Tight Self-Stabilizing Mobile Byzantine-Tolerant Atomic Register

Silvia Bonomi[⋆], Antonella Del Pozzo[⋆†], Maria Potop-Butucaru[†]

[⋆]Sapienza Università di Roma,Via Ariosto 25, 00185 Roma, Italy
{bonomi, delpozzo}@dis.uniroma1.it
[†]Université Pierre & Marie Curie (UPMC) – Paris 6, France
{maria.potop-butucaru, antonella.del-pozzo}@lip6.fr

October 23, 2015

Abstract

This paper proposes the first implementation of a self- stabilizing atomic register that is tolerant to both *Mobile Byzantine Agents* and *transient failures*. The register is maintained by n servers and our algorithm tolerates (i) any number of transient failures and (ii) up to f Mobile Byzantine Failures. In the Mobile Byzantine Failure model, faulty agents move from one server to another and when they are affecting a server, it behaves arbitrarily. Our implementation is designed for the round-based synchronous model where agents are moved from round to round. The paper considers four Mobile Byzantine Failure models differing for the diagnosis capabilities at server side i.e., when servers can diagnose their failure state (that is, be aware that the mobile Byzantine agent has left the server), and when servers cannot self-diagnose. We first prove lower bounds on the number of servers n necessary to construct a register tolerant to the presence of f Mobile Byzantine Failures for each of the Mobile Byzantine Failure models considered and then we propose a parametric algorithm working in all the models and matching the lower bounds.

keywords: Self-stabilizing Atomic Storage, Byzantine mobile agents, Round-based Synchronous Computation.

1 Introduction

To ensure high availability, storage services are usually implemented by replicating data at multiple locations and maintaining such data consistent. Thus, replicated servers represent today an attractive target for attackers that may try to compromise replicas correctness for different purposes. Some example are: to gain access to protected data, to interfere with the service provisioning (e.g. by delaying operations or by compromising the integrity of the service), to reduce service availability with the final aim to damage the service provider (reducing its reputation or letting it pay for the violation of service level agreements) etc. In addition, another emerging issue in the design of replicated services is the possibility of having *transient failures* as consequence of possible corruption in replicas state due to errors in the communication or in the computation.

A compromised replica is usually modeled trough an arbitrary failure (i.e. a Byzantine failure) that is made transparent to clients by employing Byzantine Fault Tolerance (BFT) techniques.

As pointed out in [20], in addition to classical Byzantine behaviors, it is worth to consider *mobile adversaries*. Mobile adversaries have been primarily introduced in the context of multi-party computation and they try to model an attacker that is able to progressively compromise computational entities but only for a limited period of time. Therefore, tolerating Mobile Byzantine Failures is, in some sense, like having a bounded number of compromised entities at any given time but such set changes from time to time. Such model captures phenomenon like virus injection (where viruses start to infect the network but then they are detected and progressively deleted from a set of machines), programmed maintenance with the aim of restoring potentially infected machines or self-repairing systems. From a theoretical point of view, mobile adversaries have been formalized in different *Mobile Byzantine Failures* models [10], [7], [17], [3].

In the context of distributed storage implementations (e.g. register abstraction), common approaches to BFT are based on the deployment of a sufficient large number of replicas to tolerate an estimated number f of compromised servers (i.e. BFT replication). However, few efforts have been spent in addressing multi-failures scenarios i.e., how to cope with both Byzantine and transient failures. In addition, to the best of our knowledge, no storage abstraction has been investigated so far assuming mobile adversaries.

Contribution. In this paper, we address the problem of building a self-stabilizing Multi-Writer/Multi-Reader (ss-MW MR) atomic register in the presence of both Byzantine Mobile Failures and transient failures. Concerning Mobile Byzantine Failures, we considered the four theoretical models introduced by Garay, [10], Buhrman *et al.* [7], Sasaki *et al.* [17] and Bonnet *et al.* [3].

The paper provides two main contributions: (i) it proves a set of lower bounds (one for each of the four considered model) on the number of servers n necessary to implement an atomic register in presence of both transient failures and mobile Byzantine failures (i.e., under multi-failure assumption) and (ii) a parametric algorithm implementing a self-stabilizing atomic register in a synchronous round-based message passing system under multi-failure assumption, working in all the four models.

Let us note that the complexity of the proposed algorithm matches the computed lower bounds. As a consequence, the computed bounds are tight in the considered model and the proposed algorithm is optimal. As far as we know, our construction is the first that builds a distributed self-stabilizing MWMR atomic register in the considered environment.

More in details, we proved that the cost, in terms of minimum number of servers n, of implementing a MWMR atomic register under multi-failure assumption (i.e., f mobile Byzantine Failures and an arbitrary number of transient failures) are: $n \geq 3f + 1$ in the Garay's model, $n \geq 4f + 1$ in the Sasaki *et al.*'s model and Bonnet *et al.*'s model and $n \geq 2f + 1$ Buhrman *et al.*'s model.

Roadmap. The paper is organized as follows. Section 2 discusses Related Works and Section 3 presents the system model the problem specification. Section 4 shows lower bounds on the number of server necessary to implement self-stabilizing safe register in the following Mobile Byzantine Failiure models: Garay [10], Buhrman *et al.* [7], Sasaki *et al.* [17] and Bonnet *et al.* [3]. In Section 5 we present a generic tight algorithm that implements self-stabilizing MWMR atomic register parametrized function on the considered mobile Byzantine model. The correctness of the generic algorithm is proved in Section 5.2. Finally, Section 6 concludes the paper and discusses some open research directions.

2 Related Work

Computations under Mobile Byzantine Failure Models. Concerning Mobile Byzantine Failures models, there are two main research directions: (i) Byzantines with constrained mobility and (ii) Byzantines with unconstrained mobility. Byzantines with constraint mobility were studied by Buhrman *et al.* [7]. They consider that Byzantine agents move from one node to another only when protocol messages are sent (similar to how viruses would propagate). In [7], Buhrman *et al.* studied the problem of Mobile Byzantine Agreement.

They proved a tight bound for its solvability (i.e., $n > 3t$, where t is the maximal number of simultaneously faulty processes) and proposed a time optimal protocol that matches this bound.

In the case of unconstrained mobility the motion of Byzantine agents is not tied to message exchange. Several authors investigated the agreement problem in variants of this model: [1, 3, 10, 15, 16, 17]. Reischuk [16] investigate the stability/stationarity of malicious agents for a given period of time. Ostrovsky and Yung [15] introduced the notion of mobile virus and investigate an adversary that can inject and distribute faults. However, none of these works consider the implementation of the shared memories (e.g. register).

Garay [10] and, more recently, Banu *et al.* [1] and Sasaki *et al.* [17] or Bonnet *et al.* [3] consider, in their models, that processes execute synchronous rounds composed of three phases: *send*, *receive*, *compute*. Between two consecutive rounds, Byzantine agents can move from one host to another, hence the set of faulty processes has a bounded size although its membership can change from one round to the next. The main difference between the unconstrained models presented so far is in the knowledge that processes have to have been affected from a Byzantine agent. In the Garay's model a process has the ability to detect its own infection after the Byzantine agent left it. More precisely, during the first round following the leave of the Byzantine agent, a process enters a state, called *cured*, during which it can take preventive actions to avoid sending messages that are based on a corrupted state. Garay [10] proposes in this model an algorithm that solves Mobile Byzantine Agreement provided that $n > 6t$ (dropped later to $n > 4f$ in [1]). Bonnet *et al.* [3] investigated the same problem in a model where processes do not have the ability to detect when Byzantine agents move. However, differently from Sasaki *et al.* [17], cured processes have *control* on the messages they send. This subtle difference on the power of Byzantine agents has an impact on the bounds for solving the agreement. If in the Sasaki's model the bound on solving agreement is $n > 6f$ in Bonnet's model it is $n > 5f$ and this bound is proven tight.

BFT and self-stabilizing registers. Traditional solutions to build a Byzantine tolerant storage service can be divided into two categories: *replicated state machines* [18] and *Byzantine quorum systems* [2, 12, 14, 13]. Both the approaches are based on the idea that the state of the storage is replicated among processes and the main difference is in the number of replicas involved simultaneously in the state maintenance protocol.

