Practically Stabilizing Virtual Synchrony

Practically Stabilizing Virtual Synchrony
(Preliminary Version)

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. That is, when using 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 with relation to consensus-based replicated state machine.

[*]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel. Email {dolev, liba}@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

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

1 Introduction

Virtual synchrony has been proven to be very important in the scope of fault-tolerant distributed systems [5]. Systems that support the virtual synchrony abstraction are designed to operate in the presence of fail-stop failures of a minority of the participants. Such a design fits large computer clusters, data-centers and cloud computing, where at any given time some of the processing units are nonoperational. Systems that cannot tolerate such failures degrade their functionality and availability to the degree of unuseful systems.

Group communication systems that realize the virtual synchrony 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, where 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 that are related to virtual synchrony consider the need to maintain atomic message delivery in the presence of asynchrony and crash failures. The implementation of a reliable multicast service that provides virtual synchrony can often continuously use the same group view over many multicast rounds. Thus, virtual synchrony 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.

As such large multicomputer systems are hard to manage and control, one would prefer a system that automatically recovers from unexpected failures, possibly as part of after-disaster recovery or even after benign temporal violation of the assumptions made in the design of the system. For example, the assumption that error detection ensures the arrival of correct messages and the discarding of corrupted messages. In reality, error detection is a probabilistic mechanism that may not detect a corrupted message, and therefore, the message can be regarded legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, unless there is human intervention.

The research in state machine replication (SMR) for obtaining fault tolerance is rich. E.g., SMR is known as a general method to design and implement services [19]. However, when initiating a system in an arbitrary state, there is no guarantee that the system will reach a legal state after which the participants maintain a coherent state. A self-stabilizing algorithm [10] can guarantee such an automatic recovery of the replicated state machine, starting in an arbitrary state, reached due to transient (faults) violations of the design assumptions that lead the system to an arbitrary state.

New challenges appear when designing self-stabilizing systems. One significant challenge is to provide an ordering for message delivery to the replicated state machines, which is an even more intriguing problem when the system starts with inconsistent replicas. Usually, new events are identified by a new incarnation number; a number greater than all previously used numbers. Counters of 64 bits, or so, are usually used to implement such numbers. Such designs were justified by claiming that 64-bit values suffice for implementing (practically) unbounded counters. However, a single transient fault may cause the counter to reach the upper limit at once.

Lamport was the first to introduce SMR, presenting it as an example in [16]. Schneider [19] gave a more generalized approach to the design and implementation of state replication protocols. Birman et al. were the first to present virtual synchrony in [8], and improvements in the efficiency of ordering protocols in subsequent publications [6, 7]. Birman gives a concise account of the evolution of the virtual synchrony model for state replication in [5]. Group communication services can implement SMR by providing reliable multicast that guarantees virtual synchrony [4].

The research during the last recent decades resulted in an extensive literature on ways to implement virtual synchrony and replicated state machines, as well as industrial construction of such systems. A recent research line about (practically) self-stabilizing versions of replicated state machines [1, 9, 13, 14] obtains self-stabilizing replicated state machines in shared memory as well as synchronized and asynchronous message passing systems.

The bounded labeling scheme and the usage 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. The labeling scheme in [9] extends these capabilities to the multi-writer case by exchanging vector of labels. Here, we present a simpler and significantly more communication efficient practically self-stabilizing (bounded-size never-exhausted) counter that supports many writers, where a single value rather than a vector of such values is communicated to achieve the same goal. The new counter is combined with a known self-stabilizing data-link and a token-passing algorithm, a new reliable multicast algorithm, a known failure detector algorithm, and a new virtual synchrony management scheme and replicated state machine implementation to obtain a complete practically stabilizing virtual synchrony and replicated state machine. All the components use bounded memory, e.g., counter, history, message-queues, to ensure self-stabilization in a system that can be started with arbitrary values in the (doomed to be bounded) memory. We next overview our construction, describing the core techniques and the way they establish the desired properties.

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. [Section 4.1]

As mentioned, Alon et al. [1] presented a bounded labeling scheme to implement a 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 life-span 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 identifies are used to break symmetry. Say that an active processor learns and adopts ℓ_1 as its global maximal label. Then, it learns about ℓ_2 and hence adopts it, while forgetting about ℓ_1. Then, learning of ℓ_3 it adopts it. Lastly, it learns about ℓ_1, and as it is greater than ℓ_3, it adopts ℓ_1 once more, as the greatest in the system; this can continue indefinitely. By using the bounded FIFO histories, such labels will be accumulated in the histories and hence will not be adopted again, ending this vicious cycle.

2.2 Practically infinite counter for multiple writers. [Section 4.5]

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 [17, 18]. Specifically, we say that counter $cnt_1 = \langle \ell_1, seqn_1, wid_1 \rangle$ is *smaller* than counter $cnt_2 = \langle \ell_2, seqn_2, wid_2 \rangle$ if $(\ell_1 \prec \ell_2)$ or $((\ell_1 = \ell_2)$ and $(seqn_1 < seqn_2))$ or $((\ell_1 = \ell_2)$ and $(seqn_1 = seqn_2)$ 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 algorithm supports such increments.

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 ids. 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 ℓ as the global maximum, it sets the corresponding counter $cnt = \langle \ell, 0, i \rangle$; in this case, the label creator identifier and the sequence number writer identifier is i. When there is an already established maximal label ℓ in the system and processor p_i wants to increment the counter, it increases the corresponding (to ℓ) maximal sequence number found ($maxseqn$) by one, and sets the counter $cnt = \langle \ell, maxseqn + 1, i \rangle$; 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 ℓ. 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. 4.5).

2.3 Practically self-stabilizing virtual synchrony and Replicated state machine. [Section 5]

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: Roughly speaking, 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 π_1 until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π_2, and so on and so forth. This assures that the receiver has received packet π_1 before the sender starts sending packet π_2. This can be viewed as a token exchange. We use the abstraction of the token carrying messages back and forth between any two communication entities. We use this token exchange technique when implementing a reliable multicast procedure, as well as a the basis for a *heartbeat* for detecting whether a processor is active or not; when a processor in no longer active, the token will not be returned back to the other processor.
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 $\perp$, when the participant does not have a new input). Once the coordinator receives an input from a certain participant with respect to this multicast invocation, the corresponding token will not carry any new requests to receive input from the same participant; of course, the tokens continue to move back and forth. 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 $\perp$ 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 $\perp$) from each of the processors, thus, the processors need to apply one input at a time, the processors may apply them in an agreed upon sequential order, say from the input of the first processor to the last. Alternatively, the coordinator may request one input at a time in a round-robin fashion and multicast it. Finally, to ensure that the system stabilizes when started in an arbitrary configuration, every so often, the coordinator assigns the state of its replica to the other members.

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 systems 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 $(s_1, s_2, \cdots, s_n)$, 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 steps 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 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.

4 Self-stabilizing Labeling Scheme and Increment Counter Algorithm

In this section we first present and prove correct the self-stabilizing labeling algorithm and then explain how this can be extended to implement self-stabilizing practical unbounded counters.

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

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

4.1 Labeling Algorithm for Concurrent Label Creations

4.2 Bounded Labeling Scheme.

We extend the labeling scheme of [1] to support wait-free multi-writer systems. We do so, by extending the label with a *label creator's* id, 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 *epoch*) is a triple $\langle lCreator, sting, Antistings \rangle$, where $lCreator$ is the identity of the processor that established (created) the label, $Antistings \subset D$ with $|Antistings| = k$, and $sting \in D$. Given two labels ℓ_i, ℓ_j, we define the relation $\ell_i \prec_{lb} \ell_j \equiv (\ell_i.lCreator < \ell_j.lCreator) \vee (\ell_i.lCreator = \ell_j.lCreator \wedge ((\ell_i.sting \in \ell_j.Antistings) \wedge (\ell_j.sting \notin \ell_i.Antistings)))$; we use $=_{lb}$ to say that the labels are identical. Note that the relation $\prec_{lb}$ does not define a total order. For example, when $\ell_i =_{lCreator} \ell_j$ and $(\ell_i.sting \notin \ell_j.Antistings)$ and $(\ell_j.sting \notin \ell_i.Antisting)$ these labels are *incomparable*. As in [1], we demonstrate that one can still use this labeling scheme as long as it is ensured that eventually a label greater than all other labels in the system is introduced. We say that a label ℓ ***cancels*** another label ℓ', either if they are incomparable or they have the same $lCreator$ but ℓ is greater than ℓ' (with respect to $sting$ and $Antistings$).

