Optimal Self-Stabilizing Mobile Byzantine-Tolerant Regular Register with bounded timestamp

Silvia Bonomi, Antonella del Pozzo, Maria Potop-Butucaru, Sébastien Tixeuil

Optimal Self-Stabilizing Mobile Byzantine-Tolerant Regular Register with bounded timestamps

Silvia Bonomi*, Antonella Del Pozzo*,
Maria Potop-Butucaru†, Sébastien Tixeuil†

*Sapienza Università di Roma,Via Ariosto 25, 00185 Roma, Italy
{bonomi, delpozzo}@dis.uniroma1.it
†Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6,
F-75005 Paris, France
{maria.potop-butucaru, sebastien.tixeuil}@lip6.fr

October 23, 2018

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 more powerful adversary where (i) the message delay (i.e., δ) and the period of mobile Byzantine agents movement (i.e., Δ) are completely decoupled and (ii) servers are not aware of their state i.e., they do not know if they have been corrupted or not by a mobile Byzantine agent.

The proposed protocol tolerates *(i)* any 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.

1 Introduction

Byzantine fault tolerance is a fundamental building block in distributed system as Byzantine failures include all possible faults, attacks, virus infections and arbitrary behaviors that can occur in practice (even unforeseen ones). Such bad behaviors have been typically abstracted by assuming an upper bound f on the number of

Byzantine failures in the system. However, such 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 [25] making the set of faulty processes changing along time.

Mobile Byzantine Failure (MBF) models have been recently introduced to integrate 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 is able to 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 [11] 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, 16, 21, 22, 23] agents may move independently of protocol messages. In the case of unconstrained mobility, several variants were investigated [3, 5, 16, 21, 22, 23]: Reischuk [22] considers that malicious agents are stationary for a given period of time, Ostrovsky and Yung [21] introduce the notion of mobile viruses and define the adversary as an entity that can inject and distribute faults; finally, Garay [16], and more recently Banu *et al.* [3], and Sasaki *et al.* [23] or 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, 16, 23] lies in the knowledge that hosts have about their previous infection by a Byzantine agent. In Garay's model [16], a host is able to detect its own infection after the Byzantine agent left it. Sasaki *et al.* [23] 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 obviously fake information, *e.g.* fake id). Those subtle differences on the power of Byzantine agents turns 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.* [8]. In their solution to the regular register implementation, mobile Byzantine movements are synchronized, but the period of movement is independent to that of algorithm rounds.

Alternatively, *self-stabilization* [13, 14] is a versatile technique to recover from *any number of Byzantine participants*, provided that their malicious actions only spread a *finite* amount of *time*. In more details, starting from an arbitrary global

state (that may have been caused by Byzantine participants), a self-stabilizing protocol ensures that 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* [24], and *Byzantine quorum systems* [4, 18, 20, 19]. 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 simultaneously involved in the state maintenance protocol.

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 [12, 15, 26] due to the impossibility to distinguish a 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 [27, 26, 28] 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 coloring, link coloring, or dining philosophers). Time-local algorithms [29, 30, 31] 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 Byzantine tolerant SWMR or MWMR registers [1, 9, 7]. All these works do not consider the complex case of mobile Byzantine faults.

To the best of our knowledge, the problem of tolerating both *arbitrary transient faults and mobile Byzantine faults* has been considered recently 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 described in [3, 5, 16, 23].

Our Contribution. The main contribution of the paper 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 register emulation [8, 10] as we add the self-stabilization property. In more details, we present a regular register implementation that uses bounded timestamps from the $\mathcal{Z}_{13}$ domain and it is optimal with respect to the number of replicas needed to tolerate f mobile Byzantine agents. Finally, the convergence time of our solution is upper bounded by $T_{10write()}$, where

$T_{10write()}$ is the time needed to execute ten *complete* $write()$ 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 totally 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 characterised by its *internal state*, *i.e.*, by the set of all its local variables and the corresponding values. No agreement abstraction is assumed to be available at each process (*i.e.* processes are not able to use consensus or total order primitives to agree upon the current values). Moreover, we assume that each process has the same role in the distributed computation (*i.e.*, there is no special process acting as a coordinator).

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. 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 δ_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 δ (equal to $\delta_b \geq \delta_p$), and we assume δ is known to every process. Moreover, we assume that any process is provided with a physical clock, *i.e.*, non corruptible.

Computation model. Each process of the distributed system executes a distributed protocol $\mathcal{P}_{reg}$ that is composed by a set of distributed algorithms. Each algorithm in $\mathcal{P}_{reg}$ is represented by a finite state automaton and it is composed of a sequence

of computation and communication steps. A computation step is represented by the computation executed locally to each process while a communication step is represented by the sending and the delivering events of a message. Computation steps and communication steps are generally called *events*.
The computation is *round-free* i.e., the distributed protocol $\mathcal{P}_{reg}$ does not evolve in synchronous rounds and messages can be sent, according to the protocol, at any point in time.
Given a process p_i and the protocol $\mathcal{P}_{reg}$, we say that p_i *is correctly 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 1 (Valid State at time t) *Let p_i be a process and let $state_{p_i}$ be the state of p_i at some time t. $state_{p_i}$ is said to be valid at time t if it is equal to the state of some fictional process $\bar{p}$ correctly executing $\mathcal{P}_{reg}$ in the interval $[t_0, t]$.*

Failure Model. An arbitrary number of clients may crash while 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 a server to another. We assume that, at any time t, at most f mobile Byzantine agents are in system.

In this work we consider the Δ-synchronized and Cured Unaware Model, i.e. $(\Delta S, CUM)$ MBF model, introduced in [8] that is suited for round-free computations[1]. More in details, $(\Delta S, CUM)$ can be specified as follows: the external adversary moves all the f mobile Byzantine agents at the same time t and movements happen periodically (i.e., movements happen at time $t_0 + \Delta$, $t_0 + 2\Delta$, . . . , $t_0 + i\Delta$, with $i \in \mathbb{N}$) and at any time t, no process is aware about its failure state (i.e., processes do not know if and when they have been affected from a Byzantine agent) but it is aware about the time at which mobile Byzantine agents move.

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

Definition 2 (Correct process at time t) *A process is said to be* correct at time t *if (i) it is correctly executing its protocol $\mathcal{P}$ and (ii) its state is a valid state at time t. We will denote as $Co(t)$ the set of correct processes at time t while, given a time interval $[t, t']$, we will denote as $Co([t, t'])$ the set of all the processes that are correct during the whole interval $[t, t']$ (i.e., $Co([t, t']) = \bigcap_{\tau \in [t,t']} Co(\tau)$).*

[1]The $(\Delta S, CUM)$ model abstracts distributed systems subjected to proactive rejuvenation [25] where processes have no self-diagnosis capability.

Definition 3 (Faulty process at time t) *A process is said to be* faulty at time t *if it is controlled by a mobile Byzantine agent and it is not executing correctly its protocol $\mathcal{P}$ (i.e., it is behaving arbitrarily). We will denote as $B(t)$ the set of faulty processes at time t while, given a time interval $[t, t']$, we will denote as $B([t, t'])$ the set of all the processes that are faulty during the whole interval $[t, t']$ (i.e., $B([t, t']) = \bigcap_{\tau \in [t,t']} B(\tau)$).*

Definition 4 (Cured process at time t) *A process is said to be* cured at time t *if (i) it is correctly executing its protocol $\mathcal{P}$ and (ii) its state is not a valid state at time t. We will denote as $Cu(t)$ the set of cured processes at time t while, given a time interval $[t, t']$, we will denote as $Cu([t, t'])$ the set of all the processes that are cured during the whole interval $[t, t']$ (i.e., $Cu([t, t']) = \bigcap_{\tau \in [t,t']} Cu(\tau)$).*

As in the case of round-based MBF models [3, 5, 11, 16, 23], we assume that any process has access to a tamper-proof memory storing the correct protocol code.

