i \) in line 28 of Algorithm 2, is initiated
with a \( \text{seqn} = 0 \) and \( \text{wid} = i \). A call to \( \text{process()} \) (without arguments) essentially ignores lines 19 and 20 of Algorithm 2 and executes the rest of the lines performing bookkeeping tasks. After this call to \( \text{process()} \), any exhausted counters from the initial arbitrary configuration, are enqueued as canceled to \( \text{storedCnts}[] \). Therefore, they can never be readopted in case they are proposed with a non-exhausted counter.

The increment counter algorithm executed in lines 15 to 19 follows the logic of a writer in a MWMR register emulation. Processor \( p_i \) inquires the system for the counter they believe as greatest (line 16) by calling procedure \( \text{quorumRead()} \) (lines 20–22). The responses contain the counter (\( \text{max}_j \)) that the responding processor \( p_j \) regards as the greatest (line 23). \( p_i \) aggregates the responses in its \( \text{maxC}[] \) array. Note that there can be background counter diffusion as well. The \( \text{quorumRead()} \) returns only when all the processors of one of the quorums have sent their responses (excluding responses from diffusion).

When the \( \text{quorumRead()} \) completes, the \( \text{findMaxCounter()} \) procedure is called repeatedly until a counter that is not canceled or exhausted is found; all counters that are exhausted must eventually become canceled. The function \( \text{findMaxCounter()} \) cancels any exhausted counters in \( \text{maxC}[] \) (while it holds the input from the quorum), and then calls \( \text{process()} \) (line 25) to perform bookkeeping based on the new information and to provide a valid label. When the system is stabilized this label should not change. Any corrupt exhausted counter that might not have been canceled in the \( \text{storedCnts}[] \) will, through the new call on \( \text{process()} \), become canceled, making \( p_i \) immune from adopting it if it is proposed by other processors as valid. The \( \text{getMaxSeq}() \) macro returns the maximal per \( \prec_{ct} \), legit, non-exhausted counter it finds locally inside \( \text{maxC}_i[] \).

On exiting the loop (lines 17–17), the counter in \( \text{maxC}_i[i] \) is the greatest of the counters returned by the quorum and any other processor (through diffusion), or, in case such a counter was not found, it is a newly created counter. As already stated such a counter is initiated to \( \text{seqn} = 0 \) and \( \text{wid} = i \).

Following this, a local copy of \( \text{maxC}_i[i] \) is incremented, i.e., the sequence number is increased by one, and \( \text{wid} \) is set to the identifier of \( p_i \) (line 17). The processor then attempts a write to the system (line 18) expecting responses from a quorum to return (line 27). Every processor \( p_j \) receiving \( p_i \)'s quorum write request, places it in \( \text{maxC}_j[i] \) if it is greater than the value it already has in \( \text{maxC}_i[j] \) and cancels it if it is exhausted. If the write fails for any reason to gather acknowledgments, the value does not get written to the local state as it does not satisfy the \( \text{if} \) condition of line 18.

Proof of correctness. We now prove the correctness of the counter algorithm. Initially we prove, that starting from an arbitrary configuration the system eventually reaches to a global maximal label (as given in Theorem 4.2), even in the presence of exhausted counters. We then continue to show that given such a global maximal label, the related counters are guaranteed to increment monotonically.
Lemma 4.3 Consider two processors $p_i$ taking a practically infinite number of steps and a setting as described by Theorem 4.2, adjusted for labels rather than counters as described above. Algorithm 3 guarantees that, within a bounded number of steps, every processor $p_i$ holds a counter $ct$ in $\maxC_i[i]$ that has $ct.lbl = \ell_{\text{max}}$ the globally maximal label and $\ell_{\text{max}}$ is not exhausted. Moreover, $\ell_{\text{max}}$ is the greatest of all legitimate counter pair labels in $\maxC_i[i]$ and $\text{storedCnt}_i[]$.

Proof. The proof follows the flow of the labeling algorithm proof, and provides minor amendments wherever the use of counters (instead of labels) challenges the correctness of the arguments. We show how the counter operations ensure that we reach to the globally maximal label $\ell_{\text{max}}$ becoming adopted by all the processors that take a practically infinite number of steps in execution $R$. We only require that $lbl = \ell_{\text{max}}$ while $\text{seqn}$ and $\text{wid}$ may differ.

