# Optimal Self-Stabilizing Mobile Byzantine-Tolerant Regular Register with Bounded Timestamps

Silvia Bonomi[1], Antonella Del Pozzo[3],
Maria Potop-Butucaru[2], Sébastien Tixeuil[2,4]

[1]Sapienza Università di Roma,Via Ariosto 25, 00185 Roma, Italy
bonomi@diag.uniroma1.it
[2]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
{maria.potop-butucaru, sebastien.tixeuil}@lip6.fr
[3]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
antonella.delpozzo@cea.fr
[4]Institut Universitaire de France, F-75005 Paris, France

**Abstract**

This paper proposes the first implementation of a self-stabilizing regular register emulated by $n$ servers that is tolerant to both *Mobile Byzantine Agents* and *transient failures* in a round-free synchronous model. Differently from existing Mobile Byzantine Tolerant register implementations, this paper considers a weaker model where: *(i)* the computation of the servers is decoupled from the movements of the Byzantine agents, i.e., movements may happen before, concurrently, or after the generation or the delivery of a message, and *(ii)* servers are not aware of their failure state i.e., they do not know if and when they have been corrupted by a Mobile Byzantine agent. The proposed protocol tolerates *(i)* any finite number of transient failures, and *(ii)* up to $f$ Mobile Byzantine agents. In addition, our implementation uses bounded timestamps from the $\mathcal{Z}_{13}$ domain and it is optimal with respect to the number of servers needed to tolerate $f$ Mobile Byzantine agents in the given model (i.e., $n > 6f$ when $\Delta = 2\delta$, and $n > 8f$ when $\Delta = \delta$, where $\Delta$ represents the period at which the Byzantine agents move and $\delta$ is the upper bound on the communication latency).

*Keywords:* Shared Register, Mobile Byzantine Failure, Bounded Timestamps, Self Stabilization.

## 1. Introduction

Byzantine Fault Tolerance (BFT) is a fundamental building block in distributed systems as Byzantine failures include all possible faults, attacks, virus infections and arbitrary behaviours that can occur in practice (even unforeseen ones). Such bad behaviours have been typically abstracted by assuming an upper bound $f$ on the number of Byzantine failures in the system. However, such an assumption has two main limitations: (i) it is not suited for long-lasting executions and (ii) it does not consider the fact that compromised processes/servers may be restored as infections may be blocked and confined or rejuvenation mechanisms can be put in place [33] making the set of faulty processes changing over time.

Mobile Byzantine Failure (MBF) models have been introduced to mitigate those concerns. Failures are represented by Byzantine agents that are managed by an omniscient adversary that "moves" them from a host process to another and when an agent is in some process it can corrupt it in an unforeseen manner. Models investigated so far in the context of Mobile Byzantine Failures consider mostly *round-based* computations, and can be classified according to Byzantine mobility constraints: *(i)* constrained mobility [12] agents may only move from one host to another when protocol messages are sent (similarly to how viruses would propagate), while *(ii)* unconstrained mobility [3, 5, 20, 28, 29, 31] agents may move independently of protocol messages. In the case of unconstrained mobility, several variants were investigated [3, 5, 20, 28, 29, 31]: Reischuk [29] considers that malicious agents are stationary for a given period of time, Ostrovsky and Yung [28] introduce the notion of mobile viruses and define the adversary as an entity that can inject and distribute faults; finally, Garay [20], and more recently Banu *et al.* [3], Sasaki *et al.* [31] and Bonnet *et al.* [5] consider that processes execute synchronous rounds composed of three phases: *send*, *receive*, and *compute*. Between two consecutive such synchronous rounds, Byzantine agents can move from one node to another. Hence the set of faulty processes at any given time has a bounded size, yet its membership may evolve from one round to the next. The main difference between the aforementioned four works [3, 5, 20, 31] lies in the knowledge that hosts have about their previous infection by a Byzantine agent. In Garay's model [20] and in Banu *et al.*'s model [3], a host can detect its own infection after the Byzantine agent left it and move to a *cured* state where it recovers from the compromising. Sasaki *et al.* [31] investigate a model where hosts cannot detect when Byzantine agents leave. Finally, Bonnet *et al.* [5] considers an intermediate setting where cured hosts remain in *control* on the messages they send (in particular, they send the same message to all destinations, and they do

not send fake information, *e.g.* fake id). Those subtle differences in the power of Byzantine agents turn out to have an important impact on the bounds for solving distributed problems.

A first step toward decoupling algorithm rounds from mobile Byzantine moves is due to Bonomi *et al.* [9]. In their solution to the regular register implementation, mobile Byzantine movements are synchronized, but the period of movement is independent of that of algorithm rounds.

Alternatively, *self-stabilization* [15, 16] is a versatile technique to recover from *any number of Byzantine participants*, provided that their malicious actions last for a *finite* amount of *time*. More in detail, starting from an arbitrary global state (that may have been caused by Byzantine participants), a self-stabilizing protocol ensures that the problem specification is satisfied again in finite time, without external intervention.

**Register Emulation.** Traditional solutions to build a Byzantine tolerant storage service (*a.k.a.* register emulation) can be divided into two categories: *replicated state machines* [32], and *Byzantine quorum systems* [4, 23, 25, 24]. Both approaches are based on the idea that the current state of the storage is replicated among processes, and the main difference lies in the number of replicas that are required to run the state maintenance protocol. Indeed, Schneider [32] assumes that $n > 2f$ (i.e., the number of Byzantine processes is a strict minority of the system) as operations are assumed to be ordered, and replicas just need to vote for their commitment. Conversely, in Byzantine quorum systems [4, 23, 25, 24], the ordering assumption is removed, and the number of servers required to tolerate $f$ Byzantine failures increases at least to $n > 3f$ (i.e., a strict double majority of correct processes is required).

**Multi-tolerance.** Extending the effectiveness of self-stabilization to permanent Byzantine faults is a long time challenge in distributed computing. Initial results were mostly negative [14, 27] due to the impossibility to distinguish an honest yet incorrectly initialized participant from a truly malicious one. On the positive side, two notable classes of algorithms use some locality property to tolerate Byzantine faults: *space-local* and *time-local* algorithms. Space-local algorithms [26, 27, 30] try to contain the fault (or its effect) as close to its source as possible. This is useful for problems where information from remote nodes is unimportant (such as vertex colouring, link colouring, or dining philosophers). Time-local algorithms [17, 18, 19] try to limit over time the effect of Byzantine faults. Time-local algorithms presented so far can tolerate the presence of at most a single Byzantine node. Thus,

neither approach is suitable to register emulation.

Recently, several works investigated the emulation of self-stabilizing or pseudo-stabilizing[1] Byzantine tolerant SWMR or MWMR registers [1, 8, 7]. All these works do not consider the complex case of Mobile Byzantine Failures (MBFs).

To the best of our knowledge, the problem of tolerating both *arbitrarily transient faults and mobile Byzantine faults* has been considered only in round-based synchronous systems [6]. The authors propose optimal *unbounded* self-stabilizing atomic register implementations for *round-based synchronous* systems under the four Mobile Byzantine models [3, 5, 20, 31].

**Our Contribution.** The main contribution of the paper[2] is a protocol $\mathcal{P}_{reg}$ emulating a regular register in a distributed system where both arbitrary transient failures and mobile Byzantine failures can occur. In particular, the proposed solution differs from previous work on round-free [3] register emulation [9, 10] as we add the self-stabilization property. In more detail, we present a regular register implementation that uses bounded timestamps from the $\mathcal{Z}_{13}$ domain and it is optimal[4] with respect to the number of replicas needed to tolerate $f$ mobile Byzantine agents. Finally, we prove that the maximum amount of time needed by our solution to recover a correct behavior after arbitrary transient faults occur (i.e., for starting to return a valid value as result of a $\mathsf{read}()$ operation) is upper bounded by $T_{12write()}$, where $T_{12write()}$ is the time needed to execute twelve *complete* $write()$ operations. Intuitively, this is due to the fact that the system stabilizes only when all servers have fired potentially corrupted timestamps. Considering that the number of available timestamps is $13$ and that at each $\mathsf{write}()$ operation every server adopts the most recent between the

[1] According to Burns, Gouda, and Miller [13], the difference between self-stabilization and pseudo-stabilization is subtle. On the one hand, self-stabilization requires that *if the system starts from an arbitrary state, then it is guaranteed to reach, within a finite number of transitions, a state wherefrom the system* **cannot** *deviate from its intended specification* while pseudo-stabilization requires that *if the system starts from an arbitrary state, then it is guaranteed to reach, within a finite number of transitions, a state after which the system* **does not** *deviate from its intended specification*. The direct implication is that pseudo-stabilization is strictly weaker than self-stabilization, as it only guarantees that any infinite execution has a suffix that satisfies its intended specification (but does not guarantee that a legitimate configuration is ever reached, so its stabilization time is unbounded).

[2] This paper is actually the full version of [11].

[3] A *round-free* computation is a computation that does not evolve in rounds and such computation model differs from those used in most of the related works considering MBF [3, 5, 20, 31].

[4] Our solution is optimal in the sense that using fewer replicas for tolerating $f$ mobile Byzantine agents is impossible.

one stored locally and the one associated to the write, to stabilize and converge to a shared timestamp we need at most 12 operations.

## 2. System Model

We consider a distributed system composed of an arbitrary large set of client processes $\mathcal{C}$ and a set of $n$ server processes $\mathcal{S} = \{s_1, s_2 \dots s_n\}$. Each process in the distributed system (*i.e.*, both servers and clients) is identified by a unique identifier. Servers run a distributed protocol emulating a shared memory abstraction and such protocol is transparent to clients (*i.e.*, clients do not know the protocol executed by servers). The passage of time is measured by a fictional global clock (*e.g.*, that spans the set of natural integers). At each time $t$, each process (either client or server) is characterized by its *internal state*, *i.e.*, by the set of all its local variables and the corresponding values.

**Communication model.** Processes communicate through message passing. In particular, we assume that: *(i)* each client $c_i \in \mathcal{C}$ can communicate with every server through a broadcast() primitive, *(ii)* each server can communicate with every other server through a broadcast() primitive, and *(iii)* each server can communicate with a particular client through a send() unicast primitive. We assume that communications are authenticated (*i.e.*, given a message $m$, the identity of its sender cannot be forged) and reliable (*i.e.*, spurious messages are not created and sent messages are neither lost nor duplicated).

**Timing assumptions.** Given the impossibility result proved by Bonomi et al. [9], we consider a synchronous system. The system is synchronous in the following sense: *(i)* the processing time of local computations (except for wait() statements) is negligible with respect to communication delays and is assumed to be equal to 0, and *(ii)* messages take time to travel to their destination processes. In particular, concerning point-to-point communications, we assume that if a process sends a message $m$ at time $t$ then it is delivered by time $t + \delta_p$ (with $\delta_p > 0$). Similarly, let $t$ be the time at which a process $p$ invokes the broadcast($m$) primitive, then there is a constant $\delta_b$ (with $\delta_b \geq \delta_p$) such that all servers have delivered $m$ at time $t + \delta_b$. For the sake of presentation, in the following, we consider a unique message delivery delay $\delta$ (equal to $\delta_b \geq \delta_p$), and we assume $\delta$ is known to every process. Finally, we assume that any process is equipped with a drift-free physical clock, provided by an external not corruptible trusted component, and perfectly synchronized.

**Computation model.** Each process of the distributed system executes a distributed protocol $\mathcal{P}_{reg}$ that consists of a set of distributed algorithms. Each algorithm in $\mathcal{P}_{reg}$ is represented by a finite state automaton whose transitions correspond to computation and communication steps. A computation step denotes a computation executed locally by a given process, while a communication step denotes a sending or a receipt of a message. Computation steps and communication steps are generally called *events*. Each process maintains a set of variables, and the current values of those variables denote the *state* of a process. Each communication channel maintains a multiset of messages. A message is added (resp. removed) in (from) this multiset when an adjacent process sends (resp. receives) a message to (resp. from) the other adjacent process. The set of messages in a communication channel at a given time defines the *channel state*.