Let us stress that even though at any time t, at most f servers can be controlled by Byzantine agents, during the system life time, all servers may be affected by a Byzantine agent (*i.e.*, none of the servers is guaranteed to be correct forever).

Processes may also suffer from *transient* failures, *i.e.*, local variables of any process (clients and servers) can be arbitrarily modified [14]. It is nevertheless assumed that transient failures are quiescent, *i.e.*, there exists a time τ_{no_tr} (which is unknown to the processes) after which no new transient failures happens.

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). We also assume that initially the register stores a default value $\perp$ written by a fictional write($\perp$) operation happening instantaneously at time t_0. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [17]: *safe*, *regular* and *atomic*.

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

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

- ss − Termination: Any operation invoked on the register by a non-crashed process eventually terminates.
- ss − Validity: There exists a time t_{stab} such that each read() operation invoked at time $t > t_{stab}$ returns the last value written before its invocation, or a value written by a write() operation concurrent with it.

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 $(\Delta S, CUM)$ Mobile Byzantine Failure model. Such an algorithm copes with the $(\Delta S, CUM)$ model following the same approach as in [10], which is improved with the bounded timestamps in order to design a self-stabilizing algorithm.

We implemented read() and write() operations following the classical quorum-based approach (like in the ABD protocol [2]) and exploiting the synchrony of the system to guarantee their termination. Informally, when the writer client wants to write, it simply propagates the new value to servers that update the value of the register while, when a reader client wants to read, it asks for the current value of the register and waits for replies: after 3δ time "enough"[2] replies have been received and a value is selected and returned (the reason why a read() operation lasts for 3δ is explained in the following).

In order to do that, the maintenance() operation must guarantee that there always exists a sufficient number of servers storing a valid value for the register.

[2]The exact number of replies is provided in Table 1 depending on the relationship between Δ and δ.

Thus, its aim is threefold: (i) ensuring that cured servers get a valid value at the end of maintenance(), (ii) possible concurrent written values are always taken into account by cured servers running maintenance() and (iii) correct servers do not overwrite their correct 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 (when Byzantine agents move at every $T_i = t_0 + i\Delta$, with $i \in \mathbb{N}$) executes the maintenance() operation.

The basic idea is to keep separated information that can be trusted (e.g., values received by the writer client or values sent from "enough" processes) from those that are untrusted (e.g., values stored locally that can be compromised) and to decide the current state accordingly.

To this aim, maintenance() makes use of three fundamental set variables: (i) V_i stores the knowledge of s_i at the beginning of each maintenance() operation and contains the last three values of the register and the corresponding sequence numbers (untrusted information), (ii) V_{safe} (emptied at the beginning of each the maintenance() operation) is used to collect values selected among those sent through echoes by other servers (trusted information due to the presence of "enough" correct servers) and (iii) W_i contains values and the corresponding timestamps concurrently received by the writer (untrusted information as it can be potentially compromised by the Byzantine agent before it leaves the server).

As an example, consider the execution of the i-th maintenance() operation starting at time $t_0 + i\Delta$ shown in Figure 1 for the two servers s_0 and s_1 that are respectively correct and cured.

When maintenance() starts, every server s_i echoes the relevant information stored locally (i.e., list of pending read() operations and the sets V_i and W_i). Such information are then collected by any server s_j and can be used (based on the number of occurrence of each pair $\langle$value, sequence number$\rangle$) to update the set V_{safe}. Let us note that, due to the synchrony of the system, after δ time units (i.e., at time $t_0 + i\Delta + \delta$), s_i collected at least all the values sent by every correct and cured server and it is able to decide and update its local variables. Thus, it selects the values occurring "enough times" (see footnote 2) from echoes, updates V_{safe} and empties V_i.

In Figure 1, it is possible to see that, at time $t_0 + i\Delta + \delta$, s_0 basically does not update its information while s_1 is able to update its V_{safe} set consistently with s_0 using the values gathered through echoes.

However, the maintenance() operation is not yet terminated as it could happen that a write() operation is running concurrently and the concurrently written value may not yet be in V_i and in V_{safe}. In order to manage this case, every time that a value is written, it is also relayed to all servers. In addition, in order to avoid to overwrite values just written with those selected from the maintenance() operation,

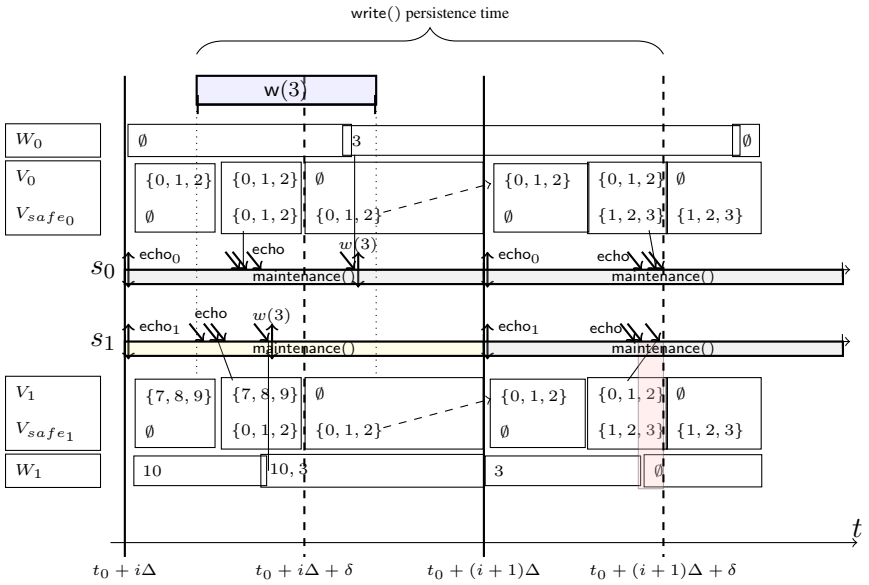

Figure 1: Example of a partial run for a correct server s_0 and a cured server s_1 with $\Delta = 2\delta$. For the sake of simplicity we report only timestamps instead of the pair $\langle value, timestamp \rangle$.

concurrently written values are temporarily stored in W_i with an associated timer (i.e., like a time-to-leave) set to 2δ.

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 will eventually appear in every non-faulty V_{safe} set (e.g., the value 3 in Figure 1). At the same time, the 2δ period is not long enough to allow mobile Byzantine agents to leverage the propagation of corrupted values to force reader clients to return a bad value (e.g., the value 10 left by the mobile Byzantine agent in W_1 at the beginning of the i-th maintenance() operation). We call the time necessary for a value to be present in V_{safe} as the write persistence time.

Note that, depending on the relationship between Δ and δ, it may happen that a maintenance() operation is triggered while the previous one is not yet terminated. This is not an issue as 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 and it prevents values to be lost having a time-to-live of 2δ which is enough to propagate it.

