Crash Resilient and Pseudo-Stabilizing Atomic Registers*

Shlomi Dolev[1], Swan Dubois[2], Maria Gradinariu Potop-Butucaru[3], Sébastien Tixeuil[4]

[1] Ben-Gurion University of the Negev, Israel
dolev@cs.bgu.ac.il
[2] EPFL, Switezerland
swan.dubois@epfl.ch
[3] UPMC Sorbonne Universités, France
[4] UPMC Sorbonne Universités & IUF, France
{maria.potop-butucaru,sebastien.tixeuil}@lip6.fr

Abstract. We propose a crash safe and pseudo-stabilizing algorithm for implementing an atomic memory abstraction in a message passing system. Our algorithm is particularly appealing for multi-core architectures where both processors and memory contents (including stale messages in transit) are prone to errors and faults. Our algorithm extends the classical fault-tolerant implementation of atomic memory that was originally proposed by Attiya, Bar-Noy, and Dolev (ABD) to a stabilizing setting where memory can be initially corrupted in an arbitrary manner. The original ABD algorithm provides no guaranties when started in such a corrupted configuration. Interestingly, our scheme preserves the same properties as ABD when there are no transient faults, namely the linearizability of operations. When started in an arbitrarily corrupted initial configuration, we still guarantee eventual yet suffix-closed linearizability.

Keywords: Fault-Tolerance, Pseudo-Stabilization, Atomic Register.

1 Introduction

Distributed computing theory has proven extremely relevant in the daily practice of current networked systems. The important properties in today's distributed systems include availability, reliability, serviceability, and fault-tolerance. The multi-core systems for example have to be able to mask the unexpected yet

* The research of the first author has been supported by the Ministry of Science and Technology, the Institute for Future Defense Technologies Research named for the Medvedi, Shwartzman and Gensler Families, the Israel Internet Association, the Lynne and William Frankel Center for Computer Science at Ben-Gurion University, Rita Altura Trust Chair in Computer Science, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium and MAFAT.

The research of the other authors has been supported in part by ANR project SHAMAN.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 135–150, 2012.

possible faults of processors and memory transient errors. In these architectures applying the classical technique consisting in restarting the system anytime an error or a fault occurs (at least once a day in current systems, but at least once every few minutes –or even seconds– in forecast exascale supercomputers) attains the limits both in terms of energy cost and the time spent in rebooting the system. In these particular systems, fault recovery mechanisms that rely on the paradigm that combines self-stabilization and fault-tolerance techniques at the application level are particularly appealing. *Self-stabilization* [8] is a versatile technique that permits forward recovery from any kind of *transient* fault (*i.e.* there exists a point in the execution after which there is no fault), while *Fault-tolerance* [15] is traditionally used to mask the effect of a limited number of *permanent* faults. Providing core building blocks for application designers (such as atomic memory construction) that are highly resilient to various kinds of failures is essential for the next generation of those systems. However, making distributed systems tolerant to both transient and permanent faults proved difficult [3,17] as impossibility results are expected in many cases.

Related Works. In the context of self-stabilization, the simulation of an atomic single-writer single-reader shared register in a message-passing system was presented in [12]. This simulation does not address the multiple readers case, and does not consider that crash faults of processors may occur in the system during execution. More recent work [11,19] focused on self-stabilizing simulation of shared registers using shared registers with weaker properties than atomicity, and still do not consider crash faults. Self-stabilizing timestamps implementations using single writer multiple readers atomic registers were suggested in [1,13], and assume that there already exists a shared memory abstraction. Most related to our work are [2], where a crash-fault tolerant and "practically" stabilizing scheme for simulating atomic memory in a message passing system is presented. There, practically means that after stabilization, the linearizability is guaranteed for practically infinite time (say time required for a process to execute 2^{64} steps). Still, in every infinite execution suffix of [2], linearizability is violated infinitely often, leaving open the question of suffix-closed linearizability guaranteeing algorithms that are both stabilizing and crash resilient.

Our contribution. In this paper, we answer positively to the open question of [2]. In more details, we propose a crash-safe and pseudo-stabilizing algorithm for implementing an atomic memory abstraction in a message passing system (provided that the writer does not crash before the first "stabilized" read, see below). Pseudo-stabilization guarantees that, starting from any configuration, any execution contains a suffix satisfying linearizability. Hence, pseudo-stabilization is stronger than practical stabilization since we ensure the closure of linearizability.

Our algorithm extends the classical fault-tolerant implementation of atomic memory that was originally proposed in [4] to a stabilizing setting where memory can be initially corrupted in an arbitrary manner. Note that the original algorithm of [4] provides no guarantees when started in such corrupted configuration. Interestingly, we preserve the same properties as the [4] scheme when

there are no transient faults, namely the linearizability of the operations. Additionally, when started in a corrupted initial configuration the algorithm still guarantees eventual yet suffix closed linearizability.

In the current paper, the writer has the major responsibility for updating the last value, unlike [4] where readers assist each other to spread the most up-to-date value. Note that when the system is started in an arbitrary configuration and the writer is crashed before the stabilization, this cascade-like update may lead to executions where the specification is never verified unless an additional mechanism is used. In [2] we used an epoch-based technique in order to circumvent this drawback. However, the solution proposed in [2] respects a weaker specification (*i.e.*, practically stabilization) while the current work respects the pseudo-stabilization specifications.

