Self-Stabilizing Virtual Synchrony*

Shlomi Dolev[1], Chryssis Georgiou[2], Ioannis Marcoullis[2], and Elad M. Schiller[3]

[1] Dept. of Computer Science, Ben-Gurion University of the Negev, Israel.
[2] Dept. of Computer Science, University of Cyprus, Cyprus.
[3] Dept. of Engineering and Computer Science, Chalmers University of Technology, Sweden.

Abstract. Virtual synchrony (VS) 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 critical for the success of such implementations since large distributed systems can be highly available as long as they do not depend on the full operational status of every system participant. Self-stabilizing systems can tolerate transient faults that drive the system to an arbitrary unpredictable configuration. Such systems automatically regain consistency from any such configuration, and then produce the desired system behavior ensuring it for practically infinite number of successive steps, e.g., 2^{64} steps. We present a new multi-purpose self-stabilizing counter algorithm establishing an efficient practically unbounded counter, that can directly yield a self-stabilizing Multiple-Writer Multiple-Reader (MWMR) register emulation. We use our counter algorithm, together with a self-stabilizing group membership and a self-stabilizing multicast service to devise the *first* practically stabilizing VS algorithm and a self-stabilizing VS-based emulation of state machine replication (SMR). As we base the SMR implementation on VS, rather than consensus, the system progresses in more extreme asynchronous settings in relation to consensus-based SMR.

1 Introduction

Virtual Synchrony (VS) has been proven to be very important in the scope of fault-tolerant distributed systems [4]. The VS property ensures that two or more processors that participate in two consecutive communicating groups should have delivered the same messages. Systems that support the VS abstraction are designed to operate in the presence of fail-stop failures of a minority of the participants. Such a design fits large computer clusters, 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 VS abstraction provide services, such as *group membership* and *reliable group multicast*. The group membership service is responsible for providing the current *group view* of the recently live and connected group members, i.e., a processor set and a unique *view identifier*, which is a sequence number of the view installation. The reliable group multicast allows the service clients to exchange messages with the group members as if it was a single communication end-point with a single network address and to which messages are delivered in an atomic fashion, thus any message is either delivered to all recently live and connected group

* The work of the first author is partially supported by the Rita Altura Trust Chair in Computer Sciences, and the Israel Science Foundation (grant 428/11). The work of the second and third authors is supported by the University of Cyprus.

members prior to the next message, or is not delivered to any member. The challenges related to VS consist of the need to maintain atomic message delivery in the presence of asynchrony and crash failures. VS facilitates the implementation of a replicated state machine [4] that is more efficient than classical consensus-based implementations that start every multicast round with an agreement on the set of recently live and connected processors. It is also usually easier to implement [4]. To the best of our knowledge, no *self-stabilizing virtual synchrony* solution exists.

Transient violations of design assumptions can lead a system to an arbitrary state. For example, the assumption that error detection ensures the arrival of correct messages and the discarding of corrupted messages, might be violated since error detection is a probabilistic mechanism that may not detect a corrupt message. As a result, the message can be regarded as legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, requiring human intervention. In the presence of transient faults, large multicomputer systems providing VS-based services, can prove hard to manage and control. One key problem, not restricted to virtually synchronous systems, is catering for counters (such as view identifiers) reaching an arbitrary value. How can we deal with the fact that transient faults may force counters to wrap around to the zero value and violate important system assumptions and correctness invariants, such as ordering of events? A self-stabilizing algorithm [7] can automatically recover from such 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. We tackle this issue in our work.

Contributions. We present the first self-stabilizing virtual synchrony solution. Specifically, we provide a self-stabilizing counter algorithm using bounded memory and communication bandwidth, and yet (many writers) can increment the counter for an unbounded number of times in the presence of processor crashes and unbounded communication delays. Our counter algorithm is modular with a simple interface for increasing and reading the counter, as well as providing the identifier of the processor that has incremented it. At the heart of our counter algorithm is the underlying labeling algorithm which extends the label scheme of Alon et al. [1] to support multiple writers, whilst the algorithm specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to "discover" the greatest label and discard all obsolete ones. An immediate application of our counter algorithm is a self-stabilizing MWMR register emulation. Our self-stabilizing counter, using the self-stabilizing reliable multicast and membership services yields our self-stabilizing VS solution, which leads to a self-stabilizing VS-based State Machine Replication (SMR) implementation. A full version of this paper can be found in [11].