Recently, few efforts have been spent in the design of self-stabilizing BFT register implementations [4], [5]. In these works, the authors considered the problem of emulating a register on top of an asynchronous message passing system prone to both Byzantine and transient failures. However, Byzantine failures are assumed to be static.

To the best of our knowledge, this paper is the fist considering both Mobile Byzantine failures and transient failure. The closest work, thus, is represented by the design of a *proactive-reactive recovery* mechanism done by Sousa et al. [19]. The basic idea is to periodically reconfigure the set of replicas to rejuvenate servers that may be under attack (proactive mode) and/or when a failure is detected (reactive mode). The rejuvenation is done according to a prefixed schedule (in TDMA fashion), independently of the effective compromising of a server. This approach seems to be effective in long executions but requires a fine tuning of the parameters (upper bound f on the number of possible compromised replicas in a given period, rejuvenation window, time required by the state transfer, etc...) and the presence of secure components in the system.However, it does not consider transient failures and does not exploit the awareness that servers may have about their cured state to avoid to spread bad information.

3 Model and Problem Definition

3.1 System Model

We consider a distributed system composed of an arbitrary large set of clients $\mathcal{C}$ (including both readers or writers) and of a set of n servers $\mathcal{S} = \{s_1, s_2 \ldots s_n\}$. Each process in the distributed system (i.e., both servers and clients) is identified through a unique integer identifier. Servers run a distributed protocol implementing a shared memory abstraction.

Communication model and timing assumptions. Processes communicate through message passing. It is assumed that processes in the distributed system may access a built-in communication abstraction, denoted *ss-broadcast* (i.e., a self-stabilizing extension of the broadcast primitive), that provides clients with an operation denoted $\mathsf{ss\_broadcast}()$, used to disseminate messages to servers, and each server with a matching operation denoted $\mathsf{ss\_deliver}()$ that delivers the message sent by the client to servers. When the reader or the writer (resp., server) access this broadcast abstraction, we consequently say that it "ss-broadcasts" (resp.,"ss-delivers") a message. This communication abstraction is formally defined and implemented in [4, 9].

More in detail, we assume that (i) each client $c_i \in \mathcal{C}$ can communicate with every server by using the $\mathsf{ss-broadcast}$ primitive (defined below), (ii) servers can communicate among themselves through a $\mathsf{ss-broadcast}$ primitive and (iii) servers can communicate with clients through point-to-point channels. We assume that communications are authenticated (i.e., given a message m, the identity of its sender cannot be forged) and reliable (i.e. messages are not created, lost or duplicated).

The system is synchronous and evolves in sequential synchronous rounds $r_0, r_1, \ldots r_i \ldots$. Every round is divided in three phases: (i) *send* where processes send all the messages for the current round, (ii) *receive* where processes receive all the messages sent at the beginning of the current round[1] and (iii) *computation* where processes process received messages and prepare those that will be sent in the next round.

Failure model. We assume that an arbitrary number of clients may crash while servers are affected by *mobile Byzantine failures* (MBF) [3, 10, 7, 17]. Informally, in the mobile Byzantine failure model, faults are represented by powerful computationally unbounded agents that move arbitrarily from a server to another. When the agent is on the server, it can corrupt its local variables, forces it to send arbitrary messages (potentially different from process to process) etc... However, the agent cannot corrupt the identity of the server. We assume that, in each round r_i, at most f servers can be affected by a mobile Byzantine failure. When an agent occupies a server s_i we will say that s_i is *faulty*. If a server has been occupied by a Byzantine agent in the previous round then the server is said to be *cured*. If a server is neither *faulty* nor *cured* then it is said to be *correct*. We assume, similar to [3, 10, 17], that each server has a tamper-proof memory where it safely stores the correct algorithm code. When the agent leaves a server s_i, the server becomes *cured* and then can recover the correct algorithm code from the tamper-proof memory. Concerning the assumptions on agent movements and the server awareness on its *cured* state, different models have been defined. In this paper we will consider all the variants of mobile Byzantine failures [3, 10, 7, 17]:

- **(M1)** *Garay's model* [10]. In this model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). When a server is in the $cured$ state it is aware of its condition and thus can remain silent for a round to prevent the dissemination of wrong information.

[1]Let us note that, in round-based computations, all $\mathsf{ss\_deliver}()$ events happen during the receive phase.

- **(M2)** *Bonnet et al.'s model* [3] and **(M3)** *Sasaki et al.'s model* [17]. As in the previous model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). Differently from the Garay's model, in both models it is assumed that servers do not know if they are correct or cured when the Byzantine agent moved. The main difference between these two models is that in the [17] model a cured process still acts as a Byzantine one extra round.
- **(M4)** *Buhrman's model* [7]. Differently from the previous models, agents move together with the message (i.e., with the send or broadcast operation). However, when a server is in the *cured* state it is aware of that.

In addition to the possibility of mobile Byzantine failures at server side, processes may also suffer form *transient* failures, i.e., local variables of any process (writer, reader, servers) can be arbitrarily modified [9]. It is nevertheless assumed that transient failures are quiescent i.e., there exists a round r_{no_tr} (which remains always unknown to the processes) after which no more transient failures are going to happen.

3.2 Self-stabilizing Atomic Registers

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely $\mathsf{read}()$ and $\mathsf{write}()$. Informally, the $\mathsf{write}()$ operation updates the value stored in the shared variable while the $\mathsf{read}()$ obtains the value contained in the variable (i.e. the last written value). In distributed settings, every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event.

An operation *op* is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply). Contrary, an operation *op* is said to be *failed* if it is invoked by a process that crashes before the reply event occurs. According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock (unknown to the processes) and the invocation time and response time of every operation are defined with respect to this fictional clock.
Given two operations *op* and *op'*, their invocation event times ($t_B(op)$ and $t_B(op')$) and their reply event times ($t_E(op)$ and $t_E(op')$), we say that *op precedes op'* ($op \prec op'$) iff $t_E(op) < t_B(op')$. If *op* does not precede *op'* and *op'* does not precede *op*, then *op* and *op'* are *concurrent* ($op||op'$). Given a $\mathsf{write}(v)$ operation, the value v is said to be written when the operation is complete.
We assume that locally any client never performs $\mathsf{read}()$ and $\mathsf{write}()$ operation concurrently (i.e., for any given client c_i, the set of operations executed by c_i is totally ordered). We also assume that initially the register stores a default value $\perp$ written by a fictional $\mathsf{write}(\perp)$ operation happening instantaneously at round r_0. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [11]: *safe*, *regular* and *atomic*.

In this paper, we consider a Self-Stabilizing Multi-Writer/ Multi-Reader (MWMR) atomic register i.e., an extension of Lamport's atomic register that considers transitory failures.

The Self-Stabilizing Multi-Writer/Multi-Reader (MWMR) atomic register is specified as follow:

- $\mathsf{ss-Termination}$: Any operation invoked on the register eventually terminates.

- $\mathsf{ss-Validity}$: There exists a round r_{stab} such that each read operation invoked in a round $r > r_{stab}$ returns the last value written before its invocation, or a value written by a $\mathsf{write}()$ operation concurrent with it.

- $\mathsf{ss-Ordering}$: There exists a round r_{stab} and a total order S such that (i) any operation invoked on the register after r_{stab} belongs to S, (ii) given op and op' belonging to S, if $op \prec op'$, then op appears before op' in S and (iii) any $\mathsf{read}()$ operation returns the value v written by the last $\mathsf{write}()$ preceding it in S.

4 Lower Bounds on the number of Replicas for a Self-Stabilizing Register Implementation

In this Section we prove lower bounds on the number of servers n needed to tolerate (i) up to f mobile Byzantine faulty servers and (ii) any number of transient failures, in the implementation of an atomic register. We prove lower bounds for all the four models [3, 10, 7, 17] presented in Section 3.

The structure of the proofs is as follow: we first prove lower bounds for the safe register[2] in absence of transient failures (Theorems 1 - 4). Secondly, we prove that the execution of a $\mathsf{write}()$ operation, when no more transient failures happen, is necessary for the existence of a self-stabilizing safe register algorithm (Theorem 5). Finally, we prove lower bounds in presence of transient failures (Theorem 6 and Corollary 1). Let us note that a safe register is weaker than an atomic one; thus, if no implementation of a safe register exists for a given pair n, f then no atomic implementation can exist.

