Optimal Mobile Byzantine Fault Tolerant Distributed Storage

[Extended Abstract]

Silvia Bonomi
DIAG - University of Rome “La Sapienza”
Rome, Italy
bonomi@dis.uniroma1.it

Antonella Del Pozzo
DIAG - University of Rome “La Sapienza”
Rome, Italy
Sorbonne Universités, UPMC, LIP6-CNRS 7606
Paris, France
delpozzo@dis.uniroma1.it

Maria Potop-Butucaru
Sorbonne Universités, UPMC, LIP6-CNRS 7606
Paris, France
maria.potop-butucaru@lip6.fr

Sébastien Tixeuil
Sorbonne Universités, UPMC, LIP6-CNRS 7606
Paris, France
sebastien.tixeuil@lip6.fr

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.

ACM ISBN .
DOI:

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 [16]. 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 con-

straints: *(i)* Byzantines with constrained mobility [6] 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, 7, 12, 13, 14] may move independently of protocol messages.

Buhrman *et al.* [6] 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, 7, 12, 13, 14]. Reischuk [13] considers that malicious agents are stationary for a given period of time. Ostrovsky and Yung [12] introduce the notion of mobile viruses and define the adversary as an entity that can inject and distribute faults. Garay [7], and more recently Banu *et al.* [1], and Sasaki *et al.* [14] 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, 7, 14] lies in the knowledge that processes have about their previous infection by a Byzantine agent. In Garay's model [7], 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 [7] 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.* [14] 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* [15], and *Byzantine quorum systems* [2, 9, 11, 10]. 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, 7, 14].

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 a protocol (Section 5) 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, 6, 7, 12, 13, 14].

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 consider two types of systems: *(i)* asynchronous (see Section 4.2), and *(ii)* round-free synchronous (see Section 5).

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 $\mathsf{broadcast}(m)$ 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, 6, 7, 12, 13, 14] 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 are to 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, 7, 6, 14] are summarized as follows:

- *Garay's model* [7]. 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* [14]. 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 [14] model a cured process still acts as a Byzantine one extra round.
- *Buhrman's model* [6]. Differently from the previous models, agents move together with the message (i.e., with the send or $\mathsf{broadcast}$ operation). However, when a server is in the *cured* state it is aware of that.

Most of the previously cited models [1, 3, 6, 7, 14] 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, 6, 7, 14], 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 us denote by $\hat{H} = (H, \rightarrow)$ 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 denote as $Co(t)$ the set of correct processes at time t while, given a time interval $[t, t']$. We 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).*

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

We denote as $B(t)$ the set of faulty processes at time t while, given a time interval $[t, t']$. We denote as $B([t, t'])$ the set of all the processes that are faulty during the whole interval $[t, t']$ (i.e., $B([t, t']) = \bigcap_{\tau \in [t,t']} B(\tau)$).

DEFINITION 5 (CURED PROCESS AT TIME t). *Let $\hat{H} = (H, \rightarrow)$ be an execution history and let $\mathcal{P}$ be the protocol generating $\hat{H}$. A process is said to be* cured at time t *if* (i) *it is correctly executing its protocol $\mathcal{P}$, and* (ii) *its state is not a valid state at time t. We denote as $Cu(t)$ the set of cured processes at time t while, given a time interval $[t, t']$. We 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 now 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 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 takes 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 $t_0 + \Delta$, $t_0 + 2\Delta$, ..., $t_0 + i\Delta$, 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

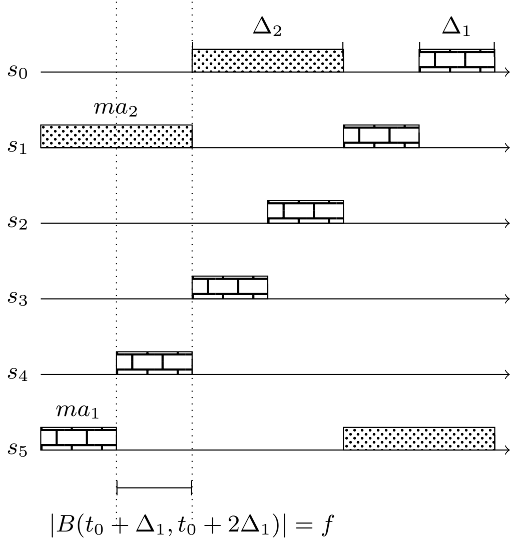

Figure 3: Example of a $(ITB, *)$ run with $f = 2$.

