Optimal Mobile Byzantine Fault Tolerant Distributed Storage

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

Optimal Mobile Byzantine Fault Tolerant Distributed Storage

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, antonella.del-pozzo, sebastien.tixeuil}@lip6.fr

February 14, 2017

Abstract

We present an optimal emulation of a server based regular read/write storage in a synchronous round-free message-passing system that is subject to mobile Byzantine failures and prove that the problem is impossible to solve in asynchronous settings. In a system with n servers implementing a regular register, our construction tolerates faults (or attacks) that can be abstracted by agents that are moved (in an arbitrary and unforeseen manner) by a computationally unbounded adversary from a server to another in order to deviate the server's computation. When a server is infected by an adversarial agent, it behaves arbitrarily until the adversary decides to "move" the agent to another server. We investigate the case where the movements of the mobile Byzantine agents are decided by the adversary and are completely decoupled from the message communication delay. Our emulation spans two awareness models: servers with and without self-diagnosis mechanism. In the first case servers are aware that the mobile Byzantine agent has left and hence they can stop running the protocol until they recover a correct state while in the second case, servers are not aware of their faulty state and continue to run the protocol using an incorrect local state. Our results, proven optimal with respect to the threshold of the tolerated mobile Byzantine faults in the first model, are significantly different from the round-based synchronous models. Another interesting side result of our study is that, contrary to the round-based synchronous consensus implementation for systems prone to mobile Byzantine faults, our storage emulation does not rely on the necessity of a core of correct processes all along the computation. That is, every server in the system can be compromised by the mobile Byzantine agents at some point in the computation. This leads to another interesting conclusion: storage is easier than consensus in synchronous settings, when the system is hit by mobile Byzantine failures.

1 Introduction

Byzantine fault tolerance is at the core of Distributed Computing and a fundamental building block in any reasonably sized distributed system. Byzantine failures encompass all possible cases that can occur in practice (even unforeseen ones) as the impacted process may simply exhibit arbitrary behavior. Specifically targeted attacks to compromise processes and/or virus infections can indeed cause malicious code execution.

In classical Byzantine fault-tolerance, attacks and infections are typically abstracted as an upper bound f on the number of Byzantine processes that a given set of n processes should be able to tolerate. Such bounds permit to characterize the solvable cases for benchmarking problems in Distributed Computing (*e.g.* Agreement and Register Emulation). However, this abstraction fails the reality test of long-lived distributed services. With new exploits being publicized daily and hackers offering services at amazingly low prices, *every* process is bound to be compromised in a long lasting execution. On the light side, dedicated cure and software rejuvenation techniques increase the possibility that a compromised node *does not remain compromised forever*, and may be aware about its previously compromised status [15].

To integrate both aspects, Mobile Byzantine Failures (MBF) models have been introduced. Then, faults are represented by Byzantine agents that are managed by a powerful omniscient adversary that "moves" them from a process to another. Mobile Byzantine failures that have been investigated so far consider round-based computations, and can be classified according to Byzantine mobility constraints: *(i)* Byzantines with constrained mobility [5] may only move from one process to another when protocol messages are sent (similarly to how viruses would propagate), while *(ii)* Byzantines with unconstrained mobility [1, 3, 6, 11, 12, 13] may move independently of protocol messages.

Buhrman *et al.* [5] studied the problem of Agreement when Byzantines have constrained mobility. In the case of unconstrained mobility, several variants were investigated, still for the Agreement problem [1, 3, 6, 11, 12, 13]. Reischuk [12] considers that malicious agents are stationary for a given period of time. Ostrovsky and Yung [11] introduce the notion of mobile viruses and define the adversary as an entity that can inject and distribute faults. Garay [6], and more recently Banu *et al.* [1], and Sasaki *et al.* [13] 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 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 [1, 3, 6, 13] lies in the knowledge that processes have about their previous infection by a Byzantine agent. In Garay's model [6], a process is able to detect its own infection after the Byzantine agent left it. More precisely, during the first round following the leave of the Byzantine agent, a process enters a state, called *cured*, during which it can take preventive actions to avoid sending messages that are based on a corrupted state. Garay [6] proposed, in this model, an algorithm that solves Mobile Byzantine Agreement provided that $n > 6f$. This bound was later dropped to $n > 4f$ by Banu *et al.* [1]. Sasaki *et al.* [13] investigated the same problem in a model where processes do not have the ability to detect when Byzantine agents move, and show that the bound raises to $n > 6f$. Finally, Bonnet *et al.* [3] considers an intermediate setting where cured processes 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); this subtle difference on the power of Byzantine agents has an important impact on the bounds for solving agreement: the bound becomes $n > 5f$ and is proven tight.

Traditional solutions to build a Byzantine tolerant storage service (*a.k.a.* register emulation) can be divided into two categories: *replicated state machines* [14], and *Byzantine quorum systems* [2, 8, 10, 9]. 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, 6, 13].

Our Contribution. The main motivation for our work comes from realizing that the hypothesis that Byzantine agent moves are tightly synchronized with protocol rounds is not a realistic assumption, when Byzantine agents are driven by an adversary that can make use of out of band resources for coordinating them. Indeed, infection and cure are independent from the protocol that is executed on the servers, and typically result from external actions.

Our first contribution (Section 3) is to propose and formalize a general mobile Byzantine model, where Byzantine agent movements are decoupled from the protocol computation steps (in particular, movements of the Byzantine agents are no more related to messages that are exchanged through the protocol). We explore the fundamental implications of the relaxed hypothesis about Byzantine agent movements, and nevertheless retain the dimension related to process awareness about its failure state.

The second contribution of the paper is a protocol to emulate a regular register in our general mobile Byzantine model. We first explore (Section 4) the instances of the model where this problem is solvable (*e.g.* we provide impossibility results for the asynchronous setting), and in the solvable cases, we present and prove protocols (Section 5 and Section 6) whose resilience is optimal with respect to the number of Byzantine agents.

2 System Models

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, 5, 6, 11, 12, 13].

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).

Timing Assumptions. We will consider two types of systems: (i) asynchronous (see Section 4.2) and (ii) round-free synchronous (see Sections 5 and 6).

The system is *asynchronous* in the sense that there not exists any upper bound on communication and computation latencies. As a consequence, messages are delivered but it is not possible do compute any upper bounds on their delivery time. 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.

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 $\hat{H}$.*

3 Adversary Model

The Mobile Byzantine Failure (MBF) models considered so far in literature [1, 3, 5, 6, 11, 12, 13] 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.

3.1 Mobile Byzantine Models for round-based computations

In all the above cited works the system evolves in synchronous rounds. Every round is divided in three phases: (i) *send* where processes send all the messages for the current round, (ii) *receive* where processes receive all the messages sent at the beginning of the current round and (iii) *computation* where processes process received messages and prepare those that will be sent in the next round. Concerning the assumptions on agent movements and servers awareness on their *cured* state the Mobile Byzantine Models defined in [3, 6, 5, 13] are summarized as follows:

- *Garay's model* [6]. In this model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). When a server is in the

cured state it is aware of its condition and thus can remain silent for a round to prevent the dissemination of wrong information.

- *Bonnet et al.'s model* [3] and *Sasaki et al.'s model* [13]. As in the previous model, agents can move arbitrarily from a server to another at the beginning of each round (i.e. before the send phase starts). Differently from the Garay's model, in both models it is assumed that servers do not know if they are correct or cured when the Byzantine agent moved. The main difference between these two models is that in the [13] model a cured process still acts as a Byzantine one extra round.

- *Buhrman's model* [5]. Differently from the previous models, agents move together with the message (i.e., with the send or broadcast operation). However, when a server is in the *cured* state it is aware of that.

Most of the previously cited models [1, 3, 5, 6, 13] consider that the Byzantine agents mobility is related to the round-based synchronous system communication. That is, processes execute synchronous rounds composed of three phases: send, receive, compute. Only between two consecutive rounds, Byzantine agents are allowed to move from one node to another. In the sequel we formalize and generalize the MBF model. Our generalization is twofold: (i) we decouple the Byzantine agents movement from the structure of the computation making it round-free and hence suitable for any distributed application and (ii) we model the infection diffusion in relation with the detection/recovery capabilities of servers.

Informally, in the MBF model, when a Byzantine agent is hosted by a process, the adversary takes the entire control of the process making it Byzantine faulty (i.e., it can corrupt the process's local variables, forces the process to send arbitrary messages etc...). Then, the Byzantine agent moves away and it leaves the process with a possible corrupted state (i.e., in cured state). Such movement abstracts, for example, a virus that has been detected and putted in quarantine or a software update/patching of the machine.

As in the case of round-based MBF models [1, 3, 5, 6, 13], 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., $B([t, t']) = \bigcap_{\tau \in [t,t']} B(\tau)$).*

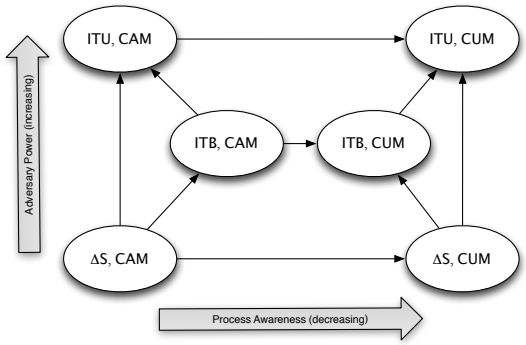

Figure 1: MBF model instances for round-free computations and their relations.

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., $Cu([t, t']) = \bigcap_{\tau \in [t,t']} Cu(\tau)$).*

3.2 Mobile Byzantine Models for round-free computations

We will discuss further how the MBF model for round-free computations introduced informally in the previous section is able to abstract different attack scenarios mentioned in Section 1. In addition, we will classify its six possible instances (see Figure 1) according to the adversary power (from the weakest adversary model $(\Delta S, CAM)$ to the strongest one (ITU, CUM)) showing the relationships between them.

Our model takes into account two different attack dimensions: (i) how the external adversary can coordinate the movement of the Byzantine agents and (ii) the process awareness about their current failure state. The first point abstracts the capability of the external adversary to propagate the infection with respect to the detection and recovery capability of processes while the second point distinguishes between detection and proactive recovery capabilities. Thus, any instance of our MBF model is characterized by a pair (X, Y), where X represents the coordination aspect (i.e., one among ΔS, ITB and ITU) and Y represents the process awareness (i.e., CAM vs. CUM).

The coordination dimension allows to characterize the infection spreading from a time point of view. In particular:

- $(\Delta S, *)$ 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 $i \in \mathbb{N}$).

- $(ITB, *)$ slightly relaxes the assumption about the time of the infection propagation. In particular, in this case the Byzantine agents may affect different servers for different periods

of time. This abstracts in some way the possible different complexities of various attack steps (each mobile agent can do a set of exploits and each exploit may take different time to succeed and then to be detected). As a consequence, we are able to capture possible differences in the detection and the rejuvenation times that are now different from server to server. More formally, each of the f Mobile Byzantine Agent ma_i is forced to remain on a process for at least a period Δ_i. Given two mobile Byzantine Agents ma_i and ma_j, their movement periods Δ_i and Δ_j may be different.

- $(ITU, *)$ further relaxes the coordination assumption and allows to consider extremely fast infection and detection/rejuvenation processes. More formally, each Mobile Byzantine agent ma_i is free to move at any time (i.e., it may occupy a process for one time unit, corrupt its state and then leave). This case can be seen as a particular case of ITB where $\Delta_i = 1$ for each mobile agent ma_i.

Let us note that, obviously, $(\Delta S, *)$ is the more restrictive coordination case with respect to the adversary power while $(ITU, *)$ represents the maximum freedom (from the coordination point of view) for the external adversary.

Example of $(\Delta S, *)$ coordination, $(ITB, *)$ coordination and $(ITU, *)$ coordination are shown respectively in Figure 2, Figure 3 and Figure 4 (where ma_i denotes the mobile Byzantine agents).

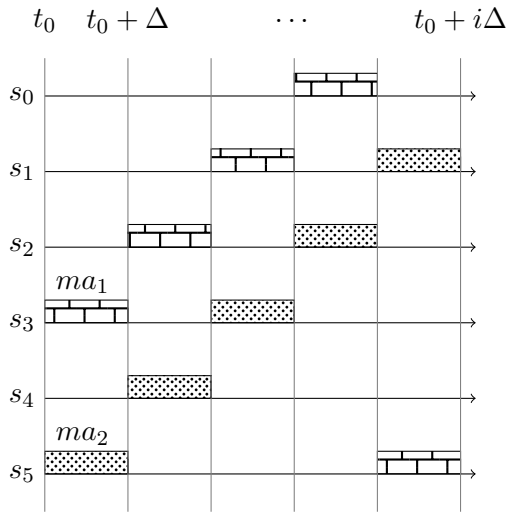

Figure 2: Example of a $(\Delta S, *)$ run with $f = 2$.

The awareness dimension allows to distinguish between servers under continuous monitoring from the non-monitored ones. Monitored systems are, in fact, characterized by detection and reaction capabilities that enable them to detect their failure state and to act accordingly. On the contrary, non-monitored servers have no self-diagnosis capabilities but they can try to prevent infections by adopting pessimistic strategies that include proactive rejuvenation. In particular:

- $(*, CAM)$ is able to capture scenarios where servers are aware of a past infection as they abstract environments characterized by the presence of monitors (e.g., antivirus, Intrusion Detection System etc...) that are able to detect the infection and notify the server when the threat is no more affecting the server.