The distributed protocol $\mathcal{P}_{reg}$ evolves in time across different configurations that define the protocol execution. More formally:

**Definition 1 (Configuration).** *A* configuration $\mathcal{C}$ *is a snapshot of the distributed system at some time* $t$ *that includes* (i) *the state of every process* $p_i$ *in the system (including both client and servers), and* (ii) *the state of every communication channel existing in the system.*

**Definition 2 (Execution).** *An* execution $\mathcal{E}$ *is a sequence of configurations* $\mathcal{C}_1, \mathcal{C}_2, \ldots$ *such that for any* $i > 1$, $\mathcal{C}_i$ *is reachable from* $\mathcal{C}_{i-1}$ *by executing one event (be it a computation or a communication event).*

**Definition 3 (Execution History).** *Given an* execution $\mathcal{E}$*, an* execution history $\mathcal{H}_{\mathcal{E}}$ *is a finite prefix of* $\mathcal{E}$ *at some time* $t$*.*

We will consider *round-free* executions [9] i.e., executions in which the distributed protocol $\mathcal{P}_{reg}$ does not evolve in synchronous rounds but where messages can be sent (according to the protocol) at any point in time and computation steps are executed as soon as their enabling conditions in the protocol are satisfied. This means that two different processes may manage the delivery of the same broadcast message at different times and may react and progress with the protocol independently.

We stress that the computation does not evolve in rounds to highlight the difference with most of the previous works on Mobile Byzantine Faults [3, 5, 20, 31] and highlight the impact that this additional degree of freedom has on the number of replicas required for enabling solutions (more details at the end of this section).

Given a process $p_i$ and the protocol $\mathcal{P}_{reg}$, we say that $p_i$ *is executing* $\mathcal{P}_{reg}$ *in a time interval* $[t, t']$ if $p_i$ never deviates from $\mathcal{P}_{reg}$ in $[t, t']$ (i.e., it always follows the automata transitions and never corrupts its local state).

**Definition 4 (Valid State at time $t$).** *Let $\mathcal{P}_{reg}$ be a distributed protocol, let $\bar{p}$ be a process, and let $E(\bar{p})$ be the set of all possible execution histories of $\mathcal{P}_{reg}$ such that $\bar{p}$ follows the protocol $\mathcal{P}_{reg}$. Let $t_0$ be the time at which $\mathcal{P}_{reg}$ starts. Let $state_{p_i}$ be the state of a process $p_i$ at some time $t$. We say that $state_{p_i}$ is valid at time $t$ if it is equal to one of the possible state of process $p_i$ at time $t$ in one of the execution histories in $E(p_i)$.*

Intuitively, the state of a process $p$ is valid at time $t$ if it could have been obtained by a process that remained correct until time $t$.

**Failure Model.** On the client-side, we simply assume that an arbitrary number of them may crash. On the server-side, we assume that servers are affected by *Mobile Byzantine Failures* i.e., failures are represented by Byzantine agents that are controlled by a powerful external adversary "moving" them from one server to another. Furthermore, we assume that at any time $t$, at most $f$ mobile Byzantine agents are in the system.

In this work we consider the $\Delta$-Synchronized and Cured Unaware Model, i.e., the $(\Delta S, CUM)$ MBF model, introduced by Bonomi et al. [9] that is suited for round-free computations[5]. More in detail, $(\Delta S, CUM)$ can be specified as follows:

- The external adversary moves all the $f$ mobile Byzantine agents at the same time instant, and movements happen periodically (i.e., movements happen at time $t_0 + \Delta$, $t_0 + 2\Delta$, ..., $t_0 + i\Delta$, with $i \in \mathbb{N}$).
- At any time $t \geq t_0$, no process is aware of its failure state (i.e., processes do not know if and when they have been affected by a Byzantine agent), but each process is aware of the time at which mobile Byzantine agents move. In other words, since processes are equipped with drift-free synchronized clocks with the same origin $t_0$, and that $\Delta$ is known, then they are aware of the time instants when movements of the Byzantine agents occur.

[5]More specifically, the $(\Delta S, CUM)$ model abstracts distributed systems subjected to proactive rejuvenation [33] where processes have no self-diagnosis capability.

Let us note that when we are considering Mobile Byzantine agents, no single process is guaranteed to be in the same failure state forever. Processes, in fact, may alternate between correct and incorrect behaviour infinitely often. As a consequence, it is fundamental to re-define the notion of correct and faulty processes as follows:

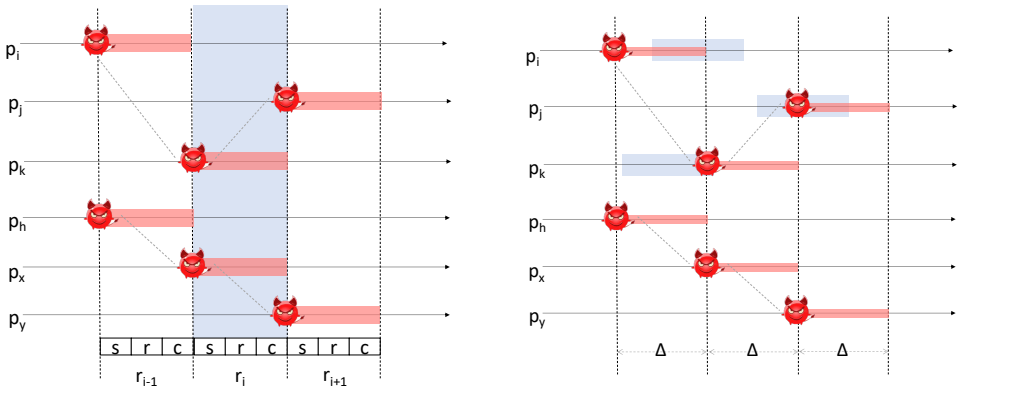
**Definition 5 (Faulty process at time $t$).** *A process is* faulty at time $t$ *if it is controlled by a mobile Byzantine agent (so it may behave arbitrarily). We denote by $B(t)$ the set of faulty processes at time $t$ while, given a time interval $[t, t']$, we denote by $B([t, t'])$ the set of processes that are faulty during the whole interval $[t, t']$ (*i.e.*, $B([t, t']) = \bigcap_{\tau \in [t,t']} B(\tau)$).*

**Definition 6 (Correct process at time $t$).** *A process is* correct at time $t$ *if* (i) *its state is valid at time $t$, and* (ii) *it is not controlled by a Byzantine agent. We denote by $Co(t)$ the set of correct processes at time $t$ while, given a time interval $[t, t']$, we denote by $Co([t, t'])$ the set of processes that are correct during the whole interval $[t, t']$ (*i.e.*, $Co([t, t']) = \bigcap_{\tau \in [t,t']} Co(\tau)$).*

**Definition 7 (Cured process at time $t$).** *A process is* cured at time $t$ *if* (i) *its state is not valid at time $t$, and* (ii) *it is not controlled by a Byzantine agent. We denote by $Cu(t)$ the set of cured processes at time $t$ while, given a time interval $[t, t']$, we denote by $Cu([t, t'])$ the set of processes that are cured during the whole interval $[t, t']$ (*i.e.*, $Cu([t, t']) = \bigcap_{\tau \in [t,t']} Cu(\tau)$).*

Let us note that, according to Definition 5, a faulty process may behave arbitrarily executing a protocol $\mathcal{P}\prime \neq \mathcal{P}$. By contrast, cured and correct processes execute the correct protocol $\mathcal{P}$. The main difference in the latter case is that cured processes may execute $\mathcal{P}$ based on an incorrect state that thus need to be cleaned. As in the case of round-based MBF models [3, 5, 12, 20, 31], we assume that every process has access to a tamper-proof memory storing the correct protocol code. So, the only processes that do not execute their prescribed protocol are those controlled by a mobile Byzantine agent. We denote by the term *honest* a process that is either correct or cured (*i.e.*, a process that honestly executes its code). Let us stress that even though at any time $t$, at most $f$ servers can be controlled by Byzantine agents, during the system lifetime, all servers may be affected by a Byzantine agent (*i.e.*, none of the servers is guaranteed to be correct forever).

In addition to being controlled by Byzantine agents, processes may also suffer from *transient* failures, *i.e.*, the state of any process (client or server) can be arbitrarily modified [16]. Let us recall that the (local) state of a process is characterized




(a) Example of a round-based computation run in Garay's MBF model.

(b) Example of a round-free computation run in $(\Delta S, *)$ MBF model.

Figure 1: Comparison of round-based and round-free executions in presence of $f = 2$ mobile Byzantine agents.

by all its local variables, including its program counter and its input buffers. It is nevertheless assumed that transient failures are quiescent, *i.e.*, in any execution of the system, there exists a time $\tau_{no\_tr}$ (unknown to the processes) after which no new transient failure occurs. However, the effect of those transient faults may persist after $\tau_{no\_tr}$, as arbitrarily modified local states may persist (after $\tau_{no\_tr}$) or propagate to other processes unless they are dealt with by the protocol.

Figure 1 provides a graphical intuition about the main differences between a round-based computation and a round-free one in presence of Mobile Byzantine Failures. In the round-based execution, we consider that the mobile Byzantine agents move at the beginning of every round (Figure 1a) as in Garay's model [20], while they move with a period of $\Delta$ in Figure 1b. We consider a period equal to the communication latency to compare the round-free execution with the round-based one. The round-based computation evolves in synchronous and synchronized rounds, where all messages are sent together at the beginning of the round, and then processed just before the end of the round. Conversely, in round-free computations, messages can be sent by any process independently from the others (see e.g., blue areas for $p_i$, $p_j$ and $p_k$). The main consequence is that in round-based computations, the $f$ mobile Byzantine agents affect processes "symmetrically" (i.e., all equally with respect to messages), and in a specific phase of the round (e.g., in Garay's model, they affect the send phase), with effects limited to messages generated by faulty processes in the considered round (see the blue area in Figure 1a). A

mobile agent may still move periodically in a round-free computation, but its movement is independent from messages potentially exchanged between processes. As a result, a mobile Byzantine agent may impact "asymmetrically" messages exchanged concurrently by processes (see the blue area in Figure 1b).

## 3. Self-Stabilizing Regular Register Specification

A register is a shared variable accessed by a set of processes, *i.e.* clients, through two operations, namely read() and write(). Informally, the write() operation updates the value stored in the shared variable while the 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. 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 write($v$) operation, the value $v$ is said to be written when the operation is complete. We assume that locally any client never performs read() and write() operations concurrently (*i.e.*, for any given client $c_i$, the set of operations executed by $c_i$ is totally ordered). 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 registers have been defined by Lamport [22]: *safe*, *regular* and *atomic*.

In this paper, we consider a Self-Stabilizing Single-Writer/Multi-Reader (SS-SWMR) regular register, *i.e.*, an extension of Lamport's regular register that considers transient failures.

Let us consider a distributed protocol $\mathcal{P}$ and let $\mathcal{E}_{\mathcal{P}}$ be the set of all the possible executions generated by $\mathcal{P}$. Then, $\mathcal{P}$ satisfies the *Single-Writer/Multi-Reader* (SWMR) register specification if and only if it satisfies the following properties:

- Termination: Every operation $op$ invoked by a non-crashed client running $\mathcal{P}$ terminates in every execution of $\mathcal{P}$.

- Validity: in every execution $e \in \mathcal{E}_{\mathcal{P}}$, every completed read() operation returns either the value written by the last completed write(), or a value written by a concurrent write() operation.