Function $nextLabel()$, Algorithm 1, gets a set of 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 $Next_b()$ in [1], but it additionally considers the label creator. Intuitively, the function composes a new $Antistings$ set from the stings of all the labels it has as input, and chooses a $sting$ that is in none of the $Antistings$ of the input labels. In this way it ensures that the new label is greater than any of the input. Note that the function takes k $Antistings$ of k labels, implying at most k^2 integers and thus the choice of $|D| = k^2 + 1$ allows to always obtain a greatest integer as the $sting$.

4.3 The Labeling Algorithm.

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" 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 $\perp$, or equal to a label that cancels ml (i.e., cl indicates whether ml is an obsolete label or not).

4.3.1 *The processor state:*

Each processor stores an array of label pairs, $max_i[]$, where $max_i[i]$ refers to p_i's maximal label pair and $max_i[j]$ considers the most recent label 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 $storedLabels_i[]$. 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.

4.3.2 *Information exchange between processors:*

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

Whenever a processor p_j sends a pair $\langle sentMax, lastSent \rangle$ to p_i, this processor stores the arriving $sentMax$ in $max_i[j]$ (line 19). Note that in a legal execution the arriving $sentMax$ is always legitimate. However, when p_j acknowledges p_i's label, it is possible that p_j needs to inform p_i of a label from p_i's domain that cancels p_i's maximal label, ml in $max_i[i]$. It does so by sending to p_i a label that cancels ml and thus it would be the case, $lastSent$ will have a $lastSent.cl$, that is not $\perp$. Specifically, it contains a label that p_j knows such that $lastSent.cl \not\preceq_{lb} lastSent.ml$, i.e., $lastSent.cl$ is either greater or incomparable to $lastSent.ml$. Thus, $lastSent$ is illegitimate and in case this still refers to p_i's 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).

4.3.3 *Label processing:*

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

After this series of tests and updates, the algorithm is ready to decide upon a maximal label based on its local information. This is the $\preceq_{lb}$-greatest legit label pair among all the ones in $max_i[]$ (line 26). When no such legit label exists, p_i request 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 $storedLabel_i[i]$ 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 $storedLabel_i[i]$.

4.4 Correctness.

We now outline the algorithm correctness. The full proof is given in Appendix A.1. The proof considers an execution R of Algorithm 2 that may start in an arbitrary configuration. We first show 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 and (2) p_i either adopts or creates the $\preceq_{lb}$-greatest legitimate local label. We then show bounds on the number of adoption steps, first in the absence of label creations and then in the presence of label creations:

Lemma 4.1 *Let $p_i, p_j \in P$, be two processors. Suppose that p_j has stopped adding labels to the system configuration (the else part of line 28), and sending (line 16) these labels during R. Processor p_i adopts (line 27) at most $(n+m)$ label pairs, $lp_j : (lp_j =_{lCreator} j)$, from p_j's unknown domain ($lp_j \notin labels_i(lp_j)$), where m is the maximum number of label pairs that can be in transit in the system.*

Lemma 4.2 *Let $p_i \in P$ be a processor. Let $L_i = lp_{i_0}, lp_{i_1}, \ldots$ be the sequence of legitimate label pairs (i.e., $lp_{i_k}.cl = \perp$), $\ell_{i_k} =_{lCreator} i$, from p_i's domain, which p_i stores in $max_i[i]$ over time, where $k \in \mathcal{N}$. It holds that $|L_i| \leq n(n^2+m)$.*

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

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

We then show that active processors can eventually stop adopting or creating labels. We show that incomparable label pairs eventually disappear from the system and thus no new labels are been adopted or created, which then implies the existence of a global maximal label. Putting everything together, we show that when starting from an arbitrary starting configuration, the system eventually reaches a configuration in which there is a global maximal label.

Theorem 4.3 *Suppose that there exists at least one processor, $p_{unknown} \in P$ with unknown identity, that takes practically infinite number of steps in R. Within a bounded number of steps, there is a legitimate label pair $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that p_i has that label pair $max_i[i] = \ell_{\max}$ when naming its (local) maximal label, $max_i[i].ml$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((max_i[j] \preceq_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max})))$.*

Proof Sketch. 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, $L_i = lp_{i_0}, lp_{i_1}, \ldots$, that processor $p_i \in P$ adds to the system configuration (the else part of line 28), where $lp_{i_k} =_{lCreator} i$ (Lemma 4.2). Thus, by the pigeonhole principle we know that within a bounded number of steps in R, there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst (all processors) p_i do not add any label pair, $lp_{i_k} =_{lCreator} i$, to the system configuration (the else part of line 28). During this practically infinite period (with respect to $p_{unknown}$), in which no label pairs are added to the system (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), $lp_j : (lp_j =_{lCreator} j)$, from p_j's unknown domain ($lp_j \notin labels_k(lp_j)$), it holds that p_k adopts such labels (line 27) only a bounded number of times in R (Lemma 4.1). Again by pigeonhole principle we can say that there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst neither p_i adds a label, $lp_{i_k} =_{lCreator} i$, to the system (the else part of line 28), nor p_k adopts labels (line 27), $lp_j : (lp_j =_{lCreator} j)$, from p_j's unknown domain ($lp_j \notin labels_k(lp_j)$). Consequently, whilst $p_{unknown}$ takes a practically infinite number of steps, all processors (that takes practically infinite number of steps in R) name the same $\preceq_{lb}$-greatest legitimate label which the theorem statement specifies. ■

4.5 Increment Counter Algorithm

In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection and obtain a practically self-stabilizing counter increment algorithm. To do so, we now need to work with practically unbounded *counters*. As already mentioned in Section 2, a counter cnt is a triplet $\langle lbl, seqn, wid \rangle$, where lbl is an epoch label as defined in the previous subsection, $seqn$ is a 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 cnt_i, cnt_j we define the relation $cnt_i \prec_{ct} cnt_j \equiv (cnt_i.lbl \prec_{lb} cnt_j.lbl) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn < cnt_j.seqn)) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn = cnt_j.seqn) \wedge (cnt_i.wid < cnt_j.wid))$. Observe that when the labels of the two counters are incomparable, the counters are also incomparable.

Therefore, the relation $\prec_{ct}$ defines a total order (as required by practically unbounded counters) only when processors share a globally maximum label, (i.e., the system runs within a "stable" epoch). As we have shown in the previous subsection, 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. Due to lack of space, we do not provide the full details of the counter increment algorithm (a detailed pseudocode is given in Appendix A.7). Instead, we highlight the main issues one needs to consider when dealing with counters rather than labels.

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 $storedLabels[]$, 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 these structures are called $maxC[]$ and $storedCnts[]$). However, now each label can yield many different counters. Therefore in order not to increase the size of these queues (with respect to the number of elements stored), for each label we keep only the highest sequence number observed (breaking ties with *wid*s). Also, note that if there are counters in the system that are corrupted (being in the initial arbitrary configuration), then they can only force a change of label if their sequence number is *exhausted* (i.e., $seqn \geq 2^{64}$). Exhausted counters are treated by the algorithm in a similar way as canceled labels in the labeling algorithm; an exhausted counter cnt_i in a counter pair $\langle cnt_i, cnt_j \rangle$ is canceled, by setting $cnt_i.lbl = cnt_j.lbl$ (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 $maxC_i[i]$). This cannot increase the number of labels that are created due to the initially corrupted ones, as the total capacity of the links in the system still corresponds to m.

Another issue worth mentioning is that the system might revert back to a previous legit label x, in case the current maximal label y is 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 it. Otherwise the queues might grow with no bound. But as mentioned above, each processor stores only the maximal sequence number learned for each label, inside $storedCnts[]$ (i.e., the counter with the maximal $(seqn, wid)$ to the corresponding lbl).