Finally, concurrently with the maintenance() and write() operations, servers may need to answer also to clients that are currently reading. In order to preserve the validity of read() operations and in order to cope with possible corrupted values stored by s_i just before the mobile Byzantine agent left, s_i replies with all the values it is storing (i.e., providing V_i, V_{safe} and W_i). Note that, given the update mechanism of local variables (designed to keep separated trusted information from untrusted ones), there could be a fraction of time where the last written value is removed from W_i (as its timer is expired) and it is not yet inserted in V_i and in V_{safe} (as the corresponding propagation message is still traveling - cfr. the red zone in Figure 1). To cope with this issue, the read() operation lasts 3δ time i.e., an extra waiting period is added for the collection of replies to guarantee that values are not lost.

In order to stabilise in a finite and known period 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 in order to guarantee that a new value possibly received by the writer is not overwritten by the maintenance() operation. In the following we will explain why using bounded timestamps guarantees a finite and known stabilisation period.

Let us note that, in order to stabilise, at least one write() operation must be executed after time τ_{no_tr}. However, due to the fact that this operation is the first one after τ_{no_tr}, if the domain of timestamps is unbounded (e.g., the domain of natural numbers $\mathbb{N}$ as in [6, 8, 10]), it could happen that the timestamp used by the writer

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

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

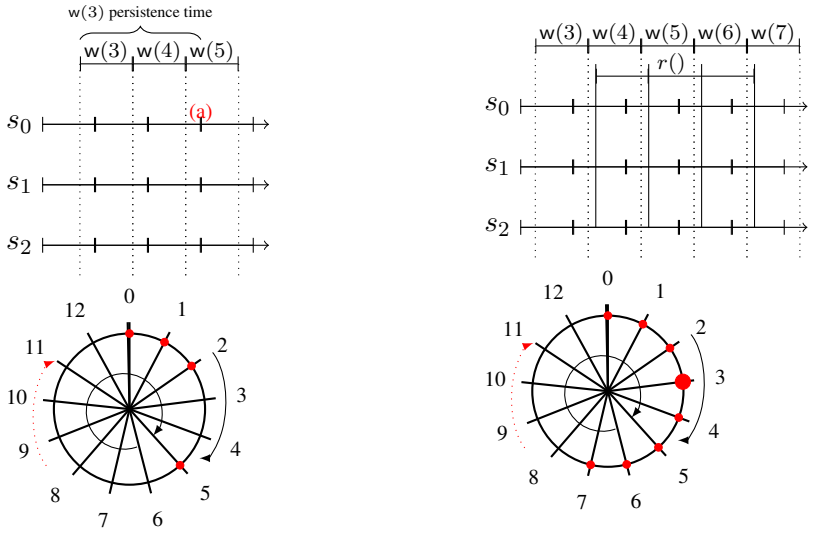

Figure 2: Runs for $\Delta = \delta$. The small vertical lines are the points where the maintenance() operations begin. For simplicity we represent values with their timestamp and we consider only correct servers s_i that store $V_i = \{0, 1, 2\}$.

is way much smaller than those stored locally by servers. This means that such an operation will be ignored and the same will happen until the writer timestamp will reach those stored by servers making the stabilisation period unknown.

We use timestamps in the domain $\mathcal{Z}_m$, with $m = 13$. Each written value is represented as $\langle val, sn \rangle$ where val is the content and sn the corresponding sequence number, $sn \in Z_m = \{0, 1, \ldots, m-1\}$. Let us define two operations on such values: addition: $+_m : Z_m \times Z_m \to Z_m, a +_m b = (a+b) \pmod m$; and subtraction: $-_m : Z_m \times Z_m \to Z_m, a -_m b = a +_m (-b)$. Note that $(-b)$ is the opposite of b. That is, the number that added to b gives 0 as result, *i.e.*, $b +_m (-b) = 0$.

Two scenarios are depicted in Figure 2 to characterize how many different values clients and servers may have to manage (and thus uniquely order) at the same time. We consider a sequence of write() operations and then what happens if a read() operation is concurrent with a sequence of write() operations. In the first case, just before the time instant marked as (a) s_0 could be ready to store values

$0, 1, 2$ and 5 that need to be ordered. In the meantime values 3 and 4 are still echoed. In any case the timestamps range that a server can manage at the same time is from 0 to 5, more in general 6 subsequent timestamps. 3 values are stored in V_i and 3 values come from the subsequent write() operations. At time (a), in Figure 2, 3 takes the place of 0, which is discharged. Let us consider the most distant values, 0 and 5. There are two ways to order them, either 0 precedes 5 or 5 precedes 0. But the second one is impossible since in that case there could be 7 timestamps around at the same time. In the second scenario, concurrently to a sequence of write() operations there is a read() operation. In this case we have to consider all values that could be returned to the client. In this case, values from 0 to 7 (thus at most at distance 7) and we notice that the last written value, 3, is always returned. Thus a client may have to order the following values $0, 3, 7$. There are three possibilities: (i) $0, 3, 7$, (ii) $3, 7, 0$, or (iii) $7, 0, 3$. In cases (ii) and (iii) we have $0 -_{13} 3 = 10$ and $3 -_{13} 7 = 9$ respectively, both of them greater than 7. Thus the only possible order the case (i).

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

Local variables at client c_i. Each client c_i maintains two sets $reply_i$ that is used during the read() operation to collect the three tuples $\langle j, \langle v, sn \rangle \rangle$ sent back from servers. Additionally, if c_i is the writer, it maintains a local sequence number csn that is incremented, respect to the $\mathcal{Z}_{13}$ arithmetic, each time it invokes a write() operation, which is timestamped with such sequence number.

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

- V_i: a set containing 3 tuples $\langle v, sn \rangle$, where v is a value and sn the corresponding sequence number.
- V_{safe_i}: this set has the same characteristic as V_i, and is populated by the function insert$(V_{safe_i}, \langle v_k, sn_k \rangle)$.
- W_i: a set where s_i stores values coming directly from the writer, associating to it a timer, $\langle v, sn, timer \rangle$. When the timer expires, the associated value is deleted.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through ECHO messages. The former set stores tuple $\langle v, sn \rangle_j$ whilst the latter set contains identifiers of concurrently reading clients in order to notify cured servers and expedite termination of read() operations.
- $pending\_read_i$: set variable used to collect identifiers of the clients that are currently reading. Notice, for simplicity we do not explicitly manage

the values discharge from such set since it has no impact on the protocol correctness.

In order to simplify the code of the algorithm, let us 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 in $echo\_vals_i$ of such tuple (ignoring the Timer value, if present).
- insert($V_{safe_i}, \langle v_k, sn_k \rangle$): this function inserts $\langle v_k, sn_k \rangle$ in V_{safe} according with the incremental order and if there are more than 3 values then the oldest one is discarded. In case it is not possible to establish an unique order among the elements in the set then V_{safe_i} is reset (this may happen due to transient failures).
- select_value($reply_i$): this function returns the newest pair $\langle v, sn \rangle$ occurring at least $\#reply_{CUM}$ times in $reply_i$ (ignoring the Timer value, if present).
- checkOrderAndTrunc(V_{safe_i}): this function checks if it is possible to uniquely order the elements in V_{safe_i} with respect to the timestamps. If yes, the 3 newest element are kept, the others are discharged. If it is not possible to uniquely establish an order for each pair of elements then all the elements are discharged.
- checkOrder(V_{safe_i}): this function checks if it is possible to establish an unique order for each couple of elements in V_{safe_i}. If not, V_{safe_i} is emptied.
- conCut(V_i, V_{safe_i}, W_i): this function takes as input three 3 dimension ordered sets and returns another 3 dimension ordered set. The returned set is composed by the concatenation of $V_{safe_i} \circ V_i \circ W_i$, without duplicates, truncated after the first 3 newest values (with respect to the timestamp). e.g., $d = 3$, $V_i = \{\langle v_a, 1 \rangle, \langle v_b, 2 \rangle, \langle v_c, 3 \rangle\}$ 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 is it not possible to establish and order in one of those sets because of transient failures then the result is $\perp$.
- checkTimer(W_i): this function removes from W_i all the values whose associated timer is 0 or strictly greater than 2δ.

