Pragmatic Self-Stabilization of Atomic Memory in Message-Passing Systems*

Noga Alon[1], Hagit Attiya[2], Shlomi Dolev[3], Swan Dubois[4], Maria Potop-Butucaru[4], and Sébastien Tixeuil[4]

[1] Sackler School of Mathematics and Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, 69978, Israel
[2] Department of Computer Science, Technion, 32000, Israel
[3] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il
[4] LIP6, Universite Pierre et Marie Curie, Paris 6/INRIA, 7606, France

Abstract. A fault-tolerant and stabilizing simulation of an atomic register is presented. The simulation works in asynchronous message-passing systems, and allows a minority of processes to crash. The simulation stabilizes in a pragmatic manner, by reaching a long execution in which it runs correctly. A key element in the simulation is a new combinatorial construction of a bounded labeling scheme accommodating arbitrary labels, including those not generated by the scheme itself.

1 Introduction

Distributed systems have become an integral part of virtually all computing systems, especially those of large scale. These systems must provide high availability and reliability in the presence of failures, which could be either permanent or transient. A core abstraction for many distributed algorithms *simulates shared memory* [3]; this abstraction allows to take algorithms designed for shared memory, and port them to asynchronous message-passing systems, even in the presence of failures. There has been significant work on creating such simulations, under various types of permanent failures, as well as on exploiting this abstraction in order to derive algorithms for message-passing systems. (See a recent survey [2].)

All these works, however, only consider permanent failures, neglecting to incorporate mechanisms for handling *transient* failures. Such failures may result from incorrect initialization of the system, or from temporary violations of the assumptions made by

* Research of the first author supported in part by an ERC advanced grant, by a USA-Israeli BSF grant, by the Israel Science Foundation. Research of the second author supported in part by the *Israel Science Foundation* (grants number 953/06 and 1227/10). The work started while the second author was visiting EPFL, and the third author was a visiting professor at LIP6. Research supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, Deutsche Telekom, US Air-Force, *Israel Science Foundation* (grant number 428/11), and Rita Altura Trust Chair in Computer Sciences.

the system designer, for example the assumption that a corrupted message is always identified by an error detection code. The ability to automatically resume normal operation following transient failures, namely to be *self-stabilizing* [9], is an essential property that should be integrated into the design and implementation of systems.

This paper presents a stabilizing simulation of an atomic register in asynchronous message-passing systems where a minority of processors may crash. The simulation is based on reads and writes to a (majority) quorum in a system with a fully connected graph topology. A key component of the simulation is a new bounded labeling scheme that needs no initialization, as well as a method for using it when communication links and processes are started at an arbitrary state. To the best of our knowledge, our scheme is the first constructive labeling scheme presenting the above properties.

Overview of our simulation. Attiya, Bar-Noy and Dolev [3] showed how to simulate a single-writer multi-reader (SWMR) atomic register in a message-passing system, supporting two procedures, read and write, for accessing the register. This simple simulation is based on a quorum approach: In a write operation, the writer makes sure that a quorum of processors (consisting of a majority of the processors, in its simplest variant) store its latest value. In a read operation, a reader contacts a quorum of processors, and obtains the latest values they store for the register; in order to ensure that other readers do not miss this value, the reader also makes sure that a quorum stores its return value.

A key ingredient of this scheme is the ability to distinguish between older and newer values of the register; this is achieved by attaching a *sequence number* to each register value. In its simplest form, the sequence number is an unbounded integer, which is increased whenever the writer generates a new value. This solution is appropriate for an *initialized* system, which starts in a consistent configuration, in which all sequence numbers are zero, and are only incremented by the writer or forwarded as is by readers. Pragmatically, a 64-bit sequence number will not wrap around for a number of writes that lasts longer than the life-span of any reasonable system.

However, when there are transient failures in the system, as is the case in the context of self-stabilization, the simulation starts at an uninitialized state, where sequence numbers are not necessarily all zero. It is possible that, due to a transient failure, the sequence numbers hold maximal values when the simulation starts running, and thus, will wrap around very quickly. Traditionally, techniques like distributed reset [5, 6] are used to overcome this problem. However, in asynchronous crash-prone environments the reset may not terminate waiting for the crashed processes to participate. Hence, a reset invocation will not ensure that the sequence numbers are set to zero.

Our solution is to partition the execution of the simulation into *epochs*, namely periods during which the sequence numbers are supposed not to wrap around. Whenever a “corrupted” sequence number is discovered, a new epoch is started, overriding all previous epochs; this repeats until no more corrupted sequence numbers are hidden in the system, and the system stabilizes. In a steady state, after the system stabilizes, it remains in the same epoch (at least until the sequence number wrap around, which is unlikely to happen).

