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, sebastien.tixeuil}@lip6.fr

Abstract

We present the first emulation of a server based regular read/write storage in a synchronous round-free message-passing system that is subject to mobile Byzantine failures. 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 moves of the mobile Byzantine agents are decided by the adversary with the only constraints of happening periodically every Δ time units (period that is completely decoupled from the message communication delay). Our emulation spans two models: servers self-diagnose their state (that is, servers are aware that the mobile Byzantine agent left), and servers without self-diagnose mechanism.

Our results related to the threshold of the tolerated mobile Byzantine faults 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 that is available in 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.

Keywords: mobile Byzantine failures, regular register, round free synchronous computation

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

1 Introduction

Cloud Computing is one of the most popular recent technologies. In a nutshell, clouds offer to organizations (*a.k.a.* clients) the possibility to store and access vast amounts of data on remote servers managed by providers. While the cloud model is very similar to the well known client-server model, clouds typically experience usual faults and errors that occur in any distributed system, but are also often the target of cyber attacks. Recently, due to multiple causes such as software bugs, errors in internal migrations, or cyber attacks, have been reported [1, 2, 3, 12] outages of known cloud providers (Google, Akamai, Microsoft). It is thus highly desirable that such distributed storage systems are able to mask to their clients the unexpected yet possible faulty behaviors of their servers. In these architectures where clients request permanent availability, applying the classical technique consisting in restarting the system anytime an error, a fault, or an attack is diagnosed is impossible due to temporary system unavailability. In this context, attack tolerant schemes studied in theoretical distributed computing become extremely relevant for the daily practice of cloud computing. Many attack models for distributed systems have been proposed, but one of the most general is the *Byzantine* model proposed by Lamport et al. [15], which simply assumes that faulty nodes can behave arbitrarily. Byzantine-tolerant storage problem has been studied in various settings and models (see *e.g.*, [5, 8, 9, 16, 17, 22] to cite just few of them). In all the aforementioned works, the set of nodes with an arbitrary behavior does not change during the entire computation (*i.e.*, they are static).

However, when a distributed system is subject to viruses propagation or cyber attacks, the faults are typically non-stationary since the attack or the viruses can propagate in the network. Conversely, previously faulty nodes may recover as countermeasures to attacks are run periodically in the system. These observations led researches to investigate a different distributed adversarial model where the set of Byzantine nodes varies throughout the execution. Such a model has been formalized starting with the pioneering works of Reischuk [20] (for the Byzantine agreement problem), and Ostrovsky and Yung [19] (that show how to compute an arbitrary global function with high probability), where it is assumed that a constant fraction of the nodes can be corrupted in a given period of time. In the mobile Byzantine fault model, transient arbitrary state corruptions, which can be abstracted as Byzantine "agents", are controlled by an omniscient adversary and moved through the network in order to corrupt the nodes they occupy. A node occupied by a Byzantine agent then behaves arbitrarily for some transient period of time. Once the Byzantine agent leaves the node, the node eventually behaves correctly. However, the Byzantine agent may "infect" another node, that previously behaved correctly.

2 Related Work and Contributions

In Mobile Byzantine Failures models, there are two main research directions: (i) Byzantines with constrained mobility and (ii) Byzantines with unconstrained mobility. Byzantines with constraint mobility were studied by Buhrman *et al.* [10]. They consider that Byzantine agents move from one node to another only when protocol messages are sent (similar to how viruses would propagate). In [10], Buhrman *et al.* studied the problem of Mobile Byzantine Agreement. They proved a tight bound for its solvability (i.e., $n > 3t$, where t is the maximal number of simultaneously faulty processes) and proposed a time optimal protocol that matches this bound.

In the case of unconstrained mobility the motion of Byzantine agents is not tied to message exchange. Several authors investigated the agreement problem in variants of this model: [4, 6, 13, 19, 20, 21]. Reischuk [20] investigate the stability/stationarity of malicious agents for a given period of time. Ostrovsky and Yung [19] introduced the notion of mobile virus and investigate an adversary that can inject and distribute faults.

Round Based [7]		Round Free		
Garay [13]	$n > 3f$	CAM	$\Delta > 4\delta$	$n > 3f$ (Th. 6)
			$k\Delta > 2\delta$ $k \in \{0, 1\}$	$n > (k+3)f$ (Th. 3)
Bonnet [6], Sasaki [21]	$n > 4f$	CUM	$\Delta > 4\delta$	$n > 4f$ (Th. 3)
			$k\Delta > 2\delta$ $k \in \{0, 1\}$	$n > (k+1)2f$ (Th. 3)

Figure 1: Summary of results and Comparison with the round-based approach.

Furthermore, they advocate that the unconstraint mobility model abstracts the concept of insider threats (hacker, cracker, black hat) or attacks (DOS, Worms, viruses or Trojan horses).

Garay [13] and, more recently, Banu *et al.* [4] and Sasaki *et al.* [21] or Bonnet *et al.* [6] consider, in their models, that processes execute synchronous rounds composed of three phases: *send*, *receive*, *compute*. Between two consecutive rounds, Byzantine agents can move from one node to another, hence the set of faulty processes has a bounded size although its membership can change from one round to the next. The main difference between the unconstrained models presented so far is in the knowledge that processes have been affected from a Byzantine agent. In the Garay's model a process has the ability 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 [13] proposed, in this model, an algorithm that solves Mobile Byzantine Agreement provided that $n > 6t$ (dropped later to $n > 4f$ in [4]). Bonnet *et al.* [6] investigated the same problem in a model where processes do not have the ability to detect when Byzantine agents move. However, differently from Sasaki *et al.* [21], cured processes have *control* on the messages they send. This subtle difference on the power of Byzantine agents has an impact on the bounds for solving the agreement. If in the Sasaki's model the bound on solving agreement is $n > 6f$ in Bonnet's model it is $n > 5f$ and this bound is proven tight.
Let us note that all the model discussed so far are applied mainly in round-based computations.

BFT and MBFT Register Emulations Traditional solutions to build a Byzantine tolerant storage service can be divided into two categories: *replicated state machines* [22] and *Byzantine quorum systems* [5, 16, 18, 17]. Both the approaches are based on the idea that the state of the storage is replicated among processes and the main difference is in the number of replicas involved simultaneously in the state maintenance protocol.

Bonomi *et al.* [7] investigated the emulation of a distributed storage on top of round-based synchronous system in four of the mobile Byzantine models cited above: Garay [13], Buhrman *et al.* [10], Sasaki *et al.* [21], and Bonnet *et al.* [6]. The respective number f of simultaneous Byzantine servers that those implementations can withstand using n servers are $n > 3f$, $n > 4f$, $n > 4f$, and $n > 2f$, respectively.

Our contribution In this paper, we extend the work presented in [7] to the synchronous round free communication model the message transfer delay is bounded by a known constant δ. We first introduce two new Byzantine Mobile Failure models for the *round-free* environment, namely CAM and CUM (extensions of the models introduced by Garay [13] respectively Bonnet *et al.* [6]), and we propose a protocol emulating a distributed storage in these models. Diferently from previous work [4, 13, 21, 6] we do not assume that a core of correct servers always exists. That is, all processes could be corrupted throughout the execution. Our adversarial model follows the lines of research by Ostrovsky and Yung [19]. However, we do not retain their round-based coupling between protocol steps and virus moves. In our model, the adversary controls, at any time, a set of f Byzantine agents that represent a fraction of the total number n of servers and that move periodically every Δ time units. We restrict our attention to a subset of representatives of this infinitely powerful adversary by relating the period Δ to the maximum message transfer delay δ. In particular, we started from a scenario where the frequency of the movement is comparable to the round-based scenario (i.e., $\Delta > \delta$) computing the number of servers that our emulation needs in order to tolerate f Mobile Byzantine Failures and then we modify the emulation to improve its performance in the CAM model when the period of movement is much larger than δ (i.e., $\Delta > 4\delta$).

Our results show a difference between the *round-based* approach of Bonomi *et al.* [7] and the round-free case (summarized in Figure 1). It should be noted that in the round-free computation the number of replicas needed to tolerate f Mobile Byzantine Failures does not depend only on f but also on the ratio between Δ and δ.