Key observation. Upon a receive event (lines 13–14) of the increment counter algorithm, lines 14, 14 and 14 cancel any exhausted counter pairs appearing as legitimate in $\text{storedCnt}_i[]$, $\maxC[i]$ and among the two received counter pairs by setting their $mct$ as their $cct$. Increment counter procedures also have incoming counters. We note that any exhausted non-canceled counters stored in $\maxC_i[i]$ by a $\text{quorumRead}()$, are canceled by the immediate call of $\text{cancelExhaustedMaxC}()$ in line 25 (through the call on $\text{findMaxCnt}()$ of line 17). Incoming counters through $\text{quorumWrite}()$ are also immediately checked for exhaustion on line 31.

In line with Lemma 4.1 we require that a full execution of a receive event has taken place, i.e., all lines 13 to 14 have been executed at least once. We now prove that all lemmas up to Lemma 4.4 in the labeling scheme’s correctness proof remain unaltered if we extend labels to counters and assume that the arbitrary state contains exhausted counters. The case of adopting an exhausted label which is then canceled, is an additional case in the body of the proof of Lemma 4.4 since all the other assumptions remain the same. Consider some processor $p_i \in P$ taking an infinite number of steps in execution $R$ and assigning the label $\ell_x$ of an exhausted counter $ct_x$ as $\maxC_i[i]$. This implies that $\ell_x$ was not canceled when line 27 of Algorithm 2 was executed. By our key observation, any counter in the local state is checked for exhaustion and canceled immediately. By the assumption that at least one iteration of $\text{receive}$ has taken place, we deduce that $\ell_x$ was adopted while canceled contradicting the conditions of line 27 of Algorithm 2 and the labeling algorithm proof. Thus, after a single iteration of $\text{receive}$ it is impossible to adopt an exhausted label.

Exhausted counters cannot therefore increase adoptions and they pose no requirement for increasing the counter queue size, since we only keep a single instance of this canceled object. We note that once the canceling operations on exhausted counters take place, the call to $\text{process}$ ensures that the canceled copies of these counters are retained in the $\text{storedCnt}_i[]$. Any new occurrences of these counter labels in $\maxC[i]$ are canceled by the corresponding canceled copies in $\text{storedCnt}_i[]$. From the arguments for label pair diffusion, which are identical for the counter pairs being diffused, any processor holding a counter $ct_x$ as its
local maximal counter that is exhausted in the local state of some other active processor $p_j$, eventually stops using $ct_x$ in favor of a counter with a different non-exhausted label. Following the results of the labeling algorithm, we deduce that our cancellation policy on the exhausted counters, enables Theorem 4.2 to also include the use of counters without any need to locally keep more counters than there are labels. By this theorem, we deduce that, eventually, any processor taking a practically infinite number of steps in $R$ will have a counter with the globally maximal label $\ell_{\text{max}}$. ■

**Theorem 4.4** 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.** Given a suffix $R'$ of the execution $R$ in which Lemma 4.3 holds throughout, we define $ct_{\text{max}}$ to be the counter with the globally maximal label that is the greatest in the system with respect to $\langle \text{seqn, wid} \rangle$. There are two cases:

**Case 1:** $ct_{\text{max}}$ is the result of a call to the incrementCounter() procedure. Since this procedure only returned when $\text{quorumWrite}(ct_{\text{max}})$ took place (line 28), therefore a quorum acknowledged the writing of this value. By the intersection property of the quorums, this counter was made known to at least one processor of every quorum. If there are concurrent writings of counters with the same seqn then the one with the greatest wid ensures monotonicity. Any subsequent call to incrementCounter() and thus to quorumRead() will, again by the intersection property of the quorums, return at least one instance of $ct_{\text{max}}$, since there is at least one processor in every quorum that acknowledged this counter.

**Case 2:** $ct_{\text{max}}$ comes from the arbitrary state. By Lemmas 4.5, 4.6 and 4.7, the risk of having a label that remains hidden and that can cause a cancellation eventually becomes zero. We have previously used this proof to enforce that all exhausted counters eventually become canceled or are eliminated from the system. In the same vein we treat the case where $ct_{\text{max}}$ is a remote counter that was not written to a quorum but may be revealed at some point to the system. Note that such a counter has the global maximal label and can indeed be adopted as a highest counter, since the adoption of this counter does not violate the monotonicity of counters, even if we go from one sequence number to a much greater one.

We also note, that this counter may have a sequence number near exhaustion. By the arguments of Case 1, the increments after this counter is adopted are monotonic and this will cause exhaustion of the counter requiring a label change in a number of increment steps that is not practically infinite. We have to mention here that this event does not increase the number of label creations, as the number of such counters that can cause eventual cancellation by exhaustion (after not practically infinite counter increments) is accounted for in the number of labels that can exist in the initial arbitrary state. The proof follows from our treatment of exhausted counters of Lemma 4.3.