Now, self-stabilization, introduced by Dijkstra [15], is a forward recovery mechanism that permits distributed systems to resume correct behaviour after arbitrary transient faults hit the system. The faults are transient in the sense they are supposed to have occurred *before* the execution is started. This is conveniently abstracted as starting the execution from an arbitrary configuration (that could have been caused by arbitrary transient faults). However, no further memory corruptions occur after the execution starts[6].

Let us consider a distributed protocol $\mathcal{P}$ and let $\mathcal{E}_{\mathcal{P}}$ be the set of all the possible executions generated by $\mathcal{P}$. Then, $\mathcal{P}$ satisfies the *Self-Stabilizing Single-Writer/Multi-Reader* (SS-SWMR) register specification if and only if there exists a non-empty subset of configurations $\mathcal{L}$ such that the following properties are satisfied:

- ss $-$ Correctness: Starting from a configuration in $\mathcal{L}$, every execution of $\mathcal{P}$ satisfies Termination and Validity.
- ss $-$ Convergence: in every execution $e \in \mathcal{E}_{\mathcal{P}}$, there exists a configuration in $\mathcal{L}$.

In general, the set $\mathcal{L}$ of legitimate configurations depends on the considered algorithm. The particular set we use is presented in Section 5.1.

In the sequel, we will say that the execution of a self-stabilizing algorithm is in a *stable* phase when it reaches a legitimate configuration $l \in \mathcal{L}$. By extension, any configuration occurring after the first legitimate configuration $l \in \mathcal{L}$ is a stable configuration.

## 4. A Self-Stabilizing Regular Register Implementation

In this section we propose a protocol $\mathcal{P}_{reg}$ implementing a self-stabilizing SWMR regular register in the $(\Delta S, CUM)$ MBF model. Our algorithm copes with

[6]Observe that, in our case, we assume both arbitrary transient faults and mobile Byzantine faults. So, the initial configuration is arbitrary, and processes may be corrupted infinitely often by mobile agents. Of course, it is impossible to distinguish an honest process whose memory is corrupted (but executes genuine code) from a faulty one (that executes malicious code).

the $(\Delta S, CUM)$ model following the path of Bonomi et al. [9], yet improves the result by making the solution self-stabilizing while retaining bounded timestamps. Our solution considers two particular movement frequencies that induce different requirements with respect to the required number of servers:

1. $\Delta = \delta$, requiring $n \geq 8f + 1$ servers and
2. $\Delta = 2\delta$, requiring $n \geq 6f + 1$ servers.

We implement read() and write() operations following a classical quorum-based approach[2] and exploiting the system synchrony to guarantee their termination. Informally, when the writer client wants to write, it simply propagates the new value to servers, that update their local copy of the register. Then, when a reader client wants to read, it asks for the current value of the register and waits for sufficiently many replies: after $3\delta$ time, "*enough*"[7] replies are received, and a consistent value can be selected and returned as the result of the read() operation.

Of course, the duration of the read() operation has an impact on the number of server replicas that are required to guarantee a correct implementation. Indeed, the longer the read() operation lasts, the higher the number of servers that can be corrupted by a mobile Byzantine agent (and that can send a corrupted value as a reply to the client) is. As a consequence, there is a direct relationship between the duration of the read(), and the number of replies (i.e., the size of the quorum) needed to ensure the validity of the operation.

To deal with Mobile Byzantine agents that can progressively compromise local copies of the register and lead to the loss of persistence of the last written value, we introduce a maintenance() operation whose aim is to guarantee the existence of a sufficient number of servers storing a *valid value* $v$ for the register *i.e.*, a value $v$ that has been written during the last write() operation or during one (or more) write() operation(s) running concurrently with the maintenance() operation. The basic idea behind the maintenance() operation is to implement a propagation mechanism (using echoes) that spreads the state of correct servers to cured ones and allows cured servers to recover from a possibly corrupted state to a valid one. In doing this, the maintenance() needs to take care of the following issues:

1. ensuring that cured servers (i.e., servers that were previously compromised by a Byzantine agent) get a valid value at the end of maintenance() and restore their state to a valid one,

[7]The exact number of expected replies is provided in Table 1, as it depends on the values of $\Delta$ and $\delta$.

2. written values that are possibly concurrent with a read() are always taken into account by cured servers running maintenance(), and
3. correct servers do not overwrite a valid value with a non-valid one.

Each server $s_i$ stores three pairs $\langle value, timestamp \rangle$ corresponding to the last three written values, and periodically (that is, when Byzantine agents move at every $T_i = t_0 + i\Delta$, with $i \in \mathbb{N}$) executes the maintenance() operation.

The key idea to recover a valid state at the end of the maintenance() is to keep separated the information that can be trusted (e.g., values received directly from the writer or values received from "enough" servers), from the untrusted information (e.g., values stored locally that could have been compromised by the Byzantine agent).

To this aim, the maintenance() operation makes use of three set variables stored by every server $s_i$:

- $V_i$ is used to store the knowledge of $s_i$ at the beginning of each maintenance() operation. It contains the last three values of the register and their corresponding timestamps as per $s_i$ knowledge. This set contains untrusted information, as the values and/or timestamps may have been corrupted by the Byzantine agent before the beginning of the maintenance() operation.

- $V_{safe_i}$ is used to collect *safe* values, i.e., values that can be considered valid since they were selected among those that received enough echoes recently. This variable is emptied at the beginning of every maintenance() operation after its value is propagated.

- $W_i$ is used to store values (and their corresponding timestamps) received from the writer during the execution of a maintenance() operation. This set contains untrusted information, as it could have been compromised by the Byzantine agent before it leaves the server.

In the following, we informally explain how the maintenance() operation is implemented, and we highlight the main issues and how we address them through an example. Let us consider the execution depicted in Figure 2 showing the $i$-th and the $(i+1)$-th maintenance() operations starting respectively at time $t_0 + i\Delta$ and $t_0 + (i+1)\Delta$. Let us consider two servers $s_0$ and $s_1$ that are respectively correct and cured at time $t_0 + i\Delta$. In the figure, both servers report values of their respective sets (i.e., $W_i, V_i, V_{safe_i}$) during the execution of two maintenance() operations and we also consider the case of a concurrent write() operation. Note that $s_0$ is correct, so values stored in $V_0$ (i.e., $V_0 = \{0, 1, 2\}$) are valid values. Now,

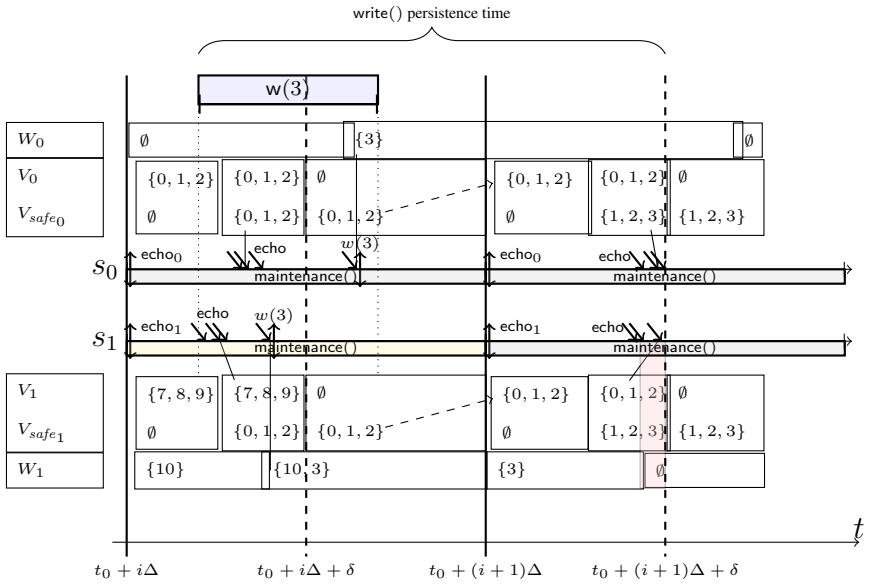



Figure 2: A possible execution segment of the protocol for a correct server $s_0$ and a cured server $s_1$, with $\Delta = 2\delta$. For the sake of simplicity, we report only timestamps instead of the pair $\langle value, timestamp \rangle$.

$s_1$ starts the maintenance() operation with $V_1 = \{7, 8, 9\}$ and $W_1 = \{10\}$ that have been altered by the Byzantine agent before departing and leaving $s_1$ in the cured state. $V_{safe_i}$ is empty for both processes since it is the set of values that has to be set during the maintenance(). In the figure, we use $\updownarrow$ for the broadcast of the ECHO messages and the WRITE messages from the client. We use incoming diagonal arrows for the delivery of the echo and write messages.

When maintenance() starts, every honest server $s_i$ echoes the relevant information stored locally (i.e., list of pending read() operations, together with the sets $V_i$ and $W_i$ represented in Figure 2 with the $\updownarrow$ symbol at the beginning of the run). Such information is then collected by every server $s_j$ and can be used (based on the number of occurrences of each pair $\langle$value, timestamp$\rangle$ received) to update the set $V_{safe_j}$. In particular, $V_{safe_j}$ receives the values that have been echoed at least $\#echo_{CUM}$ times[8]. Let us note that, due to system synchrony and reliable communication primitives, after $\delta$ time units (i.e., at time $t_0 + i\Delta + \delta$), $s_i$ collects at

[8]The exact number of required echoes depends on the relationship between $\Delta$ and $\delta$, and is discussed later in the text.

least all the values sent by all honest servers, and $s_i$ is able to decide and update its local variables. Thus, $s_i$ selects the values occurring at least $\#echo_{CUM}$ times (see Table 1) from echoes, updates $V_{safe_i}$, and empties $V_i$. In Figure 2 at time $t_0+i\Delta+\delta$, $s_0$ sets $V_{safe_0} = V_0$ (as expected since we assume $s_0$ is correct), while $s_1$ updates $V_{safe_1} = V_0$ set consistently with $s_0$ thanks to the values collected through the echo messages.

However, the maintenance() operation is not yet complete as a write() operation may occur concurrently with a value not yet in $V_{safe_i}$ (and thus in $V_i$). In order to manage this case, any time a value is written, it is relayed to all servers. In addition, to avoid overwriting values newly written with those selected by the maintenance() operation, concurrently written values are temporarily stored in $W_i$ with an associated timer (i.e., like a time-to-live) set to $2\delta$. On the one hand, the timer is set in such a way that each value in $W_i$ remains stored long enough to ensure its propagation to all servers and guarantees that written values eventually appear in $V_{safe_i}$ set for every honest server (e.g., the value 3 in Figure 2). On the other hand, the $2\delta$ period is not long enough to allow mobile Byzantine agents to take advantage of the propagation of corrupted values through echoes and to make reader clients return a corrupted value (e.g., the value 10 left by the mobile Byzantine agent in $W_1$ at the beginning of the $i$-th maintenance() operation).

Let us recall the importance of values stored in the $V_{safe_i}$ set. Indeed, those values are echoed by $s_i$ at the beginning of each maintenance(), and then recollected to assess their validity, while values in the other sets are simply reset. In the scenario depicted in Figure 2, the value 3 appears in the $V_{safe_i}$ set of both servers only at time $t_0 + (i + 1)\Delta + \delta$. At the beginning of the second maintenance() operation, the value 3 contained in $W_i$ is echoed along with the values in $V_i$ (that contains the values that were in the $V_{safe_i}$ set before $V_i$ was reset). Thus, after $\delta$ time from the beginning of the second maintenance() operation, the value 3 is in $V_{safe_i}$. We call the time necessary for a value to be present in $V_{safe_i}$ the *write persistence* time.

Note that, depending on the relationship between $\Delta$ and $\delta$, a maintenance() operation may be triggered while the previous one is not yet complete. This is not an issue since the set $V_i$ is updated before the second maintenance() operation starts, and $W_i$ is the only set that is not reset between the two maintenance() operations. Furthermore, $W_i$ values remain for a specific time-to-live of $2\delta$ that is sufficient to propagate them before they are discarded.