The proofs argument for lower bounds is focused on the number of values and information gathered by a read operation independently of its length (the number of rounds over it spans). In all the studied models we prove that if there are not enough replicas, readers may be disoriented. Therefore, they cannot return the correct value stored at the server level.

One may naturally think that since a $\mathsf{read}()$ operation does not succeed due to Byzantine agents movements, then it may be helping to iterate the $read\_reply$ round several times during a $\mathsf{read}()$ operation. The idea is the following, since Byzantine agents moves, then once a server is correct, it is aware that the previous reply might have been incorrect and thus it declares it in the current one. Something like "hey, look, during the previous read_reply round I was not correct. Do not believe on what I did declare". In the following this kind of read operation will be called *Multi-attempt read*. We prove that iteration of $\mathsf{read}()$ operations do not add useful knowledge. In particular we show that even if clients can leverage on all possible informations, i.e., values and previous state declarations, it still is not able to read.

Note that in the context of the registers implementation the concept of *Multi-attempt read* is new and it has been introduced in order to cope with the Byzantine agent movements. Also the multi attempt read is different from the notion of multi-phase read operations generally used to implement atomic register. In the case of the multi-phase read the read phase is followed by a write-back phase that helps obsolete servers to update their value.

Firstly we show that a $\mathsf{read}()$ operation can not be implemented in less than two communication steps, $read\_request$ and $read\_reply$. After we formally define the $\mathsf{read}()$ operation and then the multi attempt one, $a-\mathsf{maRead}()$. So finally we prove, for each of the four models considered, the lower bound of the number of servers n necessary to build a register and given the upper bound on the number of mobile Byzantine failures f.

[2]A safe register is a register where $\mathsf{read}()$ operations are guaranteed to return a valid value (in the sense of atomic register validity) only in the absence of concurrency with $\mathsf{write}()$ operations.

Lemma 1 *Any algorithm $\mathcal{A}_s$ implementing a safe register has the* $\mathsf{read}()$ *operation communication pattern compounded by at least two communication steps.*

Proof [Sketch] We consider a non-local $\mathsf{read}()$ operation, thus an operation involving both client and servers. Clients and servers communicate via message passing. This means that one communication step is required to send the information from servers to client. Trivially we avoid to consider implementations in which servers periodically send to clients their stored value, servers should know the whole clients set and there would be too much workload on server side. In such scenario clients just have to wait for the time in which servers send their value. Thus avoiding trivial implementations means that servers have to be aware of the occurrence of a $\mathsf{read}()$ operation in order to send their value. To such aim, the only way to make them aware for clients is to send them a message. Thus at least a two-communication pattern is required. $\Box_{Lemma\ 1}$

Definition 1 (Read operation) *Let us define the* $\mathsf{read}()$ *operation as a two phase operation. In the first phase, read_request req, the read requesting message is sent by client and delivered by servers. In the subsequent phase, read_reply rep, replies are sent by servers and delivered by client which decides the value to return.*

Since the system is round based and synchronous, then any phase lasts exactly one round.

Definition 2 (Multi-attempt read operation) *Let us define as $a-\mathsf{maRead}$() as a sequence $\mathcal{S}$ of elements $s_i \in \{req, rep\}, i \in [2, a-1]$ and $s_0 = req_0$ and $s_a = rep_a$.*

Note that since both req and rep requires one round, then an $a-\mathsf{maRead}()$ requires a rounds to be completed. In the following we use a $a-\mathsf{maRead}()$ in which $s_i = req$ when i is odd and $s_i = rep$ when i is even, with a even. It is easy to see how the used reasoning can be applied also to not interleaved sequence of req and rep, but also for instance to sequences req, rep, rep, rep.

Preliminaries: Let us define as $\text{READREQ}(i)$, $1 \leq i \leq a$, the message sent by client in the request_phase and as $\text{READREPLY}(\langle value, previousRead\_Rep\_State\rangle)$ the reply sent by servers in the reply_phase. $value$ is the stored value at server side and if $i > 1$ then $previousRead\_Rep\_State$ is used to indicate to the client if the servers was $Byzantine$ or $cured$ during the previous read_reply phase during the same $a-\mathsf{maRead}()$.

Theorem 1 *If $n \leq 3f$, there exists no algorithm that implements a MWMR Safe Register in Garay's model [10].*

Proof Let us consider a $1-\mathsf{maRead}()$ operation beginning at round r. Let us consider, at such time, the following server set states: S_{1corr}, S_{2cur} and S_{3Byz} in which correct servers are storing value v. At the beginning of the round client performs $\mathsf{read}_1()$ operation sending an READREQ message which is delivered by servers within the same round. Thus S_1 and S_2 deliver it.
At the begin of round $r+1$, during the read_reply phase, Byzantine agents move and we may have the following server set states: S_{1Byz}, S_{2corr} and S_{3cur}. In Fig. 1 is depicted how server set states change during the two phases of a single $\mathsf{read})()$.
Such servers reply as follow:

- S_{1Byz}: $\langle v'\rangle$
- S_{2cor}: $\langle v\rangle$

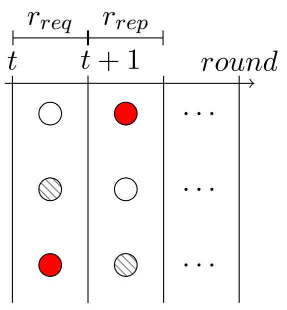


Figure 1: A generic run in which a 1-$maRead()$ is performed over two rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and Byzantine ones in red.

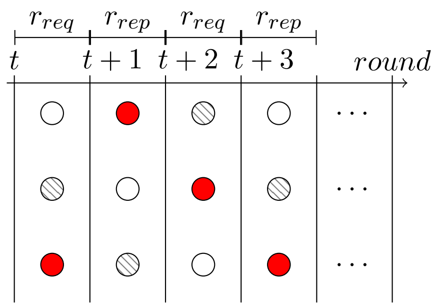


Figure 2: A generic run in which a 2-$maRead()$ is performed over four rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and finally Byzantine ones in red.

Given these information there is no way to decide a value nor to detect where Byzantine agents are placed.

Let us now consider a $2 - \mathsf{maRead}()$ operation beginning at round r. For round r and $r+1$ let us consider the same server set states and replies as in the previous $1 - \mathsf{maRead}()$ operation. Summing up, after $\mathsf{read}_1()$ operation client has the following replies:

- S_{1Byz}: $\langle v' \rangle$
- S_{2cor}: $\langle v \rangle$

At round $r+2$ Byzantine agents move and another read() operation begins, $\mathsf{read}_2()$. Let us consider the following server set states: S_{1cur}, S_{2Byz} and S_{3corr}. Thus S_1 and S_3 deliver the READREQ message. At the beginning of round $r+3$, during the read_reply phase, Byzantine agents move and we may have the following server set states: S_{1corr}, S_{2cur} and S_{3Byz}. In Fig. 2 is depicted how server set states change during the four phases of a 2-maRead().

Such servers reply as follow:

- S_{1cor}: $\langle v, Byz \rangle$
- S_{3Byz}: $\langle v', Byz \rangle$

Considering values coming from the two read() operations, a client has the following replies:

- S_{1Byz}: $\langle v' \rangle$

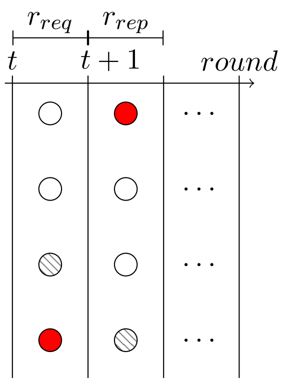


Figure 3: A generic run in which a 1-$maRead()$ is performed over two rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and finally Byzantine ones in red.