Related Work. Lamport was the first to introduce SMR, presenting it as an example in [12]. Schneider [14] gave a more generalized approach to the design and implementation of SMR protocols. Group communication services can implement SMR by providing reliable multicast that guarantees VS [3]. Birman et al. were the first to present VS and a series of improvements in the efficiency of ordering protocols [5]. Birman gives a concise account of the evolution of the VS model for SMR in [4].

Research during the last recent decades resulted in an extensive literature on ways to implement VS and SMR, as well as industrial construction of such systems. A

recent research line on (practically) self-stabilizing versions of replicated state machines [9,10,1,6] obtains self-stabilizing replicated state machines in shared memory as well as synchronized and asynchronous message passing systems.

The bounded labeling scheme and the use of practically unbounded sequence numbers proposed in [1], allow the creation of self-stabilizing bounded-size solutions to the never-exhausted counter problem in the restricted case of a single writer. In [6] a self-stabilizing version of Paxos was developed that led to a self-stabilizing consensus-based SMR implementation. To this end, a labeling scheme extending the one of [1] to allow multiple writers. Extracting this scheme for other uses does not seem intuitive. We present a simpler and significantly more communication efficient self-stabilizing (bounded-size never-exhausted) counter that also supports many writers, where a single label rather than a vector of labels needs to be communicated. Our solution is *highly modular* and can be easily used in any similar setting requiring such counters.

Practically-stabilizing VS and self-stabilizing VS are identical when VS is defined by the behaviour of classical VS algorithms that use (bounded) counters. These algorithms preserve the VS requirements as long as the counters do not reach their upper bound. In our setting, if a counter reaches the upper bound due to a transient fault our self-stabilizing/practically-stabilizing solution introduces a new epoch with new sequence numbers. It, thus, converges to act exactly as the non-stabilizing VS (for the same number of steps) as an initialized non-stabilizing VS algorithm.

2 System Settings

We consider an asynchronous message passing system that 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. The system runs on top of a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels [8] and reference therein. 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, thus the number of packets in every given instance is bounded by a constant. When processor p_i sends a packet, π, to processor p_j, the operation $send$ inserts a copy of π into the FIFO queue representing the communication channel from p_i to p_j, while respecting the capacity of the channel, possibly omitting the new packet or one of the already sent packets. Packets are retransmitted until more than the total capacity acknowledgments arrive. Acknowledgments are sent only when a packet arrives (not spontaneously). When p_j receives π from p_i, π is dequeued. We assume that packets can be spontaneously omitted (lost) from the channel, however, a packet that is sent infinitely often is received infinitely often.

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 car-

rying messages back and forth between any two communication entities and use it to implement a reliable multicast procedure, and a failure detector in Section 4

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* of the algorithm 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 thus rate of the timer is totally unknown.

A *(system) configuration* is a tuple of the form $(s_1, s_2, \cdots, s_n)$, where s_i is the state of p_i (including the values of all the variables and all messages in transit to p_i). Each algorithm step can change the processor's state. An *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ is an alternating sequence of system configurations c_x and steps a_x, such that each configuration c_{x+1}, except the initial configuration c_0, is obtained from the preceding configuration c_x by the execution of the step a_x. A *practically infinite execution* [6] 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 *VS property* requires that any two processors sharing the same sequence of views, ought to *deliver* identical message sets in these views. A legal execution of VS is defined in terms of input/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 this majority. Note that in executions lacking a majority, there is no delivery and processing guarantee, but still any delivery and processing is due to a received environment input.

3 Self-stabilizing Labeling Scheme and Counter Increment

In this section we first present the self-stabilizing labeling algorithm for multiple writers and extend this result to obtain self-stabilizing practically unbounded counters.

3.1 Labeling Algorithm for Concurrent Label Creations

Bounded Labeling Scheme. We build on the labeling scheme of Alon et al. [1] to support wait-free multi-writer systems. 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. We extend the label structure of [1] by including the epoch creator's (writer's) identity to break symmetry, to determine the most recent epoch, even when two or more creators concurrently 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 $=_{lb}$ for label identity. Note that $\prec_{lb}$ does not define a total order. For example, when $\ell_i.lCreator = \ell_j.lCreator$ and $(\ell_i.sting \notin \ell_j.Antistings)$ and $(\ell_j.sting \notin \ell_i.Antisting)$ these labels are *incomparable*. We say that a label ℓ ***cancels*** another label ℓ', if either 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) accepts a set of labels as input and returns a new label, greater than all of the input labels. It has the same functionality as the function $Next_b()$ of [1], but it additionally considers the label creator. It builds 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, thus the choice of $|D| = k^2 + 1$ ensures the existence of an integer to be used as the $sting$, which is not part of $Antistings$ of the input labels.