This raises, naturally, the question of identifying epochs. The natural idea, of using integers, is bound to run into the same problems as for the sequence numbers. Instead, we use a *bounded labeling scheme* [14, 18] for the epochs; this is a function for generating labels (in a bounded domain), that guarantees that two labels can be compared to determine the largest among them. Existing labeling schemes, however, assume that labels have specific initial values, and that new labels are introduced only by means of the label generation function. In contrast, transient failures, of the kind the self-stabilizing simulation must withstand, can create incomparable labels, so it is impossible to tell which is the largest among them or to pick a new label that is bigger than all of them.

To address this difficulty, we introduce a bounded labeling scheme that allows to define a label larger than *any set* of labels, provided that its size is bounded. We assume links have bounded capacity, and hence the number of epoch labels initially hidden in the system is bounded.

The writer tracks the set of epoch labels it has seen recently; whenever the writer discovers that its current epoch label is not the largest, or is incomparable to some existing epoch label, the writer generates a new epoch label that is larger than all the epoch labels it has. The number of bits required to represent an epoch label depends on m, the maximal size of the set, and it is in $O(m \log m)$. We ensure that the size of the set is proportional to the total capacity of the communication links, namely, $O(cn^2)$, where c is the bound on the capacity of each link (expressed in number of messages) and n is the number of processors, and hence, each epoch label requires $O((cn^2(\log n + \log c))$ bits.

It is possible to reduce this complexity, making c constant, using a self-stabilizing data-link protocols for communication among the processors for bounded capacity links over FIFO and non-FIFO communication links [10, 15].[1]

We show that, after a bounded number of write operations, the results of reads and writes can be totally ordered in a manner that respects the real-time order of non-overlapping operations, so that the sequence of operations satisfies the semantics of a SWMR register. This holds until the sequence numbers wrap around, as can happen when the unbounded simulation of [3] is deployed in realistic systems, where all values are bounded.

Note that the original design of [3] copes with non-FIFO and unreliable links. We assume that our atomic register simulation runs on top of an optimal stabilizing data-link layer that emulates a reliable FIFO communication channel over unreliable capacity bounded non-FIFO channels [10].

Related work. Self-stabilizing simulation of an single-writer *single-reader* atomic shared register in a message-passing system was presented in [12]. This simulation does not tolerate processor crashes. More recent papers [11, 19] focused on self-stabilizing simulation of shared registers from weaker shared registers. Self-stabilizing timestamps implementations using SWMR atomic registers were suggested in [1, 13]. These simulations already assume the existence of a shared memory, while, in contrast, we simulate a shared SWMR atomic register in a message-passing system.

2 Preliminaries

A *message-passing system* consists of n *processors*, $p_0, p_1, p_2, \ldots, p_{n-1}$, connected by *communication links* through which messages are sent and received. We assume that

[1] Note that these protocols are also snap-stabilizing—starting in an arbitrary configuration, the first invoked send operation succeeds to deliver the message.

the underlying communication graph is completely connected, namely, every pair of processors, p_i and p_j, have a communication link of bounded capacity c.

A processor is modeled as a state machine that executes *steps*. In each step, the processor changes its state, and executes a single communication operation, which is either a *send* message operation or a *receive* message operation. The communication operation changes the state of an attached link, in the obvious manner.

The system *configuration* is a vector of n states, a state for each processors and $2(n^2 - n)$ queues, each bounded by a constant capacity c. Note that in the scope of self-stabilization, where the system copes with an arbitrary starting configuration, there is no deterministic data-link simulation that use bounded memory when the capacity of links is unbounded [12]. Note further that non-FIFO communication links can be accommodated by mimicking FIFO delivery [10].

An *execution* is a sequence of configurations and steps, $E = (C_1, a_1, C_2, a_2 \ldots)$ such that C_i, $i > 1$, is obtained by applying a_{i-1} to C_{i-1}, where a_{i-1} is a step of a single processor, p_j, in the system. Thus, the vector of states, except the state of p_j, in C_{i-1} and C_i are identical. If the single communication operation in a_{i-1} is a send operation from p_j to processor p_k then s_{jk} in C_i is obtained from s_{jk} in C_{i-1} by enqueuing the message sent in a_{i-1}. If the resulting queue s_{jk} exceeds its size, i.e., $|s_{jk}| = c$, then an arbitrary message is deleted from s_{jk}. The rest of the message queues are unchanged. If the single communication operation in a_{i-1} is a receive operation of a (non null) message M, then M (which is the first message to be dequeued from s_{kj} in C_{i-1}) is removed from s_{kj}, all the other queues are unchanged. A receive operation by p_j from p_k may result in a null message even when the s_{kj} is not empty, thus allowing unbounded delay for any particular message. Message losses are modeled by allowing spontaneous message removals from (any place in) the queue. An edge (i, j) is operational if a message sent infinitely often by p_i is received infinitely often by p_j.

