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

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

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

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
†Université Pierre & Marie Curie (UPMC) – Paris 6, France
{maria.potop-butucaru, sebastien.tixeuil}@lip6.fr

Abstract

This paper proposes the first implementation of a regular register by n servers that is tolerant to both *mobile Byzantine agents*, and *transient failures* (it is self-stabilizing) in a round-free synchronous model. We consider the most difficult model for mobile Byzantine agents to date where the message delay, δ, and the speed of mobile Byzantine agents, Δ, are completely decoupled. Moreover, servers are not aware of their state (infected or correct) after mobile Byzantine agents left them.

The register is maintained by n servers and our algorithm tolerates *(i)* any number of transient failures, and *(ii)* up to f Mobile Byzantine agents. Our implementation uses bounded timestamps from the $\mathcal{Z}_5$ domain, and is optimal with respect to the number of tolerated mobile Byzantine agents. The convergence time of our solution is upper bounded by $3\Delta + T_{5write()}$, where $T_{5write()}$ is the time needed to execute five *complete* $write()$ operations.

Contact Author: Silvia Bonomi
Address: Dipartimento di Ingegneria Informatica, Automatica e Gestionale "A. Ruberti"
Universitá degli Studi di Roma "La Sapienza"
Via Ariosto, 25
I-00185 Roma (RM)
Italy
Telephone Number: +39 06 77 27 4017

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). The classical setting considers Byzantine participants remain so during the entire execution, yet software rejuvenation techniques increase the possibility that a corrupted node *does not remain corrupted during the whole system execution* and may be aware of its previously compromised status [27].

Mobile Byzantine Failures (MBF) models have been recently introduced to integrate those concerns. Then, faults are represented by Byzantine agents that are managed by an omniscient adversary that "moves" them from a host process to another, an agent being able to corrupt its host in an unforeseen manner. MBF investigated so far consider mostly *round-based* computations, and can be classified according to Byzantine mobility constraints: *(i)* constrained mobility [7] agents may only move from one host to another when protocol messages are sent (similarly to how viruses would propagate), while *(ii)* unconstrained mobility [1, 3, 15, 22, 23, 25] agents may move independently of protocol messages. In the case of unconstrained mobility, several variants were investigated [1, 3, 15, 22, 23, 25]: Reischuk [23] considers that malicious agents are stationarity for a given period of time, Ostrovsky and Yung [22] introduce the notion of mobile viruses and define the adversary as an entity that can inject and distribute faults; finally, Garay [15], and more recently Banu *et al.* [1], and Sasaki *et al.* [25] or Bonnet *et al.* [3] 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 hosts 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 [1, 3, 15, 25] lies in the knowledge that hosts have about their previous infection by a Byzantine agent. In Garay's model [15], a host is able to detect its own infection after the Byzantine agent left it. Sasaki *et al.* [25] investigate a model where hosts cannot detect when Byzantine agents leave. Finally, Bonnet *et al.* [3] 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.* [6]. 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* [9, 10] 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 cause by Byzantine participants), a self-stabilizing protocol ensure 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* [26], and *Byzantine quorum systems* [2, 17, 19, 18]. 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. Recently, Bonomi *et al.* [4] proposed optimal self-stabilizing atomic register implementations for round-based synchronous systems under the four Mobile Byzantine models described in [1, 3, 15, 25]. The round-free model [6] where Byzantine moves are decoupled from protocol rounds also enables optimal solutions (with respect to the number of Byzantine agents) for the implementation of regular registers.

Multitolerance. Extending the effectiveness of self-stabilization to permanent Byzantine faults is a long

time challenge in distributed computing. Initial results were mostly negative [8, 11, 21] 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 [20, 21, 24] 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 [12, 13, 14] 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. To our knowledge, the problem of tolerating both arbitrary transient faults and mobile Byzantine faults has been considered in the literature only in round-based synchronous systems [4].

Our Contribution. We consider the problem of emulating a regular register in a network where both arbitrary transient faults and mobile Byzantine faults can occur, but where processes and Byzantine agent moves are decoupled. With respect to previous work on round-free register emulation [6], we add the self-stabilization property, and bounded (memory) timestamps. With respect to previous results that are self-stabilizing and mobile Byzantine tolerant [4], we consider the more relaxed round-free hypothesis, and bounded timestamps.

In more details, we present a regular register implementation that uses bounded timestamps from the $\mathcal{Z}_5$ domain and is optimal with respect to the upper bound on the number of mobile Byzantine processes. The convergence time of our solution is upper bounded by $3\Delta + T_{5write()}$, where $T_{5write()}$ is the time needed to execute five *complete* $write()$ operations, each $write()$ operation copleting in finite time.

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). Processes in the system do not have access at the fictional global time. 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.

We assume that an arbitrary number of clients may crash while up to f servers are affected, at any time t, by *Mobile Byzantine Failures*. The Mobile Byzantine Failure adversarial model considered in this paper (and described in details below) is stronger than any other adversary previously considered in the literature [1, 3, 7, 15, 22, 23, 25].

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 trough message passing. In particular, we assume that: *(i)* each client $c_i \in \mathcal{C}$ can communicate with every server trough a broadcast() primitive, *(ii)* each server can communicate with every other server trough a broadcast() primitive, and *(iii)* each server can communicate with a particular client trough 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).

Synchronous System.

The system is *round-free synchronous* if: *(i)* the processing time of local computations (except for wait statements) are negligible with respect to communication delays, and are 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 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 $\mathcal{P}$ that is composed by a set of distributed algorithms. Each algorithm in $\mathcal{P}$ is represented by a finite state automata 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*.

Definition 1 (Execution History) *Let $\mathcal{P}$ be a distributed protocol. Let H be the set of all the events generated by $\mathcal{P}$ at any process p_i in the distributed system and let $\rightarrow$ be the happened-before relation. An execution history $\hat{H} = (H, \rightarrow)$ is a partial order on H satisfying the relation $\rightarrow$.*

Definition 2 (Valid State at time t) *Let $\hat{H} = (H, \rightarrow)$ be an execution history of a generic computation and let $\mathcal{P}$ be the corresponding protocol. 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 can be generated by executing $\mathcal{P}$ on $\hat{H}$.*

The Mobile Byzantine Failure (MBF) models considered so far in literature [1, 3, 7, 15, 22, 23, 25] assume that faults, represented by Byzantine agents, are controlled by a powerful external adversary that "moves" them from a server to another. Note that the term "mobile" does not necessary mean that a Byzantine agent physically moves from one process to another but it rather captures the phenomenon of a progressive infection, that alters the code executed by a process and its internal state.

2.1 Mobile Byzantine Models

As in the case of round-based MBF models [1, 3, 7, 15, 25], we assume that any process previously infected by a mobile Byzantine agent has access to a tamper-proof memory storing the correct protocol code. However, a healed (cured) server may still have a corrupted internal state and cannot be considered correct. As a consequence, the notions of correct and faulty process need to be redefined when dealing with Mobile Byzantine Failures.