3 System Model

We consider a distributed system composed of an arbitrary large set of client processes $\mathcal{C}$ and a set of n server processes $\mathcal{S} = \{s_1, s_2 \dots s_n\}$. Each process in the distributed system (*i.e.*, both servers and clients) is identified by a unique integer identifier. Servers run a distributed protocol implementing a shared memory abstraction.

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

The system is *synchronous* in the following sense: *(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.

Failure model. We assume that any client may fail by crashing. Contrarily, servers are affected by *Mobile Byzantine Failures* [20, 19, 4, 6, 13, 10, 21, 6] as specified in the following.

3.1 Mobile Byzantine Failure Model in Round-free computation

As in the general Mobile Byzantine Failure model introduced by Ostrovsky and Yung [19], we assume that faults are represented by Byzantine agents managed by a powerful adversary that moves the Byzantine agents from a server to another. When the Byzantine agent is hosted by a server, the adversary takes the entire control of the server (i.e. it can corrupt the server's local variables, force the server to send arbitrary messages breaking the broadcast specification or execute a different protocol). We assume that at any time t, at most f servers can be affected by a mobile Byzantine failure; however, during the system life all servers may be transitory affected by a Byzantine failure.
When an agent occupies a server s_i, s_i is *faulty*. When the agent leaves s_i, s_i is *cured* until it restores its correct internal state. If a server is neither *faulty* nor *cured*, then it is *correct*. As all the previous works [20, 19, 4, 13, 10, 21, 6], we assume that each server has a tamper-proof memory where it safely stores the correct algorithm code that can be retrieved when needed.

Differently from the approach adopted in [4, 13, 21, 6] and similar to [19], the Mobile Byzantine Failure models we consider in this paper totally decouples the scheduling of the Byzantine agents mobility from the message transmission. In particular, we assume that Byzantine agents move periodically every Δ time units. The length of the period is totally unrelated from the communication time; this means that depending on how large Δ is the Byzantine agent may move 0, 1, or multiple times while a single message is travelling. In Figure 3.1 is depicted a comparison between the round free and the round based scenario. Similarly to [13, 21, 6], we consider the possibility that a cured server may or may not be aware about its state.

More in details, we will consider the following two models:

- **Cured Aware Model (CAM)**: Byzantine agents move arbitrarily from a server to another every Δ time units. When a server is in the *cured* state, it is aware of its state (similar to the *Garay's model* [13]) and thus it can send specific messages to notify other processes about its state.

- **Cured Unaware Model (CUM)**: Byzantine agents move arbitrarily from a server to another every Δ time units. Differently from the previous model servers do not know if they are correct or cured when the Byzantine agent leaves the server. However, they execute the correct code (obtained from the tamper-proof memory).

In order to abstract the knowledge a server has on its state (i.e. *cured* or *correct*), we introduce the 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 [11], [19] for further details.

4 Regular Register Specification

A register is a shared variable accessed by a set of processes, called 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). 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 (called the invocation time and the reply time) according to the fictional global time.

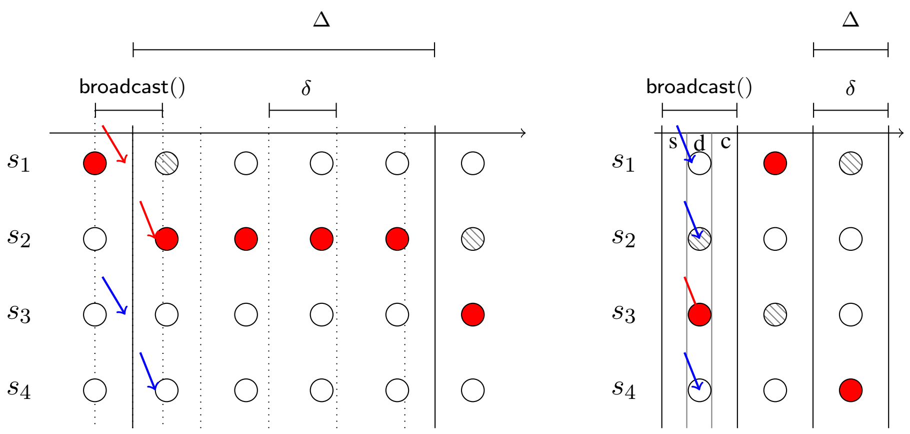

Figure 2: Comparation between the Round free model and the Round based one. In the Round free then a message is broadcast then, since it may happen during the Byzantine agent movements, it may be delivered by more than f Byzantine servers. This is not true in the Round based model, in which it is know a priori how many non faulty servers do deliver the message.

An operation *op* is *complete* if both the invocation event and the reply event occured (*i.e.*, the process executing 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, defined Lamport [14], which is specified as follows:

- **Termination**: if a correct client invokes an operation, it eventually returns from that operation (that is, the operation is complete).
- **Validity**: A read operation returns the last value written before its invocation (*i.e.* the value written by the latest write preceding it), or a value written by a write operation concurrent with it.

5 A Round-free Regular Register Implementation

In this section, we present an algorithm $\mathcal{A}_{Rreg}$ thta implements a SWMR Regular Register CAM and CUM failure models described in Section 3.1. Let us recall that mobile Byzantine agents move periodically from one server to another corrupting their internal states and making servers behaving arbitrarily. As a consequence, if not properly mastered, the movement of Byzantine agents can lead to the compromising of all the servers; in fact, when the agent leaves a server, the cured server may have arbitrary information stored locally. The result would be the loss of the register value and the consequent unavailability of the storage service.

A naive solution to master mobile Byzantine agents, could be to exploit write() operations to clean servers state and to increase the number of replicas n to ensure the presence of "enough" correct servers between two following operations. However, such solution has two strong drawbacks: (i) write() operations

Table 1: Parameters for $\mathcal{A}_{Rreg}$ and $\mathcal{A}^*_{Rreg}$ Protocols.

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

are not governed by servers and are invoked depending on clients protocols and (ii) the number of replicas needed to tolerate f mobile Byzantine agents will grow immediately linearly with the time between two consecutive write() operations. Since Byzantine agents movement is completely decoupled from the communication steps (i.e., Byzantine agents move independently from the events related to the communication), during a read() or a write() operation the number of servers behaving temporarily arbitrarily may be much larger than f, even if, at each time instant t they are bounded by f.

Our solution is based on the following key points:

- we define a state_maintenance protocol, executed periodically every Δ time units, that aims at updating the state of cured servers to a correct state (i.e., recovering the correct register value). In this way, the effect of a Byzantine agent on a server will totally disapear in a bounded period of time and the cured server can further be useful to the computation.
- 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 needed from cured server to become correct and the frequency of the Byzantine agent movement.
- 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. 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 the agent moves leaving cured servers without any trace of the message.

Concerning the frequency of Byzantine agents movement, we started by considering the case in which Byzantine agents move with a the period Δ that is large at least one communication step (i.e., $\Delta > \delta$). This means that for each message traveling in the network, Byzantine agents move at most once. Then we analyze cases with lower frequency of movement in order to understand the relation between the period Δ, the number of mobile Byzantine agents f and the total number of servers needed to provide a correct implementation.

Interestingly, we found that the number of replicas n needed to tolerate f Mobile Byzantine failures does not depend only on the value of f but also on the relation between Δ and δ. This is not true in the round-based model.

5.1 $\mathcal{A}_{Rreg}$ Detailed Description

The protocol $\mathcal{A}_{Rreg}$ is described in Figures 3 - 5.

Local variables at client c_i. Each client c_i maintains $reply_i$, that is used during the read() operation to collect the pairs $\langle v, sn \rangle$ sent back from servers. Additionally, the local sequence number csn is incremented each time a write() operation is invoked and is used to timestamp such operations.