Atomic register. For the simulation of a *single writer multi-reader* (SWMR) atomic register, we assume p_0 is the writer and $p_1, p_2, \ldots, p_{n-1}$ are the readers. There is a procedure for executing a write operation by p_0, and procedures for executing read operations by the readers.

Each invocation of a read or write operation translates into a sequence of computation steps, following the appropriate procedure. Concurrent invocations of read and write operations yield an execution in which the computation steps corresponding to invocations by different processors are interleaved. An operation op_1 *precedes* an operation op_2 in this execution, if op_1 returns before op_2 is invoked. Two operations *overlap* if neither of them precedes the other.

Each interleaved execution of an atomic register is required to be *atomic*, namely, equivalent to an execution in which the operations are executed sequentially and the order of non-overlapping operations is preserved [4]. As advocated in [7], the above definition is equivalent to say that the atomic register has to satisfy the following two properties:

– ***Regularity.*** A read operation returns either the value written by the most recent write operation that completes before the read or a value written by a concurrent write.

– ***No new / old inversions.*** If a read operation R returns the value of a concurrent write operation W, then no read operation that is started after R completes returns the value of a write operation that completes before W starts.

Pragmatically stabilizing atomic register. A message passing system simulates an atomic register is a r-pragmatically stabilizing, if there exist an integer $r' > r$, such that every execution with r' write operations has a segment of execution (fragment) with r write operations that satisfies the atomicity requirements. In particular, a large r implies the existence of a long segment with the desired behavior. In the sequel, when no confusing is possible we refer to r-pragmatically stabilizing simply as pragmatically stabilizing.

Pragmatic stabilization is reminiscent of *pseudo-stabilization* [9] in the sense that an execution has a finite number of specification violation during a long execution; in pseudo-stabilization the length of the long execution is infinite while in pragmatic stabilization the length considered is practically infinite. Roughly speaking, the use of the pigeonhole principle ensures that a partition of r' by the bound on the number of violations ensures the existence of r.

3 Overview of the Algorithm

3.1 The Basic Quorum-Based Simulation

We describe the basic simulation, which follows the quorum-based approach of [3], and ensures that our algorithm tolerates (crash) failures of a minority of the processors.

The simulation relies on a set of *read and write quorums*, each being a majority of processors.[2] The simulation specifies the write and read procedures, in terms of QuorumRead and QuorumWrite operations. The QuorumRead procedure sends a request to every processor, for reading a certain local variable of the processor; the procedure terminates with the obtained values, after receiving answers from processors that form a quorum. Similarly, the QuorumWrite procedure sends a value to every processor to be written to a certain local variable of the processor; it terminates when acknowledgments from a quorum are received. If a processor that is inside QuorumRead or QuorumWrite keeps taking steps, then the procedure terminates (possibly with arbitrary values). Furthermore, if a processor starts QuorumRead procedure execution, then the stabilizing data link [15, 16] ensures that a read of a value returns a value held by the read variable some time during its period; similarly, a QuorumWrite(v) procedure execution, causes v to be written to the variable during its period.

Each processor p_i maintains a variable, $MaxSeq_i$, supposed to be the “largest” sequence number the processor has read, and a value v_i, associated with $MaxSeq_i$, which is supposed to be the value of the implemented register.

The write procedure of a value v starts with a QuorumRead of the $MaxSeq_i$ variables; upon receiving answers $l_1, l_2, \ldots$ from a quorum, the writer picks a sequence number l_m that is larger than $MaxSeq_0$ and $l_1, l_2, \ldots$ by one; the writer assigns l_m to

[2] Standard end-to-end schemes [17] can be used to implement the quorum operation in the case of general communication graph.

$MaxSeq_0$ and calls QuorumWrite with the value $\langle l_m, v\rangle$. Whenever a quorum member p_i receives a QuorumWrite request $\langle l, v\rangle$ for which l is larger than $MaxSeq_i$, p_i assigns l to $MaxSeq_i$ and v to v_i.

The read procedure by p_i starts with a QuorumRead of both the $MaxSeq_j$ and the (associated) v_j variables. When p_i receives answers $\langle l_1, v_1\rangle$, $\langle l_2, v_2\rangle \ldots$ from a quorum, p_i finds the largest epoch label l_m among $MaxSeq_i$, and $l_1, l_2, \ldots$ and then calls QuorumWrite with the value $\langle l_m, v_m\rangle$. This ensures that later read operations will return this, or a later, value of the register. When QuorumWrite terminates, after a write quorum acknowledges, p_i assigns l_m to $MaxSeq_i$ and v_m to v_i and returns v_m as the value read from the register.