Finally, concurrently with the maintenance() and write() operations, servers are required to answer also to clients that perform read() operations. To preserve the validity of read() operations, and cope with the possible corruptions made

| $k = \lceil \frac{3\delta}{\Delta} \rceil$ | $n_\Delta \geq (2k+2)f+1$ | $\#reply_{CUM} \geq 2kf+1$ | $\#echo_{CUM} \geq kf+1$ |
|---|---|---|---|
| $\Delta = \delta, k = 3$ | $8f+1$ | $6f+1$ | $3f+1$ |
| $\Delta = 2\delta, k = 2$ | $6f+1$ | $4f+1$ | $2f+1$ |

Table 1: Parameters for $\mathcal{P}_{reg}$ Protocol.

by mobile Byzantine agents when leaving, a server $s_i$ replies with all the values it is currently storing (i.e., providing $V_i$, $V_{safe_i}$, and $W_i$). Note that, given the update mechanism of those variables (designed to separate trusted information from untrusted information), there could be a period of time where the last written value is removed from $W_i$ (as its timer is expired), yet it is not inserted in $V_{safe_i}$ nor $V_i$ (as the corresponding propagation message is still travelling - cf. the red zone in Figure 2). To cope with this issue, the read() operation lasts $3\delta$ time, i.e., an extra waiting period is added for collecting sufficiently many replies, so that values are not lost. The way the client analyzes values collected from the servers in order to return a value is explained in detail in the sequel.

In Table 1 we report parameters used by the algorithm for both scenarios $\Delta = \delta$, and $\Delta = 2\delta$. In particular, those parameters include:

- $n_\Delta$: the number of servers for the specific mobile Byzantine agent movement period $\Delta$ (when $\Delta$ is clear from the context or it is not relevant we will simplify the notation by using $n$ to denote the number of servers);
- $\#reply_{CUM}$: the number of occurrences of the same value $v$ a client needs to receive to trust value $v$ and consider it valid;
- $\#echo_{CUM}$: the number of occurrences of the same value $v$ that a server needs to receive to trust value $v$ and consider it valid for the maintenance() operation (i.e., to store it in $V_{safe_i}$).

All those parameters are a function of $k = \lceil \frac{3\delta}{\Delta} \rceil$, where $3\delta$ is the duration of both maintenance() and read() operations. Thus, $k$ is a measure of how many times Byzantine agents may move between servers during those operations. Let us recall that this parameter has a direct impact on the number of servers required to tolerate $f$ Mobile Byzantine agents. Interestingly, $\#echo_{CUM} < \#reply_{CUM}$, since the maintenance() operation starts exactly when Byzantine agents move, thus fewer servers can be affected during maintenance() than during read() (which can start at any time, and can overlap multiple movements periods).

Finally, there remains to discuss the way bounded timestamps are employed. To stabilise in finite time and manage transient failures, $\mathcal{P}_{reg}$ employs bounded timestamps. It is important to note that timestamps are necessary in the $(\Delta S, CUM)$

model as, during the maintenance(), servers must be able to distinguish new and old values, to guarantee that a new value (possibly received from the writer) is not overwritten by the maintenance() operation. In the following, we explain how using bounded timestamps guarantees a finite and known stabilization period.

Let us note that, in the presence of transient failures, a necessary condition for stabilization is that at least one write() operation is executed after time $\tau_{no\_tr}$. However, this condition is not sufficient. Since this operation is the first one after $\tau_{no\_tr}$, its associated timestamp could be arbitrary as well as other timestamps stored in the system. For example, if timestamps can take all values in $\mathbb{N}$ as in previous work [6, 9, 10], and the usual total order relationship on integers is used to compare them, the timestamp used by the writer might be strictly smaller than those stored locally by servers. Then, the newly written value is ignored by servers as it is perceived as too old. This process may repeat until the writer timestamp exceeds the highest timestamp initially stored by servers, yielding an unbounded stabilization time.

To avoid this issue, we use a bounded set of timestamps in the domain $\mathcal{Z}_m$[9] and we define an order relation on the timestamps. Let us note that using a bounded number of timestamps requires to periodically reuse the same sequence number to label different write() operations. On the other side, to guarantee the validity property of a regular register, we need that clients are able to distinguish between the current value of the register and its previous ones. Thus, it is fundamental to define a sound ordering relation between timestamps that allows clients to compare among two different timestamps, and to guarantee that eventually, the algorithm can safely reuse a previously used timestamp without generating conflicts. This can be implemented using arithmetic with module i.e., by placing timestamps over a ring, and cycling as new timestamps are selected. If we can guarantee that only timestamps "close to each other" are used at any time, then we can compare them without ambiguity, and uniquely choose the greatest one.

To this aim, we define the *addition* operation $+_m$ to increment timestamps (modulo $m$), and we define how to order timestamps based on the intuitive notion of *distance*.

**Definition 8 (Addition operation in $\mathcal{Z}_m$).** *Let $\mathcal{Z}_m$ be the subset of the first $m$ consecutive integers. We define as addition $+_m : Z_m \times Z_m \rightarrow Z_m, a +_m b = (a + b)$ (mod $m$).*

[9] We will show in the correctness proofs that $m = 13$ is enough for our algorithm to work correctly.

**Definition 9 (Distance between timestamp).** *Let $ts_i$ and $ts_j$ be two timestamps such that $ts_i, ts_j \in \mathcal{Z}_m$. The* distance *between $ts_i$ and $ts_j$, denoted by $dist(ts_i, ts_j)$, is given by the smallest number $k$ that needs to be added (modulo $m$) to $ts_i$ to get $ts_j$ (i.e., $dist(ts_i, ts_j) = k$ iff $ts_i +_m k = ts_j$).*

As an example, if we consider $ts_i, ts_j \in \mathcal{Z}_{13}$, we get that the $dist(1, 4) = 3$ and $dist(10, 2) = 5$. Let us observe that for a given $m$, given a pair of timestamps $ts_i, ts_j \in \mathcal{Z}_m$, $dist(ts_i, ts_j) + dist(ts_j, ts_i) = m$.

**Definition 10 (Ordering relation between timestamps).** *Let $m$ be the maximum number of possible timestamps, and let $\mathcal{Z}_m$ be the set of possible timestamps. For any given pair $ts_i, ts_j \in \mathcal{Z}_m$ we say that:*

- *$ts_i$ is equal to $ts_j$ i.e., $ts_i =_m ts_j$ if $dist(ts_i, ts_j) = dist(ts_j, ts_i) = 0$*
- *$ts_i$ is greater than $ts_j$ i.e., $ts_i >_m ts_j$ if $dist(ts_j, ts_i) < dist(ts_i, ts_j)$*
- *$ts_i$ is smaller than $ts_j$ i.e., $ts_i <_m ts_j$ if $dist(ts_i, ts_j) < dist(ts_j, ts_i)$*
- *$ts_i$ is incomparable to $ts_j$ i.e., $ts_i \neq_m ts_j$ if $dist(ts_i, ts_j) = dist(ts_j, ts_i)$ and $dist(ts_i, ts_j) \neq 0$*

As an example, considering timestamps in $\mathcal{Z}_{13}$, $4 >_{13} 1$ (that is, $dist(1, 4) = 3$ is smaller than $dist(4, 1) = 10$), and $2 >_{13} 12$ (that is, $dist(12, 2) = 3$ is smaller than $dist(2, 12) = 10$).

Let us note that our ordering relation is not transitive, i.e., if we have three timestamp $ts_i, ts_j, ts_k \in \mathcal{Z}_m$ with $ts_i >_m ts_j$ and $ts_j >_m ts_k$, this does not imply that $ts_i >_m ts_k$. Indeed, if we consider timestamps $1$, $5$, and $11$ in $\mathcal{Z}_{13}$, we have $11 >_{13} 5$, $5 >_{13} 1$, but $1 >_{13} 11$.

**Definition 11.** *Let $TS$ be a subset of timestamps in $\mathcal{Z}_m$ (i.e., $TS \subseteq \mathcal{Z}_m$). Then, $TS$ is* unequivocally ordered *if it is possible to construct a sequence $S_{TS} = \{ts_1, ts_2, \ldots ts_{|TS|}\}$ such that* (i) *$S_{TS}$ includes every timestamp in $TS$, and* (ii) *for any pair of timestamps $ts_i, ts_j \in S_{TS}$ (with $j > i$), we have $ts_i <_m ts_j$ (i.e., $\forall i \in [1, |TS| - 1], \forall j > i, ts_i <_m ts_j$).*

The pseudo-code for $\mathcal{P}_{reg}$ is shown in Figures 3 - 5.

**Local variables at client $c_i$.** Each client $c_i$ maintains a set $reply_i$ that is used during the read() operation to collect replies coming from servers. In particular,

$reply_i$ contains tuples $\langle v, sn \rangle_j$, where $j$ is the sender server, and $\langle v, sn \rangle$ refers to a value $v$ and its associated timestamp $sn$. Additionally, if $c_i$ is the writer, it maintains a timestamp variable $csn_i$.

**Local variables at server $s_i$.** Each server $s_i$ maintains the following local variables:

- $V_i$ and $V_{safe_i}$: two sets both storing pairs $\langle v, sn \rangle$, where $v$ is a value, and $sn$ is its corresponding timestamp. Those sets are used to collect values and select valid ones. The size of the set $V_i$ is restricted to store up to 3 pairs.
- $W_i$: a set storing temporary values coming directly from the writer. The set contains triples $\langle v, sn, timer \rangle$, where $v$ is a value, $sn$ is its associated timestamp, and $timer$ is a time-to-live timer. When the timer expires, the triple is deleted from $W_i$.
- $echo\_vals_i$: a set used to collect information propagated through ECHO messages. It stores tuples in the form $\langle v, sn \rangle_j$, where $v$ is a value, $sn$ is its associated timestamp, and $j$ is the server who sent the tuple encapsulated in the ECHO message.
- $pending\_read_i$: a set containing identifiers of the clients that are currently performing a read() operation.

To simplify the code of the algorithm and make it modular, we define the following functions:

- select_pairs($echo\_vals_i$): this function takes as input the set $echo\_vals_i$, and returns tuples $\langle v, sn \rangle$, such that there exist at least $\#echo_{CUM}$ occurrences of $\langle v, sn \rangle$ in $echo\_vals_i$.
- insert($V_{safe_i}, \langle v_k, sn_k \rangle$): this function inserts $\langle v_k, sn_k \rangle$ in $V_{safe}$ according to the order on $\mathcal{Z}_m$. If there are more than three values after insertion, then the oldest one is discarded. In case it is not possible to uniquely order the elements in the set after insertion, then $V_{safe_i}$ is reset (this may happen due to transient failures).
- select_value($reply_i$): if the subset of pairs $\langle v, sn \rangle$ that occur $\#reply_{CUM}$ times or more in $reply_i$ can be uniquely ordered (according to their timestamp), this function returns the newest such pair $\langle v, sn \rangle$. If no such ordering exists, this function returns $\emptyset$.

- checkOrderAndTrunc($V_{safe_i}$): this function checks if it is possible to uniquely order the elements in $V_{safe_i}$ with respect to their timestamps. If yes, the 3 newest elements are kept, and the other elements are removed from $V_{safe_i}$. If it is impossible to uniquely establish an order, then all the elements are deleted from $V_{safe_i}$.

- merge($V_i, W_i$): this function returns a set of tuples $\langle v, sn \rangle$ obtained by merging pairs from $V_i$ and pairs extracted from $W_i$ by removing timers from triples occurring in $W_i$.