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

- val_i and sn_i: two integer variables storing respectively the current value of the register known by s_i and its corresponding sequence number;
- old_val_i and old_sn_i, that store respectively the previous value of the register and its corresponding sequence number. These variables are initialized to $\perp$ and -1, respectively.
- $cured_i$: boolean flag updated by the cured_state oracle. In particular, while considering the CAM model, such variable will be set to true when s_i was occupied by the mobile Byzantine agent and is reset during the algorithm when s_i becomes correct. In CUM model $cured_i$ is always false.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated trough ECHO messages. The first one stores pairs $\langle v, sn \rangle$ propagated by servers just after the mobile Byzantine agents moved, while the second stores the set of concurrently reading clients, propagated trough echos, 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, we also define the following functions:

- select_pairs_max_sn($echo\_vals_i$): this function takes as input the set $echo\_vals_i$ and returns, if it exists, a 4-uple $\langle v_1, ts_1, v_2, ts_2 \rangle$, where $ts_2 = ts_1 + 1$, and there exist at least $\#reply_M$[1] occurrences in $echo\_vals_i$ of both $\langle v_1, ts_1 \rangle$ and $\langle v_2, ts_2 \rangle$. If more than such a 4-uple exist, the function returns the one with the highest sequence numbers.
- select_max_sn($echo\_vals_i$): this function takes as input the set $echo\_vals_i$, and returns, if it exists, a pair $\langle v, ts \rangle$ occurring at least $\#reply_M$ times in $echo\_vals_i$. If more than one pair satisfy the condition, it returns the one with the highest sequence number.
- select_value($reply_i$): this function takes as input the $reply_i$ set of replies collected by client c_i and returns, if exists, the pair $\langle v, sn \rangle$ occurring at least $\#reply_M$ times. If more pairs exist, it returns the one with the highest sequence number.

The state maintenance **Protocol.** The state maintenance protocol is executed by servers periodically, with period Δ. Servers start cleaning their local variables and then broadcast an ECHO message with current and past values attached, their sequence numbers, and the set $pending\_read_i$ containing identifiers of clients that are currently running a read() operation. After δ time units, servers try to update their state by checking the number of occurrences of each pair $\langle v, sn \rangle$ received trough echoes. In particular, they first check if there exist two pairs $\langle v, sn \rangle$ and $\langle v', sn' \rangle$ occurring each at least $\#reply_M$ times and such that $sn' = sn + 1$. If

[1]This threshold is set according to Table 1 depending both on the mobile Byzantine model (CAM vs. CUM) and the frequency of movement of the agents.

```
every $\Delta$ time units do:
(01) $cured_i \leftarrow$ report_cured_state(); $echo\_vals_i \leftarrow \emptyset$; $echo\_read_i \leftarrow \emptyset$;
(02) if $(\neg cured_i)$
(03)     then broadcast ECHO$(i, \langle val_i, sn_i \rangle, \langle old\_val_i, old\_sn_i \rangle, pending\_read_i)$;
(04)     else broadcast ECHO$(i, \langle \perp, 0 \rangle, \langle \perp, 0 \rangle, \emptyset)$;
(05) endif
(06) wait$(\delta)$;
(07) if $(\exists \langle -, \langle v, ts \rangle \rangle, \langle -, \langle v', ts+1 \rangle \rangle \in echo\_vals_i$ occurring at least $\#reply_M$ times)
(08)     then $\langle v_1, ts_1, v_2, ts_2 \rangle \leftarrow$ select_pairs_max_sn$(echo\_vals_i)$;
(09)          $val\_i \leftarrow v_2$; $sn_i \leftarrow ts_2$; $old\_val_i \leftarrow v_1$; $old\_sn_i \leftarrow ts_1$;
(10)     else if $(\exists \langle -, \langle v, ts \rangle \rangle \in echo\_vals_i$ occurring at least $\#reply_M$ times)
(11)             then $\langle v, ts \rangle \leftarrow$ select_max_sn$(echo\_vals_i)$;
(12)                  $old\_val_i \leftarrow v$; $old\_sn_i \leftarrow ts$; $val\_i \leftarrow \perp$; $sn_i \leftarrow 0$;
(13)          endif
(14) endif
(15) $cured_i \leftarrow$ false;
(16) for each $(j \in (pending\_read_i \cup echo\_read_i))$ do
(17)             send REPLY $(i, \langle val_i, sn_i \rangle, \langle old\_val_i, old\_sn_i \rangle)$ to $c_j$;
(18) endFor
------------------------------------------------------------
when ECHO $(j, \langle v, ts \rangle, \langle ov, ots \rangle, pr)$ is received:
(19)  $echo\_vals_i \leftarrow echo\_set_i \cup \{\langle j, \langle v, ts \rangle \rangle\}$;
(20)  $echo\_vals_i \leftarrow echo\_set_i \cup \{\langle j, \langle ov, ots \rangle \rangle\}$;
(21)  $echo\_read_i \leftarrow echo\_read_i \cup pr$.
```

Figure 3: $\mathcal{A}_{Rreg}$ state maintenance protocol (code for server s_i).

this holds, a server s_i can become correct by taking such two pairs as current and old values. Otherwise, s_i tries to find at least one pair $\langle v, sn \rangle$ occurring at least $\#reply_M$ times. In this case, s_i can deduce that there exists a concurrent write() operation that is updating the register value. Thus, s_i considers $\langle v, sn \rangle$ as the pair associated to the old value and updates its local state accordingly. 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 new value v, it increments its sequence number and propagates the value and corresponding sequence number to servers, then 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 then forward the message, trough a WRITE_FW$(i, \langle v, csn \rangle)$, to others to prevent its loss in case of mobile Byzantine agents movement. In addition, it also sends a REPLY() message to all clients that are currently reading (as far as it knows) to help them to terminate their read() operation.

When a WRITE_FW$(j, \langle v, csn \rangle)$ message is delivered, it is stored by s_i in its fw_vals_i set. Such set is constantly monitored and it is used together with the $echo\_vals_i$ set to find a couple $\langle v, sn \rangle$ that occurs at least $\#reply_M$ times and that represents the current value. This continuous check completes the update of local variables of servers just cured concurrently with the current write() operation.

The read() operation. When a reader wants to read, it broadcast a READ() request and then waits 2δ time (i.e., one round trip delay) for replies. When it is unblocked from the wait statement, it selects a value v occurring enough time from the $reply_i$ set, sends an acknowledgement message to server to inform that its operation is now ended and then 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 check if it is cured or not and in case it is not cured, it send a reply back to c_j. Note that, the REPLY() message contains both

```
operation write($v$):
(01) $csn \leftarrow csn + 1$;
(02) broadcast WRITE($v, csn$);
(03) wait ($\delta$);
(04) return write_confirmation;
```

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

```
when WRITE($v, csn$) is received:
(01) $old\_val_i \leftarrow val_i$; $old\_sn_i \leftarrow sn_1$; $val_i \leftarrow v$; $sn_i \leftarrow csn$; $cured_i \leftarrow$ false;
(02) for each $j \in (pending\_read_i \cup echo\_read_i)$ do
(03)     send REPLY $(i, \langle val_i, sn_i \rangle)$;
(04) endFor
(05) broadcast WRITE_FW($i, \langle v, csn \rangle$);
------------------------------------------------------------
when WRITE_FW($j, \langle v, csn \rangle$) is received:
(06) $fw\_vals_i \leftarrow fw\_vals_i \cup \{\langle j, \langle v, csn \rangle \rangle\}$;
------------------------------------------------------------
when $\exists \langle j, \langle v, sn \rangle \rangle \in (fw\_vals_i \cup echo\_vals_i)$ occurring at least $\#reply_M$ times:
(07) let $U = \{\langle v, sn \rangle \in (fw\_vals_i \cup echo\_vals_i) \mid \langle v, sn \rangle$ occurs at least $\#reply_M$ times and $sn > sn_i, sn_{>}old\_sn_i\}$
(08) for each $\langle v, ts \rangle \in U$;
(09)     if $(val\_i = \perp \wedge sn = old\_sn_i + 1)$
(10)       then $val\_i \leftarrow v$; $sn_i \leftarrow ts$;
(11)            $\forall j : fw\_vals_i \leftarrow fw\_vals_i \setminus \{\langle j, \langle v, ts \rangle \rangle\}$;
(12)            $\forall j : echo\_vals_i \leftarrow echo\_vals_i \setminus \{\langle j, \langle v, ts \rangle \rangle\}$;
(13)       else if $(val_i \neq \perp \wedge sn = sn_i + 1)$
(14)            then $old\_val_i \leftarrow val_i$; $old\_sn_i \leftarrow sn_i$; $val_i \leftarrow v$; $sn_i \leftarrow ts$;
(15)                 $\forall j : fw\_vals_i \leftarrow fw\_vals_i \setminus \{\langle j, \langle v, ts \rangle \rangle\}$;
(16)                 $\forall j : echo\_vals_i \leftarrow echo\_vals_i \setminus \{\langle j, \langle v, ts \rangle \rangle\}$;
(17)            endif
(18)     endif;
(19) endFor.
```

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

Figure 4: $\mathcal{A}_{Rreg}$ write() operation protocol.

current and old pairs $< value, ts >$. In any case, s_i forwards a READ_FW message to inform other servers about c_j read request, in case they missed the message as they were affected by the mobile Byzantine agents.