Definition 3 (Correct process at time t) *Let $\hat{H} = (H, \rightarrow)$ be an execution history and let $\mathcal{P}$ be the protocol generating $\hat{H}$. 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.*, $Co([t, t']) = \bigcap_{\tau \in [t,t']} Co(\tau)$).*

Definition 4 (Faulty process at time t) *Let $\hat{H} = (H, \rightarrow)$ be an execution history and let $\mathcal{P}$ be the protocol generating $\hat{H}$. 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 5 (Cured process at time t) *Let $\hat{H} = (H, \rightarrow)$ be an execution history and let $\mathcal{P}$ be the protocol generating $\hat{H}$. 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.*, $Cu([t, t']) = \bigcap_{\tau \in [t,t']} Cu(\tau)$).*

In this work we consider the $(\Delta S, CUM)$ MBF model [6], that can be specified as follows. $(\Delta S, *)$ allows to consider coordinated attacks where the external adversary needs to control a subset of machines. In this case, compromising new machines will take almost the same time as the time needed to detect the attack or the time necessary to rejuvenate. This may represent scenarios with low diversity where compromising time depends only on the complexity of the exploit and not on the target server. More formally, 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 $t_0 + \Delta, t_0 + 2\Delta, \ldots, t_0 + i\Delta$, with $i \in \mathbb{N}$).
$(*, CUM)$ represents situations where the server is not aware of a possible past infection. This scenario is typical of distributed systems subject to periodic maintenance and proactive rejuvenation. In this systems, there is a schedule that reboots all the servers and reloads correct versions of the code to prevent infections to be propagated in the whole network. However, this happens independently from the presence of a real infection and implies that there could be periods of time where the server execute the correct protocol however its internal state is not aligned with non compromised servers.

As in the round-based models, we assume that the adversary can control at most f Byzantine agents at any time (*i.e.*, Byzantine agents are not replicating themselves while moving).

In our work, only servers can be affected by the mobile Byzantine agents[1]. It follows that, at any time t, $|B(t)| \leq f$. However, during the system life time, all servers may be affected by a Byzantine agent (*i.e.*, none of the server is guaranteed to be correct forever).
In addition to the possibility of mobile Byzantine failures at server side, processes may also suffer form *transient* failures, *i.e.*, local variables of any process (writer, reader, servers) can be arbitrarily modified [10]. 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 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*

[1]It is trivial to prove that in our model when clients are Byzantine it is impossible to implement deterministically even a safe register. The Byzantine client will always introduce a corrupted value. A server cannot distinguish between a correct client and a Byzantine one.

is said to be *failed* if it is invoked by a process that crashes before the reply event occurs. According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock (unknown to the processes) and the invocation time and response time of every operation are defined with respect to this fictional clock.
Given two operations op and op', their invocation event times ($t_B(op)$ and $t_B(op')$) and their reply event times ($t_E(op)$ and $t_E(op')$), we say that op *precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* ($op||op'$). Given a 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() operation concurrently (*i.e.*, for any given client c_i, the set of operations executed by c_i is totally ordered). We also assume that initially the register stores a default value $\perp$ written by a fictional write($\perp$) operation happening instantaneously at round r_0. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [16]: *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 eventually terminates.
- ss − Validity: There exists a time τ_{stab} such that each read operation invoked at time $t > \tau_{stab}$ returns the last value written before its invocation, or a value written by a write() operation concurrent with it.

Bonomi *et al.* [6] proved the necessity of an additional maintenance() operation, executed regularly, to cope with the Byzantine agent moves between read() and write() operations. This result naturally extends to our case, as a self-stabilizing algorithm, once stabilized, must provide the same guarantees as a non-stabilizing one.

Theorem 1 *Let n be the number of servers emulating a safe register and let f be the number of Mobile Byzantine Agents affecting servers. Let $\mathcal{A}_R$ and $\mathcal{A}_W$ be respectively the algorithms implementing the* read() *and the* write() *operation assuming no communication between servers. If $f > 0$ then there exists no protocol $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ implementing a self-stabilizing safe register in any of the MBF models for round-free computations defined in [6].*

Proof Let us assume that such algorithm $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ exists, i.e., after the end of transient failures it provides a self-stabilized safe register. If $\mathcal{P}_{reg}$ is correct, it means that both $\mathcal{A}_R$ and $\mathcal{A}_W$ implementing respectively the read() and the write() operation terminates i.e., they stop to execute steps when the operation is completed. Let $t > \tau_{stab}$ be the time at which the last operation op terminated and let us assume that no other operation is invoked until time $t' > t$. Let us note that during the time interval $[t, t']$ no algorithm is running as all the operations issued in the past are completed. As a consequence, no correct server and no cured server change its state. However, considering that t' does not depend on $\mathcal{P}_{reg}$ (i.e., it is not controlled by the register protocol but it is defined by clients) and considering the mobility of the Mobile Byzantine agents, we may easily have a run where every correct server is faulty and its state can be corrupted at some time in $[t, t']$. Considering that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ and that $\mathcal{A}_R$ and $\mathcal{A}_W$ are not running in $[t, t']$ we can have that every server stores a non valid state at time t' and the register value is lost. As a consequence, $\mathcal{A}_R$

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

$k\Delta \geq 2\delta,\ k \in \{1,2\}$	$n \geq 2(k+1)f+1$	$\#reply \geq 2kf+1$	$\#echo \geq (k+1)f+1$
$k=1$	$4f+1$	$2f+1$	$2f+1$
$k=2$	$6f+1$	$4f+1$	$3f+1$

has no way to read a valid value and the validity property is violated. It follows that $\mathcal{P}_{reg}$ is does not exist and we have a contradiction. $\square_{Theorem\ 1}$

4 Regular Register implementations

Our self-stabilizing regular register emulation is composed of three parts: the *write* operation, the *read* operation, and the *maintenance* operation prescribed by our execution model.The write() and read() algorithms follow the classical quorum-based implementations. The first is in charge of writing on enough servers such that there are enough correct servers able to reply when a read() operation occurs. The maintenance() operation is in charge to keep the number of correct servers above the thresholds in Table 1 despite mobile Byzantine movements.The tricky part of the algorithm is to employ bounded timestamps from the domain $\mathcal{Z}_5$ in such a way to always define a total order on the written values with respect to their timestamps.
Each written value is represented as $\langle val, sn\rangle$ where val is the content and sn the corresponding sequence number, $sn \in Z_5 = \{0,1,2,3,4\}$. Let us define two operations on such values: addition: $+_5 : Z_5 \times Z_5 \rightarrow Z_5, a +_5 b = (a+b) \bmod 5$; and subtraction: $-_5 : Z_5 \times Z_5 \rightarrow Z_5, a -_5 b = a +_5 (-b)$. Note that $(-b)$ is the opposite of b. That is, the number that added to b gives 0 as result, *i.e.*, $b +_5 (-b) = 0$.

4.1 $\mathcal{P}_{reg}$ Detailed Description

Our emulation, protocol $\mathcal{P}_{reg}$, is described in Figures 2 - 4.
Local variables at client c_i. Each client c_i maintains a set $reply_i$ that is used during the read() operation to collect the three tuples $\langle j, \langle v, sn\rangle\rangle$ sent back from servers. Additionally, c_i maintains a local sequence number csn that is incremented, respect to the $\mathcal{Z}_5$ 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:

- $V_i[0..2]$: an array such that $V_i[0] = \langle val_0, sn_0\rangle$, $V_i[1] = \langle val_1, sn_1\rangle$ and $V_i[2] = \langle val_2, sn_2\rangle$ such that $sn_2 -_5 sn_1 \leq 2$ and $sn_1 -_5 sn_0 \leq 1$ and $sn_k \in \mathcal{Z}_5, k \in \{0,1,2\}$. This array, when a server is correct is expected to be completely filled with three values. This set is reset and repopulated at each maintenance() operation.
- FW_i: a set that contains elements $\langle val_k, sn_k\rangle, sn_k \in \mathcal{Z}_5$. This set may contain up to three values, depending on how many consecutive write() operations occur in δ. Such set is populated with values, due to a write() operation, that have been forwarded $\#reply$ times by servers. It is emptied during the maintenance() at most every δ time.
- W_i: is the set where servers store values coming directly from the writer, including an *epoch* value, $\langle v, sn, epoch\rangle$. *epoch* is set to 1 at the beginning and is decreased by maintenance() operations. Values from this set are deleted when the same value appears in the FW set or when *epoch* reaches the (-1) value.

- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated trough ECHO messages at the beginning of the maintenance() operation. $echo\_vals_i$ stores vectors vec_j whose elements are $\langle v, sn, epoch\rangle_j$ propagated by servers s_j. Every vec_j is the concatenation of V_j (which is a vector itself) and W_j, which is a set whose elements can be ordered with respect to their timestamp. The semantic of the *epoch* variable is the same as in W_j. trunc(echo_vals$_i$), with a slightly abuse of notation, is the set $\{\mathsf{trunc}(vec_j, 3), \forall vec_j \in echo\_vals_i\}$. Finally, $echo\_read_i$ stores the identifiers of concurrently reading clients in order to notify cured servers about them.

- fw_vals_i: set variable storing a triple $\langle v, sn, epoch\rangle_j$ meaning that server s_j forwarded a write message with value v and sequence number sn. The semantic of the *epoch* variable is the same as in W_i.

- $pending\_read_i$: set variable used to collect identifiers of the clients that are currently reading.

In order to simplify the code of the algorithm, let us define the following functions:

- select_three_pairs_max_sn($echo\_vals_i$): this function takes as input the set $echo\_vals_i$ whose values vec_j are the result of conCut(V_j, W_j). That function returns, if there exist, the three newest tuples $\langle v, sn\rangle$, such that there exist at least $\#echo$ occurrences in $echo\_vals_i$ of such tuple. If there are less than three tuples, the remaining tuples returned are $\langle \perp, 0\rangle$.

- select_value($reply_i$): this function takes as input the $reply_i$ set of replies collected by client c_i and returns the pair $\langle v, sn\rangle$ occurring $\#reply$ times. If there are more pairs occurring enough times, it returns the newest.

- older(Set): given a set of values, whose associated timestamps belong to $\mathcal{Z}_5$, then such function returns the older value among those in Set.

- trunc($Vector, index$): takes as input an array and returns the last $index$ elements.

- conc($Vector, Set$): takes as input an array $Vector$ and a set Set, assuming that is it possible to order univocally elements in Set, and returns the concatenation $Vector \circ Set$.

- conCut(V_i, W_i) takes as input the array V_i and the set W_i. Since we can order the elements in W_i with respect to their timestamp, then we manage it as a vector. Such function first concatenates them: $Vector \leftarrow \mathsf{conc}(V_i, W_i)$, removes $\langle \perp, \perp\rangle$ elements and returns trunc($Vector, 3$). An example: $V = [\langle v_a, 5\rangle, \langle v_b, 0\rangle, \langle v_c, 1\rangle]$ and $W = [\langle \perp, \perp\rangle, \langle w_0, 2\rangle, \langle w_1, 3\rangle]$. The concatenation is $[\langle v_a, 5\rangle, \langle v_b, 0\rangle, \langle v_c, 1\rangle, \langle \perp, \perp\rangle, \langle w_0, 2\rangle, \langle w_1, 3\rangle]$ and the returned array is $[\langle v_c, 1\rangle, \langle w_0, 2\rangle, \langle w_1, 3\rangle]$ which is composed by the last three values different from $\langle \perp, \perp\rangle$.

In the following we present a general view of our algorithm.

Maintenance operation. Such operation is executed by servers periodically at times $T_i = t_0 + i\Delta$. Each server checks if there are expired (*i.e.*, *epoch*$\notin \{0, 1\}$) or invalid values in W_i, $echo\_vals_i$ and fw_vals_i. In both cases such values are deleted. Otherwise their $epoch$ is decreased by 1. Function check is invoked on V_i to check if its values are compliant to a correct system behavior. More in details, with respect to their timestamps, values in $V[0]$ and $V[1]$ are temporarily be one after the other and value in $V[2]$ can be temporarily just after $V[1]$ or there can be a missing one. Finally it checks if all timestamps belong to $\mathcal{Z}_5$ and that there are no values with the same timestamp. Notice that V_i may contain $\perp$ values, for readability

this case is not explicitly managed, but situations as $V_i = [\langle\perp,\perp\rangle, \langle\perp,\perp\rangle, \langle v_2, sn_2\rangle]$ are allowed. Finally FW_i is emptied.
Now each server is ready to broadcast an ECHO message with the result of $conc(V_i, W_i)$ and the set $pending\_read_i$ (it contains identifiers of clients that are currently running a read() operation). After δ time units, servers try to update their state by checking the number of occurrences of each pair $\langle v, sn\rangle$ received with ECHO messages. In particular, the first empties the V set and then they try to update such set by invoking select_three_pairs_max_sn($echo\_vals_i$) function which populates V with at least one tuple $\langle v, sn\rangle$. If there is only one tuple $\langle v, sn\rangle$, s_i can deduce that there exists a concurrent write() operations that are updating the register value concurrently with the maintenance() operation. Thus, s_i considers $\langle\perp, 0\rangle$ as the pair associated to the value that is concurrently written. After that it checks if there are values in FW_i. In that case the function INSERT is invoked on such values and FW_i is emptied. Such function tries to insert the value in the proper position in V_i ($V[1]$ or $V[2]$) and then checks if the vector is properly defined invoking the check function. If not, V_i is reseted and the value is inserted in $V[2]$. Finally server starts replying to clients that are currently reading. The same check on FW_i and insertion in V_i is performed after $\Delta - \delta$ time during the maintenance() operation. Notice that this second check happens if $\Delta > \delta$.
Write operation. When the write() operation is invoked, the writer increments $csn \leftarrow csn +_5 1$, sends WRITE($vcsn$) to all servers and finally returns after δ time.

For each server s_i, two cases may occurs: (case 1) s_i delivers WRITE($\langle v, csn\rangle$) message when it is not affected by a Byzantine agent; (case 2) s_i delivers WRITE($vcsn$) message when it is affected by a Byzantine agent.
case 1 s_i stores v in W and forward it to every servers sending the WRITE_FW($i, \langle v, csn\rangle$) message. Then it is echoed at the beginning of each maintenance() operation as long as v is in W_i or V_i. In order for v to be in V_i it has to appear enough times in $fw\_vals_i \cup echo\_vals_i$, when it happens v is inserted in FW_i. At the next check on FW_i, during the maintenance() operation, v is removed from FW_i and inserted in V_i. **case 2** When s_i is no more affected it can deliver v with both WRITE_FW($j, \langle v, csn\rangle$) message and ECHO($j, V_j \cup W_j, pending\_read_i$) message, so that v goes to populate the $fw\_vals_i \cup echo\_vals_i$ set. When there are enough occurrences, v is stored in FW_i and at the next check on FW_i, during the maintenance() operation, v is removed from FW_i and inserted in V_i.
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 2δ time, while the $reply_i$ set is populated with servers replies, and from such set it picks the values occurring enough 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. After, s_j invokes conCut(V_j, W_j) function to prepare the reply message for c_i. The result of such function is sent back to c_i in the REPLY message. Such message is also computed and sent at the end of each maintenance() operation, in the case s_j was affected by Byzantine agent. Finally a REPLY message containing just one value is sent when a new value is added in FW_j 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.

