Building distributed computing abstractions in the presence of mobile byzantine failures

Antonella del Pozzo

SAPIENZA
UNIVERSITÀ DI ROMA

UPMC
SORBONNE UNIVERSITÉS

Building Distributed Computing Abstractions in the Presence of Mobile Byzantine Failures

Universitá di Roma "Sapienza"

Dottorato di ricerca in Ingegneria Informatica – XXIX Ciclo

Université Pierre et Marie Curie

Thèse de doctorat en Informatique, Télécomminication et Électronique

Candidate

Antonella Del Pozzo
ID number 1015134

Thesis Advisors

Prof. Silvia Bonomi
Prof. Maria Potop-Butucaru

Reviewers

Prof. Antonio Fernandez Anta
Prof. Xavier Defago
Prof. Achour Mostefaoui

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Informatique, Télécomminication
et Électronique

December 2016

Building Distributed Computing Abstractions in the Presence of Mobile Byzantine Failures

Ph.D. thesis. Sapienza – University of Rome
ISBN: 000000000-0

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: February 2, 2017

Author's email: delpozzo@dis.uniroma1.it

A Mamma

Contents

Chapter 1

Introduction

Nowadays high availability plays a key role for many distributed systems. Just to give an intuition, less than twenty years ago IBM Global Services quantified that due to unavailable systems in 1996 the American businesses lost $4.54 billion [41]. In 2015, this figure has dramatically increased, indeed, it has been shown that IT downtime costs $700 billion to North American companies [1]. One of the main challenges is to guarantee distributed systems availability despite accidental and malicious failures. Failures can not be avoided, increasing the importance of designing fault tolerant systems. The main characteristic of those systems is the elimination of single points of failure introducing redundancy, which allows the system to work even though some of its components are faulty. More into details, all different typologies of failures are included under the name of Byzantine failures. A Byzantine component may behave in any possible way, spanning from the crash failures (the component does not work at all) to malicious failures (the component is hijacked by an attacker). Classical Byzantine tolerant solutions assume that over n components there can be up to f that may suffer from Byzantine failures. If such hypothesis is violated, i.e., an attacker controls more than f components, the distributed system may no more be available or correct. It follows that the main limitation with this approach is that systems are built to tolerate a fixed percentage of failures over the whole components number. This fails the reality test of long-lived distributed services. With new exploits being publicized daily and hackers offering services at amazingly low prices, *every* component is bound to be compromised in a long time. On the bright side, dedicated cure and software rejuvenation techniques increase the possibility that a compromised node *does not remain compromised forever*, and may be recover from its previously compromised status [42]. Moreover, as pointed out in [52], in addition to classical Byzantine behaviors, it is worth to consider *mobile adversaries*. Mobile adversaries have been primarily introduced in the context of multi-party computation and they try to model an attacker that is able to progressively compromise computational entities but only for a limited period of time. Therefore, tolerating Mobile Byzantine Failures is, in some sense, like having a bounded number of compromised entities at any given time, but such set changes from time to time. Such model captures phenomena like virus injection (where viruses start to infect the network but then they are detected and progressively deleted from a set of machines), programmed maintenance with the aim of restoring

potentially infected machines or self-repairing systems [42]. All those considerations advocate the study of a more complex failure model to capture the dynamism of compromised component sets.
In this thesis we analyze two main problems concerning the implementation of distributed systems. Distributed Registers and Approximate Agreement. Those, in our vision, were the natural steps after the Mobile Byzantine Tolerant Consensus, the only problem solved so far in the presence of mobile Byzantine failures ([3, 5, 11, 21, 40]). Distributed Registers are the building block of a distributed data store, a fundamental component for many of the principal web services, such as Facebook (Apache Cassandra), Google (BigTable), Netflix (Druid), Amazon (Dynamo), just to cite a few. Depending on the provided consistency degree there are three register specifications, from the weakest to the strongest: safe, regular and atomic. To ensure high availability, storage services are usually implemented by replicating data at multiple locations and maintaining such data consistent. Thus, replicated servers represent today an attractive target for attackers that may try to compromise replicas correctness for different purposes. Some examples are: to gain access to protected data, to interfere with the service provisioning (*e.g.*, by delaying operations or by compromising the integrity of the service), to reduce service availability with the final aim to damage the service provider (reducing its reputation or letting it pay for the violation of service level agreements), etc. In this context, thanks to Byzantine Fault Tolerance (BFT) techniques, a compromised replica (a Byzantine failure) is made transparent to clients. In the context of distributed storage implementations (*e.g.*, register abstraction), common approaches to BFT are based on the deployment of a sufficient large number of replicas to tolerate an estimated number f of compromised servers (i.e., BFT replication). However, to the best of our knowledge, no storage abstraction has been investigated so far assuming mobile adversaries.
Along with the Distributed Register abstraction, the Approximate Agreement problem plays a key role in many distributed system classes. The emergent area of sensor networks or mobile robot networks revived recently the research on one of the most studied building blocks of distributed computing ([9, 10, 13, 27, 46, 47, 48, 49, 50]). Indeed, gathering environmental data such as temperature or atmospheric pressure, or synchronizing clocks in large scale sensor networks, typically do not require perfect agreement between participating nodes. Also, requiring autonomous mobile robots to gather at some specific location *e.g.*, to communicate or to setup a new task, tolerates a difference in the final robot positions after gathering. This is due to the robots physical size. Accepting a predetermined difference in the agreement process permits to avoid many impossibility results occurring in the perfect agreement case. The above mentioned contexts are not free from failures and in particular from Byzantine ones. In sensor networks, in fact, sensors may not transmit their values or may transmit erroneous values due to permanent or temporary failures. In mobile autonomous robot networks, some robots may move in the opposite direction as the one intended due to hardware malfunction of buggy software. In both cases the signals (transmitted data, or perceived position) sent by the faulty participants may have a tremendous impact on the approximated value that is computed by the correct ones. To handle such behaviors, the solvability of Approximate Agreement has been studied in presence of Byzantine processes [15, 22, 28]. The problem becomes even

more difficult to solve when failures may impact different participants over time. For example, in sensor or mobile robot networks, the possibility of intermittent external perturbations (*e.g.*, magnetic fields) may affect different processes of the network at various moments during system execution. Participants that are located in such affected areas may exhibit Byzantine behavior. While the Approximate Agreement problem has been deeply studied in systems prone to Byzantine faults [15, 22, 28] revealing its complexity, none has been done (as far as we know) considering Mobile Byzantine Failures. This left open some important questions about the solvability of the problem and its complexity.

1.1 Contributions and Road map

The work in this thesis can be quickly listed in the following contributions:

- we define a general round-free Mobile Byzantine Failure (MBF) model, which can be decomposed in a hierarchy of four different models (published in [8]);
- starting from the already defined round-based Mobile Byzantine Failure model, we solve the Atomic Register problem (published in [6]);
- we define a framework to prove lower bounds to solve Safe Register problem in each of the round-free synchronous MFB models;
- we propose optimal solutions to solve Regular Registers problem in each of those models (partially published in [8]);
- we propose an optimal solution for the Approximate Agreement problem in the round-based MBF models (published in [7]).

Roadmap. In Chapter 2 and Chapter 3 we respectively discuss related works and define the system model. The main contribution of this thesis is Chapter 4, where we propose and formalize a general MBF model. Along with the already defined round based MBF model, we propose a hierarchy of round-free MBF models, generated combining components awareness about their failure state and mobile agents movements freedom. Hereafter we explore the instances of the model where this problem is solvable,*e.g.*, we provide impossibility results for the asynchronous setting (Section 6.2). In Chapter 5 is presented the study of Distributed Register in the round-based MBF models. In particular we prove lower bounds on the number of replicas, and propose an optimal algorithm to solve the strongest consistency register problem in the round-based system model. In Chapter 6 we explore the Regular Register abstraction in the round-free MBF models. We prove lower bounds (Section 6.3) and present and prove the correctness of protocols (Sections 6.4 - 6.7) whose resilience is optimal with respect to the number of Byzantine agents that can be tolerated.

Finally we move to the Approximate Agreement problem in Chapter 7. In Section 7.2 we prove lower bounds for Approximate Agreement in the Mobile Byzantine failures model. Interestingly the lower bounds do not change with respect the

Agreement in the Mobile Byzantine failures model. The same happens in the case of Byzantine failures, Agreement and Approximate Agreement have the same lower bounds with respect the number of replicas. Then we map the existing variants of Mobile Byzantine models to the Mixed-Mode faults model [22]. This mapping further helps us to prove the correctness of a particular class of solutions for Approximate Agreement in the Mobile Byzantine failures model, Section 7.3.

Chapter 2

Related Work

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 behaviors. Specifically targeted attacks to compromise processes and/or virus infections can indeed cause malicious code execution. In classical Byzantine fault-tolerance, the power of attacks and infections is typically abstracted as an upper bound f on the number of Byzantine processes that a given set of n processes has to be able to tolerate. Such bounds permit to characterize the solvable cases for benchmarking problems in Distributed Computing (*e.g.*, Agreement and Register Emulation). As we stated, this abstraction fails the reality test of long-lived distributed services. Dedicated cure and software rejuvenation techniques increase the possibility that a compromised node *does not remain compromised forever*, and may be aware of its previously compromised status [42]. Rejuvenation techniques use proactive and reactive replicas recovery. Interestingly, proactive recovery has been shown to be not feasible in asynchronous systems ([43], [44]). In few words, periodically groups of replicas start to recover, but in an asynchronous system a compromised replica can delay its recovery, allowing more than f replicas to be compromised. Interestingly, in [38] is presented a theoretical model to estimate the system resilience over its lifetime based on the rejuvenation rate and the number of replicas. As will be clearer further, the implicit failure model considered in those works match particular cases of the Mobile Byzantine Failures (MBF) models. In the MBF models, faults are represented by Byzantine agents that are managed by a powerful omniscient adversary that "moves" them from a process to another. Let us note that the term "mobile" does not necessary imply that a Byzantine agent physically moves from one process to another, but it rather captures the phenomenon of a progressive infection, that modifies the code executed by a process as well as its internal state, and the subsequent cure and restoration of the correct protocol (due, for example, to the detection of the infection or to a proactive recovery mechanism). In the sequel, we first present Mobile Byzantine Failure models, then we introduce the relevant abstractions that we considered to be solved in presence of Mobile Byzantine failures, Shared Memory and Approximate Agreement.

2.1 Mobile Byzantine Failure Models for Round-based Computations

Mobile Byzantine Failures have been investigated so far in round-based computations, and can be classified according to Byzantine mobility constraints: *(i)* Byzantine agents with constrained mobility [11] may only move from one node to another when protocol messages are sent (similarly to how viruses would propagate), while *(ii)* Byzantine agents with unconstrained mobility [3, 5, 21, 36, 39, 40] may move independently of protocol messages.

Most of the previously cited models [3, 5, 11, 21, 40] consider that processes execute synchronous rounds composed of three phases: (i) *send* where processes send all the messages for the current round, (ii) *receive* where processes receive all the messages sent at the beginning of the current round and (iii) *computation* where processes process received messages and prepare those that will be sent in the next round. Only between two consecutive rounds, Byzantine agents are allowed to move from one node to another. Hence the set of faulty processes at any given time has a bounded size, yet its membership may evolve from one round to the next. The main difference between the aforementioned four works [3, 5, 21, 40] lies in the knowledge that processes have about their previous infection by a Byzantine agent. In Garay's model [21], 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.

More details are presented in Chapter 3, where we present all the system models considered in this thesis.

The Byzantine Agreement problem, introduced first by Lamport *et al.* [26] is one of the most studied building blocks in distributed computing and is specified as the conjunction of the following three properties [28]:

- **(Termination)**: All correct processes eventually decide;
- **(Agreement)**: No two correct processes decide on different values;
- **(Validity)**: If all correct processes start with the same value v, then v is the only possible decision value for a correct process.

In the Mobile Byzantine version of the problem has been the only problem solved so far presence of mobile Byzantine failures.

Garay [21] 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.* [3]. Sasaki *et al.* [40] 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.* [5] 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 IDs); 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.

2.2 Approximate Byzantine Agreement

In this thesis we explore the Approximate Byzantine Agreement problem, in which processes start with real numbers as inputs, and eventually decide a real number as output. The difference with the (exact) Byzantine Agreement is that instead of agreeing exactly, processes are allowed to disagree within a small positive margin ϵ on the decided values. The specification of the Approximate Byzantine Agreement [28] has the same termination property as the Byzantine Agreement. However, it has different agreement and validity properties:

- **(Termination)**: All correct processes eventually decide;
- **(ϵ-Agreement)**: for some $\epsilon > 0$, the decision values of any pair of correct processes are within ϵ of each other;
- **(Validity)**: any decision value for a correct process is in the range of the initial values of the correct processes.

The Approximate Byzantine Agreement problem has been studied since the eighties [15], [19]. Most of the presented solutions are based on successive rounds of exchanges of the latest value each process locally stores. Upon collecting each set of values, a correct process applies a function (*e.g.*, average) and adopts as next value the value returned by the function. The interested reader may refer to reference textbooks [28] and references herein [17, 18].

Stolz *et al.* [45] recently proposed an Approximate Byzantine Agreement solution where processes have to approximate the median value of the input values. Their algorithm achieves agreement for $n > 3f$ within $f + 1$ rounds, where f denotes the number of faulty (Byzantine) processes, while n denotes the total number of processes. Their algorithm is not included in the class of MSR-algorithms of [22] since they use a variant of the King algorithm [4]. Multidimensional agreement has been investigated by Mendes *et al.* [34, 35], where the authors also highlight the connexion between approximate agreement and convergence in mobile autonomous robot networks [9, 10]. Li *et al.* [27] and Charron-Bost *et al.* [13] consider extensions to dynamic networks. In a sustained line of work, Tseng *et al.* [46, 47, 48, 49, 50] investigate approximate agreement problem within various faults models (link crash, process crash, Byzantine) in multi-hop networks (both for the directed and the undirected cases).

Allowing different kinds of faults was investigated by Kieckhafer *et al.* [22], as they unify different algorithms into the class of MSR-algorithms (Mean - Subsequence-Reduced), which compute the mean of a subsequence of the reduced multi-set of values. The authors analyze the convergence rate and the fault-tolerance of this class of algorithm in a so-called *Mixed-Mode faults model*. In this model faults are partitioned into asymmetric (classical Byzantine), symmetric and benign. The benign faults are self-incriminating (immediately self-evident to all non faulty processes). The behavior of symmetric faults is perceived identically to all correct processes, while the asymmetric faults have a totally arbitrary behavior. That is, the behavior of processes being subject to asymmetric faults may be perceived differently by different correct processes.

2.3 Distributed Registers

A Distributed Register (or just Register) is an abstraction that provides two operation, read() to read the value on the register and write(), to write a new value on the register. This abstraction can be accessed by multiple readers and by one or multiple writers. We indicate as SWMR Register the Single Writer Multi Reader Register specification and as MWMR Register the Multi Writer Multi Reader Register specification. Distributed Registers classification have been defined in [24] depending on the operational semantics they provide. In particular those semantics aim to specify values that a read() operation is allowed to return. **Safe Register** is the weakest specification. The only assumption is that a read() operation that is not concurrent with any write() operation obtains the correct value, the most recently written one. Thus, in case the operations are concurrent, a read() operation is allowed to return any value in the register domain. The **Regular Register** is the next stronger specification. Basically, it is a safe register (a read not concurrent with a write returns the correct value) and in which a read() operation that overlaps a write() operation obtains either the old or the new value. The **Atomic Register** is the strongest specification considered. This register is like a regular register in which reads and writes behave as if they occur in some definite order. Informally, the semantic does not allow the following situation: given two consecutive (not concurrent) read() operations, r_1 and r_2, is it not possible that r_1 returns a more recent value with respect to the one returned by r_2 (the so called new old inversion). In [25] algorithms are presented to implement those Register specifications in an asynchronous system, all but the MWMR Atomic Register, whose is presented in [51]. Those works do not take into account any type of failure, [2] is the first work that introduces a solution to implement a SWMR Atomic Register in an asynchronous system prone to crash failures, and in [29], using quorum system, the MWMR Atomic Register problem is solved.
As we state, beside crash failures, Byzantine failures are the most general failures type, and Byzantine fault tolerance is at the core of Distributed Computing. To tolerate Byzantine failures two approaches are possible: (i) verifiable approach, where authenticated communication primitives are used to communicate and (ii) non-verifiable, where those primitives are not available. Concerning the verifiable approach in [12] is provided the first optimal MWMR Atomic Register, where $n > 3f$ is the lower bound on the number of replicas. Concerning the more challenging non-verifiable approach, in [31] is implemented a safe and regular register introducing the so called Byzantine Quorum System. Atomicity is solved in [37] showing that any protocol assuring the regular semantic in presence of Byzantine failures can produce an atomic register leveraging on the writeback mechanism. Finally in [32] a solution is presented matching the lower bound to implement a MWMR Atomic Register. In particular they state that the lower bound on the number of servers to implement a safe register with a *confirmable*[1] protocol is at least $n > 3f$, which matches the lower bounds of the verifiable approach.

[1] If the protocol defines the write completion predicate so that completion can be determined locally by a writer and all writes eventually completes.

Chapter 3

System Model

In this chapter we introduce the basic definitions to characterize the distributed system and the system model where it takes place.

3.1 Processes

A process models the computer program behavior. A distributed system is composed by a set of n processes, each of them running a distributed algorithm and each of them is distinguishable by a unique identifier. We denoted such set as $\Pi = \{p_1, p_2, \ldots, p_n\}$. When we consider the Client-Server paradigm, we consider a distributed system composed of an arbitrarily large set of client processes $\mathcal{C}$ and a set of n server processes $\mathcal{S} = \{s_1, s_2 \ldots s_n\}$. In the following, when it is not necessary to specify, we use the general term *process*.

The passage of time is measured by a fictional global clock 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* (or just *state*) i.e., the set of all its local variables and the corresponding values.

Each process is modeled as an I/O automaton [30]. Automaton actions are classified as either input, output, or internal. An automaton generates output and internal actions autonomously, and transmits an output to its environment (*e.g.,* the process sends a message). In contrast, the automaton input (*e.g.,* the process receives a message) is generated by the environment and transmitted to the automaton. We refer to this interface as the automaton actions signature *sig*.

More formally an I/O automaton is a tuple of the form $\mathcal{A} = \langle sig, \Gamma, st_0, \mathcal{F}, \omega \rangle$, where:

- sig is the actions signature, a finite, non empty set of input, output and internal actions;
- Γ is a finite, non empty, set of states;
- $st_0 \in \Gamma$ is the initial state;
- $\mathcal{F} \in \Gamma$ is a non empty set of final states;

- $\omega : \Gamma \times sig \rightarrow \Gamma$ is the transition function.

When automata run, they generate executions. An execution is an alternated sequence of states st_j and actions π_j starting with the initial state st_0. A distributed protocol is composed by a set of u automata, $\mathcal{P} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_u\}$. The set containing the u executions of all n processes forms the behavior of the distributed system.

3.2 Process Failures

In this work we consider a distributed system prone to failures, in particular Mobile Byzantine failures. We refer to a process experiencing a Byzantine failure as Byzantine process, faulty process or just Byzantine. A Byzantine process, contrarily to a correct process, might deviate in an arbitrarily way from the automaton specification assigned to it. Given an execution, a Byzantine process p_i is assumed to be always faulty Mobile Byzantine fault is an extension of such model, given an execution, a Byzantine process p_i is not assumed to be Byzantine forever, or in other words, all processes can be Byzantine at some point, but the number of Byzantine process can not be more than f at any time. We assume that there are f mobile Byzantine agents (or just mobile agents) that move from a process to another, in such a way that when a mobile agent affects a process, such process is said to be Byzantine. In this case the notion of time has to be explicit, we say that a process p_i is Byzantine at time t or affected by mobile Byzantine agent at time t. We assume that when a process is no more affected by Byzantine failure, it retrieves the correct protocol $\mathcal{P}$ code from a tamper prof memory but the internal state is not predictable. Such process is said to be *cured* at time t. Let $\mathcal{P} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_u\}$ be a protocol such that $\mathcal{A}_i = \langle sig_i, \Gamma, st_{i_0}, \mathcal{F}_i, \omega_i \rangle \ \forall i = 1, \ldots, u$. Intuitively, a correct process never deviates from $\mathcal{P}$ specification. On the contrary, a Byzantine process can be modeled by a process executing a protocol $\mathcal{B} \neq \mathcal{P}$. A cured process is executing $\mathcal{P}$ but with different states with respect to a correct process at the same time t. Let us now give a formal definition of correct, Byzantine and cured processes with respect to the time. To do that let us first introduce the concept of *valid internal state* at time t referred in short as *valid state* at time t.

Definition 1 (Valid State at time t) *Let $st_{i,t}$ be the internal state of a process p_i at some time t. $st_{i,t}$ is said to be a valid state at time t if it does exist a fictional process $\hat{p}_0$ always executing $\mathcal{P}$ such that $st_{0,t} = st_{i,t}$.*

Definition 2 (Correct process at time t) *A process is said to be* correct at time *t if (i) it is 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 3 (Faulty process at time t) *A process is said to be* faulty at time *t if it is controlled by a mobile Byzantine agent and it is executing a protocol $\mathcal{B} \neq \mathcal{P}$ (i.e., it is behaving arbitrarily). 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 4 (Cured process at time t**)** *A process is said to be* cured at time t *if (i) it is 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.,* $Cu([t, t']) = \bigcap_{\tau \in [t,t']} Cu(\tau)$*).*

3.3 Communication models

Processes, in order to run the distributed protocol, need to communicate. In this work we consider the **message-passing model**. In particular, we assume that: *(i)* each process can communicate with every other process through a broadcast() primitive. In the Client-Server paradigm, *(ii)* each client $c_i \in \mathcal{C}$ can communicate with every server through a broadcast() primitive and *(iii)* each server can communicate with a particular client through a send() unicast primitive. We assume that communications are authenticated (*i.e.*, given a message m, the identity of its sender cannot be forged) and reliable (*i.e.*, spurious messages are not created and sent messages are neither lost nor duplicated).

3.4 Time Assumptions

Communication between processes is either synchronous or asynchronous. We assume *(i)* processes internal steps take no time and, as stated before, *(ii)* there is a global clock. We consider two types of system time assumptions: asynchronous and synchronous.
The *asynchronous* system is characterized by no physical timing assumption on processes and communication links. Thus, there exists no upper bound on communications latency. As a consequence, messages are delivered but it is not possible to predict any upper bounds on their delivery time. On the contrary, the *synchronous* system is characterized by the following property: a message m sent at time t from process $p_i \notin B(t)$ to p_j is received by p_j a time t', $t' \leq t + \delta$ and $\delta \geq 0$. Similarly, let t be the time at which a correct process (client) $p_i \notin B(t)$ invokes the broadcast(m) primitive, then there is a constant δ such that all processes (servers) have delivered m by time $t + \delta$. δ is known to every process.

3.5 Computational models

We consider two different computational models, *round-based* and *round-free*. In the round-based system the computation evolves in sequential synchronous rounds $\{r_0, r_1, \ldots, r_i, \ldots\}$. Every round is divided in three phases: *(i)* send, where processes send all the messages for the current round, *(ii)* receive, where processes receive all the messages sent at the beginning of the current round and *(iii)* computation, where processes process received messages and prepare those that will be sent in the

next round. Contrarily to this, in the round-free system there are no rounds and no phases driving the computation.