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

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

```
operation read():
(01) $reply_i \leftarrow \emptyset$;
(02) broadcast READ($i$);
(03) wait ($2\delta$);
(04) $\langle v, sn \rangle \leftarrow$ select_value($reply_i$);
(05) broadcast READ_ACK($i$);
(06) return $v$;
------------------------------------------------------------
when REPLY $(j, \langle v, ts \rangle, \langle ov, ots \rangle)$ is received:
(07)   $reply_i \leftarrow reply_i \cup \{\langle j, \langle v, ts \rangle \rangle\}$;
(08)   $reply_i \leftarrow reply_i \cup \{\langle j, \langle ov, ots \rangle \rangle\}$;
```

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

```
when READ ($j$) is received:
(01) $pending\_read_i \leftarrow pending\_read_i \cup \{j\}$;
(02) if $(\neg cured_i)$
(03)   then send REPLY $(i, \langle val_i, sn_i \rangle, \langle old\_val_i, old\_sn_i \rangle)$;
(04) endif
(05) broadcast READ_FW($j$);
------------------------------------------------------------
when READ_FW ($j$) is received:
(06) $pending\_read_i \leftarrow pending\_read_i \cup \{j\}$;
------------------------------------------------------------
when READ_ACK ($j$) is received:
(07) $pending\_read_i \leftarrow pending\_read_i \setminus \{j\}$;
(08) $echo\_read_i \leftarrow pending\_read_i \setminus \{j\}$;
```

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

Figure 5: $\mathcal{A}_{Rreg}$ read() operation protocol.

5.2 Correctness Proofs

We first establish the termination property of our algorithm, that is independent from the model considered (CAM, or CUM) and from the time period between two following movements of mobile Byzantine agents.

Definition 1 (Valid Value Set at time t) *The* valid value set *at time t, denoted by $VVS(t)$, is the set of values containing:* (i) *the value v written by the last* write() *that 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 at time t, $VVS(t)$ contains only $\perp$.*

Definition 2 (T_i) *The time of the i-th movement of mobile Byzantine agents is denoted as $T_i = t_0 + i\Delta$, where t_0 is the starting time and $i \in \mathbb{N}$.*

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

Proof The claim simply follows by considering that write() returns a write_confirmation to the calling client c_i after δ time, independently of the behaviour of the servers (see Lines 03-04, Figure 4(a)). $\Box_{Lemma\ 1}$

Lemma 2 *If a correct client c_i invokes* read() *at itme t, 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 5(a)). $\Box_{Lemma\ 2}$

Theorem 1 (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 1 and Lemma 2. $\Box_{Theorem\ 1}$

Lemma 3 *Let op be a* write(v) *operation invoked by a correct client at time t. Any correct server s_j in the the period $[t, t + \delta]$ has $val_j = v$.*

Proof Let us note that a write() operation takes exactly δ time unit to be executed. Thus, *op* is executed in the period $[t, t + \delta]$. During this period, the mobile Byzantine agents move, or do not move. We consider the two cases separately. In the following, let $T_i = t_0 + i\Delta$ denote the time of the i-th movement of the mobile Byzantine agents (with $i \in \mathbb{N}$).

- **Case 1 - $\forall i \in \mathbb{N}, T_i \notin [t, t + \delta]$.** Then, during the execution of the write() operation, the set of faulty servers does not change. At the beginning of the write() operation, the writer broadcasts a WRITE(v) message that is delivered to at least $n - f$ non-faulty servers before $t + \delta$. When this happens every non-faulty server s_j assigns v to val_j (see line 01 in Figure 4) and cured servers become correct. Considering that a WRITE(v) message takes at most δ time units to be delivered to its destination and that messages are not lost, the claim follows for this case.

- **Case 2 - $\exists i \in \mathbb{N}, T_i \in [t, t + \delta]$.** In this case, during the execution of the write() operation, the set of faulty servers changes. Let us call $F^-_{T_i}$ the set of faulty servers before time T_i and $F^+_{T_i}$ the set of faulty servers from T_i on. At the beginning of the write() operation, the writer broadcasts a WRITE(v) message that will be delivered by time $t + \delta$.

In the worse case, (i) $F_{T_i}^- \cap F_{T_i}^+ = \emptyset$, (ii) each server $s_j \in F_{T_i}^-$ delivered the WRITE(v) message before time T_i when it is faulty and (iii) each server $s_k \in F_{T_i}^+$ delivered the WRITE(v) message after time T_i when it is faulty. Thus, considering that faulty servers during the execution of the operation are at most $2f$, we have that at least $n - 2f$ servers remain correct during the whole operation execution (i.e., in the time interval $[t, t+\delta]$). Therefore, each of the $n-2f$ servers executes line 01 in Figure 4 updating its val_j with the value v. Considering that the WRITE(v) message is sent by the writer at time t and that each message takes at most δ time units to be delivered, the claim follows.

$\square_{Lemma\ 3}$

Lemma 4 *Let Δ be the time interval between two following movements of mobile Byzantine and let δ be the upper bound on the message transfer delay. Let t_0 be the starting time of the computation and let $T_i = t_0 + i\Delta$ be the time of the i-th movement of the mobile Byzantine agent (with $i \in \mathbb{N}$). If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) (ii) $n_{CAM} \geq (k+3)f+1$ and (iii) $n_{CUM} \geq 2(k+1)f+1$, 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 in the CAM model and $(k+2)f+1$ correct servers in the CUM model storing locally a valid value v (i.e., $v \in VVS(T_2-1)$).*

Proof

Let $\perp$ be the initial value of the register at time t_0. Let us note that, in the period $[t_0, T_1 - 1]$ (with $T_1 - 1 = t_0 + \Delta - 1$) there are no changes in the set of faulty servers. Thus, due to Lemma 3, at time $T_1 - 1$, all correct servers (i.e., $n-f$) store locally a valid value.

Let us now consider the time $T_1 = t_0 + \Delta$ when mobile Byzantine agent moves and corrupt a new set of servers.

Let us note that, at time $T_1 = t_0 + \Delta$ we have, in the worse case, a set of f faulty servers, a set of f *cured* servers and a set of $n-2f$ correct servers. At time T_1 servers execute the code shown in Figure 3. In particular, each server s_j broadcasts a ECHO() message with attached the content of its local variables (lines 03-04) and then remains waiting for δ time units. Let us note that, if no write() operation is running when the ECHO() message is sent, each correct server will broadcast the same set of values (cfr. Lemma 3). Contrarily, it may happen that some servers broadcast the concurrent written value and the others echo the previous value. Let us consider separately the two cases where a write(v) happens or not.

- **Case 1 - There not exists a WRITE() message concurrent with the ECHO() message.** In this case $VVS(T_1) = \{\perp\}$, every servers will receive at least the following values:
  - f occurrences of $\langle -, \langle \perp, -1 \rangle \rangle$ and f occurrences of $\langle -, \langle \perp, 0 \rangle \rangle$ coming from cured processes (sent in line 04, Figure 3);
  - from 0 to f occurrences of $\langle -, \langle v_j, sn_j \rangle \rangle$ and f occurrences of $\langle -, \langle v_k, sn_j - 1 \rangle \rangle$ coming from faulty servers;
  - $n-2f$ occurrences of $\langle -, \langle v, sn \rangle \rangle$ and $n-2f$ occurrences of $\langle -, \langle \perp, sn' \rangle \rangle$ coming from correct processes (sent in line 04, Figure 3).
    Note that, since by assumption, no WRITE() message is concurrent with ECHO() messages, then the following can heppen: $v = \perp$, $sn = 0$ and $sn' = -1$ if no write() terminated before time T_1, or, due to Lemma 3 v is the value written by a terminated write, $sn = 1$ and $sn' = 0$ in case such operation exists.