Note that the QuorumRead operation, beginning the write procedure of p_0, helps to ensure that $MaxSeq_0$ holds the maximal value, as the writer reads the biggest *accessible* value (directly read by the writer, or propagated to a quorum that will be later read by the writer) in the system during any write.

Let $g(C)$ be the number of distinct values greater than $MaxSeq_0$ that are accessible in some configuration C, and let $C_1, C_2, \ldots$ be the configurations in the execution. Since all the processors, except the writer, only copy values and since p_0 can only increment the value of $MaxSeq_0$ it holds for every $i \geq 1$ that

$$g(C_i) \geq g(C_{i+1}) \ .$$

Furthermore,

$$g(C_i) > g(C_{i+1}) \ ,$$

whenever the writer discovers (when executing step a_i) a value greater than $MaxSeq_0$. Roughly speaking, the faster the writer discovers these values, the earlier the system stabilizes. If the writer does not discover such a value, then the (accessible) portion of the system in which its values are repeatedly written, performs reads and writes correctly.

3.2 Epochs

As described in the introduction, it is possible that the sequence numbers wrap around faster than planned, due to "corrupted" initial values. When the writer discovers that this has happened, it opens a new *epoch*, thereby invalidating all sequence numbers from previous epochs.

Epochs are denoted with labels from a bounded domain, using a *bounded labeling scheme*. Such a scheme provides a function to compute a new label, which is "larger" than any given set of labels.

Definition 1. *A* labeling scheme *over a bounded domain $\mathcal{L}$, provides an antisymmetric comparison predicate $\prec_b$ on $\mathcal{L}$ and a function $\mathbf{Next}_b(S)$ that returns a label in $\mathcal{L}$, given some subset $S \subseteq \mathcal{L}$ of size at most m. It is guaranteed that for every $L \in S$, $L \prec_b \mathbf{Next}_b(S)$.*

Note that the labeling scheme of [18], used in the original atomic memory simulation [3], cannot cope with transient failures. Section 4 describes a bounded labeling scheme that accommodates badly initialized labels, namely, those not generated by using $\mathbf{Next}$.

This scheme ensures that if the writer eventually learns about all the epoch labels in the system, it will generate an epoch label greater than all of them. After this point, any read that starts after a write of v is completed (written to a quorum) returns v (or a later value), since the writer will use increasing sequence numbers. The eventual convergence of the labeling scheme depends on invoking $\mathbf{Next}_b$ with a parameter S that is a superset of the epoch labels in the system. Estimating this set is another challenge for the simulation, as described next.

Guessing Game. We explain the intuition of this part of the simulation through the following two-player *guessing game*, between a *finder*, representing the writer, and a *hider*, representing an adversary controlling the system.

– The hider maintains a set of labels $\mathcal{H}$, whose size is at most m (a parameter fixed later).
– The finder does not know $\mathcal{H}$, but it needs to generate a label greater than all labels in $\mathcal{H}$.
– The finder generates a label L and if $\mathcal{H}$ contains a label L', such that it does not hold that $L' \prec_b L$ then the hider exposes L' to the finder.
– In this case, the hider may choose to add L to $\mathcal{H}$, however, it must ensure that the size of $\mathcal{H}$ remains at most m, by removing another label. (The finder is unaware of the hiders decision.)
– If the hider does not expose a new label L' from $\mathcal{H}$, the finder wins this iteration and continues to use L.

The strategy of the finder is based on maintaining a FIFO queue of $2m$ labels, meant to track the most recent labels. The queue starts with arbitrary values, and during the course of the game, it holds up to m recent labels produced by the finder, which turned out to be overruled by existing labels (provided by the hider). The queue also holds up to m labels that were revealed to overrule these labels.

Before the finder chooses a new label, it enqueues its previously chosen label and the label received from the hider in response. Enqueuing a label that is already in the queue pushes the label to the front of the queue; if the bound on the size of the queue is reached, then the oldest label in the queue is dequeued. *This semantics of enqueue is used throughout the paper.*

The finder chooses the next label by applying $\mathbf{Next}$, using as parameter the $2m$ labels in the queue. Intuitively, the queue eventually contains a superset of $\mathcal{H}$, and the finder generates a label greater than all the current labels of the hider.

Clearly, when the finder chooses the ith label, $i > 0$, the $2i$ items in the front of the queue consist of the first i labels generated by the finder, and the first i labels revealed by the hider. This is used to show the following property of the game.