Chapter 4

Mobile Byzantine Failures

In this chapter we define the Mobile Byzantine Failure (MBF) models, starting with models defined so far for round-based computation, and presenting after them our contribution for round-free computations. The MBF models considered so far in the literature [3, 5, 11, 21, 36, 39, 40] assume that faults, represented by Byzantine agents, are controlled by a powerful external adversary that "moves" them from a server to another. Let us remember 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.

4.1 MBF Models for round-based computations

In all the above cited works the system evolves in synchronous rounds. As we state in the previous chapter, every round is divided in three phases: *send*, *receive* and *computation*. Concerning the assumptions on agent movements and servers awareness on their *cured* state, the Mobile Byzantine Models defined in [5, 21, 11, 40] are summarized as follows:

- *Garay's model* [21] **(M1)**. 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. An example is depicted in Figure 4.1.
- *Bonnet et al.'s model* [5] **(M2)** and *Sasaki et al.'s model* [40] **(M3)**. 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 **(M3)** a cured process still acts as a Byzantine one extra round. Example are depicted in Figure 4.2 and Figure 4.3 respectively.
- *Buhrman's model* [11] **(M4)**. Differently from the previous models, agents move together with the message (i.e., with the send or broadcast operation).

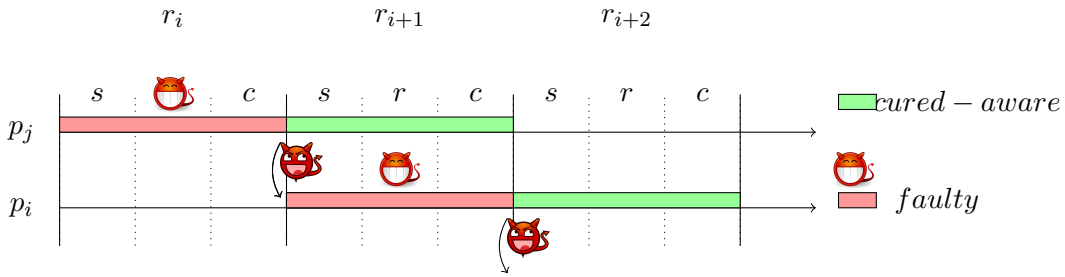


Figure 4.1. Example of a run with Garay's MBF model

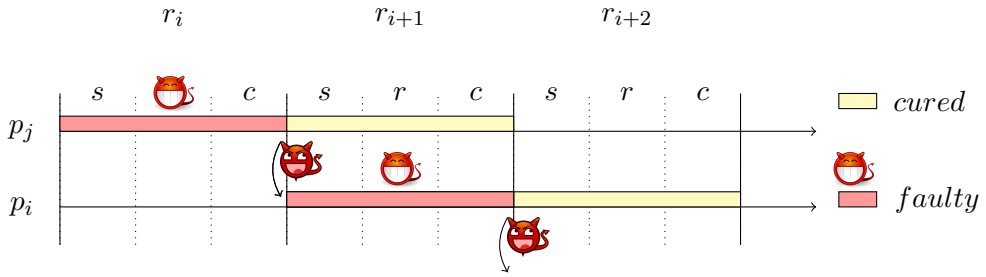


Figure 4.2. Example of a run with Bonnet's MBF model

However, when a server is in the *cured* state it is aware of that. An example is depicted in Figure 4.4.

Previously cited models [5, 21, 11, 40] 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.

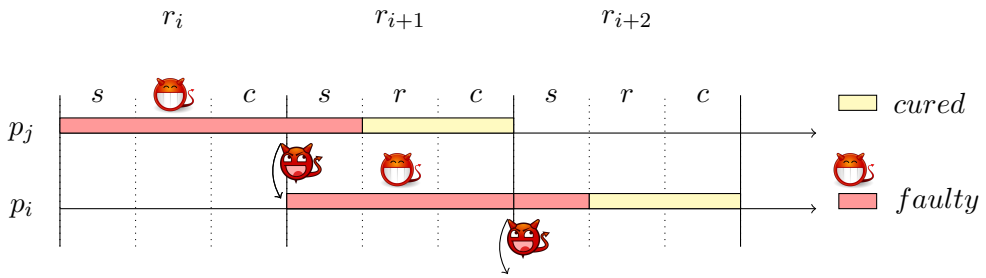


Figure 4.3. Example of a run with Sasaki's MBF model

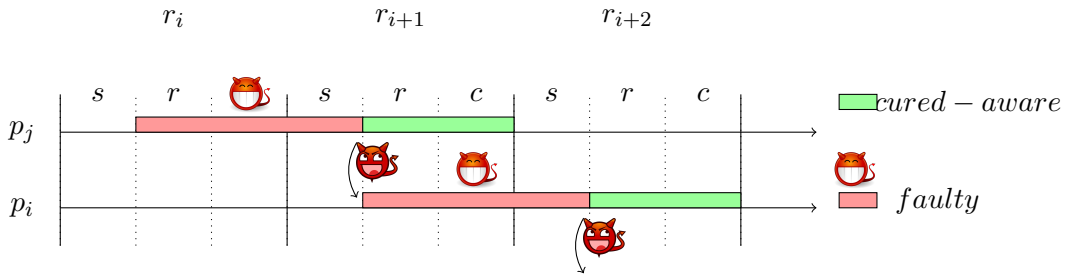


Figure 4.4. Example of a run with Burhman's MBF model

4.1.1 Mobile Byzantine Models for round-free computations

In our framework, we are interested in modeling 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.

Concerning the adversary coordination power, we can distinguish among the following three cases:

- **Δ-synchronized (or synchronized with a period Δ, denoted as ΔS)**: 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 $i \in \mathbb{N}$). An example is shown in Figure 4.5.
- **independent**: the adversary moves each of f mobile Byzantine Agents independently. Independent movements can be further decomposed in:
  - **time-bounded (ITB)**: 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. An example is shown in Figure 4.6.
  - **time-unbounded (ITU)**: 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. An example is shown in Figure 4.7.

Concerning the knowledge that each process has about its failure state, we will distinguish, as for round-based models, among the following two cases:

- **Cured Aware Model (CAM)**: at any time t, any process is aware about its failure state.
- **Cured Unaware Model (CUM)**: at any time t, any process is not aware about its failure state.

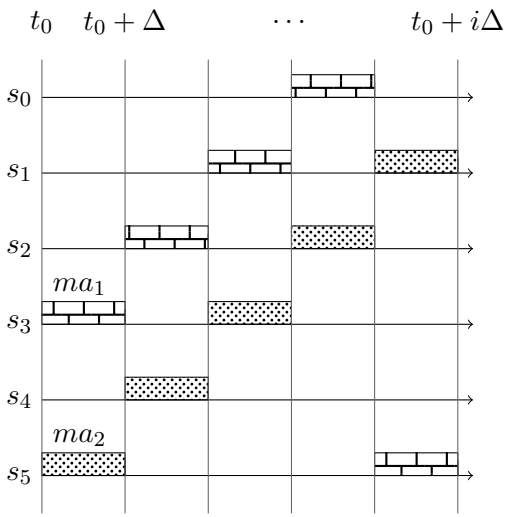


Figure 4.5. Example of a $(\Delta S, *)$ run with $f = 2$.

Any instance of our MBF framework 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). Figure 4.8 shows the six different models obtained by combining the two axis of our round-free MBF framework.

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

- $(\Delta S, *)$ allows to consider coordinated attacks where the external adversary needs to control a subset of machines. In this case, compromising new machines will take almost the same time as the time needed to detect the attack or the time necessary to rejuvenate. This may represent scenarios with low diversity where compromising time depends only on the complexity of the exploit and not on the target server. More formally, the external adversary moves all the f mobile Byzantine Agents at the same time t and movements happen periodically (i.e., movements happen at time $t_0 + \Delta$, $t_0 + 2\Delta$, $\ldots$, $t_0 + i\Delta$, with $i \in \mathbb{N}$) and such periods are known by servers.

- $(ITB, *)$ 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,

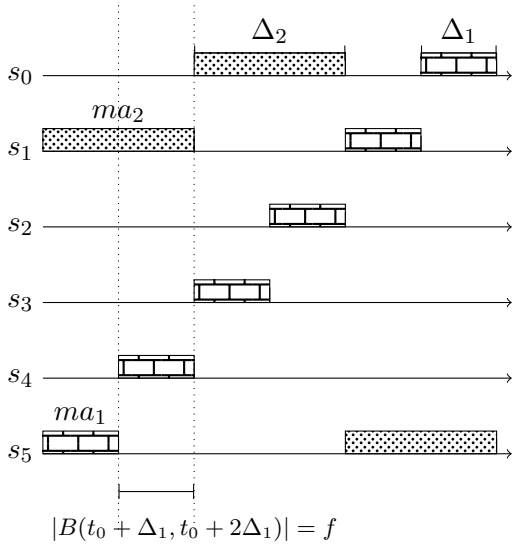


Figure 4.6. Example of a $(ITB, *)$ run with $f = 2$.

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 most 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.

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

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

- $(*, CUM)$ 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 executes the correct protocol however its internal state is not aligned with non compromised servers.

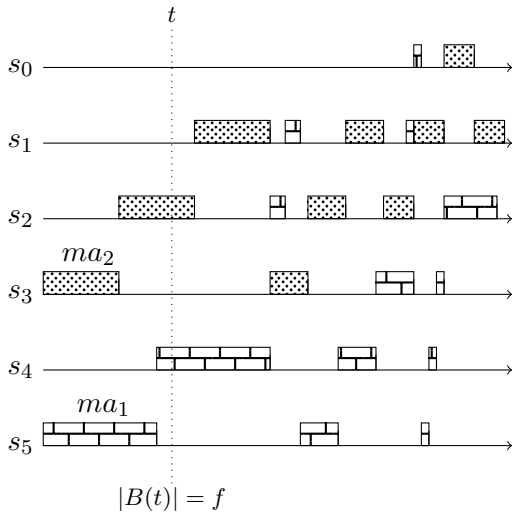


Figure 4.7. Example of a $(ITU, *)$ run with $f = 2$.

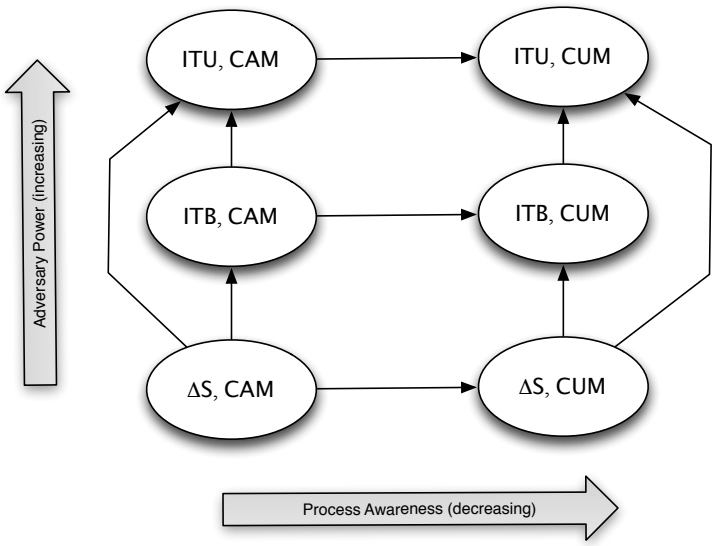


Figure 4.8. MBF model instances for round-free computations and their relations.

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.

The instance $(\Delta S, CAM)$ is the strongest one as it is the most 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 work and the reader may refer to [36] for further details.

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

Chapter 5

Distributed Registers in the Round Based Model

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

Given two operations op and op', their invocation times ($t_B(op)$ and $t_B(op')$) and reply times ($t_E(op)$ and $t_E(op')$), we say that *op precedes* op' ($op \prec op'$) if and only if $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* (noted $op||op'$). Given a write(v) operation, the value v is said to be written when the operation is complete.
In this chapter we consider the atomic register specifications.

MWMR Atomic Register

The Multi-Writer/Multi-Reader (MWMR) atomic register is specified as follow:

- Termination: Any operation invoked on the register eventually terminates.
- Validity: A read() operation, if it does not overlap any write() operation, returns the last value written before its invocation (*i.e.*, the value written by the latest completed write() preceding it).
- Ordering: There exists a total order S such that (i) any operation invoked on the register belongs to S, (ii) given op and op' belonging to S, if $op \prec op'$, then

op appears before *op*′ in S and (iii) any read() operation returns the value v written by the last write() preceding it in S.

Impossibility results are stated in the next section (Section 5.2). For simplicity those results are proven using the weak register specification, the *s*afe register (weaker than the regular register in the Lamport's hierarchy [23]). Contrarily to the Atomic register, a read() operation on a safe register concurrent with a write operation may return any value in the register domain.

SWMR Safe Register

A single-writer/multi-reader (SWMR) safe register [23] 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, if it does not overlap any write() operation, returns the last value written before its invocation (*i.e.*, the value written by the latest completed write() preceding it).

5.2 Impossibilities

In this section we start to present new problems that arise to design a MBF tolerant protocol. In particular what is the impact of mobile Byzantine movements and the consequent change of servers failure state. In the sequel we prove that in the case of MBF tolerant implementations a new operation, that we name maintenance(), must be implemented to prevent servers from losing the current register value.

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() *operations. 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-based computations.*

Proof Let us suppose by contradiction that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ is a correct protocol implementing a safe register. If $\mathcal{P}_{reg}$ is correct, it means that both $\mathcal{A}_R$ and $\mathcal{A}_W$ implementing respectively the read() and the write() operations terminate i.e., they stop to execute steps when the operation is completed. Let r be the round at which some operation *op* terminates and let us assume that no other operation is invoked until round $r' > r$. Let us note that during the interval $[r, r']$ no algorithm is running as all the operations issued in the past are completed. As a consequence, no correct server and no cured server change its state by themselves. However, considering that r' does not depend on $\mathcal{P}_{reg}$ (i.e., it is not controlled by the register protocol but it is defined by clients) and considering the mobility of the Mobile Byzantine agents, we may easily have a run where every correct server is faulty and its state is corrupted at some round in $[r, r']$. Considering that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ and that $\mathcal{A}_R$

and $\mathcal{A}_W$ are not running in $[r, r']$ we can have that every server stores a non valid state at round r' and the register value is lost. As a consequence, $\mathcal{A}_R$ has no way to read a valid value after r' and the validity (or termination, depending on how the algorithm is implemented) property is violated. It follows that $\mathcal{P}_{reg}$ is not correct and we have a contradiction. $\square_{Theorem\ 1}$

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

Definition 5 (maintenance() and $\mathcal{A}_M$) *A* maintenance() *operation is an operation that, when executed by a process p_i, terminates at some point during round r so that p_i has a valid state at the beginning of round $r + 1$ (i.e., it guarantees that p_i is correct at round $r + 1$). 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[1] so that the corollary follows:

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

Lemma 1 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in any round-based Mobile Byzantine Failure Model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_M$ must involve at least one round.*

Proof The claim follows by considering that cured servers have a compromised state and they need to receive information from correct servers in order to be able to update their state to a valid one. It follows that at least one communication step is required, which means at least one round. $\square_{Lemma\ 1}$

5.2.1 Discussion

In the previous works ([21, 3, 5, 11]) has never been pointed out the maintenance necessity condition. The reason is that the aim of those works is to solve consensus, which is a single operation and whose solution requires algorithms evolving in rounds ([20]), thus in a synchronous system there are no periods in which the protocol does not perform operations while mobile agents are moving. The only form of maintenance, in the mobile Byzantine case, is performed when consensus terminates, thus at the end of the operation. In this case algorithms keep on exchanging information to do not lose the reached agreement. Informally we can say that

[1] 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.

the maintenance() operation is embedded in the previous work given the problem nature (solution itself requires a continuous values exchange). In the register case, more operations occur and those operation may not be sequential, thus an explicit maintenance operation is required. In a toy scenario where the writer is continuously executing the write() operation the maintenance() operation would be redundant.

5.3 Lower Bounds

In this section we prove impossibilities on the number of correct servers to implement an atomic register for the four models [5, 21, 11, 40] presented in Chapter 4. An algorithm that matches those impossibilities is presented in Section 5.4, proving that results presented in the sequel are lower bounds for the considered problem.

To this aim we start proving impossibilities on the number of correct servers to implement a safe register, whose results can be directly extended to atomic register. We represent each server s_i state at each round r as the composition of the internal state and faulty state: $\langle values, fstate \rangle_{s_i,r}$, where s_i is the server identifier we are referring to at round r. $values$ represents the internal variables values of s_i (including the last written value) and $fstate$ is the failure state at round r, such that $fstate \in \{correct, cured, faulty\}$. We represent $fstate$ for simplicity in each considered failure models. Notice that in Garay [21] and Buhrman [11] models this information is available at server side contrarily to Sasaki [40] and Bonnet [5] models. Without loss of generality in the following proofs we consider that one server may be affected by a Byzantine agent each round, i.e., $f = 1$. To extend the proof then it is sufficient to substitute the faulty server with a set of f faulty servers.

As shown in Section 5.2, the $\mathcal{A}_M$ algorithm is necessary (cf. Corollary 1), therefore any protocol solving safe register has to be twofold, on one side it has to (i) allow cured servers to turn in correct and to (ii) allow a client, if there are no concurrent write() operation, to return the last written value during a read() operation. In the following impossibility proofs we violate one of the two. In particular concerning the $\mathcal{A}_M$ algorithm we consider the following. For Lemma 1 such operation requires at least one round in which correct servers exchange the information necessary for a cured server to become correct. From the system model, the computation evolves in rounds and at each round the set of Byzantine servers may change. To match the result in Lemma 1 we consider that at the beginning of each round servers, during the send phase of the round, broadcast information each others and during the delivery phase, at the end of the same round, servers collect those information.

Theorem 2 *If $n \leq 3f$, there exists no protocol $\mathcal{P}$ solving a safe register in Garay's model [21].*

Proof Let us suppose by contradiction that there exists a protocol $\mathcal{P}$ implementing the safe register in the Garay's model with $n = 3f$. Let r_1 be the first round of the computation. Assuming that the mobile agent moves at each new round, it follows that during next round r_2, the first cured server appears (server that was affected during round r_1).

Let E_1 and E_2 be two executions where at the beginning of each round servers exchange the value *values* they are storing. Execution E_1 characterized by the following three servers states at the beginning of round $r2$: $\{\langle v, correct\rangle_{s_0,r_2}, \langle v', faulty\rangle_{s_1,r_2}, \langle \perp, cured\rangle_{s_2,r_2}\}$. During round r_2 cured server s_2 collects other servers values: $\{v, v'\}$[2]. Since $\mathcal{P}$ exists then s_2, at the end of the round, changes its state storing the same values as the correct server s_0, $\langle v, correct\rangle_{s_2,r_2}$. Execution E_2 is characterized by the following three servers states at the beginning of round $r2$: $\{\langle v', correct\rangle_{s_0,r_2}, \langle v, faulty\rangle_{s_1,r_2}, \langle \perp, cured\rangle_{s_2,r_2}\}$. During r_2 cured server s_2 collects the following servers replies: $\{v, v'\}$. By hypothesis $\mathcal{P}$ implements a safe register, then s_2, at the end of the round, changes its state storing the same values as the correct server s_0, $\langle v', correct\rangle_{s_2,r_2}$. In both executions s_2 collects the same sets of replies but ends up with different states, leading to a contradiction. To conclude the proof let us consider the case in which the maintenance() operation lasts more than one round, then it is straightforward that there is no advantage. In this case the reply set collected during the round after can be either the same (if the mobile agent does not move, but then we can still build two indistinguishable executions) or different, but such difference is due to the presence of two cured servers and a Byzantine server, so there are no more correct servers. Thus if $n = 3f$ there is no maintenance() operation and from Theorem 1 safe register cannot be implemented. $\square_{Theorem\ 2}$

The previous proof is constructed on the fact that when $n \leq 3f$, cured processes cannot recover the correct state. Therefore, a client cannot return the last written value. Next proofs advocate that even though the number of correct servers increases, a client may return different values based on the same set of collected values. To be as general as possible consider that a read() operation is composed by a request phase and of t, $t \geq 1$, reply phases. In each of those reply phases servers send to the client their stored value. A read() operation whose duration is not fixed is considered because mobile agents move round after round, this means that the system composition changes respect to the faulty servers and changes at each round the set of correct servers that reply. Thus one may think that after a certain amount of time it could be possible to read. Let further assume that each server is aware of having been affected in the previous round and sends back this information to client. Notice, we are assuming that correct servers know if they were correct or not in the previous round, indifferently from the failure model considered. In this way we are giving to servers as much power as possible, despite that, we prove impossibilities. We assume that reply messages contain the following information: $\langle value, fstate_{r-1}\rangle_{s_{jr}}$, where *value* is the value stored by s_j at round r and $fstate_{r-1} \in \{correct, non\_correct\}$ is the failure of the server in the previous round, notice that *non_correct* is either *Byzantine* or *cured*.

Theorem 3 *If $n \leq 4f$, there exists no protocol $\mathcal{P}$ implementing a safe register in Sasaki's model [40].*

Proof Let us suppose that $n = 4f$ and that the $\mathcal{P}$ protocol does exist. Let r_1 be the first round of the computation, such that during r_2 the first cured server appears

[2]In Garay's model s_2 is aware of being cured, so it can ignore its own value.

(server that was affected during round r_1). Let us consider that at round r_2 client c_i issues a read() operation, such that during r_2 the request phase takes place and during r_3 until r_{3+t}, the t reply phases take place. Let *value* be the value stored by correct servers, so that each server s_j replies as follows:

- (case 1) if s_j is faulty then it replies with $\langle value', non\_correct \rangle$;
- (case 2) if s_j is cured then it replies with $\langle value', correct \rangle$;
- (case 3) if s_j is correct, but was cured in the previous round then it replies with $\langle value, non\_correct \rangle$;
- (case 4) if s_j is correct and was correct also in the previous round then it replies with $\langle value, correct \rangle$;

Let E_1 be an execution where $f = 1$ and v is the value stored by correct servers. Let us consider that mobile agent affects at each round r_i a different server in the following way: given r_i the mobile agent is on $s_{(i \mod n)}$, for simplicity let us denote such servers as $s_{\mathcal{S}_1(i)}$, 1 is because we are referring to case 1, as defined earlier. It follows that cured server at r_i is $s_{\mathcal{S}_2(i)} = s_{(i-1) \mod n}$ if $i > 1$, the correct server that was previously cured is $s_{\mathcal{S}_3(i)} = s_{i+2 \mod n}$ while the correct server that was correct also in the previous round is $s_{\mathcal{S}_4(i)} = s_{i+1 \mod n}$, as depicted in Figure 5.1. In E_1 at round r_2 the request phase takes place and from round r_3 to round r_{3+t}, c_i collects the following replies:
$\{\langle v, non\_correct \rangle_{s_1,r_3}, \langle v', correct \rangle_{s_2,r_3}, \langle v', non\_correct \rangle_{s_3,r_3}, \langle v, correct \rangle_{s_4,r_3},$
$\ldots, \langle v, non\_correct \rangle_{s_{\mathcal{S}_3(3+t)},r_{3+t}}, \langle v', correct \rangle_{s_{\mathcal{S}_2(3+t)},r_{3+t}}, \langle v', non\_correct \rangle_{s_{\mathcal{S}_1(3+t)},r_{3+t}},$
$\langle v, correct \rangle_{s_{\mathcal{S}_4(3+t)},r_{3+t}}\}$. Since $\mathcal{P}$ exists then c_i returns v.
Let E_2 be an execution where $f = 1$ and v' is the value stored by correct servers and a mobile agent is affecting at each round r_i a different server, in particular at r_i the affected server is $s_{(n-i \mod n)}$ such that the cured server is $s_{(n-(i-1) \mod n)}$ from $i \geq 1$. Fixed those two servers in each round $s_{(n-(i-2) \mod n)}$ (from $i \geq 2$) is the correct server that was previously cured. $s_{(n-(i-3) \mod n)}$ is the correct server that was correct also in the previous round, with $i \geq 3$, if $i = 2$ we consider s_2 as the resulting server. Such scenario is depicted in Figure 5.2. In E_2 at round r_2 the request phase takes place and from round r_3 to round r_{3+t}, c_i collects the following replies:
$\{\langle v', non\_correct \rangle_{s_1,r_3}, \langle v', correct \rangle_{s_2,r_3}, \langle v, non\_correct \rangle_{s_3,r_3}, \langle v, correct \rangle_{s_4,r_3},$
$\ldots, \langle v', non\_correct \rangle_{s_{\mathcal{S}_1(3+t)},r_{3+t}}, \langle v', correct \rangle_{s_{\mathcal{S}_2(3+t)},r_{3+t}}, \langle v, non\_correct \rangle_{s_{\mathcal{S}_3(3+t)},r_{3+t}},$
$\langle v, correct \rangle_{s_{\mathcal{S}_4(3+t)},r_{3+t}}\}$. Since $\mathcal{P}$ exists then c_i returns v'.