As a consequence, evaluating the condition in line 07, Figure 3 each *cured* server s_i will select the pairs $\langle -, \langle v, sn\rangle\rangle$ and $\langle -, \langle \perp, sn'\rangle\rangle$ (i.e., pairs with at least one valid value), will update accordingly its local variables (line 09, Figure 3) and finally it will become correct. Considering that ECHO() messages are sent at time T_i and take at most δ time unit to be delivered to their destination, we have that at time $T_1 + \delta$ it follows that $n_{CAM} - f$ ($n_{CUM} - f$ respectively) servers are correct and they store locally a valid value.

Note that, the set of faulty and correct servers does not change before time T_2 and that $n_{CAM} \geq (k+3)f + 1$ and $n_{CUM} \geq 2(k+1)f + 1$, it follows that there always exist at least $(k+1)f+1$ correct servers in the CAM model and $(k+2)f+1$ correct servers in the CUM model storing locally a valid value v at time $T_1 + \delta$. Considering that $T_2 = T_1 + \Delta$ and that $i\Delta > 2\delta$ it follows that $T_1 + \delta < T_2$. Thus, due to Lemma 3 and considering that each servers stores, in addition to the current value, also the previous one, the set of correct servers will continue to keep a valid value until the mobile Byzantine agent moves again i.e., until time $T_2 - 1$ and the claim follows.

- **Case 2 - There exists a WRITE$(v, 1)$ message concurrent with the ECHO() message.** In this case, the set of valid value at time T_1 is $VVS(T_1) = \{\perp, v\}$. As far as faulty and cured servers, we still have the following situation where each servers will receive:

  - f occurrences of $\langle -, \langle \perp, -1\rangle\rangle$ and f occurrences of $\langle -, \langle \perp, 0\rangle\rangle$ coming from cured processes (sent in line 04, Figure 3);
  - from 0 to f occurrences of $\langle -, \langle v_j, sn_j\rangle\rangle$ and f occurrences of $\langle -, \langle v_k, sn_j - 1\rangle\rangle$ coming from faulty servers;

  Concerning ECHO() messages coming from correct servers, their content will depend on the time at which they will deliver the concurrent WRITE() message. Let $x < n - 2f$ be the number of correct servers that will deliver the WRITE$(v, 1)$ message before sending the ECHO() at time T_1. We have the following:

  - x occurrences of $\langle -, \langle v, 1\rangle\rangle$ and x occurrences of $\langle -, \langle \perp, 0\rangle\rangle$;
  - from $n - 2f - x$ occurrences of $\langle -, \langle \perp, 0\rangle\rangle$ and $n - 2f - x$ occurrences of $\langle -, \langle \perp, -1\rangle\rangle$

  Evaluating the condition in line 07, each *cured* server will find it false and will execute line 10 by selecting $\langle -, \langle \perp, 0\rangle\rangle$ as old value.

  Note that, the set of faulty and correct servers does not change before time T_2. Considering that $T_2 = T_1 + \Delta$ and that $i\Delta > 2\delta$ it follows that $T_1 + \delta < T_2$. Thus, due to Lemma 3 and considering that each servers stores, in addition to the current value, also the previous one, the set of correct processes will continue to keep a valid value until the mobile Byzantine agent moves again i.e., until time $T_2 - 1$ and the claim follows.

  It follows that the number of correct servers storing a valid value is given by all servers that was correct also before time T_1 that are in the worse case $n_{CAM} - 2f$ ($n_{CUM} - 2f$ respectively). However, considering that $n_{CAM} \geq (k+3)f + 1$ and $n_{CUM} \geq 2(k+1)f + 1$, it follows that there always exist at least $(k+1)f+1$ correct servers in the CAM model and $(k+2)f+1$ correct servers in the CUM model storing locally a valid value v at time $T_1 + \delta$. Considering that $T_2 = T_1 + \Delta$ and that $i\Delta > 2\delta$ it follows that $T_1 + \delta < T_2$. Thus, due to Lemma 3 and considering that each servers stores, in addition to the current value, also the previous one, the set of correct servers will continue to keep a valid value until the mobile Byzantine agent moves again i.e., until time $T_2 - 1$ and the claim follows.

Note that, since the $\text{WRITE}(v, 1)$ message is concurrent with the $\text{ECHO}()$ message, it follows that is has been sent at latest at time $T_1 - 1$. Thus the $\mathsf{write}(1)$ ends latest at time $T_i - 1 + \delta$ and the value $\perp$ is no more valid. However, when delivering such a $\text{WRITE}(v, 1)$ message, each server s_j also broadcasts a $\text{FW\_WRITE}(j, \langle v, 1\rangle)$ message. In particular, such message will be sent by $n - 2f - x$ processes that did not took in to account the new value in the ECHO. It follows that by time $T_1 - 1 + 2\delta$ each servers will stores $n_{CAM} - 2f$ ($n_{CUM} - 2f$ respectively) occurrences of the pair $\langle v, 1\rangle$ and execute lines 07 - 19.

$\square_{Lemma\ 4}$

Lemma 5 *Let Δ be the time interval between two following movements of mobile Byzantine and let δ be the upper bound on the message transfer delay. Let t_0 be the starting time of the computation and let $T_i = t_0 + i\Delta$ be the time of the i-th movement of the mobile Byzantine agent (with $i \in \mathbb{N}$). If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) (ii) $n_{CAM} \geq (k + 3)f + 1$ and (iii) $n_{CUM} \geq 2(k + 1)f + 1$, 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 in the CAM model and $(k+2)f+1$ correct servers in the CUM model storing locally a valid value v (i.e., $v \in VVS(T_i - 1)$).*

Proof The claim follows by induction from Lemma 4 considering that at time T_i the number of correct servers storing a valid value is always at least $n_{CAM} - 2f$ ($n_{CUM} - 2f$ respectively). $\square_{Lemma\ 5}$

Form this Lemma, the following Corollary directly follows.

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

Theorem 2 (Validity) *If (i) $k\Delta \geq 2\delta$ (with $k \in \{1, 2\}$) (ii) $n_{CAM} \geq (k + 3)f + 1$ and (iii) $n_{CUM} \geq 2(k + 1)f + 1$, then any* $\mathsf{read}()$ *operation returns the last value written before its invocation, or a value written by a* $\mathsf{write}()$ *operation concurrent with it.*

Proof Let t be the time at which the reader client c_i invokes a $\mathsf{read}()$ operation and let $T_i < t$ the last time when the mobile Byzantine agent moved before the operation invocation. Let v be a valid value at time $T_i - 1$.

Let us suppose by contradiction that the $\mathsf{read}()$ operation invoked by c_i does not return a valid value. The value v_i returned by c_i is selected among those stored in the $reply_i$ set and it is the value occurring at least $\#reply_{CAM}$ ($\#reply_{CUM}$ respectively) times with the highest sequence number. Note that the $reply_i$ set is emptied at the beginning of the $\mathsf{read}()$ operation and it is filled in with the pairs received by c_i during the period $[t, t + 2\delta]$. Such pairs correspond, for each correct server s_j, with its current local copy of the register value and old register value. Let us note that, due to Corollary 1, there always exist enough correct servers answering to a $\text{READ}()$ message. Thus, if c_i returns a value v_i that is not valid, it means that v_i is an old value or a value never written. Let us consider separately the two cases.

- **Case 1 - v_i is an old value.** Due to Lemma 3, at the end of each $\mathsf{write}()$ operation, all the correct servers stores the new value. Considering that (i) $n_{CAM} \geq (k + 3)f + 1$ ($n_{CUM} \geq 2(k + 1)f + 1$

respectively), (ii) there is a single writer in the system that generates sequence numbers following a total order and (iii) messages are not lost, it follows that there always exists at least $n_{CAM} - 2f \geq \#reply_{CAM}$ ($n_{CUM} - 2f \geq \#reply_{CUM}$) correct servers answering to the read and we have a contradiction.

- **Case 2 - v_i is a value never written.** If v_i is a value never written, it means that is has been generated by faulty processes. However, considering that a $\mathsf{read}()$ operation lasts exactly 2δ time we have that the set of processes answering with a wrong value is at most $2f$ (i.e., cured with no new value and faulty ones). Due to Lemma 3 and Lemma 5 we have that written values are propagated along time to at least $n_{CAM} - 2f$ ($n_{CUM} - 2f$ respectively) correct processes that will answer to the read and the claim follows.