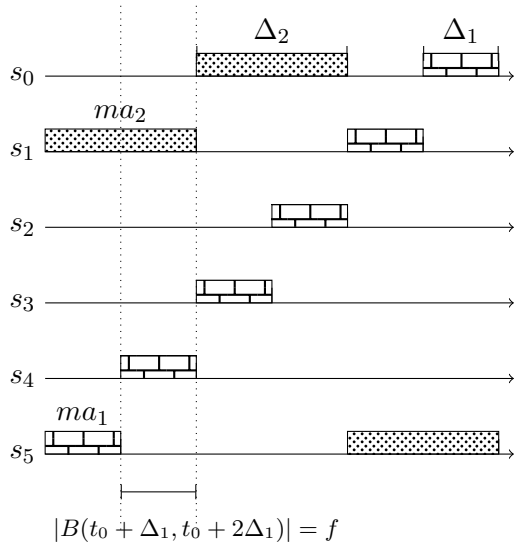

Figure 3: Example of a $(ITB, *)$ run with $f = 2$.

- $(*, CUM)$ 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.

It is easy to prove that CAM is a stronger awareness condition with respect to CUM and thus represents a restriction over the adversary power.

Combining together a type of movement and one of the two awareness conditions, we obtain six different instances of our MBF model for round-free computations, see Figure 1. The instance $(\Delta S, CAM)$ is the strongest one as it is the more restrictive for the external adversary and it provides cured processes with the highest awareness while the instance (ITU, CUM) represents the weakest model as it considers the most powerful adversary and provides no awareness to cured processes.

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, all servers may be affected by a Byzantine agent (i.e., none of the server is guaranteed to be correct forever). In order to abstract the knowledge a server has on its state (i.e. *cured* or *correct*), we assume the existence of a cured_state oracle. When invoked via report_cured_state() function, the oracle returns, in the CAM model, true to *cured* servers and false to others. Contrarily, the cured_state oracle returns always false in the CUM model.

[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.

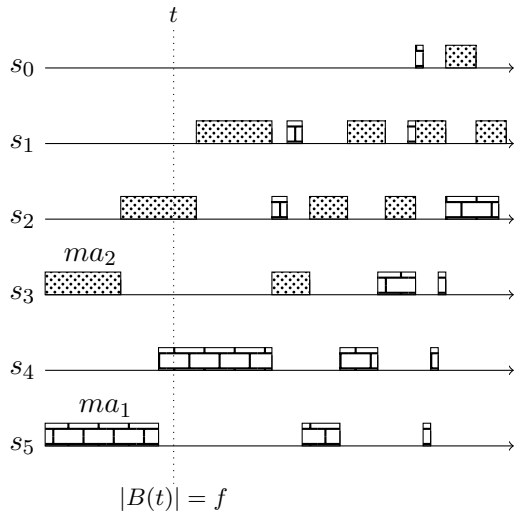

Figure 4: Example of a $(ITU, *)$ run with $f = 2$.

The implementation of the oracle is out of scope of this paper and the reader may refer to [11] for further details.

4 Registers in MBF model

4.1 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). The register state is maintained by the set of servers $\mathcal{S}$. Every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundaries: an *invocation* event and a *reply* event. These events occur at two time instants (i.e., invocation time and the reply time) according to the fictional global time.
An operation *op* is *complete* if both the invocation event and the reply event occurred (*i.e.*, the client issuing the operation does not crash between the invocation time and the reply time). Then, an operation *op* is *failed* if it is invoked by a process that crashes before the reply event occurs.

Given two operations *op* and op', their invocation times ($t_B(op)$ and $t_B(op')$) and reply times ($t_E(op)$ and $t_E(op')$), we say that *op precedes* op' ($op \prec op'$) if and only if $t_E(op) < t_B(op')$. If *op* does not precede op' and op' does not precede *op*, then *op* and op' are *concurrent* (noted $op||op'$).
Given a write(v) operation, the value v is said to be written when the operation is complete.
In this paper, we consider a single-writer/multi-reader (SWMR) regular register [7] specified as follows:
— **Termination**: if a correct client invokes an operation, it eventually returns from that operation (i.e., every operation issued by a correct client eventually terminates).
— **Validity**: A read() operation returns the last value written before its invocation (*i.e.* the value written by the latest completed write() preceding it), or a value written by a concurrent write() operation.

Our impossibility results (reported in the next section) are proven for the case of *s*afe register (weaker than the regular register in the Lamport's hierarchy [7]). A read operation on a safe register concurrent with a write operation may return any value in the register domain.

We will consider in the sequel only execution histories related to the register computation. In particular, the set of relevant computation events H will be defined by the set of all the operations issued on the register and the happened-before relation will be substituted by the precedence relation $\prec$ between operations. Thus, we will consider a *register execution history* specified as $\hat{H}_R = (H, \prec)$.

From the specification above, we can define the notion of *valid value at time t* as follow:

Definition 6 (Valid Value at time t) *Let $\hat{H}_R = (H, \prec)$ be a register execution history of a regular register $\mathcal{R}$. A* valid value at time t *is any value returned by a fictional* read() *operation on the register $\mathcal{R}$ executed instantaneously at time t.*

A protocol $\mathcal{P}_{reg}$ is a collection of distributed algorithms implementing basic register operations (i.e., $\mathcal{P}_{reg} \subseteq \{\mathcal{A}_R, \mathcal{A}_W\}$ where $\mathcal{A}_R$ is the algorithm implementing the read() operation and $\mathcal{A}_W$ is the algorithm implementing the write() operation). We will say that $\mathcal{P}_{reg}$ *is correct with respect to its specification* if it implements a register satisfying the specification.

4.2 Impossibility results

In this section we prove that, contrary to the static Byzantine tolerant implementations of registers, in the case of MBF tolerant implementations a new operation, namely maintenance(), must be implemented to prevent servers from losing the current register value. Then, we will show that in an asynchronous system and in the presence of single Mobile Byzantine Agent, there is no protocol $\mathcal{P}_{reg}$ implementing a safe register and consequently a regular register.

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 safe register in any of the MBF models for round-free computations.*

Proof Let us suppose by contradiction that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ is a correct protocol implementing a 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 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 will 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, $\mathcal{A}_R$ has no way to read a valid value and the validity property is violated. It follows that $\mathcal{P}_{reg}$ is not correct and we have a contradiction. $\square_{Theorem\ 1}$

From Theorem 1 it follows that, in presence of Mobile Byzantine Agents, a new operation must be defined to allow cured servers to restore a valid state and avoid the loss of the register value.

Definition 7 (maintenance() and $\mathcal{A}_M$) *A* maintenance() *operation is an operation that, when executed by a process p_i, terminates at some time t leaving p_i with a valid state at time t (i.e., it guarantees that p_i is correct at time t). A maintenance algorithm $\mathcal{A}_M$ is an algorithm that implements the* maintenance() *operation.*

As a consequence, any correct protocol $\mathcal{P}_{reg}$ must include one more algorithm implementing the maintenance() operation[2] so that the corollary follows:

Corollary 1 *Let n be the number of servers emulating a register and let f be the number of Mobile Byzantine Agents in the system. That is, if $f > 0$ then any correct protocol $\mathcal{P}_{reg}$ implementing a register in the round-free Mobile Byzantine Failure model must include an algorithm $\mathcal{A}_M$ (i.e., $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$).*

Corollary 2 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in the $(\Delta S, CAM)$ Mobile Byzantine Failure Model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_M$ must involve at least one communication step.*

Proof The claim simply follows by considering that cured servers have a compromised state thus, they need to receive information from correct servers in order to be able to update their state to a valid one. $\square_{Lemma\ 2}$

4.3 Impossibilities in Asynchronous System

Lemma 1 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in the $(\Delta S, CAM)$ MBF model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_W$ and $\mathcal{A}_R$ must involve at least one* send − reply *(resp.* request − reply*) communication pattern (i.e., two communication steps).*

Proof Let us recall that read() and write() operations are issued by clients and that the set of clients $\mathcal{C}$ and the set of servers $\mathcal{S}$ maintaining the register are disjoint. As a consequence, when a client c_i wants to write a new value v in the register, it has necessarily to propagate it in the server set. The same happens when a client c_j wants to read: it has to ask servers the most up-to-date value. It follows that a send (request) communication step is necessary.

Let us now show that the send communication step is not sufficient to provide a correct implementation of $\mathcal{A}_W$ and $\mathcal{A}_R$.

In oder to be correct, $\mathcal{A}_W$ must ensure the *termination* property. As a consequence, c_i must be able to decide when it can trigger the write_return event. In particular, this can be done when at least one correct server[3] updated its internal state.

Let us recall that (i) processes communicate only by exchanging messages, (ii) clients (and in particular the writer) do not know the failure state of servers and (iii) the system is asynchronous.

[2]Let us note that such an operation can also be embedded in the other algorithm. However, for the sake of clarity, we will consider here only protocols where valid state recovery is managed by a specific operation.

[3]The exact number of processes is given by the implementation of $\mathcal{A}_W$ algorithm. In any case, such number does not affect the proof and it must be at least one.

As a consequence, the only way c_i has to know that at least one server s_j updated its state is to wait for an acknowledgement from s_j. As a consequence, a second communication step, i.e., a reply step, is necessary for a correct implementation of $\mathcal{A}_W$.

The same reasoning applies for the termination of the read() operation and the claim follows.

$\square_{Lemma\ 1}$

Lemma 2 *Let n be the number of servers emulating a safe register and let f be the number of Byzantine agents in the $(\Delta S, CAM)$ Mobile Byzantine Failure model. Let t be a time instant at which f servers are faulty and f other servers are cured. Let op be a* maintenance() *operation issued at time t. There does not exist a maintenance algorithm $\mathcal{A}_M$ able to terminate in asynchronous settings leaving cured servers with a valid state.*

Proof Consider an arbitrary cured server s in the set of cured servers that triggers a maintenance operation *op*. Assume that *op* is implemented by an algorithm $\mathcal{A}_M$ in asynchronous settings. Two cases may happen: (i) there is no write() operation concurrent with *op* or (ii) there is at least one concurrent write() operation.

- **Case 1:** $\nexists$ write$(v)||op$. Considering that no write() is concurrent with *op*, the only way s has to come back to be correct is to get the valid value from correct servers. As a consequence, every $\mathcal{A}_M$ must include a communication step where correct servers send their stored value to the cured server s (see Lemma 2). Let us recall that the system is asynchronous; thus it is not possible to bound, a priori, the time needed by such messages to reach s. In addition, s is aware just about its failure state but it is not aware about other failure states (in other words, s cannot know, for any time t, the sets $Co(t)$ and $B(t)$).

  As a consequence, the termination condition of $\mathcal{A}_M$ will depend on messages delivered by s and coming from other servers. Let us recall that cured servers have a non-valid state and, in order to terminate, $\mathcal{A}_M$ must be able to decide a valid value to update the state of the cured server.

  Thus, the termination condition of $\mathcal{A}_M$ must be able to select a valid value by considering all the information received by s.

  Let us now show that due to the Byzantine agents movement and the asynchrony of the communication, we can always have an indistinguishability situation between valid values and non valid values.

  The indistinguishability comes from the following observations:

  1. in every time interval $[t_0 + j\Delta, t_0 + (j+1)\Delta]$ (with $j \in \mathbb{N}$) the number of correct servers sending valid values is $n-(j+1)f$. In fact, at any movement, the adversary may decide to move the Byzantine agents on a totally disjoint set of servers (corrupting each time f new servers) until everyone is corrupted. Where t_0 is the initial time where f servers and all the other servers where correct.
  2. messages may take an arbitrary time to reach their destination and, in the worse case, all the messages sent in a long time period may be delivered at the same time and not following the FIFO order.

3. when a server is affected by the Byzantine agents, it can send an arbitrary number of messages with an arbitrary content. In particular, given the sequence of messages sent by a server before its compromising, such sequence can be permuted and sent again creating a symmetry condition.

As a consequence, each time that s evaluates a set of messages, it can always have a symmetric set and it will be forced to wait forever. Hence, the maintenance operation never terminates which contradicts the assumption. The same scenario may happen for every cured server starting a maintenance() operation and there is a time t' such that $Co(t') = \emptyset$ and none of the maintenance() operation will terminate.

- **Case 2:** $\exists$ write$(v)||op$. Due to the asynchrony of the system, every write() operation may be completed by interacting with servers always in the time period in which they are faulty. As a consequence, the resulting computation is equivalent to the one in which the write() never happened, we fall down in the previous case and the claim follows.

$\square_{Lemma\ 2}$

Theorem 2 *Let n be the number of servers emulating the register and let f be the number of Byzantine agents in the $(\Delta S, CAM)$ Mobile Byzantine Failure model. If $f > 0$, then there exists no protocol $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ implementing a safe register in a asynchronous system.*

Proof Let us consider the time t_0 at which the distributed computation starts. At time t_0, we have that f servers are affected by the mobile Byzantine agents (i.e., $|B(t_0)| = f$) while all the others are correct (i.e., $|Co(t_0) = n - f$).