In both executions c_i returns different values even though it collects the same set of replies, thus there exist no protocol $\mathcal{P}$ solving the safe register if $n \leq 4f$. Notice that this is true despite that the mobile agent affects different servers in the two executions and despite the variable duration of the read() operation, which concludes the proof. $\square_{Theorem\ 3}$

Corollary 2 *If $n \leq 4f$, there exists no protocol $\mathcal{P}$ in Bonnet model [5].*

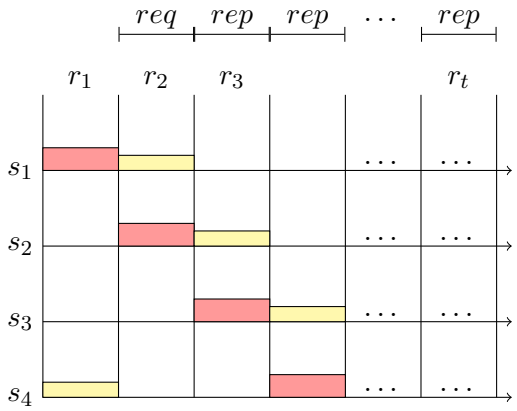


Figure 5.1. Scenario representing the mobile agent movement in execution E_1

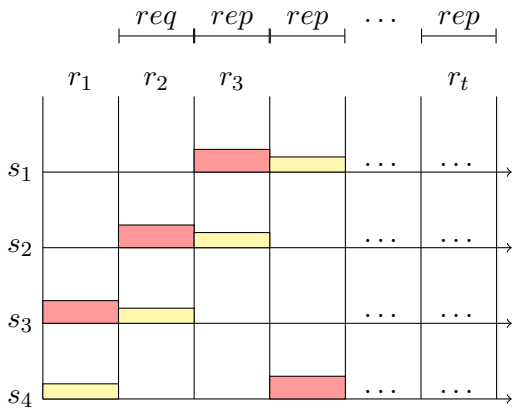


Figure 5.2. Scenario representing the mobile agent movement in execution E_2

Proof The claim simply follows by considering that the Bonnet model is a particular case of Sasaki model, in which cured servers act as less powerful faulty servers, forced to send the same message to all. The same reasoning as in the proof of Theorem 3 is applied. $\square_{Corollary\ 2}$

In the Burhman's model [11] a key role is played by the moment at which mobile agents move. In this case mobile agents, rather than moving at the beginning of the round, move during the sending phase. Moreover cured servers, as in Garay's model, are aware about their failure state.

Theorem 4 *If $n \leq 2f$, there exists no protocol $\mathcal{P}$ implementing a safe register in the Burhman's model [11].*

Proof The proof is direct considering that to tolerate Byzantine servers, with non-confirmable writes, $n \geq 2f + 1$ is the minimal required number of servers [33]. $\square_{Theorem\ 4}$

5.4 Upper Bounds

In this section, we present an algorithm $\mathcal{A}_{reg}$ implementing a MWMR Atomic Register resilient to the presence of up to f mobile Byzantine agents for the four Round-Based MBF models (i.e., M1-M4). The algorithm follows the basic quorum-based approach to implement read() and write() operations.

Let us recall that mobile Byzantine agents move from one server to another corrupting their internal states. As a consequence, if not properly mastered, this can bring to the compromising of all the servers and to the loss of the register value (cf. Theorem 1). A naive solution would be to exploit write() operations to clean values of cured processes and increase the number of replicas n to ensure the presence of "enough" correct servers to select a valid value. However, such solution has two strong drawbacks: (i) write() operations 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 grows immediately linearly in the number of rounds between two following write() operations (as the number of cured servers grows).
To handle the presence of mobile Byzantine agents, we started from this intuition and we defined a value propagation mechanism that is used to help cured servers to recover and to update their local variables to a correct state. Such mechanism is executed at the beginning of each round and it pushes information between servers allowing cured ones to become correct in one round. The immediate benefit is the reduction of the number of replicas required to master the mobility.

The algorithm presented in the following is defined in a parametric way in order to fit all the four round-based mobile Byzantine failure models presented in Chapter 4. The first parameter of the algorithm, denoted as α, is used to relate the global number of required servers n to the number of mobile Byzantine agents f that can be tolerated. In particular, we relate such two values by the following inequality $n \geq \alpha f + 1$ with $\alpha \in \{2, 3, 4\}$ depending on the mobile Byzantine failure model considered. The second parameter, denoted as β, is used to define the minimal number of occurrences of a same value that a client needs to collect in order to

Failure model	Mid	α	β	Oracle
Garay [21]	M1	3	2	enabled
Bonnet *et al.* [5]	M2	4	2	disabled
Sasaki *et al.* [40]	M3	4	2	disabled
Burhman *et al.* [11]	M4	2	1	enabled

Table 5.1. $\mathcal{A}_{reg}$ parameters for the four different Mobile Byzantine Failure models.

select a valid value at the end of the read() operation. Such number is denoted by s and it is defined as $s = n - \beta f$, with $\beta \in \{1, 2\}$. Finally, in order to abstract the knowledge that a server has of its failure state (i.e., *cured* or *correct*), we introduce the *cured_state* oracle. When invoked via report_cured_state() function, it returns, in the Garay [21] and Buhrman *et al.* [11] models, true to *cured* servers and false to others. In this case the oracle is said to be enabled. In Sasaki *et al.* [40] and Bonnet *et al.* [5] model the *cured_state* oracle returns always false. In this case the oracle is said disabled. The implementation of the oracle is out of scope of this work and the reader may refer to [14], [36] for further details.

Table 5.1 summarizes the above parameters for each model.

5.4.1 $\mathcal{A}_{reg}$ Algorithm Detailed Description

The pseudo-code of the algorithm is presented in Figures 5.3-5.5. The algorithm exploits the round based nature of the system.
Any write() operation spans at most two rounds. The operation may, in fact, be invoked in the middle of a round and in this case it effectively starts in the send phase of the next round r. The writer broadcasts the value and all servers deliver it in the same round r. In the receive phase of the same round, servers delivers WRITE() messages and, if more than one write() operation is executed in the same round, servers update the register by selecting the value coming from the client with the highest identifier. Due to the synchrony assumptions no acknowledgement message is required and the operation can terminate at the end of the round r.
The read() operation spans at most three rounds. As for the write(), it effectively starts with the send phase of the round starting after its invocation, and takes such round to send a read request to servers, and the following one to gather replies. In the computation phase of the round after, the reader selects the value occurring at least $s = n - \beta_{Mi} f$ times concluding the operation.

The value propagation mechanism is implemented by letting servers disseminate the stored value through ECHO() messages at the beginning of each round. Such ECHO() messages are collected during the receive phase and are used by cured processes to select a value and to update their value of the register. In such way, they are able to cope with f servers that may have lost their value during the previous round due to the Byzantine mobility.

Local variables at client c_i. Each client c_i manages the following variables to implement the read() operation:
$-$ $op_R\_start_i$: is a variable that keeps track of the state of a read() operation at client c_i and it can have the following values: $\{0 =$ request_round$, 1 =$ reply_round$, \perp =$

```
Init():
(1)  value_i ← ⊥;
-----------------------------------------------------------------
At the beginning of each round r
(2)  echo_vals_i ← ∅;
(3)  current_writes_i ← ∅;
(4)  cured_i ← report_cured_state();
-----------------------------------------------------------------
Send Phase of round r
(5)  if (¬cured_i)
(6)     then broadcast ECHO(value_i, i);
(7)          for each j ∈ current_reads_i do
(8)              send REPLY(value_i, i) to c_j;
(9)          endFor
(10) endif
(11) current_reads_i ← ∅;
-----------------------------------------------------------------
Receive Phase of round r
(12) for each ECHO(v, j) message delivered do
(13)     echo_vals_i ← echo_vals_i ∪ {v};
(14) endFor
(15) for each WRITE(v, j) message delivered do
(16)     current_writes_i ← current_writes_i ∪ {< v, j >};
(17) endFor
(18) for each READ(j) message delivered do
(19)     current_reads_i ← current_reads_i ∪ {j};
(20) endFor
-----------------------------------------------------------------
Computation Phase of round r
(21) if (current_writes_i ≠ ∅)
(22)    then let v such that ∃ < v, j >∈ current_writes_i
(23)              ∧j = arg max_k(< −, k >∈ current_writes_i);
(24)         value_i ← v;
(25)    else if (∃v ∈ echo_vals_i | #occurrence(v) ≥ n − β_M i f)
(26)            then value_i ← v;
(27)            else value_i ← ⊥;
(28)         endif
(29) endif
```

Figure 5.3. $\mathcal{A}_{reg}$ implementation: code executed by any server s_i.

no_read_running}.
− $replies_i$: is a set that collects REPLY messages during a read() operation. It is set to $\emptyset$ at the beginning of the operation.

Local variables at server s_i. Each server s_j manages the following variables:
− $value_i$: it stores the current value of the register.
− $echo\_vals_i$: is a set variable (emptied at the beginning of each round) where servers store the ECHO messages received in the current round.
− $current\_writes_i$: is a set variable (emptied at the beginning of each round) where servers store values received through a WRITE() message.
− $current\_reads_i$: is a set variable where servers store identifiers of clients that are currently reading. It is emptied after the reply to such clients.
− $cured_i$: is a boolean variable set through the report_cured_ state() event. It is set to true by the cured_state oracle (if enabled) when s_i is in a *cured* state. Otherwise it is always false.

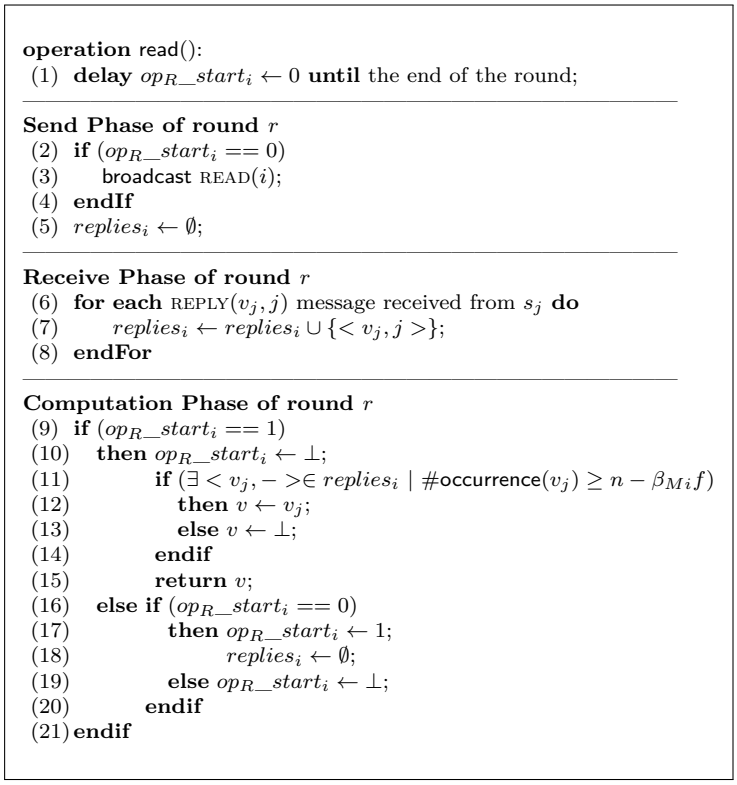


Figure 5.4. $\mathcal{A}_{reg}$ implementation: code executed by any client c_i for the read() operation.

Server maintenance. In the *send* phase of each round, each servers s_i, whose variable $cured_i$ is set by the oracle as false, performs the broadcast of ECHO(val, i) message (line 6, Figure 5.3). If no write() operations happen in the current round (the condition at line 21 is not verified), then servers use such collected values are then used during the *computation* phase (line 25, Figure 5.3) to select a value occurring at least $n - \beta_{Mi} f$ times and update their state.

Write operation. In order to write a value v a client c_i has to broadcast the WRITE(v, i) message to all servers (line 1, Figure 5.5). Since an operation invocation may happen in any time during a round, then the broadcast() is delayed until the next send phase. At the server side this message is delivered within the same round during the *receive* phase and any *correct* and *cured* server s_j stores it in $current\_writes_j$ set (lines 15-16, Figure 5.3). At the end of the round, during the *computation* phase, if $current\_writes_j$ is not empty then the value associated to the highest client identifier is stored in $value_j$ (lines 21-24, Figure 5.3). Back to the client side, during its *computation* phase, it returns the write_confirmation to the application layer (line 4, Figure 5.5).

Read operation. When a read() operation is invoked by a client c_i, $op_R\_start_i$ is set to 0 at the end of the current round (line 1, Figure 5.4), thus at the next *send* phase the condition at line 2 is true and the READ(i) message is broadcast (line 3). Regardless the value of $op_R\_start_i$ at each round the $replies_i$ set is emptied

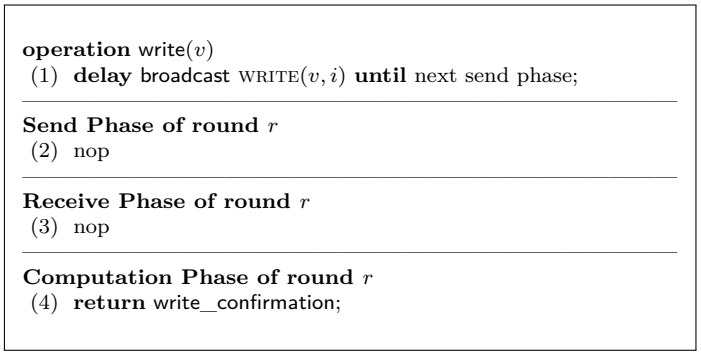


Figure 5.5. $\mathcal{A}_{reg}$ implementation: code executed by any client c_i for the write() operation.

(line 18). In the *computation* phase, the condition at line 16 is true ($op_R\_start_i$ is equal to 0) and $op_R\_start_i$ is set to 1. This means that the read_request phase is over and the next one is the read_reply one. At server side (Figure 5.3), the READ(i) message is delivered within the same invocation round. Once the message is delivered, any server s_j stores the identifier of the reader in the $current\_reads_j$ set in order to send back a REPLY() message at the beginning of the next round (lines 18-19, Figure 5.3).
At client side (Figure 5.4), when the next round begins, the condition at line 2 is not true, thus during the *send* phase the $replies_i$ set is emptied. Such set is filled with REPLY($value_j$) messages during the *receive* phase (lines 6 - 8, Figure 5.4). During the *computation* phase the condition at line 9 is true, thus $op_R\_start_i$ is set to $\perp$ and the value in $replies_i$ which occurs at least $n - \beta_{Mi}f$ times is returned to the application layer (lines 8-15, Figure 5.4).

5.4.2 Correctness proofs

Lemma 2 *Any* write() *operation eventually terminates.*

Proof The proof follows by considering that the write() operation generates a write_confirmation event at the end of the computation phase in which the operation is effectively started (line 4, Figure 5.5). $\square_{Lemma\ 2}$

Lemma 3 *Any* read() *operation eventually terminates.*

Proof When a reader invokes a read() operation op_r at round r, it executes line 1 in Figure 5.4 by setting $op_R\_start_i = 0$ just before entering in the send phase when it sends the read request, let us say at round $r + 1$. Then $op_R\_start_i$ is set to 1 in the computation phase of $r + 1$ (line 17, Figure 5.4). During the computation phase of round $r + 2$, c_i executes lines 9-15, Figure 5.4 returning from the operation and the claim follows.

$\square_{Lemma\ 3}$

Theorem 5 (Termination) *Any operation invoked on the register eventually terminates.*

Proof The proof directly follows from Lemma 2 and Lemma 3. $\square_{Theorem\ 5}$

Lemma 4 *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 5.1 and used by the algorithm in Figures 5.3-5.5. Let $n > \alpha_{Mi} f$ for each failure model Mi considered. At the end of each round at least $n - f$ correct servers store the same value v in their $value_i$ local variable.*

Proof The proof is done by induction.

- **Basic Step.** At the end of each round, each non-faulty server updates its $value_i$ local variable (i) in line 24 (i.e., if there exists at least a pair in the $current\_writes_i$ local variable) or (ii) in line 26 (i.e., $current\_writes_i$ is empty and there exist at least $n - \beta_{Mi} f$ same values in $echo\_vals_i$).

  Let us recall that at round r_0 all correct servers store the same default value $\perp$ in their local variable $value_i$. As a consequence, in r_0 there exists at least $n - 2f$ (f are Byzantine and f are cured, the remaining servers are correct) correct servers storing v.

  Let us first prove that one of the two cases always happens and then we prove that the number of non-faulty servers storing the same values v at the end of r_0 is $n - f$.

  The $current\_writes_i$ local variable is initialized by any non-faulty server s_i to $\emptyset$ at the beginning of each round r (cfr. line 3) and it is updated when a WRITE() message is received by s_i[3]. Thus, case (i) corresponds to a scenario where at least a write() operation is executed in round r_0 and case (ii) corresponds to a scenario where no write() is running.

  - **Case (i): $\mathbf{current\_writes_i} \neq \emptyset$.** In this case the claim simply follows by observing that the $current\ \_writes_i$ local variable is filled in when servers deliver a WRITE() message. Considering that (i) writer clients broadcast a WRITE(v, j) message in the send phase of round r, (ii) clients are correct and send the same set of values to all servers that will apply a deterministic function to select the value v and (iii) at most f servers are faulty and may skip the update of their $value_i$ variable, the claim follows.
  - **Case (ii): $\mathbf{current\_writes_i} = \emptyset$ and line 25 is true.** In this case, the $value_i$ variable is updated according to the values stored in $echo\_vals_i$. Such variable is emptied by every non-faulty process at the beginning of each round (cfr. line 2) and is filled in when an ECHO() message is delivered. Such message is sent at least by any server, believing it is correct, at the beginning of each round.

    At the beginning of r_0, at least $n - f - x$ correct servers will send an ECHO(v, j) message, where x is the number of non-faulty processes that become faulty in r_0 (i.e., $x = f$ for all the models but Burhman's one where $x = 0$ as faulty processes move during the send phase and not at

[3]Recall that such WRITE() message is sent by the writer client in the send phase of the first round starting after the write() invocation and it is delivered by any non-faulty server in the same round.

the beginning of the round). Let us note that the condition in line 25 is verified if and only if $n - 2f \geq n - \beta_{Mi} f$ that is true in any model ($n - 2f$ is the number of correct servers sending the ECHO() message in r_0). Therefore, considering that at the end of round r_0 non-faulty servers are exactly $n - f$, we have that $n - f$ processes will execute this update.

- **Inductive Step.** Iterating the reasoning for any r the claim follows.

$\square_{Lemma\ 4}$

Theorem 6 (Validity) *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 5.1 and used by the algorithm in Figures 5.3-5.5. Let $n > \alpha_{Mi} f$ for each failure model, Mi, considered. A* read() *operation, if it does not overlap any* write() *operation, returns the last value written before its invocation (*i.e.*, the value written by the latest completed* write() *preceding it).*

Proof Let r_{w1} be the round in which op_w terminates and let v_0 be the value written by op_w. Without lost of generality, let us consider the first write(v) operation op'_w and the first read() operation op_r issued after r_{w1}. Three cases may happen: (i) $op_r \prec op'_w$, (ii) $op'_w \prec op_r$ and (iii) $op'_w \mid\mid op_r$. Let us note that op_r spans over at least two rounds and during the first one the client sends the READ() message while in the second one it collects replies.

- **Case (i): $\mathbf{op_r \prec op'_w}$.** This case follows directly from Lemma 4 considering that (i) at the end of the first round of op_r at least $n - f$ correct processes have the same value v_0 written by op_w, (ii) while moving to the second round of op_r, at most x processes can get faulty (with $x \leq f$ for models M1-M3 and $x = 0$ for M4), (iii) $n - f - x \geq n - \beta_{Mi} f$ (i.e., $\beta_{Mi} f \geq f + x$) for each model (i.e., there will always be enough replies from correct servers to select a value) and (iv) $n - \beta_{Mi} f > f$ (i.e., $(\alpha_{Mi} - \beta_{Mi}) f + 1 > f$) for each model. It follows that faulty processes cannot force the client to select a wrong value and the claim follow in this case.

- **Case (ii): $\mathbf{op'_w \prec op_r}$.** Let $r_{w'}$ be the round at which op'_w terminates and let $r_{w'} + 1$ be the round at which op_r is invoked. Due to Lemma 4, at round $r_w + 2$ there are at least $n - \beta_{Mi} f$ of the last written value. So, applying the same reasoning of case (i) the claim follows.

- **Case (iii): $\mathbf{op'_w \mid\mid op_r}$.** Let us note that a read() operation spans two rounds, i.e., the round of the request r_{req} and the round of the reply r_{reply}. So, let us consider them separately.

  - **Case (iii.a):** op'_w is concurrent with op_r during r_{req}. In that case the value v is delivered to correct server at the end of r_{req}. Due to Lemma 4, at the end of r_{req} at least $n - f$ correct servers store the new written value v, we fall down into case (ii) and the claim follows.

– **Case (iii.b):** op'_w is concurrent with op_r during r_{replay}. Since, in every round, the send phase is executed before the receive phase, it follows that at least all the correct servers will reply with the value written before the invocation of the write() operation, we fall down into case (i) and the claim follows.

$\square_{Theorem\ 6}$

Theorem 7 (Ordering) *Let α_{Mi} and β_{Mi} be the parameters for each of the 4 failure models Mi as reported in Table 5.1 and used by the algorithm in Figures 5.3-5.5. Let $n > \alpha_{Mi} f$ for each failure model Mi considered. There exists a total order S such that (i) any operation invoked on the register belongs to S, (ii) given op and op′ belonging to S, if $op \prec op'$, then op appears before op′ in S and (iii) any* read() *operation returns the value v written by the last* write() *preceding it in S.*

Proof Let r_{w1} be the round in which op_w terminates and let v_0 be the value written by op_w.