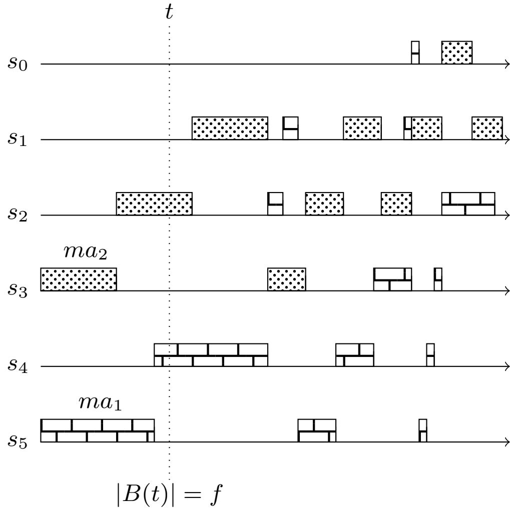

Figure 4: Example of a $(ITU, *)$ run with $f = 2$.

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.

- $(*, CUM)$ 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. The implementation of the oracle is out of scope of this paper and the reader may refer to [12] 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

[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 always introduce a corrupted value. A server cannot distinguish between a correct client and a Byzantine one.

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 regular register (SWMR) [8] 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 [8]). A read operation on a safe register concurrent with a write operation may return any value in the register domain.

We consider in the sequel only execution histories related to the register computation. In particular, the set of relevant computation events H are defined by the set of all the operations issued on the register and the happened-before relation is substituted by the precedence relation $\prec$ between operations. Thus, we 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 us denote by $\hat{H}_R = (H, \prec)$ 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. In the sequel, we consider $\mathcal{P}_{reg} \subseteq \{\mathcal{A}_R, \mathcal{A}_W\}$, where $\mathcal{A}_R$ is the algorithm implementing the read() operation and $\mathcal{A}_W$ is the algorithm implementing the write() operation. We 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 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. Due to the lack of space, we report here only the statement of the main Theorems. Complete proofs can be found in [5].

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.*

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 $\mathcal{A}_M$). *Let us define a* maintenance() *operation as 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]. 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., $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$).

The next theorem proves that the safe register and consequently the regular register cannot be implemented in asynchronous settings in MBF model. The basic intuition is that, even considering the weakest adversary $(\Delta S, CAM)$ and a single mobile Byzantine agent the maintenance() operation is not implementable and the agent may be able to compromise every server before the maintenance() completion.

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 an asynchronous system.*

4.3 Lower bounds

In the following we present lower bounds with respect to the number of correct servers to implement a safe register in the $(\Delta S, CAM)$ model, the extension to the $(\Delta S, CUM)$ model is presented in [5]. In particular we consider different cases depending on the relationship between Δ and δ. Due to lack of space, the corresponding theorems and proofs are delegated to [5].

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

5. IMPLEMENTING AN OPTIMAL REGULAR REGISTER

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. The same protocol with small modifications

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

implements a SWMR Regular Register in the $(\Delta S, CUM)$ model. This extension is proposed in [5]).