At time $t_0 + \Delta$, the adversary moves mobile agents and, in the worse case, such agents affect a set of f servers completely disjoint from the previous one. Thus, in the computation we have f servers controlled by the Byzantine agents ($|B(t_0 + \Delta)| = f$) , f different servers entering in the cured state ($|Cu(t_0 + \Delta)| = f$)) and $n - 2f$ correct processes ($|Co(t_0 + \Delta)| = n - 2f$). Let us recall that, by assumption, cured servers know about their state (see CAM model) and thus they can start executing the maintenance operation running the maintenance algorithm $\mathcal{A}_M$. Each of the cured servers at time $t_0 + \Delta$, s, will start a maintenance() operation. Following Lemma 2 there is no $\mathcal{A}_M$ maintenance algorithm able to terminate leaving s with a valid state under asynchronous communication model. As a consequence, the value of the register is lost and no client is able to return a valid value of the register.

$\square_{Theorem\ 2}$

Note that the above result extends to any register specification and to any MBF instance defined in Section 2 since $(\Delta S, CAM)$ is the weakest adversary and safe is the weakest specification.

4.4 Lower bounds

Before to begin with lower bounds let us prove the following auxiliary Lemma.

Lemma 3 *Let t be the time at which server s_i enters the cured set. There exists no maintenance algorithm $\mathcal{A}_M$ that guarantees s_i to become correct before $t + \delta$.*

Proof The proof simply follow from the proof of Lemma 2, which commands that at least one communication step completes. Being in a synchronous system, such a step requires at most δ time to complete. $\square_{Lemma\ 3}$

In Sections 4.5 and 4.6, we consider the following two possibilities for Δ: *(i)* $\delta \leq \Delta < 2\delta$, and *(ii)* $2\delta \leq \Delta < 3\delta$, both for the $(\Delta S, CAM)$ and the $(\Delta S, CUM)$ models. In the following, we consider lower bounds with respect to the number of correct servers to implement a safe register. Lower bounds for safe registers trivially extend to any stronger register, as those give the weakest set of guarantees in Lamport's hierarchy [7]. For each of the cases we consider, we assume that a read() operation with no concurrent write() operation occurs. Now, each read() operation consists of two phases: in the first one, a client requests the current value to all servers, and in the second one, the client collects replies from servers. The necessity for these two steps can be proved following a similar reasoning as in Lemma 1. In each scenario, we assume that each message sent to or by faulty servers is instantaneously delivered, while each message sent to or by correct servers requires δ time. Without loss of generality, let us assume that all faulty servers send the same value and send it only once, for each period where they are faulty. Moreover, we make the assumption that each cured server (in the CAM model) does not reply as long as it is cured. Yet, in the CUM model, it behaves similarly to faulty servers, with the same assumptions on message delivery time. We suppose there exists (thanks to Lemma 3) a maintenance() protocol that guarantees cured servers to become correct within δ time. In the sequel, m_{s_j} denotes the message m sent by s_j.

4.5 $\delta \leq \Delta < 2\delta$, $(\Delta S, CAM)$ and $(\Delta S, CUM)$ models.

Theorem 3 *If $\delta \leq \Delta < 2\delta$, $\gamma \leq \delta$ and $n \leq 5f$, then there exists no protocol $\mathcal{P}_{reg}$ that implements a safe register in the $(\Delta S, CAM)$ model.*

Proof Let us suppose for the purpose of contradiction that such a protocol $\mathcal{P}_{reg}$ exists. Suppose now that $\mathcal{P}_{reg}$ implements a read() operation whose duration is exactly 2δ. When a client c_i invokes a read() operation *op*, c_i successfully reads the register value (*i.e.*, the valid value at $t_B(op)$), by the safe register validity property in the scenario with no concurrent write() operation. In such a scenario (see Figure 5, where we depicted in red and in green the time in which a server is affected and in a cured state respectively), let us consider a first execution E_1 in which the register valid value is 1 and c_i invokes *op*. We also consider one mobile Byzantine agent. Now, if each faulty server replies with 0, then c_i collects: $\{1_{s_0}, 0_{s_1}, 0_{s_2}, 1_{s_3}, 0_{s_3}, 1_{s_4}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 1. Let us now consider a second execution E_0 in which the register stores 0, and c_i invokes *op*. If each faulty server replies with 1, then c_i collects: $\{0_{s_0}, 1_{s_1}, 1_{s_2}, 0_{s_3}, 1_{s_3}, 0_{s_4}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 0. In both executions E_1 and E_0, c_i collects the same set of replies, yet gives two different values, hence a contradiction.

We now extend the indistinguishability argument to longer durations for $\mathcal{P}_{reg}$ completion. Assume $\mathcal{P}_{reg}$ implements a read() operation has duration 3δ. In such a case (see Figure 6), the previous executions E_1 and E_0 evolve in E'_1 (in which c_i collects: $\{1_{s_0}, 0_{s_1}, \mathbf{1_{s_1}}, 0_{s_2}, 1_{s_3}, 0_{s_3}, 1_{s_4}, \mathbf{0_{s_4}}\}$) and E'_0 (in which c_i collects: $\{0_{s_0}, 1_{s_1}, \mathbf{0_{s_1}}, 1_{s_2}, 0_{s_3}, 1_{s_3}, 0_{s_4}, \mathbf{1_{s_4}}\}$), respectively. Again, since there are not concurrent write() operations, due to the safe register validity property, c_i return two different values in the two executions, although the replies c_i gets are the same in both cases. The case

of duration 4δ is similar. The previous executions E'_1 and E'_0 evolve in E''_1 (in which c_i collects: $\{1_{s_0}, \mathbf{0_{s_0}}, 0_{s_1}, 1_{s_1}, 0_{s_2}, \mathbf{1_{s_2}}, 1_{s_3}, 0_{s_3}, 1_{s_4}, 0_{s_4}\}$) and E''_0 (in which c_i collects: $\{0_{s_0}\mathbf{1_{s_0}}, 1_{s_1}, 0_{s_1}, 1_{s_2}\mathbf{0_{s_2}}, 0_{s_3}, 1_{s_3}, 0_{s_4}, 1_{s_4}\}$), respectively (see Figure 7). Again, with no concurrent write() operations, c_i returns two different values although receiving the same set of replies. At this point each server replied with two different values. A simple induction argument implies that waiting more does not bring any new way to break symmetry. $\square_{Theorem\ 3}$

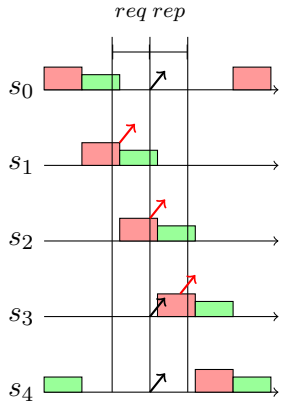

Figure 5: A 2δ read() operation is performed in a CAM model for $\delta \leq \Delta < 2\delta$.

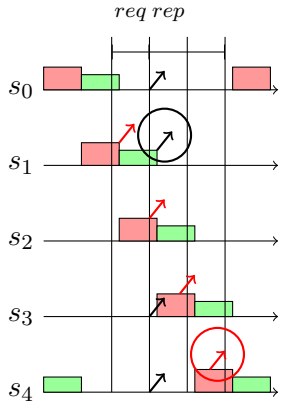

Figure 6: A 3δ read() operation is performed in a CAM model for $\delta \leq \Delta < 2\delta$.

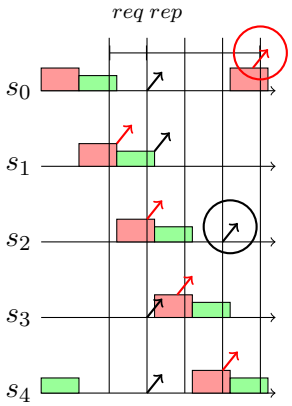

Figure 7: A 4δ read() operation is performed in a CAM model for $\delta \leq \Delta < 2\delta$.

Theorem 4 *If $\delta \leq \Delta < 2\delta$, $\gamma \leq 2\delta$ and $n \leq 8f$, then there exists no protocol $\mathcal{P}_{reg}$ that implements a safe register abstraction in the $(\Delta S, CUM)$ model.*

Proof Let us suppose for the purpose of contradiction that such a protocol $\mathcal{P}_{reg}$ exists. Suppose now that $\mathcal{P}_{reg}$ implements a read() operation whose duration is exactly 2δ. When a client c_i invokes a read() operation *op*, c_i successfully reads the register value (*i.e.*, the valid value at $t_B(op)$), by the safe register validity property in the scenario with no concurrent write() operation. In such a scenario (see Figure 8, where we depicted in red and in yellow the time in which a server is affected and in a cured state respectively), let us consider a first execution E_1 in which the register valid value is 1 and c_i invokes *op*. We also consider one mobile Byzantine agent. Now, if each faulty server

replies with 0, then c_i collects: $\{0_{s_0}, 1_{s_0}, 0_{s_1}, 0_{s_2}, 0_{s_3}, 1_{s_4}, 0_{s_4}, 1_{s_5}, 1_{s_6}, 1_{s_7}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 1. Let us now consider a second execution E_0 in which the register stores 0, and c_i invokes op. If each faulty server replies with 1, then c_i collects: $\{1_{s_0}, 0_{s_0}, 1_{s_1}, 1_{s_2}, 1_{s_3}, 0_{s_4}, 1_{s_4}, 0_{s_5}, 0_{s_6}, 0_{s_7}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 0. In both executions E_1 and E_0, c_i collects the same set of replies, yet gives two different values, hence a contradiction.

We now extend the indistinguishably argument to longer durations for $\mathcal{P}_{reg}$ completion. Assume $\mathcal{P}_{reg}$ implements a read() operation has duration 3δ. In such a case (see Figure 4.5), the previous executions E_1 and E_0 evolve in E_1' (in which c_i collects: $\{0_{s_0}, 1_{s_0}, 0_{s_1}, \mathbf{1_{s_1}}, 0_{s_2}, 0_{s_3}, 1_{s_4}, 0_{s_4}, 1_{s_5}, \mathbf{0_{s_5}}, 1_{s_6}, 1_{s_7}\}$) and E_0' (in which c_i collects: $\{1_{s_0}, 0_{s_0}, 1_{s_1}, \mathbf{0_{s_1}}, 1_{s_2}, 1_{s_3}, 0_{s_4}, 1_{s_4}, 0_{s_5}, \mathbf{1_{s_5}}, 0_{s_6}, 0_{s_7}\}$), respectively. Again, since there are not concurrent write() operations, due to the safe register validity property, c_i returns two different values in the two executions, although the replies c_i gets are the same in both cases.

The case of duration 4δ is similar. The previous executions E_1' and E_0' evolve in E_1'' (in which c_i collects: $\{0_{s_0}, 1_{s_0}, 0_{s_1}, 1_{s_1}, 0_{s_2}, \mathbf{1_{s_2}}, 0_{s_3}, 1_{s_4}, 0_{s_4}, 1_{s_5}, 0_{s_5}, 1_{s_6}, \mathbf{0_{s_6}}, 1_{s_7}\}$) and E_0'' (in which c_i collects: $\{1_{s_0}, 0_{s_0}, 1_{s_1}, 0_{s_1}, 1_{s_2}, \mathbf{0_{s_2}}, 1_{s_3}, 0_{s_4}, 1_{s_4}, 0_{s_5}, 1_{s_5}, 0_{s_6}, \mathbf{1_{s_6}}, 0_{s_7}\}$), respectively (see Figure 4.5). Again, with no concurrent write() operations, c_i returns two different values although receiving the same set of replies.

Finally, the case of duration 5δ is similar. The previous executions E_1'' and E_0'' evolve in E_1''' (in which c_i collects: $\{0_{s_0}, 1_{s_0}, 0_{s_1}, 1_{s_1}, 0_{s_2}, 1_{s_2}, 0_{s_3}, \mathbf{1_{s_3}}, 1_{s_4}, 0_{s_4}, 1_{s_5}, 0_{s_5}, 1_{s_6}, 0_{s_6}, 1_{s_7}, \mathbf{0_{s_7}}\}$) and E_0''' (in which c_i collects: $\{1_{s_0}, 0_{s_0}, 1_{s_1}, 0_{s_1}, 1_{s_2}, 0_{s_2}, 1_{s_3}, \mathbf{0_{s_3}}, 0_{s_4}, 1_{s_4}, 0_{s_5}, 1_{s_5}, 0_{s_6}, 1_{s_6}, 0_{s_7},\ 1_{s_7}\}$), respectively (see Figure 4.5). Again, with no concurrent write() operations, c_i returns two different values although receiving the same set of replies. At this point each server replied with two different values. A simple induction argument implies that waiting more does not bring any new way to break symmetry. $\square_{Theorem\ 4}$

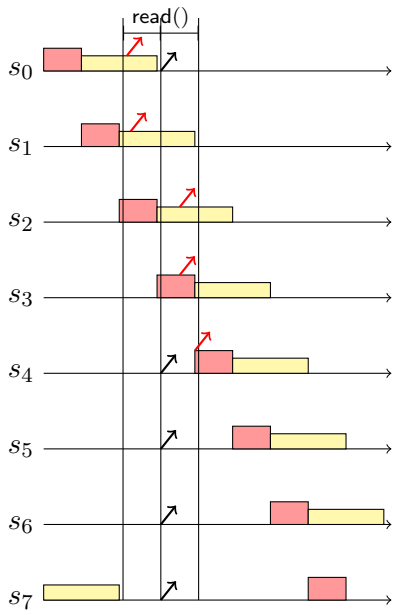

Figure 8: A 2δ read() operation is performed in the CUM model for $\delta \leq \Delta < 2\delta$ and $\gamma \leq 2\delta$.