Recall that our algorithm allows processor $p_i$ to readopt a counter $cnt_i$ with $p_i$’s own label that has a different label creator than the one it used in the
previous iteration of the labeling algorithm. Readoptions are only possible when cnt\textsubscript{i} has not been canceled. In the case of such a readoption it is implied that cnt\textsubscript{i} was dropped in favor of a counter cnt' with higher a lCreator identifier that was eventually canceled. This implies that cnt' must come from the initial arbitrary configuration. Hence these “breaks” in monotonicity can only occur a bounded number of times in the execution, since counters such as cnt' are in number and are handled by the Labeling algorithm.

Our algorithm stores every incoming counter with a label that was created by \( p_i \) in the \( storedCnts_i[i] \) queue and by keeping the instance with the greatest \( \langle seqn, wid \rangle \), (see lines 14, 19 and 31). So if \( p_i \) is to backstep to cnt\textsubscript{i}, then the greatest instance that \( p_i \) has learned about cnt\textsubscript{i} is adopted from \( storedCnts_i[i] \). The only way for a new value of cnt\textsubscript{i} to be missed by \( p_i \), is for \( p_i \) to not hear of a quorum read incrementing \( p_i \) before cnt' was adopted. Again, as explained above, this is attributed to the bounded number of remnant counters from the arbitrary configuration that are dealt by the Labeling and Counter algorithms as Lemma 4.3 describes.

Now, under a legal execution where Lemma 4.3 holds, Case 2 can only occur a bounded number of times (since the counters in the initial arbitrary state are bounded in number). Furthermore, Case 1 is eventually true for the rest of the execution. In any case, the increment of the counter is monotonic with respect to \( \prec_{ct} \) in every subsequent call to \textit{incrementCounter}().

Having a self-stabilizing counter increment algorithm, it is not hard to implement a 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 \( \bot \) 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 standard requirement for preserving the consistency of the register (c.f. [3, 19]).

5 Virtually Synchronous Stabilizing Replicated State Machine

We now present a self-stabilizing reliable multicast algorithm that provides fault-tolerance, with respect to processor crashes and communication asynchrony, by considering the \( (current\ group) \) view of the changing processor set at the end of the group communication endpoint. We propose a self-stabilizing algorithm that guarantees the VS property. Namely, any two processors that are members of the same view, ought to deliver identical message sets to their SMRs as long as
they continue to share the same view, which indeed may change. This way, SMR algorithms can use the multicast service to synchronize their state transitions, i.e., the group members multicast their current automaton state, and the last received input that had led to that state.

**Overview.** A key advantage of multicast services (with virtual synchrony) is the ability to reuse the same view during many multicast rounds, and thus every automaton step requires a single multicast round. The aim of the proposed algorithm is to demonstrate in a self-stabilizing manner the most important ways to cut down the number of times in which the service needs to agree on a new view, and when it does, to perform it swiftly. Similar to [6], we assume that the service works in the network’s primary partition (see Definition 5.1) and require that a majority of processors are present in every view set. We do not however require all (local) failure detectors to agree on the set of recently alive and connected processors.

Multicast services that provide VS often leverage on the system’s ability to preserve (when possible) the coordinator during view transitions rather than electing a new coordinator. The motivation here is that the coordinator has the most recent automaton state and holds a copy of the set of unstable messages, which are the ones that were delivered to at least one view member, but the (alive and connected) view members have yet to receive a delivery acknowledgement for these. Our solution naturally follows this approach since it often helps the service to abstain from electing a leader upon every view change, as well as to avoid view transitions that require the coordinator to first investigate about all unstable messages (and the most recent automaton state) among all view members that continue to the next view. This is done so that the service can provide the virtual synchrony property. Thus, we consider the notion of coordinators that a majority of processors never suspects and we show that, in the existence of such processors, one of these coordinator will be eventually used in all subsequent views (Definition 5.1). As explained in Section 2, the algorithm, uses the counter increment algorithm, as well as a reliable multicast and a failure detector built over a self-stabilizing FIFO data link.

**Definition 5.1** 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 $|P_{maj}| > |n/2|$ and $(p_\ell \in (P_{maj} \cap FD_\ell)) \iff (p_\ell \in (P_{maj} \cap FD_\ell))$ in every $c \in R$, where $FD_x$ returns the set of processors that according to $p_\ell$’s failure detector are active.

### 5.1 Detailed Description of Algorithm 4

The existence of coordinator $p_\ell$ is in the heart of Algorithm 4. Processors that belong to and accept $p_\ell$’s view proposal are called the followers of $p_\ell$. The algorithm determines the availability of a coordinator and acts towards the
Algorithm 4: A self-stabilizing automaton replication using virtual synchronization, 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 ($pid, crdID$), $inc()$ returns a counter from the increment counter algorithm.