2 Model and Definitions

This section is devoted to the presentation of the background of this paper. First, we present the distributed system and fault-tolerance model in Sections 2.1 and 2.2, we specify formally our problem in Section 2.3. Finally, we present in details the ABD simulation on which our protocol is built in Section 2.4.

2.1 Message Passing Model

A message-passing distributed system consists of n vertices (*a.k.a.* processes), $v_0, v_1, v_2, \ldots, v_{n-1}$, connected by communication links through which messages are sent and received. Two vertices connected through a communication link are referred in the following as neighboring vertices. The communication graph of the distributed system is assumed to be fully connected (*i.e.* any pair of vertices are neighboring vertices).

We assume in the following that the capacity of each communication link is bounded and that its capacity is c packets (*i.e.* low level messages). We assume that c is known to the protocol. Note that in the scope of self-stabilization, where the system copes with an arbitrary starting configuration, the initial content of each communication link may be arbitrary.

The channels are *unreliable and non-FIFO* (*i.e.* packets may not follow the FIFO order and may be lost). Additionally, their delivery time is unbounded - the system is *asynchronous*. That is, any non lost packet is received in a finite but unbounded time. Each communication link is weakly fair in the sense that if the sender sends infinitely often a packet on the channel, then the receiver receives this packet an infinite number of time. Sending a packet to a channel whose capacity is exhausted (*i.e.* the channel already contains c packets) results in losing a packet (either a packet already in the channel or the packet being sent).

As we deal with arbitrary initial corruptions, a channel may initially contain up to c *ghost packets* (*i.e.* packets that have never been sent and contain arbitrary content).

A vertex is modeled by a state machine that executes steps. Channels are modeled as sets (rather than queues to reflect the non-FIFO order). For example, the c-bounded channel (i, j) (used to send messages from v_i to v_j) is modeled by a c-sized set denoted by s_{ij}.

In each step, a vertex changes its local state (*i.e.* the state of its local memory), and executes a single communication operation, which is either a *send* operation or a *receive* operation. The communication operation changes the state of an attached channel. In case the communication operation is a send operation from v_i to v_j then s_{ij} is a union of s_{ij} in the previous state with the sent packet. If the obtained union does not respect the bound $|s_{ij}| \leq c$ then an arbitrary message in the obtained union is deleted. In case the communication operation is a receive operation of a (non null) packet m (m must exist in s_{ji} of the previous state), then m is removed from s_{ji}. A receive operation by p_i from p_j may result in a null packet even when the s_{ji} is not empty, thus allowing unbounded delay for any particular packet. Packet losses are modeled by allowing spontaneous packet removals from the set.

A configuration of the system is the product of the local states of processes in the system and of their incident channels. An execution is a sequence of configurations, $\sigma = (\gamma_1, \gamma_2, \ldots)$ such that γ_i, $i > 1$, is obtained from γ_{i-1} when at least one process in the system executes a step. We assume that executions are fully asynchronous.

Finally, we assume that the distributed system is simultaneously subject to transient (*i.e.* of finite duration) faults and to (permanent) crash faults (*i.e.* faults in which affected processes stop to execute steps). The number of crash faults is bounded by a constant f. Transient faults may be arbitrary in nature but there exists a point of the execution after that they no longer occur. Hence, we assumed that the processes local state and channels contents are arbitrary in the initial configuration of the system (and that transient faults no longer corrupt the system during the execution).

2.2 Pseudo-Stabilization and Fault-Tolerance

In this paper, we focus on joint tolerance to transient and crash faults. The classical approach for such a tolerance is fault-tolerant self-stabilization (FTSS for short) [3,17] that ensures that the distributed system stabilizes to its specification in a finite time from any arbitrary initial configuration in spite of crash faults. This strong fault tolerance property leads to numerous impossibility results, see *e.g.* [5]. Hence, we choose in this paper a weaker fault tolerance definition, called pseudo-stabilization [6], in which any execution contains a suffix satisfying the specification. Note that, contrarily to self-stabilization, it is not required that this suffix is reached in a finite time.

Definition 1 (Fault-tolerant pseudo-stabilization [7]). *A distributed protocol π is f-fault-tolerant and pseudo-stabilizing (f-ftps for short) for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f crashed vertices has a suffix satisfying spec.*

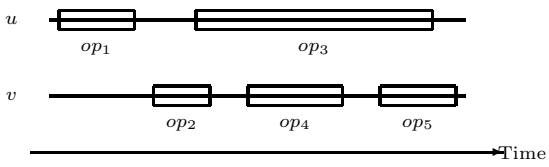

Fig. 1. In this example, op_1 happens before op_2 while op_3 is concurrent with op_2, op_4, and op_5. Operation op_2 and op_4 are consecutive.