- S_{1cor}: $\langle v \rangle$
- S_{2cor}: $\langle v, Byz \rangle$
- S_{3Byz}: $\langle v', Byz \rangle$

In such scenario, server set S_1 replied differently during the two read operations, thus, since there are no concurrent write() operation, in one of the two it has been affected by Byzantine agents. But nor the information gathered from S_2 neither the ones gathered from S_3 helps the client to understand which are the correct values or where Byzantine agents where placed. That is, the information about the previous state gathered during the read$_2$() does not help to distinguish the correct value coming from read$_1$(). More general since Byzantine servers may declare to have been affected during the previous reply phase, the information gathered during a generic read$_k$() do not improve the knowledge gathered during the previous read$_{k-1}$(). $\Box_{Theorem\ 1}$

Theorem 2 *If $n \leq 4f$, there exists no algorithm that implements a MWMR Safe Register in Sasaki's model [17].*

Proof Let us consider an $1 -$ maRead() operation beginning at round r. Let us consider, at t, the following server set states: S_{1corr}, S_{2corr}, S_{3cur} and S_{4Byz} in which correct servers are storing value v. At the beginning of the round client performs read$_1$() operation sending an READREQ message which is delivered by servers within the same round. Thus S_1, S_2 and S_3 deliver it.
At the beginning of round $r+1$ Byzantine agents move and we may have the following server set states: S_{1Byz}, S_{2corr}, S_{3corr} and S_{4cur}. In Fig. 3 is depicted how server set states change during the four phases of a 1-maRead().

Such servers may reply as follow:

- S_{1Byz}: $\langle value' \rangle$
- S_{2cor}: $\langle value \rangle$
- S_{3cor}: $\langle value \rangle$
- S_{4cur}: $\langle value \rangle$

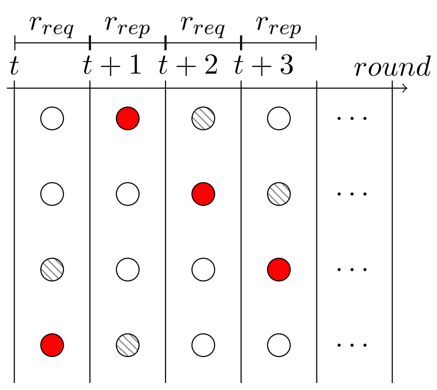


Figure 4: A generic run in which a 2-$maRead()$ is performed over four rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and finally Byzantine ones in red.

Given these information there is no way to decide a value nor to detect where Byzantine agents are placed.

Let us now consider a $2 - \mathsf{maRead}()$ operation beginning at round r. For round r and $r+1$ let us consider the same server set states and replies as in the $1 - \mathsf{maRead}()$ operation case. Thus after $\mathsf{read}_1()$ operation client has the following replies:

- S_{1Byz}: $\langle value' \rangle$
- S_{2corr}: $\langle value \rangle$
- S_{3corr}: $\langle value \rangle$
- S_{4cur}: $\langle value \rangle$

At round $r+2$ Byzantine agents move and another $\mathsf{read}()$ operation begins, $\mathsf{read}_2()$. Let us consider the following server set states: S_{1cur}, S_{2Byz}, S_{3corr} and S_{4corr}. Thus S_1, S_3 and S_4 deliver the READREQ message. At the beginning of round $r+3$ Byzantine agents move and we may have the following server set states: S_{1corr}, S_{2cur}, S_{3Byz} and S_{4corr}. In Fig. 4 is depicted how server set states change during the four phases of a 2-$\mathsf{maRead}()$.

Such servers may reply as follow:

- S_{1corr}: $\langle value, Byz \rangle$
- S_{2cur}: $\langle value', Byz \rangle$
- S_{3Byz}: $\langle value', cur \rangle$
- S_{4corr}: $\langle value, cur \rangle$

Considering values coming from the two $\mathsf{read}()$ operations, a client has the following replies:

- S_{1Byz}: $\langle value' \rangle$
- S_{1corr}: $\langle value, Byz \rangle$
- S_{2corr}: $\langle value \rangle$

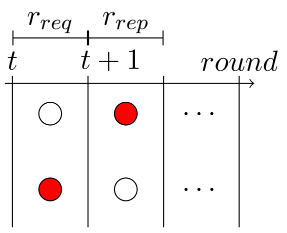


Figure 5: A generic run in which a 1-$maRead()$ is performed over two rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and finally Byzantine ones in red.

- S_{2cur}: $\langle value', Byz\rangle$
- S_{3corr}: $\langle value\rangle$
- S_{3Byz}: $\langle value', cur\rangle$
- S_{4cur}: $\langle value\rangle$
- S_{4corr}: $\langle value, cur\rangle$

In such scenario, all server sets replied differently during the two read operations, thus, since there are no concurrent write() operation, in one of the two, all of them have been affected or just left by Byzantine agents. It is clear that the information about the previous state gathered during the $\mathsf{read}_2()$ do not help to distinguish the correct value coming from $\mathsf{read}_1()$. Since Byzantine server may declare as well to have been affected during the previous reply phase, the information gathered during a generic $\mathsf{read}_\mathsf{k}()$ do not improve the knowledge gathered during the previous $\mathsf{read}_{\mathsf{k}-1}()$. $\square_{Theorem\ 2}$

Theorem 3 *If $n \leq 4f$, there exists no algorithm that implements a MWMR Safe Register in Bonnet's model [3].*

Proof The claim simply follows by considering that the Bonnet's model is a particular case of Sasaki model, in which cured servers act as less powerful faulty servers, forced to send the same message to all. The same reasoning as in the proof of Theorem 2 is applied. $\square_{Theorem\ 3}$

Theorem 4 *If $n \leq 2f$ there exists no algorithm that implements a MWMR safe Register in Burhman's model [7].*

Proof Let us consider an $1-\mathsf{maRead}()$ operation beginning at round r. Let us consider, at such time, the following server set states: S_{1corr} and S_{2Byz} in which correct servers are storing value v. At the beginning of the round client performs $\mathsf{read}_1()$ operation sending an READREQ message which is delivered by servers within the same round. Thus S_1 delivers it.
At the beginning of round $r+1$ Byzantine agents move and we may have the following server set states: S_{1Byz} and S_{2corr}. In Fig. 5 is depicted how server set states change during the four phases of a 1-$\mathsf{maRead}()$.
Such servers may reply as follow:

- S_{1Byz}: $\langle value'\rangle$

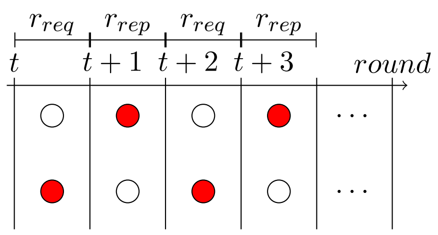


Figure 6: A generic run in which a 2-$maRead()$ is performed over four rounds. Each circle represents a set of f servers, correct ones in white, cured ones in shadow and finally Byzantine ones in red.

- S_{2cor}: $\langle value \rangle$

Given these information there is no way to decide a value nor to detect where Byzantine agents are placed.

Let us now consider a $2 -$ maRead() operation beginning at round r. For round r and $r+1$ let us consider the same server set states and replies as in the $1 -$ maRead() operation case. Thus after read$_1$() operation client has the following replies:

- S_{1Byz}: $\langle value' \rangle$
- S_{2cor}: $\langle value \rangle$

At round $r+2$ Byzantine agents move and another read() operation begins, read$_2$(). Let us consider the following server set states: S_{1corr} and S_{2Byz} (the same as in round r). Thus S_1 delivers the READREQ message. At the beginning of round $r+3$ Byzantine agents move and we may have the following server set states: S_{1Byz} and S_{2corr} (the same as in round $r+1$). In Fig. 6 is depicted how server set states change during the four phases of a 2-maRead().

Such servers may reply as follow:

- S_{1Byz}: $\langle value' \rangle$
- S_{2cor}: $\langle value \rangle$

In such case Byzantine servers do not declare Byzantine during the previous read$_1$(), since they indeed were Byzantine.

Considering values coming from the two read() operation, client has the following replies:

- S_{1Byz}: $\langle value' \rangle$
- S_{2cor}: $\langle value \rangle$
- S_{1Byz}: $\langle value' \rangle$
- S_{2cor}: $\langle value \rangle$

In such scenario, all server sets replied the same during the two read operations. Thus in such situation the client has still no way to decide which is the correct value. It follows that since Byzantine servers may declare as well to have been affected during the previous reply phase or not, depending on the situation, the information gathered during a generic $\mathsf{read_k}()$ do not improve the knowledge acquired during the previous $\mathsf{read_{k-1}}()$. $\Box_{Theorem\ 4}$

Intuitively, to prove the previous Theorems we show that even using all available information provided by servers (e.g., values and information about the failure state in previous rounds), clients are not able to select a correct value to return.