Each processor p_i is required to "clean 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). To achieve this, p_i maintains a bounded FIFO history of such labels that it has recently learned while communicating with the other processors, and creates a greater label by passing the labels in its queue to $nextLabel()$; call this new label p_i's *local maximal label*. Performing the above tasks is aimed at having each processor learn the *globally maximal label*, that is, the label in the system that is the greatest among the local maximal ones and *adopt* it. Unfortunately, when some processors are not active, finding a global maximal becomes challenging, since these processors will not "clean up" their local labels. Active processors have to do this indirectly without knowing which processors are inactive. Note that this is not a concern in [1], since the sole writer is responsible of "cleaning" obsolete labels as long as it is active; once the single writer becomes inactive nothing can be done with respect to new label creation.

Let us explain why obsolete labels from inactive processors are problematic when they are not cleaned (canceled). Consider a system starting in a state that includes a cycle of labels $\ell_1 \prec_{lb} \ell_2 \prec_{lb} \ell_3 \prec_{lb} \ell_1$, all of the same creator, say p_x. If p_x is active, it eventually learns about these labels and creates 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, since the identifier is 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.

As a solution, each processor maintains a bounded queue for each other processor, where a label with $lCreator = j$, is stored in the queue corresponding to processor p_j. Obsolete labels eventually accumulate in these bounded FIFO queues and are never again adopted, ending cyclic adoptions. We show that given a majority of active processors and any initial state, the system eventually converges to a global maximal label.

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** $\langle i, pick(1, D \setminus (newAntistings \cup \{\cup_{\ell_j \in S} \ell_j.Antistings\})), newAntistings \rangle$;

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).

The processor's state: Each processor stores an array of label pairs, $max_i[]$, where $max_i[i]$ refers to p_i's maximal label pair and $max_i[j]$ is 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., 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.

Information exchange between processors: Processor p_i takes a step whenever it receives two pairs $\langle sentMax, lastSent \rangle$ from some other processor, say p_j. 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 p_i receives a pair $\langle sentMax, lastSent \rangle$ from p_j, p_i 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 $lastSent.ml$. In case this $lastSent.ml$ 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 in line 20. Lines 21 to 28 show how p_i processes the two pairs received.

Label processing: Having received a new pair message $\langle sentMax, lastSent \rangle$ from some p_j, processor p_i starts a step by removing *stale* information, i.e., misplaced or doubly represented labels (line 9) in the label storage. When stale information exists, the algorithm empties the entire 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). Based on the new pairs added to the label store, the algorithm determines whether it is possible to cancel a non-canceled label pair (which may well be a 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 creator identity. Line 24 checks whether any pair of the $max_i[]$ array can cancel a record in the label storage, and line 25 removes any canceled records that share the same ml. 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 $\prec_{lb}$-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 $storedLabel_i[i]$.

Correctness proof outline. Consider 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 $\prec_{lb}$-greatest legitimate local label. We then bound on the number of adoptions, first in the absence of label creations and then in their presence.

Lemma 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 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 $|L_i| \leq n(n^2 + m)$.*

Active processors can now be shown to eventually stop adopting or creating labels. We show that incomparable label pairs eventually disappear from the system and thus no new labels are adopted or created, which then implies the existence of a global maximal label. Combining all the above, we deduce that starting from any initial configuration, the system eventually reaches a configuration in which there is a global maximal label.

Theorem 1. *Suppose that there exists at least one processor, $p_u \in P$ 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 $((max_i[j] \prec_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \prec_{lb} \ell_{\max})))$.*

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 2). Thus, by the pigeonhole principle, we know that within a bounded number of steps in R, there is a period during which p_u 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).