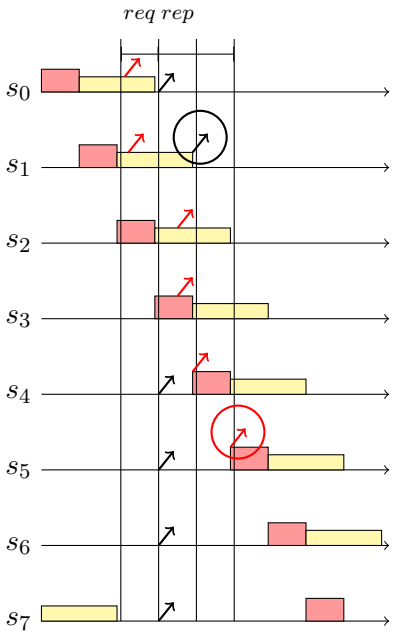

Figure 9: A 3δ read() operation is performed in the CUM model for $\delta \leq \Delta < 2\delta$ and $\gamma \leq 2\delta$.

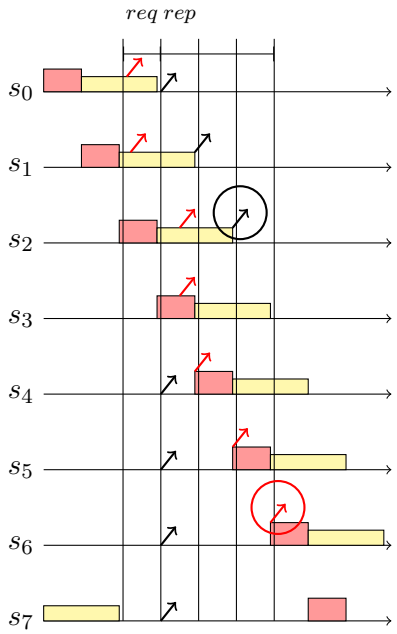

Figure 10: A 4δ read() operation is performed in the CUM model for $\delta \leq \Delta < 2\delta$ and $\gamma \leq 2\delta$.

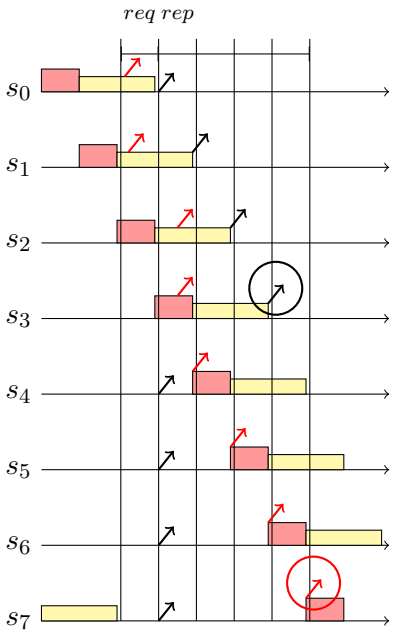

Figure 11: A 4δ read() operation is performed in the CUM model for $\delta \leq \Delta < 2\delta$ and $\gamma \leq 2\delta$.

4.6 $2\delta \leq \Delta < 3\delta$, $(\Delta S, CAM)$ and $(\Delta S, CUM)$ models.

Theorem 5 *If $2\delta \leq \Delta < 3\delta$, $\gamma \leq \delta$ and $n \leq 4f$, then there exists no protocol $\mathcal{P}_{reg}$ that implements a safe register abstraction in the $(\Delta S, CAM)$ model.*

Proof Let us suppose for the purpose of contradiction that such a protocol $\mathcal{P}_{reg}$ exists. Suppose now that $\mathcal{P}_{reg}$ implements a read() operation whose duration is exactly 2δ. When a client c_i invokes a read() operation *op*, c_i successfully reads the register value (*i.e.*, the valid value at $t_B(op)$), by the safe register validity property in the scenario with no concurrent write() operation. In such a scenario (see Figure 12, where we depicted in red and in green the time in which a server is affected and in a cured state respectively), let us consider a first execution E_1 in which the register valid value is 1 and c_i invokes *op*. We also consider one mobile Byzantine agent. Now, if each faulty server replies with 0, then c_i collects: $\{0_{s_0}, 1_{s_1}, 1_{s_2}, 0_{s_3}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 1. Let us now consider a second execution E_0 in which the register stores 0, and c_i invokes *op*. If each faulty server replies with 1, then c_i collects: $\{1_{s_0}, 0_{s_1}, 0_{s_2}, 1_{s_3}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 0. In both executions E_1 and E_0, c_i collects the same set of replies, yet gives two different values, hence a contradiction.

We now extend the indistinguishability argument to longer durations for $\mathcal{P}_{reg}$ completion. Assume $\mathcal{P}_{reg}$ implements a read() operation has duration 3δ. In such a case (see Figure 13), the previous executions E_1 and E_0 evolve in E_1' (in which c_i collects: $\{0_{s_0}, \mathbf{1_{s_1}}, 1_{s_1}, 1_{s_2}, \mathbf{0_{s_2}}, 0_{s_3}\}$) and E_0' (in which c_i collects: $\{1_{s_0}, \mathbf{0_{s_0}}, 0_{s_1}, 0_{s_2}, \mathbf{1_{s_2}}, 1_{s_3}\}$), respectively. Again, since there are not concurrent write() operations, due to the safe register validity property, c_i return two different values in the two executions, although the replies c_i gets are the same in both cases. A duration of 4δ allows the same two executions E_1' and E_0' as in the 3δ case (see Figure 14), leading to an identical outcome. The case of duration 5δ is similar. The previous executions E_1' and E_0' evolve in E_1'' (in which c_i collects: $\{0_{s_0}, 1_{s_1}, 1_{s_1}, \mathbf{0_{s_1}}, 1_{s_2}, 0_{s_2}, 0_{s_3}, \mathbf{1_{s_3}}\}$) and E_0'' (in which c_i collects: $\{1_{s_0}, 0_{s_0}, 0_{s_1}, \mathbf{1_{s_1}}, 0_{s_2}, 1_{s_2}, 1_{s_3}, \mathbf{0_{s_3}}\}$), respectively (see Figure 15). Again, with no concurrent write() operations, c_i returns two different values although receiving the same set of replies. At this point each server replied with two different values. A simple induction argument implies that waiting more does not bring any new way to break symmetry. $\square_{Theorem\ 3}$

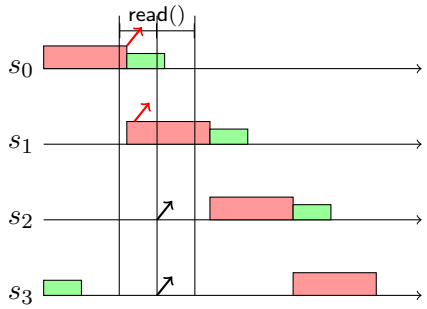

Figure 12: A 2δ read() operation is performed in a CAM model in which $2\delta \leq \Delta < 3\delta$.

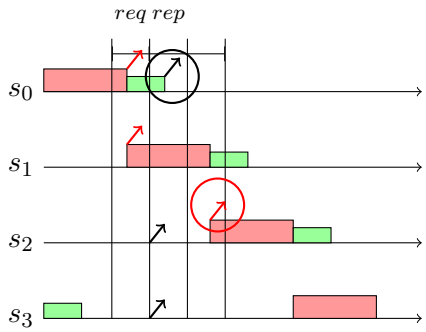

Figure 13: A 3δ read() operation is performed in a CAM model in which $2\delta \leq \Delta < 3\delta$.

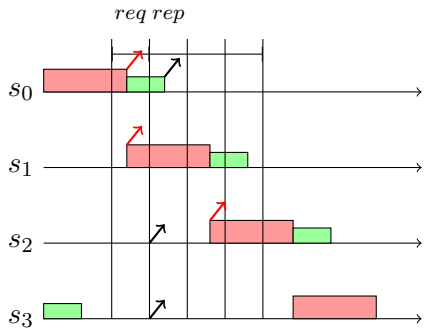

Figure 14: A 4δ read() operation is performed in a CAM model in which $2\delta \leq \Delta < 3\delta$.

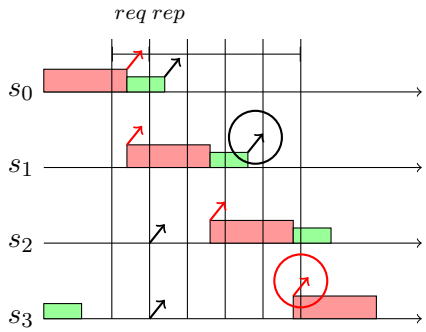

Figure 15: A 5δ read() operation is performed in a CAM model in which $2\delta \leq \Delta < 3\delta$.

Theorem 6 *If $2\delta \leq \Delta < 3\delta$, $\gamma \leq 2\delta$ and $n \leq 5f$, then there exists no protocol $\mathcal{P}_{reg}$ that implements a safe register abstractions in the $(\Delta S, CUM)$ model.*

Proof Let us suppose for the purpose of contradiction that such a protocol $\mathcal{P}_{reg}$ exists. Suppose now that $\mathcal{P}_{reg}$ implements a read() operation whose duration is exactly 2δ. When a client c_i invokes a read() operation *op*, c_i successfully reads the register value (*i.e.*, the valid value at $t_B(op)$), by the safe register validity property in the scenario with no concurrent write() operation. In such a scenario (see Figure 16, where we depicted in red and in green the time in which a server is affected and in a cured state respectively), let us consider a first execution E_1 in which the register valid value is 1 and c_i invokes *op*. We also consider one mobile Byzantine agent. Now, if each faulty server replies with 0, then c_i collects: $\{0_{s_0}, 0_{s_1}, 1_{s_2}, 1_{s_3}, 0_{s_4}, 1_{s_4}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 1. Let us now consider a second execution E_0 in which the register stores 0, and c_i invokes *op*. If each faulty server replies with 1, then c_i collects: $\{1_{s_0}, 1_{s_1}, 0_{s_2}, 0_{s_3}, 1_{s_4}, 0_{s_4}\}$. Since $\mathcal{P}_{reg}$ implements a safe register and there are not concurrent write() operations, then c_i reads the valid value at $t_B(op)$, that is, 0. In both executions E_1 and E_0, c_i collects the same set of replies, yet gives two different values, hence a contradiction.

We now extend the indistinguishably argument to longer durations for $\mathcal{P}_{reg}$ completion. Assume $\mathcal{P}_{reg}$ implements a read() operation has duration 3δ. In such a case (see Figure 4.6), we consider $n \leq 6f$, so that we build the following executions, E_1' (in which c_i collects: $\{0_{s_0}, 0_{s_1}, 1_{s_2}, \mathbf{0_{s_2}}, 1_{s_3}, \mathbf{1_{s_4}}, 0_{s_5}, 1_{s_5}\}$) and E_0' (in which c_i collects: $\{1_{s_0}, 1_{s_1}, 0_{s_2}, \mathbf{1_{s_2}}, 0_{s_3}, \mathbf{0_{s_4}}, 1_{s_5}, 0_{s_5}\}$), respectively. Again, since there are not concurrent write() operations, due to the safe register validity property, c_i returns two different values in the two executions, although the replies c_i gets are the same in both cases.

When we analyze the case of duration 4δ we consider directly $n \leq 5$. The previous executions E_1 and E_0 evolve in E_1'' (in which c_i collects: $\{0_{s_0}, \mathbf{1_{s_0}}, 0_{s_1}, 1_{s_2}, \mathbf{0_{s_2}}, 1_{s_3}, 0_{s_4}, 1_{s_4}\}$) and E_0'' (in which c_i collects: $\{1_{s_0}, \mathbf{0_{s_0}}, 1_{s_1}, 0_{s_2}, 0_{s_3}, \mathbf{1_{s_3}}, 1_{s_4}, 0_{s_4}\}$), respectively (see Figure 4.6). Again, with no concurrent write() operations, c_i returns two different values although receiving the same set of replies.

When we consider the case of duration 5δ we consider $n \leq 6$. The previous executions E_1' and E_0' evolve in E_1''' (in which c_i collects: $\{0_{s_0}, \mathbf{1_{s_0}}, 0_{s_1}, 1_{s_2}, 0_{s_2}, 1_{s_3}, \mathbf{0_{s_3}}, 1_{s_4}, 0_{s_5}, 1_{s_5}\}$) and E_0''' (in which c_i collects: $\{0_{s_0}, \mathbf{1_{s_0}}, 0_{s_1}, 1_{s_2}, 0_{s_2}, 1_{s_3}, \mathbf{0_{s_3}}, 1_{s_4}, 0_{s_5}, 1_{s_5}\}$), respectively (see Fig. 4.6). Again, with no concurrent write() operations, c_i returns two different values although receiving the same

set of replies. It follows that if does not exist $\mathcal{P}_{reg}$ solving the regular register for $n \leq 6f$ then it is straightforward that such protocol does not exist for $n \leq 5f$.