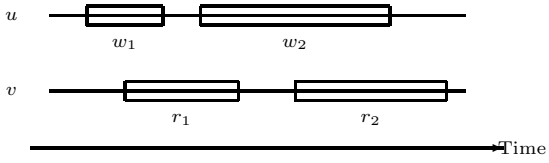

Fig. 2. If r_2 returns the value written by w_1 and r_1 returns the value written by w_2, we have a new/old inversion

2.3 Problem and Specification

In this paper, we emulate an atomic register on top of a message passing system. Registers have been introduced by Lamport [20,21] as a model of communication between vertices of a distributed system. A register is a variable (over a domain D) shared by all vertices of the distributed system that provides two operations: a read operation that returns the value of the register to the invoking vertex and a write operation that allows the invoking vertex to modify the value of the register. Given a register, we call readers the vertices that are able to invoke the read operation of the register and writers the vertices that are able to invoke the write operation of the register. In the following, we consider only single-writer registers. As readers of a register may be distinct from its writer, read and write operations may be interleaved in some executions of the distributed system. Then, we must clarify the result of read operations in such cases. Lamport [20,21] distinguishes three types of registers according to read operation properties: safe, regular and atomic. In the following, we focus on the strongest one, the atomic register.

Note that read and write operations on the register are not instantaneous. Each operation starts when a vertex invokes it and ends when it returns. We say that an operation op_1 happens before an operation op_2 if op_1 ends before op_2 starts. Two operations op_1 and op_2 are concurrent if they satisfy: op_1 does not happen before op_2 and op_2 does not happen before op_1. Two operations op_1 and op_2 are consecutive if op_1 is the most recent operation that happens before op_2. See Figure 1 for an illustration. We introduce now new/old inversions. Consider two consecutive read operations r_1, r_2 and two consecutive write operations w_1, w_2 such that r_1 is concurrent with both w_1 and w_2 and r_2 is concurrent only with w_2 (see Figure 2). We say that a new/old inversion occurs when r_2 returns the value written by w_1 and r_1 returns the value written by w_2.

The writer that is supplied with two operations: *read* and *write* while other vertices, the readers, are supplied with only one operation: *read*. Each *read* invocation needs no parameter and returns a value from D, the domain of the register. Each *write* invocation needs a parameter from D and returns no value. We say that a value v is written to the register when the operation $write(v)$ returns. Intuitively, an atomic register is a register such that all its read and write operations appear as if they have been executed sequentially, this sequential total order respecting the real time order of the operations. More formally, we can define it as follows.

Specification 1 ($spec_{ARS}$). *An execution σ satisfies $spec_{ARS}$ if and only if it complies with the following two properties:*

Regularity: *Each read operation returns either the value written by the most recent write operation that happens before it or a value written by a concurrent write operation.*

No new/old Inversion: *If a read operation r returns a value written by a concurrent write operation w then no read operation that happens after r returns a value written by a write operation that happens before w.*

2.4 The ABD Simulation

This section aims to present in details the fault-tolerant single-writer multi-reader atomic register ABD simulation provided by Attiya, Bar-Noy, and Dolev [4]. Their assumptions on the distributed system follow. They assume a complete identified communication graph (*i.e.* each process has a distinct identifier) and an asynchronous distributed system subject to a minority of crash faults (that is, $2n > f$). Vertex v_0 (also denoted w in the sequel) is the writer (that is, it can invoke both the write and the read operation) while vertices from v_1 to v_{n-1} are readers (that is, they can invoke the read operation only).

In the following, we present only the bounded ABD simulation (the unbounded version makes use of natural numbers to label values of the register and can be easily derived from the bounded version). In this simulation, the authors assume the existence of a sequential bounded labeling system [18]. Israeli and Li defined in [18] time-stamps as "numerical labels which enable a system to keep track of temporal precedence relation among its data elements". Labels are elements of a set enhanced with a total antisymmetric binary relation (to compare labels) and a function to compute a new label given a set of existing labels.

The ABD simulation works as follows. First, they define a communication primitive, called **Communicate**, that ensures the communication by quorum. This primitive broadcasts a given message to all vertices and waits until getting an acknowledgment for a majority of them (it is always possible since at most $\frac{n}{2} - 1$ vertices may crash in any execution). Note that this communication primitive is designed to deal with the properties of the considered message passing model (non reliable and non FIFO communication links).

A label (from the sequential bounded labeling system) is associated to each value of the register. As the labeling system is bounded, the writer must take in

account all existing labels in the distributed system before computing a new one to ensure correctness. Indeed, the new label does not depend only of the writer label as in the unbounded version. Note that the set of gathered labels may be greater and contains obsolete labels.

To reach this goal, the **Write** operation operates as follow. The writer collects (via the primitive **Communicate**) the existing labels in the distributed system (readers send labels that they have for the writer and the most recent labels that they have sent to other vertices). The writer computes then a new label greater than each label it collected. The problem is that the primitive **Communicate** ensures only the collect from a majority of vertices. In consequence, any correct vertex must ensure that its labels are stored at a majority (at least) of vertices at any time. In this way, the writer is able to gather all existing labels when it collects labels from any majority.

To this end, whenever a vertex adopts a new label (that it believes to be the maximum label of the writer), it invokes a procedure **Record** that stores this label and all the recent labels it has sent to other vertices using the primitive **Communicate**. A vertex receiving a recording message simply stores all the labels in its memory. In response to a query from the writer, a reader sends all labels it has stored. This implies that no label may be lost (since a majority of vertices stores these labels). Note that, to avoid chain reaction where a recording message causes other recording messages, vertices ignore the labels carried by recording messages even if their label is greater than their current writer label.