4.2 Correctness

Let us fist characterize the correct system behavior, i.e., when the protocol is correctly executed after τ_{stab} (the end of the transient failure).

```
function epochCheck(Set):
(01) for each (⟨v, sn, epoch⟩_j ∈ Set) do
(02)        if (epoch ∉ {0, 1})
(03)          Set ← Set \ ⟨v, sn, epoch⟩_j;
(04)          else epoch ← epoch − 1;
(05)        endif
(06) endFor
```

```
function check(V_i): % V_i[k] = ⟨val_k, sn_k⟩, k ∈ {0, 1, 2}
(07) if (¬(sn_1 −_5 sn_0 = 0 ∧ (sn_2 −_5 sn_1 = 1 ∨ sn_2 −_5 sn_1 = 2)))∨
(08)   (∃i, j ∈ {0, 1, 2}s.t.sn_i = sn_j) ∨ (∃sn_k, k ∈ {0, 1, 2}, s.t. sn_k ∉ Z_5)
(09)   then return FALSE;
(10)   else return TRUE;
(11) endif
```

```
function insert(V_i, ⟨val, sn⟩):
(12) if ∃V[k] = ⟨val, sn⟩, k ∈ {0, 1, 2}
(13)   then return;
(14) endIf
(15) if (sn_1 +_5 1 = sn = sn_2 −_5 1)
(16)   then V[0] ← V[1]; V[1] ← ⟨val, sn⟩;
(17) endIf
(18) if (sn_1 +_5 2 = sn_2 +_5 1 = sn)
(19)   then V[0] ← V[1]; V[1] = V[2]; V[2] ← ⟨val, sn⟩;
(20) endIf
(21) if ¬(check(V_i));
(22)   thenV_i[0] ← ⟨⊥, ⊥⟩, V_i[1] ← ⟨⊥, ⊥⟩, V_i[2] ← ⟨val, sn⟩;
(23) endif
```

Figure 1: Auxiliary functions.

Definition 6 (legal sequence) *Let $op_{W_1}, op_{W_2}, \ldots, op_{W_k}, op_{W_{k+1}}, \ldots$ be S a sequence of consecutive* write() *operations issued on the regular register after τ_{stab} and let $\langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle, \ldots, \langle v_k, sn_k\rangle, \langle v_{k+1}, sn_{k+1}\rangle, \ldots$ be the respective written values. The sequence S' is legal if S' is obtained from S after applying the following rule: for each adjacent couple of elements in S, $\langle v_k, sn_k\rangle, \langle v_{k+1}, sn_{k+1}\rangle$, s.t. $sn_k +_5 1 = sn_{k+1}$, every element can be swapped with an adjacent one at most once.*

Definition 7 (legal state) *For any correct server s_i, V_i is in a legal state if its three elements belong to a subsequence of a legal sequence.*

Definition 8 (legal value) *Let V_i be in a legal state and let v be a value to be inserted in V_i in such a way that all elements in V_i are ordered from the oldest to the newest. v is said to be a legal value if after its insertion in V_i, V_i is still in a legal state.*

To prove the correctness of $\mathcal{P}_{reg}$, we first demonstrate that the termination property is satisfied i.e, that read() and write() operations terminates. Let us note that the termination property is independent from the specific instance of the MBF model considered.

Lemma 1 *If a correct client c_i invokes* 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 03-04, Figure 3). $\square_{Lemma\ 1}$

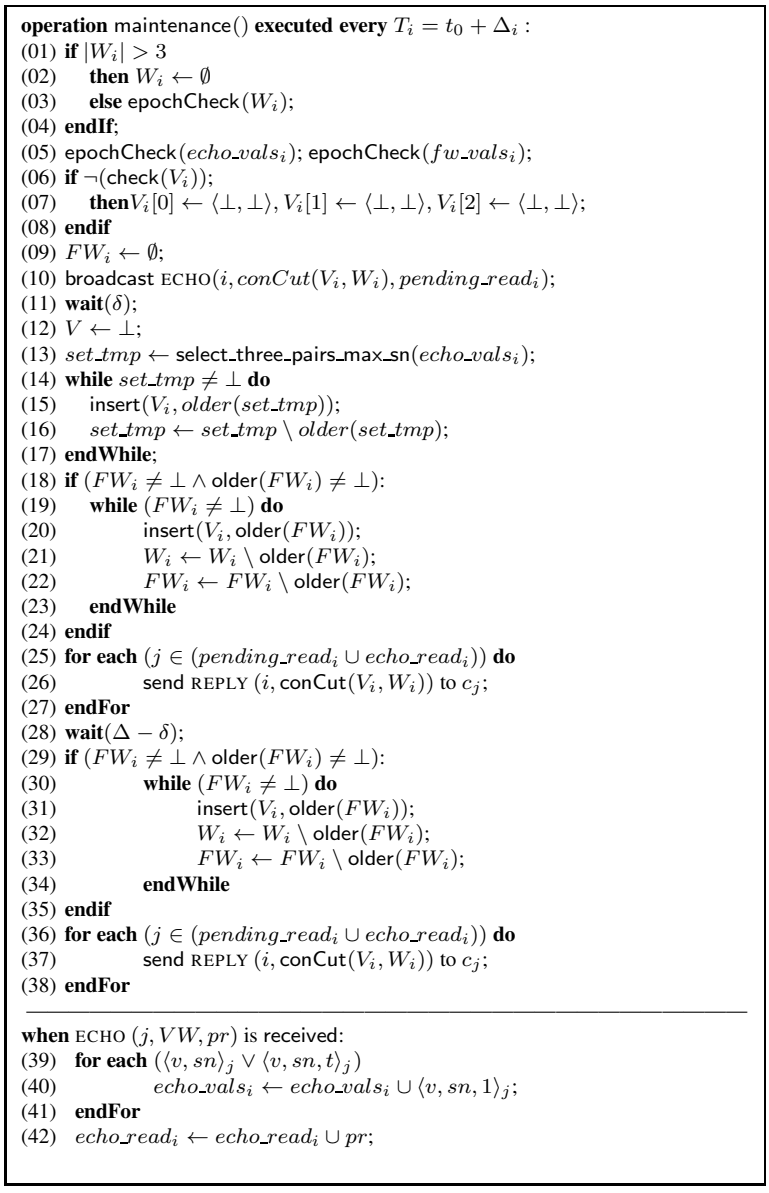

Figure 2: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CUM)$ model with bounded timestamp.

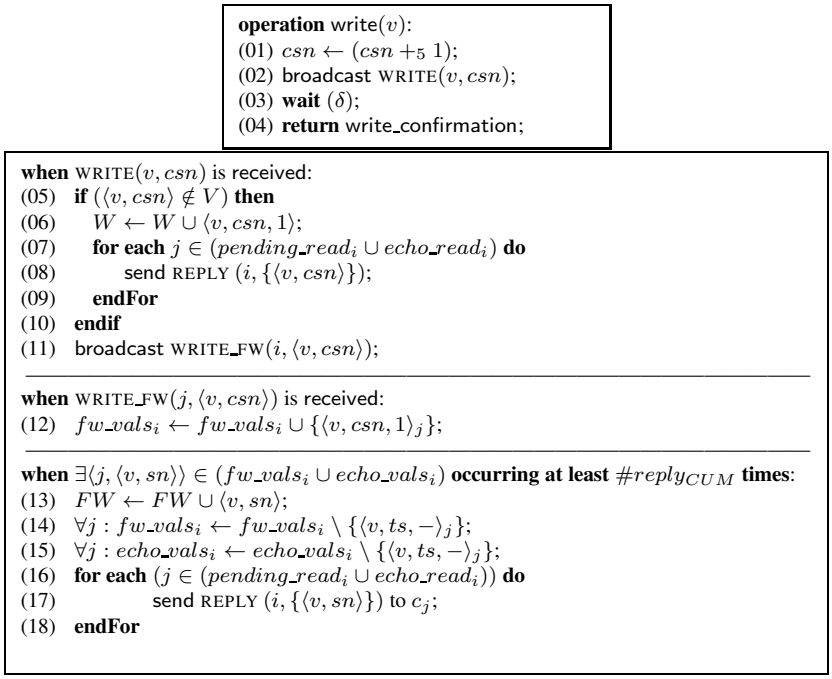

Figure 3: $\mathcal{A}_W$ algorithms, server side and client side respectively, implementing the write(v) operation in the $(\Delta S, CUM)$ model with bounded timestamp.