In order to prove the claim, we have to show that the algorithm in Figures 5.3-5.5 is eventually able to build a total order of operations S that preserves (i) the *read from last write* property and that includes all the operations from a certain round on.

Let us observe the following:

1. any write() operation is "effectively" executed in one round (i.e., the round in which the value is propagated) even if it has been invoked during the previous round;

2. any read() operation is "effectively" executed in two rounds (i.e., r_{req} the round in which the request for reading the value is sent to servers and r_{rep} where replies are collected at the client side) even if it has been invoked during the previous round;

3. at the beginning of any round $r > r_{w1}$, since no more transient failures are going to happen, there always exist at least $n - 2f$ correct servers storing the same value v (see Lemma 4);

4. correct servers answer to read request by sending back their local values.

Let us suppose by contradiction that a total order S does not exists. S cannot exist iff the scenario in Figure 5.6 happens. However, considering the observations above and that the algorithm evolves in synchronous rounds, all the possible executions follow patterns similar to those shown in Figure 5.7, i.e., there can not exist a write() operation that overlaps two different read() operations op_{r1} and op_{r2} such that $op_{r1} \prec op_{r12}$, from which we have a contradiction. $\square_{Theorem\ 7}$

Theorem 8 *Let $\mathcal{A}_{reg}$ be the algorithm in Figures 5.3-5.5 and let $n \geq \alpha f$. If $\alpha = 3$, for each round r $\mathcal{A}_{reg}$ implements a MWMR Atomic register in the Garay model.*

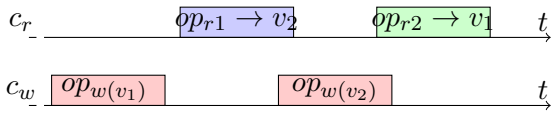


Figure 5.6. An example of new/old inversion.

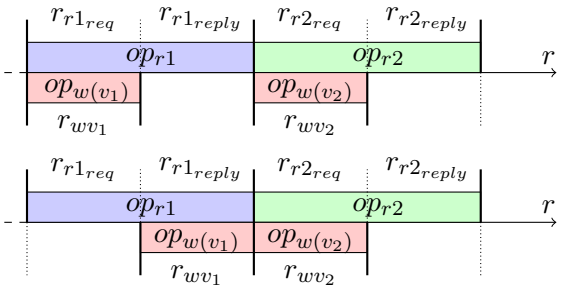


Figure 5.7. Examples of runs showing in details how operations can be aligned given the round-based nature of the system.

Proof It follows directly from Theorems 5, 6 and 7.

$\square_{Theorem\ 8}$

Theorem 9 *Let $\mathcal{A}_{reg}$ be the algorithm in Figures 5.3-5.5 and let $n \geq \alpha f$. If $\alpha = 4$, for each round r $\mathcal{A}_{reg}$ implements a MWMR Atomic register in the Bonnet model.*

Proof It follows directly from Theorems 5, 6 and 7.

$\square_{Theorem\ 9}$

Theorem 10 *Let $\mathcal{A}_{reg}$ be the algorithm in Figures 5.3-5.5 and let $n \geq \alpha f$. If $\alpha = 4$, for each round r $\mathcal{A}_{reg}$ implements a MWMR Atomic register in the Sasaki model.*

Proof It follows directly from Theorems 5, 6 and 7.

$\square_{Theorem\ 10}$

Theorem 11 *Let $\mathcal{A}_{reg}$ be the algorithm in Figures 5.3-5.5 and let $n > \alpha f$. If $\alpha = 2$ then for each round r $\mathcal{A}_{reg}$ implements a MWMR Atomic register in the Burhman model.*

Proof It follows directly from Theorems 5, 6 and 7.

$\square_{Theorem\ 11}$

Concluding remarks

The results found so far can be quickly summarized in Table 5.2. As we can see, with respect to the Consensus problem, are required f fewer servers to solve the Atomic Register problem. This is no true only for the Sasaki's model that requires the same lower bound as the Bonnet's model. Intuitively from the Register point of

MBF model	Atomic Register tight bound	Consensus tight bound
Burhman	$n \geq 2f + 1$	$n \geq 3f + 1$ [3]
Garay	$n \geq 3f + 1$	$n \geq 4f + 1$ [21]
Bonnet	$n \geq 4f + 1$	$n \geq 5f + 1$ [5]
Sasaki	$n \geq 4f + 1$	$n \geq 6f + 1$[40]

Table 5.2. Comparison between lower bounds to solve the Atomic Register and Consensus problems in the round-based MBF models.

view, the fact that a cured server sends different non valid values (Sasaki's model) or the same (Bonnet's model) has not affect on the solution. Roughly speaking, the maintenance() operation act as a Consensus algorithm where all correct servers propose the same value. Concerning the read() operation, it is transparent the fact that a cured server can send the same non valid value or different non valid values since it is interacting only with one client per time.

Chapter 6

Distributed Registers in the Round-free Model

In this Chapter we consider the round-free MBF models. We first state the Safe Register and Regular Register problems and we prove that in an asynchronous system such problems are unsolvable. Then, in the remaining part of this Chapter, only synchronous round-free MBF models are considered. In particular, for each instance of the round-free MBF model, we prove lower bounds and propose optimal solutions with respect to the required number of replicas.

6.1 Register Specification

As define in Section 5.1, a register is a shared variable accessed by a set of processes (i.e., clients) through two operations, namely read() and write().

In this chapter we consider the following register specifications in the round free models.

SWMR Regular Register

A single-writer/multi-reader (SWMR) regular register [23] 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 a *s*afe register (weaker than the regular register in the Lamport's hierarchy [23]). A read operation on a safe register concurrent with a write operation may return any value in the register domain.

SWMR Safe Register

A single-writer/multi-reader (SWMR) safe register [23] 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, if it does not overlap any write() operation, returns the last value written before its invocation (*i.e.*, the value written by the latest completed write() preceding it).

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

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

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

6.2 Impossibilities

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, independently from the system synchrony. 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.

Theorem 12 *Let n be the number of servers emulating a safe register and let f be the number of Mobile Byzantine Agents affecting servers. Let $\mathcal{A}_R$ and $\mathcal{A}_W$ be respectively the algorithms implementing the* read() *and the* write() *operation assuming no communication between servers. If $f > 0$ then there exists no protocol $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ implementing a safe register in any of the MBF models for round-free computations.*

Proof Let us suppose by contradiction that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ is a correct protocol implementing a safe register. If $\mathcal{P}_{reg}$ is correct, it means that both $\mathcal{A}_R$ and $\mathcal{A}_W$ implementing respectively the read() and the write() operation terminates i.e., they stop to execute steps when the operation is completed. Let t be the time at which the last operation *op* terminated and let us assume that no other operation is invoked until time $t' > t$. Let us note that during the time interval $[t, t']$ no algorithm is running as all the operations issued in the past are completed. As a consequence,

no correct server and no cured server will change its state. However, considering that t' does not depend on $\mathcal{P}_{reg}$ (i.e., it is not controlled by the register protocol but it is defined by clients) and considering the mobility of the Mobile Byzantine agents, we may easily have a run where every correct server is faulty and its state can be corrupted at some time in $[t, t']$.

Considering that $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W\}$ and that $\mathcal{A}_R$ and $\mathcal{A}_W$ are not running in $[t, t']$ we can have that every server stores a non valid state at time t' and the register value is lost. As a consequence, $\mathcal{A}_R$ has no way to read a valid value and the validity property is violated. It follows that $\mathcal{P}_{reg}$ is not correct and we have a contradiction. $\square_{Theorem\ 12}$

From Theorem 12 it follows that, in presence of Mobile Byzantine Agents, a new operation must be defined to allow cured servers to restore a valid state and avoid the loss of the register value.

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

As a consequence, any correct protocol $\mathcal{P}_{reg}$ must include one more algorithm implementing the maintenance() operation[1] so that the corollary follows:

Corollary 3 *Let n be the number of servers emulating a register and let f be the number of Mobile Byzantine Agents in the system. That is, if $f > 0$ then any correct protocol $\mathcal{P}_{reg}$ implementing a register in the round-free Mobile Byzantine Failure model must include an algorithm $\mathcal{A}_M$ (i.e., $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$).*

Corollary 4 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in the $(\Delta S, CAM)$ Mobile Byzantine Failure Model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_M$ must involve at least one communication step.*

Proof The claim simply follows by considering that cured servers have a compromised state thus, they need to receive information from correct servers in order to be able to update their state to a valid one. $\square_{Lemma\ 4}$

6.2.1 Impossibilities in Asynchronous System

Lemma 5 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in the $(\Delta S, CAM)$ MBF model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_W$ and $\mathcal{A}_R$ must involve at least one* send − reply *(resp.* request − reply*) communication pattern (i.e., two communication steps).*

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

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

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

In oder to be correct, $\mathcal{A}_W$ must ensure the *termination* property. As a consequence, c_i must be able to decide when it can trigger the write_return event. In particular, this can be done when at least one correct server[2] updated its internal state.

Let us recall that (i) processes communicate only by exchanging messages, (ii) clients (and in particular the writer) do not know the failure state of servers and (iii) the system is asynchronous. As a consequence, the only way c_i has to know that at least one server s_j updated its state is to wait for an acknowledgement from s_j. As a consequence, a second communication step, i.e., a reply step, is necessary for a correct implementation of $\mathcal{A}_W$.

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

$\square_{Lemma\ 5}$

Lemma 6 *Let n be the number of servers emulating a safe register and let f be the number of Byzantine agents in the $(\Delta S, CAM)$ Mobile Byzantine Failure model. Let t be a time instant at which f servers are faulty and f other servers are cured. Let op be a* maintenance() *operation issued at time t. There does not exist a maintenance algorithm $\mathcal{A}_M$ able to terminate in asynchronous settings leaving cured servers with a valid state.*

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

- **Case 1:** $\nexists$ write$(v)||op$. Considering that no write() is concurrent with *op*, the only way s has to come back to be correct is to get the valid value from correct servers. As a consequence, every $\mathcal{A}_M$ must include a communication step where correct servers send their stored value to the cured server s (see Corollary 4). Let us recall that the system is asynchronous; thus it is not possible to bound, a priori, the time needed by such messages to reach s. In addition, s is aware just about its failure state but it is not aware about other failure states (in other words, s cannot know, for any time t, the sets $Co(t)$ and $B(t)$).

  As a consequence, the termination condition of $\mathcal{A}_M$ will depend on messages delivered by s and coming from other servers. Let us recall that cured servers

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

have a non-valid state and, in order to terminate, $\mathcal{A}_M$ must be able to decide a valid value to update the state of the cured server.

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

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

The indistinguishability comes from the following observations:

1. in every time interval $[t_0 + j\Delta, t_0 + (j+1)\Delta]$ (with $j \in \mathbb{N}$) the number of correct servers sending valid values is $n - (j+1)f$. In fact, at any movement, the adversary may decide to move the Byzantine agents on a totally disjoint set of servers (corrupting each time f new servers) until everyone is corrupted. Where t_0 is the initial time where f servers and all the other servers where correct.
2. messages may take an arbitrary time to reach their destination and, in the worst case, all the messages sent in a long time period may be delivered at the same time and not following the FIFO order.
3. when a server is affected by the Byzantine agents, it can send an arbitrary number of messages with an arbitrary content. In particular, given the sequence of messages sent by a server before its compromising, such sequence can be permuted and sent again creating a symmetry condition.

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

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

$\square_{Lemma\ 6}$

Theorem 13 *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.*

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

At time $t_0 + \Delta$, the adversary moves mobile agents and, in the worse case, such agents affect a set of f servers completely disjoint from the previous one. Thus, in the computation we have f servers controlled by the Byzantine agents ($|B(t_0+\Delta)| = f$) , f different servers entering in the cured state ($|Cu(t_0+\Delta)| = f$)) and $n-2f$ correct processes ($|Co(t_0+\Delta)| = n-2f$). Let us recall that, by assumption, cured servers know about their state (see CAM model) and thus they can start executing the maintenance operation running the maintenance algorithm $\mathcal{A}_M$. Each of the cured servers at time $t_0 + \Delta$, s, will start a maintenance() operation. Following Lemma 6 there is no $\mathcal{A}_M$ maintenance algorithm able to terminate leaving s with a valid state under asynchronous communication model. As a consequence, the value of the register is lost and no client is able to return a valid value of the register.

$\square_{Theorem\ 13}$

Note that the above result extends to any register specification and to any MBF instance defined in Chapter 4 since $(\Delta S, CAM)$ is the weakest adversary and safe is the weakest specification.

Lemma 7 *Let $\mathcal{P}_{reg} = \{\mathcal{A}_R, \mathcal{A}_W, \mathcal{A}_M\}$ be a protocol implementing a safe register in the $(\Delta S, CAM)$ Mobile Byzantine Failure Model. Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. Any algorithm $\mathcal{A}_W$ must involve at least one communication step.*

Proof Let op_W be a write operation invoked by client c_i and let v be the value to be written (i.e., to be stored in the register). The claim simply follows by considering that servers need to receive information from client c_i in order to be able to store v.

$\square_{Lemma\ 7}$

Lemma 8 *Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. If $n \leq 3f$ then there exists no algorithm $\mathcal{A}_M$ implementing the* maintenance() *operation in the $(\Delta S, CAM)$ MBF model.*

Proof Let us suppose that $\mathcal{A}_M$ does exists. Let E_1 be an execution for $f = 1$ and $n = 3$. Let us consider the generic time T_i when the mobile agent moves from s_1 to s_2 and $\Delta > 0$ arbitrarily big. At T_i, s_0 is correct and stores v, s_1 is cured and finally s_2 is Byzantine. s_1 is aware to be in a cured state and starts a maintenance() operation. In order to terminate it s_1 needs to get values from other servers, s_1 and s_2, to became correct (cf. Lemma 4). During such operation s_1 gets v from s_0 and $v' \neq v$ from s_2. By hypothesis $\mathcal{A}_M$ does exist, so s_1 become correct storing v. Let us consider another execution E_2, for $f = 1$ and $n = 3$. Let us consider the generic time T_i when the mobile agent moves from s_1 to s_2 and $\Delta > 0$ arbitrarily big. At T_i, s_0 is correct and stores v', s_1 is cured and finally s_2 is Byzantine. As in E_1, s_1 is aware to be in a cured state so it starts the maintenance() operation. In order to terminate it s_1 needs to get values from others server, s_0 and s_2, to become correct (cf. Lemma 4). During such operation s_1 gets v' from s_0 and $v \neq v'$ from s_2. By hypothesis $\mathcal{A}_M$ does exist, so s_1 became correct storing v'. In both execution s_1 terminates the maintenance() operation with different values, but reasoning on the same set of values, both executions are indistinguishable leading to a contradiction.

Considering that all Byzantine agents move in a coordinated way, for a generic $f \geq 1$, is it enough to consider S_0, in place of s_0, as a set of f correct servers, S_1, in place of s_1, as a set of cured servers and finally S_2, in place of s_2 as a set of f Byzantine servers and the result does not change. $\square_{Lemma\ 8}$

Lemma 9 *Let $f > 0$ be the number of Byzantine Agents controlled by the external adversary. If $n \leq 4f$ then there exists no algorithm $\mathcal{A}_M$ implementing the* maintenance() *operation in the $(\Delta S, CUM)$ MBF model.*

Proof Let us suppose that $\mathcal{A}_M$ does exists. From Lemma 4 such algorithm has to involve at least one information exchange among servers. Since those servers are not aware about their state we assume that $\mathcal{A}_M$ is triggered at some point by all servers (not necessarily at the same time). We consider a scenario for $f = 1$ and $n = 4$, $\{s_0, s_1, s_2, s_3\}$. Let T_i be the generic time when the mobile agent moves from s_2 to s_3 and $\Delta > 0$ arbitrarily big. s_2, is not aware to be in a cured state and at some point it triggers the maintenance() operation. We consider two cases, a server invoking $\mathcal{A}_M$ may use or not the value of its internal state.

- Server uses its internal state.
  - Let E_1 be an execution, such that at T_i s_0, s_1 are correct and storing v, s_2 is cured and s_3 is Byzantine. s_2 at some points starts the maintenance() operation and gets v from s_0, s_1 and $v' \neq v$ from s_3 and itself. By hypothesis $\mathcal{A}_M$ does exist, so s_2 became correct and stores v.
  - Let us consider another execution E_2 such that at T_i s_0, s_1 are correct and storing v', s_2 is cured and s_3 is Byzantine. s_2 at some points starts the maintenance() operation and gets v' from s_0, s_1 and $v \neq v'$ from s_3 and itself. By hypothesis $\mathcal{A}_M$ does exist, so s_2 became correct storing v'.

  In both execution s_2 terminates the maintenance() operation with different values, but values collected are the same in both executions, leading to a contradiction.

- Server does not use its internal state.
  - Let E_1 be an execution, such that at T_i s_0, s_1 are correct and storing v, s_2 is cured and s_3 is Byzantine. Correct server s_0 at some points starts the maintenance() operation and gets v from s_1 and $v' \neq v$ from s_2, s_3. By hypothesis $\mathcal{A}_M$ does exist, so s_0 is still correct and stores v (value coming from the correct process).
  - Let us consider another execution E_2 such that at T_i s_0, s_1 are correct and storing v', s_2 is cured and s_3 is Byzantine. s_0 at some points starts the maintenance() operation and gets v' from s_1 and s_3 and $v \neq v'$ from s_2. By hypothesis $\mathcal{A}_M$ does exist, so s_0 is still correct and stores v' (value coming from the correct process).

  In both execution s_0 terminates the maintenance() operation with different values, but values collected are the same in both executions, leading to a contradiction.

It follows that there exists no algorithm $\mathcal{A}_M$ that solves the maintenance in the $(\Delta S, CUM)$ model if $n \leq 4f$. $\square_{Lemma\ 9}$

6.3 Lower Bounds for the Synchronous MBF models

In this section we prove lower bounds with respect to the minimum fraction of correct servers to implement safe registers in presence of mobile Byzantine failures [3]. In particular we first prove lower bounds for the $(\Delta S, CAM)$ and $(\Delta S, CUM)$ models and then we extend those results to all the other models. The first observation that raises is that in presence of mobile agents in the round-free models there are several parameters to take into account with respect to the round-based model. Let us start considering that the set of Byzantine servers changes its composition dynamically time to time. This yields to the following question: does it impact on the read() duration? Or, in other words, such operation has to last as less as possible or until it eventually terminates? In this chapter we consider the read() operation duration as a parameter itself, allowing us to easily verify when the variation of such parameter has any impact on lower bounds. Here below the list of parameters we take into account.

- servers knowledge about their failures state (CAM, CUM);
- the relationship between δ and Δ (that states how many Byzantine servers there may be during an operation);
- T_r, the read() operations duration;
- γ, the upper bound on the time during which a server can be in a cured state (the design of an optimal maintenance() operation is out of the scope of this thesis, thus we use such upper bound as another parameter).

Those parameters allow us to describe different failure models and help us to provide a general framework that produces lower bounds for each specific instance of the MBF models. In the sequel it will be clear that γ varies depending on the coordinated/uncoordinated mobile agents movements $(\Delta S, ITB, ITU)$. In other words, in this parameter is hidden the movements model taken into account, so we do not need to explicitly parametrize it.

Before to start let us precise that we do not consider the following algorithm families: (i) full information algorithm families (processes exchange information at each time instant); (ii) algorithms characterized by a read operation that does not require a request-reply pattern; (iii) algorithms with non quiescent operation (the message exchange triggered by an operation eventually terminates); and finally (iv) algorithms where clients interact with each other. All results presented in the sequel consider a families of algorithms such that previous characteristics do not hold.

The lower bounds proof leverages on the classical construction of two indistinguishable executions. The tricky part is to characterize the set of messages delivered by a client from correct and incorrect servers depending of the read() operation

[3] Results on safe register can be directly extended to the other register specifications.

duration. Let T_r, $T_r \geq 2\delta$ be such duration (according to Lemma 5, each read() operation requires at least a request-reply pattern). We first characterize the correct and incorrect sets of messages, delivered during T_r time, with respect to Δ and γ.

For clarity, in the sequel we note *correct message/request/reply* a message that carries a valid value when it is sent (*i.e.,* sent by a correct process). Otherwise, the message is *incorrect*.

Corollary 3 proves that a protocol $\mathcal{P}_{reg}$ implementing a regular register in a mobile Byzantine setting in addition to the mandatory read() and write() operations must include the additional maintenance operation. Let us recall that such operation when executed by a process p_i, whose internal state is clean from mobile agent effects, terminates at some time t such that p_i has a valid state at time t.

Such operation has a direct impact on the number of correct processes in any time instant. For that reason it is important to characterize its duration, in particular its upper bound in terms of time. The following definition defines γ, the upper bound of the time during which a server can be in a cured state.

Definition 8 (Curing time, γ) *We define γ as the maximum time a server can be in a cured state. More formally, let T_c the time at which server s_c is left by a mobile agent, let op_M the first* maintenance *operation that correctly terminates, then $t_E(op_M) - T_c \leq \gamma$.*

Lemma 10 *There no exist a* maintenance() *operation that correctly terminates in less than δ time.*

Proof The proof follows from Corollary 4 and considering that δ is the upper bound on the message delivery delay. $\square_{Lemma\ 10}$

γ is strictly dependent on the considered Byzantine agent movement model and as we state, the design of an optimal maintenance() operation in the different models is not the scope of such work. For the sake of simplicity in the following depicted figures we consider $\gamma \leq \delta$ in the $(\Delta S, *)$ models and $\gamma \leq 2\delta$ in the others. Intuitively, in the first model the time at which mobile agents move is known, thus the maintenance() operation can start right after agents movement, thus it lasts just the time necessary for messages exchange. In the other models such time it is not known, thus we reasonably assume $\gamma \leq 2\delta$. It holds if servers exchange messages each δ time periods or employ a request-reply pattern.

We define below a scenario of agents movements S^*, with respect to we build the two indistinguishable executions for the lower bounds proof.

Definition 9 (Scenario S^*) *Let S^* be the following scenario: for each time $T_i, i \geq 0$ the affected servers are $s_{(i \mod n)f+1}, \ldots, s_{(i \mod n)f+f}$.*

Figure 6.1 depicts S^*. In particular, the red part is the time where f agents are affecting f servers and the gray part is the time during which servers are in a cured state.

Let us characterize the $\mathcal{P}_{reg}$ protocol in the most general possible way. By definition a register abstraction involves read() and write() operations issued by clients. From Lemma 5, a read operation involves at least a request − reply communication

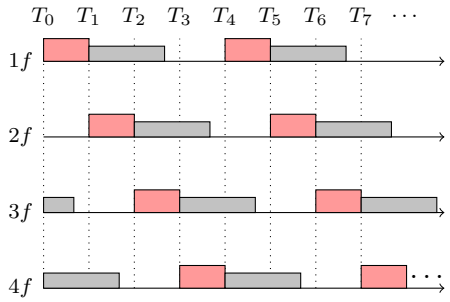


Figure 6.1. Representation of S^* where mobile agents affect groups of f different servers each T_i period. In particular here $\gamma > \Delta$. The gray rectangles represent the time during which servers are in a cured state.

pattern (i.e., two communication steps). Thus, given the system synchrony, a read() operation op_R lasts at least $T_r \geq 2\delta$ time. Moreover we consider that a correct server sends a *reply* message in two occasions: *(i)* after the delivery of a *request* message, and *(ii)* right after it changes its state, at the end of the maintenance operation if an op_R is occurring. The latter case exploits the maintenance operation allowing servers to reply with a valid value in case they were Byzantine at the beginning of the read operation. Moreover we assume that in $(*, CAM)$ model servers in a cured state do not participate to the read operation. Notice that those servers are aware of their current cured state and are aware of their impossibility to send correct replies. Even though those may seems not very general assumptions, let us just consider that we are allowing servers to correctly contribute to the computation as soon as they can and stay silent when they can not and under those assumptions we prove lower bounds. Thus if we remove those assumptions the lower bounds do not decreases. Scenario and protocol has been characterized. Now we aim to characterize the set of servers, regarding their failure states, that can appear during the execution of the protocol, in particular during the read() operation. Those sets allow us to characterize correct and incorrect messages that a client delivers during a read() operation.