Lemma 1. *After at most $m + 1$ labels, the finder generates a label that is larger than all the labels held by the hider.*

Proof. Note that a response cannot expose a label that has been introduced or previously exposed in the game since the finder always choose a label greater than all labels in the queue. Thus, if the finder does not win when introducing the mth label, all the m labels that the hider had when the game started were exposed and therefore, stored in the queue of the finder together with all the recent m labels introduced by the finder, before the $m + 1$st label is chosen. Therefore, the $m + 1$st label is larger than every label held by the hider, and the finder wins. □

Note that a step of the hider that exposes more than one label unknown to the finder, accelerates the convergence to a winning stage.

4 A Bounded Labeling Scheme with Uninitialized Values

Let $k > 1$ be an integer, and let $K = k^2 + 1$. We consider the set $X = \{1, 2, .., K\}$ and let $\mathcal{L}$ (the set of labels) be the set of all ordered pairs (s, A) where $s \in X$ is called in the sequel the *Sting* of the label, and $A \subseteq X$ has size k and is called in the sequel the *Antistings* of the label. It follows that $|\mathcal{L}| = \binom{K}{k} K = k^{(1+o(1))k}$.

The comparison operator $\prec_b$ among the bounded labels is defined to be:

$$(s_j, A_j) \prec_b (s_i, A_i) \equiv (s_j \in A_i) \wedge (s_i \notin A_j)$$

Note that this operator is antisymmetric by definition, yet may not be defined for every pair (s_i, A_i) and (s_j, A_j) in $\mathcal{L}$ (e.g., $s_j \in A_i$ and $s_i \in A_j$).

We define now a function to compute, given a subset S of at most k labels of $\mathcal{L}$, a new label which is greater (with respect to $\prec_b$) than every label of S. This function, called $\mathbf{Next}_b$ (see the left side of Figure 1) is as follows. Given a subset of k labels $(s_1, A_1), (s_2, A_2), \ldots, (s_k, A_k)$, we take a label (s_i, A_i) that satisfies:

– s_i is an element of X that is not in the union $A_1 \cup A_2 \cup \ldots \cup A_k$ (as the size of each A_s is k, the size of the union is at most k^2, and since X is of size $k^2 + 1$ such an s_i always exists).
– A_i is a subset of size k of X containing all values $(s_1, s_2, \ldots, s_k)$ (if they are not pairwise distinct, add arbitrary elements of X to get a set of size exactly k).

It is simple to compute A_i and s_i given a set S with k labels, and can be done in time linear in the total length of the labels given, i.e., in $O(k^2)$ time.

Lemma 2. *Given a subset S of k labels from $\mathcal{L}$, $(s_i, A_i) = \mathbf{Next}_b(S)$ satisfies:*

$$\forall (s_j, A_j) \in S, (s_j, A_j) \prec_b (s_i, A_i)$$

Proof. Let (s_j, A_j) be an element of S. By construction, $s_j \in A_i$ and $s_i \notin A_j$, and the result follows from the definition of $\prec_b$. □

Timestamps. Each value is tagged with a *timestamp*—a pair (l, i) where l is a bounded label, and i is a sequence number, and integer between 0 and a fixed bound $r \geq 1$.

The $\mathbf{Next}_e$ operator compares between two timestamps, and is described in the right part of Figure 1. Note that in Line 3 of the code we use $\tilde{S}$ for the set of labels (with sequence numbers removed) that appear in S. The comparison operator $\prec_e$ for timestamps is:

$$(x, i) \prec_e (y, j) \equiv x \prec_b y \vee (x = y \wedge i < j)$$

In the sequel, we use $\prec_b$ to compare timestamps only by their labels (ignoring their sequence numbers).

Next$_b$

```
input: $S = (s_1, A_1), (s_2, A_2), \ldots, (s_k, A_k)$: labels set
output: $(s, A)$: label
function: For any $\emptyset \neq S \subseteq X$, $pick(S)$ returns arbitrary
(later defined for particular cases) element of $S$
1: $A := \{s_1\} \cup \{s_2\} \cup \ldots \cup \{s_k\}$
2: while $|A| \neq k$
3:    $A := A \cup \{pick(X \setminus A)\}$
4:    $s := pick\,(X \setminus (A \cup A_1 \cup A_2 \cup \ldots \cup A_k))$
5: return $(s, A)$
```

Next$_e$

```
input: $S$: set of $k$ timestamps
output: $(l, i)$: timestamp
1: if $\exists (l_0, j_0) \in S$ such that
       $\forall (l, j) \in S, (l, j) \neq (l_0, j_0),$
           $(l, j) \prec_e (l_0, j_0) \wedge j_0 < r$
2:    then return $(l_0, j_0 + 1)$
3: else return $(\mathbf{Next}_b(\tilde{S}), 0)$
```