- conCut($V_i, V_{safe_i}, W_i$): this function takes as input the three sets $V_i$, $V_{safe_i}$, and $W_i$, and returns a set of pairs $\langle v, ts \rangle$. The returned set is obtained by *(i)* merging the pairs $\langle v, ts \rangle$ extracted from $V_{safe_i}$, $V_i$, and $W_i$, *(ii)* selecting the 3 newest values (with respect to the timestamps). As an example, let us consider $\mathcal{Z}_{13}$ and the following sets: $V_i = \{\langle v_a, 1\rangle, \langle v_b, 2\rangle, \langle v_c, 3\rangle\}$ and $V_{safe_i} = \{\langle v_b, 2\rangle, \langle v_d, 4\rangle, \langle v_f, 5\rangle\}$, and $W_i = \emptyset$, then the returned set is

  $$\{\langle v_c, 3\rangle, \langle v_d, 4\rangle, \langle v_f, 5\rangle\}$$

  If the set of timestamps cannot be unequivocally ordered, then the function returns $\emptyset$.

- checkTimer($W_i$): this function checks all triples $\langle v, sn, timer \rangle$ in $W_i$, and removes those whose timer has expired, or is greater than $2\delta$ (this may only happen due to a transient failure).

**The** maintenance() **operation (Figure 3).** This operation is executed by servers periodically at every time $t_j = t_0 + j\Delta$, for every $j \in \mathbb{N}$ and its triggered when $timer\_maintenance$ expires (lines 1 - 3 and lines 15 - 17). Each time maintenance() is executed, a server $s_i$ stores the content of $V_{safe_i}$ in $V_i$, and all $V_{safe_i}$ and $echo\_vals_i$ sets are reset. Each server $s_i$ then broadcasts an ECHO message with $V_i$, $W_i$, and $pending\_read_i$ set. When a server $s_j$ receives the ECHO message, it updates its local variable storing the current received information and then, when there is at least one value in $echo\_vals_j$ set that occurs at least $\#echo_{CUM}$ times, $s_j$ tries to update $V_{safe_j}$ to store values occurring "enough" time (lines 19 - 27). To conclude, after $\delta$ time since the beginning of the current maintenance() operation, $V_i$ is reset. Informally speaking, during the maintenance() operation, $V_{safe_i}$ is filled with safe values, then the content in $V_i$ is no longer necessary. Remind that the content of $W_i$ is continuously monitored so that expired values are removed (line 18).

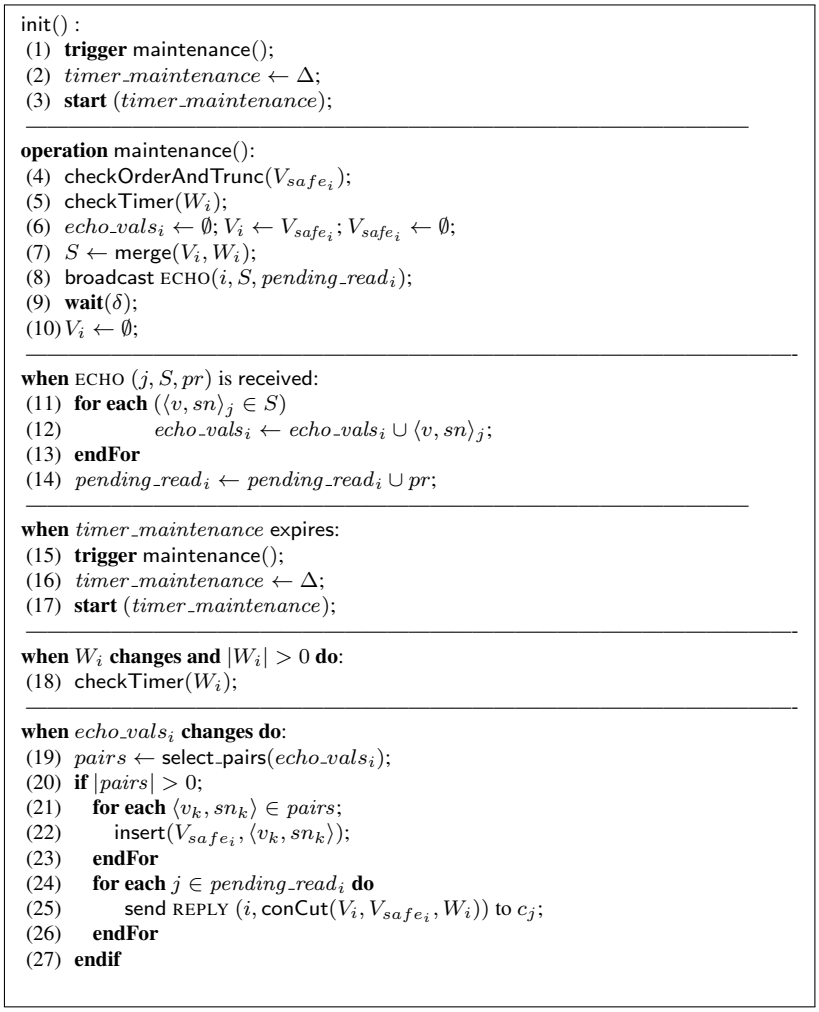



Figure 3: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server $s_i$) in the $(\Delta S, CUM)$ model with bounded timestamp.

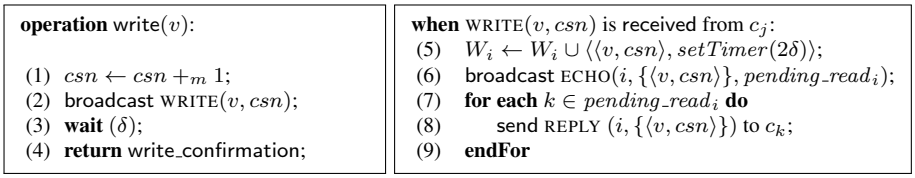



Figure 4: $\mathcal{A}_W$ algorithms, client-side and server side respectively, implementing the write($v$) operation in the $(\Delta S, CUM)$ model with bounded timestamp.

**The** write() **operation (Figure 4)**. When the write() operation is invoked, the writer $c_i$ increments its local sequence number to timestamp the current operation, sends WRITE($\langle v, csn \rangle$) to all servers, and finally returns from the operation after waiting $\delta$ time (i.e., the maximum time needed to deliver the WRITE message to honest servers). When an honest server $s_j$ delivers a WRITE($\langle v, csn \rangle$) message it stores $v$ in $W_j$, and forwards ECHO($j, \langle v, csn \rangle, pending\_read_j$) to every server. Let us note that such value is further echoed at the beginning of each subsequent maintenance() operation, as long as $\langle v, csn \rangle$ is in $W_j$ or $V_j$[10]. Finally, every honest server will also answer to any pending read request stored in $pending\_read_i$ by sending the newly written value (to ensure that the reader will eventually collect "enough" replies to return a valid value).

**The** read() **operation (Figure 5)**. When the read() operation is invoked, the reader $c_i$ empties $reply_i$, sends to all servers the READ($i$) message and remains waiting for $3\delta$ time to collect replies. At the end of this waiting period, the reader $c_i$ picks the newest value (according with the timestamp unique ordering sequence) occurring at least $\#echo_{CUM}$ times by invoking select_value($reply_i$), and returns it. Notice that before returning, $c_i$ sends to every server the read termination notification READ_ACK($i$) message.

On the server side, when $s_j$ delivers the READ($i$) message, client $c_i$'s identifier is stored in the $pending\_read_j$ set. This set is part of the content of the ECHO messages in every maintenance() operation, which populates the $pending\_read_j$ set, so that cured servers can be aware of the reading clients. Afterwards, $s_j$ invokes conCut($V_j, V_{safe_i}, W_i$) function to prepare the reply message for $c_i$. The result of this function is sent back to $c_i$ in the REPLY message. Lastly, $s_j$ broadcast a READ_FW($i$) message to spread the information that client $c_i$ is currently reading. This is done to let cured servers aware about the read as they could have potentially miss it due to a mobile Byzantine agent. When a READ_FW($j$) message is delivered

[10]This holds for at least $\#echo_{CUM}$ correct servers.

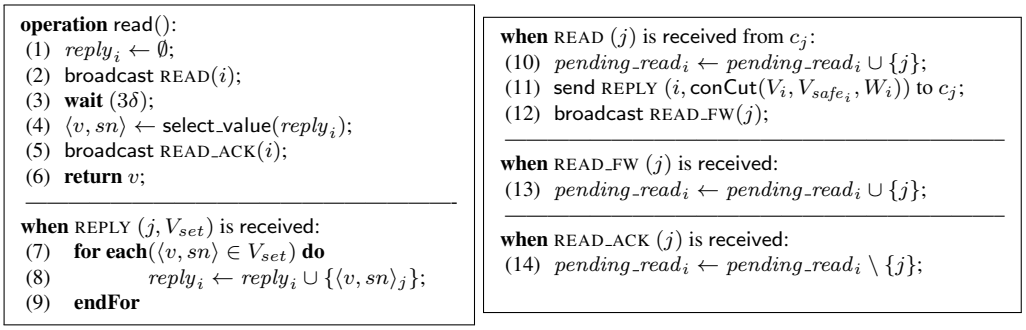



Figure 5: $\mathcal{A}_R$ algorithms, client-side and server-side respectively, implementing the read() operation in the $(\Delta S, CUM)$ model with bounded timestamp.

by server $s_i$, the identifier of client $c_j$ is stored in the $pending\_read_i$ set. When the READ_ACK$(i)$ message is delivered from $c_i$, then its identifier is removed from the $pending\_read_j$.

## 5. Correctness proofs

In this Section, we prove that $\mathcal{P}_{reg}$ is correct, and is optimal with respect to the number of servers needed to cope with $f$ Mobile Byzantine agents in the system model we consider. In particular, the Section is structured as follows:

1. in Section 5.1, we introduce some preliminary definitions that will be used in the proofs;
2. in Section 5.2, we prove that $\mathcal{P}_{reg}$ satisfies the Termination property independently from the initial state of the algorithm;
3. in Section 5.3, we show the sufficient conditions for $\mathcal{P}_{reg}$ to reach a legitimate configuration $\mathcal{L}$ starting from an arbitrary initial configuration;
4. in Section 5.4, we prove that when starting from a legitimate configuration $l \in \mathcal{L}$, $\mathcal{P}_{reg}$ satisfies the Validity property. Thus, $\mathcal{P}_{reg}$ satisfies ss $-$ Correctness as soon as it starts from a legitimate configuration $l \in \mathcal{L}$;
5. finally, Section 5.5 provides a proof that $\mathcal{P}_{reg}$ is optimal with respect to the number of servers needed in the considered system model to cope with $f$ Mobile Byzantine agents.

### *5.1. Preliminary Definitions*

In this section, we define *legitimate configurations* for Protocol $\mathcal{P}_{ref}$. Since our convergence proof in Section 5.3 makes use of *convergence stairs* [21] (that is,

intermediate predicates satisfied by configurations), we also define *semi-legitimate* configurations, a weaker requirement than legitimate configurations. In addition, we introduce notations to denote and identify the *set of faulty servers in a time interval* and the *maximum number of faulty servers in a time interval*. Finally, we show that under our MBF system model, it is possible to establish an upper bound on the maximum number of faulty processes in a given time interval, based on model parameters.

**Definition 12 (Semi-legitimate configuration $\mathcal{SL}$ for $\mathcal{P}_{reg}$).** *A configuration $c$ is a* semi-legitimate configuration *for Protocol $\mathcal{P}_{reg}$ (and its set is denoted by $\mathcal{SL}$) if it is a configuration at time $t$ where there exists at least $n - 2f$ servers storing the last value $v$ written on the register by a completed* write() *operation (i.e., $\langle v, - \rangle \in state_{p_i}$ for at least $n - 2f$ servers $s_i$).*

**Definition 13 (Legitimate Configuration for $\mathcal{P}_{reg}$).** *A configuration $c$ is a* legitimate configuration *for Protocol $\mathcal{P}_{reg}$ (and its set is denoted by $\mathcal{L}$) if it is a configuration at time $t$ where there exists at least $n - 2f$ servers with a valid state at time $t$.*

Informally, a legitimate configuration is a snapshot where at least $n-2f$ servers are maintaining locally valid values according to the execution history generated. Let us note that, since we are considering a single-writer regular register, the state of every server can be updated only by the writer itself and it is valid if it stores values associated with the last completed write() operation or to the concurrent written one. Considering that the system is synchronous and that at most two valid states exist at any given time, it follows that requiring just $n - 2f$ valid states is enough as they will always be consistent among them.