The maintenance() **operation.** Such operation is executed by servers periodically at any time $T_i = t_0 + i\Delta$. Each server first stores the content of V_{safe_i} in V_i and

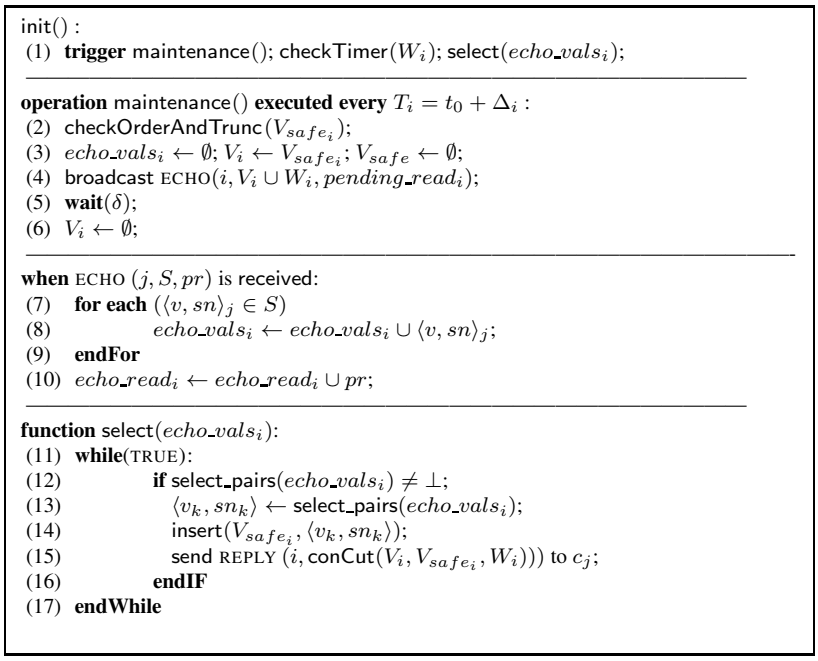

Figure 3: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CUM)$ model with bounded timestamp.

all V_{safe_i} and $echo\_vals_i$ sets are reset. Each server broadcasts an ECHO message with the content of V_i, W_i and the $pending\_read_i$ set. When there is a value in the $echo\_vals_i$ set that occurs at least $\#echo_{CUM}$ times, s_i tries to update V_{safe_i} set by invoking insert on the value returned by the select_pairs($echo\_vals_i$) function. To conclude, after δ time since the beginning of the operation, the V_i set is reset. Informally speaking, during the maintenance() operation V_{safe_i} is filled with safe values, then the content in V_i is not longer necessary. Notice that the content of W_i is continuously monitored so that expired values are removed.

The write() **operation**. When the write() operation is invoked, the writer increments $csn \leftarrow csn +_m 1$, sends WRITE($\langle v, csn\rangle$) to all servers and finally returns after δ time. For each server s_i, two cases may occurs, s_i delivers WRITE($\langle v, csn\rangle$) message when it is not affected by a Byzantine agent or when it is affected by a Byzantine agent. In the first case s_i stores v in W_i and forwards it to every server sending the ECHO($i, \langle v, csn\rangle, pending\_read_i$) message. Such value is further echoed at the beginning of each next maintenance() operation as long as $\langle v, csn\rangle$ is in W_i or V_i, this is true for $\#echo_{CUM}$ correct servers. When $\langle v, csn\rangle$

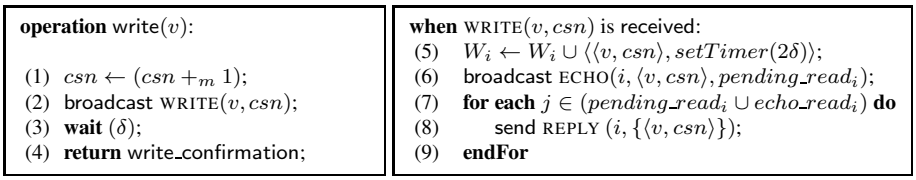

Figure 4: $\mathcal{A}_W$ algorithms, server side and client side respectively, implementing the write(v) operation in the $(\Delta S, CUM)$ model with bounded timestamp.

occurs $\#echo_{CUM}$ times in $echo\_vals_i$ then s_i tries to update V_{safe_i} set by invoking insert on the value returned by the select_pairs$(echo_vals_i)$ function.

The read() **operation**. At client side, when the read() operation is invoked at client c_i, it empties the $reply_i$ set and sends to all servers the READ(i) message. Then c_i waits 3δ time, while the $reply_i$ set is populated with servers replies, and from such set it picks the newest value occurring $\#echo_{CUM}$ times 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. At server side when s_j delivers the READ(i) message, client c_i identifier is stored in the $pending\_read_j$ set. Such set is part of the content of ECHO message in every maintenance() operation, which populates the $echo\_read_j$ 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 such function is sent back to c_i in the REPLY message. Finally a REPLY message containing just one value is sent when a new value is added in W_i and there are clients in the $pending\_read_j \cup echo\_read_j$ set. When the READ_ACK(i) message is delivered from c_i then its identifier is removed from the $pending\_read_j$ and $echo\_read_j$ sets.

4.1 Correctness proofs

In the following we prove that the protocol defined in Section 4 is correct.

Definition 5 (Faulty servers in the interval I) *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 6 ($Max\tilde{B}(t, t+T)$) *Let $[t, t+T]$ be a time interval. The cardinality of $\tilde{B}(t, t+T)$ is maximum if for any t', $t' > 0$, is it true that $|\tilde{B}(t, t+T)| \geq |\tilde{B}(t', t'+T)|$. Let $Max\tilde{B}(t, t+T)$ be such cardinality.*

Lemma 1 *If $\Delta > 0$ and $T \geq \delta$ then $Max\tilde{B}(t, t+T) = (\lceil \frac{T}{\Delta} \rceil + 1)f$.*

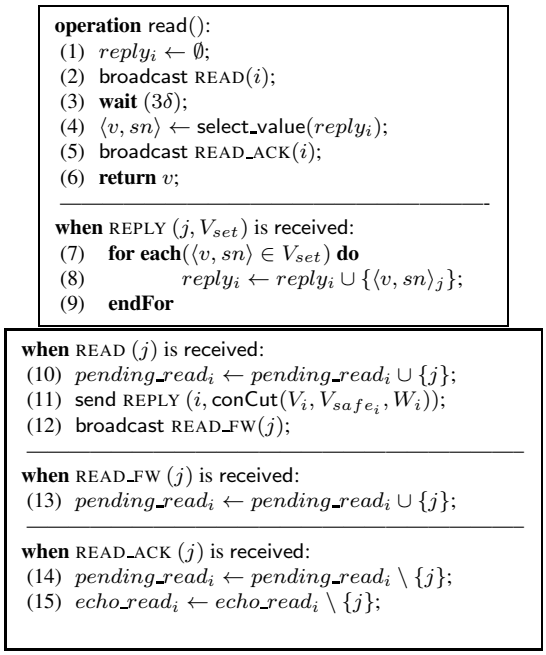

Figure 5: $\mathcal{A}_R$ algorithms, client side and server side respectively, implementing the read() operation in the $(\Delta S, CUM)$ model with bounded timestamp.