$\square_{Theorem\ 2}$

Theorem 3 *If (i) $k\Delta \geq 2\delta$ (with $k \in 1, 2$) (ii) $n_{CAM} \geq (k+3)f + 1$ and (iii) $n_{CUM} \geq 2(k+1)f + 1$, then the algorithm $\mathcal{A}_{Rreg}$ implements a SWMR Regular Register resilient to the presence of up to f Mobile Byzantine failures.*

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

6 An Efficient Algorithm for low-frequency movement in the CAM Model

The algorithm $\mathcal{A}^*_{Rreg}$ proposed in this section is a modified version of $\mathcal{A}_{Rreg}$ that exploits the knowledge about the cured state available in the CAM model to reduce the number of replicas n_{CAM}. The basic idea is that, when movements are not so frequent, cured servers have enough time to run the state_maintenance protocol, getting correct and help to complete $\mathsf{read}()$ operations. This implies that cured servers can now be part of the quorums answering to $\mathsf{read}()$ operations. As a consequence, the global number of servers n_{CAM} needed to tolerate f Mobile Byzantine agents is decreased. The price to pay is having $\mathsf{read}()$ operations lasting more than in the previous case.

In the following we report the pseudo-code of the algorithm and its description. The main difference here is that we need to store only the last value forgetting the previous one. In addition, servers are able to filter out some spurious messages like WRITE_FW or ECHO coming from faulty servers that are now cured.

This protocol works when $\Delta > 4\delta$.

The state maintenance **protocol.** Each server executes every Δ time units the protocol in Figure 6. The protocol starts by cleaning local variables and broadcasting a ECHO message: non-cured servers propagate their current value (together with its sequence number) and their set of currently reading clients, while cured servers propagate some default values. Cured servers then keep waiting for 2δ time units in order to collect information propagated by other servers and check if they are able to clean their state with a valid value. They first filter possible duplicate ECHO messages and write forwarded messages sent by servers that declared themselves as cured (lines 08-12). This is done to avoid to consider bad messages sent just before the Byzantine agent leaves a servers. Once the filtering has been done, cured servers may select the value used to update their state. This is done by considering all the values occurring at least $n - 2f$ times: for each of them a reply is sent to help the $\mathsf{read}()$ termination and the value is chosen if the corresponding timestamp is greater than the current one. At this point, all cured servers become correct.

```
every $\Delta$ time units do:
(01) $cured_i \leftarrow$ report_cured_state();
(02) $echo\_vals_i \leftarrow \emptyset$; $echo\_read_i \leftarrow \emptyset$; $fw\_vals_i \leftarrow \emptyset$;
(03) if $(\neg cured_i)$
(04)    then broadcast ECHO$(i, \langle val_i, sn_i \rangle, pending\_read_i)$;
(05)    else broadcast ECHO$(i, \langle \bot, -1 \rangle, \emptyset)$;
(06) endif
(07) wait$(2\delta)$;
(08) for each $j$ s.t.
(09)     $(\exists \langle j, \langle \bot, -1 \rangle \rangle, \in echo\_vals_i)$
(10)     $\forall \langle j, \langle v, ts \rangle \rangle \in fw\_vals_i$ : remove from $fw\_vals_i$;
(11)     $\forall \langle j, \langle v, ts \rangle \rangle \in echo\_vals_i$ : remove from $echo\_vals_i$;
(12) endFor
(13) for each $(\langle v, ts \rangle \in echo\_vals_i \cup fw\_vals_i$
     occurring $\geq n - 2f$ times $\wedge$ $ts > -1)$
(14)     for each $(j \in (pending\_read_i \cup echo\_read_i))$ do
(15)         send REPLY $(i, \langle val_i, sn_i \rangle)$ to $c_j$;
(16)     endFor
(17)     if $(ts > sn_i)$ then $val\_i \leftarrow v$; $sn_i \leftarrow ts$; $cured_i \leftarrow$ false;
(18) endFor
---------------------------------------------------------------
when ECHO $(j, \langle v, ts \rangle, pr)$ is received:
(19)  $echo\_vals_i \leftarrow echo\_vals_i \cup \{\langle j, \langle v, ts \rangle \rangle\}$;
(20)  $echo\_read_i \leftarrow echo\_read_i \cup pr$;
```

Figure 6: $\mathcal{A}^*_{Rreg}$ state_maintenance protocol (code for server s_i).

```
when WRITE$(v, csn)$ is received:
(01)  $val_i \leftarrow v$;
(02)  $sn_i \leftarrow csn$;
(03)  if $(cured_i)$
(04)     then $cured_i \leftarrow$ false;
(05)  endif
(06)  for each $j \in (pending\_read_i \cup echo\_read_i)$ do
(07)      send REPLY $(i, \langle val_i, sn_i \rangle)$;
(08)  endFor
(09)  broadcast WRITE_FW$(i, \langle v, csn \rangle)$;
---------------------------------------------------------------
when WRITE_FW$(j, \langle v, csn \rangle)$ is received:
(10)  $fw\_vals_i \leftarrow fw\_vals_i \cup \{\langle j, \langle v, csn \rangle \rangle\}$;
```

Figure 7: $\mathcal{A}^*_{Rreg}$ write() operation protocol.

The write() operation. The client code is the same as in the previous case.

When a server s_j delivers a WRITE(v, csn) message, it updates its local copy and then it checks if it was in the cured state: if so, it resets its state to correct itself, and starts answering to pending read() operations. Each time a WRITE(v, csn) message is delivered, it is also forwarded to all other servers to avoid that some server s_j delivers the WRITE(v, csn) message just before the mobile Byzantine agent leaves it and later ignores the message when it becomes cured.

The read() operation. When a client c_i wants to read, it broadcasts a READ message, and then waits for 4δ time. Then, c_i selects from its $reply_i$ set the pairs that occurs at least $n - f$ times, broadcasts a READ_ACK (i) message to inform servers that its operation has terminated, and finally returns the selected value.

When a server s_j delivers a READ message, it first adds the client to the set of concurrent readers and checks its current state: if it is not cured, it answers immediately by sending back its local state, otherwise

```
operation read():
(01) $reply_i \leftarrow \emptyset$;
(02) broadcast READ$(i)$;
(03) wait $(4\delta)$;
(04) $\langle val, sn\rangle \leftarrow$ select$(reply_i)$;
(05) broadcast READ_ACK$(i)$;
(06) return $val$;
--------------------------------------------
when REPLY $(j, \langle val, sn\rangle)$ is received:
(07)   $reply_i \leftarrow reply_i \cup \{\langle j, \langle val, sn\rangle\rangle\}$;
```

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

```
when READ $(j)$ is received:
(01)  $pending\_read_i \leftarrow pending\_read_i \cup \{j\}$;
(02)  if $(\neg cured_i)$
(03)     then send REPLY $(i, \langle val_i, sn_i\rangle)$;
(04)  endif
(05)  broadcast READ_FW$(j)$;
--------------------------------------------
when READ_FW $(j)$ is received:
(06)  $pending\_read_i \leftarrow pending\_read_i \cup \{j\}$;
--------------------------------------------
when READ_ACK $(j)$ is received:
(07)  $pending\_read_i \leftarrow pending\_read_i \setminus \{j\}$;
(08)  $echo\_read_i \leftarrow echo\_read_i \setminus \{j\}$;
```

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

Figure 8: $\mathcal{A}^*_{Rreg}$ read() operation protocol.

it postpones the answer for the moment where it has correct its state.Delivering a READ_ACK message, s_j simply removes the identifier of the client from the list of concurrent readers.

6.1 Correctness Proofs

We first establish the termination property of our algorithm, that is independent of the model considered (CUM and CAM).

Lemma 6 *If a correct client c_i invokes* write(v) *at time t, this operation terminates at time $t + \delta$.*

Proof The claim simply follows by considering that write() returns a write_confirmation to the calling client c_i after δ time, independentely of the behavior of the servers (see Lines 03-04, Figure 4(a)). $\Box_{Lemma\ 6}$

Lemma 7 *If a correct client c_i invokes* read() *at itme t, this operation terminates at time $t + 4\delta$.*

Proof The claim simply follows by considering that a read() returns a value to the client after 4δ time, independentely of the behavior of the servers (see Line 06, Figure 8(a)). $\Box_{Lemma\ 7}$

Theorem 4 (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 6 and Lemma 7. $\Box_{Theorem\ 4}$

In the following section, we assume that when a mobile Byzantine leaves a server s_i, then s_i is aware of being in the *cured* state if s_i makes use of the report_cured_state oracle. In the remaining of the section, Δ denotes the time between two consecutive moves of the mobile Byzantine agents, and δ denotes the upper bound on any message transfer delay. Also, t_0 denotes the time of the beginning of the execution.