Then, processors increment counters as described in Section 2 (by communicating with majorities of processors). A pseudocode for the counter increment procedure can be found in Appendix A.7. So, putting everything together we conclude the following:

Theorem 4.4 *Given an execution of the counter increment algorithm in which up to a minority of processors may become inactive, starting from an arbitrary configuration, the algorithm eventually ensures that counters increment monotonically.*

Proof Sketch. Using similar arguments as in the correctness of the labeling algorithm, the system starts from an arbitrary configuration and reaches a configuration in which there exists a global maximal label. Then the claim (which now focuses on $(seqn, wid)$ pairs), follows by the intersection property of the majorities used by the processors to increment the counter and the definition of $\prec_{ct}$. ■

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 $\perp$ so the processor needs to attempt to read again (i.e., the system hasn't converged to a maximal label yet). If a maximal counter exists, it sends this together with the associated value to all the processors, and once it collects a majority of responses, it returns the counter with the associated value (the second phase is a standard requirement for preserving the consistency of the register (c.f. [3, 18]).

5 Virtually Synchronous Stabilizing Replicated State Machine

In this section, we present a practically self-stabilizing virtual synchrony algorithm that emulates state-machine replication. As explained in Section 2, the algorithm, besides the counter incremental algorithm of Section 4.5, also uses a reliable multicast and a failure detector built over a self-stabilizing FIFO data link. Following [6], we assume that the algorithm works over the network's primary partition and require a majority of processors to be present in every view set. More specifically:

Definition 5.1 (Primary partition executions) *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, $P_{maj} \subseteq P$, that (mutually) never suspect at least one processor, i.e., $\exists p_\ell \in P$ for which $|P_{maj}| > \lfloor n/2 \rfloor$ and $(p_i \in (P_{maj} \cap FD_\ell)) \iff (p_\ell \in (P_{maj} \cap FD_i))$ in every $c \in R$, where FD_x returns the set of processors that according to x's failure detector are active.*

We first describe the algorithm and then prove its correctness. For a full description of Algorithm 3 see Appendix B.1.

5.1 The algorithm.

The existence of coordinator p_ℓ is in the heart of Algorithm 3. The algorithm determines p_ℓ's availability and acts towards finding a new coordinator when no valid coordinator exists (lines 5 to 9). The pseudocode details the coordinator-side (lines 10 to 14) and the follower-side (lines 15 to 19) actions before it explains how p_ℓ and its followers exchange messages (lines 21 to 24). The processor's state and interfaces are defined in lines 1 to 3.

5.1.1 *Determining coordinator availability:*

The algorithm takes an agile approach for multicasting with atomic delivery guarantees. Namely, a new view is installed whenever the coordinator sees a change to its local failure detector, $failureDetector()$,

Algorithm 3: A self-stabilizing automaton replication using virtual synchrony, code for processor p_i

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

which p_i stores in FD_i (line 5). Processor p_i can see the set of processors, $seemCrd_i$, that each seems to be the view coordinator, because p_i stored a message from $p_\ell \in FD_i$ in which $p_\ell = rep[\ell].propV.ID.wid$. Note that p_i cannot consider p_ℓ as a (seemly) coordinator unless the conditions in line 6 hold. Also, as explained in Section 2, using the failure detector heartbeat exchange, processors can detect initially corrupted states. A processor considers as a valid coordinator, the processor in $seemCrd_i$ that has the $\preceq_{ct}$-greatest view identifier (line 7). Note that set $valCrd_i$ is either a singleton or empty (line 8). In the latter case, p_i will not propose a new view before its failure detector indicates that there exists a supportive majority of live and connected processors that also do not have a valid coordinator (line 9).

5.1.2 *The coordinator-side:*

Processor p_i is aware of its valid coordinatorship when $(valCrd_i = \{p_i\})$ (line 10). During a normal Multicast round, p_i observes the round end, once for every view member p_j it holds that $(rep_i[j].(view, status, rnd) = (view_i, status_i, rnd_i))$. Depending on its *status*, the coordinator p_i proceeds once it observes a 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). After a successful Install round, the coordinator proceeds to a Multicast round after installing the proposed view and the first round number.

5.1.3 *The follower-side:*

Processor p_i considers p_ℓ as its coordinator when $(valCrd_i = \{p_\ell\})$ and $i \neq \ell$ (line 15). It has to act upon merely new messages, i.e., the first message round when installing a new view $(rep[\ell].rnd = 0)$, the first time a message arrives $(rnd < rep[\ell].rnd)$ or a new view is proposed $(rep[\ell].(view \neq propV))$. During normal Multicast rounds (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). While in a Propose round, p_i does not overwrite its round number, and so the coordinator can know the last round number that p_i delivered a message during the latest installed view. Both the coordinator and the followers periodically send their current replica (line 23) and store the replicas received (line 24). As an optimization, during normal Multicast rounds, processors transmit their full replica state every *PCE* rounds, where *PCE* is a predefined constant.

5.2 Correctness.

The correctness proof shows that starting from an arbitrary state in an execution R of Algorithm 3 and once the primary partition property (Definition 5.1) holds throughout R, we reach a configuration $c \in R$ in which some processor p_ℓ proposes a view including a majority of processors and this view is accepted by all its members. We conclude by proving that any execution suffix in R that begins from such a configuration c will preserve the virtual synchrony property and implement state machine replication. The full proof is in Appendix B.2.

We first show that a coordinator without a supporting majority stops being the coordinator. Then we show that when there is no coordinator, a processor with a supporting majority eventually proposes a view and all such processors propose at most once, leading to a unique coordinator.

Lemma 5.1 *If the conditions of Definition 5.1 hold throughout an execution R of Algorithm 3, then starting from an arbitrary configuration, the system reaches a configuration in which any processor p_ℓ with a supporting majority may propose itself as the coordinator at most once. As a consequence, the system reaches a configuration in which one of these processors is the global coordinator until the end of the execution.*

Then we show the main result:

Theorem 5.2 *Starting from any configuration, an execution R of Algorithm 3 satisfying Definition 5.1, emulates automaton replication preserving the virtual synchrony property.*

Proof Sketch. 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) and assume that this prefix is sufficiently long for Lemma 5.1 to hold. I.e., we reach a configuration c_{safe} in which there exists a global coordinator for a majority of processors. Then by careful consideration of the code and the way the coordinated multicast steps take place we argue the claim of the theorem. $\blacksquare$.

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, 15]. 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].

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*, *in press*, 2015. A preliminary version has appeared in the *Proc. of SSS 2011*.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *Proceedings of FTCS'92*, pages 76–84, 1992.

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

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

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

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

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

[8] 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.

[9] 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.

[10] Shlomi Dolev. *Self-Stabilization*, MIT press, 2000.

[11] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.

[12] Shlomi Dolev, 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

Appendix

A Omitted Details from Section 4

A.1 Correctness of Algorithm 2

We now give in detail the proof of Algorithm 2. We begin with some terminology and notation.

A.2 Definitions.

We define $\mathcal{H}$ to be the set of all label pairs that can be in transit in the system; $|\mathcal{H}| = m$. So in the arbitrary configuration, there can be up to m corrupted label pairs in the system. 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.

Recall that the data structures used (e.g., $max_i[]$, $storedLabels_i[]$, etc) store label pairs. For convenience of presentation and when clear from context when we will be referring to a label rather than a label pair we mean the ml part of the pair. When we say a *legitimate* label we essentially mean that the cl part of the label is $\perp$.

A.3 Correctness proof.

The proof considers an execution R of Algorithm 2 that may start in an arbitrary configuration. It start by showing some basic fact, 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 (Corollary A.1), and (2) p_i either adopt or create the $\preceq_{lb}$-greatest legitimate local label (Argument A.2). The proof then presents bounds on the number adoption steps (arguments A.3 and A.4).

The proof continues and show that active processors can eventually stop adopting of creating labels. In are particularity interested in looking into cases in which canceled label pairs and incomparable ones. We show that they eventually disappear from the system (Argument A.5) and thus no new label are been adopted or created (Argument A.6), which then implies the existence of a global maximal label (Argument A.7). Namely, there is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any forever active processor $p_j \in P$, it holds that $((max_i[j] \preceq_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max})))$.

We then demonstrates that, when starting from an arbitrary starting configuration, the system eventually reaches a configuration in which there is a global maximal label (Theorem A.2). Note that the convergence holds also when starting from a configuration, $c \in R$, that is obtained by taking a configuration c' in which $risk = \emptyset$ and apply at least one step in which one processor $p_i \in P$ (that takes practically infinite number of steps in R) calls $useOwnLabel()$, say, due to the exhaustion of the sequence number in its timestamp.