However, when the environment faces both crashes and transient corruptions of the memory the ABD simulation fails to satisfy its specification. This fact is due to the building blocks that compose the ABD simulation: the communication primitive and the labeling scheme and also to the way the labels are included in the viable set. First, the primitive **Communicate** is not resilient to an arbitrary initial content of communication links. Second, the underlying labeling scheme used by the ABD simulation may be unable to compute a new label greater than the existing ones when started in an arbitrary configuration. Finally, the ABD simulation itself cannot deal with arbitrary initialization of labels since some initially corrupted labels may remain unknown to the writer and may be included infinitely often in the **Read** function decision sets.

The next section presents two recent achievements in the area of self-stabilization that allow us to bypass the problems related to the communication primitive and the labeling scheme. Section 4 extends the ABD simulation in order to manage also corrupted labels that have not been generated by the scheme itself but are present in the system due to some transient memory corruptions.

3 Necessary Tools

3.1 Data-Link Protocol

This section sums up the contributions of [10] in which we provided a data-link protocol that ensures optimal fault resiliency above bounded, non-reliable but

fair, non-FIFO communication channels. The main goal is to provide a communication protocol between two vertices that allows us to neglect the actual characteristics of the communication channel. The specification we provide in this paper is borrowed from [22] but we adapt it to the stabilizing context. In particular, we introduce the idea to bound the number of lost, duplicated, ghost and re-ordered messages by some constants.

Consider a system of two vertices v_i and v_j. A distributed application needs to send some messages from v_i to v_j. We say that the application layer of v_i sends a message when it requests the communication protocol to carry this message to v_j. This message is delivered to v_j when the communication protocol releases this message to the application layer of v_j. A ghost message is a message delivered to v_j whereas v_i did not send it previously (due to the arbitrary content of communication channels in the initial configuration). A duplicated message is a message that is delivered several times to v_j whereas v_i sent it only once. A message is lost when v_i sends it but v_j never delivers it. A message m is reordered when it is delivered to v_j before a message m' whereas m has been sent after m' by v_i. Intuitively, the goal of a data-link protocol is to provide a communication black box that ensures there is no lost, duplicated, ghost, or reordered messages during any execution. In the sequel, we formally specify the data-link problem.

Specification 2 (Data-link communication). *For any non negative integers α, β, γ, and δ, the **$(\alpha, \beta, \gamma, \delta)$-Stabilizing Data-Link communication** over c-bounded channels satisfies the following properties starting from an arbitrary configuration (with v_i and v_j being respectively the sender and the receiver) for any execution σ:*

*- **α-Loss:** The first α messages sent by v_i (in the worst case) may be lost.*
*- **β-Duplication:** The first β messages delivered to v_j (in the worst case) may be duplicated ones.*
*- **γ-Creation:** The first γ messages delivered to v_j (in the worst case) may be ghost messages.*
*- **δ-Reordering:** The first δ messages delivered to v_j (in the worst case) may be reordered.*

In [10], we proved that it is impossible to perform a $(\alpha, \beta, \gamma, \delta)$-Stabilizing Data-Link communication with $\beta = 0$, $\gamma = 0$, or $\delta = 0$. We also provided a data-link protocol (called $\mathcal{SDL}$) that achieves this optimal fault-resiliency.

In the following of this paper, we reuse this data-link protocol that provides to each vertex several functions. For each neighbor v_j, a vertex v_i is supplied with two functions: $\mathcal{SDL}$-**Send**$_j(m)$ that allows v_i to send messages to v_j using $\mathcal{SDL}$ and **DeliverMessage**$_i(m)$ that allows v_i to receive messages sent by v_j using $\mathcal{SDL}$.

3.2 Bounded Labeling Scheme

To the best of our knowledge, any existing bounded labeling system including the scheme used in the ABD simulation ([18,9,16]) does not tolerate corrupted initial

Algorithm 1. $\mathcal{PSARS}$: FTPS single-writer multi-reader atomic register simulation (read operation for any vertex v_i, write operation for the writer $w = v_0$).

Variables:
L_i: a matrix $n \times n$ with the following constraints:
- For any $j \neq k$, the element $L_i[j,k]$ contains two fields: $L_i[j,k].sent$ and $L_i[j,k].ack$. The first field is the last label that v_j sent to v_k in the last **Read** operation of v_j known at v_i. The second field contains the last label known at v_i sent by v_j to v_k when v_j replied to the v_k label request.
- For any j, the element $L_i[j,j]$ has two fields. The field $L_i[j,j].value$ provides information on the last label of the writer known by v_j. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer.
$label\_set_i$: a set of labels
Functions:
MaxLabel: returns the maximum label (according to $\prec$) of the label set supplied as parameter if it exists, $\perp$ otherwise
Next: returns a label greater than (according to $\prec$) any label of the set given as parameter
PickValue: returns an arbitrary element of any circuit (according to $\prec$) of the label set supplied as parameter if possible, $\perp$ otherwise