We can proceed in the same way for read() operation whose duration is 6δ (Fig. 4.6) and 7δ ((Fig. 4.6)).

At this point each server replied with two different values. A simple induction argument implies that waiting more does not bring any new way to break symmetry. $\square_{Theorem\ 6}$

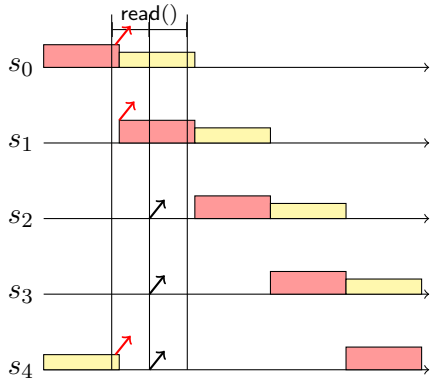

Figure 16: A 2δ read() operation performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

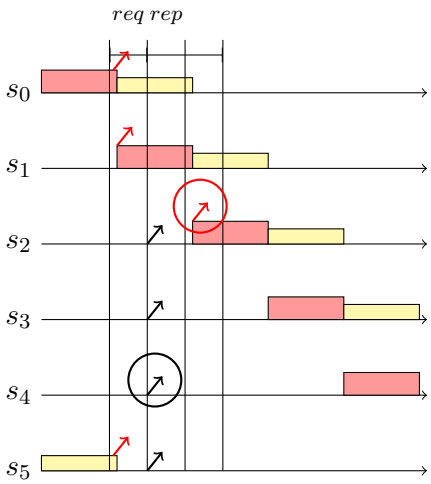

Figure 17: A 3δ read() operation is performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

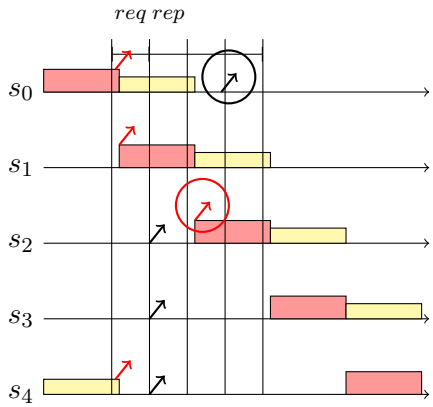

Figure 18: A 4δ read() operation is performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

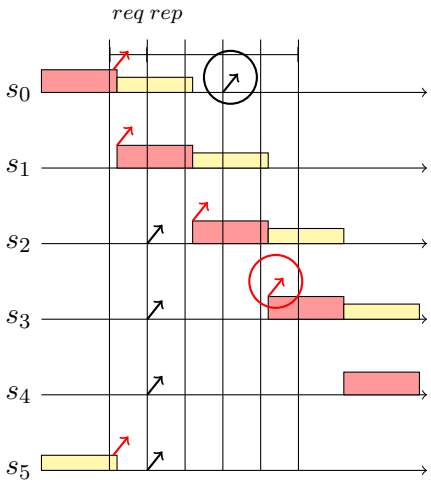

Figure 19: A 5δ read() operation is performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

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

$k\Delta \geq 2\delta,\ k \in \{1,2\}$	$n_{CAM} \geq (k+3)f+1$	$\#reply_{CAM} \geq (k+1)f+1$
$k=1$	$4f+1$	$2f+1$
$k=2$	$5f+1$	$3f+1$

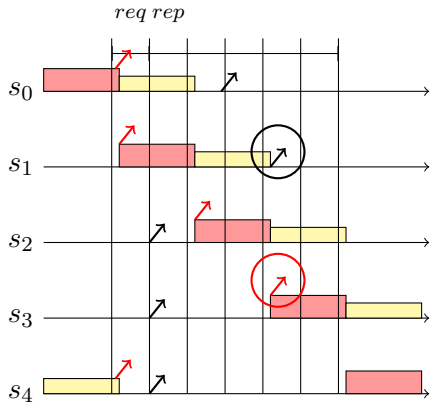

Figure 20: A 6δ read() operation is performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

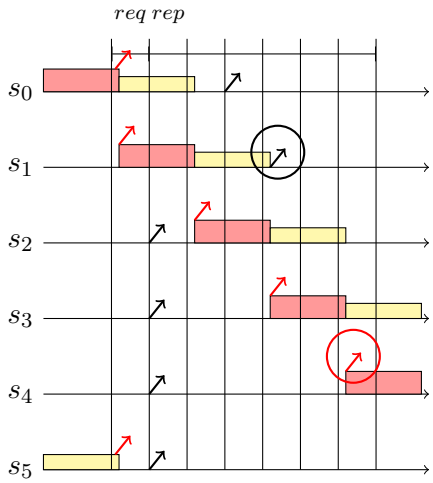

Figure 21: A 7δ read() operation is performed in the CUM model for $2\delta \leq \Delta < 3\delta$ and $\gamma \leq 2\delta$.

5 Optimal Regular Register Implementation in the $(\Delta S, CAM)$ model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ that implements a SWMR Regular Register in a round-free synchronous system for $(\Delta S, CAM)$ instance of the proposed MBF model. Our solution is based on the following three key points: (1) we implement a maintenance() operation that is executed periodically at each $T_i = t_0 + i\Delta$ time. In this way, the effect of a Byzantine agent on a server disappears in a bounded period of time; (2) we implement read() and write() operations following the classical quorum-based approach. The size of the quorum needed to carry on the operations, and consequently the total number of servers required by the computation, is computed by taking into account the time to terminate the maintenance() operation, δ and Δ; (3) we define a forwarding mechanism to avoid that READ() and WRITE() messages are "lost" by some server s_i due to a concurrent movement of the Byzantine agent during such operations. Note that even though communication channels are reliable, we may have the following situation: a message is sent by a client at time t and the Byzantine agents move at some $t' < t + \delta$. As a consequence, some faulty servers may receive the message in the interval $[t, t']$ and then agents move leaving cured servers without any trace of the message.

Interestingly, we found that the number of replicas needed to tolerate f Byzantine agents does not depend only on f but also on the Δ and δ relationship (see Table 1).

5.1 $\mathcal{P}_{reg}$ Detailed Description

The protocol $\mathcal{P}_{reg}$ for the $(\Delta S, CAM)$ model is described in Figures 22 - 24, which present the maintenance(), write(), and read() operations, respectively.

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 also maintains a local sequence number csn that is incremented each time it invokes a write() operation and is used to timestamp such operations.

Local variables at server s_i. Each server s_i maintains the following local variables (we assume these variables are initialized to zero, false or empty sets according their type):

- V_i: an ordered set containing tree tuples $\langle v, sn \rangle$, where v is a value and sn the corresponding sequence number. Such tuples are ordered incrementally according to their sn values. The function insert$(V_i, \langle v_k, sn_k \rangle)$ places the new value in V_i according to the incremental order and, if there are more than three values, it discards from V_i the value associated to the lowest sn.
- $cured_i$: boolean flag updated by the cured_state oracle. In particular, such variable is set to true when s_i becomes aware of its cured state and it is reset during the algorithm when s_i becomes correct.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated trough ECHO messages. The first one stores tuple $\langle j, \langle v, sn \rangle \rangle$ propagated by servers just after the mobile Byzantine agents moved, while the second stores the set of concurrently reading clients in order to notify cured servers and expedite termination of read().
- fw_vals_i: set variable storing a triple $\langle j, \langle v, sn \rangle \rangle$ meaning that server s_j forwarded a write message with value v and sequence number sn.
- $pending\_read_i$: 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 $echo\_vals_i$ and returns, if they exist, three tuples $\langle v, sn \rangle$, such that there exist at least $2f + 1$ occurrences in $echo\_vals_i$ of such tuple. If more than three of such tuple exist, the function returns the tuples with the highest sequence numbers. Otherwise if there are two tuples, the third tuple returned is $\langle \perp, 0 \rangle$.
- 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 at least $\#reply_{CAM}$ times (see Table 1). If there are more pairs satisfying such condition, it returns the one with the highest sequence number.

The maintenance() operation. Such operation is executed by servers periodically at any time $T_i = t_0 + i\Delta$.

If a server s_i is not in a cured state then it broadcasts an ECHO message carrying the set V_i and the set $pending\_read_i$ (it contains identifiers of clients that are currently running a read()

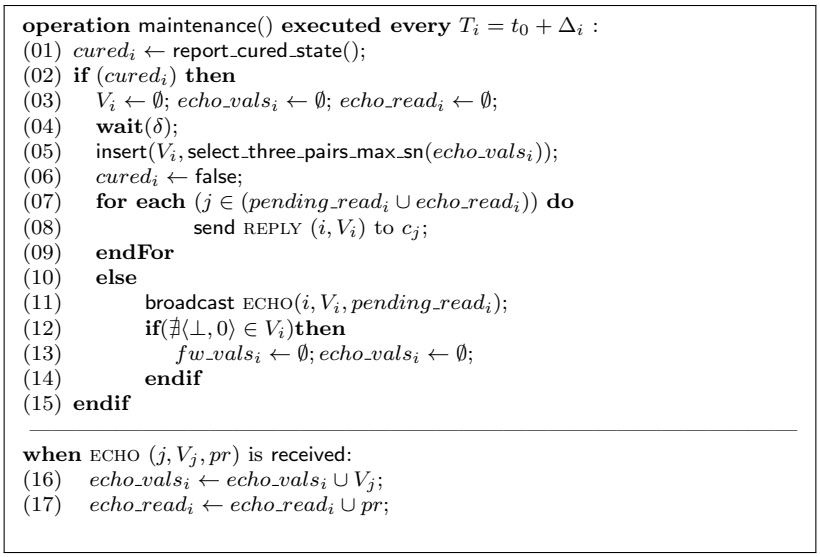

Figure 22: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CAM)$ model.

operation). Moreover if in V_i there are no $\langle \perp, 0 \rangle$ values then it empties the fw_vals_i and $echo\_vals_i$ sets, meaning that it is not trying to retrieve a value lost because s_i was affected by a Byzantine agent while such value has been written.

Otherwise, if a server s_i is in a cured state it first cleans its local variables and then, after δ time units, tries to update its state by checking the number of occurrences of each pair $\langle v, sn \rangle$ received with ECHO messages. In particular, it updates V_i invoking the select_three_pairs_max_sn($echo\_vals_i$) function that populates V_i with three or at least two tuples $\langle v, sn \rangle$. If there are only two tuple $\langle v, sn \rangle$, it means that there exists a concurrent write() operation that is 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. Finally, it assigns false to $cured_i$, meaning that it is now correct and starts replying to clients that are currently reading.

The write() operation. When the writer wants to write a value v, it increments its sequence number csn and propagates v and csn to all servers. Then it waits for δ time units (the maximum message transfer delay) before returning.

When a server s_i delivers a WRITE, it updates its local variables and forwards the message, trough a WRITE_FW$(i, \langle v, csn \rangle)$, to others servers. This prevents the message loss in case servers deliver such message while they are affected by mobile Byzantine agents. In addition, it also sends a REPLY() message to all clients that are currently reading (clients in $pending\_read_i$) to allow them to terminate their read() operation.

When s_i delivers a WRITE_FW$(j, \langle v, csn \rangle)$ message, it stores such message in fw_vals_i set. Such set is constantly monitored together with $echo\_vals_i$ set to find a couple $\langle v, sn \rangle$ occurring at least $\#reply_{CAM}$ times. This continuous check enables servers in a cured state to store the new value and reply to a reading client as soon as possible even in case they delivered such value when affected by mobile Byzantine agents.

```
operation write(v):
(01) csn ← csn + 1;
(02) broadcast WRITE(v, csn);
(03) wait (δ);
(04) return write_confirmation;
```

(a) Client code (code for client c_i).

```
when WRITE(v, csn) is received:
(01)   insert(V_i, ⟨v, csn⟩);
(02)   for each j ∈ (pending_read_i ∪ echo_read_i) do
(03)       send REPLY (i, {⟨v, csn⟩});
(04)   endFor
(05)   broadcast WRITE_FW(i, ⟨v, csn⟩);
---------------------------------------------------------------
when WRITE_FW(j, ⟨v, csn⟩) is received:
(06)   fw_vals_i ← fw_vals_i ∪ {⟨j, ⟨v, csn⟩⟩};
---------------------------------------------------------------
when ∃⟨j, ⟨v, sn⟩⟩ ∈ (fw_vals_i ∪ echo_vals_i) occurring at least #reply_CAM times:
(07)   insert(V_i, ⟨v, sn⟩);
(08)   ∀j : fw_vals_i ← fw_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(09)   ∀j : echo_vals_i ← echo_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(10)   for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(11)           send REPLY (i, {⟨v, sn⟩}) to c_j;
(12)   endFor
```

(b) Server code (code for server s_i).

Figure 23: $\mathcal{A}_W$ algorithm implementing the write(v) operation in the $(\Delta S, CAM)$ model.

The read() operation. When a client wants to read, it broadcasts a READ() request to all servers and waits 2δ time (i.e., one round trip delay) to collect replies. When it is unblocked from the wait statement, it selects a value v invoking the select_value function on $reply_i$ set, sends an acknowledgement message to servers to inform that its operation is now terminated and returns v as result of the operation.