A.3.1 No stale information

Argument A.1 says the predicate $staleInfo()$ (line 9) might not hold only but only during the first execution of the $receive()$ event (line 17).

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

Proof. Since R starts in an arbitrary configuration, there could be a queue in $storedLabels_i[]$ that holds two label records from the same creator, a label that is not stored according to its creator identifier, or more than one legitimate label. Therefore, $staleInfo_i()$ might hold during the first execution of the $receive()$ event. However, as we show, during that event execution (and any event execution after) p_i adds records to a queues in $storedLabels_i[]$ (according to the creator identifier) only after checking whether $recordDoesntExist()$ holds (line 22). Any other access to $storedLabels_i[]$ merely update cancelations or remove duplicates. Namely, canceling labels that are not $\preceq_{lb}$-greatest among the ones that share the same creating processors (line 23) and canceling record that were canceled by other processors (line 24) as well as removing legitimate records that share the same name (line 25). ■

Argument A.1, line 9 and line 26, implies Corollary A.1.

Corollary A.1 *After any step that include the execution of the $receive()$ event, other than the first one, it holds that $\forall p_i, p_j \in P$, the state of p_i encodes at most one legitimate label, $\ell_j =_{lCreator} j$, which appears in $storedLabels_i[j]$ rather than $storedLabels_i[k] : k \neq j$ and possibly in $max_i[]$ as well.*

A.3.2 Local $\preceq_{lb}$-greatest legitimate local label

Argument A.2 considers processors for which $staleInfo()$ (line 9) does not hold. Note that $staleInfo()$ holds at any time after the first step that includes the $receive()$ event (Argument A.1). Argument A.2 shows that p_i either adopt or create the $\preceq_{lb}$-greatest legitimate local label and stores it in $max_i[i]$.

Argument A.2 *Let $p_i \in P$ be a processor, such that $\neg staleInfo_i()$ (line 9), and $L_{pre}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j]) \wedge (\exists \langle max_i[j].ml, x\rangle \in (labels(max_i[j]) \setminus \{max_i[j]\}) \Rightarrow (x = \perp))\}$ be the set of $max_i[]$'s labels that, before p_i executes lines 21 to 28, are legitimate both in $max_i[]$ and in $storedLabels_i[]$'s queues. Let $L_{post}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ and $\langle \ell, \perp\rangle$ be the value of $max_i[i]$ immediately after p_i executes lines 21 to 28. The label $\langle \ell, \perp\rangle$ is the $\preceq_{lb}$-greatest legitimate label in $L_{post}(i)$. Moreover, suppose that $L_{pre}(i)$ has a $\preceq_{lb}$-greatest legitimate label, then that label is $\langle \ell, \perp\rangle$.*

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

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

A.4 Bounding the number of labels

Arguments A.3 and A.4 present bounds on the number adoption steps.

A.4.1 Maximum number of label adoption in the absence of creations

Suppose that there exists a processor, p_j, that has stopped adding labels to the system (the else part of line 28), say, because it became inactive (crashed), or it names a maximal label that is the $\preceq_{lb}$-greatest label among all the ones that the network ever delivers to p_j. Argument A.3 bounds the number of labels from p_j's domain that any processor $p_i \in P$ adopts in R.

Argument A.3 *Let $p_i, p_j \in P$, be two processors. Suppose that p_j has stopped adding labels to the system configuration (the else part of line 28), and sending (line 16) these labels during R. Processor p_i adopts (line 27) at most $(n + m)$ labels, $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_i(\ell_j)$) where m is the maximum number of label pairs that can be in transit in the system.*

Proof. Let $p_k \in P$. At any time (after the first step in R) processor p_k's state encodes at most one legitimate label, ℓ_j, for which $\ell_j =_{lCreator} j$ (Corollary A.1). Whenever processor p_i adopts (line 27) a label, $\ell_j : (\ell_j =_{lCreator} j)$ from p_j's domain, because ℓ_j is not added to the p_i's state (the else part of line 28) and send (line 16) during R (as in this argument statement). Thus, ℓ_j must come from p_k's state or delivered via the network. The bound holds since there are n processors, such as p_k, and m bounds the number of labels in transient. ■

A.4.2 Maximum number of label creations

Argument A.4 shows a bound on the number adoption steps that does not depends whether the labels are from the domain of an active or (eventually) inactive processor.

Argument A.4 *Let $p_i \in P$ be a processor. Let $L_i = \ell_{i_0}, \ell_{i_1}, \dots$ be the sequence of legitimate labels, $\ell_{i_k} =_{lCreator} i$, from p_i's domain, which p_i stores in $max_i[i]$ over time, where $k \in \mathcal{N}$. It holds that $|L_i| \leq n(n^2 + m)$.*

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

The latter case (the else part of line 28) refers to labels, ℓ_{i_k}, that p_i stores in $max_i[i]$ only after checking that there are no legitimate labels stored in $max_i[]$ or $storedLabels_i[]$. Note that every time that p_i executes (the else part of line 28), p_i stores the returned label, ℓ_{i_k}, in $storedLabels_i[i]$. After that, there are only three ways for ℓ_{i_k} not to be stored as a legitimate label in $storedLabels_i[i]$: (i) execution of line 21, (ii) the network delivers to p_i a label, ℓ', that either cancels ℓ_{i_k} or for which $\ell' \not\preceq_{lb} \ell_{i_k}$, and (iii) ℓ_{i_k} overflows from $storedLabels_i[i]$ after exceeding the $(n(n^2 + m) + 1)$ limit.

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

Case (ii) cannot occur after p_i has learned all the labels $\ell \in remoteLabels_i$ for which $\ell \notin storedLabels_i[i]$, where $remoteLabels_i = (((\cup_{p_j \in P} localLabels_{i,j}) \cup \mathcal{H}) \setminus localLabels_{i,i})$ and $localLabels_{i,j} = \{\ell' : \ell' =_{lCreator} i$ $\exists p_j \in P : ((\ell' \in storedLabels_j[i]) \vee (\exists\, p_k \in P : \ell' = max_j[k].ml))\}$. During this learning process p_i cancels (or update the cancellation) labels in $localLabels_{i,i}$ before adding a new legitimate label. Thus, this learning process can be seen as moving labels from $remoteLabels_i$ to $storedLabels_i[i]$ and then keeping at most one legitimate label available in $storedLabels_i[i]$. Once $storedLabels_i[i]$ accumulates another label, ℓ, that was unknown to p_i until the returned value, ℓ_{i_k}, that p_i gets from *nextLabel*() is $\preceq_{lb}$-greater-equal than any label $storedLabels_i[i]$ as well as the ones in $remoteLabels_i$.

Note that $remoteLabels_i$'s labels must come from the (arbitrary) start of the system, because p_i is the only one that can add label to the system from its domain. We show that case (ii) stops occurring before case (iii) can occur by demonstrating that $|remoteLabels_i| < n(n^2+m)$. Namely, $|remoteLabels_i| = (n-1)(|max[]| + |storedLabels[i]|) + |\mathcal{H}| = (n-1)(n + (n^2+m)) + m = n^3 + (m-1)n$.

Note that, since we are interested in a bound on the number of adoption steps, this proof does not distinguish between processors that takes bounded or practically infinite number of steps in R and considers all processors as the ones that take a practically infinite number of steps. ■

A.5 Pair diffusion

The proof continues and show that active processors can eventually stop adopting or creating labels. We are particularity interested in looking into cases in which canceled label pairs and incomparable ones. We show that they eventually disappear from the system (Argument A.5) and thus no new label are been adopted or created (Argument A.6), which then implies the existence of a global maximal label (Argument A.7).

Arguments A.5 and A.6, as well as Argument A.7 and Theorem A.2 assume the existence of at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Suppose that processor $p_i \in P$ takes a bounded number of step in R during a period in which $p_{unknown}$ takes a practically infinite number of steps. We say that p_i has become inactive (crashed) during that period and assume that it does not resume to take steps at any later stage of R (in the manner of fail-stop failures, as in Section 3).