**Definition 14 (Faulty servers in an interval).** *Let us define as $\tilde{B}([t, t + T])$ the set of servers that are affected by a Byzantine agent for at least one time unit in the time interval $[t, t + T]$. More formally: $\tilde{B}([t, t + T]) = \bigcup_{\tau \in [t, t+T]} B(\tau)$.*

**Definition 15 (Maximum number of faulty servers in an interval).** *Let $[t, t+T]$ be a time interval. The cardinality of $\tilde{B}([t, t + T])$ is maximum if for any $t'$, $t' > 0$, it is true that $|\tilde{B}([t, t + T])| \geq |\tilde{B}([t', t' + T])|$. Let $Max\tilde{B}([t, t + T])$ denote the maximum such cardinality.*

**Lemma 1.** *Let $\Delta$ be the time period between two consecutive movements of the Mobile Byzantine agents and let $T$ be the length of a specific time period. If $\Delta > 0$, then $Max\tilde{B}([t, t + T]) = (\lceil \frac{T}{\Delta} \rceil + 1)f$.*

**Proof** For simplicity, let us first consider a single Mobile Byzantine agent $ma_1$ (i.e., $f = 1$), and let us consider a generic time interval $[t, t + T]$. In this period, if $T \geq \Delta$, $ma_1$ can affect a different server moving every $\Delta$ time (in the worse cast at every time $t_i = t + i\Delta$ with $t_i \leq t + T$). It follows that the number of times that $ma_1$ may "jump" from a server to another is $\frac{T}{\Delta}$. In the worst case, $ma_1$ moves every time to compromise a server that is still correct and thus the affected servers are at most $\lceil \frac{T}{\Delta} \rceil$ plus the server on which $ma_1$ is at $t$. Now, if $0 < T < \Delta$, a Byzantine agent can affect at most two nodes during $T$. In any case, a Byzantine agent can corrupt up to $\lceil \frac{T}{\Delta} \rceil + 1$ nodes during $T$. Generalizing the reasoning to $f$ agents we get $Max\tilde{B}([t, t + T]) = (\lceil \frac{T}{\Delta} \rceil + 1)f$. $\square_{Lemma\ 1}$

*5.2. Termination proof*

The termination property is guaranteed by design, as the proposed protocol exploits the synchronous timing assumption to infer when every operation can be considered completed based on the worse case message communication pattern, i.e., after a fixed period of time all operations invoked by correct clients terminate.

**Lemma 2.** *If a client $c_i$ invokes* write$(v)$ *at time $t$, and does not crash, then* write$(v)$ *terminates at time $t + \delta$.*

**Proof** The claim simply follows from the fact that a write_confirmation event is returned to the writer client $c_i$ after $\delta$ time, regardless of the servers' behaviour (see lines 3-4, Figure 4). $\square_{Lemma\ 2}$

**Lemma 3.** *If a client $c_i$ invokes a* read() *operation at time $t$, and does not crash, then* read() *terminates at time $t + 3\delta$.*

**Proof** The claim simply follows from the fact that read() returns a value to the client after $3\delta$ time, regardless of the behaviour of the servers (see lines 3-6, Figure 5). $\square_{Lemma\ 3}$

**Theorem 1.** *If a client $c_i$ invokes an operation op on the register and does not crash, then op eventually terminates.*

**Proof** The claim simply follows from Lemma 2 and Lemma 3. $\square_{Theorem\ 1}$

*5.3. Convergence Proof*

In this section, we show the conditions for which our protocol $\mathcal{P}_{reg}$ converges to a legitimate configuration $l \in \mathcal{L}$ starting from an arbitrary configuration $\mathcal{C}$.

**Lemma 4.** *Let us consider a* write$(v)$ *operation* $op$ *starting at time* $t_B(op)$ *(with* $t_B(op) > t_{no\_tr}$*) from an arbitrary configuration* $\mathcal{C}$*, and labeled by the client writer with a timestamp* $ts \in \mathcal{Z}_m$*. Let* $n_\Delta$ *be the number of servers defined according to Table 1. At time* $t_E(op)$*, the system reaches a configuration* $\mathcal{C}'$ *such that, for every honest server* $s_i$ *at time* $t_E(op)$*, either* $V_{safe_i} = \emptyset$*, or* $\langle v, ts \rangle \in V_{safe_i}$*, or all timestamps known by* $s_i$ *are uniquely ordered and greater than* $ts$*.*

**Proof** At the beginning of the write$(v)$ operation $op$, the client executes lines 1-2 of Figure 4, incrementing its local timestamp to $ts$, and then propagating the pair $\langle v, ts \rangle$ to all servers at time $t_B(op)$. Every honest server delivers this pair by time $t_B(op) + \delta$, and executes lines 5-9 of Figure 4, inserting $v$ in $W_i$, and propagating it through an ECHO message. Let us note that during the period $[t_B(op), t_B(op) + \delta]$, the Byzantine adversary may move its agents at most once. Thus, the number of honest servers that execute correctly lines 5-9 of Figure 4 is at least $n_\Delta - 2f$. Let us note that the ECHO message takes at most $\delta$ time to be delivered to all honest servers. Thus, the ECHO message is delivered by time $t_B(op) + 2\delta$ by at least $n_\Delta - 2f$ honest servers, that in turn execute lines 11-14 and lines 19-27 of Figure 3. Considering that $n_\Delta - 2f > \#echo\_CUM$, it follows that $\langle v, ts \rangle$ is returned by select_pairs$(echo_vals_i)$ by at least $n_\Delta - 2f$ honest servers. Given an honest server $s_i$, while executing line 22 Figure 3 the following cases may happen:

1. $ts$ cannot be uniquely ordered with respect to values already stored in $V_{safe_i}$: in this case $s_i$ drops all values, and $V_{safe_i}$ is emptied due to the inconsistency. In this case, the claim follows.
2. $ts$ can be uniquely ordered with respect to values already stored in $V_{safe_i}$: in this case, two sub-cases may happen depending on how $ts$ is ordered by $s_i$. In particular, if $ts$ is smaller than any other timestamp stored by $s_i$, it is discarded. Now, if $ts$ is one of the most recent values, it is stored in $V_{safe_i}$. In both cases, the claim follows.

$\square_{Lemma\ 4}$

**Lemma 5.** *Let* $Z_m$ *be the set of timestamps used by Protocol* $\mathcal{P}_{reg}$*, with* $m \geq 13$*. Let* $n_\Delta$ *be the number of servers defined according to Table 1. If there exists a*

*sequence $S = op_1, op_2, \ldots, op_k$ of* write$(v)$ *operations (with $k \geq m-3$) that starts after transient failures cease to occur (i.e., $t_B(op_1) > t_{no\_tr}$), then $\mathcal{P}_{reg}$ converges to a configuration $\mathcal{C}'$ at time $t_E(op_k)$, where for every honest server $s_i$, the value $v$ written by $op_k$ is the state of $s_i$ (i.e., $v \in V_i$ or $v \in V_{safe_i}$ or $v \in W_i$).*

**Proof** Let us note that at the beginning of the write$(v)$ operation $op$, the client executes lines 1-4 of Figure 4, incrementing its local timestamp and then propagating the pair $\langle v, ts \rangle$ to all servers at time $t_B(op)$. Considering that *(i)* no new transient failure occurs, *(ii)* timestamps are selected sequentially in $\mathcal{Z}_m$, and *(iii)* during every maintenance, every server echoes at most 6 pairs $\langle v, ts \rangle$[11], we deduce that to avoid reusing timestamps too early, the maximum distance between any pair of timestamps to be ordered must be smaller than half the number of available timestamps, then it follows that $\mathcal{P}_{reg}$ requires $m \geq 13$. Due to Lemma 4, with $m \geq 13$, it follows that at most $m - 3$ values can be discarded by an honest server $s_i$ before reaching a configuration $\mathcal{C}'$ where $s_i$ stores $v \in V_{safe_i}$. Iterating the following reasoning for every honest server, the claim follows. $\square_{Lemma\ 5}$

**Corollary 1.** *Let $Z_m$ be the set of timestamps used by Protocol $\mathcal{P}_{reg}$, with $m \geq 13$. Let $n_\Delta$ be the number of servers defined according to Table 1. If there exists a sequence $S = op_1, op_2, \ldots, op_k$ of* write$(v)$ *operations (with $k \geq m - 3$) starts after transient failures cease to occur (i.e., $t_B(op_1) > t_{no\_tr}$), then $\mathcal{P}_{reg}$ converges to a semi-legitimate configuration $s \in \mathcal{SL}$ at time $t_E(op_k)$.*

**Proof** The claim follows from Lemma 5 considering that *(i)* at the beginning of $op_k$, there are $n_\Delta - f$ honest servers, *(ii)* during a write() operation, the Byzantine adversary may move the $f$ Byzantine agents at most once, and thus at the end of $op_k$, there are at least $n_\Delta - 2f$ correct servers that correctly executed $\mathcal{P}_{reg}$ and stored $v$ in $W_i$. $\square_{Corollary\ 1}$

**Lemma 6.** *Let $n_\Delta$ be the number of servers defined according to Table 1. If the system is in a semi-legitimate configuration $s \in \mathcal{SL}$ at time $t$, and no new* write() *operations are issued after $t$, then $\mathcal{P}_{reg}$ guarantees that any configuration $s'$ at time $t' > t$ is a semi-legitimate configuration $s' \in \mathcal{SL}$.*

[11] Let us note that for every server $s_i$, at the beginning of the maintenance() operation $|V_i| \leq 3$ as its content is replaced with the one of $V\_safe_i$ and, by construction of the algorithm, it can store at most 3 ordered pairs $\langle v, ts, \rangle$. In addition, also $|W_i| \leq 3$ as every entry has an associated timer of $2\delta$ and in every $2\delta$ time period there could be at most 3 write() operations that can require to store a pair $\langle v, ts, \rangle$ in $W_i$.

**Proof** If $s \in \mathcal{SL}$ is a semi-legitimate configuration at time $t$, it follows that the last written value $v$ is stored in some local variable by at least $n_\Delta - 2f$ servers at time $t$. In particular, looking at Figures 3-5, we have the following cases:

1. $v \in W_i$: this happens when a honest server $s_i$ delivers a WRITE message (Figure 4), and stores the new value (together with its timestamp and a $2\delta$ timer) in $W_i$;
2. $v \in V_{safe_i}$: this happens during the maintenance() operation when an honest server $s_i$ checks the values (and related timestamps) gathered during the echo phase and selects $v$ as it is occurring $\#echo\_cum$ times, and it is associated with one of the last 3 timestamps in the uniquely ordered sequence of collected timestamps (Figure 3);
3. $v \in V_i$: this happens during the maintenance() operation when an honest server $s_i$ copies values previously stored in $V_{safe_i}$, i.e., those selected during the previous maintenance() operation as they were occurring at least $\#echo\_cum$ times, and were associated with one of the last 3 timestamps in the uniquely ordered sequence of collected timestamps (Figure 3).

Let us note that those local variables are updated during every maintenance() by every honest server $s_i$. Thus, to prove our claim, we have to show that the maintenance *(i)* does not delete those values from correct servers, and *(ii)* propagates them to cured servers.

Let us consider the time $t$ where a semi-legitimate configuration $s \in \mathcal{SL}$ is reached. From Corollary 1, $t$ is the time when the last write() operation (in the sequence of at least 10) terminates. Let us note that at time $t$, at least $n_\Delta - 2f$ honest servers $s_i$ store $v$ in their $W_i$ local variable, and that $v$ remains stored in $W_i$ until time $t + 2\delta$, for any honest server.