Fig. 1. Next$_b$ and Next$_e$. $\tilde{S}$ is the set of labels appearing in S

5 Putting the Pieces Together

Each processor p_i, holds, in $MaxTS_i$, two fields $\langle ml_i, cl_i \rangle$, where ml_i is the timestamp associated with the last write of a value to the variable v_i and cl_i is a *canceling timestamp* possibly empty ($\perp$), which is not smaller than ml_i in the $\prec_b$ order. The canceling field is used to let the writer (finder in the game) know an evidence on the existence of unknown (non smaller) epoch label. A timestamp (l, i) is an evidence for timestamp (l', j) if and only if $l \not\prec_b l'$. When the writer faces an evidence it changes the current epoch label.

The pseudo code for the read and write procedures appears in Figure 2. Note that in Lines 2 and 10 of the write procedure, an epoch label is enqueued if and only if it is not equal to $MaxTS_0$. Note further, that $Next_e$ in Line 5 of the write procedure, first tries to increment the sequence number of the epoch label in $MaxTS_0$ and if the sequence number already equals to the upper bound r then p_0 enqueues the value of $MaxTS_0$ and uses the updated $epochs$ queue to choose a new value for $MaxTS_0$, which is a new epoch label Next$_b$($epochs$) and sequence number 0.

write$_0(v)$

```
1:$\langle\langle ml_1, cl_1\rangle, v_1\rangle, \langle\langle ml_2, cl_2\rangle, v_2\rangle, \cdots$ :=QuorumRead
2:$\forall i$, if $ml_i \neq MaxTS_0.ml$ then enqueue($epochs, ml_i$)
3:$\forall i$, if $cl_i \neq MaxTS_0.ml$ then enqueue($epochs, cl_i$)
4:if $\forall\ l \in epochs\ l \preceq_e MaxTS_0.ml$ then
5:    $MaxTS_0 := \langle Next_e(MaxTS_0.ml \cup epochs), \perp\rangle$
6:else
7:    enqueue($epochs, MaxTS_0.ml$)
8:    $MaxTS_0 := \langle (Next_b(epochs), 0), \perp\rangle$
9:QuorumWrite($\langle MaxTS_0, v\rangle$)

Upon a request of QuorumWrite $\langle l, v\rangle$
10:if $l \neq MaxTS_0.ml$ then enqueue($epochs, l$)
```

read

```
1:$\langle\langle ml_1, cl_1\rangle, v_1\rangle, \langle\langle ml_2, cl_2\rangle, v_2\rangle, \cdots$ :=QuorumRead
2:if $\exists m$ such that $cl_m = \perp$ and
3:    ($\forall\ i \neq m\ ml_i \prec_e ml_m$ and $cl_i \prec_e ml_m$) then
4:        QuorumWrite$\langle ml_m, v_m\rangle$
5:        return($v_m$)
6:else return($\perp$)

Upon a request of QuorumWrite $\langle l, v\rangle$
7:if $MaxTS_i.ml \prec_e l$ and $MaxTS_i.cl \prec_e l$ then
8:    $MaxTS_i := \langle l, \perp\rangle$
9:    $v_i := v$
10:else if $l \not\prec_b MaxTS_i.ml$ and $MaxTS_i.ml \neq l$
            then $MaxTS_i.cl := l$
```

Fig. 2. write(v) and read

The write of a value v starts with a QuorumRead of the $MaxTS_i$ variables, and upon receiving answers $l_1, l_2, \ldots$ from a quorum, the writer p_0 enqueues the epoch labels of the received ml and non-$\perp$ cl which are not equal to $MaxTS_0$, to the epochs queue (Lines 1-3). The writer then computes $MaxTS_0$ to be the Next_e timestamp, namely if the epoch label of $MaxTS_0$ is the largest in the *epochs* queue and the sequence number of $MaxTS_0$ less than r, then p_0 increments the sequence number of $MaxTS_0$ by one, leaving the epoch label of $MaxTS_0$ unchanged (Lines 4-5). Otherwise, it is necessary to change the epoch label: p_0 enqueues $MaxTS_0$ to the *epochs* queue and applies $Next_b$ to obtain an epoch label greater than all the ones in the $epochs$ queue; it assigns to $MaxTS_0$ the timestamp made of this epoch label and a zero sequence number (Lines 7-8). Finally, p_0 executes the QuorumWrite procedure with $\langle MaxTS_0, v\rangle$ (Line 9).

Whenever the writer p_0 receives (as a quorum member) a QuorumWrite request containing an epoch label that is not equal to $MaxTS_0$, p_0 enqueues the received epoch label in the *epochs* queue (Line 10). (Recall the rules for enqueuing the queue from Section 3.2.)