Given processor $p_i \in P$ that takes any number (bounded or practically infinite) of steps in R, two processors $p_i, p_j \in P$ (that take a practically infinite number of steps in R), we use the following definitions for estimating whether there are label pairs, ℓ and ℓ', that have the potential to disturb the system by bringing p_j to either add a label, $\ell_j =_{lCreator} i$, to the system configuration (the else part of line 28), or adopt labels (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_k(\ell_j)$).

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

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

Argument A.5 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R during a period whilst neither p_j adds a label, $\ell_j =_{lCreator} i$, to the system (the else part of line 28), nor p_j adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$). Then $risk = \emptyset$ eventually.*

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

The case of $(\ell_j, \ell_k) \in max_{i,j,k}$. Here the label pairs ℓ_j and ℓ_k are named by p_j and p_k as their local maximal label. The assumptions that, throughout R, processors p_j and p_k are alive, as well as $(\ell_j, \ell_k) \in max_{i,j,k} = \{(max_j[j], max_k[k])\}$ implies that eventually p_j and p_k exchange messages. Moreover, $(\ell_j, \ell_k) \in max_{i,j,k}$ implies that $risk_{i,j,k}(\ell_j, \ell_k)$ holds and thus we cannot have that both p_j and p_k continue forever in R to name ℓ_j, and respectively, ℓ_k as their local maximal label pairs. Thus, a contradiction.

The case of $(\ell_j, \ell_k) \in ack_{i,j,k}$**.** This case follows by similar arguments to the case of $(\ell_j, \ell_k) \in max_{i,j,k} = \{(max_j[j], max_k[j])\}$ the shows that eventually processor p_k replaces $max_k[j]$ with a more recent value of $max_j[j]$ and thus $(\ell_j, \ell_k) \notin ack_{i,j,k}$, a contradiction, or processor p_j eventually receives $\ell_k = max_k[j]$ from p_k. Since $(\ell_j, \ell_k) \in ack_{i,j,k}$ then $risk_{i,j,k}(\ell_j, \ell_k)$ holds and thus p_j must change its the value of $max_j[j] = \ell_j$ in a way that will take into account the received label pair ℓ_k and after this change it holds that $(\ell_j, \ell_k) \notin ack_{i,j,k}$. Thus, a contradiction.
The case of $(\ell_j, \ell_k) \in hName_{i,j,k}$**.** $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ This case follows by the same arguments to the case of $(\ell_j, \ell_k) \in max_{i,j,k}$.
The case of $(\ell_j, \ell_k) \in hAck_{i,j,k}$**.** $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$ This case follows by the same arguments to the case of $(\ell_j, \ell_k) \in ack_{i,j,k}$.
The case of $(\ell_j, \ell_k) \in stored_{i,j,k}$**.** $stored_{i,j,k} = \{max_j[j]\} \times storedLabels_k[i]$ This case follows by similar arguments to the case of $(\ell_j, \ell_k) \in max_{i,j,k}$. Namely, p_k eventually receives the label pair $\ell_j = max_j[j]$. The assumption that $risk_{i,j,k}(\ell_j, \ell_k)$ holds implies that one of the tests in lines 23 and 26 will either update $storedLabels_k[i]$, and respectively, $max_k[j]$ with canceling values. We note that for the latter case we argue that p_j eventually received the canceled label pair in $max_k[j]$, because we assume that p_j does not change the value of $max_j[j]$ throughout R. Therefore, in both cases we have a contradiction. ■

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

Argument A.6 *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Neither p_j adds a label, $\ell_j =_{lCreator} i$, to the system (the else part of line 28), nor p_j adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$).*

Proof Sketch. Note that the definition of $risk$ consider almost every possible combination of two label pairs ℓ_j and ℓ_k from p_i's domain that are stored by processor p_j, and respectively, p_k (or in the channels to them). The only combination that is no considered is $(\ell_j, \ell_k) \in storedLabels_j[i] \times storedLabels_k[i]$. However, this combination can indeed reside in the system during a legal execution and it cannot led to a disruption for the case of $risk = \emptyset$ in every configuration throughout R because before that could happen, either p_j or p_k would have to adopt ℓ_j, and respectively, ℓ_k, which means a contradiction with the assumption that $risk = \emptyset$. Similarly one can argue for the case of two messages in transit, $\mathcal{H}_{j,k} \times \mathcal{H}_{k,j}$. ■

Argument A.7 *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. There is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes practically infinite number of steps in R), it holds that $((max_i[j] \preceq_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max})))$.*

Proof Sketch. Since, throughout R, any two active processors forever exchange their local maximal label pairs and yet $risk = \emptyset$ is empty, we have that all of the no two active processors ever name in R two incomparable (local) maximal labels. Therefore, there is a (local) maximal label, ℓ_{max}, that is $\prec_{lb}$-greater than all other labels that other active processors name as their (local) maximal labels. Thus, label ℓ_{max} will be adopted by every active processors in the system eventually. ■

A.6 Convergence

Theorem A.2 demonstrates that, when starting from an arbitrary starting configuration, the system eventually reaches a configuration in which there is a global maximal label.

Theorem A.2 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that take practically infinite number of steps in R. Within a bounded number of steps, there is a legitimate label pair $\ell_{\max}$, such that for any processor $p_i \in P$ (that take a practically infinite number of steps in R), it holds that p_i has that label pair $max_i[i] = \ell_{\max}$ when naming its (local) maximal label, $max_i[i].ml$. Moreover, for any processor $p_j \in P$ (that take a practically infinite number of steps in R), it holds that $((max_i[j] \preceq_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max})))$.*

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 $\ell_{i_k} =_{lCreator} i$ (Argument A.4). Thus, by the pigeonhole principle we know that, within a bounded number of steps in R, there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst (all processors) p_i do not add any label pair, $\ell_{i_k} =_{lCreator} i$, to the system configuration (the else part of line 28). During this practically infinite period (with respect to $p_{unknown}$), in which no label pairs are added to the system configuration (the else part of line 28), we know that for any processor $p_j \in P$ that take any number of (bounded or practically infinite) steps in R, and processor $p_k \in P$ that adopts labels in R (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_k(\ell_j)$), it holds that p_k adopts such labels (line 27) for at most a bounded number times in R (Argument A.3). Therefore, we can again follow the pigeonhole principle and say that there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst neither p_i adds a label, $\ell_{i_k} =_{lCreator} i$, to the system (the else part of line 28), nor p_k adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_k(\ell_j)$). Consequently, whilst $p_{unknown}$ takes a practically infinite number of steps, all processors (that takes practically infinite number of steps in R) name the same $\preceq_{lb}$-greatest legitimate label which the theorem statement specify (Argument A.7). ■

A.7 Pseudocode and Description of the Counter Algorithm

We provide a pseudocode of the counter increment algorithm that extends the labeling algorithm using sequence numbers.

As detailed in Section 4.5 the counter increment algorithm enhances the labeling algorithm to take care of the counter increment once a maximal label exists in the system. The structures $max[]$ and $storedLabels[]$ that were used for label pairs $\langle ml, ct \rangle$ in the labeling algorithm are now defined to contain counter pairs $\langle mct, cct \rangle$ and are renamed to $maxC$ and $storedCnts[]$ respectively . The maintenance of these structures is similar to the labeling algorithm. We define $enqueue(ctp)$ to add a counter pair to a queue if it doesn't exist, or to keep the one which is greatest w.r.t. $\prec_{ct}$ if it exists. In case at least one of the two counter pairs is canceled then we keep a canceled version. Note that in this way the system can revert back to a label that had not been canceled but was not used because a greater one existed. Procedure $cancelExhausted()$ (line 29), cancels an exhausted counter pair ctp, by setting $ctp.cct = ctp.mct$ (i.e., the counter's own label cancels it) and hence making the counter non-legit (thus it cannot be used as a maximal counter).

The increment counter algorithm executed in lines 12 to 20 follows the logic of a writer in a MWMR register emulation. The processor inquires a quorum (lines13) and aggregates the responses, maintaining the structures (lines 25- 26). Note that $process()$ (line 15) performs the same operations as lines 21 to 28 in the labeling algorithm, to keep the consistency of the counter storage. The maximal legit non-exhausted counter in $maxC_i[]$ is the greatest of the counters from the quorum, or, in case such a counter was not found, it is a newly created counter. This is increased in line 19 and written to a quorum (line 20).