3. **Variables**: $rep[i] = (view = (ID, set), status \in \{Multicast, Propose, Install\}, (multicast round number) rnd, (replica) state, (last delivered messages) msg[n])$ (to the state machine), $(last fetched) input$ to the state machine, $propV = (ID, set)$, $(no coordinator alive) noCrd$, (recently live and connected component) $FD_i$ : 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 seemsCrd = \{p_k = rep[l].propV.ID.wid \in FD : ([rep[l].propV.set] > |n/2|) \land ([rep[l].FD] > |n/2|) \land (p_k \in rep[l].propV.set) \land (p_k \in rep[l].propV.set \leftrightarrow p_i \in rep[k].FD) \land ((rep[l].status = Multicast) \rightarrow (rep[l].view = propV) \land crd(l) = i) \land ((rep[l].status = Install) \rightarrow crd(l) = i)\};
7. **let valCrd = \{p_k \in seemsCrd : rep[k].propV.ID \leq_{st} rep[l].propV.ID\};
8. **noCrd \leftarrow (valCrd \neq \emptyset); crdID \leftarrow \{\}$;
9. **if ([|FD|] > |n/2|) \land ((valCrd \neq \emptyset) \land ((p_k \in FD : p_i \in rep[k].FD \land rep[k].noCrd) > |n/2|) \land (valCrd = \{p_k\}) \land (FD \neq propV.set) \land ((p_k \in FD : rep[k].propV = propV)) \land (\leq_{st}) then (status, propV) \leftarrow (Propose, (inc(), (FD))
10. **else if (valCrd = \{p_k\}) \land (\forall p_j \in view.set : rep[j].view = Status, status, rnd) = (view, status, $\land (\forall p_j \in propV.set : rep[j].propV, status = (propV, Propose)) then**
11. **if status = 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 \bot \land rnd \leftarrow rnd + 1;
14. **else if status = Propose then**
15. **(state, status, msg) \leftarrow (synchState(rep), Install, synchMsgs(rep)) else if status = Install then (view, status, rnd) \leftarrow (propV, Multicast, 0)
16. **else if valCrd = \{p_k\} \land \ell \neq i \land ((rep[l].rnd = 0 \lor rnd < rep[l].rnd \lor rep[l].(view \neq propV)) then**
17. **if rep[l].status = Multicast then**
18. **if rep[l].state = \bot then rep[l].state \leftarrow state /\star PCE optimization, line 21 /\star rep[l] \leftarrow rep[l]; apply(state, rep[l], msg); /\star for the sake of side-effects /\star input \leftarrow fetch();
19. **else if rep[l].status = Install then rep[i] \leftarrow rep[l]. else if rep[l].status = Propose then (status, propV) \leftarrow (rep[l].status, propV)
20. **let m = rep[l] /\star sending messages: all to coordinator and coordinator to all /\star
either if status = Multicast \land rnd(mod PCE) \neq 0 then m.state \leftarrow \bot /\star PCE optimization, line 17 /\star
either let sendSet = (seemsCrd \cup \{p_k \in propV.set : valCrd = \{p_k\}\}) \cup \{p_k \in FD : noCrd \lor (status = Propose)\}
21. **foreach p_j \in sendSet do send(m)
22. **Upon message arrival m from p_j do rep[j] \leftarrow m;**
election of a new one when no valid such exists (lines 5 to 9). The pseudocode
details the coordinator-side (lines 10 to 14) and the follower-side (lines 15 to 19)
actions. At the end of each iteration the algorithm defines how \( p_t \) and its
followers exchange messages (lines 21 to 24).

**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 mul-
ticast 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 indi-
cate 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 24), where \( rep[i].\langle view, \ldots, noCrd \rangle \) is an array that includes the
last delivered messages to the state machine, which is the input fetched by each
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 avail-
able (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 col-
lects 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 4.3). 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.** Algorithm 4 takes an agile ap-
proach to message multicasting with atomic delivery guarantees. Namely, a new
view is installed whenever the coordinator sees a change to its local failure de-
tector, \( failureDetector() \), which \( p_i \) stores in \( FD_i \) (line 5). Processor \( p_t \) can see
the set of processors, $seemCrd_i$, that each “seems” to be the view coordinator, because $p_i$ stored a message from $p_\ell \in FD_i$ for which $p_\ell = rep[\ell].propV.ID.wid$. Note that $p_i$ cannot consider $p_\ell$ as a (seemly) coordinator when $p_\ell$’s proposal view does not include a majority, or if $p_\ell$ is not a member in the view it claims to coordinate. In the case of Multicast rounds, their view fields must match their view proposal fields (line 6). Also, using the failure detector heartbeat exchange, processors communicate the identifier of the processor they consider to be their coordinator, or $\bot$ if none. As shown in the correctness proof, this helps to detect initially corrupted states where a processor $p_i$ might consider processor $p_j$ to be its coordinator, but processor $p_j$ does not consider itself to be the coordinator.