Definition 10 (Failure State of servers in a time interval) *Let $[t, t + T_t]$ be a time interval and let t', $t' > 0$, be a time instant. Let s_i be a server and $state_i$ be s_i state, $state_i \in \{correct, cured, Byzantine\}$. Let $S(t')$ be the set of servers s_i that are in the state $state_i$ at t', $S(t') \in \{Co(t'), Cu(t'), B(t')\}$. $\tilde{S}(t, t + T_r)$ is the set of servers that have been in the state $state_i$ for at least one time unit during $[t, t + T_r]$. More formally, $\tilde{S}(t, t + T_r) = \bigcup_{t \leq t' \leq t+T_r} S(t')$.*

For instance, let op_R be a read() operation, $\tilde{B}(t_B(op_R), t_E(op_R))$ is the set containing all servers that have been Byzantine for at least one time unit during op_R. Similarly, $\tilde{Co}(t_B(op_R), t_E(op_R))$ and $\tilde{Cu}(t_B(op_R), t_E(op_R))$ are the sets containing all servers that have been Correct and Cured respectively, for at least one time unit during op_R. In the following, we define two additional sets: (i) the set of servers that during op_R contribute sending to the client both correct and incorrect replies and (ii) the set of servers that during op_R do not reply at all.

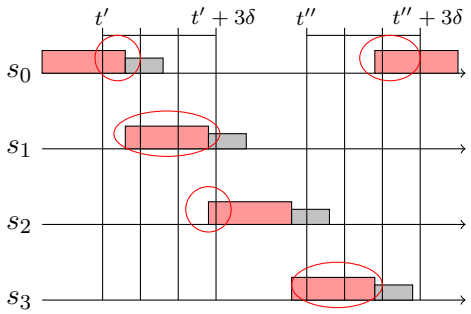


Figure 6.2. Let $[t, t+T_r]$ be time a interval such that in the given scenario $|\tilde{B}(t, t+T_r)| = Max\tilde{B}(t, t+T_r)$. In particular we have that in the time interval $[t', t'+T_r]$, $|\tilde{B}(t', t'+T_r)| = Max\tilde{B}(t, t+T_r)$. While in the time interval $[t'', t''+T_r]$, $|\tilde{B}(t'', t''+T_r)| < Max\tilde{B}(t, t+T_r)$.

Definition 11 ($C\tilde{B}C(t, t+T_r)$) *Let $[t, t+T_r]$ be a time interval, $C\tilde{B}C(t, t+T_r)$ denotes servers that during a time interval $[t, t+T_r]$ belong first to $\tilde{B}(t, t+T_r)$ or $Cu(t)$ (only in $(*, CUM)$ model) and then to $Co(t+\delta, t+T_r-\delta)$ or vice versa. In particular let us denote:*

- *$\tilde{BC}(t, t+T_r)$ servers that during a time interval $[t, t+T_r]$ belong to $\tilde{B}(t, t+T_r)$ or $Cu(t)$ (only in $(*, CUM)$ model) and to $\tilde{Co}(t+\delta, t+T_r-\delta)$.*
- *$\tilde{CB}(t, t+T_r)$ servers that during a time interval $[t, t+T_r]$ belong to $\tilde{Co}(t+\delta, t+T_r-\delta)$ and to $\tilde{B}(t, t+T_r)$.*

Definition 12 ($Sil(t, t+T_r)$) *Let $[t, t+T_r]$ be a time interval. $Sil(t, t+T_r)$ is the set of servers in $Cu(t, t+T_R-\delta)$.*

Servers belonging to $Sil(t_B(op_R), t_E(op_R))$ are servers that do no participate to op_R. In oder words, those servers in the worst case scenario became correct after $t_E(opR) - \delta$, thus if they send back a correct reply it is not sure that client delivers such reply before the end of T_r time. Now we can define the worst case scenarios for the sets we defined so far with respect to S^*.

Definition 13 ($Max\tilde{B}(t, t+T_r)$) *Let S be a scenario and $[t, t+T_r]$ a time interval. The cardinality of $\tilde{B}_S(t, t+T_r)$ is maximum with respect to S if for any t', $t' > 0$, we have that $|\tilde{B}_S(t, t+T_r)| \geq |\tilde{B}_S(t', t'+T_r)|$. Then we call the value of such cardinality as $Max\tilde{B}_S(t, t+T_r)$. If we consider only one scenario per time then we can omit the subscript related to the scenario and write directly $Max\tilde{B}(t, t+T_r)$.*

This value quantifies in the worst case scenario how many servers can be Byzantine, for at least one time unit, during a read() operation. Figure 6.2 depicts a scenario where $T_r = 3\delta$ and during the time interval $[t', t'+T_r]$ there is a maximum number of Byzantine servers while in $[t'', t''+T_r]$ this number is not maximal.

Definition 14 ($MaxSil(t, t+T_r)$) *Let S be a scenario and $[t, t+T_r]$ a time interval. The cardinality of $Sil_S(t, t+T_r)$ is maximum with respect to S if for any t', $t' \geq 0$ we have that $|Sil(t, t+T_r)| \geq |Sil(t', t'+T_r)|$ and $\tilde{B}(t, t+T_r) = Max\tilde{B}(t, t+T_r)$. Then we call the value of such cardinality as $MaxSil_S(t, t+T_r)$. If we consider only*

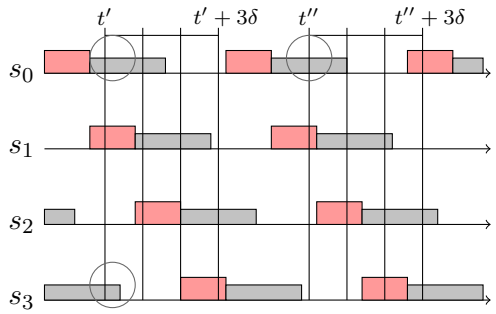


Figure 6.3. Let us consider the time instant t and the depicted scenario such that $|Cu(t)| = MaxCu(t)$. In particular, in this case $|Cu(t')| = MaxCu(t)$ and $|Cu(t'')| < MaxCu(t)$.

one scenario per time then we can omit the subscript related to the scenario and write directly $minSil(t, t + T_r)$.

This value quantifies the maximum number of servers that begin in a cured state a read() operation and are still cured after $T_r - \delta$ time. So that any correct reply sent after such period has no guarantees to be delivered by the client and such servers are assumed to be silent.

Definition 15 ($MaxCu(t)$) *Let S be a scenario and t be a time instant. The cardinality of $Cu_S(t)$ is maximum with respect to S if for any t', $t' \geq 0$, we have that $|Cu_S(t')| \leq |Cu_S(t)|$ and $\tilde{B}(t, t + T_r) = Max\tilde{B}(t, t + T_r)$. We call the value of such cardinality as $MaxCu_S(t)$. If we consider only one scenario per time then we can omit the subscript related to it and write directly $MaxCu(t)$.*

This value quantifies, in the worst case scenario, how many cured servers there may be at the beginning of a read() operation. Figure 6.3 depicts a scenario where at time t' there are the maximum number of cured server while at t'' this value is not maximum. Notice that in such figure, in case of a shorter time interval $[t', t' + 2\delta]$ s_0 would be silent.

Definition 16 ($min\tilde{C}o(t, t + T_r)$) *Let S be a scenario and $[t, t + T_r]$ be a time interval then $min\tilde{C}_S(t, t+T_r)$ denotes the minimum number of correct servers during a time interval $[t + \delta, t + T_r - \delta]$. If we consider only one scenario per time then we can omit the subscript related to it and write directly $min\tilde{C}(t, t + T_r)$.*

Notice that we are not interested in servers that are always correct during the read() operation op_R, but in servers that surely can reply. A *reply* sent before $t_E(op_R) - \delta$ is for sure delivered by client.

Figure 6.4 depicts a scenario where during the both intervals $[t', t' + T_r]$ and $[t'', t'' + T_r]$ the number of correct servers is minimum.

Definition 17 ($minC\tilde{B}C(t, t + T_r)$) *Let $[t, t+T_r]$ be a time interval then $minC\tilde{B}C(t, t+T_r)$ denotes the minimum number of servers that during a time interval $[t, t + T_r]$ belong first to $\tilde{B}(t, t + T_r)$ or $Cu(t)$ (only in (ITB, CUM) model) and then to $Co(t + \delta, t + T_r - \delta)$ or vice versa and $\tilde{B}(t, t + T_r) = Max\tilde{B}(t, t + T_r)$.*
In particular let us denote as:

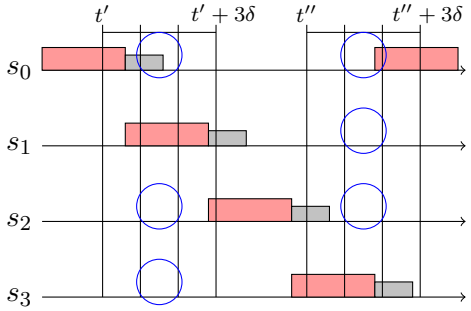


Figure 6.4. Let $[t, t+T_r]$ be a time interval such that in the depicted scenario $|\tilde{Co}(t, t+T_r)| = min\tilde{Co}(t, t+T_r)$. Then in both time intervals $[t', t'+T_r]$ and $[t'', t''+T_r]$ we have that $|\tilde{Co}(t', t'+T_r)| = |\tilde{Co}(t'', t''+T_r)| = min\tilde{Co}(t, t+T_r)$.

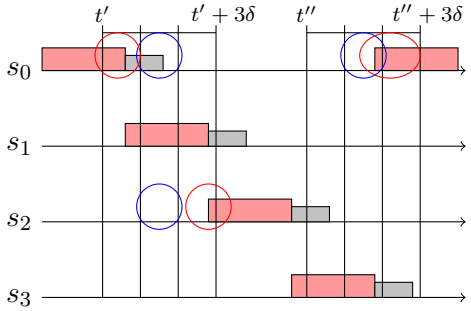


Figure 6.5. Let $[t, t+T_r]$ a time interval such that in the depicted scenario $C\tilde{B}C(t, t+T_r) = minC\tilde{B}C(t, t+T_r)$. Then $C\tilde{B}C(t', t'+T_r) > minC\tilde{B}C(t, t+T_r)$ and $C\tilde{B}C(t'', t''+T_r) = minC\tilde{B}C(t, t+T_r)$.

- $min\tilde{BC}(t, t+T_r)$ *the minimum number of servers that during a time interval* $[t, t+T_r]$ *belong to* $\tilde{B}(t, t+T_r)$ *or* $Cu(t)$ *(only in* (ITB, CUM) *model) and to* $\tilde{Co}(t+\delta, t+T_r-\delta)$.
- $min\tilde{CB}(t, t+T_r)$ *the minimum number of servers that during a time interval* $[t, t+T_r]$ *belong to* $\tilde{Co}(t+\delta, t+T_r-\delta)$ *and to* $\tilde{B}(t, t+T_r)$.

As we stated before, Byzantine servers set changes during the read() operation op_R, so there can be servers that are in a Byzantine state at $t_B(op_R)$ and in a correct state before $t_E(op_R) - \delta$ (cf. s_0 during $[t', t'+3\delta]$ time interval in Figure 6.5). Those servers contribute with an incorrect message at the beginning and with a correct message after. The same may happen with servers that are correct from $t_B(op_R)$ to at least $t_B(op_R) + \delta$ (so that for sure deliver the read request message and send the reply back) and are affected by a mobile agent after $t_B(op_R) + \delta$ (cf. s_0 during $[t'', t''+3\delta]$ time interval in Figure 6.5).

Lemma 11 $Max\tilde{B}(t, t+T_r) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f$.

Proof For simplicity let us consider a single agent ma_k, then we extend the same reasoning to all the f agents. In $[t, t+T_r]$ time interval, with $T_r \geq 2\delta$, ma_k can affect a different server each Δ time. It follows that the number of times it may change

server is $\frac{T_r}{\Delta}$. Thus the affected servers are $\lceil \frac{T_r}{\Delta} \rceil$ plus the server that was affected at t. Finally, extending the reasoning to f agents, $Max\tilde{B}(t, t+T_r) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f$, which concludes the proof. $\square_{Lemma\ 11}$

As we see in the sequel, the value of $Max\tilde{B}(t, t+T_r)$ is enough to compute the lower bound. Now we can define the worst case scenario for a read() operation with respect to S^*. Let *op* be a read operation issued by c_i. We want to define, among the messages that can be deliver by c_i during *op*, the minimum amount of messages sent by server when they are in a correct state and the maximum amount of messages sent by servers when they are not in a correct state.
In each scenario, we assume that each message sent to or by Byzantine servers is instantaneously delivered, while each message sent to or by correct servers requires δ time. Without loss of generality, let us assume that all Byzantine servers send the same value and send it only once, for each period where they are Byzantine. Moreover, we make the assumption that each cured server (in the CAM model) does not reply as long as it is cured. Yet, in the CUM model, it behaves similarly to Byzantine servers, with the same assumptions on message delivery time.

Definition 18 (*$MaxReplies\_NCo(t, t+T_r)_k$*) *Let $MaxReplies\_NCo(t, t+T_r)_k$ be the multi-set maintained by client c_k containing m_{ij} elements, where m_{ij} is the $i-th$ message delivered by c_k and sent at time $t', t' \in [t, t+T_r]$ by s_j such that $s_j \notin Co(t')$.*

Considering the definitions of both $Max\tilde{B}(t, t+T_r)$ and $MaxCu(t)$ the next Corollary follows:

Corollary 5 *In the worst case scenario, during a* read *operation lasting $T_r \geq 2\delta$ issued by client c_i, c_i delivers $Max\tilde{B}(t, t+T_r)$ incorrect replies in the $(*, CAM)$ model and $Max\tilde{B}(t, t+T_r) + MaxCu(t)$ incorrect replies in the $(*, CUM)$ model .*

Definition 19 (*$minReplies\_Co(t, t+T_r)_k$*) *Let $minReplies\_Co(t, t+T_r)_k$ be the multi-set maintained by client c_k containing m_{ij} elements, where m_{ij} is the $i-th$ message delivered by c_k and sent at time $t', t' \in [t, t+T_r]$ by s_j such that $s_j \in Co(t')$.*

Note that correct replies come from servers that *(i)* have never been affected during the time interval $[t, t+T_r]$, or *(ii)* where in a cured state at t but do not belong to the $Sil(t, t+T_r)$ set, or *(iii)* servers that reply both correctly and incorrectly. The next Corollary follows.

Corollary 6 *In the worst case scenario, during a* read *operation lasting $T_r \geq 2\delta$ issued by client c_i, c_i delivers $n-(Max\tilde{B}(t, t+T_r)+MaxSil(t, t+T_r))+minC\tilde{B}C(t, t+T_r)$ correct replies in the $(\Delta S, CAM)$ model and $n-[Max\tilde{B}(t, t+T_r)+MaxCu(t)]+minC\tilde{B}C(t, t+T_r)$ correct replies in the $(\Delta S, CUM)$ model.*

In the following, given a time interval, we characterize correct and incorrect servers involved in such interval. Concerning correct servers, let us first analyze when a client collects $x \leq n$ different replies and then we extend such result to $x > n$. Then we do the same for incorrect replies.

Lemma 12 *Let op be a* read *operation issued by client c_i in a scenario S^*, whose duration is $T_r \geq 2\delta$. Let $x, x \geq 2$, be the number of messages delivered by c_i during op. If $x \leq n$ then $minReplies\_Co(t, t+T_r)_k$ contains replies from x different servers.*

Proof Let us suppose that $minReplies\_Co(t, t+T_r)_k$ contains replies from $x-1$ different servers (trivially it can not be greater than x). Without lost of generality, let us suppose that c_i collects replies from $s_1, \ldots, s_{x-1}$. It follows that there is a server $s_i, i \in [1, x-1]$ that replied twice and a server s_x that did not replied. Let us also suppose *w.l.g.* that there is one Byzantine mobile agent ma_k (i.e., $f = 1$). If during the time interval $[t, t+T_r]$ s_x never replied, then s_x has been affected at least during $[t+\delta, t+T_r-\delta-\gamma+1]$. This implies that $T_r \leq \Delta + 2\delta + \gamma$. Since s_i replies twice then two scenarios are possible during *op*: *(i)* s_i was first affected by ma_k and then became correct (so it replied once), then affected again and then correct again (so it replied twice); *(ii)* s_i was correct (so it replied once), then it was affected by ma_k and then correct again (so it replied twice). Let us consider case *(ii)* (case *(i)* follows trivially). Since s_i had the time to reply (δ), to be affected and then became correct ($\Delta + \gamma$) and reply again (δ) this means that $T_r > \Delta + 2\delta + \gamma$. A similar result we get in case *(i)* where the considered execution requires a longer time. This is in contradiction with $T_r \leq \Delta + 2\delta + \gamma$ thus c_i gets replies for x different servers.

$\square_{Lemma\ 12}$

If a client delivers $n > x$ messages then we can apply the same reasoning of the previous Lemma to the first chunk of n messages, then to the second chunk of n messages and so on. Roughly speaking, if $n = 5$ and a client delivers 11 messages from correct processes, then there are 3 occurrences of the message coming from the first server and 2 occurrences of the messages coming from the remaining servers. Thus the next Corollary directly follows.

Corollary 7 *Let op be a* read *operation issued by client c_i in a scenario S^*, op duration is $T_r \geq 2\delta$. Let $x, x \geq 2$, be the number of messages delivered by c_i during op, then $minReplies\_Co(t, t+T_r)_k$ contains $x \mod n$ messages m_{ij} whose occurrences is $\lfloor \frac{x}{n} \rfloor + 1$ and $(n - x \ (mod\ n))$ messages whose occurrences is $\lfloor \frac{x}{n} \rfloor$.*

The case of $MaxReplies\_NCo(t, t+T_r)_k$ directly follows from scenario S^*, since by hypotheses mobile Byzantine agents move circularly from servers to servers, never passing on the same server before having affected all the others. Thus, the following corollary holds.

Corollary 8 *Let op be a* read *operation issued by client c_i in a scenario S^*, op duration is $T_r \geq 2\delta$. Let $x, x \geq 2$, be the number of messages delivered by c_i during op, then $MaxReplies\_NCo(t, t+T_r)_k$ contains $x \mod n$ messages m_{ij} whose occurrences is $\lfloor \frac{x}{n} \rfloor + 1$ and $(n - x \ (mod\ n))$ messages whose occurrences is $\lfloor \frac{x}{n} \rfloor$.*

At this point we can compute how many correct and incorrect replies a client c_k can deliver in the worst case scenario during a time interval $[t, t+T_r]$. Trivially, c_k in order to distinguish correct and incorrect replies needs to get $minReplies\_Co(t, t+T_r)_k > MaxReplies\_NCo(t, t+T_r)_k$. It follows that the number of correct servers

$n_{CAM_{LB}}$	$[2Max\tilde{B}(t,t+T_r) + MaxSil(t,t+T_r) - minC\tilde{B}C(t,t+T_r)]f$
$n_{CUM_{LB}}$	$[2(Max\tilde{B}(t,t+T_r) + MaxCu(t,t+T_r)) - minC\tilde{B}C(t,t+T_r)]f$

Table 6.1. How to compute the number of replicas in each model.

has to be enough to guarantee this condition. Table 6.1 follows directly from this observation. In a model with b Byzantine (non mobile) a client c_i requires to get at least $2b+1$ replies to break the symmetry and thus $n \geq 2b+1$. In presence of mobile Byzantine we have to sum also servers that do not reply (silent) and do not count twice servers that reply with both incorrect and correct values.

Theorem 14 *If $n < n_{CAM_{LB}}$ ($n < n_{CUM_{LB}}$) as defined in Table 6.1, then there not exists a protocol $\mathcal{P}_{reg}$ solving the safe register specification in $(\Delta S, CAM)$ model ($(\Delta S, CUM)$ model respectively).*

Proof Let us suppose that $n < n_{CAM_{LB}}$ ($n < n_{CUM_{LB}}$) and that protocol $\mathcal{P}_{reg}$ does exist. If a client c_i invokes a read operation op, lasting $T_r \geq 2\delta$ time, if no write operations occur, then c_i returns a valid value at time $t_B(op)$. Let us consider an execution E_0 where c_i invokes a read operation op and let 0 be the valid value at $t_B(op)$. Let us assume that all Byzantine severs involved in such operation reply once with 1. From Corollaries 5 and 6, c_i collects $MaxReplies\_NCo(t,t+T_r)_i$ occurrences of 1 and $minReplies\_Co(t,t+T_r)_i$ occurrences of 0. Since $\mathcal{P}_{reg}$ exists and no write operations occur, then c_i returns 0. Let us now consider a another execution E_1 where c_i invokes a read operation op and let 1 be the valid value at $t_B(op)$. Let us assume that all Byzantine severs involved in such operation replies once with 0. From Corollaries 5 and 6 and Corollary 7 and Corollary 8, c_i collects $MaxReplies\_NCo(t,t+T_r)_i$ occurrences of 0 and $minReplies\_Co(t,t+T_r)_i$ occurrences of 1. Since $\mathcal{P}_{reg}$ exists and no write operations occur, then c_i returns 1.

From Lemmas 11 and using values in Table 6.1 we obtain following equations for both models:

- $(\Delta S, CAM)$:
  - $MaxReplies\_NCo(t,t+T_r)_i = Max\tilde{B}(t,t+T_r) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f$
  - $minReplies\_Co(t,t+T_r)_i = n - [Max\tilde{B}(t,t+T_r) + MaxSil(t,t+T_r)] + minC\tilde{B}C(t,t+T_r) =$

$$[2(Max\tilde{B}(t,t+T_r)) + MaxSil(t,t+T_r) - minC\tilde{B}C(t,t+T_r)] +$$
$$-[(Max\tilde{B}(t,t+T_r) + MaxSil(t,t+T_r)) + minC\tilde{B}C(t,t+T_r)] =$$
$$Max\tilde{B}(t,t+T_r) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f$$

- $(\Delta S, CUM)$:
  - $MaxReplies\_NCo(t,t+T_r)_i = Max\tilde{B}(t,t+T_r) + MaxCu(t) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f + MaxCu(t)$
  - $minReplies\_Co(t,t+T_r)_i = n - [Max\tilde{B}(t,t+T_r) + MaxCu(t)] + minC\tilde{B}C(t,t+T_r) =$

$$[2Max\tilde{B}(t,t+T_r) + 2MaxCu(t)) - minC\tilde{B}C(t,t+T_r)]+$$
$$-[Max\tilde{B}(t,t+T_r) + MaxCu(t)] + minC\tilde{B}C(t,t+T_r) =$$
$$Max\tilde{B}(t,t+T_r) + MaxCu(t) = (\lceil \frac{T_r}{\Delta} \rceil + 1)f + MaxCu(t)$$

It follows that in E_0 and E_1 c_i delivers the same occurrences of 0 and 1, both executions are indistinguishable leading to a contradiction.