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 $max[j].cl \neq \perp$).
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 $legit(lp) = (lp = \langle \bullet, \perp\rangle)$
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**$(\perp)$
12 $canceled(lp) =$ **if** $(\exists lp' \in labels(lp) : ((lp' =_{ml} lp) \wedge \neg legit(lp')))$ **then return**(lp') **else return**$(\perp)$
13 $needsUpdate(j) = (\neg legit(max[j]) \wedge \langle max[j].ml, \perp\rangle \in labels(max[j]))$
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** $storedLabels[i].add(max[i] \leftarrow \langle nextLabel(), \perp\rangle)$ `// 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** $p_j \in P, lp \in storedLabels[j] : (legit(lp) \wedge (notgeq(j, lp) \neq \perp))$ **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** $p_j \in P : (legit(max[j]) \wedge (canceled(max[j]) \neq \perp))$ **do** $max[j] \leftarrow canceled(max[j])$;
27 **if** $legitLabels() \neq \emptyset$ **then** $max[i] \leftarrow \langle \max_{\prec_{lb}}(legitLabels()), \perp\rangle$;
28 **else** $useOwnLabel()$;

During this period, 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 1). Again, by the pigeonhole principle, there is a period during which p_u takes practically infinite steps in R where neither p_i adds a label, $lp_{i_k} =_{lCreator} i$, to the system (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_u takes practically infinite steps, all processors (that take practically infinite steps in R) name the same $\preceq_{lb}$-greatest legitimate label as the theorem statement specifies. ■

3.2 Increment Counter Algorithm

In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection to obtain a practically self-stabilizing counter increment algorithm supporting multiple writers. To do so, we extend the labeling scheme to handle *counters*. A counter cnt is a triplet $\langle lbl, seqn, wid \rangle$, where lbl is an epoch label as defined in the previous subsection, the sequence number $seqn$ is an integer ranging from 0 to 2^b, where b is large enough, say $b = 64$, 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 $cnt_i \prec_{ct} cnt_j \equiv (cnt_i.lbl \prec_{lb} cnt_j.lbl) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn < cnt_j.seqn)) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn = cnt_j.seqn) \wedge (cnt_i.wid < cnt_j.wid))$. When the labels of the two counters are incomparable, the counters are also incomparable.

The relation $\prec_{ct}$ defines a total order (as required by practically unbounded counters) only when processors share a globally maximum label. In this case, processors can increment a shared counter even when attempting to do so concurrently. Note that by the correctness of the labeling algorithm, starting from any initial state, we eventually reach a configuration where the active processors adopt the same maximal label. Thus, the system stabilizes to use a global maximal label, and so the pair of the sequence number and the identifier of the processor who created this sequence number can be used as an unbounded counter, as used, for example, in MWMR register implementations [13].

Let us 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 maintained 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, each label can now yield many different counters. In order to avoid increasing the size of these queues (with respect to the number of elements stored), we only keep the highest sequence number observed for each label (breaking ties with wids).

If there are corrupt counters in the system (from the initial arbitrary state), then they can only force a change of label if their sequence number is *exhausted* (i.e., $seqn \geq 2^b$). Exhausted counters are treated by the algorithm in a similarly to 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).

Algorithm description: 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. In a procedure similar to the labeling algorithm, p_i (eventually) finds the maximal epoch label and the maximal sequence number for this label. In other words, it collects counters and finds the one(s) with the largest global label; there can be more than one such counter. In this case, it returns the one with the highest sequence number, breaking symmetry with the wids. It then checks whether this maximal sequence number is *exhausted*, i.e., if it is equal or greater than 2^b. If so, 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, incrementing it by one, and setting its own identifier as the writer of this new sequence number. It then sends the new counter to all processors, awaiting 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 [13].

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, the label creator identifier and the sequence number writer identifier need not be the same, i.e., if p_i was not the creator of label ℓ. From the above, we have the following correctness result:

Theorem 2. *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.*

Having a self-stabilizing counter increment algorithm, we can 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 writer finds a maximal counter, it increments and associates it with the value to be written. It then communicates this to a majority of processors. The read operation is similar: the reader queries all processors about the maximum counter they are aware of, and waits for a majority to respond. If it does not receive such a counter, it returns $\perp$ so the read has to be repeated; i.e., the system has yet to converge to a maximal label. If a maximal counter exists, it sends this together with the associated value to all the processors, and once it is acknowledged by a majority, it returns the counter with the associated value. The second phase is a standard requirement to preserve the register's consistency [2,13].

4 Virtually Synchronous Stabilizing Replicated State Machine

In this section, we present our practically stabilizing VS algorithm that emulates SMR.

4.1 Preliminaries

As already mentioned, group communication systems providing the VS property implement two main services: a membership service and a reliable multicast service, whilst they assume there is access to an unbounded counter to use as unique view identifiers. We provide these services in a coordinator-based solution, considering a *primary-group*

implementation [5]. To assign view identifiers, we use our counter increment algorithm. Specifically, a counter defines a view identifier, and the counter's writer identifier is that of the view's coordinator. This defines a simple interface with the counter algorithm, which provides an identical output. The output of the coordinator's failure detector defines the set of view members; this helps to maintain a consistent membership among the group members, despite inaccuracies between the various failure detectors. Pairing the coordinator's member set with the counter we obtain a $view$. The coordinator is also responsible for the consistency of the multicast mechanism within the group. We first suggest a possible implementation of a failure detector (to provide membership) and of a reliable multicast service over the self-stabilizing FIFO data link given in Section 2.