Read$_i$()

```
01: label_set_i := ReadQuorum_i(read)
02: if MaxLabel(label_set_i) ≠ ⊥ then
03:   if L_i[i,i].value ≺ MaxLabel(label_set_i)
04:     L_i[i,i].value := MaxLabel(label_set_i)
05:     L_i[i,i].conflict := ⊥
06:     WriteQuorumPromote_i()
07:   WriteQuorumRecord_i()
08:   return L_i[i,i].value
09: else
10:   L_i[i,i].conflict := PickValue(label_set_i)
11:   WriteQuorumRecord_i()
12:   return abort
```

Write$_0$()

```
01: label_set_0 := ReadQuorum_0(write)
02: L_0(0,0).value := Next(label_set_0)
03: WriteQuorumPromote_0()
```

configurations. We defined and provided in [2] for the first time a stabilizing bounded labeling system: for any subset of at most k labels, there exists a label that dominates each label of the subset. In this way, we are ensured that a stabilizing bounded labeling system can deal with any arbitrary initialization since it is always possible to compute a label greater than the existing ones. We can define formally a stabilizing bounded labeling system in the following way:

Definition 2 (Stabilizing bounded labeling system). *A k-stabilizing bounded labeling system ($k \geq 2$) is a triplet $(L, \prec, next)$ where L is a finite set, $\prec$ is a total antisymmetric binary relation over L and $next$ is a function $next : L^k \to L$ such that:*

$$\forall L' \subseteq L, |L'| \leq k \Rightarrow \forall \ell \in L', \ell \prec next(L')$$

4 Our FTPS Simulation

This section proposes our extension to the ABD simulation that can tolerate, in addition to permanent crash faults, any transient memory corruption. We present a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation over the message passing model. As far as we know, it is the first

time when a simulation with such strong guarantees is designed. Note that our previous work, [2], proposed a simulation that satisfies a weaker property than the pseudo-stabilization. That is, in each infinite run of system the atomicity specification is violated infinitely often. The significant amelioration of our current simulation stems from guaranteeing that each infinite run of the system has an infinite suffix where the atomicity specification is satisfied. First, we describe our distributed protocol in Section 4.1. We prove its correctness and provide its space complexity in Section 4.2.

4.1 Distributed Protocol

As we previously claimed, our distributed protocol is the pseudo-stabilizing version of the ABD simulation presented in details in Section 2.4. In this section, we explain first the differences between our simulation and the ABD simulation. Then, we present formally our distributed protocol. Note that, for the sake of simplicity, we ignore the actual value of the register and we concentrate only on the label associated to it (as in [4]).

Recall that we assume an asynchronous distributed system simultaneously subject to transient and (permanent) crash faults (with a maximal number of crashed vertices f such that $2n > f$). The communication graph is complete and identified. One vertex is distinguished to be the writer. We denote this vertex by $w = v_0$. Vertices from v_1 to v_{n-1} are readers. We also assume that any pair of vertices are able to communicate using the data-link protocol defined in Section 2. More precisely, if a vertex v_i has a message m to send to v_j, it invokes $\mathcal{SDL}$-**Send**$_j(m)$. The data-link protocol delivers this message to v_j by invoking **DeliverMessage**$_i(m)$. Finally, we assume the existence of a stabilizing bounded labeling system as the one described in Section 2. This labeling system provides a set of labels L and two functions. The first one, **Next**, computes a label greater than (according to $\prec$) any label of the set given as parameter. The second one, **MaxLabel**, returns the maximum label (according to $\prec$) of the label set supplied as parameter if this maximum exists, $\perp$ otherwise. Note that **MaxLabel** returns $\perp$ when there exists a circuit in the set of labels supplied as parameter (that is, there exists a subset of labels $\ell_0, \ldots, \ell_t$ such that $\ell_0 \prec \ell_1 \prec \ldots \prec \ell_t \prec \ell_0$).

Our distributed protocol makes use of a similar data structure as the ABD simulation. Each vertex v_i stores an $n \times n$ label matrix L_i. For any $j \neq k$, the element $L_i[j,k]$ contains the same fields as in the ABD simulation: $L_i[j,k].sent$ and $L_i[j,k].ack$. The ith row $L_i[i]$ is updated dynamically by v_i according to messages it sends while other rows $L_i[j]$ ($j \neq i$) are updated by messages that v_i received from v_j (that is, $L_i[j]$ is the latest view of v_i on $L_j[j]$). Each element $L_i[i,j]$ (for $j \neq i$), contains two fields: $L_i[i,j].sent$ and $L_i[i,j].ack$ that store respectively the last label that v_i sent to v_j and the last label acknowledged by v_j to v_i.

The only difference with the ABD simulation matrix is that, for any j, the element $L_i[j,j]$ contains now two fields: $L_i[j,j].value$ and $L_i[j,j].conflict$. The field $L_i[j,j].value$ provides the last label of the writer known by v_j. In particular $L_i[i,i].value$ contains the last label of the writer that the v_i is aware. Note that

this field is equivalent to the field $L_i[j,j]$ of the ABD simulation. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer. This field is used to avoid that some initially corrupted label remains unknown to the writer but is included infinitely often in **Read** function decision set.