The algorithm considers a processor as the valid coordinator, if it belongs to $seemCrd$ and has the $\preceq_{ct}$-greatest view identifier among the set of seemly coordinators (line 7). Note that the set $valCrd_i$ either includes a single processor, $p_\ell$ which $p_i$ considers to be a valid coordinator, or $p_i$ does not consider any processor to be a valid coordinator that was recently live and connected (line 8). In the latter case, $p_i$ will not propose a new view before its (local) failure detector indicates that it is within the primary component and that a supportive majority of recently live and connected processors also do not observe the availability of a valid coordinator (line 9). Note that in the case where $p_i$ is a valid coordinator, it will create and propose a new view whenever the last proposed view does not match the set of processors that were recently live and connected according to its (local) failure detector. In such a case no other processor but $p_i$ may propose, because it is the only one that retains a majority of processors that have accepted the previous view.

**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 tole 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 ($rep_i[\ell].(view, status, \text{rnd}) = (view_i, status_i, \text{rnd}_i)$). For the case of Propose and Install rounds, the algorithm does not need to consider the round number, $\text{rnd}$.

Depending on its status, the coordinator $p_i$ proceeds once it observes the successful round conclusion. At the end of a normal Multicast round, the coordinator increments the round number after aggregating the followers’ input (line 11). 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 14). 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. (Note that implicitly the coordinator creates a new view if it detects that the round number is exhausted ($\text{rnd} > 2^{64}$), 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 $\text{rnd}$ part of the state.)
The follower-side. Processor \( p_i \) is aware of its coordinator’s identity when \((\text{valCrd}_i = \{p_\ell\})\) and \( i \neq \ell \) (line 15). 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 \((\text{rep}[\ell].\text{rnd} = 0)\), the first time a message arrives \((\text{rnd} < \text{rep}[\ell].\text{rnd})\) or a new view is proposed \((\text{rep}[\ell].(\text{view} \neq \text{propV}))\).

During normal Multicast rounds (line 16) the follower \( p_i \) applies the aggregated message of this round to its current automaton state so that it produces the needed side-effects before adopting the coordinator’s replica (line 19). Note that, in the case of a Propose round, 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.

The exchanging message and PCE optimization. Each processor periodically sends its current replica (line 23) and stores the received ones (line 24). As an optimization, we propose to avoid sending the entire replica state in every Multicast round. Instead, we consider a predefined constant, \( \text{PCE} \) (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 17 and 21). Note that the greater the \( \text{PCE} \)’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 \( \text{view} \) and \( \text{propV} \), one might want to reduce the communication costs that are associated with the set field and the epoch part of the \( \text{ID} \) field.

5.2 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 5.1) holds throughout \( R \), we reach a configuration \( c \in R \) in which some processor with supporting majority \( p_\ell \) will propose a view including its supporting majority. This view is either accepted by all its member processors or in the case where \( p_\ell \) 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.

Once the system considers processor \( p_\ell \) as the view coordinator (Definition 5.1) its supporting majority can extend the support throughout \( R \) and thus \( p_\ell \) 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 5.1 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 \( \text{valCrd} = \{ p_j \} \). \( p_j \) is also considered the global (view) coordinator if for all \( p_k \) in \( p_j \)'s proposed view (\( \text{propV} \)), it holds that \( \text{valCrd}_k = \{ p_j \} \). When \( p_i \) has a (local) coordinator then \( p_i \)'s local variable \( \text{noCrd} = \text{False} \), whilst when it has no local coordinator, \( \text{noCrd} = \text{True} \).

Moving to the proof, we consider the following useful remark on Definition 5.1.

**Remark 5.1** Definition 5.1 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 \( P_{\text{maj}}(i) \) and \( P_{\text{maj}}(j) \) we do not require that \( P_{\text{maj}}(i) = P_{\text{maj}}(j) \), but \( P_{\text{maj}}(i) \cap P_{\text{maj}}(j) \neq \emptyset \) trivially holds.