Algorithm 4: Counter Increment; code for p_i

```
1  Variables:
2  A label $\langle lbl \rangle$ is extended to the triple $\langle lbl, seqn, wid \rangle$ called a counter where: $seqn$, is the sequence number
   related to the $lbl$ label, and $wid$ is the identity of the creator of this $seqn$ (as detailed in section 4.5). A
   counter pair $\langle mct, cct \rangle$ is the natural extension of a label pair, where $cct$ is a canceling counter for $mct$, such
   that $cct.lbl \not\preceq_{lb} mct.lbl$ or $cct.lbl = \bot$. We extend the $storedLabels()$ and $max[]$ structures of Algorithm 2 to
   structures of counter pairs (instead of label pairs) under the names $storedCnts[]$ and $maxC[]$. We refer to
   $process()$ as the execution of lines 21 to 28 of Algorithm 2. To ease the presentation of the pseudocode, we
   assume that a label created by processor $p_i$ in line 28 of the labeling algorithm is initiated with a $seqn$ set to 0
   and a $wid = i$ before added to $maxC[i]$ and $storedCnts[i]$.
3  The operator $enqueue(ctp)$ places a counter pair $ctp$ at the front of a queue. If $ctp.mct.lbl$ already exists in the
   queue, it keeps the greatest counter w.r.t. $\prec_{ct}$ and places it at the front of the queue. If one counter pair is
   canceled then the canceled copy is the one retained.
4  Notation: Let $y$ and $y'$ be two records that include the field $x$. We denote $y =_x y' \equiv (y.x = y'.x)$.
5  Macros:
6  $exhausted(ctp) = (ctp.mct.seqn \geq 2^{64})$
7  $legit(ctp) = (ctp.cct.lbl = \bot \rangle)$
8  $retCounterQ(ct)$ : return $(storedCnts[ct.lbl.lCreator])$
9  $diffSeq(mct, mct') = (mct \neq mct' \wedge mct =_{lbl} mct')$
10 $legitCounters() = \{maxC[j].mct : \exists p_j \in P \wedge legit(maxC[j])\}$
11 $getMaxSeq()$ : return $maxC_{wid}(\{maxC_{seqn}(\{ctp : ctp.mct \in legitCounters() \wedge maxC[i] =_{mct.lbl} ctp)\}\})$
12 procedure $incrementCounter()$ begin
13     $quorumRead()$;
14     repeat
15         $process()$;
16         $maxC[i] \leftarrow getMaxSeq()$;
17         if $exhausted(maxC[i])$ then $cancelExhausted()$
18     until $\neg exhausted(maxC[i])$;
19     $maxC[i].mct.seqn \leftarrow maxC[i].mct.seqn + 1$; $maxC[i].mct.wid \leftarrow i$;
20     if $\exists ctp \in retCounterQ(maxC[i]) : ctp =_{mct.lml} maxC[i]$ then $retCounterQ(ctp.mct).enqueue(maxC[i])$
       $quorumWrite(maxC_i[i])$
21 procedure $quorumRead()$ begin
22     foreach $p_j \in P$ do send $quorumMaxRead()$ to $p_j$ repeat
23         upon receipt of $\langle sentMax_j, lastSent_j \rangle$ from $p_j \in P$
24         begin
25             $maxC[j] \leftarrow sentMax_j$;
26             if $\neg legit(lastSent_j) \wedge maxC[i] =_{mct.lbl} lastSent_j$ then $maxC[i] \leftarrow lastSent_j$ foreach $ctp'.mct' \in$
               $\{sentMax_j, lastSent_j\}, ctp \in retCounterQ(mct') : legit(ctp') \wedge legit(ctp) \wedge diffSeq(ctp, ctp')$ do
               $retCounterQ(mct').enqueue(ctp')$
27     until responses from a quorum received;
28 upon request for $quorumMaxRead()$ from $p_j$ do send$(\langle maxC_i[i], maxC_i[j] \rangle)$ to $p_j$;
29 procedure $cancelExhausted()$ begin
30     foreach $p_j \in P,\ ctp \in maxC[j] \wedge retCounterQ(ctp) : ctp =_{ml} maxC[i]$ do
       $retCounterQ(ctp).cct \leftarrow ctp.mct$
31 procedure $quorumWrite(maxC_i[i])$ begin
32     foreach $p_j \in P$ do send $quorumMaxWrite(maxC_i[i])$ wait for $ACK$ from a quorum;
33 upon request for $quorumMaxWrite(max_j)$ from $p_j$ begin
34     $maxC_i[j] \leftarrow max_j$;
35     if $\exists ctp \in retCounterQ(maxC_j) : ctp =_{mct} max_j$ then $retCounterQ(ctp.mct).enqueue(max_j)$ send
       $ACK$ to $p_j$;
```

B Omitted Details from Section 5

B.1 Detailed Description of Algorithm 3

The existence of coordinator p_ℓ is in the heart of Algorithm 3. The algorithm determines p_ℓ's availability and acts towards the election of a new one when no valid coordinator exists (lines 5 to 9). The pseudocode details the coordinator-side (lines 10 to 14) and the follower-side (lines 15 to 19) actions before it explains how p_ℓ and its followers exchange messages (lines 21 to 24).

B.1.1 *The processor state and interfaces:*

The state of each processor includes its current $view$, $status$, which can refer to normal operation during Multicast rounds, or view recovery rounds in which the coordinator can Propose a new view and Install a new one once all preparations are done (line 3). During multicast rounds, rnd denotes the round number, $state$ stores the replica, $msg[n]$ is an array that includes the last delivered messages to the state machine, which is the $input$ fetched by each group member and then aggregated by the coordinator during the previous multicast round. During normal multicast rounds, it holds that $propV = view$. However, whenever $propV \neq view$ we consider $propV$ as the newly proposed view and $view$ as the last installed one. Each processor also uses $noCrd$ and FD to indicate whether it is aware of the absence of recently active and connected valid coordinator, and respectively, 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 $rep[i].(view, \ldots, noCrd)$ is an alias to aforementioned variables and $rep[j]$ refers to the last arriving message from processor p_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 3 assumes access to application's message queue via $fetch()$, which returns the next multicast message, or $\perp$ when no such message is available (line 2). It also assume the availability of the automaton state transition function, $apply(state, msg)$, which applies the aggregated input array, msg, to the replica's $state$ and produces the local side effects. The algorithm also collects the followers' replica states and uses $synchState(replica)$ to return the new state. The function $failureDetector()$ provides access to p_i's failure detector, and the function $inc()$ (counter increment) fetches a new and unique (view) identifier, ID, that can be totally ordered by $\preceq_{ct}$ and $ID.wid$ is the identity of the processor that incremented the counter, resulting in counter value ID (hence view IDs are counters as defined in 4.5). 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).

B.1.2 *Determining coordinator availability:*

Algorithm 3 takes an agile approach for multicasting with atomic delivery guarantees. Namely, a new view is installed whenever the coordinator sees a change to its local failure detector, $failureDetector()$, which p_i stores in FD_i (line 5). Processor p_i can see the set of processors, $seemCrd_i$, that are 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_ℓ as a (seemly) coordinator when p_ℓ's proposal view does not includes a majority, p_ℓ is not a member in the view it claims to coordinate and, in the case of Multicast rounds, their view fields 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 $\perp$ if none. As shown in the correctness proof, this helps to detect initially corrupted states where a processor p_i might consider processor $p_j j$ to be its coordinator, but processor j does not consider itself to be the coordinator.

The algorithm considers as a valid coordinator the processor that seems to have 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_ℓ 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 case the p_i is a valid coordinator, it will 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.

B.1.3 ***The coordinator-side:***

Processor p_i is aware of its valid coordinatorship when ($valCrd_i = \{p_i\}$) (line 10). It has to act upon its coordinatorship upon the round end. During a normal Multicast rounds, p_i observes the round end once for every view member p_j it holds that ($rep_i[j].(view,\ status,\ rnd) = (view_i,\ status_i,\ rnd_i)$). For the case of Propose and Install rounds, the algorithm does not need to consider the round number, *rnd*.