Our distributed protocol is composed of two primitives: **Read** (for any vertex) and **Write** (only for the writer v_0). When a reader v_i invokes its **Read** primitive, it collects first the labels of at least a majority of vertices and computes the maximum with **MaxLabel**. Two cases can appear:
1) MaxLabel returns a label. This value (if it exceeds the current label of the reader) is recorded in the distributed system in order to refresh the views of the other vertices on the last label of v_i. Note that, after the reception of this new value, a vertex updates the corresponding entry in its matrix. Vertex v_i finishes its **Read** operation by promoting its value in the distributed system. Upon the reception of the value to be promoted, the vertex v_j compares its current label with the label of the received value. If its local value is obsolete (the local label is less than the received label), then v_j adopts the new value and pushes it in the distributed system.
2) MaxLabel returns bottom whenever the maximum cannot be computed (when the set of collected labels contains a circuit). Then, the **Read** operation aborts. The circuit in the label set may have been introduced either by a corrupted label present in the system at the initialization or by the writer that computed the next label based on partial information from the non stabilized system. Then, the reader changes its $L_i[i,i].conflict$ field to one of the labels that form a circuit. The idea is to help in revealing all the corrupted labels. Indeed, the conflicting value is then recorded in the matrices of a majority of vertices that prevents such conflicting values to disturb further **Read** operations. This case is the main difference with the ABD simulation.

The **Write** operation is similar to the one of the ABD simulation. When the writer invokes this primitive, it first collects the latest labels in the system (by asking any majority of vertices), then computes its next label using the **Next** function. Finally it starts a promotion of the new value in the distributed system.

Algorithms 1 and 2 provide the formal implementation of our fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation.

4.2 Proof of Correctness

This section is devoted to the proof of the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$ for $spec_{ARS}$. According to properties of our data-link protocol described in Section 2, we know that any execution has an infinite suffix in which no ghost, duplicated or re-ordered messages are delivered (since there is only a finite number of communication links in the distributed system). We can conclude that any execution has an infinite suffix in which any delivered message was actually sent. For the sake of simplicity, we consider only such suffixes of executions in the sequel of this proof. Note that this assumption does not restrict the generality of the proof since we want to prove the pseudo-stabilization of our

Algorithm 2. $\mathcal{PSARS}$: Auxiliary functions (for any vertex v_i).

Notations:
For any j, the notation $L_i[j]$ represents the jth row of the matrix L_i.
Variables:
$return\_set_i$: a set of labels
$read\_answer_i$: array of n booleans
$record\_answer_i$: array of n booleans
$promote\_answer_i$: array of n booleans

ReadQuorum$_i(type)$

01: $read\_answer_i := [0, 0, \ldots, 0]$
02: $read\_answer_i[i] := 1$
03: $return\_set_i := \emptyset$
04: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
05: $\quad \mathcal{SDL}$-**Send**$_j(Inquiry(type))$
06: while $|\{j, read\_answer_i[j] = 1\}| \leq n/2$ do
07: $\quad$ wait
08: return $(return\_set_i)$

upon **DeliverMessage**$_j(Inquiry(type))$
09: if $type =' read'$ then
10: $\quad \mathcal{SDL}$-**Send**$_j(Answer\_Read(L_i[i, i]))$
11: $\quad L_i[i, j].ack := L_i[i, i].value$
12: $\quad$ **WriteQuorumRecord**$_i()$
13: else
14: $\quad \mathcal{SDL}$-**Send**$_j(Answer\_Write(L_i))$

upon **DeliverMessage**$_j(Answer\_Read(L_j[j, j]))$
15: $L_i[j, j] := L_j[j, j]$
16: $read\_answer_i[i] := 1$
17: $return\_set_i := return\_set_i \cup L_i$

upon **DeliverMessage**$_j(Answer\_Write(L_j))$
18: $L_i[j] := L_j[j]$
19: $read\_answer_i[i] := 1$
20: $return\_set_i := return\_set_i \cup L_i \cup L_j$

WriteQuorumPromote$_i()$

01: $promote\_answer_i := [0, 0, \ldots, 0]$
02: $promote\_answer_i[i] := 1$
03: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
04: $\quad \mathcal{SDL}$-**Send**$_j(Promote(L_i[i, i]))$
05: while $|\{j, promote\_answer_i[j] = 1\}| \leq n/2$
06: $\quad$ wait
07: foreach $promote\_answer_i[j] \neq 0$ do
08: $\quad L_i[i, j].sent := L_i[i, i].value$

upon **DeliverMessage**$_j(Promote(L_j[j, j]))$
10: if $L_i[i, i].value \prec L_j[j, j].value$ then
11: $\quad L_i[i, i] := L_j[j, j]$
12: $\quad$ **WriteQuorumRecord**$_i()$
13: $\quad \mathcal{SDL}$-**Send**$_j(Ack\_Promote())$

upon **DeliverMessage**$_j(Ack\_Promote())$
14: $promote\_answer_i[j] := 1$

WriteQuorumRecord$_i()$

01: $record\_answer_i := [0, 0, \ldots, 0]$
02: $record\_answer_i[i] := 1$
03: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
04: $\quad \mathcal{SDL}$-**Send**$_j(Record(L_i[i]))$
05: while $|\{j, record\_answer_i[j] = 1\}| \leq n/2$
06: $\quad$ wait

upon **DeliverMessage**$_j(Record(L_j[j]))$
07: $L_i[j] := L_j[j]$
08: $\mathcal{SDL}$-**Send**$_j(Ack\_Record())$

upon **DeliverMessage**$_j(Ack\_Record())$
09: $record\_answer_i[j] := 1$

distributed protocol (that is, only the existence of an infinite suffix satisfying the specification, not the finiteness of a prefix that does not satisfy the specification).