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 5 - 7, 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, while considering the CAM model, 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() 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

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

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

```
operation maintenance() executed every T_i = t_0 + Δ_i :
(01) cured_i ← report_cured_state();
(02) if (cured_i) then
(03)    V_i ← ∅; echo_vals_i ← ∅; echo_read_i ← ∅;
(04)    wait(δ);
(05)    insert(V_i, select_three_pairs_max_sn(echo_vals_i));
(06)    cured_i ← false;
(07)    for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(08)              send REPLY (i, V_i) to c_j;
(09)    endFor
(10)    else
(11)          broadcast ECHO(i, V_i, pending_read_i);
(12)          if(∄⟨⊥, 0⟩ ∈ V_i)then
(13)              fw_vals_i ← ∅; echo_vals_i ← ∅;
(14)          endif
(15) endif
--------------------------------------------------------------------
when ECHO (j, V_j, pr) is received:
(16)    echo_vals_i ← echo_vals_i ∪ V_j;
(17)    echo_read_i ← echo_read_i ∪ pr;
```

Figure 5: $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CAM)$ model.

reply to a reading client as soon as possible even in case they delivered such value when affected by mobile Byzantine agents.

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.

5.2 Correctness Proofs

We report here only the statement of the main Lemmas and Theorems. Complete proofs can be found in [5].

LEMMA 1. *If a correct client c_i invokes* write(v) *operation at time t then this operation terminates at time $t + \delta$.*

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

THEOREM 3 (TERMINATION). *If a correct client c_i invokes an operation, c_i returns from that operation in finite time.*

LEMMA 3. *Let op be a* write(v) *operation invoked by a correct client at time t. Any $s_j \in Co([t, t+\delta])$ has v in V_j at time $t + \delta$.*

LEMMA 4. *If* (i) *$k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), and* (ii) *$n_{CAM} \geq (k+3)f + 1$, then at time $T_2 - 1$ (where $T_2 - 1 = t_0 + 2\Delta - 1$) there exists at least $(k+1)f + 1$ correct servers storing locally a valid value v (i.e., $v \in VVS(T_2 - 1)$).*

LEMMA 5. *If* (i) *$k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), and* (ii) *$n_{CAM} \geq (k+3)f + 1$, then at time $T_i - 1$ (where $T_i - 1 = t_0 + i\Delta - 1$) there exists at least $(k+1)f + 1$ correct servers storing locally a valid value v (i.e., $v \in VVS(T_i - 1)$).*

COROLLARY 1. *If* (i) *$k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), and* (ii) *$n_{CAM} \geq (k+3)f + 1$, then there always exist at least $(k+1)f + 1$ correct servers storing locally a valid value v (i.e., $v \in VVS(T_i - 1)$).*

COROLLARY 2. *If* (i) *$k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$), and* (ii) *$n_{CAM} \geq (k+3)f + 1$, then each server that became cured at T_i, at $T_i + \delta$ has at least one valid value.*

LEMMA 6. *Let op be a* read() *operation issued at time t and terminating at time $t + 2\delta$. Let $Max\tilde{B}(t, t + 2\delta)$ be the*

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

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

maximum number of servers that can be faulty for at least one time unit in the interval $[t, t+2\delta]$. If (i) $k\Delta \geq 2\delta$ *(with $k \in \{1,2\}$), and* (ii) $n_{CAM} \geq (k+3)f+1$*, then* $|Co(t+\delta)| > Max\bar{B}(t, t+2\delta)$.

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.*

THEOREM 4 (VALIDITY). *If* (i) $k\Delta \geq 2\delta$ *(where $k \in \{1,2\}$), and* (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.*

THEOREM 5. *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 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.*

6. 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 $\delta \leq \Delta < 2\delta$. The extension of the current reported results to the $(\Delta S, CUM)$ model is provided in [5]. We currently are investigating the solvability of other distributed building blocks under the proposed models.

Acknowledgements

The authors would like to thank Francois Bonnet and Marc Shapiro for their comments and suggestions on a previous version of this paper. For Italian authors, this work is partially supported by the Ateneo 2015 project (grant n. C26A157FBX) and by the EURASIA project.

7. 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, Singapore, January 2016. ACM.

[5] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal mobile byzantine fault tolerant distributed storage. *(Available on line on arXiv)*, 2016.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.