Proof For simplicity let us consider a single agent ma_1, then we extend the reasoning to all the f agents. In the $[t, t+T]$ time interval, with $T \geq \delta$, ma_1 can affect a different server each Δ time. It follows that the number of times it may "jump" from a server to another is $\frac{T}{\Delta}$. Thus the affected servers are at most $\lceil \frac{T}{\Delta} \rceil$ plus the server on which ma_1 is at t. Finally, extending the reasoning to f agents, $Max\tilde{B}(t, t+T) = (\lceil \frac{T}{\Delta} \rceil + 1)f$, concluding the proof. $\square_{Lemma\ 1}$

In the following we first characterize the correct system behavior, i.e., when the protocol is correctly executed after τ_{stab} (the end of the transient failure and system stabilization). In doing this we assume that it is always possible to establish the correct order among the values that are collected by clients and servers. After we prove that it is always possible to establish an order among those values and finally, we prove that the protocol is self-stabilizing after a finite number of write() operations.
Concerning the protocol correctness, the termination property is guaranteed by the way the code is designed, after a fixed period of time all operations terminate. The validity property is proved with the following steps:

1. maintenance() operation works (i.e., at the end of the operation $n-f$ servers store valid values). In particular, for a given value v stored by $\#echo_{CUM}$ correct servers at the beginning of the maintenance() operation, there are $n - f$ servers that store v after δ time since the beginning of the operation;

2. given a write() operation that writes v at time t and terminates at time $t + \delta$, there is a time $t' < t + 3\delta$ after which $\#reply_{CUM}$ correct servers store v;

3. at the next maintenance() operation after t' there are $\#reply_{CUM} - f = \#echo_{CUM}$ correct servers that store v, for step (1) this value is maintained in the register;

4. the validity property follows considering that the read() operation is long enough to include the t' of the last written value in such a way that servers have enough time to reply and after t' this value is maintained in the register, step (3), as long as there are no others write() operations. To such purpose we show that V_i is big enough to do not be full filled with new values before that the last written value is returned.

Correctness proofs considering $t > t_{stab}$.

In the following we prove the correctness of the protocol when there are no transient failures, the system is stable and thus timestamp are not bounded, thus it is always possible to uniquely order all values.

Lemma 2 *If a client c_i invokes a* write(v) *operation at time t then this operation terminates at time $t + \delta$.*

Proof The claim simply follows by considering that a write_confirmation event is returned to the writer client c_i after δ time, independently of the servers behavior (see lines 3-4, Figure 4). $\square_{Lemma\ 2}$

Lemma 3 *If a client c_i invokes a* read() *operation at time t then this operation terminates at time $t + 3\delta$.*

Proof The claim simply follows by considering that a read() returns a value to the client after 3δ time, independently of the behavior of the servers (see lines 3-6, Figure 5). $\square_{Lemma\ 3}$

Theorem 1 *Any operation invoked on the register eventually terminates.*

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

Lemma 4 (Step 1.) *Let v be a value stored at $\#echo_{CUM}$ correct servers $s_j \in Co(T_i)$, $v \in V_j \forall s_j \in Co(T_i)$. Then $\forall s_c \in Cu(T_i)$ at $T_i + \delta$ (i.e., at the end of the* maintenance()*) v is returned by the function* select_pairs$(echo_vals_i)$.

Proof By hypothesis at T_i there are $\#echo_{CUM}$ correct servers s_j storing the same v and running the code in Figure 3. In particular each server broadcasts a ECHO() message with attached the content of V_j which contains v (line 4). Messages sent by $\#echo_{CUM}$ correct servers are delivered by s_c and stored in $echo\_vals_c$. The communication channels are synchronous, thus by time $Ti+\delta$ function select_pairs$(echo_vals_c)$ returns v. $\square_{Lemma\ 4}$

Lemma 5 *Let s_i be a correct server running the* maintenance() *operation at time T_i, then if v is returned by the function* select_pairs$(echo_vals_i)$ *there exist a* write() *operation that wrote such value.*

Proof Let us suppose that select_pairs$(echo_vals_i)$ returns v' and there no exist a write(v'). This means that s_i collects in $echo\_vals_i$ at least $\#echo_{CUM}$ occurrences of v' coming from cured and Byzantine servers. Let us consider a cured server s_c running the maintenance() operation at time T_c. At the beginning of the maintenance() operation s_c broadcasts values contained in V_i and W_i (Figure 3, line 4). V_i is reset at each operation with the content of V_{safe_i} which is reset at

each maintenance() operation (line 3). It follows that s_c broadcasts non valid values contained in V_i only during the maintenance() operation run at T_c. Contrarily, values in W_i, depending on k, are broadcast only at T_c or also at T_{c+1}. Let us consider two cases: $k = 2$ and $k = 3$.
case $k = 2$**:** In this case since $\Delta = 2\delta$ and the maximum value of the timer associated to a value is 2δ, thus each cured server s_c broadcasts a non valid value contained in W_i only during the first maintenance() operation. So, during each maintenance() operation there are f Byzantine servers and f cured servers, those are not enough to send $\#echo_{CUM} = 2f + 1$ occurrences of v'. For Lemma 4 this is the necessary condition to return v' invoking select_pairs($echo\_vals_i$), leading to a contradiction.
case $k = 3$**:** $\Delta = \delta$ and the maximum value of the timer associated to a value is 2δ, thus each cured server s_c broadcasts a non valid value contained in W_i during the two subsequent maintenance() operations. Summing up, during each maintenance() operation at time T_i there are f Byzantine servers, f cured servers and f servers that were cured during the previous operation. Those servers are not enough to send $\#echo_{CUM} = 3f + 1$ occurrences of v', for Lemma 4 this is the necessary condition to return v' invoking select_pairs($echo\_vals_i$), leading to a contradiction and concluding the proof. $\square_{Lemma\ 5}$

From the reasoning used in this Lemma, the following Corollary follow.

Corollary 1 *Let s_i be a non faulty process and v a value in W_i. Such value is in W_i during at most $k - 1$ sequential* maintenance() *operations.*

Finally, considering that servers reply during a read() operation with values in W_i, V_i and V_{safe_i}. V_{safe_i} is safe by definition, V_i is reset after the first maintenance() operation then it follows that servers can be in a cured state for 2δ time, the time that never written values can be present in W_i.

Corollary 2 *Protocol $\mathcal{P}$ implements a* maintenance() *operation that implies $\gamma \leq 2\delta$.*

Lemma 6 *Let T_c be the time at which s_c becomes cured. Each cured server s_c can reply back with incorrect message to a* READ() *message during a period of 2δ time.*

Proof The proof directly follows considering that the content of a REPLY() message comes from the V_c, V_{safe_c} and W_i sets. The first one is filled with the content of V_{safe_c} at the beginning of each maintenance() operation and after δ time is reset (cf. Figure 3, lines 5-6). The second one is emptied at the beginning of each

maintenance() operation and the third one keeps its value during k maintenance() operations (cf. Corollary 1). Thus by time $T_c + 2\delta$ s_c cleans all the values that could come from a mobile agent. $\square_{Lemma\ 6}$

Lemma 7 (Step 2.) *Let op_W be a* write(v) *operation invoked by a client c_k at time $t_B(op_W) = t$ then at time $t + 3\delta$ there are at least $n - 2f \geq \#reply_{CUM}$ correct servers such that $v \in V_{safe_i}$ and is returned by the function* concCut().