The main difficulty in proving our atomic register simulation comes from the presence of corrupted labels (due to the arbitrary initialization of matrices) in the distributed system that may disturb the good functioning of the distributed protocol.

The key idea of our proof is to show that the writer includes in its decision set (records) all the viable labels in the system (defined below). A label ℓ is *viable* and in the responsibility of vertex v_i if it satisfies one of the following properties:

- $L_i[i, i].value = \ell$ or $L_i[i, i].conflict = \ell$
- $L_i[i, k].sent = \ell$ or $L_i[i, k].ack = \ell$

- there is a vertex v_j such that $L_j[i]$ contains ℓ in one of the fields *sent*, *ack*, *value* or $conflict$.

A viable label is *recorded* if this label is stored in the writer matrix or the matrix of any majority of vertices. In the following, we show that any label in the responsibility of a vertex eventually becomes recorded. Note that once a label is stored in the matrix of the writer or in the matrix of a majority of vertices, this label is included in the computation of the new label of the writer and it does not generate new conflicts.

This observation motivates the following necessary assumption for the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$: if the writer crashes in an execution, then this crash must happen after the first stabilized **Write** invocation (that is, a **Write** invocation during which the label set supplied to **Next** includes all the viable labels in the distributed system). In other words, an execution has an infinite suffix that satisfies $spec_{ARS}$ if the writer does not crash during this execution or if the writer crashes after the first stabilized **Write** invocation (we cannot provide any properties in the contrary case). In the sequel of this section, we consider only such executions. Otherwise, corrupted labels may generate incoherent read outputs. Note that when started in a correct state this assumption is not necessary and the behavior of our simulation is exactly the same as the ABD's simulation. Also note that the ABD simulation cannot cope with corrupted labels.

Lemma 1. *Any execution of $\mathcal{PSARS}$ has an infinite suffix where every **Read** invocation does not abort if $n > 2f$.*

Lemma 2. *Any execution of $\mathcal{PSARS}$ has an infinite suffix where, for any vertex, the labels in its responsibility become recorded either at the writer or in a majority, or are never included in the label set of a read operation if $n > 2f$.*

From now, a viable label refers only to labels that do not stay forever out of the computation.

Lemma 3. *Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the regularity property of $spec_{ARS}$ if $n > 2f$.*

Proof. Let σ be an infinite execution of $\mathcal{PSARS}$. Following Lemma 1 and Lemma 2, σ contains an infinite suffix, σ', where no **Read** invocation aborts and any **Write** operation includes in its decision set all the viable labels in the distributed system. By contradiction, assume there is a vertex v_i such that its **Read** invocations return an obsolete label infinitely often in σ'.

That is, there exists a **Read** invocation r by v_i such that the label returned by r is either a corrupted label or a label corresponding to a previous write but not the most recent. In σ', r returns the output value of **MaxLabel** invoked over the set of labels returned by **ReadQuorum**.

Let w_1 and w_2 be two **Write** operations such that w_1 happens before w_2 and r. Since w_1 happens before r then the label computed by w_1 is promoted and recorded in at least a majority of vertices and is greater than any label in

the distributed system. When r starts invoking **ReadQuorum** two cases may appear: (i) w_2 did not modify the writer label and did not start the promotion of the new label via **WriteQuorumPromote** or (ii) w_2 executed **WriteQuorumPromote**. In the first case, w_1's label is the largest label in the distributed system. When r invokes the **ReadQuorum**, it gets w_1's label (otherwise w_1 is not terminated) and returns this label. Hence, r cannot return a value older than the one written by w_1. In the second case, some vertices contacted during the **ReadQuorum** execution may send the w_1's label, other vertices the w_2's label. Since the label computed in w_2 is greater than the label computed in w_1, **MaxLabel** invoked in r returns w_2's label. Hence, r returns the last written value, that contradicts its construction.

Lemma 4. *Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the no new/old inversion property of $spec_{ARS}$ if $n > 2f$.*

Proof. Let σ be an execution of $\mathcal{PSARS}$. Following Lemmas 1 and 3, σ has an infinite suffix, σ', that satisfies the regularity property of $spec_{ARS}$ and in which any **Read** invocation does not abort. In the following, we prove that σ' does not violate the new/old inversion property of $spec_{ARS}$.

Consider two **Write** operations w_1 and w_2 in σ' such that w_1 happens before w_2. Consider also two **Read** operations r_1 and r_2 such that r_1 happens before r_2 and w_1 happens before r_1 (following the transitivity of the relation "happens before", w_1 also happens before r_2). Assume that r_1 and r_2 are concurrent with w_2 and that a new/old inversion happens. That is, r_1 returns the label ℓ_2 written by w_2 and r_2 returns the label ℓ_1 written by w_1.