**Lemma 5.1** Let \( R \) be an execution with an arbitrary initial configuration, of Algorithm 4 such that Definition 5.1 holds. Consider a processor \( p_i \in P_{\text{maj}} \) 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 \( \text{FD}_i \) to \( p_i \) where \( p_k \notin \text{FD}_i \). The threshold we set for our failure detector (see Section 2) determines how soon \( p_k \) is suspected. By the first condition of line 6 we have that \( p_k \notin \text{FD}_i \Rightarrow p_k \notin \text{seemCrd} \Rightarrow p_k \notin \text{valCrd}_i \), 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. \( \text{valCrd}_i = \{ p_k \} \). Two subcases exist:

**Case 2(a):** \( p_k \) considers itself to have a supporting majority, and \( p_i \in \text{propV}_k \). Note that the latter assumption implies that \( p_k \) is forced by lines 20 - 23 to propagate \( \text{rep}_k[k] \) to \( p_i \) in every iteration. By the failure detector, there exists an iteration where \( p_k \) will have \( |\text{FD}_k| = n/2 + 1 \) and is informed that some \( p_j \in \text{propV}_k \) has \( p_k \notin \text{FD}_j \) and so the condition of line 6 \( (\text{FD} > |n/2|) \) fails for \( p_k \), which stops being the coordinator of itself. If \( p_k \) does not find a new coordinator, hence \( \text{noCrd}_k = \text{True} \), then \( p_k \) propagates its \( \text{rep}_k[k] \) to \( p_i \). But this implies that \( p_i \) receives \( \text{rep}_k[k] \) and stores it in \( \text{rep}_i[k] \). Upon the next iteration of this reception, \( p_i \) will remove \( p_k \) from its \( \text{seemCrd} \) set because \( p_k \) does not satisfy the condition \( |\text{rep}_i[k].\text{FD}| < |n/2| \) 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 \( p_k \not\in seemCrd_i \) 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 5.2** Processor \( p_\ell \in P \) with status = Proposal, is said to propose a view \( propV_\ell \), if in a complete iteration of Algorithm 4, \( p_\ell \) 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 20–23 to all the members of \( FD_\ell \).

The above definition does not imply that \( p_\ell \) will continue proposing the view \( propV \), since the replicas received from other processors may force \( p_\ell \) to either exclude itself from \( valCrd_\ell \) or create a new view (see Lemma 5.3). 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 \( p_\ell \) but bears \( p_\ell \)’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 5.2** If the conditions of Definition 5.1 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 5.2).
Proof. By Definition 5.1, at least one processor with supporting majority exists. Denote one such processor as \( p_\ell \). Assume for contradiction that throughout \( R \), no processor \( p_\ell \) with supporting majority proposes a view. \( p_\ell \) either has a local coordinator (that is not global) or does not have a coordinator.

**Case 1:** \( p_\ell \) **does not have a coordinator** (\( \text{noCrd}_\ell = \text{True} \)). If \( p_\ell \) does not propose a view (as per the “propose” Definition 5.2), 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, \( (|FD| > |n/2|) \) is always satisfied by our assumption that \( p_\ell \) is not suspected by a majority throughout \( R \). In the second condition, both (i) \( (|\{valCrd_\ell| \neq 1\}) \land\) (\( \{p_\ell \in FD_\ell : p_\ell \in rep[i].FD_\ell \land rep[i].\text{noCrd}\} > |n/2| \)) and (ii) \( (valCrd_\ell = \{p_\ell\}) \land \neg (FD_\ell \neq propV_\ell.set) \land (\{p_\ell \in FD : rep[i].propV = propV_\ell\} > |n/2|) \) must fail due to our assumption that \( p_\ell \) never proposes. Indeed (ii) fails since \( noCrd_\ell = \text{True} \Rightarrow valCrd_\ell \neq \{p_\ell\} \). If the first expression also fails, this implies that throughout \( R \), \( p_\ell \) does not know of a majority of processors with \( noCrd_\ell = \text{True} \) 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 \( noCrd_j = \text{False} \) to \( \text{True} \) in order for \( p_\ell \) to gain a majority of processors without a coordinator. But if \( noCrd_j = \text{False} \) 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 5.1 guarantees that \( p_j \) stops considering \( p_k \) as its local coordinator. Thus \( p_j \) eventually goes to \( noCrd = \text{True} \) and by the propagation of its replica, \( p_\ell \) 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. This implies that the Lemma trivially holds, and so the following case must be true.

**Case 2:** \( p_k \) **has a coordinator**, say \( p_k \). 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 \( p_\ell \) will eventually propose a label, or that \( p_k \) has a proposed view, thus contradicting our assumption and so the lemma follows.