Proof Due to the communication channel synchrony, the WRITE messages from c_k are delivered by servers within the time interval $[t, t + \delta]$; any non faulty server s_j executes the correct algorithm code. When s_j delivers a WRITE message it stores the value in W_j and sets the associated timer to 2δ (line 5, Figure 4).
For Lemma 1 in the $[t, t+\delta]$ time interval there are maximum $2f$ Byzantine servers, thus at $t + \delta$ v is stored in W_j at $n - 2f \geq \#echo_{CUM}$ correct servers s_j. All those servers broadcast v by time $t + \delta$, so by time $t + 2\delta$ there are $\#echo_{CUM}$ occurrences of v in $echo\_vals_i$, each server s_i stores v in V_{safe_i}. If a Byzantine agent movement happens before $t+2\delta$, i.e., $T_i \in [t+\delta, t+2\delta]$ then at time T_i, due to Byzantine agents movements, there are $n - 3f \geq \#echo_{CUM}$ correct servers that run the maintenance() operation and broadcast v. Thus at time $t + 3\delta$, for Lemma 4, all correct servers are storing $v \in V_{safe_i}$ and by construction v is returned by the function conCut(). We conclude the proof by considering that there are at least $n - 2f \geq \#reply_{CUM}$. $\square_{Lemma\ 7}$

For simplicity, from now on, given a write() operation op_W we call $t_B(op_W) + 3\delta = t_{wP}$ the **persistence time** of op_W, the time at which there are at least $\#reply_{CUM}$ servers s_i storing the value written by op_W in V_{safe_i}.

Lemma 8 (Step 3/1.) *Let op_W be a* write() *operation and let v be the written value. If there are no other* write() *operations, the value written by op_W is stored by all correct servers forever (i.e., v is returned invoking the* conCut() *function).*

Proof From Lemma 7 at time t_{wP} there are at least $n - 2f \geq \#reply_{CUM}$ correct servers s_j that have v in V_{safe_i}. At the next Byzantine agents movement there are $n - 2f - f \geq \#echo_{CUM}$ correct server storing v in V_{safe_i}, which is moved to V_i and broadcast during the maintenance() operation. For Lemma 4, after δ time, all non Byzantine servers are storing v in V_{safe_i}. At the next Byzantine agents movement there are f less correct servers that store v in V_{safe_i}, but those servers are still more than $\#echo_{CUM}$. It follows that cyclically before each agent movement there are f servers more that store v thanks to the maintenance() and f servers that lose v because faulty, but this set of non faulty servers is enough to

successfully run the maintenance() operation (cf. Lemma 4)). By hypothesis there are no more write() operations, then v is never overwritten and all correct servers store v forever.

$\square_{Lemma\ 8}$

Lemma 9 (Step 3/2.) *Let $op_{W_0}, op_{W_1}, \ldots, op_{W_{k-1}}, op_{W_k}, op_{W_{k+1}}, \ldots$ be the sequence of* write() *operation issued on the regular register. Let us consider a generic op_{W_k}, let v be the written value by such operation and let t_{wP} be its persistence time. Then v is in the register (there are $\#reply_{CUM}$ correct servers storing it) up to time at least $t_B W_{k+3}$.*

Proof The proof simply follows considering that:

- for Lemma 8 if there are no more write() operation then v, after t_{wP}, is in the register forever;
- any new written value eventually is stored in ordered set V_{safe}, whose dimension is 3;
- write() operation occur sequentially.

It follows that after 3 write() operations, $op_{W_{k+1}}, op_{W_{k+2}}, op_{W_{k+3}}$, v is no more stored in the regular register. $\square_{Lemma\ 9}$

Before to prove the validity property, let us consider how many Byzantine and cured servers can be present during a read() operation that last 3δ. For simplicity, to do that we refer to the scenarios depicted in Figure 6. If $k = 3$ there can be up to $4f$ (cf. Lemma 1) Byzantine servers and $2f$ cured servers. If $k = 2$ there can be up to $3f$ Byzantine servers (cf. Lemma 1) and f cured servers. In Figure 6 we depicted the extreme case in which there is a read() operation just after the last write() operation. The line marked as t_{wP} represents the time at which for sure correct servers are storing and thus replies with the last written value (cf. Lemma 7). Notice that when $\delta = \Delta$ s_4 has just the time to correctly reply to the client before being affected. Notice that if t_{wP} was concurrent with Byzantine agents movements, then during $[t, t + \delta]$ s_4 was still able to reply with the last written value because still present in W_i, i.e., the reply message happens before the 2δ timer expiration. In any case there are $\#reply_{CUM}$ correct servers that reply with the last written value and the number of those replies is greater than the number of replies coming from cured and Byzantine servers. From those observations the next Corollary follows.

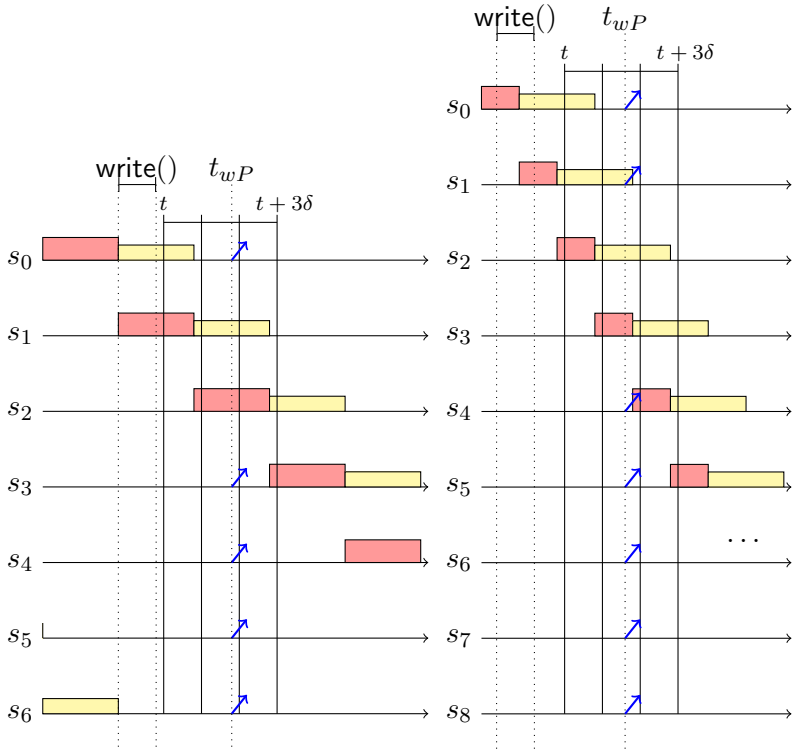

Figure 6: In the first scenario $\Delta = 2\delta$ and in the second one is $\Delta = \delta$. In red the period during which servers are faulty and in yellow the period during which servers are in a cured state. Blue arrows are the correct replies sent back by correct servers.