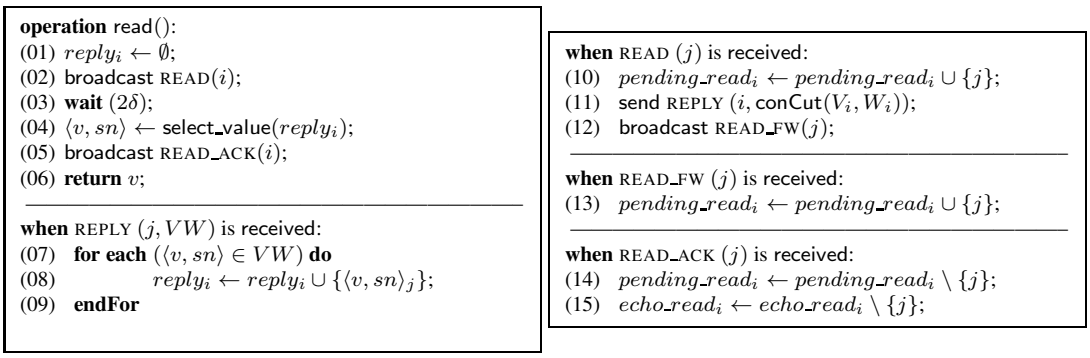

Figure 4: $\mathcal{A}_R$ algorithms, server side and client side respectively, implementing the read() operation in the $(\Delta S, CUM)$ model with bounded timestamp.

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

Proof The claim simply follows by considering that a read() returns a value to the client after 2δ time, independently of the behavior of the servers (see lines 03-06, Figure 4). $\square_{Lemma\ 2}$

Theorem 2 (ss-Termination) *Any operation invoked on the register eventually terminates.*

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

Lemma 3 *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), (ii) $n \geq 2(k+1)f+1$, (iii) there are $\#echo$ server $s_j \in Co(T_i)$ such that $V_j = V_k, \forall s_j, s_k \in Co(T_i)$ and (iv) there are no* write() *operations during $[T_i, T_i + \delta]$, then $\forall s_c \in Cu(T_i)$, $s_c \in Co(T_i + \delta)$ and all servers in $Co(T_i + \delta)$ are storing V_j.*

Proof By hypotheses at T_i there are $\#echo$ correct servers s_j storing the same $V_j = [\langle v_0, sn_0\rangle, \langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle]$ and running the code in Figure 2. In particular each server broadcasts a ECHO() message with attached the content of conCut(V_i, W_i) (line 08). By hypothesis there are no write() operations during $[T_i, T_i + \delta]$, thus $W_j = \emptyset$ and each correct server broadcasts the same set of values V_j. Since those servers are $\#echo$ then after δ time all non Byzantine servers collect $\#echo$ occurrences of all values in V_j. Thus all correct and cured servers set $V_c = V_j = [\langle v_0, sn_0\rangle, \langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle]$. $\square_{Lemma\ 3}$

Lemma 4 *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), (ii) $n \geq 2(k + 1)f + 1$. (iii) there are $\#echo$ server $s_j \in Co(T_i)$, Then $\forall s_c \in Cu(T_i)$, $s_c \in Co(T_i + \delta)$ and for every server $s_k \in Co(T_i + \delta)$, $\bigcap V_k \neq \perp$ in particular among the common values there is the last written value before T_i or the value belong to op_W such that $T_i \in [t_B(op_W), t_E(op_R)]$.*

Proof Let us start considering that at T_i there are $\#echo$ correct servers storing $V_i = [\langle v_0, sn_0\rangle, \langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle]$ and running the code in Figure 2. Each server broadcasts an ECHO() message whose content is conCut(V_i, W_i) (line 08). Let op_{W1} be a write() operation, such that $T_i \in [t_B(W1), t_E(W1)]$ and let $\langle v_3, sn_3\rangle$ be the value to be written. When a non Byzantine server delivers a WRITE() message $\langle v_3, sn_3\rangle \in W_i$ set (Figure 3 line 06). Since $T_i \in [t_B(W1), t_E(W1)]$, at the beginning of the maintenance() operation non all correct servers have $\langle v_3, sn_3\rangle \in W_i$. Thus, $\#echo$ servers in $Co(T_i)$ broadcast different values as result of conCut(V_i, W_i): $[\langle v_0, sn_0\rangle, \langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle]\}$ or $[\langle v_1, sn_1\rangle, \langle v_2, sn_2\rangle, \langle v_3, sn_3\rangle]$. At $T_i + \delta$ non Byzantine servers s_i select values occurring at least $\#echo$ times setting $V_i = [\langle v_1, 1\rangle, \langle v_2, 2\rangle]$.
At $T_i + \delta$ it may also happen that another write() operation, op_{W2} occurs. Let $\langle v_4, sn_4\rangle$ be the value written by op_{W_2} subsequent to op_{W1}, such that $T_i + \delta \in [\, t_B(op_{W2}), t_B(op_{W2})\,]$. In that case it may happen that all servers that were in $Co(t)$ and are now in $Co(t + \delta)$ delivers the WRITE$(\langle v_4, sn_4\rangle)$ message and no yet the servers that were in $Cu(T_i)$. So that the first group of servers is storing in $V \cup W$, $\{\langle v_1, 1\rangle, \langle v_2, 2\rangle, \{\langle v_3, 3\rangle\}$ and the second group is storing $\{\langle v_2, 2\rangle, \langle v_3, 3\rangle, \langle v_4, 4\rangle\}$. All of those servers are storing $\langle v_2, 2\rangle$ in common, which is the last written value respecting to T_i, concluding the proof.

$\square_{Lemma\ 4}$

Corollary 1 *The* maintenance() *operation guarantees that $\forall T_i, i \in \mathbb{N}$, $\forall s \in Cu(T_i)$, then $s \in Co(T_i + \delta)$.*

Definition 9 (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 10 ($Max\tilde{B}(t, t+T)$) *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 5 *If $\delta \leq \Delta < 3\delta$ and $T \geq \delta$ then $Max\tilde{B}(t, t+T) = (\lceil \frac{T}{\Delta} \rceil + 1)f$.*

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. $\square_{Lemma\ 5}$

Lemma 6 *Let op be a* read() *operation issued at time t and terminating at time $t+2\delta$. Let $Max\tilde{B}(t, t+2\delta)$ be the maximum number of servers that can be faulty for at least one time unit in the interval $[t, t+2\delta]$. If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) and (ii) $n \geq 2(k+1)f + 1$, then $|Co(t, t+\delta)| > |Max\tilde{B}([t, t+2\delta])| + |Cu(t)|$.*

Proof

- **Case 1 - $(\Delta S, CUM)$ with $2\delta \leq \Delta$.**

  Let us note that the maximum number of faulty servers in any interval $[t, t+2\delta]$ is strictly related to the Δ value. From Lemma 5, $Max\tilde{B}(t, t+2\delta) = (\lceil \frac{2\delta}{\Delta} \rceil + 1) \times f$. Considering that $\delta \leq \Delta < 2\delta$, we obtain $Max\bar{B}(t, t+2\delta) = 3f$.
  In addition to Byzantine servers, in the $(\Delta S, CUM)$ model also cured servers may send a reply. Let us consider that the maintenance() operation code run in δ time. At any T_i servers sends their value and after δ time the collected values are analyzed. Thus we can consider that δ time is enough to terminate the maintenance() operation). It follows that for each Byzantine server there can be only one server that is in the cured state (the one that was previously affected by the same agent) whose become correct before the agent affect another server. Thus there are, in the worst case, f more non correct servers that may reply. It follows that $|Cu(t)| \leq f$.