The read of a reader p_i starts with a QuorumRead of the $MaxTS_j$ and the (associated) v_j variables (Line 1). When p_i receives answers $\langle\langle ml_1, cl_1\rangle, v_1\rangle$, $\langle\langle ml_2, cl_2\rangle, v_2\rangle \ldots$ from a quorum, p_i tries to find a maximal timestamp ml_m according to the $\prec_e$ operator from among ml_i, cl_i, ml_1, cl_1, ml_2, cl_2 .... If p_i finds such maximal timestamp ml_m, then p_i executes the QuorumWrite procedure with $\langle ml_m, v_m\rangle$. Once the QuorumWrite terminates (the members of a quorum acknowledged) p_i assigns $MaxTS_i := \langle ml_m, \perp\rangle$, and $v_i := v_m$ and returns v_m as the value read from the register (Lines 2-5). Otherwise, in case no such maximal value ml_m exists, the read is aborted (Line 6).

When a quorum member p_i receives a QuorumWrite request $\langle l, v\rangle$, it checks whether both $MaxTS_i.ml \prec_b l$ and $MaxTS_i.cl \prec_b l$. If this is the case, then p_i assigns $MaxTS_i := \langle l, \perp\rangle$ and $v_i := v$ (Lines 7-9). Otherwise, p_i checks whether $l \not\prec_b MaxTS_i.ml$ and if so assigns $MaxTS_i.cl := l$ (Line 10). Note that $\perp \prec_b l$, for any l.

Diffusing labels over the data-link. Note that we assume an underlying stabilizing data-link protocol [9, 15]. The data-link protocol is used for repeatedly diffusing the value of $MaxTS$ from one processor to another. If the $MaxTS_i.cl$ of a processor p_i is $\perp$ and p_i receives from processor p_j a $MaxTS_j$ such that $MaxTS_j.ml \not\prec_b MaxTS_i.ml$ then p_i assigns $MaxTS_i.cl := MaxTS_j.ml$, otherwise, when $MaxTS_j.cl \not\prec_b MaxTS_i.ml$ then p_i assigns $MaxTS_i.cl := MaxTS_j.cl$ Note also that the writer will enqueue every diffused value that is different from $MaxTS_0.ml$ (similarly to lines 10 of the reader and the writer, where each of $MaxTS_j.ml$ and $MaxTS_j.cl$ are considered l).

6 Outline of Correctness Proof

The correctness of the simulation is implied by the game and our previous observations, which we can now summarize, recapping the arguments explained in the description of the individual components. Note that the writer may enqueue several unknown epochs in a single write operation and only then introduce a greater epoch, such a scenario will result in a shorter winning strategy in the game as the writer gains more knowledge concerning the existing (hidden) labels before introducing a new epoch.

In the simulation, the finder/writer may introduce new epoch labels even when the hider does not introduce an evidence. We consider a timestamp (l, i) to be an evidence for timestamp (l', j) if and only if $l \not\prec_b l'$. Using a large enough bound r on the sequence number, we ensure that either there is an execution with r writes in which the finder/writer introduces new timestamps with no epoch label change, and therefore with growing sequence numbers, and well-defined timestamp ordering, or a new epoch label is frequently introduced due to the exposure of hidden unknown epoch labels. The last case follows the winning strategy described for the game.

The sequence numbers allow the writer to introduce many timestamps, exponential in the number of bits used to represent r, without storing all of them, as their epoch label is identical. The sequence numbers are a simple extension of the bounded epoch labels just as a least significant digit of a counter; this allows the queues to be proportional to the bounded number of the epoch labels in the system. Thus, either the writer introduces an epoch label greater than any one in the system, and hence will use this epoch label to essentially implement a register for an execution of r writes, or the readers never introduce some existing bigger epoch label letting the writer increment the sequence number practically infinitely often. Note that if the game continues, while the finder is aware of (a superset including) all existing epoch labels and introduces a greater epoch label, there exist an execution of r writes before a new epoch label is introduced.

In the simulation of a SWMR atomic register, following the first write of a timestamp greater than any other timestamp in the system, with a sequence number 0, to a majority quorum, any read in an execution with r writes, will return the last timestamp that has been written to a quorum. In particular, if a reader finds a timestamp introduced by the writer that is larger than all other timestamps but not yet completely written to a majority quorum, the reader assists in completing the write to a majority quorum before returning the read value.