Depending on its *status*, the coordinator p_i proceeds once it observes the successful round conclusion. At the end of a normal Multicast round, 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 ($rnd > 2^{64}$), 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 *rnd* part of the state.)

B.1.4 ***The follower-side:***

Processor p_i is aware of its coordinator's identity when ($valCrd_i = \{p_\ell\}$) and $i \neq \ell$ (line 15). It has to act upon merely new messages, i.e., the first message round when installing a new view ($rep[\ell].rnd = 0$), the first time a message arrives ($rnd < rep[\ell].rnd$) or a new view is proposed ($rep[\ell].(view \neq propV)$).

During normal Multicast rounds (line 16) the follower p_i applies the aggregated message of this round to it current automaton state so that it produces the needed side-effects before adopting the coordinator replica (line 19). Note that in the case of a Propose round it avoids overwriting its round number so that the coordinator could know what was the last round number that it delivered during the latest installed view.

B.1.5 ***The exchanging message and PCE optimization:***

Each processor periodically send it 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, 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 send its state to them (lines 17 and 21). Note that the greater the PCE's size, the longer it takes to recover from transient faults. Therefore, one has to take this into consideration when extending the approach of periodic consistency enforcement to other elements of replica, e.g., in *view* and $propV$, one might want to reduce the communication costs that are associated with the *set* field and the epoch part of the ID field.

B.2 Correctness Proof of Algorithm 3

The following remark is used in the correctness proof.

Remark B.1 *As Definition 5.1 suggests, we can have more than one such processor p_ℓ. Note that in this case, it is not necessary to have the* same *supporting majority. Thus for two such processors p_i, p_j, we define the supporting majority of p_i as $P_{maj}(i)$ and we note that $P_{maj}(i) \cap P_{maj}(j) \neq \emptyset$.*

The correctness proof shows that starting from an arbitrary state in an execution R of Algorithm 3 and once the primary partition property (Definition 5.1) holds throughout R, we reach a configuration $c \in R$ in which some processor p_ℓ proposes a view including a majority of processors and this view is accepted by all its member processors. We conclude by proving that any execution suffix in 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_ℓ as the view coordinator (Definition 5.1) its supporting majority can extend the support throughout R and thus p_ℓ continues to emulate the automaton with them. Furthermore, there is no clear guarantee for a view coordinator to continue to coordinate for an unbounded period when it does not meet Definition 5.1's criteria 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 the Algorithm 3 and a processor $p_i \in P$. We define the *local (view) coordinator* of p_i, say p_j, to be the only processor that, based on p_i's local information, has a proposed view satisfying the conditions of lines 6 and 7 such that $valCrd = p_j$. p_j is also considered the *global (view) coordinator* if $\forall p_k \in P_{maj}(j)$ in configuration c, it holds that every p_k's local variable $valCrd = p_j$. When p_i has a (local) coordinator then p_i's local variable $noCrd = \mathsf{False}$, whilst when it has no local coordinator $noCrd = \mathsf{True}$. We are now ready to prove the correctness of the algorithm.

Lemma B.1 *Let c be an arbitrary configuration in an execution, R, of Algorithm 3 such that Definition 5.1 holds. Consider a processor $p_i \in P_{maj}$ which has a local coordinator p_k, such that p_k is either inactive or it does not have a supporting majority throughout R. There is a configuration $c \in R$, after which p_i does not consider p_k to be its local coordinator.*

Proof. There are the two possibilities regarding processor p_k:
Case 1: We first consider the case where p_k is inactive throughout R. By the design of our failure detector, p_i is informed of p_k's inactivity such that line 5 will return an FD_i to p_i where $p_k \notin FD_i$. The threshold we set for our failure detector (Section 2) determines how soon p_k is suspected. By the first condition of line 6 since $p_k \notin seemCrd$ we deduce $p_k \notin valCrd_i$ i.e., p_i stops considering p_k as its local coordinator. It is clear that p_i does not stop suspecting p_k throughout R.

We now turn to the case where p_k is active, however it does not have a supporting majority throughout R, but p_i still considers p_k as its local coordinator, i.e. $valCrd_i = p_k$. Two sub-cases exist:
Case 2(a): p_k considers itself to have a supporting majority, and $p_i \in propV_k$. Note that the latter assumption implies that p_k is forced by lines 20 - 23 to propagate $rep_k[k]$ to p_i in every iteration. By the failure detector, there exists an iteration where p_k will be informed that $|FD| < n/2$. If p_k does not find a new coordinator thus $noCrd_k = \mathsf{True}$, then p_k propagates its $rep_k[k]$ to p_i. But this implies that p_i receives $rep_k[k]$ and stores it in $rep_i[k]$. Upon the next iteration of this reception, p_i will remove p_k from its $seemCrd$ set because p_k does not satisfy the condition $|rep_i[k].FD| < \lfloor n/2 \rfloor$ of line 6. We conclude that p_i seizes considering p_k as its local coordinator if it 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 himself, then it only propagates $rep_k[k]$ to its coordinator. 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 sends $rep_k[k]$ only to its coordinator, and thus the proof of case 2(a) is not helpful. As explained in Algorithm 3, the failure detector carries the identity (pid) of any processor it regards as active, as well as the identity of the local coordinator of each such processor. 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 (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. But p_k does not consider itself as its coordinator (by case 2(b) assumption) and thus it holds that $crd(k) \neq k$. Therefore,

eventually the condition $crd(k) = k$ required in line 6 fails and $p_k \notin seemCrd$ and thus $valCrd_i \neq p_k$. We conclude that p_k stops being p_i's coordinator and by the assumption of definitive suspicions we reach to the result. ∎

Lemma B.2 *If the conditions of Definition 5.1 hold throughout an execution R of Algorithm 3, 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.*

Proof. By Definition 5.1, at least one processor with supporting majority exists. Denote any such processor as p_ℓ. Assume for contradiction that no such processor p_ℓ proposes a view, i.e., it does not hold or create a $propV$ that can satisfy the conditions in lines 6 and 7 of the algorithm. This implies that p_ℓ either has a coordinator that is not global or does not have a coordinator, but also does not know of a majority of processors that do not have a coordinator and thus cannot propose by the third condition of line 9. Note that the first condition in line 9 is satisfied by our assumption that p_ℓ is not suspected by a majority throughout R.

If p_ℓ has a local coordinator, say p_k, then there are two sub-cases: Either this coordinator has a supporting majority or it does not. If this coordinator does not have a supporting majority then by Lemma B.1 the execution will reach a configuration in which p_ℓ does not have p_k as its coordinator.

Therefore, it must be that this coordinator p_k has a supporting majority $P_{maj}(k)$ but is not the global coordinator. If its proposal is the result of a view $propV_k$ that existed in its arbitrary state and it is not accepted by some processors of its majority, then there must exist some condition that based on arbitrary local state, prevents some processor $p_r \in propV_k.set$ to accept $propV_k$. We note that all the conditions that are based on $rep[k]$ must hold for all processors in $propV.set$, since $rep_k[k]$ is propagated by p_k to the whole $propV.set$. Thus any third processor $p_r \in propV_k.set$ that receives this, is able to exclude or include p_k from its $seemCrd$ set in the same way as p_ℓ can, since this becomes common knowledge. There are only two other conditions:
(1) $crdID(k) = k$ in line 6. But this should eventually be true for all processors in FD_k if p_k considers itself as the coordinator.
(2) Condition ($p_j \in rep_r[k].propV.set \Leftrightarrow p_k \in rep_r[j].FD$) requires that a processor p_r will accept $propV_k$ only if its local knowledge of some processor $q \in propV_k.set$ suggests that $p_k \notin FD_q$. This may indeed be true due to corrupt initial state. But if this is the case, then p_k's failure detector will also eventually exclude q from FD_k and by this it will hold for p_k that $valCrd_k \neq p_k$ it is a coordinator. This leads p_k to either choose a different coordinator or set $noCrd_k = True$.