Theorem 5 *Let r_{no_tr} the round after which no more transitory failures are going to happen. If no* $\mathsf{write}()$ *operation is invoked at some round $r > r_{no\_tr} + 1$ then there is no self-stabilizing algorithm implementing a self-stabilizing safe register with any number of replicas.*

Proof Consider a system with n servers and assume that there exists a protocol $\mathcal{A}_{SR}$ implementing a self-stabilizing safe register.

Consider a $\mathsf{write}(v)$ operation op_w. Note that, since $\mathcal{A}_{SR}$ is correct, op_w eventually terminates and the we can consider the time interval $[t_B(op_w), t_E(op_w)]$ in which it is executed. Since, by assumption, there not exists any $\mathsf{write}()$ operation happening after $r_{no\_tr} + 1$, it means that $t_E(op_w) < r_{no\_tr} + 1$[3]. As a consequence, transient failures still happen until round r_{no_tr} and thus the state of servers can be changed arbitrarily. In addition, server may still receive corrupted messages (i.e., messages introduced in the channels in the transitory phase) and take arbitrary decisions also in round $r_{no\_tr} + 1$. Without loss of generality, let us assume that op_w is the last $\mathsf{write}()$ operation executed before $r_{no\_tr} + 1$.

Let us now consider a $\mathsf{read}()$ operation op_r starting at time $t_B(op_r)$, where $t_B(op_r) > r_{no\_tr} + 1$. Such $\mathsf{read}()$ operation can only reasons on the set of values $\{v1_1, v1_2, \dots\ v1_{\frac{n}{2}+1}, v_{\frac{n}{2}+2}, \dots, v_n\}$ provided by servers and either returns $v1 \neq v$ or not deciding for a value and keep waiting. This contradicts the assumption that $\mathcal{A}_{SR}$ implements a safe register as, in absence of concurrency, op_r should return the value v written by op_w.

$\Box_{Theorem\ 5}$

In the following we prove that the bounds previously computed for environments free of transient faults still hold for the case of environments prone to transient faults even when a $\mathsf{write}()$ operation is correctly executed after the transitory period.

Theorem 6 *Let r_{no_tr} the round after which no more transient failures happen and let op_w the first* $\mathsf{write}()$ *operation invoked after $r_{no\_tr} + 1$. If $n \leq \alpha f$ (with α selected according with Table 1), then there is no self-stabilizing protocol implementing a MWMR Safe Register in any considered model.*

Proof The proof simply follows by considering that op_w happens after the transitory phase ends. Thus the state we have at the end of the operation is equivalent to the state we have at r_0 in a system that is not prone to transitory failures. As a consequence, the scenarios happening in Theorems 1 - 4 hold and the claim follows.

$\Box_{Theorem\ 6}$

From Theorem 6 the next Corollary directly follows

Corollary 1 *Let r_{no_tr} the round after which no more transient failures happen and let op_w the first* $\mathsf{write}()$ *operation invoked after $r_{no\_tr} + 1$. If $n \leq \alpha f$ (with α selected according with Table 1), then there is no self-stabilizing protocol implementing a MWMR Atomic Register in any considered model.*

[3]With a slight abuse of notation, $t_E(op_w) < r_{no\_tr} + 1$ means that op_w terminates before beginning of round $r_{no\_tr} + 1$.

5 Tight Self- Stabilizing MWMR Atomic Register Implementation

In this section, we present an algorithm $\mathcal{A}_{SSAreg}$ implementing a Self-Stabilizing MWMR Atomic Register resilient to the presence of up to f Byzantine agents affecting servers and moving at each round.

The algorithm follows the basic quorum-based approach to implement read() and write() operations.

Let us recall that mobile Byzantine agents move from one server to another corrupting their internal states. As a consequence, if not properly mastered, this can bring to the compromising of all the servers and to the loss of the register value (even in the absence of transitory failures). A naive solution would be to exploit write() operations to clean values of cured processes and increase the number of replicas n to ensure the presence of "enough" correct servers to select a valid value. However, such solution has two strong drawbacks: (i) write() operations are not governed by servers and are invoked depending on clients protocols and (ii) the number of replicas needed to tolerate f mobile Byzantine agents will grow immediately linearly in the number of rounds between two following write() operations.
To handle the presence of mobile Byzantine agents, we started form this intuition and we defined a value propagation mechanism that is used to help cured servers to recover and to update their local variables to a correct state. Such mechanism is executed at the beginning of each round and it pushes information between servers allowing cured ones to become correct in one round. The immediate benefit is the reduction of the number of replicas required to master the mobility.

The second issue we faced in our implementation is the presence of transient failures that may alter unexpectedly processes state and channels state. To master corruptions and limit the impact of transient failures, we defined an algorithm that uses a number of local variables as small as possible. In addition, we exploited the synchrony of the system to ensure that our algorithm stabilizes quickly as soon as a write() happens (necessary condition for the stabilization) guaranteeing both safety and liveness properties. Let us note that the tricky part is guarantee that read() operations eventually terminate and eventually return a valid value. To this end, we introduced a control variable $op_R\_start_i$ that is used to identify in which step of the operation is the client. Such variable is used in the algorithm to determine termination conditions and avoid that a client remain blocked forever after the end of transient failures.

The algorithm presented in the following is defined in a parametric way in order to fit all the four mobile Byzantine failure models presented in Section 3. The first parameter of the algorithm, denoted as α, is used to relate the global number of servers required n to the number of mobile Byzantine agents f that we want to tolerate. In particular, we will relate such two values by the following inequality $n > \alpha f$ with $\alpha \in \{2, 3, 4\}$ depending on the mobile Byzantine failure model considered.
The second parameter, denoted as β, is used to define the minimal number of occurrences of a same value that a client needs to see in order to terminate a read() operation and select a valid value. Such value is denoted by s and it is defined as $s = n - \beta f$, with $\beta \in \{1, 2\}$.

Finally, in order to abstract the knowledge that a server has of its failure state (i.e. *cured* or *correct*), we introduce the $cured\_state$ oracle. When invoked via report_cured_state() function, it returns, in the Garay [10] and Buhrman *et al.* [7] models, true to *cured* servers and false to others. In this case the oracle is said to be enabled. In Sasaki *et al.* [17] and Bonnet *et al.* [3] model the $cured\_state$ oracle returns always false. In this case the oracle is said disabled. The implementation of the oracle is out of scope of this paper and the reader may refer to [8], [15] for further details.

Table 1 summarizes the above parameters for each model.

Table 1: $\mathcal{A}_{SSAreg}$ parameters for the four different Mobile Byzantine Failure models.

Failure model	Mid	α	β	Oracle
Garay [10]	M1	3	2	enabled
Bonnet *et al.* [3]	M2	4	2	disabled
Sasaki *et al.* [17]	M3	4	2	disabled
Burhman *et al.* [7]	M4	2	1	enabled

5.1 $\mathcal{A}_{SSAreg}$ Algorithm Detailed Description

The pseudo-code of the algorithm is presented in Figures 7-9. The algorithm exploits the round based nature of the system.
Any write() operation spans at most two rounds. The operation may, in fact, be invoked in the middle of a round and in this case it effectively starts in the send phase of the next round. The writer ss-broadcast the value and all servers ss-deliver it in the same round. In the receive phase of the same round, servers delivers WRITE() messages and, if more than one write() operation is executed in the same round, servers will update the register by selecting the value coming from the client with the highest identifier. Due to the synchrony assumptions no acknowledgement message is required and the operation can terminate at the end of the round.
The read() operation spans at most three rounds. As for the write(), it effectively starts in the send phase of the first round starting after its invocation and takes such round to send a read request to servers and the following one to gather replies. In the computation phase of the second round, the reader selects the value occurring at least $s = n - \beta f$ times.