$\square_{Theorem\ 14}$

$MaxReplies\_NCo(t,t+T_r)_i$ and $minReplies\_Co(t,t+T_r)_i$ are equal independently from the value assumed by T_r, the read() operation duration. From the equation just used in the previous lemma the next Corollary follows.

Corollary 9 *For each $T_r \geq 2\delta$ if $n > n_{CAM_{LB}}$ ($n > n_{CUM_{LB}}$) then $MaxReplies\_NCo(t,t+T_r)_i < minReplies\_Co(t,t+T_r)_i$.*

At this point we compute $minCu(t)$, $MaxSil(t,t+T_r)$ and $minC\tilde{B}C(t,t+T_r)$ to finally state exact lower bounds depending on the system parameters, in particular depending on Δ, γ and the servers awareness, i.e., $(*,CAM)$ and $(*,CUM)$.

Let us adopt the following notation. Given the time interval $[t,t+T_r]$ let $\{s_1,s_2,\ldots,s_b\} \in B(t,t+T_r)$ be the servers affected sequentially during T_r by the mobile agent ma_k. Let $\{s_{-1},s_{-2},\ldots,s_{-c}\} \in Cu(t)$ be the servers in a cured state at time t such that s_{-1} is the last server that entered in such state and s_c the first server that became cured. Let $t_BB(s_i)$ and $t_EB(s_i)$ be respectively the time instant in which s_i become Byzantine and the time in which the Byzantine agent left. $t_BCu(s_i)$ and $t_ECu(s_i)$ are respectively the time instant in which s_i become cured and the time instant in which it became correct. Considering that ma_k moves each Δ time then we have that $t_BB(s_{i-1}) - t_BB(s_i) = \Delta$ and $t_BCu(s_{-j}) - t_BCu(s_{-j+1}) = \Delta$. The same holds for the t_E of such states. Moreover $t_BB(s_1) = t_BCu(s_{-1})$. Now we are ready to build the read scenario with respect to S^*. In particular we build a scenario for the $(\Delta S,CAM)$ model and one for the $(\Delta S,CUM)$ model. Intuitively, the presence of cured servers do not have the same impact in the two models, thus in the $(\Delta S,CUM)$ model we maximize such number. Let $[t,t+2\delta]$ be the considered time interval and let ϵ be a positive number arbitrarily smaller, then we consider in the $(\Delta S,CAM)$ scenarios $t = t_EB(s_1) - \epsilon$ (cf. Figure 6.6) and in the $(\Delta S,CUM)$ scenarios $t_BB(s_b) = t + 2\delta - \epsilon$ (cf. Figure 6.7).

In the sequel we use the notion of Ramp Function:

$$\mathcal{R}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Lemma 13 *Let us consider a time interval $[t,t+T_r], T_r \geq 2\delta$ and an arbitrarily small number $\epsilon > 0$, then in fthe $(\Delta S,CAM)$ model $MaxCu(t) = \mathcal{R}(\lceil \frac{\gamma-\Delta+\epsilon}{\Delta} \rceil)$.*

Proof As we defined, s_{-1} is the most recent server that entered in a cured state, with respect to the considered time interval. Intuitively each s_{-j} is in $Cu(t)$ if $t_ECu(s_{-j}) > t$. Considering that $t_ECu(s_{-j}) - t_ECu(s_{-j-1}) = \Delta$ then the number

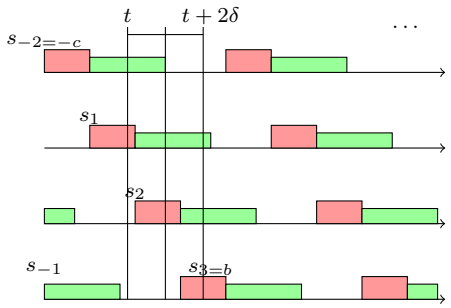


Figure 6.6. Representation of S^* when we consider a $(\Delta S, CAM)$ model, in particular $t_E B(s_1) = t + \epsilon$, for $\epsilon > 0$ and arbitrarily small.

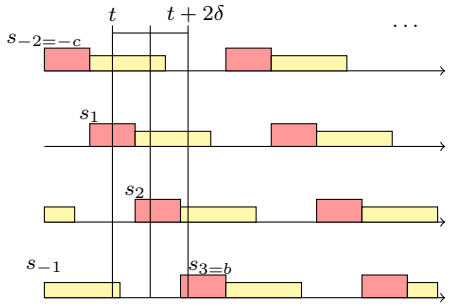


Figure 6.7. Representation of S^* when we consider a $(\Delta S, CUM)$ model, in particular $t_B B(s_c) = t + 2\delta - \epsilon$, for $\epsilon > 0$ and arbitrarily small.

of servers in a cured state at t is $MaxCu(t) = \lceil \frac{t_E Cu(s_1) - t}{\Delta} \rceil$. [4] As we stated, for $(*, CAM)$ models we consider scenarios in which t, the beginning of the considered time interval, is just before $t_E B(s_1)$. Thus given an arbitrarily small number $\epsilon > 0$, let $t = t_E B(s_1) - \epsilon$. By construction we know that $t_B B(s_1) = t_E B(s_1) - \Delta = t_B Cu(s_{-1})$. Substituting $t_B Cu(s_{-1}) = t + \epsilon - \Delta$, since we consider γ the upper bound for the curing time, then $t_E Cu(s_{-1}) = t + \epsilon - \Delta + \gamma$. So finally, $MaxCu(t) = \lceil \frac{t_E Cu(s_1) - t}{\Delta} \rceil = \lceil \frac{\gamma - \Delta + \epsilon}{\Delta} \rceil$ and since there can no be a negative result then $MaxCu(t) = \mathcal{R}(\lceil \frac{\gamma - \Delta + \epsilon}{\Delta} \rceil)$. This concludes the proof. $\square_{Lemma\ 13}$

Lemma 14 *Let us consider a time interval* $[t, t+T_r], T_r \geq 2\delta$ *and an arbitrarily small number* $\epsilon > 0$*, then in the* $(\Delta S, CUM)$ *model* $MaxCu(t) = \mathcal{R}(\lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil \Delta + \gamma}{\Delta} \rceil)$.

Proof As we defined, s_{-1} is the most recent server that entered in a cured state, with respect to the considered interval. Intuitively, s_{-j} is in $Cu(t)$ if $t_E Cu(s_{-j}) > t$. Considering that $t_E Cu(s_{-j}) - t_E Cu(s_{-j-1}) = \Delta$ then the number of servers in a cured state at t is $MaxCu(t) = \lceil \frac{t_E Cu(s_1) - t}{\Delta} \rceil$. As we state, for $(*, CUM)$ models we consider scenarios in which the end of the considered time interval, is just after $t_B B(s_b)$. Thus given an arbitrarily small number $\epsilon > 0$, let $t_B B(s_b) = t + T_r - \epsilon$. By construction we know that $t_B B(s_1) = t_E B(s_1) - \Delta = t_B Cu(s_{-1})$ and

[4] Consider Figure 6.6, s_2 is the most recent server that entered in the cured state. This is the server that spend more time in such state with respect to the others. It follows that other servers are in a cured state if during this time interval there is enough time for a "jump"

$t_B B(s_1) = t_B B(s_b) - \lceil \frac{T_r}{\Delta} \rceil \Delta$ (cf. Lemma 11). Substituting and considering that $t_E Cu(s_{-1}) = t_B Cu(s_{-1}) + \gamma$) we get the following: $t_E Cu(s_{-1}) = t + T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil + \gamma$. Finally $MaxCu(t) = \lceil \frac{t_E Cu(s_1) - t}{\Delta} \rceil = \lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil + \gamma}{\Delta} \rceil$ and since there can not be a negative result then $MaxCu(t) = \mathcal{R}(\lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil \Delta + \gamma}{\Delta} \rceil)$. This concludes the proof. $\square_{Lemma\ 14}$

Lemma 15 *Let us consider a time interval* $[t, t+T_r], T_r \geq 2\delta$ *and an arbitrarily small number* $\epsilon > 0$*, then in the* $(\Delta S, CAM)$ *model* $MaxSil(t, t+T_r) = \mathcal{R}(\lceil \frac{\gamma - \Delta + \epsilon - T_r + \delta}{\Delta} \rceil)$.

Proof As we defined, s_{-1} is the most recent server that entered in a cured state, with respect to the considered interval. Intuitively, s_{-j} is in $Sil(t, t+2\delta)$ if $t_E Cu(s_{-j}) > T_r - \delta$. Considering that $t_E Cu(s_{-j}) - t_E Cu(s_{-j-1}) = \Delta$ then the number of servers in a silent state at t is $MaxSil(t, t+2\delta) = \lceil \frac{t_E Cu(s_1) - T_r + \delta}{\Delta} \rceil$. As we stated for $(\Delta S, CAM)$ models we consider scenarios in which t, the beginning of the considered time interval, is just before $t_E B(s_1)$. Thus given an arbitrarily small number $\epsilon > 0$, let $t = t_E B(s_1) - \epsilon$. By construction we know that $t_B B(s_1) = t_E B(s_1) - \Delta = t_B Cu(s_{-1})$. Substituting $t_B Cu(s_{-1}) = t + \epsilon - \Delta$, since we consider γ the upper bound for curing time, then $t_E Cu(s_{-1}) = t + \epsilon - \Delta + \gamma$. So finally, $MaxSil(t, t+T_r) = \lceil \frac{t_E Cu(s_1) - T_r + \delta}{\Delta} \rceil = \lceil \frac{\gamma - \Delta + \epsilon - T_r + \delta}{\Delta} \rceil$, then since there can not be a negative result $MaxSil(t, t+2\delta) = \mathcal{R}(\lceil \frac{\gamma - \Delta + \epsilon - T_r + \delta}{\Delta} \rceil)$. $\square_{Lemma\ 15}$

Lemma 16 *Let us consider a time interval* $[t, t+T_r], T_r \geq 2\delta$ *and an arbitrarily small number* $\epsilon > 0$*, then in the* $(\Delta S, CUM)$ *model* $MaxSil(t, t+T_r) = \lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil \Delta + \gamma - \delta}{\Delta} \rceil$.

Proof As we defined, s_{-1} is the most recent server that entered in a cured state, with respect to the considered interval. Intuitively, s_{-j} is in $Sil(t, t+T_r)$ if $t_E Cu(s_{-j}) > T_r - \delta$. Considering that $t_E Cu(s_{-j}) - t_E Cu(s_{-j-1}) = \Delta$ then the number of servers in a silent state at t is $MaxSil(t, t+T_r) = \lceil \frac{t_E Cu(s_1) - T_r + \delta}{\Delta} \rceil$. As we stated for $(\Delta S, CUM)$ models we consider scenarios in which $t + T_r$, the end of the considered time interval, is just after $t_B B(s_b)$. Thus given an arbitrarily small number $\epsilon > 0$, let $t_B B(s_b) = t + T_r - \epsilon$. By construction we know that $t_B B(s_1) = t_E B(s_1) - \Delta = t_B Cu(s_{-1})$ and $t_B B(s_1) = t_B B(s_b) - \lceil \frac{T_r}{\Delta} \rceil \Delta$ (cf. Lemma 11). Substituting and considering that $t_E Cu(s_{-1} = t_B Cu(s_{-1}) + \gamma)$ we get the following: $t_E Cu(s_{-1} = t + T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil + \gamma$. Finally $MaxSil(t, t+T_r) = \lceil \frac{t_E Cu(s_1) - T_r + \delta}{\Delta} \rceil = \lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil + \gamma - T_r + \delta}{\Delta} \rceil$, then since there can not be a negative result, $MaxSil(t, t+T_r) = \lceil \frac{T_r - \epsilon - \lceil \frac{T_r}{\Delta} \rceil \Delta + \gamma - T_r + \delta}{\Delta} \rceil$. $\square_{Lemma\ 16}$

Lemma 17 *Let us consider a time interval* $[t, t+T_r], T_r \geq 2\delta$ *then in the* $(\Delta S, CAM)$ *model.* $minC\tilde{B}C = \mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\delta}{\Delta} \rceil) + \mathcal{R}(\lceil \frac{T_r - \gamma - T_r + \delta}{\Delta} \rceil)$.

Proof By definition $minC\tilde{B}C(t, t+T_r) = min\tilde{CB}(t, t+T_r) + min\tilde{BC}(t, t+T_r)$.
- $min\tilde{CB}(t, t+T_r)$ is the minimum number of servers that correctly reply and then, before $t + T_r$ are affected and incorrectly reply. Let us observe that a correct server correctly reply if belongs to $Co(t, t+\delta)$, it follows that servers in $\tilde{B}(t, t+\delta)$ do not

correctly reply. Thus, $\tilde{minCB}(t, t+T_r) = \tilde{MaxB}(t, t+T_r) - \tilde{MaxB}(t, t+\delta)$. It may happen that $\tilde{MaxB}(t, t+T_r) < \tilde{MaxB}(t, t+T_r-\delta)$, but obviously there can no be negative servers, so we consider only non negative values, $\tilde{minCB}(t, t+T_r) = \mathcal{R}(\tilde{MaxB}(t, t+T_r) - \tilde{MaxB}(t, t+\delta))$.

- $\tilde{minBC}(t, t+2\delta)$ is the minimum number of servers that incorrectly reply and then become correct in time that the correct reply is delivered. A server is able to correctly reply if it is correct before $t+T_r-\delta$ (the reply message needs at most δ time to be delivered). Thus we are interested in servers that are affected by a mobile agent up to $t+T_r-\gamma-\delta$. For (Δ, CAM) models we consider scenarios in which t, the beginning of the considered time interval, is just before $t_EB(s_1)$. Thus given an arbitrarily small number $\epsilon > 0$, let $t = t_EB(s_1) - \epsilon$. In the time interval $[t, t+T_r-\gamma-\delta]$ the number of the mobile agent "jumps" is given by $\lceil \frac{T_r-\gamma-\delta}{\Delta} \rceil$ Trivially, we can not have a negative number, so it becomes $\mathcal{R}(\lceil \frac{T_r-\gamma-\delta}{\Delta} \rceil)$. Summing up $\tilde{minCBC} = \mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\delta}{\Delta} \rceil) + \mathcal{R}(\lceil \frac{T_r-\gamma-\delta}{\Delta} \rceil)$, which concludes the proof. $\square_{Lemma\ 17}$

Lemma 18 *Let us consider a time interval $[t, t+T_r], T_r \geq 2\delta$, let $\epsilon > 0$ be an arbitrarily small number. If $maxCu(t) > 0$ or $\gamma > \Delta$ then in the $(\Delta S, CUM)$ model $\tilde{minCB} = \lceil \frac{T_r-\epsilon-\delta}{\Delta} \rceil$ otherwise $\tilde{minCB} = \mathcal{R}(\tilde{MaxB}(t, t+T_r) - \tilde{MaxB}(t, t+T_r-\delta))$.*

Proof $\tilde{minCB}(t, t+T_r)$ is the minimum number of servers that correctly reply and then, before $t+T_r$ are affected by a mobile agent and incorrectly reply. We are interested in the maximum number of Byzantine servers in $B(t, t+T_r-\delta)$, so that the remaining ones belong to $B(t+T_r-\delta, t+T_r)$, which means that servers in $B(t+T_r-\delta, t+T_r)$ are in $Co(t, t+\delta)$ (considering the scenario S^*). Thus, considering that in the (Δ, CUM) model we consider $t_BB(s_b) = t+T_r-\epsilon$ ($\epsilon > 0$ and arbitrarily small) then we consider the maximum number of "jumps" there could be in the time interval $[t+\delta, t+T_r-\epsilon]$. Thus $\tilde{minCB}(t, t+T_r) = \lceil \frac{t+T_r-\epsilon-t-\delta}{\Delta} \rceil = \lceil \frac{T_r-\epsilon-\delta}{\Delta} \rceil$. If $MaxCu(t) = 0$ or $\gamma > \Delta$ then it has no sense to consider the $(\Delta S, CUM)$ worst case scenario that aims to maximize cured servers. Thus in this case we consider the $(\Delta S, CAM)$ worst case scenario, $\tilde{minCB} = \mathcal{R}(\tilde{MaxB}(t, t+T_r) - \tilde{MaxB}(t, t+T_r-\delta))$, concluding the proof. $\square_{Lemma\ 18}$

Lemma 19 *Let us consider a time interval $[t, t+T_r], T_r \geq 2\delta$ then in the $(\Delta S, CUM)$ model then if $maxCu(t) > 0$ $\tilde{minCBC} = \lceil \frac{T_r-\epsilon-\delta}{\Delta} \rceil + \mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\gamma-\delta}{\Delta} \rceil) + (MaxCu(t) - MaxSil(t, t+T_r))$, otherwise $\tilde{minCBC}$ assumes the same values as in the $(\Delta S, CAM)$ case.*

Proof By definition $\tilde{minCBC}(t, t+T_r) = \tilde{minCB}(t, t+T_r) + \tilde{minBC}(t, t+T_r)$. From Lemma 18, if $maxCu(t) > 0$ or $\Delta > \gamma$ then in the $(\Delta S, CUM)$ model $\tilde{minCB} = \lceil \frac{T_r-\epsilon-\delta}{\Delta} \rceil$ otherwise $\tilde{minCB} = \mathcal{R}(\tilde{MaxB}(t, t+T_r) - \tilde{MaxB}(t, t+T_r-\delta))$. $\tilde{minBC}(t, t+T_r)$ is the minimum number of servers that incorrectly reply and then, before $t+T_r-\delta$ become correct so that are able to correctly reply in time such that their reply is delivered. In the $(\Delta S, CUM)$ model servers may incorrectly reply because affect by a mobile agent or because in a cured state. In the first case, a

Table 6.2. Values for a general read() operation that terminates after T_r time.

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

	$minC\tilde{B}C(t, t+T_r)$
$(\Delta S, CAM)$	$\mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\delta}{\Delta} \rceil) + \mathcal{R}(\lceil \frac{T_r - \gamma - \delta}{\Delta} \rceil)$
$(\Delta S, CUM)$	$\lceil \frac{T_r - \epsilon - \delta}{\Delta} \rceil^5 + \mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\gamma + \delta}{\Delta} \rceil) + (MaxCu(t) - MaxSil(t, t+T_r))$

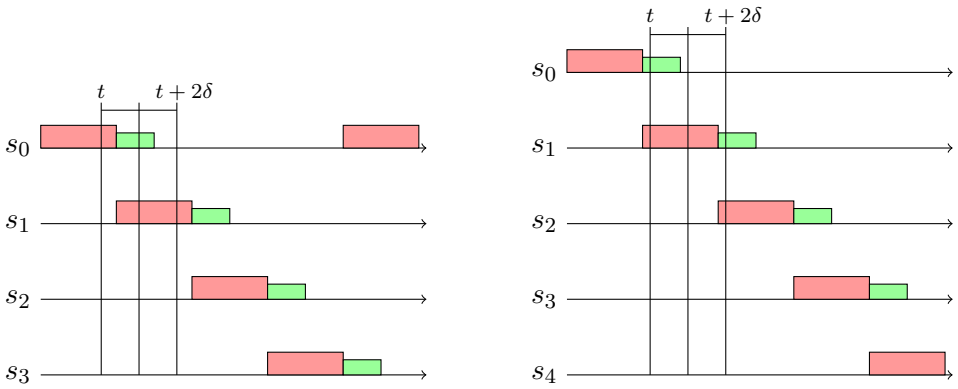
server is able to correctly reply if it become correct before $t + T_r - \delta$ (the reply message needs at most δ time to be delivered). Thus we consider the maximum number of servers that can be affected in the period $t + T_r - \gamma - \delta, t + T_r$, which is $\lceil \frac{\gamma+\delta}{\Delta} \rceil$. Thus, among the Byzantine servers (i.e., $Max\tilde{B}(t, t+T)$) we consider servers not affected in the time interval $[t + T_r - \gamma + \delta, t + T_r]$. In other words such servers have γ time to became correct and δ time to reply before the end of the operation. Thus $Max\tilde{B}(t, t+T_r) - Max(t+T_r-\gamma+\delta, t+T_r)$. Again we can not have a negative number, so it becomes $\mathcal{R}(\lceil \frac{T_r}{\Delta} - \frac{\gamma-\delta}{\Delta} \rceil)$. Concerning servers that incorrectly reply when in a cured state, we are interested in servers that correctly reply after in time such that the reply is delivered by the client, i.e., they are not silent. This number is easily computable, $MaxCu(t) - MaxSil(t, t + T_r)$. Thus $min\tilde{BC}(t, t + 2\delta) = (MaxCu(t) - MaxSil(t, t + T_r))$. Summing up if $maxCu(t) > 0$ or $\Delta > \gamma$, then $minC\tilde{B}C = \lceil \frac{T_r-\epsilon-\delta}{\Delta} \rceil + \mathcal{R}(\lceil \frac{T_r}{\Delta} \rceil - \lceil \frac{\gamma-\delta}{\Delta} \rceil) + (MaxCu(t) - MaxSil(t, t+2\delta))$, otherwise $minC\tilde{B}C$ assumes the same values as in the $(\Delta S, CAM)$ model, which concludes the proof. $\square_{Lemma\ 19}$

In Table 6.2 are reported all the results found so far for $(\Delta S, *)$ models.

Such results have been proved considering $f = 1$. Extending such results to scenario for $f > 1$ is straightforward in the $(\Delta S, *)$ model. The extension to $f > 1$ in the $(ITB, *)$ and $(ITU, *)$ models is less direct. What is left to prove is that the results found for $f = 1$ can be applied to all other models in which mobile agents move independently from each other. In the following Lemma we employ $*$ to indicate that the result holds for $*$ assuming consistently the value CAM or CUM.

Lemma 20 *Let $n_{*_{LB}} \leq \alpha_*(\Delta, \delta, \gamma)f$ be the impossibility result holding in the $(\Delta S, *)$ model for $f = 1$. If there exists a tight protocol $\mathcal{P}_{reg}$ solving the safe register for $n \geq \alpha_*(\Delta, \delta, \gamma)f + 1$ ($f \geq 1$) then all the Safe Register impossibility results that hold in the $(\Delta S, *)$ models hold also in the $(ITB, *)$ and $(ITU, *)$ models.*

Proof Let us consider the scenario S^* for $f = 1$ and a read() operation time interval $[t, t + T_r]$, $t \geq 0$. Depending on the value of t there can be different (but finite) read scenarios, $rs_1, rs_2, \ldots, rs_s$. By hypothesis there exists $\mathcal{P}_{reg}$ solving the safe register for $n \geq \alpha_* f(\Delta, \delta, \gamma) + 1$ then among the read scenarios $\mathcal{RS} = \{rs_1, rs_2, \ldots, rs_s\}$ all the possible worst case scenarios $\{wrs_1, \ldots, wrs_w\} \subseteq \mathcal{RS}$ hold for $n = \alpha_*(\Delta, \delta, \gamma)f$ (meaning that $\mathcal{P}_{reg}$ does not exist). We can say that those worst scenarios are equivalent in terms of replicas, i.e., for each wsr_k is it possible to build an impossibility run if $n = \alpha_*(\Delta, \delta, \gamma)$ but $\mathcal{P}_{reg}$ works if $n = \alpha_*(\Delta, \delta, \gamma)+1$ (if we consider $f = 1$). Let us now consider $(\Delta S, *)$ for $f > 1$. In this case, mobile agents move all together, thus



(a) $(\Delta S, CAM)$ scenario. **(b)** $(\Delta S, CUM)$ scenario.