When a server s_i delivers a READ(j) message from client c_j it first puts its identifier in the set $pending\_read_i$ to remember that c_j is reading and needs to receive possible concurrent updates, then s_i checks if it is in a cured state and if not, it sends a reply back to c_j. Note that, the REPLY() message carries the set V_i, which contains three tuples $\langle value, ts\rangle$ or two tuples $\langle value, ts\rangle$ and one $\langle \perp, 0\rangle$.

The last case occurs if s_i was affected by a Byzantine agent when the last write() operation occurred so that s_i is still retrieving such value. As soon as s_i retrieve such value through the fw_vals_i and $echo\_vals_i$ sets, such value is sent back to c_j.

In any case, s_i forwards a READ_FW message to inform other servers about c_j read request. This is useful in case some server missed the READ(j) message as it was affected by mobile Byzantine agent when such message has been delivered.

When a READ_FW(j) message is delivered, c_j identifier is added to $pending\_read_i$ set, as when the read request is just received from the client.

When a READ_ACK(j) message is delivered, c_j identifier is removed from both $pending\_read_i$ and $echo\_read_i$ sets as it does not need anymore to receive updates for the current read() operation.

```
operation read():
(01) reply_i ← ∅;
(02) broadcast READ(i);
(03) wait (2δ);
(04) ⟨v, sn⟩ ← select_value(reply_i);
(05) broadcast READ_ACK(i);
(06) return v;
---
when REPLY (j, V_j) is received:
(07)   for each (⟨v, sn⟩ ∈ V_j) do
(08)           reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(09)   endFor
```

(a) Client code (code for client c_i).

```
when READ (j) is received:
(01)   pending_read_i ← pending_read_i ∪ {j};
(02)   if (¬cured_i)
(03)     then send REPLY (i, V_i);
(04)   endif
(05)   broadcast READ_FW(j);
---
when READ_FW (j) is received:
(06)   pending_read_i ← pending_read_i ∪ {j};
---
when READ_ACK (j) is received:
(07)   pending_read_i ← pending_read_i \ {j};
(08)   echo_read_i ← echo_read_i \ {j};
```

(b) Server code (code for server s_i).

Figure 24: $\mathcal{A}_R$ algorithm implementing the read() operation in the $(\Delta S, CAM)$ model.

5.2 Correctness proofs

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. Due to the algorithm implementation, such property is independent from the specific instance of the MBF model considered.

Lemma 4 *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 behavior of the servers (see lines 03-04, Figure 23(a)). $\square_{Lemma\ 4}$

Lemma 5 *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 behaviour of the servers (see lines 03-06, Figure 24(a)). $\square_{Lemma\ 5}$

Theorem 7 (Termination) *If a correct client c_i invokes an operation, c_i returns from that operation in finite time.*

Proof The proof simply follows from Lemma 4 and Lemma 5. $\square_{Theorem\ 7}$

Definition 8 (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 9 ($Max\tilde{B}(t, t + T)$) *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 6 *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. $\square_{Lemma\ 6}$

Table 2: Table summing up the value that we get substituting δ and Δ in the formulas.

	$(Max\bar{B}(t, t+2\delta)$	$(Max\bar{B}(t, t+\delta)$	$\lceil \frac{2\delta}{\Delta} \rceil$	$\lceil \frac{\delta}{\Delta} \rceil$
$k=1$	$2f$	$2f$	1	1
$k=2$	$3f$	$2f$	2	1

Definition 10 ($min\tilde{C}o(t, t+T)$) *Let $[t, t+T]$ be a time interval. The cardinality of $\tilde{C}o(t, t+T)$ is minimum if for any t', $t' > 0$, is it true that $|\tilde{C}o(t, t+T)| \leq |\tilde{C}o(t, t'+T)|$. Let $min\tilde{C}o(t, t+T)$ be such cardinality.*

Any read() operation op_R lasts from $t_B(op_{op_R})$ to $t_E(op_{op_R})$ for 2δ time. The previous definition gives us the number of servers that are correct from the beginning to the end of the operation. What is really useful is the number of servers that can correctly reply during such time. Servers that are correct from $t_B(op_{op_R})+\delta$ to $t_E(op_{op_R})-\delta$. Being correct from from $t_B(op_{op_R})+\delta$ means that in the time interval $[t, t_B(op_{op_R})+\delta]$ servers were no Byzantine, at most they could have been in a cured state, so able to deliver the READ() message and since are correct at $t_B(op_{op_R})+\delta$ their replies are also delivered by the reader by time $t_B(op_{op_R})+2\delta = t_E(op_{op_R})$. For $t_E(op_{op_R})-\delta$ the reasoning is the similar. So let us define the minimum number of servers able to reply to a read() operation.

Definition 11 ($min\tilde{C}oR(t, t+T)$) *Let $[t, t+T]$ be a time interval. then $\tilde{C}oR(t, t+T)$ denotes the minimum number of never affected servers during a time interval $[t, t+T-\delta]$[4] The cardinality of $\tilde{C}oR(t, t+T)$ is minimum if for any t', $t' > 0$, is it true that $|\tilde{C}o(t, t+T)| \leq |\tilde{C}o(t, t'+T)|$. Let $min\tilde{C}oR(t, t+T)$ be such cardinality.*

Now we can compute the minimum number of servers that correctly reply during the read() operation as if each message to/from them requires δ time to be delivered. So that we consider $min\tilde{C}oR(t_B(op_R), t_E(op_R))$. Concerning Byzantine servers that can reply we consider $max\tilde{B}(t_B(op_R), t_E(op_R))$ as if each message to/from them is delivered instantaneously.

Lemma 7 *Let op be a read() operation issued at time t and terminating at time $t+2\delta$. If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) and (ii) $n_{CAM} \geq (k+3)f+1$, then $min\tilde{C}oR(t, t+2\delta) > Max\tilde{B}(t, t+2\delta)$ and $min\tilde{C}oR(t, t+2\delta) \geq \#reply_{CAM}$.*

Proof

Let us consider the two cases separately, $k=2$ so that $\delta \leq \Delta < 2\delta$ and $k=1$ so that $2\delta \leq \Delta$

[4]We are interested in correct servers that surely can reply, so that a READ() message sent by a client is for sure delivered by servers by $t+\delta$ and a *reply* sent by $t+T-\delta$ is for sure delivered by client.

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

  From Lemma 6, $Max\tilde{B}(t, t+2\delta) = (\lceil \frac{2\delta}{\Delta} \rceil + 1) \times f$. Considering that $\delta \leq \Delta < 2\delta$, we obtain $Max\bar{B}(t, t+2\delta) = 3f$.

  The number of correct servers at time $t+\delta$ is given by the number of serves that are non-faulty in the whole interval $(n_{CAM} - Max\bar{B}(t, t+2\delta))$ 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., $Max\bar{B}(t, t+2\delta) - Max\bar{B}(t+\delta, t+2\delta)$).

  Thus

$$min\tilde{C}oR(t, t+2\delta) = n_{CAM} - Max\bar{B}(t, t+2\delta) + Max\bar{B}(t, t+2\delta) - Max\bar{B}(t+\delta, t+2\delta)$$

$$min\tilde{C}oR(t, t+2\delta) = n_{CAM} - Max\bar{B}(t+\delta, t+2\delta)$$

$$min\tilde{C}oR(t, t+2\delta) = n_{CAM} - (\lceil \frac{2\delta}{\Delta} \rceil + 1) \times f$$

$$min\tilde{C}oR(t, t+2\delta) = 5f + 1 - 2f = 3f + 1 = \#reply_{CAM}$$

- **Case 2 - $(\Delta S, CAM)$ with $2\delta \leq \Delta$.** Following the consideration done in Case 1, we obtain that $Max\bar{B}(t, t+2\delta) = 2f$ for $2\delta \leq \Delta$ and that

$$min\tilde{C}oR(t, t+T) = n_{CAM} - Max\bar{B}(t, t+2\delta) + Max\bar{B}(t, t+2\delta) - Max\bar{B}(t+\delta, t+2\delta)$$

$$min\tilde{C}oR(t, t+2\delta) = n_{CAM} - Max\bar{B}(t+\delta, t+2\delta)$$

$$min\tilde{C}oR(t, t+2\delta) = n_{CAM} - (\lceil \frac{2\delta}{\Delta} \rceil + 1) \times f$$

$$min\tilde{C}oR(t, t+2\delta) = 4f + 1 - 2f = 2f + 1 = \#reply_{CAM}$$

From which the claim fallows.

$\square_{Lemma\ 7}$

Definition 12 (Valid Value Set at time t) *The* valid value set *at time t, denoted by $VVS(t)$, is the set of valid values at time t. More in details it may contain:* (i) *value v written by the last* write() *operation terminated before t, and* (ii) *any values v' written by a* write() *operation running at time t. As we assume a single writer model, there can be at most one such v'. If no* write() *has started before time t, $VVS(t)$ contains only $\perp$.*

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 7 the next corollary follows

Corollary 3 *Let op be a* read() *operation issued at time t and terminating at time $t+2\delta$. The number of replies carrying valid values at some time $\tau \in [t, t+2\delta]$ is always greater than the number of replies carrying non valid values.*

Definition 13 (write() completion time t_{wE}) *Let* write() *be an operation op_W writing v on the register and let S the set of servers in $B(t_B(op_W))$ that missed the* WRITE(v) *message. Then t_{wE} is the time at which every server $s_k \in S$ retrieve such value v, so that, by this time v is stored by $\#reply_{CAM} + f$ correct servers.*

Lemma 8 *Let op be a* write(v) *operation invoked by a client c_k at time $t_B(op) = t$, then (i) any $s_j \notin B(t, t+\delta)$ has $v \in V_j$ at time $t+\delta$ and (ii) the write completion time $t_{wE} \leq t + 2\delta$.*

Proof The first part of the proof simply follows considering that: due to the communication channel synchrony, the WRITE messages from c_k are delivered by servers within the time interval $[t, t+\delta]$; any correct server s_j during the whole time interval $[t, t+\delta]$ executes the correct algorithm code. Thus, when s_j delivers the WRITE message it executes line 01 in Figure 23(b) storing v in V. Concerning the second part, let us call $S \subseteq B(t)$ the set of servers whose missed the WRITE() message sent by c_k because affected by a Byzantine agent at the beginning of *op*. Let T_i the time at which servers in S became cured, so that $t < T_i < t + \delta$. Any server $s_i \in S$ does execute line 07 in Figure 23(b) (i.e., it stores v in V) when there are at least $\#reply_{CAM} = (k+1)f + 1$ occurrences of v in the set $fw\_vals_i \cup echo\_vals_i$. Considering that during the write() operation at most $2f$ servers may be affected by Byzantine agents (cf. Lemma 6 for a δ time interval), then there are $min\tilde{Co}(t, t+\delta) = n - 2f = 3(k+1)f + 1 - 2f = (k+1)f + 1$ non faulty servers. In other words, there are at least $(k+1)f+1$ non faulty servers executing line 05. Servers in $\tilde{Co}(t, t+\delta)$ may deliver the WRITE() message from c_k before of after T_i. Consequently they can send the WRITE_FW() message before of after T_i. In the first case the WRITE_FW() message can be missed as well (i.e. delivered by servers in S before T_i, when still Byzantine). When those servers deliver the WRITE() message, v is inserted in the V set (Figure 23 line 01) such that at the next maintenance() operation, at T_i, v is in the ECHO() message (Figure 22 line 08). This means that servers in S deliver the written value v at most at $T_i + \delta$. In the second case, when the WRITE() message from c_k is delivered by servers in $\tilde{Co}(t, t+\delta)$ after T_i , the WRITE_FW() message is sent by servers in the time interval $[T_i, t_E(op)]$ (Figure 23 line 05). Since a message is delivered by δ time, then by $t_E(op) + \delta = t + 2\delta$ any servers in S, that missed the write() message, has enough occurrences of v in the $fw\_vals_i \cup echo\_vals_i$ set so that line 07 is executed and v is stored in V. Thus at the write completion time there are $\#reply_{CAM} + f$ servers storing v. To conclude, let us consider that, each server in V can store up to three values. So that even if there are subsequent write() operations, such that the completion time of the first overlap the second write(), the new value does not overwrite the previous one before the write() is effective. $\square_{Lemma\ 8}$

Lemma 9 *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), (ii) $n_{CAM} \geq (k+3)f + 1$, (iii) there are no* write() *operations during $[T_i, T_i + \delta]$. Then $\forall s \in Cu(T_i)$, $s \in Co(T_i, T_i + \delta)$.*

Proof Let us assume that at T_i there are $2f+1$ correct servers storing $V = \{\langle v_0, 0\rangle, \langle v_1, 1\rangle, \langle v_2, 2\rangle\}$ and running the code in Figure 22. In particular each server $s_j \notin Cu(T_1)$ broadcasts a ECHO() message with attached the content of V (line 11) while each server $s_i \in Cu(T_1)$ waits δ time units (line 04) to gather all the ECHO() messages. Let us note that, by hypothesis there are no write() operations during $[T_i, T_i + \delta]$. This means the $2f + 1$ broadcast the same set of values V. So that cured servers set their state to correct and $V = \{\langle v_0, 0\rangle, \langle v_1, 1\rangle, \langle v_2, 2\rangle\}$. This is always true if we consider that at the beginning, at T_0, there are f servers affected by mobile Byzantine agent and $n - f > 2f + 1$ correct servers. So that what we proved always holds. At the end of the

maintenance() there are the same number of correct servers (there have no been cured servers yet) so that at T_1 there are $n - 2f \geq 2f + 1$ correct servers running the maintenance() and at the end of it there are $n - f$ correct servers, whose became $n - 2f \geq 2f + 1$ at T_2 and so on.