Failure detector implementation: Every processor p, maintains a heartbeat integer counter for every other processor q. Whenever p receives the token from q over their data link, p resets q's counter 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 p considers q as inactive. In other words, the failure detector at 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 implementation mentioned in [6]. Note that for the correctness of our VS 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.

Reliable multicast implementation: We use the coordinator, some processor say p_ℓ, 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 p_ℓ decides to collect inputs, it waits for the token to arrive from each group participant. Whenever a token arrives from a participant, p_ℓ 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 p_ℓ 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, p_ℓ combines them and again uses the token to carry the updated information. The coordinator can then proceed to the next round of input collection.

4.2 Self-Stabilizing Virtually Synchrony and SMR Algorithm

We now present our self-stabilizing virtual synchrony and SMR algorithm. The guarantees for VS hold under the assumption that a *primary partition* exists as defined below.

Definition 1 (Primary partition). *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.*

Algorithm 3: Self-stabilizing automaton replication using VS, code for proc. p_i

Constants: PCE (periodic consistency enforcement) number of rounds between global state check;
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;
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.
Do forever begin
  let $FDin = failureDetector()$;
  let $seemCrd = \{p_\ell = rep[\ell].propV.ID.wid \in FD : (|rep[\ell].propV.set| > \lfloor n/2 \rfloor) \wedge (|rep[\ell].FD| > \lfloor n/2 \rfloor) \wedge (p_\ell \in rep[\ell].propV.set) \wedge (p_k \in rep[\ell].propV.set \leftrightarrow p_\ell \in rep[k].FD) \wedge ((rep[\ell].status = \mathsf{Multicast}) \rightarrow rep[\ell].(view = propV)) \wedge crdID(\ell) = \ell\}$;
  let $valCrd = \{p_\ell \in seemCrd : (\forall p_k \in seemCrd : rep[k].propV.ID \preceq_{ct} rep[\ell].propV.ID)\}$;
  $noCrd \leftarrow (|valCrd| \neq 1)$;
  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** $(status, propV) \leftarrow (\mathsf{Propose}, \langle inc(), FD\rangle)$;
  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**
    if $status = \mathsf{Multicast}$ **then**
      $apply(state, msg)$; $input \leftarrow fetch()$;
      foreach $p_j \in P$ **do if** $p_j \in view.set$ **then** $msg[j] \leftarrow rep[j].input$ **else** $msg[j] \leftarrow \perp$;
      $rnd \leftarrow rnd + 1$;
    else if $status = \mathsf{Propose}$ **then** $(state, status, msg) \leftarrow (synchState(rep), \mathsf{Install}, synchMsgs(rep))$;
    else if $status = \mathsf{Install}$ **then** $(view, status, rnd) \leftarrow (propV, \mathsf{Multicast}, 0)$;
  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**
    if $rep[\ell].status = \mathsf{Multicast}$ **then**
      if $rep[\ell].state = \perp$ **then** $rep[\ell].state \leftarrow state$ /* PCE optimization, line 25 */;
      $rep[i] \leftarrow rep[\ell]$; $apply(state, rep[\ell].msg)$; /* for the sake of side-effects */
      $input \leftarrow fetch()$;
    else if $rep[\ell].status = \mathsf{Install}$ **then** $rep[i] \leftarrow rep[\ell]$;
    else if $rep[\ell].status = \mathsf{Propose}$ **then** $(status, propV) \leftarrow rep[\ell].(status, propV)$;
  let $m = rep[i]$ /* sending messages: all to coordinator and coordinator to all */ ;
  if $status = \mathsf{Multicast} \wedge rnd \pmod{PCE} \neq 0$ **then** $m.state \leftarrow \perp$ /* PCE optimization, line 19 */ ;
  let $sendSet = (seemCrd \cup \{p_k \in propV.set : valCrd = \{p_i\}\} \cup \{p_k \in FD : noCrd \vee (status = \mathsf{Propose})\})$
  foreach $p_j \in sendSet$ **do** $send(m)$;
Upon message arrival m **from** p_j **do** $rep[j] \leftarrow m$;

Note that Definition 1, allows for more than one such processor p_ℓ; in this case, it is not necessary for these processors to have the *same* supporting majority.