Corollary 3 *Let c_i be a client that invokes a* read() *operation that lasts 3δ time. During such time, the number of replies coming from correct servers is strictly greater than the number of replies coming from Byzantine and cured servers.*

Theorem 2 (Step 4.) *Any* read() *operation 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. We are interested in the time interval $[t_B(op_R), t_B(op_R)+\delta]$. The operation lasts 3δ, thus reply messages sent by correct servers within $t_B(op_R) + 2\delta$ are delivered by the reading client. During $[t, t + 2\delta]$ time interval there are at least $\#reply_{CUM}$ correct servers that have the time to deliver the read request and reply (cf. Corollary 3). We have to prove that

what those correct servers reply with is a valid value. There are two cases, op_R is concurrent with some write() operations or not.

- 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. For Lemma 8 after the write persistence time t_{wP} there are at least $\#reply_{CUM}$ correct servers storing v (i.e., $v \in \mathsf{conCut}(V_i, V_{safe_i}, W_i)$. Since $t_B(op_R) + 2\delta \geq t_C w$, then there are $\#reply_{CUM}$ correct servers replying with v. So the last written value is returned.

- op_R **is concurrent with some** write() **operation**. Let us consider the time interval $[t_B(op_R), t_B(op_R)+2\delta]$. In such time there can be at most three sequential write() operations $op_{W_1}, op_{W_2}, op_{W_3}$. Let op_{W_0} be the last write operation before op_R. In the extreme case in which those operations happen one after the other we have the following situation. $t_E(op_{W_0} < t_B(op_R))$ and the write persistence time of op_{W_0}, $t_{wP_0} < t_B(op_{W_0}) + 3\delta < t_B(op_R) + 2\delta < t_B(op_{W_3})$. Basically, the value written by op_{W_0} is overwritten in V_i by the value written op_{W_3}, but not before $t_B(op_R)+2\delta$, thus all correct servers have the time to reply with the last written value. Notice 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, for Lemma 6 Byzantine and cured servers can no force correct servers to store and thus to reply with a never written value. Only cured and Byzantine servers can reply with non valid values. As we stated, if $k = 2$ there are up to $4f$ non correct servers. If $k = 3$ there are $6f$ non correct servers. In both cases the threshold $\#reply_{CUM}$ is higher than the occurrences of non valid values that a reader can deliver. Mobile agents can not force the reader to read another or older value and even if an older values has $\#reply_{CUM}$ occurrences the one with the highest sequence number is chosen. $\square_{Theorem\ 2}$

From the reasoning used to prove Theorem 2 the next Corollary follows.

Corollary 4 *When a client c_i invokes a* read() *operation the last written value occurs in $reply_i$ at least $\#reply_{CUM}$ times.*

Theorem 3 *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 with no transient failures. Let δ be the upper bound on the communication latencies 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 SWMR Regular Register in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model.*

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

Self-Stabilization correctness proofs

What is left to prove are the necessary conditions for the system to self-stabilize after τ_{no_tr}. We first prove that with timestamps in $\mathcal{Z}_{13}$ it is always possible to uniquely order the values that clients and servers manage at the same time. Then, we prove that when the system is not stable, given the fact the timestamps are bounded, after a finite number of write() operations the system becomes stable.

Lemma 10 *During each* write() *operation* op_W *such that* $t_B(op_W) > \tau_{stab}$*, each non faulty server* s_i *has at most* 6 *values returned by the function* select_pairs($echo\_vals_i$) *during the same* maintenance() *operation and it is always possible to uniquely order them.*

Proof For sake of simplicity let us consider the scenario depicted in Figure 7, where $op_W(3), op_W(4), op_W(5)$, a sequence of write() operations, occurs (we represent each value with its associated timestamp $\in \mathcal{Z}_{13}$). By hypothesis the system is stable $t_B(op_W) > \tau_{stab}$, thus each correct server s_i has $V_i = \{0, 1, 2\}$ and those values are broadcast at the beginning of the maintenance() operation along with values in W_i as long as there are not enough occurrences of those values to be stored in V_{safe_i} and then V_i. Considering that:

- from Lemma 7, for each write() operation op_W, by time $t < t_B(op_W) + 3\delta$ the written value is stored in V_{safe_i};
- a written value v is removed from W_i after 2δ time, so at most by time $t_B(op_W) + 3\delta$ $v \notin W_i$;
- write() operations are sequential and last δ time (cf. 2)
- V_{safe_i} is a 3 dimension set.

It follows that given a sequence of at least three write() operations $op_W(3), op_W(4), op_W(5)$, before that $op_W(5)$ terminates the value written by $op_W(3)$ is in V_{safe_i} and 0 has been discharged (cf. Figure 7 the point marked by (a)), i.e. $t_E(op_W(5) > t_B(op_w(3)+3\delta)$. It follows if $op_W(6)$ occurs such that $t_B(op_W(6)) = t_E(op(W(5)))$ then 0 is no more in V_i and during this time 1 is overwritten by 3. Generalizing, during each maintenance() operation there are at most the 3 values in V_i and values belonging to the last 3 write() operations.
Let us now prove the second part of the statement. Considering that:

- timestamps are generated sequentially;
- during the same maintenance() operation timestamps can span a range of 6 values.

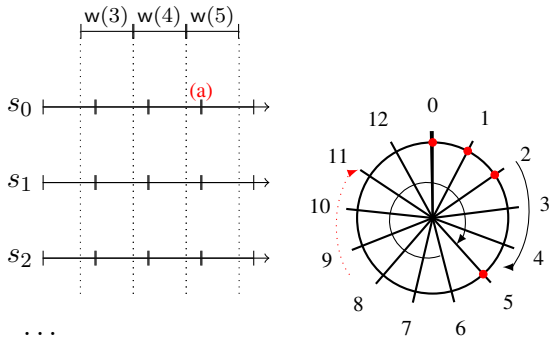

Figure 7: Example for $\Delta = \delta$. The small vertical lines are the points where maintenance() operations terminates and begins. Servers are storing $V_i = \{0, 1, 2\}$, for simplicity we represent values with their timestamp and we consider only correct servers.

Then for each couple of timestamp ts_q and ts_p returned by select_pairs($echo\_vals_i$) during the maintenance() operation, if ts_q has been generated before ts_p then $ts_p -_m ts_q \leq 5$. This means that given $\mathcal{Z}_{13}$ and ts_q, ts_p there is only one way to order them. If ts_p is generated before ts_q then $ts_q -_m ts_p \geq 7$ which is a contradiction with the fact that during the same maintenance() operation timestamps can span a range of 6 values (cf. the "clock" depicted in Figure 7). $\square_{Lemma\ 10}$

Lemma 11 *During each* read() *operation* op_R *such that* $t_B(op_R) > \tau_{stab}$*, each client* c_i *delivers at most* 9 *values whose occurrence is* $\#reply_{CUM}$ *and it is always possible to uniquely order them.*

Proof For simplicity let us consider the scenario depicted in Figure 8, where the read() operation op_R happens after the end of the last write() operations $op_W(3)$ and $op_W(4)$ but before their persistence time t_{wP}. Moreover op_R is concurrent with four subsequent write() operations $op_W(4), op_W(5), op_W(6), op_W(7)$. During op_W we have the following:

- t_{wP} of $op_W(3)$ and $op_W(4)$ are after $t_B(op_R)$;
- for times constraints all the previous write() operations are in V_i at each correct servers;
- by hypothesis the system is stable, thus each correct server s_i can have $V_i = \{0, 1, 2\}$ (no yet overwritten by $op_W(3)$ and $op_W(4)$ values);

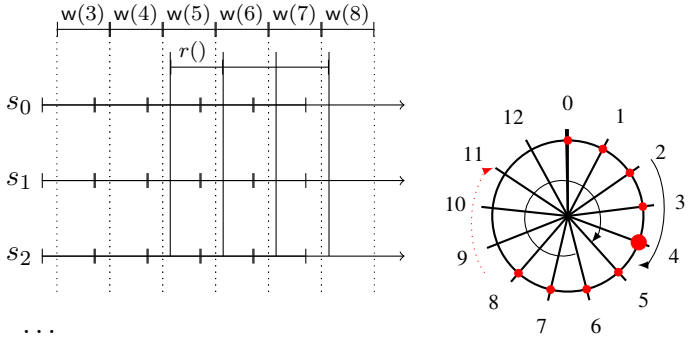

Figure 8: Example for $\Delta = \delta$. The small vertical lines are the points where maintenance() operations terminates and begins. In both cases servers are storing $V_i = \{0, 1, 2\}$, for simplicity we represent values with their timestamp and we consider only correct servers.