Lemma 5.2 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 5.3** If the conditions of Definition 5.1 hold throughout an execution \( R \) of Algorithm 4, then starting from an arbitrary configuration, the system reaches a configuration in which any processor \( p_\ell \) 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 $p_t$ that has a supporting majority throughout $R$. According to Lemma 5.2, $p_t$ must eventually propose a view $\text{prop}V_t$, based on the current $FD_t$ reading (line 5) which becomes the $\text{prop}V_t.set$. By Lemma 5.1, any other processor without a supporting majority will eventually stop being the local coordinator of any $p_j \in \text{prop}V_t.set$ and since such processors do not have a supporting majority, the first condition of line 9 will prevent them from proposing.

Processor $p_t$ continuously proposes $\text{prop}V_t$ until all processors in $\text{prop}V_t.set$ have sent a replica showing that they have adopted $\text{prop}V_t$ as their $\text{prop}V$. Every processor that is alive throughout $R$ and in $FD_t$ should receive this replica through the self-stabilizing reliable communication. The only condition that may prevent $p_j$ to adopt $\text{prop}V_t$ is if for some $p_r \in \text{rep}_j[\ell], \text{prop}V_t.set$ it holds that $p_r \notin \text{rep}_j[\ell].\text{FD}$ (line 6). Plainly put, $p_j$ believes that $p_r$ suspects $p_t$.

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

Case 1(b): If $p_j$’s information is false—remnant of some arbitrary state—, then $p_t \in FD_r$ and since $p_r$, by the last condition of line 22, sends $\text{rep}_r[\ell]$ infinitely often to $p_j$, then $\text{rep}_j[\ell]$ will be corrected and $p_j$ will accept $\text{prop}V_t$.

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

Case 2: More than one processor with supporting majority. Consider two processors $p_t, p_{t'}$ that have a supporting majority such that each creates a view (line 9). By the correctness of our counter algorithm, $\text{inc}()$ returns two distinct and ordered counters to use as view identifiers. Without loss of generality, we assume that $\text{prop}V_t$ proposed by $p_t$ has the greatest identifier of all the counters created by calls to $\text{inc}()$. We identify the following four subcases:

Case 2(a): $p_t \in FD_{t'} \land p_{t'} \in FD_t$. In this case $p_{t'}$ will propose its view $\text{prop}V_{t'}$ and wait for all $p_i \in \text{prop}V_{t'}.set$ to adopt it (line 10). Whenever $p_t$ receives $\text{prop}V_{t'}$, it will store it but will not adopt it, since $\text{prop}V_{t'}.ID \not\leq_{ct} \text{prop}V_t.ID$ (line 7). The proposal $\text{prop}V_t$ is also propagated to every $p_i \in
propV\textsubscript{i}.set. Since there is no greater proposed view identifier than propV\textsubscript{i}.ID, this is adopted by all \( p_i \in \text{propV}_i \) which also includes \( p_{v} \) as well. Thus any processor with supporting majority that belonged to the proposed set of \( p_i \) will propose at most once, and \( p_i \) will become the sole coordinator. Note that if \( p_{v} \) is prevented from adopting propV\textsubscript{i} 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 \( p_i \) is also tackled as in Case 1 of this lemma, by noticing that if \( p_{v} \) manages to get a majority of processors of propV.set then \( p_{v} \) will change its proposed view without losing this majority.

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

What is more crucial, is that \( p_{v} \) 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\textsubscript{i}.set and propV\textsubscript{i}.set). Since any processor \( p_k \) in the intersection propV\textsubscript{i}.set \( \cap \text{propV}_i.set \) has \( p_i \) as its coordinator, \( p_{v} \) does not satisfy the condition \( |\{ p_k \in FD_{v} : p_{v} \in rep[k].FD \land rep[k].noCrd \}| > \lceil n/2 \rceil \) of line 9, and thus cannot propose a new view. Processor \( p_{v} \) will install its view and remains the sole coordinator. Also, \( p_{v} \) 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\textsubscript{i}.set as opposed to \( p_{v} \).

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

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

### Theorem 5.4

Starting from an arbitrary configuration, any execution \( R \) of Algorithm 4 satisfying Definition 5.1, simulates automaton replication preserving the virtual synchrony property.

**Proof.** We consider a finite prefix \( R' \) of \( R \) which has an arbitrary configuration \( c \), and in which there exists a primary partition (as per Definition 5.1). Assume that this prefix is sufficiently long for Lemma 5.3 to hold, i.e., to reach a configuration \( c_{\text{safe}} \) in which there exists a global coordinator for a majority of processors. For this configuration we define a view \( v \) that has a coordinator \( p_\ell \) and that any processor \( p_i \in v \) that is not the coordinator is a follower of \( p_\ell \). We define a *multicast round* to be a sequence of ordered events: (i) \( \text{fetch()} \) input and propagate to coordinator, (ii) coordinator disseminates messages to be delivered in this new round, (iii) messages delivered and (iv) side effects produced by all processors. Our proof is broken into three steps that map the three possible transitions:

**Step 1:** Virtual synchrony is preserved between any two multicast rounds.

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 13 only once and a follower executes lines 16 to 18 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 \( \text{fetch()} \) is called only once per round to collect input from the environment. This cannot be changed/overwritten since followers can never access \( \text{rep}[i] \leftarrow \text{rep}[\ell] \) of line 17 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 \( \text{apply()} \)) based on the messages and state of the previous round. Similarly, the coordinator executes \( \text{fetch()} \) 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 \( p_\ell \) continuously propagates its current replica but cannot change it by the Remark and until condition \((\forall p_i \in \text{v.set} : \text{rep}[i] \leftarrow \text{rep}[\ell], (\text{view}, \text{status}, \text{rnd}) = (\text{view}_\ell, \text{status}_\ell, \text{rnd}_\ell)) \) (line 10) holds again. This ensures that the coordinator will change its *msg* array only when every follower has executed line 17 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 \( \text{rep}[i] \leftarrow \text{rep}[\ell] \) 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.

**Step 2: Virtual synchrony is preserved in two consecutive view installations where there is no change of coordinator.**

We now turn to the case where from one configuration \( c_{\text{safe}} \) we move to a new \( c'_{\text{safe}} \) that has a different view \( v' \) but has the same coordinator \( p_\ell \). Once \( p_\ell \) is in an iteration where the condition \( FD \neq \text{propV.set} \) of line 9 holds, a view change is required. Since \( p_\ell \) is the global coordinator holds, no other processor can satisfy the condition \((|\{p_k \in FD_\ell : \text{rep}[k]_\ell, \text{propV} = \text{propV}_\ell\}| > \lfloor n/2 \rfloor)\) of line 9, and so only \( p_\ell \). For more on why this holds one can prefer to Lemma 5.2. Processor \( p_\ell \) creates a new \( \text{propV} \) 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 \( p_\ell \) will not execute lines 12 to 14 and thus will not change its \( \text{rep.}(\text{state}, \text{input}, \text{msg}) \) 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 \( \text{valCrd} = \{p_\ell\} \). Moreover, the proposal satisfies the condition of line 15 and the followers of the view enter status \text{Propose} leading to the installation of the view. What is important is that virtual synchrony is preserved since no follower is changing \( \text{rep.}(\text{state}, \text{input}, \text{msg}) \) during this procedure, and moreover each sends its replica to \( p_\ell \) by line 22. Once the replicas of all the followers have been collected, the coordinator creates a consolidated \text{state} and \text{msg} array of all messages that were either delivered or pending. \( p_\ell \)'s new replica is communicated to the followers who adopt this state as their own (line 19). 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.

**Step 3: Virtual synchrony is preserved in two consecutive view installations where the coordinator changes.**

We assume that \( p_\ell \) 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 \( p_\ell \). Notice that since Definition 5.1 is required to hold, then some other processor with supporting majority \( p_{\ell'} \), will by Lemma 5.2 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, \( (\text{state}, \text{input}, \text{msg}) \) is retained within the majority.

As detailed in step 2, if a processor \( p_i \) had \( \text{noCrd} = \text{True} \) for some time or was in status \text{Propose} it did not incur any changes to its replica. If it entered the \text{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. ■

6 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, 16]. To this respect, we have presented the first self-stabilizing algorithm that guarantees virtual synchrony, and used it to obtain a 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 [9]. One of the key components of the virtual synchrony algorithm is a novel self-stabilizing counter algorithm, that establishes an efficient practical unbounded counter, which in turn can be directly used to implement a 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 [9].

Acknowledgements

We thank Iosif Salem for providing comments to improve the readability of this paper.

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. *Journal of Computer and System Sciences*, 81(4):692 – 701, 2015.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
Alberto Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.

Ken Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, pages 91–120, 2010.

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

Ken Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

Kenneth P. Birman, Thomas A. Joseph, Thomas Ruchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Trans. Softw. Eng.*, 11(6):502-508, 1985.

Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing Paxos replicated state-machine. In *Proc. of the 2nd International Conference of Networked Systems (NETYS’14)*, pages 99–121, 2014.

Shlomi Dolev. *Self-Stabilization*, MIT press, 2000.

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.

Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Proc. of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS’12)*, pages 133–147, 2012.

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.

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.

Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.

Roger Khazan, Alan Fekete, and Nancy A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proc. of the 12th International Symposium on Distributed Computing (DISC’98)*, pages 258–272, 1998.
[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. pages 558–565, 1978. Commun. ACM, 21(7):558–565, 1978.

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

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

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