Figure 6.8. $(\Delta S, CAM)$ and $(\Delta S, CUM)$ scenarios considering $2\delta \leq \Delta < 3\delta$.

the same wrs_k scenario is reproduced f times. For each wrs_k scenario is it possible to build an impossibility run if $n = \alpha_*(\Delta, \delta, \gamma)f$, i.e., $\alpha_*(\Delta, \delta, \gamma) - 1$ non Byzantine servers are not enough to cope with 1 Byzantine server, then it is straightforward that $\alpha_*(\Delta, \delta, \gamma) - f$ non Byzantine servers are not enough to cope with f Byzantine servers, the same scenario is reproduced f times.
In the case of unsynchronized movements (ITB and ITU) we consider $\Delta = min \{\Delta_1, \ldots, \Delta_f\}$. Each mobile agent generates a different read scenarios, those scenario can be up to f. As we just stated, if $\mathcal{P}_{reg}$ exists, those worst case scenarios are equivalent each others in terms of replicas. Since all the worst case scenarios are equivalent in terms of replicas, thus impossibility results holding for mobile agents moving together hold also for mobile agent moving in an uncoordinated way.

$\square_{Lemma\ 20}$

6.3.1 Examples

In this last part we see with some example how to compute the lower bounds for some particular case.

If Figure 6.8 we consider $2\delta \leq \Delta < 3\delta$. In case (a) we consider the $(\Delta S, CAM)$ model. In such case we have $Max\tilde{B}(t, t+2\delta) = 2$ (cf. Lemma 11), $MaxCu(t) = 0$ (cf. Lemma 13), thus $MaxSil(t, t + 2\delta) = 0$ as well (cf. Lemma 15) and $minC\tilde{B}C = 0$ (cf. Lemma 17). In particular, s_0 and s_1 incorrectly reply, contrarily to s_2 and s_3. Thus, considering the reasoning in Theorem 14 we have that for $n = 4$ we can build two indistinguishable executions. In case (b) we consider the $(\Delta S, CUM)$ model. In such case we have $Max\tilde{B}(t, t + 2\delta) = 2$ (cf. Lemma 11), $MaxCu(t) = 1$ (cf. Lemma 14), $MaxSil(t, t + 2\delta) = 0$ (cf. Lemma 16) and $minC\tilde{B}C = 1$ (cf. Lemma 19). s_0, s_1 and s_2 incorrectly reply, contrarily to s_3 and s_4. Moreover s_2 is correct before $t + \delta$, thus reply correctly as well.Thus, considering the reasoning in Theorem 14 we have that for $n = 5$ we can build two indistinguishable executions.

If Figure 6.9 we consider $\delta \leq \Delta < 2\delta$. In case (a) we consider the $(\Delta S, CAM)$ model. In such case we have $Max\tilde{B}(t, t+2\delta) = 3$ (cf. Lemma 11), $MaxCu(t) = 1$ (cf. Lemma 13), $MaxSil(t, t + 2\delta) = 0$ (cf. Lemma 15) and $minC\tilde{B}C = 1$ (cf. Lemma

(a) $(\Delta S, CAM)$ scenario. **(b)** $(\Delta S, CUM)$ scenario.

Figure 6.9. $(\Delta S, CAM)$ and $(\Delta S, CUM)$ scenarios considering $\delta \leq \Delta < 2\delta$.

17). In particular, s_0, s_1 and s_3 incorrectly reply, contrarily to s_3 and s_4. Moreover s_2 is correct before $t+\delta$, thus reply correctly as well. Thus, considering the reasoning in Theorem 14 we have that for $n = 5$ we can build two indistinguishable executions. In case (b) we consider the $(\Delta S, CUM)$ model. In such case we have $Max\tilde{B}(t, t+2\delta) = 3$ (cf. Lemma 11), $MaxCu(t) = 1$ (cf. Lemma 14), $MaxSil(t, t + 2\delta) = 0$ (cf. Lemma 16) and $minC\tilde{B}C = 2$ (cf. Lemma 19). s_1, s_2, s_3 and s_4 incorrectly reply, contrarily to s_0 and s_5. Moreover s_4 is correct before $t + \delta$ and s_1 complete the maintenance() operation before $t + \delta$, thus both reply correctly as well. Thus, considering the reasoning in Theorem 14 we have that for $n = 6f$ we can build two indistinguishable executions.

When we consider the model in which mobile agents are free to move we consider $\gamma \leq 2\delta$. If Figure 6.10 we consider $2\delta \leq \Delta < 3\delta$ and $f = 2$. In such case we have $Max\tilde{B}(t, t + 2\delta) = 2f = 4$ (cf. Lemma 11), $MaxCu(t) = 1f = 2$ (cf. Lemma 13),$MaxSil(t, t+2\delta) = 0$ (cf. Lemma 15) and $minC\tilde{B}C = 0$ (cf. Lemma 17). Thus, considering the reasoning in Theorem 14 we have that for $n = 4f$ we can build two indistinguishable executions. In this case we can see that the two mobile agents generates two different read scenarios, but those scenarios are equivalent in terms of replies. Let ma_0 the mobile agent on s_0 at time t and Let ma_1 be the mobile agent on s_4 at time t. ma_0 generates the following scenario: s_0 and s_1 incorrectly reply, contrarily to s_2 and s_7 (that starts the maintenance() when s_0 becomes Byzantine). On the other side, ma_1 generates the following read scenario: s_4 and s_5 incorrectly replies, s_3 is silent, but s_5 correctly replies before to be affected and s_6 correctly replies as well.

In the remaining part of the Chapter we propose optimal solutions to solve the Regular Register problem in the hierarchy of models we proposed (cf. Figure 4.8). Notice that all those solutions are optimal with respect to the γ deriving from the specific maintenance() operation employed, but we do not always have clues about the optimality of such operation. Before to proceed, is it worthy to discuss the relationship between the protocol structure and the MBF model considered. As we

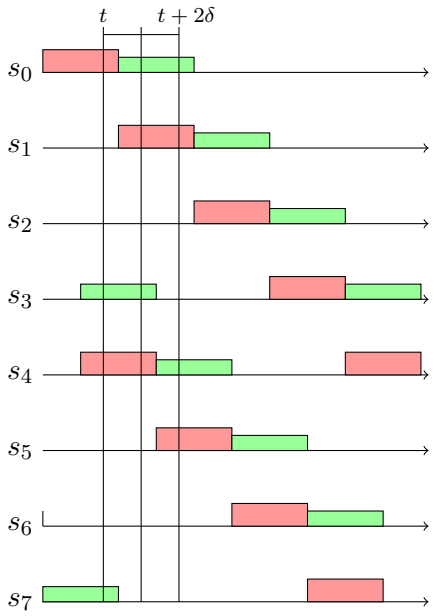


Figure 6.10. (ITB, CAM) scenario for $2\delta \leq \Delta < 3\delta$ and $f = 2$.

will see, the read() and write() operations slightly change from a model to the next. Informally speaking, such operations need to interact with a large enough fraction of correct servers, the dimension of such fraction depends on δ, Δ and γ. What really changes, from a model to the next, is the maintenance() operation. Indeed such operation copes with mobile agent movements and allows cured servers to become correct as soon as possible. In the next sections we will see how such operation has to change with respect to the MBF failure model considered and consequently the impact of γ in the protocols upper bounds we define.

6.4 Upper Bounds for the $(\Delta S, CAM)$ Synchronous model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ with respect to the number of replicas, that implements a SWMR Regular Register in a round-free synchronous system for $(\Delta S, CAM)$ instance of the proposed MBF model. Our solution is based on the following three key points: (1) we implement a maintenance() operation that is executed periodically at each $T_i = t_0 + i\Delta$ time (the time at which mobile agents move is known). 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 dependent by 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. Notice that when we say that a message is lost we are referring to the following situation: a client send a message at time t, thus it is delivered by all servers in a non Byzantine state in the time interval $[t, t + \delta]$. As

$k = \lceil \frac{2\delta}{\Delta} \rceil$	$n_{CAM} \geq (k+3)f + 1$	$\#reply_{CAM} \geq (k+1)f + 1$	$\#echo_{CAM}$	T_r	d
$k = 1$	$4f + 1$	$2f + 1$	$2f + 1$	2δ	3
$k = 2$	$5f + 1$	$3f + 1$	$2f + 1$	2δ	3

Table 6.3. Parameters for $\mathcal{P}_{Rreg}$ Protocol in the $(\Delta S, CAM)$ model for $\delta \leq \Delta < 3\delta$.

a consequence, servers in $\tilde{B}(t, t+\delta)$ may deliver such message when affected by a mobile agent, so that, after the mobile agent move to another server there is not trace of the delivered message. Thus we say that such message is lost.
Protocol $\mathcal{P}_{reg}$ is presented in details for $\delta \leq \Delta < 3\delta$ (6.4.1). Then we present slight modifications to apply to algorithms in both cases $\Delta < \delta$ (6.4.2) and $\Delta > 3\delta$ (6.4.3). Finally are presented parametrized joint proofs for those three cases (6.4.4). For simplicity we consider Δ as a multiple of δ or vice versa when $\delta > \Delta$ and we use $k = \lceil \frac{2\delta}{\Delta} \rceil$ as a parameter. Roughly speaking k represents how many mobile agent "jumps" there may be during a 2δ temporal window and $\frac{k}{2}$ how many "jumps" there can be in a δ time period.

6.4.1 $\mathcal{P}_{reg}$ Detailed Description for $\delta \leq \Delta < 3\delta$

The protocol $\mathcal{P}_{reg}$ for the $(\Delta S, CAM)$ model is described in Figures 6.11 - 6.13, which present the maintenance(), write(), and read() operations respectively. Parameters for such protocol are reported in Table 6.3 respect to $k = \lceil \frac{2\delta}{\Delta} \rceil$. n_{CAM} is the minimum number of required replicas, $\#reply_{CAM}$ is the minimum number of expected reply messages carrying the same value from $\#reply_{CAM}$ different servers and $\#echo_{CAM}$ is the minimum number of echo messages carrying the same value from $\#echo_{CAM}$ different servers. T_r is the read() operation duration and d is the number of values each server stores.

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. In particular v is the value, sn is the associated sequence number and j is the identifier of server s_j that sent the reply back. 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 monotonically.

Server side is more complex than client side. Servers, besides to store values and provide them when required, have also to manage the maintenance operation. As we proved in Theorem 12, such operation is necessary to cope with Byzantine movements to do not lose the last written values in the register.
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 d 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 v_k with sequence number sn_k in V_i according to the incremental order

with respect to the sequence numbers. If there are more than d values, it discards from V_i the value associated to the lowest sequence number.

- $pending\_read_i$: set variable used to collect identifiers of the clients that are currently reading.
- $cured_i$: boolean flag updated by the cured_state oracle. In particular, such variable is set to true when s_i becomes aware of its cured state and it is reset during the algorithm when s_i becomes correct.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through 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.
- $just\_correct_i$: boolean flag used to prevent servers, that were cured during the previous maintenance() operation, to reset the auxiliary variables ($echo\_vals_i$, $echo\_read_i$ and fw_vals_i listed before). In particular this variable is set to true just after s_i becomes correct and it is reset during at the end of the next maintenance() operation.

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

- select_d_pairs_max_sn($echo\_vals_i$): this function takes as input the set $echo\_vals_i$, selects all the tuples $\langle v, sn \rangle$ whose occurrence in $echo\_vals_i$ is at least $\#echo_{CAM} = 2f + 1$ and among those returns the d tuples (if exist) 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 the pair $\langle v, sn \rangle$ occurring at least $\#reply_{CAM}$ times. If there are more pairs satisfying such condition, it returns the pair containing the highest sequence number.

The maintenance() operation. Such operation is executed by servers periodically at any time instant $T_i = t_0 + i\Delta$. In the $(*, CAM)$ models servers knows when a mobile agent leaves them, thus depending on such knowledge they execute different actions. In particular, if a server s_i is not in a cured state then it broadcasts an ECHO message carrying V_i and $pending\_read_i$ sets. Moreover if s_i is not in a $just\_correct_i$ case, it empties fw_vals_i and $echo\_vals_i$ sets, meaning that there is not need to retrieve lost values because s_i was not recently affected by a mobile agent.

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_d_pairs_max_sn($echo\_vals_i$) function that populates V_i with up to d tuples $\langle v, sn \rangle$. If there are less than d tuples $\langle v, sn \rangle$, it means that there are concurrent

```
operation maintenance() executed every T_i = t_0 + iΔ :
(1)  cured_i ← report_cured_state();
(2)  if (cured_i) then
(3)     V_i ← ∅; echo_vals_i ← ∅; echo_read_i ← ∅; fw_vals_i ← ∅;
(4)     wait(δ);
(5)     insert(V_i, select_d_pairs_max_sn(echo_vals_i));
(6)     cured_i ← false;
(7)     just_correct_i ← true;
(8)     for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(9)                 send REPLY (i, V_i) to c_j;
(10)    endFor
(11)    else
(12)        broadcast ECHO(i, V_i, pending_read_i);
(13)        if ¬(just_correct_i)then
(14)            fw_vals_i ← ∅; echo_vals_i ← ∅;
(15)            else just_correct_i ← false;
(16)        endif
(17) endif
-------------------------------------------------------
when ECHO (j, V_j, pr) is received:
(18)  echo_vals_i ← echo_vals_i ∪ V_j;
(19)  echo_read_i ← echo_read_i ∪ pr;
```

Figure 6.11. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CAM)$ model for $\delta \leq \Delta < 3\delta$.

write() operations updating the register value concurrently with the maintenance() operation. For the moment, s_i considers $\langle \perp, 0 \rangle$ as the pair associated to the value that is concurrently written. At the end s_i assigns $false$ to $cured_i$ variable, meaning that it is now correct (has valid value to reply with) and can start to reply to clients that are currently reading and assigns $true$ to $just\_correct_i$ variable, to avoid to empty the auxiliary variables during the next maintenance() operation.

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 message, it updates V_i invoking the function INSERT() and forwards the message, through a WRITE_FW$(i, \langle v, csn \rangle)$, to all others servers. This helps to cope with 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 $pending\_read_i$ set) 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 of just cured state to store the new value and reply to a reading client as soon as possible.

The read() operation. When a client wants to read, it broadcasts a READ() message 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

```
========= Client code ==========
operation write(v):
(1)  csn ← csn + 1;
(2)  broadcast WRITE(v, csn);
(3)  wait (δ);
(4)  return write_confirmation;

========= Server code ==========
when WRITE(v, csn) is received:
(5)    insert(V_i, ⟨v, csn⟩);
(6)    for each j ∈ (pending_read_i ∪ echo_read_i) do
(7)        send REPLY (i, {⟨v, csn⟩});
(8)    endFor
(9)    broadcast WRITE_FW(i, ⟨v, csn⟩);
-------------------------------------------------
when WRITE_FW(j, ⟨v, csn⟩) is received:
(10)   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:
(11)   insert(V_i, ⟨v, sn⟩);
(12)   ∀j : fw_vals_i ← fw_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(13)   ∀j : echo_vals_i ← echo_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(14)   for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(15)           send REPLY (i, {⟨v, sn⟩}) to c_j;
(16)   endFor
```

Figure 6.12. $\mathcal{A}_W$ algorithm implementing the write(v) operation in the $(\Delta S, CAM)$ model for $\delta \leq \Delta < 3\delta$.

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 the client identifier in $pending\_read_i$ set 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 up to d tuples $\langle value, ts \rangle$.

As we said earlier, V_i may contains less than d values if s_i was affected by a Byzantine agent when the last write() operations occurred. As soon as s_i retrieve such values through the fw_vals_i and $echo\_vals_i$ sets, such values are 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.

6.4.2 $\mathcal{P}_{reg}$ for $\Delta < \delta$

When $\Delta < \delta$ the previous protocol changes with respect to the maintenance() operation. In this case, during such operation operation, mobile Byzantine agent movements may occur. Informally the maintenance() operation could be run at each T_i, but since $T_{i+1} - T_i < \delta$, then a new maintenance() would be started before the

```
========= Client code ==========
operation read():
(1)  reply_i ← ∅;
(2)  broadcast READ(i);
(3)  wait (T_r);
(4)  ⟨v, sn⟩ ← select_value(reply_i);
(5)  broadcast READ_ACK(i);
(6)  return v;
---------------------------------------
when REPLY (j, V_j) is received:
(7)    for each (⟨v, sn⟩ ∈ V_j) do
(8)              reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(9)    endFor


========= Server code ==========
when READ (j) is received:
(10)  pending_read_i ← pending_read_i ∪ {j};
(11)  if (¬cured_i)
(12)     then send REPLY (i, V_i);
(13)  endif
(14)  broadcast READ_FW(j);
---------------------------------------
when READ_FW (j) is received:
(15)  pending_read_i ← pending_read_i ∪ {j};
---------------------------------------
when READ_ACK (j) is received:
(16)  pending_read_i ← pending_read_i \ {j};
(17)  echo_read_i ← echo_read_i \ {j};
```

Figure 6.13. $\mathcal{A}_R$ algorithm implementing the read() operation in the $(\Delta S, CAM)$ model for $\delta \leq \Delta < 3\delta$.

$\Delta \leq \delta$	$n_{CAM} \geq (k + \frac{k}{2} + 2)f + 1$	$\#reply_{CAM} \geq (k+1)f + 1$	$\#echo_{CAM} \geq kf + 1$	T_r	d
$\frac{\delta}{2} \leq \Delta < \delta$	$8f + 1$	$5f + 1$	$4f + 1$	2δ	3

Table 6.4. Parameters for $\mathcal{P}_{Rreg}$ Protocol in the $(\Delta S, CAM)$ model for $\Delta < \delta$.

previous one terminate, so that the operation never terminate. This means that it no possible to run a new maintenance() at each T_i. To overcome this problem we introduce a new variable $curing\_state_i$ that is used in place of $cured_i$ in the following way. When $cured_i$ is set to TRUE, then the maintenance() operation starts. To avoid the run of a new operation too early (at the next T_i), $cured_i$ is set to FALSE and $curing\_state_i$ to TRUE. So that $curing\_state_i$ is used during the read() operation to avoid cured servers to reply when are storing any valid value. Variable $counter_i$ takes the place of $just\_cured_i$. Basically, after the end of the maintenance() operation, some value just written could be still missing, thus $echo\_vals_i$ and fw_vals_i can not be emptied, in this case, for δ time after the end of the operation. $counter_i$ keeps track of the number of maintenance() operations that occur during δ time. In Figure 6.14 and Figure 6.15 are reported, with slight modifications, the maintenance() and read() operations respectively (write() operation is unchanged). In Table 6.4 are listed the new parametrized values. In particular, in the second line there is an example for $\frac{\delta}{2} \leq \Delta < \delta$. As we can see the number of replicas increases to cope with the number of Byzantine servers that increases during a δ time period.

```
operation maintenance() executed every T_i = t_0 + iΔ :
(1)  cured_i ← report_cured_state();
(2)  if (cured_i) then
(3)     V_i ← ∅; echo_vals_i ← ∅; echo_read_i ← ∅; fw_vals_i ← ∅;
(4)     curing_state_i ← true;
(5)     cured_i ← false;
(6)     wait(δ);
(7)     insert(V_i, select_d_pairs_max_sn(echo_vals_i));
(8)     curing_state_i ← false;
(9)     counter_i ← k/2;
(10)    for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(11)            send REPLY (i, V_i) to c_j;
(12)    endFor
(13)    elseIf ¬(curing_state_i)
(14)          broadcast ECHO(i, V_i, pending_read_i);
(15)          if counter_i = 0 then
(16)             fw_vals_i ← ∅; echo_vals_i ← ∅;
(17)             else counter_i ← counter_i − 1;
(18)          endif
(19) endif
---------------------------------------------------------------
when ECHO (j, V_j, pr) is received:
(20)  echo_vals_i ← echo_vals_i ∪ V_j;
(21)  echo_read_i ← echo_read_i ∪ pr;
```

Figure 6.14. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CAM)$ model for $\Delta < \delta$.

```
========== Client code ==========
operation read():
(1)  reply_i ← ∅;
(2)  broadcast READ(i);
(3)  wait (2δ);
(4)  ⟨v, sn⟩ ← select_value(reply_i);
(5)  broadcast READ_ACK(i);
(6)  return v;
---------------------------------------------------------------
when REPLY (j, V_j) is received:
(7)     for each (⟨v, sn⟩ ∈ V_j) do
(8)             reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(9)     endFor


========== Server code ==========
when READ (j) is received:
(10)  pending_read_i ← pending_read_i ∪ {j};
(11)  if (¬curing_state_i)
(12)     then send REPLY (i, V_i);
(13)  endif
(14)  broadcast READ_FW(j);
---------------------------------------------------------------
when READ_FW (j) is received:
(15)  pending_read_i ← pending_read_i ∪ {j};
---------------------------------------------------------------
when READ_ACK (j) is received:
(16)  pending_read_i ← pending_read_i \ {j};
(17)  echo_read_i ← echo_read_i \ {j};
```

Figure 6.15. $\mathcal{A}_R$ algorithm implementing the read() operation in the $(\Delta S, CAM)$ model for $\Delta < \delta$.

$\Delta \geq 3\delta$	n_{CAM}	$\#reply_{CAM} \geq (k+1)f+1$	$\#echo_{CAM}$	T_r	d
$k=1$	$3f+1$	$2f+1$	$2f+1$	3δ	4

Table 6.5. Parameters for $\mathcal{P}_{Rreg}$ Protocol in the $(\Delta S, CAM)$ model for $\Delta \geq 3\delta$.

6.4.3 $\mathcal{P}_{reg}$ for $\Delta \geq 3\delta$

For $\Delta \geq 3\delta$ the protocol slightly changes with respect to the $\delta \leq \Delta < 3\delta$ protocol (cf. 6.4.1), in particular the main changes are in the maintenance() operation. The idea is the following, with respect to 6.4.1, n_{CAM} is composed by f fewer correct servers but a longer read() operation is employed. This gives to servers in a cured state during a read() operation the time to become correct and contribute to such operation. To cope with a fewer number of correct servers, in particular in the forwarding mechanism and maintenance() operation, messages coming from cured servers are ignored, which implies to wait the maitenance() termination (if a server does not send an ECHO() message then is faulty or cured) to take any decision. In Figure 6.16 and Figure 6.17 the maintenance() and the write() operation respectively. The read() operation does not change, the only difference is the different values assumed by T_r that according to Table 6.5 is 3δ.

For each server s_i there are the following changes:

- V_i dimension is $d = 4$ rather than $d = 3$;
- during the maintenance() operation fw_vals_i are purged from values coming from cured servers using the following function;
  - delete_cured_values(echo_vals$_i$, set$_i$): this function takes as input $echo\_vals_i$ and set_i set and removes from the latter all values coming from servers that omit to send the ECHO() message.
- at this point in $fw\_vals_i \cup echo\_vals_i$ there can be at most f values coming from Byzantine servers, then a value is chosen from such set if occurs at least $f + 1 = \#echo_{CAM}$ times and no decision are taken when the server is still in a cured state, meaning that the delete_cured_values(echo_vals$_i$, fw_vals$_i$) and delete_cured_values(echo_vals$_i$, echo_vals$_i$) functions have not been invoked yet.
- there is no more need of $just\_cured_i$ variable. This variable was used to avoid to delete the $fw\_vals_i \cup echo\_vals_i$ sets after the end of the maintenance() operation in case, before the beginning of such operation, a write() operation occurred. To allows the cured server to get the value the forwarding mechanism takes place. In this case, being $\Delta \geq 3\delta$ the new maintenance() operation do no begin before the end of the forwarding mechanism and thus there is no more need to keep those values during more than one maintenance() operation.