Let us consider the first maintenance() operation $op\_m$ starting at time $t_B(op\_m) > t$. Let us note that $t < t_B(op\_m) \leq t + 2\delta$, as the maintenance is executed periodically every $\Delta$ time period (with $\Delta = \delta$, or $\Delta = 2\delta$).

At time $t_B(op\_m)$, every honest server $s_i$ executes lines 4-8 of Figure 3, and spreads trough an ECHO message the pairs $\langle v, ts \rangle$ that are stored locally in $V_i$ and $W_i$.

Every honest servers $s_j$ collects such pairs in its $echo\_vals_j$ variable, and by time $t_B(op\_m)$, $s_j$ has collected at least $n_\Delta - f$ ECHO messages coming from the other honest servers. Then, at least $n_\Delta - 3f$ such ECHO messages contain the pair $\langle v, ts \rangle$ that holds the last written value $v$ (as it was propagated inside $W_i$).

Considering that *(i)* during the current maintenance() operation, the Byzantine adversary may fake at most $2f$ ECHO messages with fake timestamps (those sent

by Byzantine servers and those sent by cured servers), *(ii)* those timestamps are not sufficiently many to be selected by the select_pairs() function, *(iii)* the timestamp associated to $v$ is now one of the three most recent in the uniquely ordered sequence of selected timestamps, and *(iv)* $\langle v, ts \rangle$ occurs more than $\#echo\_CUM$ times, it follows that any honest server $s_i$ selects $v$, and store it in its $V_{safe_i}$ variable.

Since there are at least $n_\Delta - f$ honest servers at time $t_B(op\_m) + \delta$, and that all of them store $v$ in their $V_{safe_i}$ variable, it follows that the system reaches a semi-legitimate configuration $s' \in \mathcal{SL}$ at time $t_B(op\_m) + \delta$.

Now, since $\langle v, ts \rangle \in V_{safe_i}$ for at least $n_\Delta - f$ honest servers, and is associated with one of the three most recent timestamps, and that no other write() operation is executed, we can iterate the previous reasoning for all the subsequent maintenance() operations, and the claim follows. $\square_{Lemma\ 6}$

**Lemma 7.** *Let $Z_m$ be the set of timestamps used by Protocol $\mathcal{P}_{reg}$, with $m \geq 13$. Let $n_\Delta$ be the number of servers defined according to Table 1. If there exists a sequence $S = op_1, op_2, \ldots, op_k$ of* write() *operations (with $k \geq m - 1$) that start after transient faults occur (i.e., $t_B(op_1) > t_{no\_tr}$), then Protocol $\mathcal{P}_{reg}$ converges to a legitimate configuration $l \in \mathcal{L}$ at time $t_E(op_k)$.*

**Proof** Due to Lemma 5, after a sequence of $m-3$ write() operations, every honest server no longer drops values coming from the writer, and the system reaches a semi-legitimate configuration $s \in \mathcal{SL}$. However, compromised pairs $\langle v, ts \rangle$ may still be stored in $V_{safe_i}$ for some server $s_i$, and thus its state is not yet valid. Let us note that such compromised timestamps are comparable with the one associated by the writer to the last written value. In particular, since we are considering a single writer and the system is synchronous, the last written value has a timestamp among the three most recent stored by at least $n_\Delta - 2f$ honest servers. After two more write() operations, this stale information is cleaned, and no fake timestamps remain in the state of honest processes. All honest processes (i.e., at most $n - 2f$) store the same pairs $\langle v, ts \rangle$, and the system reaches a legitimate configuration $l \in \mathcal{L}$. $\square_{Lemma\ 7}$

**Corollary 2.** *Let $Z_m$ be the set of timestamps used by Protocol $\mathcal{P}_{reg}$ with $m \geq 13$. Let $n_\Delta$ be the number of servers defined according to Table 1. Let $S = op_1, op_2, \ldots, op_k$ be the sequence of* write() *operations (with $k \geq m-1$) that starts after transient faults occur (i.e., $t_B(op_1) > t_{no\_tr}$), and let $l \in \mathcal{L}$ be the legitimate configuration at time $t_E(op_k)$. For every correct server $s_i$ at time $t_E(op_k)$, $V_{safe_i}$ stores three pairs $\langle v, ts \rangle$ with consecutive timestamps.*

**Proof** The claim follows from Lemma 5 observing that the client uses sequential timestamps. $\square_{Corollary\ 2}$

**Theorem 2.** *Let $Z_m$ be the set of timestamps used by the Protocol $\mathcal{P}_{reg}$ with $m \geq 13$. Let $S = op_1, op_2, \ldots, op_k$ be the sequence of* write() *operations (with $k \geq m - 1$) that starts after transient faults occur (i.e., $t_B(op_1) > t_{no\_tr}$). Let $n_\Delta$ be the number of servers defined according to Table 1. Protocol $\mathcal{P}_{reg}$ satisfies* ss − Convergence *(i.e., in every execution $e \in \mathcal{E}_\mathcal{P}$, there exists a legitimate configuration $l \in \mathcal{L}$).*

**Proof** The claim follows from Lemma 7. $\square_{Theorem\ 2}$

*5.4. Validity proof starting from a legitimate configuration $l \in \mathcal{L}$*

**Lemma 8.** *Let $op$ be a* maintenance() *operation starting at time $t$ from a legitimate configuration $l \in \mathcal{L}$. Let $n_\Delta$ be the number of servers defined according with Table 1. For every time $t' > t$, we have:*

$$|Co(t')| \geq n_\Delta - 2f.$$

**Proof** Let us note that if $l \in \mathcal{L}$ is a legitimate configuration at time $t$, then $|Co(t)| \geq n_\Delta - 2f$. To prove our claim, we need to show that the size of the set of cured processes does not increase when the mobile Byzantine agents move.

Without loss of generality, let us assume that $op$ is the first maintenance() operation executed, and let us prove our claim by induction.

Let us recall that the adversary moves the $f$ mobile Byzantine agents every $\Delta$ time units and that a maintenance() operation starts when the agents move. Thus, the next movements happen at time $t + \Delta$. Hence, to prove our claim, we just need to prove that by time $t + \Delta$, sufficiently many cured processes become correct. Since a cured process becomes correct when its state becomes valid, we have to show that by time $t + \Delta$, the state of a cured process is equal to that of a correct process.

At the beginning of the maintenance, every honest process executes lines 4-7 of Figure 3, and updates its state by cleaning $echo\_vals_i$, copying the content of $V_{safe_i}$ in $V_i$, and then cleaning $V_{safe_i}$. Let us note that at this point, the variables that may store a value that differs between correct and cured servers are reduced to $V_i$ and $W_i$. Thus, we have to show that before time $t + \Delta$, those variables are updated consistently.

Every honest server $s_i$ then executes line 8 of Figure 3, and broadcasts the content of $V_i$ and $W_i$. Those values are stored by every honest server in $echo\_vals_i$ by time $t + \delta$. Let us note that correct processes propagate the same $V_i$, and a potentially different set $W_i$ depending on the existence of a concurrent write() operation. Let us consider the two cases independently:

- **Case 1 - no concurrent** write() **exists.** In this case, $W_i$ becomes the same for every correct server (and could be empty, or may store a pair $\langle v, ts \rangle$ related to the last completed write() whose timer elapsed during the current maintenance()). Thus, every cured server $s_i$ has at least $n_\Delta - 2f$ copies of the same sets, and thus when updating $V_{safe_i}$, all of them consider the same input, terminate with the same values, and the claim follows.

- **Case 2 - there exists a** write() **operation executed concurrently with the maintenance.** Let us denote with $op\_m$ the current maintenance() operation, and with $op\_w$ the concurrent write() operation. Let us note that if $t_B(op\_w) > t_B(op\_m)$ (i.e., the write() operation starts after the current maintenance() sent out the ECHO messages), we are in a situation similar to the previous case. Thus, we now consider the case where $t_B(op\_m) - \delta \leq t_B(op\_w)$. In this case, the WRITE$(v, csn)$ message may be delivered by some server before time $t_B(op\_m)$, and by some other after. Thus, some honest servers may have already executed line 5, Figure 4 at time $t_B(op\_m)$, storing $\langle v, cns \rangle$ in $W_i$, and then propagating it with the ECHO message while some other do not. However, such servers only delay sending $\langle v, cns \rangle$ in an ECHO message, as they eventually do it executing line 6, Figure 4 upon delivery. It follows that by time $t_B(op\_m) + 2\delta$, at least $n_\Delta - 2f$ honest servers sent an ECHO message containing the pair $\langle v, cns \rangle$. Then, at time $t_B(op\_m) + 2\delta$, cured servers may execute lines 19-27, Figure 3, and get a valid state (hence becoming correct). Let us now show that this period is long enough to ensure that enough servers remain correct, and let us consider the following sub-cases:

  - $\Delta = 2\delta$. In this case, all servers that are cured at the beginning of the current maintenance() operation become correct before the next movement of the mobile Byzantine agents, and the claim follows.
  - $\Delta = \delta$. In this case, the Byzantine adversary may compromise $f$ additional processes before servers that were cured at the beginning of the current maintenance complete their cure to become correct. However, there also exists $f$ honest servers that started their cure at

time $t_B(op\_m) - \Delta$ (i.e., in the previous maintenance() operation) that become correct before the next movement of the Byzantine agents, and the claim follows.

$\square_{Lemma\ 8}$

**Corollary 3.** *Let $\gamma$ be the maximum amount of time a server remains in the cured state. When executed starting from a legitimate configuration $l \in \mathcal{L}$, Protocol $\mathcal{P}_{reg}$ implements a* maintenance() *operation that ensures $\gamma \leq 2\delta$.*

**Theorem 3.** *Let $n_\Delta$ be the number of servers defined according to Table 1. Any* read() *operation executed using Protocol $\mathcal{P}_{reg}$ starting from a legitimate configuration $l \in \mathcal{L}$ returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

**Proof** Let us consider a read() operation $op\_r$ at client $c_i$. Such operation waits $3\delta$ time before analyzing the $reply_i$ data structure and selecting the value that should be returned i.e., the value associated with the latest timestamp and occurring at least $\#reply\_CUM$ (i.e., $n_\Delta - 2f$) times in $reply_i$. The $reply_i$ variable is emptied at the beginning of the read() and it is updated with a pair $\langle v, ts \rangle$ each time that a REPLY message is delivered by the client. Such a message is sent by servers as an answer to a READ message and contains at most three pairs $\langle v, ts \rangle$ selected by the sending server among those stored locally (the $V_i$, $W_i$ and $V_{safe_i}$ local variables). In particular, those 3 pairs are those having timestamps that can be uniquely ordered, and that are the highest in the sequence. Thus, to prove our claim, we have to show that, starting from a legitimate configuration $l \in \mathcal{L}$, *(i)* a value $v$ is always selected by the select_value() function, and *(ii)* this value is a valid value according to the register execution history.

To prove the first point, we have to show that there are always "enough" honest servers that can provide a reply to the client. Due to Lemma 8, at time $t_B(op\_r)$ we have at least $n_\Delta - 2f$ correct servers, and $f$ that are executing a maintenance() operation (i.e., we have $n_\Delta - f$ honest servers). Considering that the system is synchronous (messages take at most $\delta$ time to be delivered) and that messages cannot be lost, it follows that at time $t_B(op_r) + \delta$ the READ message is delivered to every honest server, and the number of servers that remain honest while the READ message is spread, and that answer providing a suitable reply is at least $n_\Delta - 2f \geq \#reply_{CUM}$.

Now, we have just to prove that replies sent by honest servers allow the client to select a valid value. We need to consider two cases: $op_r$ is concurrent with some write() operations or not.