Lemma 8 *Let op be a* write(v) *operation invoked by a correct client at time t. Then any server s_j that remains correct in the interval $[t, t + \delta]$ has $val_j = v$.*

Proof The claim simply follows from the fact that: *(i)* at time t, the writer sends a $\text{WRITE}(v, sn)$ message to all servers, *(ii)* messages are not lost, and *(iii)* by time $t + \delta$ any correct server s_j executes Line 01, Figure 4(b). $\Box_{Lemma\ 8}$

Lemma 9 *Let op be a* $\mathsf{write}(v)$ *operation executed by a client* c_i *during* $[t, t + \delta]$*. If there exists* $i \in \mathbb{N}$ *such that* $T_i \in [t, t + \delta]$*, then by time* $t + 2\delta$*, there exist at least* $n - f$ *correct servers* s_j *such that* $val_j = v$*.*

Proof Due to Lemma 6, the operation op is executed in the period $[t, t + \delta]$. During the execution of the $\mathsf{write}()$ operation, the set of faulty servers changes. Let us call $F^-_{T_i}$ the set of faulty servers before time T_i and $F^+_{T_i}$ the set of faulty servers from T_i on. At the beginning of the $\mathsf{write}()$ operation, the writer broadcasts a $\text{WRITE}(v)$ message (line 02, Figure 4(a)) that will be delivered by time $t + \delta$.

In the worse case, (i) $F^-_{T_i} \cap F^+_{T_i} = \emptyset$, (ii) each server $s_j \in F^-_{T_i}$ delivered the $\text{WRITE}(v)$ message before time T_i when it is faulty and (iii) each server $s_k \in F^+_{T_i}$ delivered the $\text{WRITE}(v)$ message after time T_i when it is faulty.

Thus, considering that faulty servers during the execution of the operation are at most $2f$, we have that at least $n - 2f$ servers, i.e., at least $f + 1$ servers, remain correct during the whole operation execution (i.e., in the time interval $[t, t + \delta]$). Such set will execute lines 01-09 in Figure 7 updating its val_j with the value v and then they will broadcast a $\text{FW\_WRITE}()$ message. Note that, such message is broadcast at latest at time $t + \delta - 1$. Considering that $\text{FW\_WRITE}()$ message takes at most δ time to be delivered to its destination and considering that at least $f + 1$ correct forward the same pair $\langle v, sn\rangle$, it follows that any server $s_j \in s_k \in F^-_{T_i}$ will execute lines 13- 17 in Figure 6 storing v by time $t + 2\delta$ and the claim follows.

$\Box_{Lemma\ 9}$

Corollary 2 *Let op be a* $\mathsf{read}()$ *operation executed by a client* c_i *during* $[t, t + 4\delta]$*. If there exists* $i \in \mathbb{N}$ *such that* $T_i \in [t, t+\delta]$*, then by time* $t+2\delta$*, there exist at least* $n-f$ *correct servers* s_j *having* $c_i \in pending\_read_j$*.*

Proof [Sketch] The proof follows exactly the same reasoning used in Lemmas 8 and 9, where instead of considering the WRITE and WRITE_FW messages, we consider the READ and READ_FW messages. $\Box_{Corollary\ 2}$

Lemma 10 *If* $\Delta \geq 4\delta$ *and* $n \geq 3f + 1$*, then at time* $T_1 + 2\delta$ *there exists at least* $n - f$ *correct servers* s_j *with* $val_j \in VVS(T_1 + 2\delta)$*.*

Proof Let t_0 be the time at which the computation starts and let $\perp$ be the initial value of the register. Let us note that, in the period $[t_0, T_1 - 1]$ (with $T_1 - 1 = t_0 + \Delta - 1$) there are no changes in the set of faulty servers.

Let us now consider the time $T_1 = t_0 + \Delta$ when the mobile Byzantine agent moves and corrupts a new set of servers.

Let us note that, at time $T_1 = t_0 + \Delta$ we have, in the worse case, a set of f faulty servers, a set of f *cured* servers and a set of $n - 2f$ (i.e., at least $f + 1$) correct servers.

At time T_1 servers execute the code shown in Figure 6. In particular, each server s_j broadcasts a $\text{ECHO}()$ message with attached the content of its local variables (lines 03-05) and then remains waiting for δ time units. Let us note that, if no $\mathsf{write}()$ operation is running when the $\text{ECHO}()$ message is sent, each correct server will broadcast the same set of values. Contrarily, it may happen that some servers broadcast the concurrent written value and the others echo the previous value. Let us consider separately the two cases where a $\mathsf{write}(v)$ happens or not.

- **Case 1 - $|VVS(T_1, T_1 + \delta)| = 1$ (i.e., there not exists a $\textsc{write}()$ message concurrent with the $\textsc{echo}()$ message).** In this case, every servers will receive at least the following values:
  - f occurrences of $\langle -, \langle \bot, -1 \rangle \rangle$ coming from cured processes (sent in line 04, Figure 6);
  - from 0 to f occurrences of $\langle -, \langle v_j, sn_j \rangle \rangle$ coming from faulty servers;
  - $n - 2f$ occurrences of $\langle -, \langle v, sn \rangle \rangle$ coming from correct processes (sent in line 05, Figure 6). Note that, since by assumption, no $\textsc{write}()$ message is concurrent with $\textsc{echo}()$ messages, then the following can heppen: $v = \bot$, $sn = 0$ if no $\mathsf{write}()$ terminated before time T_1, or v is the value written by a terminated write and $sn = 1$ and in case such operation exists.

  Considering that $n \geq 3f + 1$, evaluating the condition in line 13, Figure 6 each *cured* server s_i will select the pair $\langle -, \langle v, sn \rangle \rangle$ (i.e., a pair corresponding to a valid value), will update accordingly its local variables (line 17, Figure 6) becoming correct.

  Considering that $\textsc{echo}()$ messages are sent at time T_i and take at most δ time unit to be delivered to their destination, we have that at time $T_1 + \delta$, $n - f$ servers (i.e., at least $2f + 1$) are correct and they store locally a valid value. Note that, due to Lemma 8 the set of correct processes will continue to keep a valid value until the mobile Byzantine agent moves again i.e., until time $T_2 - 1$ and the claim follows.

- **Case 2 - $|VVS(T_1, T_1 + \delta)| > 1$ (i.e., there exists a $\textsc{write}(v, 1)$ message concurrent with the $\textsc{echo}()$ message).** In this case, the set of valid value at time T_1 is $VVS(T_1) = \{\bot, v\}$. As far as faulty and cured servers, we still have the following situation where each servers will receive:
  - f occurrences of $\langle -, \langle \bot, -1 \rangle \rangle$ coming from cured processes (sent in line 05, Figure 6);
  - from 0 to f occurrences of $\langle -, \langle v_j, sn_j \rangle \rangle$ coming from faulty servers;

  Concerning $\textsc{echo}()$ messages coming from correct servers, their content will depend on the time at which they will deliver the concurrent $\textsc{write}()$ message. Let $x < n - 2f$ be the number of correct servers that will deliver the $\textsc{write}(v, 1)$ message before sending the $\textsc{echo}()$ at time T_1. We have the following:

  - x occurrences of $\langle -, \langle v, 1 \rangle \rangle$;
  - from $n - 2f - x$ occurrences of $\langle -, \langle \bot, 0 \rangle \rangle$.