Since r_1 happens before r_2, then r_1 executes the following actions (before the start of r_2): it modifies its local label to ℓ_2, it also executes **WriteQuorumPromote** in order to help w_2 to push its label in the distributed system and finally it executes **WriteQuorumRecord** in order to inform the distributed system on its new value. Since **WriteQuorumPromote** returns before r_1 finishes, then the label ℓ_2 is already adopted by at least a majority of vertices. That is, since $\ell_2 \succ \ell_1$ (w_1 happens before w_2), then ℓ_2 replaces ℓ_1 in the matrices of at least a majority of vertices and also a majority of vertices proceeds to the record of their new label.

We assumed r_2 returns ℓ_1. Since r_1 happens before r_2 then r_2 starts its **ReadQuorum** after r_1 returned, in particular after r_1 completed its **WriteQuorumPromote** operation. This implies that ℓ_2 is the label adopted by at least a majority of vertices and at least one vertex in this majority responds while r_2 invokes its **ReadQuorum**. That is, r_2 collects at least one label ℓ_2 and since $\ell_2 \succ \ell_1$, r_2 should return this value. This contradicts the assumption r_2 returns ℓ_1. It follows that σ' satisfies the no new/old inversion property of $spec_{ARS}$.

Lemma 5. *$\mathcal{PSARS}$ requires $O(n^5 \times log_2(n))$ bits per vertex. Consequently, the total amount of memory on the distributed system is in $O(n^6 \times log_2(n))$ bits.*

Proof. Note that the set *label_set* which is the input of **Next** contains $2n^3$ labels. Hence, following [2], one label needs $O(n^3 \times log_2(n))$ bits to be stored. Since any vertex must store $2n^2$ labels, we have the result.

Theorem 1. *$\mathcal{PSARS}$ is a f-ftps distributed protocol for $spec_{ARS}$ provided that $n > 2f$ and that the writer can crash only after its first stabilized **Write** invocation. It requires $O(n^6 log_2(n))$ bits of memory on the whole distributed system.*

5 Conclusion

We presented a distributed solution for implementing a shared register in a network where processors communicate by exchanging messages. To our knowledge, this is the first such construction to be both pseudo-stabilizing and fault tolerant. Note that our simulation verifies also the eventual linearizability specification [14,23]. Differently from the eventual linearizable simulations proposed so far our simulation tolerates initial corrupted memory. Also, we do not reorder operations nor maintain locally the history of the system execution.

We expect future research to tackle the following open issues. A generalization to the multi-writer (and multi-reader) case looks challenging. Indeed, previous transformers for the crash fault model do handle memory corruption, and the multiplicity of writers enable the possibility that fake writers (*i.e.* stale writer identifiers) are initially present in the network.

References

1. Abraham, U.: Self-stabilizing timestamps. Theoretical Computer Science 308(1-3), 449–515 (2003)
2. Alon, N., Attiya, H., Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Pragmatic Self-stabilization of Atomic Memory in Message-Passing Systems. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 19–31. Springer, Heidelberg (2011)
3. Anagnostou, E., Hadzilacos, V.: Tolerating Transient and Permanent Failures (Extended Abstract). In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
4. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. Journal of the ACM 42(1), 124–142 (1995)
5. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self stabilization: impossibility results and solutions using self-stabilizing failure detectors. IJSS 28(11), 1177–1187 (1997)
6. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. DC 7(1), 35–42 (1993)
7. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Stabilizing leader election in partial synchronous systems with crash failures. JPDC 70(1), 45–58 (2010)
8. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. CACM 17(11), 643–644 (1974)
9. Dolev, D., Shavit, N.: Bounded concurrent time-stamping. SIAM J. on Comp. 26(2), 418–455 (1997)
10. Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Stabilizing data-link over non-fifo channels with optimal fault-resilience. IPL 111(18), 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.: Uniform dynamic self-stabilizing leader election. IEEE TPDS 8(4), 424–440 (1997)
13. Dolev, S., Kat, R.I., Schiller, E.M.: When consensus meets self-stabilization. JCSC 76(8), 884–900 (2010)
14. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. TCS 220(1), 113–156 (1999)
15. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of ACM 32(2), 374–382 (1985)
16. Gawlick, R., Lynch, N., Shavit, N.: Concurrent Timestamping Made Simple. In: Dolev, D., Rodeh, M., Galil, Z. (eds.) ISTCS 1992. LNCS, vol. 601, pp. 171–183. Springer, Heidelberg (1992)
17. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC 1993, pp. 195–206 (1993)
18. Israeli, A., Li, M.: Bounded time-stamps. DC 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.: On interprocess communication. Part i: Basic formalism. DC 1(2), 77–85 (1986)
21. Lamport, L.: On interprocess communication. Part ii: Algorithms. DC 1(2), 86–101 (1986)
22. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
23. Serafini, M., Dobre, D., Majuntke, M., Bokor, P., Suri, N.: Eventually linearizable shared objects. In: PODC 2010, pp. 95–104 (2010)