The value propagation mechanism is implemented by letting servers disseminate the stored value through ECHO() messages at the beginning of each round. Such ECHO() messages are collected in the receive phase and are used by cured processes to select a value and to update their value of the register. In such way, they are able to cope with f servers that may have lost their value due to the Byzantine mobility.

Local variables at client c_i. Each client c_i needs to manage the following variables for the implementation of the read() operation:
$-$ $op_R\_start_i$: is a variable used to keep track of the state of a read() operation at client c_i and it can have the following values: $\{0 = \text{request\_round}, 1 = \text{reply\_round}, \perp = \text{no\_read\_running}\}$.
$-$ $replies_i$: is a set used to collect REPLY messages for a read() operation. It is set to $\emptyset$ at the beginning of a read() operation.

Local variables at server s_i. Each server s_j manages the following variables:
$-$ $value_i$: it stores the current value of the register.
$-$ $echo\_vals_i$: is a set variable (emptied at the beginning of each round) where servers store values collected trough ECHO messages in the current round.
$-$ $current\_writes_i$: is a set variable (emptied at the beginning of each round) where servers store values sent trough a WRITE() message.
$-$ $current\_reads_i$: is a set variable where servers store identifiers of clients that are currently reading. It is emptied after the reply to such clients.
$-$ $cured_i$: is a boolean variable set through the report_cured_ state() event. It is set to true by the cured_state oracle (if enabled) when s_i is in a $cured$ state. Otherwise it is always false.

```
At the beginning of each round $r$
(01) $echo\_vals_i \leftarrow \emptyset$;
(02) $current\_writes_i \leftarrow \emptyset$;
(03) $cured_i \leftarrow$ report_cured_state();
------------------------------------------------------------
Send Phase of round $r$
(04) if $(\neg cured_i)$
(05)     then ss − broadcast ECHO$(val, i)$;
(06)         for each $j \in current\_reads_i$ do
(07)             send REPLY$(value_i, i)$ to $c_j$;
(08)         endFor
(09) endif
(10) $current\_reads_i \leftarrow \emptyset$;
------------------------------------------------------------
Receive Phase of round $r$
(11) for each ECHO$(v, j)$ message ss-delivered do
(12)     $echo\_vals_i \leftarrow echo\_vals_i \cup \{v\}$;
(13) endFor
(14) for each WRITE$(v, j)$ message ss-delivered do
(15)     $current\_writes_i \leftarrow current\_writes_i \cup \{< v, j >\}$;
(16) endFor
(17) for each READ$(j)$ message ss-delivered do
(18)     $current\_reads_i \leftarrow current\_reads_i \cup \{j\}$;
(19) endFor
------------------------------------------------------------
Computation Phase of round $r$
(20) if $(current\_writes_i \neq \emptyset)$
(21)     then let $v$ such that $\exists < v, j > \in current\_writes_i$
(22)             $\wedge j = \max_k(< -, k >)$;
(23)         $value_i \leftarrow v$;
(24)     else if $(\exists v \in echo\_vals_i \mid \#\text{occurrence}(v) \geq n - \beta f)$
(25)         then $value_i \leftarrow v$;
(26)         else $value_i \leftarrow \perp$;
(27)     endif
(28) endif
```

Figure 7: $\mathcal{A}_{SSAreg}$ implementation: code executed by any server s_i.

Server maintenance. In the *send* phase of each round, servers, whose $cured\_state$ oracle returns false, ss − broadcast a ECHO(val, i) message (line 05, Figure 7). If no write() operations happen in the current round (the condition at line 20 is not verified), such collected values are then used during the *computation* phase (line 24, Figure 7) by cured servers to select a value occurring at least $n - \beta f$ occurrences and update their state. Note that during the transitory period, it could happen that there are not enough occurrences of the same value and then any servers will set its $value_j$ to $\perp$.

Write operation. In order to write a value v a client c_i has to ss-broadcast the WRITE(v, i) message to all servers (line 01, Figure 9). Since an operation invocation may happen in any time during a round, then the ss − broadcast() is delayed until the next send phase. At the server side this message is ss-delivered within the same round during the *receive* phase and any $correct$ and $cured$ server s_j stores it in $current\_writes_j$ set (lines 14-15, Figure 7). At the end of the round, during the *computation* phase, if $current\_writes_j$ is not empty then the value associated to the highest client identifier is stored in $value_j$ (lines 20-23, Figure 7). Back to the client side, during its *computation* phase, it returns the write_conformation to the application layer (line 03, Figure 9).

```
operation read():
(01) delay $op_R\_start_i \leftarrow 0$ until the end of the round;
------------------------------------------------------------
Send Phase of round $r$
(02) if ($op_R\_start_i == 0$)
(03)     ss − broadcast READ($i$);
(04) endIf
(05) $replies_i \leftarrow \emptyset$;
------------------------------------------------------------
Receive Phase of round $r$
(06) for each REPLY($v_j, j$) message received from $s_j$ do
(07)     $replies_i \leftarrow replies_i \cup \{< v_j, j >\}$;
(08) endFor
------------------------------------------------------------
Computation Phase of round $r$
(09) if ($op_R\_start_i = 1$)
(10)   then $op_R\_start_i \leftarrow \perp$;
(11)         if ($\exists < v_j, - > \in replies_i \mid$ #occurrence($v_j$) $\geq n - \beta f$)
(12)           then $v \leftarrow v_j$;
(13)           else $v \leftarrow \perp$;
(14)         endif
(15)         return $v$;
(16)   else if ($op_R\_start_i = 0$)
(17)         then $op_R\_start_i \leftarrow 1$;
(18)         else $op_R\_start_i \leftarrow \perp$;
(19)              return $\perp$;
(20)        endif
(21) endif
```

Figure 8: $\mathcal{A}_{SSAreg}$ implementation: code executed by any client c_i for the read() operation.

Read operation. When a read() operation is invoked by a client c_i, $op_R\_start_i$ is set to 0 at the end of the current round (line 01, Figure 8), thus at the next *send* phase the condition at line 02 is true and the READ(i) message is ss-broadcasted (line 03). Regardless the value of $op_R\_start_i$ at each round the $replies_i$ set is emptied (line 05). In the *computation* phase, the condition at line 16 is true ($op_R\_start_i$ is equal to 0) and $op_R\_start_i$ is set to 1. This means that the read_request phase is over and the next one is the read_reply one. At server side (Figure 7), the READ(i) message is ss-delivered within the same invocation round. Once the message is ss-delivered, any server s_j stores the identifier of the reader in the $current\_reads_j$ set in order to send back a REPLY() message at the beginning of the next round (lines 17-18, Figure 7).
At client side (Figure 8), when the next round begins, the condition at line 02 is not true, thus during the *send* phase the $replies_i$ set is emptied. Such set is filled with REPLY($value_j$) messages during the *receive* phase (lines 06 - 08, Figure 8). During the *computation* phase the condition at line 09 is true, thus $op_R\_start_i$ is set to $\perp$ and the value in $replies_i$ which occurs at least $n - \beta f$ times is returned to the application layer (lines 08-15, Figure 8).

5.2 Correctness Proofs

Lemma 2 *Any* write() *operation eventually terminates.*

Proof The proof trivially follows by considering that the writer generates a write_confirmation event at the end of the computation phase in which the operation is effectively started (line 03, Figure 9).

$\Box_{Lemma\ 2}$

```
operation write($v$)
(01) delay ss − broadcast WRITE($v, i$) until next send phase;
----------------------------------------------------------------
Send Phase of round $r$
----------------------------------------------------------------
Receive Phase of round $r$
(02) nop
----------------------------------------------------------------
Computation Phase of round $r$
(03) return write_confirmation;
```

Figure 9: $\mathcal{A}_{SSAreg}$ implementation: code executed by any client c_i for the write() operation.

Lemma 3 *Any* read() *operation eventually terminates.*

Proof Let r be the round in which a read() operation op_r is invoked and let r_{no_tr} the round in which transient failures stop to happen. Let us note that when a reader invokes a read() operation, it executes line 01 in Figure 8 by setting $op_R\_start_i = 0$ just before entering in a send phase and sending the read request, let's say at round $r + 1$. Two cases may happen: (1) $r > r_{no\_tr}$ and (2) $r < r_{no\_tr}$.