  The number of correct servers at time $t+\delta$ is given by the number of serves that are non-faulty in the whole interval ($n - Max\bar{B}(t, t+2\delta) - |Cu(t)| = f$) plus the number of server that were not correct at time t but that had "enough" time to terminate the maintenance operation before time $t+\delta$ (i.e., $Max\bar{B}(t, t+2\delta) - Max\bar{B}(t+\delta, t+2\delta)$). On the other side, if a sever s_i begins a read() operation in a cured state, then the agent left s_i at most $t - \delta + 1$. Thus, it can not move again before $t+\delta$. Thus the server that will be affected after $t+\delta$ is correct at $t+\delta$. So, there are as many servers being correct at t and faulty after as much as the servers in $|Cu(t)| = f$.
  Finally each servers in $|Cu(t)|$ has the time to became correct at time $t+\delta$ (for Corollary 1).

  Thus

$$|Co(t, t+\delta)| = n - (Max\bar{B}(t, t+2\delta) + |Cu(t)|) + Max\bar{B}(t, t+2\delta) - Max\bar{B}(t+\delta, t+2\delta) + 2 \times |Cu(t)|$$

$$|Co(t,t+\delta)| = n - |Cu(t)| - Max\bar{B}(t+\delta, t+2\delta) + 2 \times |Cu(t)|$$
$$|Co(t,t+\delta)| = n - Max\bar{B}(t+\delta, t+2\delta) + |Cu(t)|$$
$$|Co(t,t+\delta)| = 4f + 1 - 2f + f = 3f + 1$$

- **Case 2 -** $(\Delta S, CUM)$ **with** $\delta \leq \Delta < 2\delta$**.** Following the consideration done in Case 1, we obtain that $Max\bar{B}(t, t+2\delta) = 3f$ for $\delta \leq \Delta < 2\delta$ and also $|Cu(t) = f|$. Note that in this case, the presence of $|Cu(t)|$ does not implies an extra presence of the same amount of server that are correct at $t+\delta$ and then became faulty. Thus we have that:

$$|Co(t,t+\delta)| = n-(Max\bar{B}(t,t+2\delta)+|Cu(t)|)+Max\bar{B}(t,t+2\delta)-Max\bar{B}(t+\delta,t+2\delta)+|Cu(t)|$$
$$|Co(t,t+\delta)| = n - |Cu(t)| - Max\bar{B}(t+\delta, t+2\delta) + |Cu(t)|$$
$$|Co(t,t+\delta)| = n - Max\bar{B}(t+\delta, t+2\delta)$$
$$|Co(t,t+\delta)| = 6f + 1 - 2f = 4f + 1$$

From which the claim follows.

$\square_{Lemma\ 6}$

Considering the worst case scenario where each message sent to and by non correct servers is instantaneously delivered, while each message sent to and by correct servers needs δ time, from Lemma 6 the next corollary follows

Corollary 2 *Let op be a* read() *operation issued at time t and terminating at time $t+2\delta$. The number of replies sent by correct servers at some time $\tau \in [t, t+2\delta]$ is always greater than the number of replies sent by non correct servers.*

Definition 11 (write() **completion time t_{wE})** *Let* write() *be an operation op_W writing v on the register. t_{wE} is the time after which, if a* read() *operation occurs, there are always at least $\#reply$ correct servers that reply with v.*

For simplicity let us first prove that the algorithm as it has been presented in [5] works, even if we consider the three values result of conCut(V, W), instead of $V \cup W$, assuming we can order them from the oldest to the newest. The main difference is that in [5] all values in $V \cup W$ are ordered by sequential timestamp. This is not true in conCut(V, W), but those values can be univocally order from the oldest to the newest. Such proof is moved after.

Lemma 7 *Let op be a* write(v) *operation invoked by a correct client at time $t_B(op) = t$, then the write completion time $t_{wE} \leq t + 2\delta$.*

Proof To prove this Lemma we have to prove that by time $t_{wE} \leq t + 2\delta$, v is always in conCut(V, W) in $\#reply$ correct servers.
Due to the communication channel synchrony, WRITE messages are delivered by servers within the time interval $[t, t+\delta]$; any server $s_j \in Co(t, t+\delta)$ executes the correct algorithm code. Thus, when s_j delivers WRITE message it checks if the value is already stored (line 05, Figure 3), otherwise it executes line 06 storing the value in W_j and setting the associated epoch to 1.

Let us consider case $k = 2$. By time $tE(op_W)$ there are $n - Max\tilde{B}(t_B(op_W), t_E(op_W)) \geq \#reply$ servers able to reply with v to a read() operation. This is true up to the next Byzantine agent movement $T_i > tE(op_W)$, in other words, if $T_{i-1} \in [t_B(op_W), t_E(op_W)]$ then the hypothesis of Lemma 3 does not hold (there are no $\#echo$ servers having the same vector V, such that $v \in V$), so that at T_i there are at least $\#reply - f$ servers that can reply with messages whose content is v, whose are not enough. Since $\#reply - f = \#echo$ then at T_i there are enough correct servers that during the maintenance() that send v in the ECHO() message. So for Lemma 3 at T_{i+1} all correct servers are able to reply with v. What is left to prove is that $t_{wE} \in [T_i, T_{i+1}]$ and $t_{wE} \leq t + 2\delta$. During the write() operation there are at least $n - Max\tilde{B}(t_B(op_W), t_E(op_W)) \geq \#reply = 4f + 1$ servers always correct, $Co(t_B(op_W), t_E(op_W))$. Let $B(t_B(op_W), T_{i-1})$ the set of servers that missed the WRITE(v) message. Some servers in $Co(t_B(op_W), t_E(op_W))$ may deliver the WRITE() message before of after T_{i-1}, and thus send the WRITE_FW() message before of after T_{i-1}. In the first case the WRITE_FW() message can be lost as well, but v is also present in the result of conCut(V, W) (Figure 3 line 06) and sent at T_{i-1} in the ECHO() message (Figure 2 line 08) so that servers in $B(t_B(op_W), T_{i-1})$ deliver it at most at $T_i + \delta$. In the second case, the WRITE_FW() message is sent by servers in the time interval $[T_{i-1}, t_E(op)]$ (Figure 3 line 11). Since a message is delivered at most after δ time, is it true that at most at $t_E(op) + \delta = t + 2\delta$ any servers that missed the write() message has now enough occurrence of it in the $fw\_vals_i \cup echo\_vals_i$ set so that line 13 in Figure 3 by time $t + 2\delta$ is executed storing v in FW_i, which is sent back to any reading client, concluding the proof. If $k = 1$ the proof structure is similar. $\square_{Lemma\ 7}$

Considering Lemma 7 and that every time $t > t_{no\_tr}$ the function insert(V, v) is invoked, $\exists k \in \{0, 1, 2\}, V[k] = v$ (every branch of such function ends with an insertion), then the following Corollary holds.