The simulation fails when the set of timestamps does not include a timestamp greater than the rest. That is, read operations may be repeatedly aborted until the writer writes new timestamps. Moreover, a slow reader may store a timestamp unknown to the rest (and in particular to the writer) and eventually introduce the timestamp. In the first case, the convergence of the system is postponed till the writer is aware of a superset of the existing timestamps. In the second case, the system operates correctly, implementing read and write operations, until the timestamp unknown to the rest is introduced.

Each read or write operation requires $O(n)$ messages. The size of the messages is linear in the size of a timestamp, namely the sum of the size of the epoch label and $\log r$. The size of an epoch label is $O(m \log m)$ where m is the size of the *epochs* queue, namely, $O(cn^2)$, where c is the capacity of a communication link.

Note that the size of the *epochs* queue, and with it, the size of an epoch label, is proportional to the number of epoch labels that can be stored in a system configuration. Reducing the link capacity also reduces the number of epoch labels that can be "hidden" in the communication links. This can be achieved by using a stabilizing *data-link* protocol,[10, 15, 16], in a manner similar to the ping-pong mechanism used in [3].

7 Discussion

We have presented a self-stabilizing simulation of a single-writer multi-reader atomic register, in an asynchronous message-passing system in which at most half the processors may crash.

Given our simulation, it is possible to realize a self-stabilizing *replicated state machines* [20]. The self-stabilizing consensus algorithms presented in [13] uses SWMR registers, and our simulation allows to port them to message-passing systems. More generally, our simulation allows the application of any self-stabilizing algorithm that is designed using SWMR registers to work in a message-passing system, where less than the majority of the processors may crash.

Our work leaves open many interesting directions for future research. Note that our algorithms can be initialized [8] to respect the atomicity requirements for the beginning of a practically infinite execution. Still one of the most interesting research directions is to find a stabilizing simulation, which will operate correctly even after sequence numbers wrap around, without an additional convergence period. This seems to mandate a more careful way to track epoch labels, perhaps by incorporating a self-stabilizing analogue of the *viability* construction [3].

Acknowledgments. We thank Ronen Kat and Eli Gafni for helpful discussions.

References

1. Abraham, U.: Self-stabilizing timestamps. Theoretical Computer Science 308(1-3), 449–515 (2003)
2. Attiya, H.: Robust Simulation of Shared Memory: 20 Years After. EATCS Distributed Computing Column (2010)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message-Passing Systems. Journal of the ACM 42(1), 124–142 (1995)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. Wiley Press, Chichester (2004)
5. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-Stabilization by Local Checking and Global Reset. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 326–339. Springer, Heidelberg (1994)
6. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Bounding the Unbounded. In: INFOCOM, pp. 776–783 (1994)
7. Baldoni, R., Bonomi, S., Kermarrec, A.-M., Raynal, M.: Implementing a Register in a Dynamic Distributed System. In: ICDCS (2009)
8. Delaet, S., Dolev, S., Peres, O.: Safe and Eventually Safe: Comparing Stabilizing Algorithms and non-Stabilizing Algorithms on a Common Ground. In: Proc. of the 2009 International Conference on Principles of Distributed Systems (OPODIS), pp. 315–329 (2009)
9. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
10. Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Stabilizing data-link over non-FIFO channels with optimal fault-resilience. Information Processing Letters 111, 912–920 (2011)
11. Dolev, S., Herman, T.: Dijkstra's self-stabilizing algorithm in unsupportive environments. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 67–81. Springer, Heidelberg (2001)

12. Dolev, S., Israeli, A., Moran, S.: Resource Bounds for Self-Stabilizing Message-Driven Protocols. SIAM J. Comput. 26(1), 273–290 (1997)
13. Dolev, S., Kat, R.I., Schiller, E.M.: When Consensus Meets Self-stabilization, Self-stabilizing Failure-Detector, and Replicated State-Machine. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 45–63. Springer, Heidelberg (2006)
14. Dolev, D., Shavit, N.: Bounded Concurrent timestamping. SIAM Journal on Computing 26(2), 418–455 (1997)
15. Dolev, S., Tzachar, N.: Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. Theoretical Computer Science 410(6-7), 514–532 (2009)
16. Dolev, S., Tzachar, N.: Spanders: distributed spanning expanders. In: SAC (2010)
17. Dolev, S., Welch, J.L.: Crash Resilient Communication in Dynamic Networks. IEEE Trans. Computers 46(1), 14–26 (1997)
18. Israeli, A., Li, M.: Bounded timestamps. Distributed Computing 6(4), 205–209 (1993)
19. Johnen, C., Higham, L.: Fault-tolerant Implementations of Regular Registers by Safe Registers with Applications to Networks. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 337–348. Springer, Heidelberg (2008)
20. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
21. Lamport, L.: On Interprocess Communication. Part I: Basic Formalism. Distributed Computing 1(2), 77–85 (1986)