By the same reasoning as in case (2b) of Lemma B.1, p_ℓ will also stop regarding p_k to be a coordinator. If p_ℓ is not entering a $noCrd$ state it must be that it also holds some other proposal that is still valid. But with the same conjectures as above, we can argue that p_ℓ will reject any label from any processor that cannot become the global coordinator. If it can become the global coordinator then this is a contradiction because we assumed that no processor with supporting majority proposes. Thus p_ℓ is forced to set $noCrd = True$. But so will any processor in $P_{maj}(\ell)$. Since $P_{maj} > \lfloor n/2 \rfloor$ some p_ℓ will be able to satisfy the condition in line 9 and propose at least one view. ∎

Lemma B.3 *If the conditions of Definition 5.1 hold throughout an execution R of Algorithm 3, then starting from an arbitrary configuration, the system reaches a configuration in which any processor p_ℓ with a supporting majority may propose itself as the coordinator at most once. As a consequence, the system reaches a configuration in which one of these processors is the global coordinator until the end of the execution.*

Proof. We distinguish the following cases:
Case 1: Assume there is only a single processor p_ℓ that has a supporting majority throughout R. By Lemma B.2 it is deduced that p_ℓ will propose a view $propV_\ell$. Then by Lemma B.1 any other processor that does not have supporting majority will eventually stop being a local coordinator for any $p_j \in P_{maj}(\ell)$, and since they do not have a supporting majority the first condition of line 9 will prevent them from proposing. Assume that p_ℓ proposes for a second time. Given that no other processor can propose, there exists some proposal that has a greatest view identifier than the one p_ℓ has proposed. We note that the increment counter

algorithm will return the greatest identifier of all the previous generated. In cases of concurrent calls, each processor proposes the view with its own view identifier without any guarantees on which is the greatest. Thus, there must have been a second processor that called $inc()$ either concurrently or after p_ℓ. But p_ℓ was the only processor that can propose (is the only one having a supporting majority), hence a second proposal is not possible. By the propagation of the proposal (lines 20-24) we conclude that that every processor in $propV_\ell.set$ will accept $propV_\ell$ (line 19) and p_ℓ will not propose itself again throughout R. Furthermore, since no other processor can propose (the ones in the supporting majority of p_ℓ have a coordinator, and the others, even if they believe p_ℓ is inactive, they cannot form a majority of processors with no coordinator (they are a minority).

Case 2: Consider two processors $p_\ell, p_{\ell'}$ that have a supporting majority such that they each create an ID (line 9), with which they propose a new view. Without loss of generality, we assume that $propV_\ell$ proposed by p_ℓ has the greatest identifier of all the labels created by the concurrent calls to $inc()$. We identify two subcases:

Case 2(a): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. In this case $p_{\ell'}$ will propose its view $propV_{\ell'}$ and wait for all $p_i \in propV_{\ell'}$ to adopt it (line 10). p_ℓ may or may not receive $propV_{\ell'}$ but it ignores it since $propV_{\ell'}.ID \preceq_{ct} propV_\ell.ID$ (line 7). Then $propV_\ell$ is propagated to all $p_i \in propV_\ell.set$. Since there is no other greatest proposed view ID than $propV_\ell.ID$ this is adopted by all $p_i \in propV_\ell$ which also includes $p_{\ell'}$ as well. Thus any processor with supporting majority that belonged to the proposed set of p_ℓ will propose at most once, and p_ℓ will become the sole coordinator.

Case 2(b): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Since both processors were able to propose, this implies that a majority of processors that belonged to each of their supporting majority informed them that it had no coordinator (line 9). Each of these processors proposes its view to its $propV.set$, and waits for acknowledgments *from all* the processors in set $propV.set$ (line 10), in order to install the view. Since $p_\ell \notin FD_{\ell'}$, $p_{\ell'}$ does not consider $propV_\ell$ a valid proposal (line 6) and retains its own proposal that it propagates. The same is done by p_ℓ. Since p_ℓ has the greatest label, any $p_i \in P_{maj}(\ell) \cap P_{maj}(\ell')$ might initially adopt $propV_{\ell'}$ but it will eventually choose the greatest $propV_\ell$. If $p_{\ell'}$'s proposal was accepted by all then this means that $p_{\ell'}$ became the global coordinator but will then lose the coordinatorship because of $propV_\ell$ that will be established with a greater view ID. Nevertheless, $p_{\ell'}$ cannot make another proposal, since it will not have a majority of processors that do not have a coordinator. This is deduced from the intersection property of the two majorities. Since the processors in the instersection $p_i \in P_{maj}(\ell) \cap P_{maj}(\ell')$ have p_ℓ as a coordinator, $p_{\ell'}$ does not satisfy condition $(|\{p_k \in FD_{\ell'} : p'_\ell \in rep[k].FD \wedge rep[k].noCrd\}| > \lfloor n/2 \rfloor))$ of line 9 thus cannot propose a new view. Processor p_ℓ installs its view and thus does not create a new view and remains the sole coordinator. ■

Theorem B.4 *Starting from an arbitrary configuration, any execution R of Algorithm 3 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) and assume that this prefix is sufficiently long for Lemma B.3 to hold. I.e., we reach a configuration c_{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_ℓ and that any processor $p_i \in v$ that is not the coordinator is a *follower* of p_ℓ. We define a *multicast round* to be a sequence of ordered events: $fetch()$ input and propagate to coordinator, coordinator disseminates messages to be delivered in this new round, messages delivered and side effects produced by all processors. Our proof is broken into three steps:

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 $fetch()$ is called only once per round to collect input from the environment. This cannot be changed/overwritten since followers can never access $rep[i] \leftarrow rep[\ell]$ that is the only line

modifying the *input* field, unless the receive a new round number greater than the one they currently hold. We notice that the followers have produced side effects for the previous round (using $apply()$) based on the messages and state of the previous round. Similarly, the coordinator executes $fetch()$ exactly once and only before it populates the msg array and after it has produced the side effects for the environment that were based on the previous messages (line 12). Line 13 populates the msg array with messages and including m. The coordinator then continuously propagates its current replica but cannot change it by our remark and condition ($\forall\ p_i \in v.set : rep_\ell[i].(view,\ status,\ rnd) = (view_\ell,\ status_\ell,\ rnd_\ell)$). This ensures that the coordinator will change its msg array only when every follower has executed line 17 which allows the condition to hold. Any follower that keeps a previous round number does not allow the coordinator to move to the next round. If the coordinator moves to a new round it is implied that $rep[i] \leftarrow rep[\ell]$ and thus message m was received by any follower p_i, by our assumptions that the replica is propagated infinitely often and the data links are stabilizing. Thus, given the assumptions, any message m is certainly delivered in 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_{safe} we move to a new c'_{safe} that has a different view v' but has the same coordinator. Once p_ℓ (the coordinator) is in an iteration where the condition $FD \neq propV.set$ of line 9 holds, a view change is detected. Note that since p_ℓ is the global coordinator, no other processor can propose a new view given that Lemma B.2 know holds. p_ℓ creates a new $propV$ with a new view ID taken from the increment counter algorithm, which is greater than the previous established view ID $v.ID$. The last condition of line 10 guarantees that p_ℓ will not execute lines 12 to 14 and thus will not change its $rep.(state, input, msg)$ fields, until all the expected followers of the proposed view have sent their replicas. Followers that receive the proposal will accept it, since none of the conditions that existed before can change that $valCrd = \{p_\ell\}$, and thus enter status Propose and adopt the new view. What is important is that virtual synchrony is preserved since no follower is changing $rep.(state, input, msg)$ fields, and moreover each sends its replica to the p_ℓ by line 22. Once the replicas of all the followers have been collected, the coordinator creates a consolidated *state* and *msg* array of all messages that where delivered or pending. Interfaces $synchState()$ and $synchMsg$ return a consolidated state and message array, based on the message and the *view* and round number *rnd* it was sent in. p_ℓ's new replica is communicated to the followers who adopt this state as their own. 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 before the view change.
Step 3: Virtual synchrony is preserved in two consecutive view installations where the coordinator changes. We assume that p_ℓ had a supporting majority throughout R'. We define a matching suffix R'' to prefix R', such that R'' results from the loss of supporting majority by p_ℓ. Notice that since Definition 5.1 is required to hold, then some other processor with supporting majority, say p'_ℓ will by Lemma B.2 propose the view v' with the highest view ID. We note that by the intersection property and the fact that a view set can only be formed by a majority set, $\exists p_i \in v \cap v'$. Thus, the "knowledge" of the system, $(state, input, msg)$ is retained within the majority.

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

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