- for Corollary 4 each correct server replies with the last written value 4;
- if messages are fast enough each correct server can reply with also $\{5, 6, 7, 8\}$.

Thus c_i has potentially 9 values that occur $\#reply_{CUM}$ times.
To prove the second part of the statement, consider that for Corollary 4 the last written value is always present. Thus, there can be up to 9 sequential values ts_p and the last written value ts_{ls} is in the middle, it follows that the distance between each value ts_p and the one in the middle ts_{ls} is such that $|ts_p -_m ts_{lw}| \leq 4$ (cf. the "clock" in Figure 8). If follows that for each triple of value there always exist a value in the middle such that there is one only way to order them. For example, let us consider the set $\{7, 0, 3\}$ it can be ordered in three ways: $7, 0, 3$, $3, 7, 0$ and $0, 3, 7$. In the first two cases $7 -_m 0 > 4$ and $7 -_m 0 > 4$ respectively, thus the order $0, 3, 7$ is the only possible one. $\square_{Lemma\ 11}$

Lemma 12 *Let $op_{W_1}, \ldots, op_{W_{10}}$ be a sequence of 10* write() *operations, occurring after τ_{no_tr}. At time $t > t_E(op_{W_{10}})$ the system is self-stabilized.*

Proof For Lemma 4 if there are $\#echo_{CUM}$ correct servers storing the same value v then such value is stored by all correct servers after δ time since the beginning of the maintenance() operation. Thus given the first maintenance() operation after τ_{no_tr} correct servers are either storing the same values or empty set. If correct servers are storing nothing, then after the end of the first write() operation the system is stable, since for Corollary 4 such a value is returned by the next read()

operation.
If V_{safe_i} is not empty then different scenarios may happen.

- case a. If values stored in V_{safe_i} have not an unique order (e.g., $\exists ts_p, ts_q \in \mathcal{Z}_{13} : ts_p = ts_q \vee |ts_p -_m ts_q| > 5$), then at the next maintenance() operation, the function checkOrderAndTrunc(V_{safe_i}) resets such set. Notice that such reset may happen at the beginning of different maintenance() operations, depending on Δ. For sake of simplicity let us consider Figure 7, 3 can be stored in V_{safe_0} during the three different maintenance() operations that occur since the beginning of the write() operation and the point marked as (a), the t_{wP}. In such time interval two other write() operations may occur. But after the point marked as (a) all non Byzantine servers reset their V_{safe_i} set. Thus now, after the end of the next write() operation the system is stable, indeed, for Corollary 4 such value is returned by the next read() operation. Thus in such a case after four write() operations the system is stable;

- case b. If values stored in V_{safe_i} can be uniquely ordered, e.g. $V_{safe_i} = \{0, 1, 2\}$ then three scenarios may happen;

  - the next written value does not have an unique order with respect to each value in V_{safe_i}, (e.g., $0, 1, 2, 6, 7$) then again the set is reset and case (a.) takes place;
  - the next written value timestamp is ordered as newest with respect to the values in V_{safe_i} (e.g., $3, 4, 5$) and then, at the end of the write() operation, the system is stabilized. Indeed for Corollary 4 such value is returned to the next read() operation;
  - the next written value is ordered as older with respect to each value in V_{safe_i} and thus is dropped. This happens up to the write() operation that writes a value equal to a value in V_{safe_i}, when this happens V_{safe_i} is reset and after four write() operations the system is stable (cf. case a.). If $V_{safe_i} = \{0, 1, 2\}$ then in the worst case are needed 6 write() operations, e.g., $8, 9, 10, 11, 12, 0$. Then at the next maintenance() operation, when two values associated with 0 are in V_{safe_i} the function checkOrderAndTrunc(V_{safe_i}) resets such set. Thus, after the end of the next four write() operations the system is stable (cf. case a.), indeed for Corollary 4 such value is returned by the next read() operation. Thus, after 10 write() operations the system is stable.

Considering all those cases, the worst case scenario happens when 10 write() operations are required to stabilise the system. The claim trivially follows generalizing the argumentation for general timestamps in $\mathcal{Z}_{13}$. $\square_{Lemma\ 12}$

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

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

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

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 δ be the upper bound on the communication latencies 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 3 and Lemma 12. $\square_{Theorem\ 4}$

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

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

5 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 [8, 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 $T_{10write()}$, where $T_{10write()}$ is the time

needed to execute ten *complete* write() operations. Contrary to the $(\Delta S, CAM)$ model, $(\Delta S, CUM)$ model required to design a longer maintenance() operation (that lasts 2δ time). As a side effect, also the read() operation completion time increased and it has a direct impact on the size of the bounded timestamp domain that characterize 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. Nevertheless, interesting is the study of optimal maintenance() solutions.

References

[1] Noga Alon and Hagit Attiya and Shlomi Dolev and Swan Dubois and Maria Potop-Butucaru and Sébastien Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. J. Comput. Syst. Sci. 2015.

[2] Attiya, Hagit and Bar-Noy, Amotz and Dolev, Danny. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)* (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 and Shlomi Dolev and 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).

[8] 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.

[9] Silvia Bonomi and Maria Potop-Butucaru and Sébastien Tixeuil. Byzantine Tolerant Storage. In *Proceedings of the International Conference on Parallel and Distributed Processing Systems* (IEEE IPDPS 2015).

[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*.

[11] 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.

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

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

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

[15] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] Mikhail Nesterenko and Anish Arora. Tolerance to Unbounded Byzantine Faults. *21st Symposium on Reliable Distributed Systems (SRDS 2002)*

[27] 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 (2007)*,

[28] Yusuke Sakurai and Fukuhito Ooshita and Toshimitsu Masuzawa. A Self-stabilizing Link-Coloring Protocol Resilient to Byzantine Faults in Tree Networks. *8th International Conference on Principles of Distributed Systems (OPODIS 2005)*.

[29] Swan Dubois and Toshimitsu Masuzawa and Sébastien Tixeuil On Byzantine Containment Properties of the min+1 Protocol. *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*.

[30] Swan Dubois and Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the Impact of Unbounded Attacks in Stabilization. *IEEE Transactions on Parallel and Distributed Systems (2011)*.

[31] Swan Dubois and Toshimitsu Masuzawa and Sébastien Tixeuil. Maximum Metric Spanning Tree made Byzantine Tolerant. *25th International Symposium on Distributed Computing (DISC 2011)*.