- **Case 1 - $op_r$ is not concurrent with any write() operation**. Let $op_w$ be the last write() operation such that $t_E(op_w) \leq t_B(op_r)$, and let $v$ be the last written value. Due to Lemma 8, at any time $t$ we have that $|Co(t)| \geq n_\Delta - 2f$. It follows that also between time $t_B(op_r)$ (i.e., when the read() operation starts) and time $t_B(op_r) + \delta$ (i.e., the latest time when a READ request can be received) there always exists at least $n_\Delta - 2f \geq \#reply_{CUM}$ correct servers. Considering that the read() starts from a legitimate configuration $l \in \mathcal{L}$, and that no write() is executed concurrently, it follows that every correct process stores the same valid state (that includes $v$), and replies to the client by sending $v$. Finally, considering that timestamps associated with values in legitimate configurations are evolving sequentially and that when the system reaches a legitimate configuration $l \in \mathcal{L}$, the highest timestamp is the one associated with the last completed write, it follows that $v$ is the value selected by the select_value() function at the client-side (it is occurring at least $\#reply_{CUM}$ times, and its associated timestamp is the latest one in the uniquely ordered sequence of provided timestamps). Thus, the claim follows in this case.

- **Case 2 - $op_r$ is concurrent with some write() operation**. Let $op_{W_0}$ be the last write() operation completed before $op_r$, and let $v$ be the value written by such operation. Let us consider the time interval $[t_B(op_r), t_B(op_r) + \delta]$ (i.e., the maximum time interval needed to ensure that the READ message is delivered to any honest process). Considering that a write() operation lasts $\delta$ time units, it follows that in this time interval, there can be at most two sequential write() operations, namely $op_{W_1}$ and $op_{W_2}$. Let us consider the worst case scenario where $op_{W_1}$ and $op_{W_2}$ are executed one after the other without any delay between them. Due to Lemma 8, at any time $t$ we have that $|Co(t)| \geq n_\Delta - 2f$. It follows that also between time $t_B(op_r)$ (i.e., when the read() operation starts) and time $t_B(op_r) + \delta$ (i.e., the latest time when a READ request can be received), there always exists at least $n_\Delta - 2f \geq \#reply_{CUM}$ correct servers. Let us note that when starting from a legitimate configuration $l \in \mathcal{L}$, timestamps are generated sequentially, and every correct server stores locally in its valid state only the last three that can be uniquely ordered. It follows that any correct server that is answering provides three pairs including $\langle v, ts \rangle$. Thus, $v$ is one of the valid values

that could be returned. Let us now show that no other values except those concurrently written could be returned. Let us note that the concurrently written values may be returned if the WRITE() and REPLY() messages are fast enough to be delivered before the end of the read() operation. To conclude, from Lemma 8, Byzantine and cured servers cannot force correct servers to store (and thus to reply) with a never written value (otherwise their state would not be valid). Only cured and Byzantine servers can reply with values that are not valid. Let us note that, if $\Delta = 2\delta$, then up to $4f$ servers are not correct. If $\Delta = \delta$, then up to $6f$ servers are not correct. In both cases, the threshold $\#reply_{CUM}$ is higher than the number of occurrences of invalid values that a reader can deliver. Mobile agents cannot force the reader to read another (or older) value, and even if an older value has $\#reply_{CUM}$ occurrences, the one with the highest timestamp is chosen, hence the claim follows.

$\square_{Theorem\ 3}$

**Theorem 4.** *Let $n$ be the number of servers emulating the register, and let $f$ be the number of Byzantine agents in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model. Let $\delta$ be the upper bound on the communication latency in the synchronous system. If* (i) $n \geq 6f + 1$ *for* $\Delta = 2\delta$*, and* (ii) $n \geq 8f + 1$ *for* $\Delta = \delta$*, then $\mathcal{P}_{reg}$ implements a Self-Stabilizing SWMR Regular Register in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model.*

**Proof** The proof simply follows from Theorem 1, Theorem 2, and Theorem 3.
$\square_{Theorem\ 4}$

### *5.5. $\mathcal{P}_{reg}$ optimality proof*

In this section, we reuse the results from Bonomi et al. [10] about the minimum number of replicas $n_{CUM_{LB}}$ to solve the Regular Register problem with Mobile Byzantine Failures in the $CUM$ model. In particular, the general formula is the following:

$$n_{CUM_{LB}} = [2(Max\tilde{B}([t, t + T_r]) + MaxCu(t)) - minC\tilde{B}C([t, t + T_r])]f$$

where $MaxCu(t)$ and $minC\tilde{B}C([t, t + T_r])]f$ are respectively the maximum number of servers that are in a cured state at time $t$, and the minimum number of servers that are first Byzantine or cured, and then correct (or vice versa) during the

| | $Max\tilde{B}([t, t+3\delta])$ | $MaxCu(t)$ | $MaxSil([t, t+3\delta])$ |
|---|---|---|---|
| $(\Delta S, CUM)$ | $\lceil \frac{3\delta}{\Delta} \rceil + 1$ | $\mathcal{R}(\lceil \frac{3\delta - \epsilon - \lceil \frac{3\delta}{\Delta} \rceil \Delta + \gamma}{\Delta} \rceil)$ | $\lceil \frac{\gamma + \delta - \epsilon - \lceil \frac{3\delta}{\Delta} \rceil \Delta}{\Delta} \rceil$ |

| | $minC\tilde{B}C([t, t+3\delta])$ |
|---|---|
| $(\Delta S, CUM)$ | $\lceil \frac{3\delta - \epsilon - \delta}{\Delta} \rceil + \mathcal{R}(\lceil \frac{3\delta}{\Delta} \rceil - \lceil \frac{\gamma + \delta}{\Delta} \rceil) + (MaxCu(t) - MaxSil([t, t+3\delta]))$ |

Table 2: Values for a general read() operation that terminates after $3\delta$ time [10].

time interval $[t, t + T_r]$. In Table 2, we state how to compute such values using the following Ramp function:

$$\mathcal{R}(x) = \begin{cases} x & \textit{if } x \geq 0 \\ 0 & \textit{if } x < 0 \end{cases}$$

For completeness, $MaxSil([t, t + T_r])$ is the maximum number of correct servers that do not participate in the operation computation in the time interval $[t, t + T_r]$, e.g., at the beginning of the time interval those servers were in a cured state, and by the end of the time interval they get into a correct state but there is not enough time left to send a reply to a client or server. An interested reader can find more details in the aforementioned work [10].

**Theorem 5.** *Protocol $\mathcal{P}_{reg}$ is optimal with respect to the number of replicas.*

**Proof** The proof follows considering that Theorem 4 proved that $\mathcal{P}_{reg}$ implements a Regular Register with the upper bounds provided in Table 1. Those bounds match the lower bounds proved in Theorem 1 of Bonomi et al. [10]. In particular this theorem states that no safe register problem can be solved if $n_{CUM_{LB}} = [2(Max\tilde{B}([t, t + T_r]) + MaxCu(t)) - minC\tilde{B}C([t, t + T_r])]f$, where $T_r$ is the upper bound on the read() operation duration. Each term can be computed applying Table 2 considering $\gamma = 2\delta$ (Corollary 3). In particular if $\Delta = \delta$, then $n_{CUM_{LB}} = [2(4 + 2) - 4]f = 8f$, while if $\Delta = 2\delta$ then $n_{CUM_{LB}} = [2(3 + 1) - 2]f = 6f$, concluding the proof. $\square_{Theorem\ 5}$

## 6. Concluding remarks

This paper proposed a self-stabilizing regular register emulation in a distributed system where both transient failures and mobile Byzantine failures can occur, and where messages and Byzantine agent movements are decoupled. The proposed protocol improves existing works on mobile Byzantine failures [9, 6, 10] being the

first self-stabilizing regular register implementation in a round-free synchronous communication model and to do so it uses bounded timestamps from the $\mathcal{Z}_{13}$ domain to guarantee finite and known stabilization time. In particular, the convergence time of our solution is upper bounded by the time it takes to complete twelve write() operations. Contrary to the $(\Delta S, CAM)$ model, the $(\Delta S, CUM)$ model required to design a longer maintenance() operation (that lasts $2\delta$ time). As a side effect, the read() operation completion time also increased and directly impacted the size of the bounded timestamp domain that characterizes the stabilization time. However, it is interesting to note that all these improvements have no additional cost with respect to the number of replicas that are necessary to tolerate $f$ mobile Byzantine processes and our solution is optimal with respect to established lower bounds.

An interesting future research direction is to study upper and lower bounds for *(i)* memory, and *(ii)* convergence time complexity of self-stabilizing register emulations tolerating mobile Byzantine faults. We are currently studying how to generalize our approach and study the required number of replicas also for different relationships of $\Delta$ and $\delta$. Another interesting future work is to study the implication of moving from a regular register to an atomic one. Regular and atomic registers have the same computational power but they differ in the complexity of the proposed solutions that require to implement a *write-back* mechanism to transform a regular register into an atomic one. This implies modifying the read() operation and to make it longer. In our settings, this may have an impact on the number of servers needed to ensure correctness and it may be interesting to study if it is possible to keep the same lower bounds obtained for the regular register case.

## Acknowledgments

This work has been partially supported by the INOCS Sapienza Ateneo 2017 Project (protocol number RM11715C816CE4CB). This work is supported in part by ANR Project ESTATE (ANR-16-CE25-0009-03).

## References

[1] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *J. Comput. Syst. Sci.*, 81(4):692–701, 2015.

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

[3] 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.

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

[5] François Bonnet, Xavier Défago, Thanh Dang Nguyen, and Maria 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.

[6] Silvia Bonomi, Antonella del Pozzo, and Maria Potop-Butucaru. Tight self-stabilizing mobile byzantine-tolerant atomic register. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pages 6:1–6:10, New York, NY, USA, 2016. ACM.

[7] Silvia Bonomi, Shlomi Dolev, Maria Potop-Butucaru, and Michel Raynal. Stabilizing server-based storage in byzantine asynchronous message-passing systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2015)*, Donostia San-Sebastian, Spain, July 2015. ACM Press.

[8] Silvia Bonomi, Maria Potop-Butucaru, and Sébastien Tixeuil. Byzantine tolerant storage. In *Proceedings of the International Conference on Parallel and Distributed Processing Systems (IEEE IPDPS 2015)*, Hyderabad, India, May 2015. IEEE Press.

[9] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal mobile byzantine fault tolerant distributed storage. In *Proceedings of the ACM International Conference on Principles of Distributed Computing (ACM PODC 2016)*, Chicago, USA, July 2016. ACM Press.

[10] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal storage under unsynchronized mobile byzantine faults. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*, pages 154–163, 2017.

[11] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Optimal self-stabilizing mobile byzantine-tolerant regular register with bounded timestamps. In *Stabilization, Safety,*

*and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, pages 398–403, 2018.

[12] 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.

[13] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distrib. Comput.*, 7(1):35–42, November 1993.

[14] Ariel Daliot and Danny Dolev. Self-stabilization of byzantine protocols. In *7th International Symposium on Self-Stabilizing Systems (SSS 2005)*, pages 48–67, 2005.

[15] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[16] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[17] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. On byzantine containment properties of the min+1 protocol. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, 2010.

[18] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[19] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Maximum metric spanning tree made byzantine tolerant. In *25th International Symposium on Distributed Computing (DISC 2011)*, 2011.

[20] 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.

[21] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

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

[23] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[24] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 311–325, London, UK, UK, 2002. Springer-Verlag.

[25] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Small byzantine quorum systems. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 374–383. IEEE, 2002.

[26] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology*, 1(1):1–13, 2007.

[27] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 22–29. IEEE Computer Society, 2002.

[28] 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.

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

[30] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *8th International Conference on Principles of Distributed Systems (OPODIS 2005)*, pages 283–298, 2005.

[31] 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.

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

[33] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel & Distributed Systems*, (4):452–465, 2009.