$\square_{Lemma\ 9}$

Lemma 10 *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), (ii) $n_{CAM} \geq (k + 3)f + 1$. Then $\forall s \in Cu(T_i)$, $s \in Co(T_i, T_i + \delta)$ and every server in $Co(T_i, T_i + \delta)$ is storing at least one common value, in particular such value is the last written value before T_i.*

Proof Let us assume that at T_i there are $2f+1$ correct servers storing $V = \{\langle v_0, 0\rangle, \langle v_1, 1\rangle, \langle v_2, 2\rangle\}$ and running the code in Figure 22. In particular each server $s_j \notin Cu(T_1)$ broadcasts a ECHO() message with attached the content of V (line 11) while each server $s_i \in Cu(T_1)$ waits δ time units (line 04) to gather all the ECHO() messages. If there is a write() operation op_{W1} writing $\langle v_3, 3\rangle$, such that $T_i \in [t_B(op_{W1}), t_E(op_{W1})]$, then the $2f + 1$ servers in $Co(T_i)$ may broadcast different sets of values V, $\{\langle v_0, 0\rangle, \langle v_1, 1\rangle, \langle v_2, 2\rangle\}$ or $\{\langle v_1, 1\rangle, \langle v_2, 2\rangle, \langle v_3, 3\rangle\}$. At the end of the maintenance(), servers in $Cu(T_i)$ select values occurring at least $2f + 1$ times, thus they set at least $V = \{\langle v_1, 1\rangle, \langle v_2, 2\rangle, \langle \perp, \perp\rangle\}$. At $T_i + \delta$ it may also happen that another write() operation, op_{W2} writing $\langle v_4, 4\rangle$ subsequent to op_{W1}, occurs, 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($\langle v_4, 4\rangle$) message and no yet the servers that were in $Cu(T_i)$. So that the first group of servers is storing $\langle v_2, 2\rangle, \langle v_3, 3\rangle, \{\langle v_4, 4\rangle\}$ and the second group is storing $\{\langle v_2, 2\rangle, \langle v_3, 3\rangle, \langle \perp, \perp\rangle\}$. 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.

$\square_{Lemma\ 10}$

From Lemmas 9 and 10 the next Corollary follows.

Corollary 4 *The* maintenance() *operation guarantees that $\forall T_i, i \in \mathbb{N}$, $\forall s \in Cu(T_i)$, then $s \in Co(T_i, T_i + \delta)$.*

Lemma 11 *Let op_W be a* write() *operation and let v be the written value. Let $t_E w$ be its completion time and let T_i the time of the first Byzantine agent movement after $t_E w$. Then if there are no other* write() *operations after op_W, the value written by op_W is stored by $\#reply_{CAM}$ servers forever.*

Proof Let us consider the case $k = 2$. Let us first consider how many Byzantine servers there may be in the time interval $[t_B(op_W), T_i]$. The time from $t_B(op_W)$ and $t_E w$ is at most 2δ (cf. Lemma 8) and the time from $t_E w$ to T_i is at most δ since we are considering $k = 2$. Thus the time interval $[t_B(op_W), T_i]$ is at most 3δ. In such period, being the agent movement aligned to T_i and being $k = 2$, there may be at most $3f$ Byzantine servers. Moreover, $n \geq 5f + 1$, then at T_i there are at least $f + 1$ servers never affected in the considered time interval and thus, for Lemma 8, such servers are storing v by time $t_B(op_W) + \delta$. For the same Lemma, by time $t_E w$, the f servers that were in $B(t_B(op_W))$ are storing v. Thus, at time T_i there are at least $2f + 1$ correct servers storing v, so for Lemma 9 at the end of the maintenance(), servers that were cured at T_i are now storing the same values as servers in $Co(T_i)$. Thus at T_i there are $n - f \geq 4f + 1$ correct servers storing v. At T_{i+1} the new maintenance() operation run with $n - 2f$ servers storing v, which at the end of the maintenance() operation is $n - f$ again and so on. Thus if there are no more write() operation v is stored forever. Case $k = 1$ is similar concluding the proof. $\square_{Lemma\ 11}$

Lemma 12 *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() *operations issued on the regular register. Let us consider a particular op_{W_k}, let v be the value written by op_{W_k} and let $t_E w_k$ be its completion time. Then the register stores v (there are at least $\#reply_{CAM}$ correct servers storing it) up to time at least $t_B W_{k+3}$.*

Proof The proof simply follows considering that:

- for Lemma 11 if there are no more write() operation then v, after $t_E w$, is in the register forever.
- any new written value is store in an ordered set V (cf. Figure 23 line 01) whose dimension is 3.
- write() operations occur sequentially.

From that is it clear that after the beginning of 3 write() operations, $op_{W_{k+1}}, op_{W_{k+2}}, op_{W_{k+3}}$, v it may be no more stored in the regular register. $\square_{Lemma\ 12}$

Theorem 8 *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) (ii) $n_{CAM} \geq (k+3)f + 1$ then 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 $[t_B(op_R), t_B(op_R) + \delta]$. Since such operation lasts 2δ, the reply messages sent by correct servers within $t_B(op_R) + \delta$ are delivered by the reading client. For Lemma 7, in such period there are $\#reply_{CAM}$ correct servers that sent back a reply message to the reading client. 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. From Lemma 8 and Lemma 11 after the write completion time there are $\#reply_{CAM}$ correct servers storing v. Since $t_B(op_R) + \delta \geq t_E w$, then there are $\#reply_{CAM}$ 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 $[t_B(op_R), t_B(op_R) + \delta]$. In such time there can be at most two write() operations. Thus for Lemma 12 the last written value before $t_B(op_R)$ is still present in $\#reply_{CAM}$ 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 3 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. $\square_{Theorem\ 8}$

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 $\#reply_{CAM}$ correct servers that reply with a value $v \in VVS(t_B(op_R))$.

Theorem 9 *Let n be the number of servers emulating the register and let f be the number of Byzantine agents in the $(\Delta S, CAM)$ round-free Mobile Byzantine Failure model. Let δ be the upper*

Table 3: Parameters for $\mathcal{P}_{Rreg}$ Protocol in the $(\Delta S, CUM)$.

$k = \lceil \frac{2\delta}{\Delta} \rceil, \delta \leq \Delta < 3\delta$	$n_{CUM} \geq (3k+2)f+1$	$\#reply_{CUM} \geq (2k+1)f+1$	$\#echo_{CUM} \geq (k+1)f+1$
$k=2$	$8f+1$	$5f+1$	$3f+1$
$k=1$	$5f+1$	$3f+1$	$2f+1$

bound on the communication latencies in the synchronous system. If (i) $k\Delta \geq 2\delta$ (with $k \in 1, 2$) and (ii) $n \geq (k+3)f+1$, then $\mathcal{P}_{reg}$ implements a SWMR Regular Register in the $(\Delta S, CAM)$ round-free Mobile Byzantine Failure model.

Proof The proof simply follows from Theorem 7 and Theorem 8. $\square_{Theorem\ 9}$

6 Optimal Regular Register Implementation in the $(\Delta S, CUM)$ model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ that implements a SWMR Regular Register in a round-free synchronous system for $(\Delta S, CUM)$ instance of the proposed MBF model.

Our solution is based on the following three key points: (1) we implement a maintenance() operation that is executed periodically at each $T_i = t_0 + i\Delta$ time. In this way, the effect of a Byzantine agent on a server disappears in a bounded period of time; (2) we implement read() and write() operations following the classical quorum-based approach. The size of the quorum needed to carry on the operations, and consequently the total number of servers required by the computation, is computed by taking into account the time to terminate the maintenance() operation, δ and Δ; (3) we define a forwarding mechanism to avoid that READ() and WRITE() messages are "lost" by some server s_i due to a concurrent movement of the Byzantine agent during such operations. Note that even though communication channels are reliable, we may have the following situation: a message is sent by a client at time t and the Byzantine agents move at some $t' < t + \delta$. As a consequence, some faulty servers may receive the message in the interval $[t, t']$ and then agents move leaving cured servers without any trace of the message.
Moreover, contrarily to the $(\Delta S, CAM)$ case, the values that populate auxiliary variables (i.e., not the register stored value) have a fixed life time. This is necessary since servers are never aware to be in a cured state and thus Byzantine processes may force them to take wrong decisions.

Interestingly, we found that the number of replicas needed to tolerate f Byzantine agents does not depend only on f but also on the Δ and δ relationship (see Table 3).

6.1 $\mathcal{P}_{reg}$ Detailed Description

The protocol $\mathcal{P}_{reg}$ for the $(\Delta S, CUM)$ model is described in Figures 25 - 27.

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 also maintains a local sequence number csn that is incremented each time it invokes a write() operation and is used to timestamp such operations.

Local variables at server s_i. Each server s_i maintains the following local variables (we assume these variables are initialized to zero, false or empty sets according their type):

- V_i: an ordered set containing 3 tuples $\langle v, sn \rangle$, where v is a value and sn the corresponding sequence number. Such tuples are ordered incrementally according to their sn values.
- V_{safe_j}: this set has the same characteristic as V_j. The function insert($V_{safe_i}, \langle v_k, sn_k \rangle$) places the new value in V_{safe_i} according to the incremental order and if dimensions exceed 3 then it discards from V_{safe_i} the value associated to the lowest sn.
- W_i: is the set where servers store values coming directly from the writer, associating to it a timer, $\langle v, sn, timer \rangle$. Values from this set are deleted at the end of the maintenance() operation when the timer expires or has a value non compliant with the protocol.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through ECHO messages at the beginning of the maintenance() operation. The first one stores tuple $\langle v, sn \rangle_j$ propagated by servers just after the mobile Byzantine agents moved. Set $echo\_read_i$ stores identifiers of concurrently reading clients in order to notify cured servers and expedite termination of read().
- $pending\_read_i$: 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 $echo\_vals_i$ and returns, if they exist, 3 tuples $\langle v, sn \rangle$, such that there exist at least $\#echo_{CUM}$ occurrences in $echo\_vals_i$ of such tuple. If more than 3 of such tuples exist, the function returns the tuples with the highest sequence numbers.
- 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 at least $\#reply_{CUM}$ times. If there are more pairs with the same occurrence, it returns the one with the highest sequence number.
- 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., $V_i = \{\langle v_a, 1 \rangle, \langle v_b, 2 \rangle, \langle v_c, 3 \rangle, \langle v_d, 4 \rangle\}$ 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 $\{\langle v_c, 3 \rangle, \langle v_d, 4 \rangle, \langle v_f, 5 \rangle\}$.

The maintenance() operation. Such operation is executed by servers periodically at any time $T_i = t_0 + i\Delta$. Each server first checks if there are expired values in W_i then all the content of V_{safe_i} is stored in V_i and all V_{safe_i} and $echo\_vals_i$ sets are reset. Each server broadcast an ECHO message with the content of V_i, W_i (purged of the timer information) and the set $pending\_read_i$. When there is a value in $echo\_vas_i$ set that occurs at least $\#echo_{CUM}$ times, it updates V_{safe_i} set by invoking select_three_pairs_max_sn($echo\_vals_i$) function. To conclude, after δ time since the beginning of the operation, the W_i set is pruned from expired values and V_i is reset. Informally speaking, at this point V_i is no more used, since V_{safe_i} during the maintenance() operation is filled with values, then the content in V_i is not more necessary.

The write() operation. When the writer wants to write a value v, it increments its sequence number csn and propagates v and csn to all servers. Then it waits for δ time units (the maximum message transfer delay) before returning.

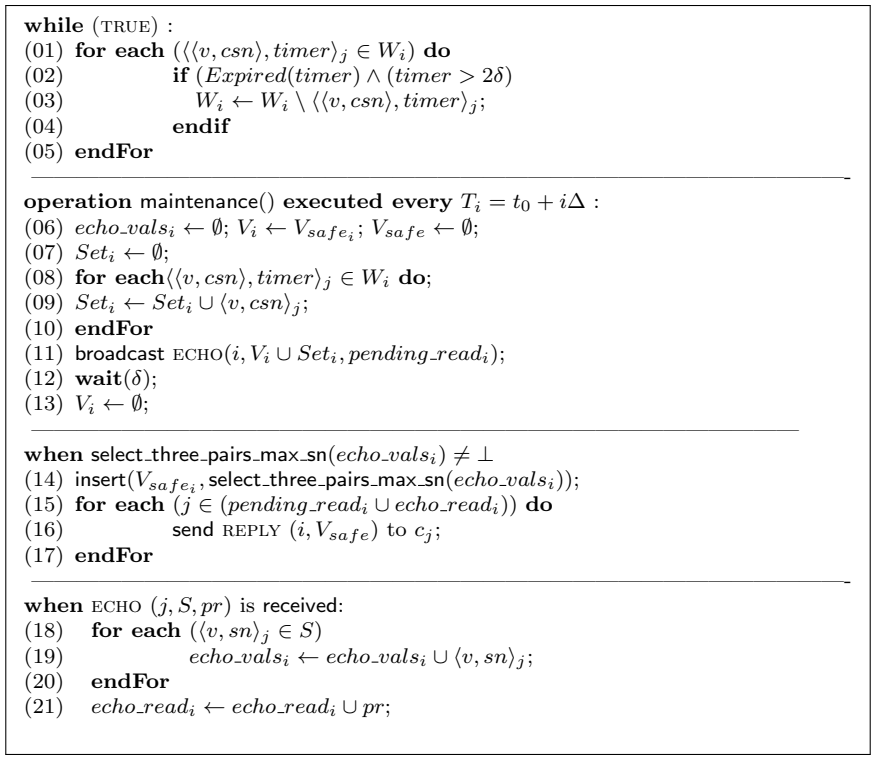

Figure 25: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CUM)$ model.