- **Case (1):** $r > r_{no\_tr}$**:** If no transient failures occur at the reader client c_i then $op_R\_start_i$ is set to 1 in the computation phase of $r + 1$ (line 17, Figure 8) Then, in the computation phase of round $r + 2$, c_i will execute lines 09-15, Figure 8 by returning from the operation and the claim follows in this case.
- **Case (2):** $r \leq r_{no\_tr}$**:** in this case, $op_R\_start_i$ may be altered by transitory failures. Let us call Σ the set of all possible values that the variable $op_R\_start_i$ may take and let us consider what happen during the computation phase for each value $v \in \Sigma$.
  - **Case (2.1)** $op_R\_start_i = 0$**:** in this case, the transient failure brings the execution in a situation equivalent to Case (1) causing a "logical restart" of the operation. However, the round r_{no_tr} exists and is finite. Thus, after a finite number of iterations, we fall down in Case (1) and the claim follows.
  - **Case (2.2)** $op_R\_start_i = 1$**:** in this case the claim follows as c_i will execute lines 09-15, Figure 8 by returning from the operation.
  - **Case (2.3)** $op_R\_start_i = v$ **with** $v \in \Sigma \setminus \{0, 1\}$**:** also in this case the claim follows as c_i will execute lines 18-19, Figure 8 by returning from the operation.

$\Box_{Lemma\ 3}$

Theorem 7 (ss-Termination) *Any operation invoked on the register eventually terminates.*

Proof The proof directly follows from Lemma 2 and Lemma 3. $\Box_{Theorem\ 7}$

Lemma 4 *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 1 and used by the algorithm in Figures 7-9. Let $n > \alpha_{Mi} f$ for each failure model Mi considered. If there are no transient failures, then at the end of each round at least $n - f$ correct servers store the same value v in their $value_i$ local variable.*

Proof The proof is done by induction.

- **Basic Step.** At the end of each round, each non-faulty server updates its $value_i$ local variable (i) in line 23 (i.e., if there exists at least a pair in the $current\_writes_i$ local variable) or (ii) in line 25 (i.e., $current\_writes_i$ is empty and there exist at least $n - \beta f$ same values in $echo\_vals_i$).

  Let us recall that at round r_0 all correct servers store the same default value $\perp$ in their local variable $value_i$. As a consequence, in r_0 there exists at leas $n - 2f$ correct servers storing v.

  Let us first prove that one of the two cases always happens and then we prove that the number of non-faulty servers storing the same values v at the end of r_0 is $n - f$.

  The $current\_writes_i$ local variable is initialized by any non-faulty server s_i to $\emptyset$ at the beginning of each round r (cfr. line 02) and it is updated when a WRITE() message is received by s_i[4]. Thus, case (i) corresponds to a scenario where at least a write() operation is executed in round r_0 and case (ii) corresponds to a scenario where no write() is running.

  - **Case (i): $\mathbf{current\_writes_i} \neq \emptyset$.** In this case the claim simply follows by observing that the $current\ \_writes_i$ local variable is filled in when servers deliver a WRITE() message. Considering that (i) writer clients broadcast a WRITE(v, j) message in the send phase of round r, (ii) clients are correct and send the same set of values to all servers that will apply a deterministic function to select the value v and (iii) at most f servers are faulty and may skip the update of their $value_i$ variable, the claim follows.
  - **Case (ii): $\mathbf{current\_writes_i} = \emptyset$ and line 24 is true.** In this case, the $value_i$ variable is updated according to the values stored in $echo\_vals_i$. Such variable is emptied by every non-faulty process at the beginning of each round (cfr. line 01) and is filled in when an ECHO() message is delivered. Such message is sent at least by any server, believing it is correct, at the beginning of each round.

    At the beginning of r_0, at least $n - f - x$ correct servers will send an ECHO(v, j) message, where x is the number of non-faulty processes that become faulty in r_0 (i.e. $x = f$ for all the models but Burhman's one where $x = 0$ as faulty processes move during the send phase and not at the beginning of the round). Let us note that the condition in line 24 is verified if and only if $n - 2f$[5] $\geq n - \beta f$ that is true in any model. Therefore, considering that at the end of round r_0 non-faulty servers are exactly $n - f$, we have that $n - f$ processes will execute this update.

- **Inductive Step.** Iterating the reasoning for any r the claim follows.

$\Box_{Lemma\ 4}$

Theorem 8 (ss-Validity) *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 1 and used by the algorithm in Figures 7-9. Let $n > \alpha_{Mi} f$ for each failure model, Mi, considered. If there exists a* write() *operation op_w issued at some round $r > r_{no\_tr} + 1$ then there exists a round $r_{stab} \geq r$ such that each* read() *operation invoked in a round $r' > r_{stab}$ returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

[4] Recall that such WRITE() message is sent by the writer client in the send phase of the first round starting after the write() invocation and it is delivered by any non-faulty server in the same round.

[5] $n - 2f$ is the number of correct servers sending the ECHO() message in r_0.

Proof Let us recall that, due to Theorem 5, op_w do exists. Let r_{w1} be the round in which op_w terminates and let v_0 be the value written by op_w. Clearly, $r_{w1} \geq r_{no\_tr} + 1$. Without lost of generality, let us consider the first $\mathsf{write}(v)$ operation op'_w and the first $\mathsf{read}()$ operation op_r issued after r_{w1}. Three cases may happen: (i) $op_r \prec op'_w$, (ii) $op'_w \prec op_r$ and (iii) $op'_w \mid\mid op_r$. Let us note that op_r spans over at least two rounds and during the first one the client sends the READ() message while in the second one it collects replies.

- **Case (i): $\mathbf{op_r \prec op'_w}$.** This case follows directly from Lemma 4 considering that (i) at the end of the first round of op_r at least $n - f$ correct processes have the same value v_0 written by op_w, (ii) while moving to the second round of op_r, at most x processes can get faulty (with $x \leq f$ for models M1-M3 and $x = 0$ for M4), (iii) $n - f - x \geq n - \beta_{Mi} f$ (i.e. $\beta_{Mi} f \geq f + x$) for each model (i.e. there will always be enough replies from correct servers to select a value) and (iv) $n - \beta_{Mi} f > f$ (i.e. $(\alpha_{Mi} - \beta_{Mi}) f + 1 > f$) for each model. It follows that faulty processes cannot force the client to select a wrong value and the claim follow in this case.
- **Case (ii): $\mathbf{op'_w \prec op_r}$.** Let $r_{w'}$ be the round at which op'_w terminates and let $r_{w'} + 1$ be the round at which op_r is invoked. Due to Lemma 4, at round $r_w + 2$ there are at least $n - \beta f$ of the last written value. So, applying the same reasoning of case (i) the claim follows.
- **Case (iii): $\mathbf{op'_w \mid\mid op_r}$.** Let us note that a $\mathsf{read}()$ operation spans two rounds, i.e., the round of the request r_{req} and the round of the reply r_{reply}. So, let us consider them separately.
  - **Case (iii.a):** op'_w is concurrent with op_r during r_{req}. In that case the value v is delivered to correct server at the end of r_{req}. Due to Lemma 4, at the end of r_{req} at least $n - f$ correct servers store the new written value v, we fall down into case (ii) and the claim follows.
  - **Case (iii.b):** op'_w is concurrent with op_r during r_{replay}. Since, in every round, the send phase is executed before the receive phase, it follows that at least all the correct servers will reply with the value written before the invocation of the $\mathsf{write}()$ operation, we fall down into case (i) and the claim follows.

$\square_{Theorem\ 8}$

Theorem 9 (ss-Ordering) *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 1 and used by the algorithm in Figures 7-9. Let $n > \alpha_{Mi} f$ for each failure model Mi considered. If there exists a* $\mathsf{write}()$ *operation op_w issued at some round $r > r_{no\_tr} + 1$ then there exists a round r_{stab} and a total order S such that (i) any operation invoked on the register after r_{stab} belongs to S, (ii) given op and op' belonging to S, if $op \prec op'$, then op appears before op' in S and (iii) any* $\mathsf{read}()$ *operation returns the value v written by the last* $\mathsf{write}()$ *preceding it in S.*

Proof Let us recall that, due to Theorem 5, op_w do exists. Let r_{w1} be the round in which op_w terminates and let v_0 be the value written by op_w. Clearly, $r_{w1} \geq r_{no\_tr} + 1$.