Corollary 3 *Let op_W be a* write() *operation such that $t_B(op_W) > \tau_{no\_tr}$ and let v be the value to be written in the register. Then for every $s_i \in Co(t_B(op_W) + 2\delta)$, $v \in FW_i$ by time $t_B(op_W) + 2\delta$.*

Lemma 8 *Let op_W be a* write() *operation such that $t_B(op_W) > \tau_{no\_tr}$ and let v be the written value. Let $t_E w$ be its time completion and let T_i the time of the next Byzantine agent movement just after $t_E w$. Then if there are no other* write() *operation, the value written by op_W is stored by $\#reply$ servers forever.*

Proof The proof follows directly from Lemma 7 and considering that if there a no more write() operation $W = \emptyset$, so at every maintenance() operation there are at least $\#echo$ servers storing $v \in V$ so that $v \in$ conCut(V, W). $\square_{Lemma\ 8}$

Lemma 9 *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 register after τ_{stab}. Let us consider a generic op_{W_k} and let v be the written value by such operation and $t_E w_k$ be its completion time. Then v is in the register (there are $\#reply$ correct servers such that $v \in V$) 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 $t_E w$, is in the register forever.
- any new written value eventually is stored in vector V (cf. Figure 2 line 12 or line 15) whose dimension is three.
- write() operation occur sequentially.

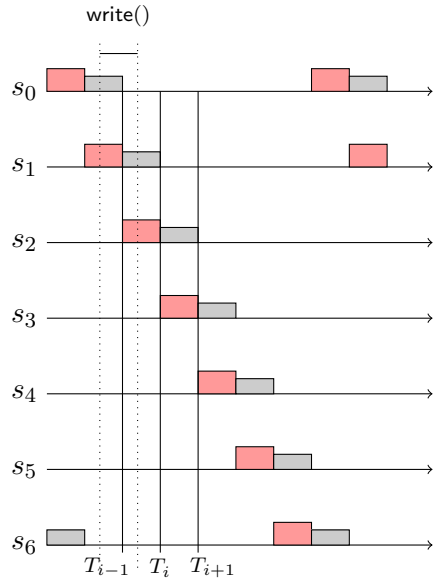

Figure 5: write() operation in a scenario where $\delta \leq \Delta < 2\delta$.

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

Let us now finally prove that the result of conCut(V, W) is compliant to the expected behavior that we would have from $V \cup W$ set of elements ordered with respect to sequential timestamps.

Lemma 10 *For each server s_j issuing a* maintenance() *operation op_{M_j}, such that $t_B(op_{M_j}) > \tau_{stab}$ and $s_j \notin B(t_B(op_{M_j}))$, $|FW_i| \leq 3$ and values in FW_i belong to a legal subsequence.*

Proof Consider that:

0 every write() operation, such that $t_B(op_{W_j}) > \tau_{no\_tr}$, terminates after δ time from its invocation (Figure 3 line 01-04);

1 for every write() operation op_{W_j}, such that v_j is the value to be written in the register and $t_B(op_{W_j}) > \tau_{no\_tr}$, then for every $s_i \in Co(t_B(op_{W_j}) + 2\delta)$, $v \in FW_i$ by time $t_B op_{W_j} + 2\delta$ (Corollary 3);

2 FW is analyzed and emptied at most any δ time during any maintenance() operation run by non Byzantine servers (Figure 2 lines 13 - 18), let us call such time interval M.;

3 the writer executes write() operations sequentially.

From point 1 it follows that during M there can be in FW_i a value v_j concerning a write() operation op_{W_j} issued before M such that $t_E(W_j) \notin M$ but $t_B(W_j) + 2\delta \in M$. Combining point 0 and 1 we have that given the write() operation $op_{W_{j-1}}$ then $t_B(W_{j-1}) + 2\delta \notin M$. Combining point 2 and 3, we have that the time interval M can be overlapped by at most two write() operations, let us name them op_{W_1} and op_{W_2}. Combining those results, in FW_i there can be values coming from op_{W_j}, op_{W_1} and op_{W_2}. Let us consider again point 1, it is possible to have in FW_i $op_{W_j}, op_{W_j}, op_{W_2}$ and op_{W_j}, op_{W_1} as well and all of them are a legal subsequence, which concludes the proof. $\square_{Lemma\ 10}$

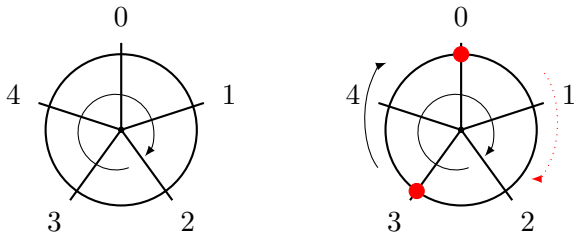

Figure 6: The left figure is a general representation of Z_5. The right figure shows that given two points there is only one possible direction, from 3 to 0 and not vice versa, since the distance between these can be at most 2.

Lemma 11 *For any servers $s_i \notin B(t), t > \tau_{stab}$, it is always possible to univocally order the elements in FW_i, from the oldest to the newest, with respect to their timestamp.*

Proof The proof follows considering that, the algorithm depicted in Figure 3 generates timestamps in a sequence and that in FW_i, for Lemma 10 there are at most 3 elements whose belong to a legal subsequence. Let $op_{W_k}, op_{W_{k+1}}, op_{W_{k+2}}$ be the three subsequent write() operations that respectively generate v_k, v_{k+1}, $v_{k+2} \in FW_i$ whose respective timestamps are $z, z +_5 1, z +_5 2$. Since those elements are sequentially generated then for each couple of them the difference between those timestamps is at most 2. Let us consider a couple of elements v_k and v_{k+2}, two cases are possible: (i) v_k has been generate before v_{k+2} or (ii) vice versa. Let us suppose that case (ii) is true, so that v_{k+2} has been generate before v_k. Computing $z -_5 z + 2 = 3$, which would mean that those two values belong to a sequence of four values, but in FW_i there is at most a sequence of three values (Lemma 10). It follows that case (1) is the only possible one, where $z + 2 -_5 z = 2$. Figure 6 provides a graphical representation of what has been presented, showing that there is an unique way to order a legal sequence of three elements, concluding the proof. $\square_{Lemma\ 11}$

Lemma 12 *For any $t > \tau_{stab}$, for any $s_i \in Co(t)$, W_i contains at most three values.*

Proof Considering that:

1. there is a new value v in W_i any time a WRITE() message is delivered from the writer issuing a write() operation op_W;
2. v is deleted from W_i when it is present in FW_i, during the maintenance() operation at the next check on FW_i, line 21 or line 32, Figure 2. So this check is performed at most any δ time;
3. for Corollary 3 $v \in FW_i$ at most by time $t_B(op_W) + 2\delta$;
4. write() operations are issued sequentially.

Combining point 1,2 and 3 a value v is removed from W_i at most by time $t_B(op_W) + 3\delta$. From point 4 it follows directly that there are no more than three values is W_i. To be more clearer, if there are four sequential write() operations $op_{W_1}, op_{W_2}, op_{W_3}, op_{W_4}$, since those are sequential, when op_{W_4} occurs, the value in W_i due to op_{W_1} is no more present in W_i. $\square_{Lemma\ 12}$

From Lemma 11 we have that if in a set there are at most three values belonging to a legal subsequence is always possible to order them, so that, considering Lemma 12 the same reasoning can be applied to W_i.