Algorithm outline. Each participant maintains a replica $rep[]$ of the state machine.We bound the memory used to store the history of the replica by only keeping the encapsulated influence of the history represented by the current state of the replica (variable $state$). Each participant also maintains the last delivered (composite) message, $msg[n]$, ensuring common reliable multicast, in case the coordinator becomes inactive before ensuring delivery by all members of the group.

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 16) and the follower-side (lines 17 to 22) actions and how the two sides exchange messages. Lines 1–3 define the processor's state and interfaces.

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, $failureDetector()$, which p_i stores in FD_i (line 5). Nevertheless, we might reach a configuration without a view coordinator as a result of an arbitrary initial configuration, or of a coordinator becoming inactive. Using the failure detector heartbeat exchange, processors can detect such initially corrupted states. Each participant that detects that it has no coordinator, seeks for potential candidates based on the exchanged information.

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

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

The coordinator-side: Processor p_i is aware of its valid coordinatorship if $(valCrd_i = \{p_i\})$ (line 10). During a normal Multicast round, p_i observes the round end, when 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 and the last delivered messages of each processor to install a synchronized state of the emulated automaton (line 15). After a successful Install round (line 16), the coordinator proceeds to a Multicast round after installing the proposed view and the first round number.

As part of the multicast procedure, the coordinator (line 13), collects inputs (possibly $\perp$) received from the environment and ensures that all group members apply these inputs to the replica producing possible side-effects. The processors need to apply one input at a time, maybe 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.

The follower-side: Processor p_i considers p_ℓ as its coordinator when $(valCrd_i = \{p_\ell\})$ and $i \neq \ell$ (line 17). 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 18) 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 22). Once a processor does not have a coordinator, and 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 27) and store the replicas received (line 28). As an optimization, during normal Multicast rounds, processors transmit their full replica state every *PCE* rounds, where *PCE* is a predefined constant.

Correctness Outline. We show that starting from an arbitrary state in an execution R of Algorithm 3 and once the primary partition property (Definition 1) holds throughout R, we reach a configuration $c \in R$ where some processor p_ℓ proposes a view including a majority of processors and this view is accepted by all its members. We then prove 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. All such processors propose at most once, leading to a unique coordinator. We conclude by proving that any execution suffix in R that begins from such a configuration c will preserve the VS property and implement SMR.

Lemma 3. *If the conditions of Definition 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 3. *Starting from any configuration, an execution R of Algorithm 3 satisfying Definition 1, emulates automaton replication preserving the VS property.*

Proof Sketch. We consider a finite prefix R' of R with an arbitrary configuration c, and a primary partition (as per Definition 1) and assume that this prefix is sufficiently long for Lemma 3 to hold. I.e., we reach a safe configuration in which there exists a global coordinator for a majority of processors. By careful consideration of the code and the way the coordinated multicast steps take place we argue the claim of the theorem. ■

5 Conclusion

We have presented the first self-stabilizing algorithm that guarantees VS, and used it to obtain a self-stabilizing VS-based SMR emulation; within this emulation, the system progresses in more extreme asynchronous executions compared to consensus-based SMRs. A key component of the VS algorithm is a novel modular 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.

References

1. N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *Journal of Computer and System Sciences*, 2015.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
3. A. Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.
4. K. Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, pages 91–120, 2010.
5. K. Birman and R. Van Renesse. *Reliable distributed computing with the Isis toolkit*. Wiley-IEEE Computer society press, Los Alamitos, 1994.
6. P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët. Practically self-stabilizing Paxos replicated state-machine. In *Proc. of NETYS'14*, pages 99–121, 2014.
7. S. Dolev. *Self-Stabilization*, MIT press, 2000.
8. S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Proc. of SSS'12*, pages 133–147, 2012.
9. S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.
10. S. Dolev, L. Lahiani, N. A. Lynch, and T. Nolte. Self-stabilizing mobile node location management and message routing. In *Self-Stabilizing Systems*, pages 96–112, 2005.
11. S. Dolev, C. Georgiou, I. Marcoullis and E. Schiller. Practically Stabilizing Virtual Synchrony. CoRR abs/1502.05183, 2015.
12. L. Lamport. Time, clocks, and the ordering of events in a distributed system. pages 558–565, 1978. *Commun. ACM*, 21(7):558–565, 1978.
13. N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of FTC'1997*, pages 272–281, 1997.
14. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.