  Each correct server forwards the value written by the $\mathsf{write}(v)$ operation trough the $\textsc{write\_fw}()$ message sent in line 09, Figure 7. This will happen at latest at time $T_1 + \delta - 1$. Considering that messages are not lost and they take at most δ time units to be delivered to their destinations, we have that by time $T_1 + 2\delta - 1$ each cured servers will deliver also $n - 2f - x$ $\textsc{write\_fw}()$ messages with the pair $\langle v, 1 \rangle$. As a consequence, at latest at time $T_1 + 2\delta - 1$, each of the f cured server will execute lines 13-17, Figure 6 as they received at least $f + 1$ times the same pair $\langle v, 1, \rangle$ and they become correct storing a valid value (i.e., the value 1). Thus, at time $T_1 + 2\delta - 1$ no server is in the *cured* state and we have $n - f$ correct servers. Let us note that if a second $\mathsf{write}(v_2)$ operation is issued before time $T_1 + 2\delta - 1$ and the corresponding $\textsc{write}()$ message is delivered before the execution of lines 13-17, Figure 6, cured servers s_j become correct. Considering that $\Delta \geq 4\delta$ we have that $T_1 + 2\delta - 1 < T_2$ and thus, due to Lemma 8, the set of correct processes continue to keep a valid value until the mobile Byzantine agent moves again i.e., until time $T_2 - 1$ and the claim follows.

$\square_{Lemma\ 10}$

Lemma 11 *If $\Delta \geq 4\delta$ and $n \geq 3f+1$, then at time $T_i + 2\delta$ (with $i \in \mathbb{N}$) there exists at least $n-f$ correct servers s_j with $val_j \in VVS(T_i + 2\delta)$).*

Proof The claim follows by induction from Lemma 10 considering that at time T_i the number of correct servers storing a valid value is always at least $n-2f$. $\square_{Lemma\ 11}$

Corollary 3 *If $\Delta \geq 4\delta$ and $n \geq 3f+1$ then at time $T_i - 1$ (with $i \in \mathbb{N}$) there exists at least $n-f$ correct servers storing locally a valid value v (i.e., $v \in VVS(T_i - 1)$)*

Proof The Corollary follows directly form the definition of valid values and form Lemma 8 and Lemma 5. $\square_{Corollary\ 3}$

Theorem 5 (Validity) *Any* read() *operation returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

Proof

Let $t_B(op)$ the time at which the read() operation has been invoked. When the read() is invoked, the reader client c_i broadcasts a READ() message to all the servers and then it waits until time $t_B(op) + 4\delta$ collecting replies.

Due to Corollary 3, at time $T_i - 1$ there exists at least $n-f$ correct servers storing a value $v \in VVS(T_i - 1)$.

Informally speaking, being $\Delta > 4\delta$, depending on the time at which the read() operation is invoked, it can be concurrent with a Byzantine agents movement at time T_i. In such case the $read()$ operation involves at most $2f$ *faulty* servers, i.e. $MaxW = 2f$, and at most $2f$ *cured* servers, let us call S_i *cured* servers after T_i. In the best case scenario the operation does not cross any T_i involving at most f *faulty* servers, i.e. $MaxW = f$. Let us consider only the worst case scenario.
Since the read() operation lasts 4δ then all the messages sent within $[t_B(op), t_B(op) + 3\delta]$ are delivered by the client for sure. Thus we are interested in to count how many correct reply are delivered for sure. Let us define $[T_k, T_k + 2\delta]$ the time window in which the f servers (*faulty* in $[T_{k-1}, T_k - 1]$) are *cured*.

Let us suppose that during a read() operation there are not $n-f$ *correct* servers correctly replying. There are the following case, depending on the presence or absence of concurrency with a write() operation.
case a) no write(v) operations are concurrent with the read(), we have the following case: A read() operation lasts 4δ, then the *request phase* is $[t_B(R), t_B(R) + 3\delta]$. Since $\Delta > 4\delta$, then in $[t_B(R), t_B(R) + 3\delta]$, at most f *faulty* servers exist. Moreover there may be f more *cured* ones. From Lemma 11, the time to become correct is 2δ. Then, during $[t_B(R) + 2\delta, t_B(R) + 3\delta]$ there are at least $n-f$ *correct* servers.
case b) there is a write(v') operation concurrent with the read(). Let us call v the value previously stored by *correct* servers (or $\perp$ if no previous write occurred). If the client is not able to read a valid value (i.e. to gather $n-f$ occurrences of the same value (v' or v)) then it mean that due to the concurrency with the write(v'), it collects: x occurrences of v and $n-f-x$ occurrences of v'. Let $t_B(W)$ and $t_E(W)$ be the beginning and ending time of write(v'), then there are two sub-cases:

- $T_i \in [t_B(W), t_E(W)]$: in that case by Lemma 11 after 2δ time $n - f$ servers are storing v'. If some x *correct* servers do not reply with v' then $t_B(W) > t_B(R) + 2\delta$. Since T_i happens after $t_B(R) + 2\delta$ then in $[t_B(R), t_B(R) + 2\delta]$ there are at least $n - f$ *correct* servers replying with v, thus, we get a contradiction.
- $T_i \notin [t_B(W), t_E(W)]$: in that case by Lemma 11 after δ time $n - f$ servers are storing v'. There are two sub sub cases:
  - $t_B(W) \leq t_B(R)$ in that case the $\mathsf{write}(v')$ operation starts before the $\mathsf{read}()$, thus since it lasts δ then all *correct* servers receive v' and reply with it. Thus we have a contradiction.
  - $t_B(R) + \delta \leq t_B(W) \leq t_E(R) - 2\delta$: if $T_i \in [t_B(R), t_B(R) + \delta)]$ then for Lemma 2 at most at $t_B(R) + 2\delta$ all correct servers reply to client c_i. It delivers enough occurrences of v or v' depending on the arrival order between $echo$ messages carrying v and c_i to respect the $write(v')$ message which determines the message replies. Thus we have a contradiction.
  - $t_B(R) + 2\delta \leq t_B(W) \leq t_E(R) - \delta$: same reasoning as before.
  - $t_B(W) > t_E(R) - \delta$, since during the $\mathsf{write}(v')$ operation there is no T_i by hypothesis and being it at the end of the $\mathsf{read}()$ operation then it is not concurrent with the $[t_E(R), t_E(R) + 3\delta]$ time interval. This is case a) proving that there are $n - f$ *correct* servers correctly replying. Thus we have a contradiction.

$\square_{Theorem\ 5}$

Theorem 6 *If (i)* $\Delta > 4\delta$ *and* $n_{CAM} \geq 3f + 1$, *then the algorithm* $\mathcal{A}^*_{Rreg}$ *implements a SWMR Regular Register resilient to the presence of up to* f *Mobile Byzantine failures.*

Proof The proof simply follows considering Theorem 4 and Theorem 5. $\square_{Theorem\ 6}$

7 Conclusion

This paper presented the first round-free emulation of a distributed storage with regularity property that tolerates Mobile Byzantine Failures. We proved that, differently from the synchronous round-based case, in round-free computations the number of replicas n needed to tolerate f Mobile Byzantine Failures depends not only on f but also on the ratio between the frequency of the movement Δ and the upper bound on the message latency δ (see Figure 1). Interestingly, such relation holds only for "frequent" movements. In the case of low-frequency movements we obtain the same bounds as for the synchronous round-based case. We conjecture that our bounds are tight for the synchronous round-free model since the proposed algorithm targets the better trade-off between time needed to cure servers and operations length. The formal proof of the tightness of our bounds is not trivial and is one of the open directions of this work.

References

[1] `http://www.infoworld.com/article/2666755/security/akamai-outage-hobbles-google--microsoft--others.html`. Accessed September, 2015.

[2] http://www.informationweek.com/cloud/cloud-storage/microsoft-azure-outage-blamed-on-bad-code/d/d-id/1318331. Accessed September, 2015.

[3] Cloud computing vulnerability incidents: A statistical overview. http://www.cert.uy/wps/wcm/connect/975494804fdf89eaabbdab1805790cc9/Cloud_Computing_Vulnerability_Incidents.pdf?MOD=AJPERES. Accessed September, 2015.

[4] 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.

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

[6] 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.

[7] Silvia Bonomi, Antonella del Pozzo, and Maria Potop-Butucaru. Tight self-stabilizing mobile byzantine-tolerant atomic register. In *(to appear) 17th International Conference on Distributed computing and Networking (ICDCN 2016)*, 2016. Preliminary Technical Report Version available at http://arxiv.org/pdf/1505.06865v1.pdf.

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

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

[10] 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.

[11] Dorothy E Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (2):222–232, 1987.

[12] Ben Treynor (VP Engineering). http://googleblog.blogspot.fr/2014/01/todays-outage-for-several-google.html. Accessed September, 2015.

[13] 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.

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

[15] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[17] 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.

[18] 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, 2002.

[19] 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.

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

[21] 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.

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