Corollary 4 *For any servers $s_i \notin B(t), t > \tau_{stab}$ it is always possible to univocally order the elements in W_i with respect to their timestamp.*

Lemma 13 *For any $t > \tau_{stab}$, the* CONCUT *function returns at most three values, such that in those values is it present the last written values before t and, if present, the concurrently written one.*

Proof The proof follows considering how the conCut function concatenate V_i and W_i and truncate it. For Corollary 4 is it possible to order elements in W_i. The same is trivially true for V_i since it is an ordered set. Thus the concatenation of those two lead to an ordered sequence of values, so that considering the last three of them implies that we are considering the last written values and if it is present also the concurrent one. $\square_{Lemma\ 13}$

Theorem 3 (ss-Validity) *There exists a time τ_{stab} such that each read operation invoked at time $t > \tau_{stab}$ 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 and the time interval $[t_B(op_R), t_B(op_R) + \delta]$, i.e., the first δ period of the read() operation. Since such operation lasts 2δ, the reply messages sent by correct servers, within the considered period, are delivered by the reading client. For Lemma 2, in such period there are $\#reply$ correct servers that sent back a reply message to the reading client. There is to prove that in those $\#reply$ there is at least one common value that is the last written value or the concurrently written one.
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 op_R happens after it, i.e., $t_E(op_W) \leq t_B(op_R)$, and let v be the last written value. From Lemma 7 and Lemma 8 after the write completion time there are $\#reply$ correct servers storing v in $V \cup W$ such that, for Lemma 13, it is returned by CONCUT(V_i, W_i). So the last written value is returned.
op_R **is concurrent with some** write() **operation**. During the $[t_B(op_R), t_B(op_R) + \delta]$ time interval there can be at most two write() operations. Thus for Lemma 9 and 13 the last written value before $t_B(op_R)$ is still present in $\#reply$ correct servers. Thus at least the last written value is returned. 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. Note that Byzantine servers may not force the reader to read another or older value since for Lemma 2 the number of correct replies is greater than the number of incorrect ones and because even if an older values has $\#reply$ occurrences the one with the highest sequence number is chosen. $\square_{Theorem\ 3}$

Basically we can say that thanks to the maintenance() operation and the forwarding mechanism, when a read() operation op_R begins at time $t_B(op_R)$, at time $t_B(op_R) + \delta$ there are $\#reply$ correct servers that reply with a value $v \in VVS(t_B(op_R))$.

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) $k\Delta \geq 2\delta$ (with $k \in 1, 2$) and (ii) $n \geq 2(k + 1)f + 1$, 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 2 and Theorem 3. $\square_{Theorem\ 4}$

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

Proof The proof simply follows considering that Theorems 2-3 proved that $\mathcal{P}_{Rreg}$ works with bounds provided in Table 1. Those match the previously known lower bounds [6] for the $(\Delta S, CUM)$ model. $\square_{Theorem\ 5}$

What is left to prove are the necessary conditions for the system to self-stabilize after τ_{no_tr}.

Lemma 14 *Let $t > \tau_{no\_tr}$, then after 3Δ for any server s_i, the effect of transient failures disappear in all variables but V_i and $echo\_val_i$.*

Proof After τ_{no_tr}, all non Byzantine servers s_i execute the correct code of the algorithm. So that at the beginning of every maintenance() operations FW_i is emptied and the EPOCHCHECK() function is invoked on W_i, $echo\_vals_i$ and fw_vals_i sets. So that values populate these sets for at most the time needed for two maintenance() operations. In fact, the epoch associated to each value has to be in the set $\{0, 1\}$ which is decreased by 1 at the beginning of every maintenance() operation. When $epoch \notin \{0, 1\}$ the element associated to it is deleted. Thus, in the worst case scenario *epoch* is set to 1. During the first maintenance() operation, it is decreased to 0. During the second maintenance() operation is decreased to 1. At the next one $epoch \notin \{0, 1\}$ and so is deleted. Follows that after 2Δ time those sets, W_i and fw_vals_i, are cleaned which is not true for $echo\_vals_i$ and V_i whose are populated at each maintenance() operation. Since the end of the transient failures is not aligned to the maintenance() operations we consider a Δ time more. $\square_{Lemma\ 14}$

Lemma 15 *Let $op_{W_1}, \ldots, op_{W_5}$ be a sequence of 5 consecutive* write() *operations, occurring after $\tau_{no\_tr} + 3\Delta$, then each servers $s_i \in Co(t)$, $t > \tau_{no\_tr} + 3\Delta + t_E(op_{W_5})$, is storing V_i in a legal state and populated only by* write() *operations issued after τ_{no_tr}.*

Proof From Lemma 14, after $\tau_{no\_tr} + 3\Delta$ all variables but V_i and $echo\_vals_i$, for every s_i not Byzantine, are cleaned from the effect of transient failures. So that in order to have a stabilized system V_i has to be completely populated with values belonging to correctly invoked write() operations, i.e., operation invoked after τ_{no_tr}. Since V_i by definition contains 3 values, then at most 3 write() operations are necessary. Is it to prove that at most two extra write() operations can occur. At the beginning of any maintenance() operation is invoked the function $check(V_i)$. So that this set can be in one of the following states (for simplicity we represent each element by its timestamp ts and we omit the modulo operation [2]):

a. $V_i = ts_k, ts_{k+1}, ts_{k+2}$;

b. $V_i = ts_k, ts_{k+1}, ts_{k+3}$;

c. $V_i = \perp, \perp, \perp$;

d. $V_i = \perp, \perp, ts_k$;

e. $V_i = \perp, ts_k, ts_{k+1}$;

f. $V_i = \perp, ts_k, ts_{k+2}$;

[2] $ts_k, ts_{k+1}, \ldots$ in the extended form is $ts_k, ts_{k+_5 1}, \ldots$.

Let us consider the following legal sequence: $ts_{k-1}, ts_{k+1}, ts_k, ts_{k+2}, \ldots$ [3] and the state (d.) $V_i = \perp, \perp, ts_k$. The result of the invocation of INSERT on V_i and ts_{k-1} is $V_i = \perp, ta_{k-1}, ts_k$. The value after, ts_{k+1} produces the following legal state $V_i = ts_{k-1}, ts_k, ts_{k+1}$. Finally the value after ts_k, since it is already in V_i produces the following state $V_i = \perp, \perp, ts_k$. The hereafter values to be inserted belong to a legal sequence as the value in V_i, so that after two more write() operations V_i is in a legal sequence and all the next values are legal value. To generalize, after the third write() operation, if V_i contains only values coming from a legal sequence (i.e., $\perp$ has never been inserted in the previous three insertions), then all the next values are legal values, since the values in V_i belong to the same legal sequence. This mean that there can not be more than two extra write() operation in addiction to the three necessary ones.
From now on, values in V_i and FW_i are consecutive elements of a legal sequence. So that every older element in FW_i is a legal value for V_i, in other words, any time that a new value has to be inserted in V_i such operation succeed, which is true for each correct server. $\square_{Lemma\ 15}$

5 Concluding remarks

We proposed a self-stabilizing regular register emulation in a network where both arbitrary transient faults and mobile Byzantine faults can occur, and where processes and Byzantine agent moves are decoupled. Our solution improves the existing work considering mobile Byzantines faults [6, 4] in several key aspects: *(i)* it is the first self-stabilizing regular register implementation in round-free synchronous communication model, and*(ii)* it uses bounded timestamps from the $\mathcal{Z}_5$ domain. All these improvements have no additional cost with respect to the number of replicas that are necessary to tolerate f mobile Byzantine processes: that is, our solution is optimal with respect to established lower bounds [6]. Additionally, the convergence time of our solution is upper bounded by $3\Delta + T_{5write()}$, where $T_{5write()}$ is the time needed to execute five *complete* $write()$ operations, each $write()$ operation completing in finite time.

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.

[3] Is it enough to swap ts_{k+1} and ts_k to obtain values generated by a sequence of write() operations

References

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

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

[3] 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.

[4] 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.

[5] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal Mobile Byzantine Fault Tolerant Distributed Storage. Research report, UPMC - Université Paris 6 Pierre et Marie Curie, July 2016.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.