When a server s_i delivers a WRITE, it stores v in W_i. Then server sends a reply carrying such value to each reading client and broadcast such value as an ECHO() message to other servers.

The read() operation. When a client wants to read, it broadcasts a READ() request to all servers and waits 3δ time to collect replies. When it is unblocked from the wait statement, it selects a value v occurring $\#reply_{CUM}$ number of times from the $reply_i$ set, sends an acknowledgement message to servers to inform that its operation is now terminated and returns v as result of the operation.

When a server s_i delivers a READ(j) message from client c_j it first puts its identifier in the set $pending\_read_i$ to remember that c_j is reading and needs to receive possible concurrent updates, then s_i sends a reply back to c_j. Note that, in the REPLY() message is carried the result of conCut(V_i, V_{safe_i}, W_i). In this case, if the server is correct then V_i contains valid values, and V_{safe_i} contains valid values by construction, since it comes from values sent during the current maintenance(). If the server is cured, then V_i and W_i may contain any value. Thus, considering the function conCut(), a cured server may send a non valid value during 2δ time. Finally, s_i forwards a READ_FW message to inform other servers about c_j read request. This is useful in case some server missed the READ(j) message as it was affected by mobile Byzantine agent when such message has been delivered.

When a READ_FW(j) message is delivered, c_j identifier is added to $pending\_read_i$ set, as when the read request is just received from the client.

When a READ_ACK(j) message is delivered, c_j identifier is removed from both $pending\_read_i$ and $echo\_read_i$ sets as it does not need anymore to receive updates for the current read() operation.

```
========== Client code ==========
operation write(v):
(01) csn ← csn + 1;
(02) broadcast WRITE(v, csn);
(03) wait (δ);
(04) return write_confirmation;

========== Server code ==========
when WRITE(v, csn) is received:
(05)   W_i ← W_i ∪ ⟨⟨v, csn⟩, setTimer(2δ)⟩;
(06)   broadcast ECHO(i, ⟨v, csn⟩, pending_read_i);
(07)   for each j ∈ (pending_read_i ∪ echo_read_i) do
(08)       send REPLY (i, {⟨v, csn⟩});
(09)   endFor
```

Figure 26: $\mathcal{A}_W$ algorithm implementing the write(v) operation in the $(\Delta S, CUM)$ model.

```
========== Client code ==========
operation read():
(01) reply_i ← ∅;
(02) broadcast READ(i);
(03) wait (3δ);
(04) ⟨v, sn⟩ ← select_value(reply_i);
(05) broadcast READ_ACK(i);
(06) return v;
---------------------------------
when REPLY (j, V_set) is received:
(07)   for each (⟨v, sn⟩ ∈ V_set) do
(08)           reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(09)   endFor


========== Server code ==========
when READ (j) is received:
(10)   pending_read_i ← pending_read_i ∪ {j};
(11)   send REPLY (i, conCut(V_i, V_safe_i, W_i));
(12)   broadcast READ_FW(j);
---------------------------------
when READ_FW (j) is received:
(13)   pending_read_i ← pending_read_i ∪ {j};
---------------------------------
when READ_ACK (j) is received:
(14)   pending_read_i ← pending_read_i \ {j};
(15)   echo_read_i ← echo_read_i \ {j};
```

Figure 27: $\mathcal{A}_R$ algorithm implementing the read() operation in the $(\Delta S, CUM)$ model.

6.2 Correctness proofs

Definition 14 (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 15 ($Max\tilde{B}(t, t+T)$) *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 13 *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. $\square_{Lemma\ 13}$

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. Moreover we show that V_i is big enough to do not be full filled with new values before t'.

Lemma 14 *If a 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 26). $\square_{Lemma\ 14}$

Lemma 15 *If a client c_i invokes* 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 03-06, Figure 27). $\square_{Lemma\ 15}$

Theorem 10 (ss-Termination) *Any operation invoked on the register eventually terminates.*

Proof The proof simply follows from Lemma 14 and Lemma 15. $\square_{Theorem\ 10}$

Lemma 16 (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 hypotheses at T_i there are $\#echo_{CUM}$ correct servers s_j storing the same v and running the code in Figure 25. In particular each server broadcasts a ECHO() message with attached the content of V_j which contains v (line 11). Messages sent by $\#echo_{CUM}$ correct servers are delivered by s_c and stored in $echo\_vals_c$. The system is synchronous, thus by time $Ti + \delta$ function select_pairs($echo\_vals_c$) returns v. $\square_{Lemma\ 16}$

Lemma 17 *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 cured servers s_c at time T_c. At the beginning of the maintenance() operation s_c broadcasts values contained in V_i and W_i (Figure 25 line 11). V_i is reset at each operation with the content of V_{safe_i} which is reset at each operation (line 06). It follows that s_c broadcasts non valid values contained in V_i only during the maintenance() operation run a 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: $k = 1$ and $k = 2$.
case $k = 1$**:** In this case since $\Delta \geq 2\delta$ and the maximum value of the timer associated to a value is 2δ, then each cured server s_c broadcasts a non valid value contained in W_i only during the first maintenance() operation. Thus, 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 16 this is the necessary condition to return v' invoking select_pairs($echo\_vals_i$), leading to a contradiction.
case $k = 2$**:** $\delta \leq \Delta < 2\delta$ and the maximum value of the timer associated to a value is 2δ, then 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 16 this is the necessary condition to return v' invoking select_pairs($echo\_vals_i$), leading to a contradiction and concluding the proof. $\square_{Lemma\ 17}$

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

Corollary 5 *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 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 6 *Protocol $\mathcal{P}$ implements a* maintenance() *operation that implies $\gamma \leq 2\delta$.*

Lemma 18 *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 manteneance() operation and after δ time is reset (cf. Figure 25 lines 12-13). The second one is emptied at the beginning of each manteneance() operation and the third one keeps its value during k maintenance() operations (cf. Corollary 5). Thus by time $T_c + 2\delta$ s_c cleans all the values that could come from a mobile agent. $\square_{Lemma\ 18}$

Lemma 19 (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 WRITE message it executes line 05 Figure 26, it stores the value in W_j and sets the associated timer to 2δ.
For Lemma 13 in the $[t, t + \delta]$ time interval there are maximum $2f$ Byzantine servers, thus at $t + \delta$ $v \in W_j$ at $n - 2f \geq \#echo_{CUM}$ correct servers. All those servers broadcast v by time $t + \delta$, so by time $t + 2\delta$ there are $\#echoi_{CUM}$ 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 16, all correct servers are storing $v \in V_{safe_i}$ and by construction v is returned by the function conCut() by construction. We conclude the proof by considering that there are at least $n - 2f \geq \#reply_{CUM}$. $\square_{Lemma\ 19}$

For simplicity, from now on, given a write() operation op_W we call $t_B(op_W) + 3\delta = t_{wC}$ the **completion 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 20 (Step 3.) *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 19 at time t_{wC} 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 16, 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 movements 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 16)). By hypotheses there are no more write() operation, then v is never overwritten and all correct servers store v forever.

$\square_{Lemma\ 20}$

Lemma 21 (Step 3.) *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_{wC} be its completion 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 20 if there are no more write() operation then v, after t_{wC}, 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. $\square_{Lemma\ 21}$

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, cf. Figure 28. If $k = 2$ there can be up to $4f$ (cf. Lemma 13) Byzantine servers and $2f$ cured servers. If $k = 1$ there can be up to $3f$ Byzantine servers (cf. Lemma 13) and f cured servers. In Figure 28 we depicted the extreme case in which there is a read() operation just after the last write() operation. T_{wC} lines represent the time at which for sure correct servers replies with the last written value. Notice that when $\delta = \Delta$ s_4 has just the time to correctly reply to the client before being affected. In both cases there are $\#reply_{CUM}$ correct servers that replies with the last written value and the number of those replies is greater than number of replies coming from cured and Byzantine servers.

Theorem 11 (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. Figure 28). 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 20 after the write completion time t_{wC} 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 completion time of op_{W_0}, $t_{wC_0} < t_B(op_{W_0}) + 3\delta < t_B(op_R) + 2\delta < t_B(op_{W_3})$. 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 18 Byzantine and cured servers can no force correct servers to store and thus to reply with a never written value. Only cured and

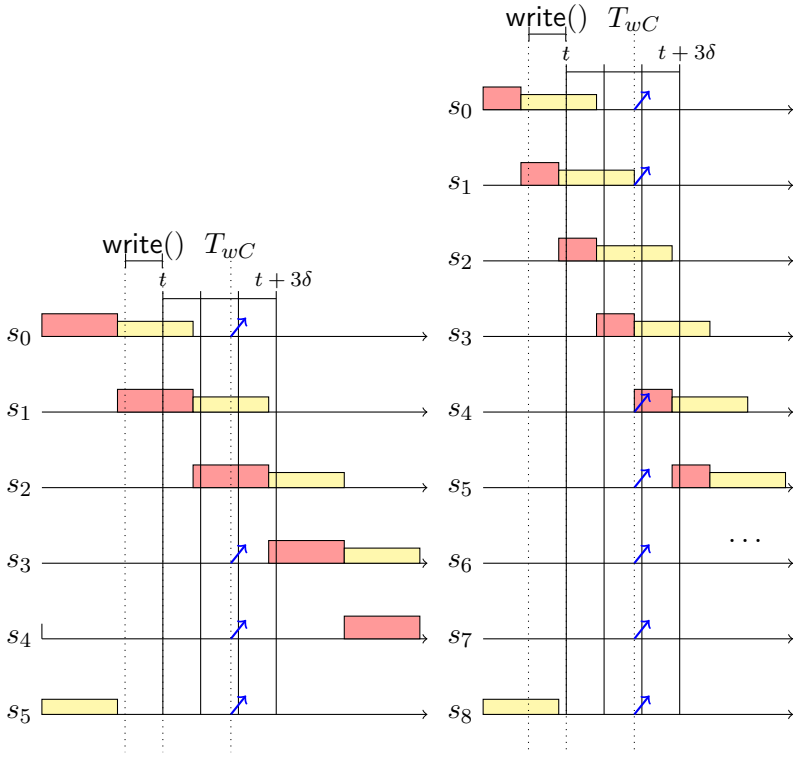

Figure 28: In the first scenario $\Delta \geq 2\delta$ and in second one is $\Delta \geq \delta$. Blue arrows are the correct replies sent back by correct servers.

Byzantine servers can reply with non valid values. As we stated, if $k = 1$ there are up to $4f$ non correct servers. If $k = 2$ 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. $\square_{Theorem\ 11}$

Theorem 12 *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 SWMR Regular Register in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model.*

Proof The proof simply follows from Theorem 10 and Theorem 11. $\square_{Theorem\ 12}$

Theorem 13 *Protocol $\mathcal{P}_{Rreg}$ is tight with respect to the number of replicas.*

Proof The proof simply follows considering that Theorems 10-11 proved that $\mathcal{P}_{Rreg}$ works with bounds provided in Table 3. Those match the previously known lower bounds proved in Theorem 4 and Theorem 6 for the $(\Delta S, CUM)$ model. $\square_{Theorem\ 13}$

7 Conclusion

This paper addressed the problem of emulating multi-reader regular registers under the MBF adversarial model for round-free computations. We first formalized MBF adversarial model in order to capture dynamic failures in generic (round-free) distributed computations and then we studied solvability issues raised by this powerful adversary. In particular, we proved that in the presence of mobile Byzantine agents a new operation, namely *maintenance*() must be defined. Then, we proved that in asynchronous distributed systems it is not possible to emulate a safe or regular register even in the presence of one Byzantine agent governed by the weakest $(\Delta S, CAM)$ adversary. We then considered the case of round-free synchronous systems and we proved that an emulation of an optimal regular register is possible against $(\Delta S, CAM)$ adversary provided that, $n_{CAM} \geq 4f + 1$ if $2\delta \leq \Delta < 3\delta$ and $n_{CAM} \geq 5f + 1$ if $\delta \leq \Delta < 2\delta$. While an emulation of an optimal regular register is possible against $(\Delta S, CUM)$ adversary provided that, $n_{CUM} \geq 5f + 1$ if $2\delta \leq \Delta < 3\delta$ and $n_{CUM} \geq 8f + 1$ if $\delta \leq \Delta < 2\delta$. We currently are investigating the solvability of other distributed building blocks under the proposed models.

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] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.