In order to prove the claim, we have to show that the algorithm in Figures 7-9 is eventually able to build a total order of operations S that preserves (i) the *read from last write* property and that includes all the operations from a certain round on.

Let us observe the following:

1. any $\mathsf{write}()$ operation is "effectively" executed in one round (i.e., the round in which the value is propagated) even if it has been invoked during the previous round;

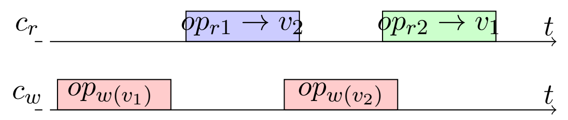
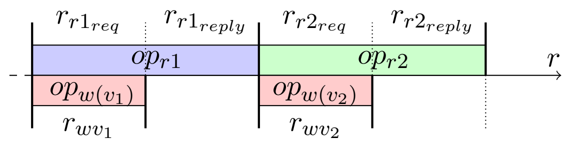
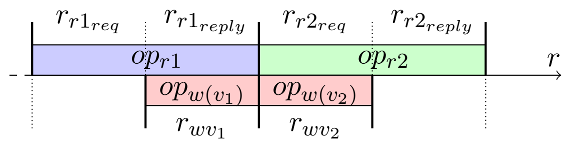


Figure 10: An example of new/old inversion.



(a)
b



(b)
b

Figure 11: Examples of runs showing in details how operations are aligned given the round-based nature of the system.

2. any $\mathsf{read}()$ operation is "effectively" executed in two rounds (i.e., r_{req} the round in which the request for reading the value is sent to servers and r_{rep} where replies are collected at the client side) even if it has been invoked during the previous round;

3. at the beginning of any round $r > r_{w1}$, since no more transient failures are going to happen, there always exist at least $n - 2f$ correct servers storing the same value v (see Lemma 4);

4. correct servers answer to read request by sending back their local values.

Let us suppose by contradiction that a total order S does not exists. S cannot exist iff the scenario in Figure 10 happens. However, considering the observations above and that the algorithm evolves in synchronous rounds, all the possible executions follow patterns similar to those shown in Figure 11, i.e., there can not exist a $\mathsf{write}()$ operation that overlaps two different $\mathsf{read}()$ operations op_{r1} and op_{r2} such that $op_{r1} \prec op_{r12}$, from which we have a contradiction.

$\Box_{Theorem\ 9}$

Let us remark that from proofs of Theorem 8 and Theorem 9 we have that a $\mathsf{write}()$ operation after round $r_{no\_tr} + 1$ is enough to ensure the correctness of our protocols and the stability is reached. From the previous observation the following Corollario follows:

Corollary 2 *If there exists a* $\mathsf{write}()$ *operation op_w issued at some round $r > r_{no\_tr} + 1$ then $r_{stab} = r + 1$.*

Theorem 10 *Let $\mathcal{A}_{SSAreg}$ be the algorithm in Figures 7-9 and let $n > \alpha f$. If $\alpha = 3$ and $\beta = 1$ then for each round r, such that $r > r_{stab}$ $\mathcal{A}_{SSAreg}$ implements a Self-Stabilizing MWMR Atomic register in the Garay's model.*

Proof It follows directly from Theorems 7, 8 and 9.

$\square_{Theorem\ 10}$

Theorem 11 *Let $\mathcal{A}_{SSAreg}$ be the algorithm in Figures 7-9 and let $n > \alpha f$. If $\alpha = 4$ and $\beta = 2$ then for each round r, such that $r > r_{stab}$ $\mathcal{A}_{SSAreg}$ implements a Self-Stabilizing MWMR Atomic register in the Bonnet's model.*

Proof It follows directly from Theorems 7, 8 and 9.

$\square_{Theorem\ 11}$

Theorem 12 *Let $\mathcal{A}_{SSAreg}$ be the algorithm in Figures 7-9 and let $n > \alpha f$. If $\alpha = 4$ and $\beta = 2$ then for each round r, such that $r > r_{stab}$ $\mathcal{A}_{SSAreg}$ implements a Self-Stabilizing MWMR Atomic register in the Sasaki's model.*

Proof It follows directly from Theorems 7, 8 and 9.

$\square_{Theorem\ 12}$

Theorem 13 *Let $\mathcal{A}_{SSAreg}$ be the algorithm in Figures 7-9 and let $n > \alpha f$. If $\alpha = 2$ and $\beta = 1$ then for each round r, such that $r > r_{stab}$ $\mathcal{A}_{SSAreg}$ implements a Self-Stabilizing MWMR Atomic register in the Burhman's model.*

Proof It follows directly from Theorems 7, 8 and 9.

$\square_{Theorem\ 13}$

6 Conclusion

This paper addressed the first implementation of a self-stabilizing multi-writer multi-reader atomic register tolerant to mobile Byzantine agents altogether with lower bounds on the number of replicas. We investigate four models of mobile Byzantines in round-based synchronous systems: the model of Garay *et al.* [10], where nodes have the capability to detect an infection and clean their state after the Byzantine agent leaves the node; the models of Sasaki *et al.* [17] and Bonnet *et al.* [3], where infected nodes may execute their code with a corrupted state even though the mobile agent is not anymore located at the node and finally, the model of Buhrman *et al.* [7] where Byzantines moves are tide to messages and move during the send phase. We prove that in the Garay's model self-stabilizing atomic registers can be implemented provided that in each round the number of Byzantine nodes (nodes occupied by a Byzantine agent), f, is less than $n/3$ where n is the number of correct nodes in that round while in the Bonnet's and Sasaki's models the number of Byzantine nodes f is less than $n/4$. Finally, for the case of Buhrman's model we show that f should be less than $n/2$. The convergence time of our implementation is constant. This study can be extended to the investigation of the self-stabilizing storage problem in the round-free synchronous and furthermore in the asynchronous settings.

References

[1] N. Banu, S. Souissi, T. Izumi, and K. Wada. An improved byzantine agreement algorithm for synchronous systems with mobile faults. *International Journal of Computer Applications*, 43(22):1–7, April 2012.

[2] R. A. Bazzi. Synchronous byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[3] F. Bonnet, X. Défago, T. D. Nguyen, and M. Potop-Butucaru. Tight bound on mobile byzantine agreement. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 76–90, 2014.

[4] S. Bonomi, S. Dolev, M. Potop-Butucaru, and M. Raynal. Stabilizing server-based storage in byzantine asynchronous message-passing systems: Extended abstract. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 471–479, New York, NY, USA, 2015. ACM.

[5] S. Bonomi, M. Potop-Butucaru, and S. Tixeuil. Stabilizing byzantine-fault tolerant storage. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 894–903. IEEE, 2015.

[6] S. Bonomi, A. D. Pozzo, and M. Potop-Butucaru. Tight self-stabilizing mobile byzantine-tolerant atomic register. *(Available on line on arXiv)*, 2015.

[7] H. Buhrman, J. A. Garay, and J.-H. Hoepman. Optimal resiliency against mobile faults. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'95)*, pages 83–88, 1995.

[8] D. E. Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (2):222–232, 1987.

[9] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[10] J. A. Garay. Reaching (and maintaining) agreement in the presence of mobile faults. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, volume 857, pages 253–264, 1994.

[11] L. Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[12] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[13] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Distributed Computing*, pages 311–325. Springer, 2002.

[14] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 374–383. IEEE, 2002.

[15] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 51–59, 1991.

[16] R. Reischuk. A new solution for the byzantine generals problem. *Information and Control*, 64(1-3):23–42, January-March 1985.

[17] T. Sasaki, Y. Yamauchi, S. Kijima, and M. Yamashita. Mobile byzantine agreement on arbitrary network. In *Proceedings of the 17th International Conference on Principles of Distributed Systems (OPODIS'13)*, pages 236–250, December 2013.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[19] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *Parallel and Distributed Systems, IEEE Transactions on*, 21(4):452–465, 2010.

[20] M. Yung. The mobile adversary paradigm in distributed computation and systems. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 171–172. ACM, 2015.