For each client c_i the only change concern the read() operation, that terminates after 3δ time rather than 2δ time.

```
operation maintenance() executed every T_i = t_0 + iΔ :
(1)  cured_i ← report_cured_state();
(2)  if (cured_i) then
(3)     V_i ← ∅; echo_vals_i ← ∅; echo_read_i ← ∅; fw_vals_i ← ∅;
(4)     wait(δ);
(5)     delete_cured_values(echo_vals_i, echo_vals_i);
(6)     insert(V_i, select_d_pairs_max_sn(echo_vals_i));
(7)     delete_cured_values(echo_vals_i, fw_vals_i);
(8)     cured_i ← false;
(9)     for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(10)            send REPLY (i, V_i) to c_j;
(11)    endFor
(12)    else
(13)        broadcast ECHO(i, V_i, pending_read_i);
(14) endif
-----------------------------------------------------------
when ECHO (j, V_j, pr) is received:
(15)  echo_vals_i ← echo_vals_i ∪ V_j;
(16)  echo_read_i ← echo_read_i ∪ pr;
```

Figure 6.16. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CAM)$ model for $\Delta \geq 3\delta$.

6.4.4 Correctness $(\Delta S, CAM)$

Proofs for $\mathcal{P}_{Rreg}$ protocol are similar for all the three cases presented. Termination property is guaranteed by the way the code is designed, after a fixed period of time all operations terminate. Validity property is proved with the following steps:

- 1. maintenance() operation works (i.e., at the end of the operation $n-f$ servers store valid values). In particular, for a given value v stored by $\#echo$ correct servers at the beginning of the maintenance() operation, there are $n-f$ servers that may store v at the end of the operation;
- 2. given a write() operation that writes v at time t and terminates at time $t+\delta$, there is a time $t' > t+\delta$ after which $\#reply$ correct servers store v.
- 3. at the next maintenance() operation after t' there are $\#reply - f = \#echo$ correct servers that store v, for step (1) this value is maintained.
- 4. the validity follows considering that the read() operation is long enough to include the t' of the last written value before the read() and V is big enough to do not be full filled with new values before t'.

Those steps are used along all the chapter for all the algorithms we present.

We now show that the termination property is satisfied i.e, that read() and write() operations terminates. Due to the algorithm implementation, such property is independent from the specific instance of the MBF model considered.
Notice, from now on we refer to the protocol code lines in 6.4.1, for the others instances ($\delta > \Delta$ 6.4.2 and $\Delta \geq 3\delta$ 6.4.3) we point out code lines when it is necessary.

Lemma 21 *If a correct client c_i invokes* write(v) *operation at time t then this operation terminates at time $t+\delta$.*

```
========= Client code ==========
operation write(v):
(1)  csn ← csn + 1;
(2)  broadcast WRITE(v, csn);
(3)  wait (δ);
(4)  return write_confirmation;

========= Server code ==========
when WRITE(v, csn) is received:
(5)    insert(V_i, ⟨v, csn⟩);
(6)    for each j ∈ (pending_read_i ∪ echo_read_i) do
(7)        send REPLY (i, {⟨v, csn⟩});
(8)    endFor
(9)    broadcast WRITE_FW(i, ⟨v, csn⟩);
-------------------------------------------------------------
when WRITE_FW(j, ⟨v, csn⟩) is received:
(10)   fw_vals_i ← fw_vals_i ∪ {⟨j, ⟨v, csn⟩⟩};
-------------------------------------------------------------
when ∃⟨j, ⟨v, sn⟩⟩ ∈ (fw_vals_i ∪ echo_vals_i) occurring at least #echo_CAM times ∧¬(cured_i):
(11)   insert(V_i, ⟨v, sn⟩);
(12)   ∀j : fw_vals_i ← fw_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(13)   ∀j : echo_vals_i ← echo_vals_i \ {⟨j, ⟨v, ts⟩⟩};
(14)   for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(15)           send REPLY (i, {⟨v, sn⟩}) to c_j;
(16)   endFor
```

Figure 6.17. $\mathcal{A}_W$ algorithm implementing the write(v) operation in the $(\Delta S, CAM)$ model for $\Delta \geq 3\delta$.

Proof The claim follows by considering that a write_confirmation event is returned to the writer client c_i after δ time, independently of the behavior of the servers (see lines 3-4, Figure 6.12). $\square_{Lemma\ 21}$

Lemma 22 *If a correct client c_i invokes* read() *operation at time t then this operation terminates at time $t + T_r$.*

Proof The claim follows by considering that a read() returns a value to the client after 2δ time, independently of the behavior of the servers (see lines 12-15, Figure 6.13). $\square_{Theorem\ 22}$

Theorem 15 (Termination) *If a correct client c_i invokes an operation, c_i returns from that operation in finite time.*

Proof The proof follows from Lemma 21 and Lemma 22. $\square_{Theorem\ 15}$

Notice, in the following, when not explicitly defined, we always consider as hypothesis that n_{CAM} follows values defined in Tables 6.3, 6.4 and 6.5.

Lemma 23 (Step 1) *Let v be the value stored at $\#echo_{CAM}$ correct servers $s_j \in Co(T_i)$, such that $v \in V_j \forall s_j \in Co(T_i)$. Then $\forall s_c \in Cu(T_i)$ v is returned by the function* select_d_pairs_max_sn($echo\_vals_i$) *at $T_i + \delta$ (i.e., at the end of the* maintenance() *operation).*

Proof By hypotheses at T_i there are $\#echo_{CAM}$ correct servers s_j storing the same value v in V_j and running the correct code. Non faulty servers run code in Figure 6.11. In particular each correct server broadcasts a ECHO() message with attached the content of V_j (line 11) while each server $s_i \in Cu(T_1)$ waits δ time (line 4) to gather all the ECHO() messages. Since those servers are $\#echo_{CAM}$ then after δ time all non Byzantine servers collect $\#echo_{CAM}$ occurrences of all correct values in V_j. Thus all cured servers get v when invoke the function select_d_pairs_max_sn($echo\_vals_i$). To conclude, at the beginning of the maintenance() operation the *echo_vals* set is reset. During such operation there are at most $\frac{2\delta}{\Delta} = kf$ Byzantine servers (servers just affected and servers affected in the previous δ time period, that can send an incorrect ECHO() message. When $0 < \Delta \leq 3\delta$, $\#echo_{CAM} = kf + 1 > kf$ then Byzantine servers are not able to force cured servers to chose a never written or older value. The same reasoning holds for $\Delta \geq 3\delta$ considering that, contrarily to the previous case, we remove values coming from Byzantine servers affected before the maintenance() operation (cf. Figure 6.16 line 6), thus cured servers have to cope only with f Byzantine servers, being $\#echo_{CAM} = f + 1 > f$ then Byzantine servers may not force the cured servers to chose a never written or older value, concluding the proof. $\square_{Lemma\ 23}$

In the next lemma we prove that, thanks to the forwarding mechanism, after a shorter time, T_r after the end of the write() operation, there are enough servers storing the last written value to allow a read() operation to return it. We refer to T_r, which is the read() duration since this time is strictly related to the availability of the last written value to be read. This is always true except for $\Delta \geq 3\Delta$. Looking at Figure 6.20 is it clear that during each maintenance() operation there are never $\#reply_{CAM}$ servers. Now it should be clear why T_r in this case is longer, to allows cured servers to become correct and reply.

Lemma 24 (Step 2) *Let op_W be a* write(v) *operation invoked by a client c_k at time $t_B(op_W) = t$ then at time $t + T_r$ there are at least $\#reply_{CAM}$ servers $s_j \in Co(t + 2\delta)$ such that $v \in V_j$.*

Proof Let us proceed by construction. We first consider how many correct servers are storing v at $t+\delta$ (the end of op_W, cf. Lemma 21), then we evaluate this number at time $t + T_r$. Due to the communication channel synchrony, the WRITE messages from c_k are delivered by non faulty servers within the time interval $[t, t+\delta]$; any non faulty server in the time interval $[t, t + \delta]$ executes the correct algorithm code. Thus, each s_j delivers the WRITE message and executes line 5 in Figure 6.12, storing v in V_j. For Lemma 11 in the $[t, t + \delta]$ time interval there are maximum $\frac{2\delta}{\Delta} + 1 = k + 1$ Byzantine servers, thus at $t + \delta$ v is stored by at least $n - (k + 1)f = (k + 1)f + 1$ correct servers. For clarity let us now consider separately the three cases: (i) $0 < \Delta < 2\delta$, (ii) $2\delta \leq \Delta < 3\delta$ and (iii) $\Delta \geq 3\delta$. Let us remember that $T_r = 2\delta$ in case (i) and case (ii) and $T_r = 3\delta$ in case (iii).

case (i) $0 < \Delta < 2\delta$**,** $k \geq 1$ (cf. Figure 6.18), at $t + \delta$ there are $(k + 1)f + 1 = \#reply_{CAM}$ correct servers storing v. We have to prove that despite a Byzantine movement $T_i \in [t + \delta, t + 2\delta]$ (at T_i there are $\#reply_{CAM} - f = \#echo_{CAM}$ correct servers storing v) at time $t + 2\delta$ there are $\#reply_{CAM}$ correct servers storing v.

Those $\#reply_{CAM}$ correct servers may deliver the WRITE() message from c_k before of after T_{i-1}. Consequently they can send the WRITE_FW() (or ECHO()) message **before** of **after** T_i. In the first case v is insert in the V set (Figure 6.12 line 5) and at the next maintenance() operation, at T_i, v is in the ECHO() message (Figure 6.11 line 12). In the second case, when the WRITE() message from c_k is delivered by servers in $\tilde{Co}(t, t+\delta)$ after T_i , the WRITE_FW() message is sent by servers in the time interval $[T_i, t_E(op)]$ (Figure 6.12 line 9). Since a message is delivered by δ time, then by $t_E(op) + \delta = t + 2\delta$ any server has enough occurrences of v in the $fw\_vals_i \cup echo\_vals_i$ set, line 15 Figure 6.12 is executed and v is stored in V. Thus by this time, all servers that are no Byzantine during $[T_i, t_E(opW) + \delta]$ (i.e.,$n - f > \#reply_{CAM}$) store v.

case (ii) $2\delta \leq \Delta < 3\delta$ **,** $k = 1$, being $\Delta \geq 2\delta$ then during a 2δ time interval can occur only one Byzantine movement, cf. Figure 6.19. Let be T_i the time of such movement, there are two cases: (a) $T_i \in [t, t+\delta]$ and case (b) $T_i \in [t+\delta, t+2\delta]$. In **case (a)**, at $t+\delta$ there are $(k+1)f + 1 = 2f + 1$ correct servers storing v and there are not further movements up to $t + 2\delta$. To completeness, at $t + 2\delta$, thanks to the forward mechanism shown in case (i), servers that were in $B(t)$ are now storing v as well. Thus at $t + 2\delta$ there are at least $3f + 1 > \#reply_{CAM}$ servers storing v. Finally, **case (b)**, since $T_i \in [t+\delta, t+2\delta]$ then $T_i \notin [t, t+\delta]$ so that at $t+\delta$ there are $n - f = 3f + 1$ correct servers storing v. At $t + 2\delta$, due to Byzantine movements in T_i, there are $2f + 1 = \#reply_{CAM}$ servers storing v.

case (iii) $\Delta \geq 3\delta$, being $\Delta \geq 3\delta$ then during a $T_r = 3\delta$ time interval can occur only one Byzantine movement, cf. Figure 6.20. Let be T_i the time of such movement, there are two cases: (a) $T_i \in [t, t+\delta]$ and case (b) $T_i \in [t+\delta, t+3\delta]$. In **case (a)**, at $t+\delta$ there are $f + 1$ correct servers storing v and there are not further movements up to $T_i + 3\delta$. As for case (i) those $f + 1$ correct servers may deliver the WRITE() message from c_k before of after T_i. Consequently they can send the WRITE_FW() (or ECHO()) message **before** of **after** T_i. In the first case v is insert in the V set (Figure 6.17 line 5) and at the next maintenance() operation, at T_i, v is in the ECHO() message (Figure 6.16 line 13). In the second case, when the WRITE() message from c_k is delivered by servers in $\tilde{Co}(t, t+\delta)$ after T_i , the WRITE_FW() message is sent by servers in the time interval $[T_i, t_E(op)]$ (Figure 6.17 line 9). Since a message is delivered by δ time, then by $t_E(op) + \delta = t + 2\delta$ any server has $f + 1$ occurrences of v in the $fw\_vals_i \cup echo\_vals_i$ sets, and after $T_i + \delta$ values coming from cured servers are removed, thus line 11 is executed and v is stored in V. Thus by this time, all servers that are no Byzantine during $[T_i, t_E(opW) + \delta]$ (i.e.,$\#reply_{CAM}$), store v.
Finally, **case (b)**, since $T_i \in [t+\delta, t+3\delta]$ then $T_i \notin [t, t+\delta]$ so that at $t+\delta$ there are $n - f = 2f + 1$ correct servers storing v. At $t + 2\delta$, due to Byzantine movements in T_i, there are $f + 1 = \#echo_{CAM}$ servers storing v and for Lemma 23 from time $T_i + \delta \leq t + 3\delta$ there are $\#reply_{CAM}$ servers storing v.

To conclude the proof, let us consider that cured servers reset the fw_vals and $echo\_vals$ when they begin the maintenance() operation and in $T_r = 2\delta$ time there are not enough Byzantine servers to force a cured server to store in V a never

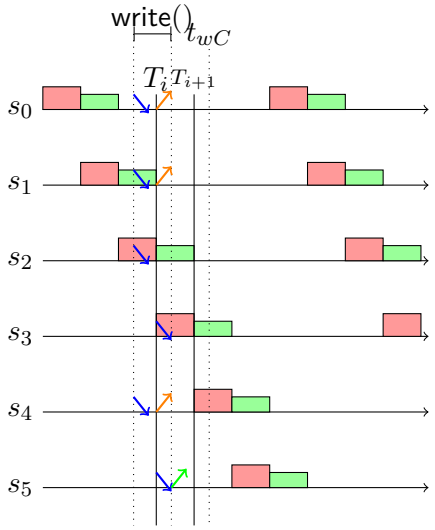


Figure 6.18. Blue arrows are the WRITE() message delivery, green arrows are the WRITE_FW() messages and orange arrows are the ECHO() messages sent.

written value. If $T_r = 3\delta$ we have that messages coming from cured servers (and thus servers that were previously Byzantine) are ignored (Figure 6.16 line 5 and line 7). $\square_{Lemma\ 24}$

For simplicity, for now on, given a write() operation op_W we call $t_B(op_W) + T_r = t_{wC}$ the **completion time** of op_W, the time at which there are at least $\#reply_{CAM}$ servers storing the value written by op_W.

Lemma 25 (Step 3) *Let op_W be a* write() *operation occurring at $t_B(op_W) = t$ and let v be the written value and t_{wC} its completion time. Then if there are no other* write() *operations after op_W, the value written by op_W is stored by all correct servers forever.*

Proof Let T_i be the time of the first Byzantine agent movement after t_{wC}, let us consider the same cases as in Lemma 24: (i) $0 < \Delta < 2\delta$, (ii) $2\delta \leq \Delta < 3\delta$ and (iii) $\Delta \geq 3\delta$.

case (i) $0 < \Delta < 2\delta$, for Lemma 24, at $t + T_r = t + 2\delta = t_{wC}$ there are $n-f > \#reply_{CAM} = (k+1)f+1$ servers storing v. Thus at time T_i, due to Byzantine movements, there are at least $n - 2f \geq \#rely_{CAM}$ servers storing v and performing the maintenance() operation. For Lemma 23 at the end of maintenance() operation all cured servers, when invoke select_d_pairs_max_sn($echo\_vals_i$) returns v. By hypothesis there are no concurrent write() operation, thus all those servers store v. When $\Delta \geq \delta$, at the end of this maintenance() operation there are $n-f \geq \#reply_{CAM}$ servers storing v, whose became $n - 2f \geq \#reply_{CAM} \geq \#echo_{CAM}$ at T_{i+1}, again, applying Lemma 23, at the end of maintenance() operation there are $n - f$ correct servers storing v. This reasoning can be iterated forever. When $\Delta < \delta$ before the end of the maintenance() other movements occur. In particular in δ time may occur $\lceil \frac{\delta}{\Delta} \rceil = \frac{k}{2} = k'$ movements. Thus each time there are up to f servers that lose v. This

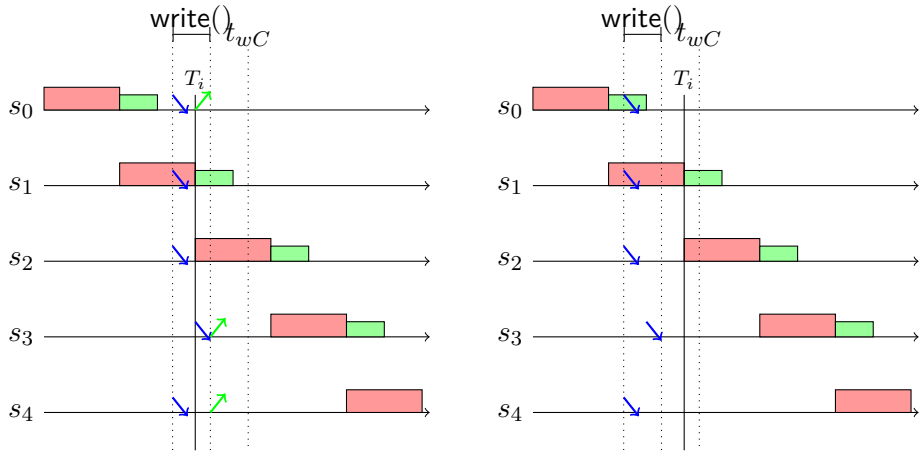


Figure 6.19. Scenario representing case (a) and case (b) for $\delta \leq \Delta < 3\delta$. Blue arrows are the WRITE() message delivery, green arrows are the WRITE_FW() messages sent.

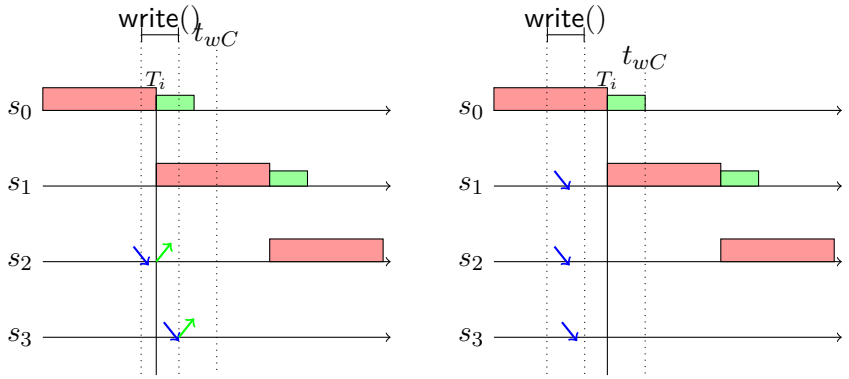


Figure 6.20. Scenario representing case (a) and case (b) for $\Delta > 3\delta$. Blue arrows are the WRITE() message delivery, green arrows are the WRITE_FW() messages sent.

is true for k' times. At time $T_{i+k'} > T_i + \delta$, servers that executed the maintenance() operation at T_i, for Lemma 23 are now able to select v, and since there a no write() operations, v is stored in V_j. Thus, at time $T_{i+k'}$ there are $n-(k'-1)f$ servers storing v since T_i and f servers that terminate the maintenance() operation started at T_i. So there are $n-(k'-1)f+f = n-k'f = (k+k'+2)f+1-k'f = (k+1)f+1 \geq \#reply_{CAM}$ servers storing v. From now on, at each T_j there are up to f servers that lose v and f servers that recover it, thus v is always stored at all correct servers.

case (ii) $2\delta \leq \Delta < 3\delta$**, case (a)** $T_{i-1} \in [t, t+\delta]$, from Lemma 24 at time t_{wC} there are $3f+1$ correct servers storing v. Using the same reasoning as in case (i), at T_i there are $2f+1 = \#echo_{CAM} = \#reply_{CAM}$ correct servers storing v, by hypothesis there are no further write() operation, thus for Lemma 23, at the end of the maintenance() operation there are $n - f \geq 3f + 1 \geq \#reply_{CAM}$ correct servers storing v. Applying the same iterative reasoning as in case (i) we have that v is stored in all correct servers forever.
case (b) $T_{i-1} \in [t+\delta, t+2\delta]$, as show in Lemma 24, at $t+\delta$ there are $n-f$ correct servers storing v. Thus at T_i there are $n-2f \geq \#reply_{CAM} \geq \#echo_{CAM}$ correct servers storing v. By hypothesis there are no further write() operations, so we can apply Lemma 23 and at the end of the maintenance() operation there are $n-f \geq \#reply_{CAM}$ correct servers storing v. Applying the same iterative reasoning as in case (i) we have that v is stored in all correct servers forever.

case (iii) $\Delta \geq 3\delta$. From Lemma 24 at time t_{wC} there are $n-f$ correct servers storing v. At the next maintenance() operation those servers are $n-2f = \#echo_{CAM}$, for Lemma 23 those servers are enough to correctly terminate the maintenance() operation. Thus, before next mobile agent movements, there are $n-f$ correct servers storing v and again this reasoning can be iterated forever concluding the proof.

$\square_{Lemma\ 25}$

Notice, for $\Delta \geq 3\delta$ case, contrarily to other cases, a value v is present in $\#reply$ correct servers only from the end of the maintenance() time to the next mobile agent movement. In other words, there is a period of δ time in which is not possible to read(), to cope with that the read() operation lasts 3δ time rather than 2δ.

Lemma 26 (Step 3) *Let $op_{W_0}, op_{W_1}, \ldots, op_{W_{q-1}}, op_{W_q}, op_{W_{q+1}}, \ldots$ be the sequence of* write() *operations issued on the regular register. Let us consider a particular op_{W_q}, let v be the value written by op_{W_q} and let $t_{wC}k$ be its completion time. Register stores v (there are at least $\#reply_{CAM}$ correct servers storing it) up to time at least t_BW_{q+d}.*

Proof The proof simply follows considering that:

- for Lemma 25 if there are no more write() operation then v, after t_{wC}, is in the register forever.
- any new written value is store in an ordered set V (cf. Figure 6.12 line 5) whose dimension is d.
- write() operations occur sequentially.

It follows that after the beginning of d write() operations, $op_{W_{q+1}}, \ldots, op_{W_{q+d}}$, v it may be no more stored in the regular register. $\square_{Lemma\ 26}$

Considering values in Tables 6.3, 6.4 and 6.5, we always have that if $0 < \Delta < 3\delta$ then $n - \tilde{B}(t, t+\delta) \geq \#reply_{CAM}$. We can rewrite it as $n_{CAM} - (\lceil \frac{\delta}{\Delta} \rceil + 1)f \geq \#reply_{CAM}$.

- $\delta \leq \Delta < 3\delta$, Table 6.3. $(k+3)f + 1 - 2f = (k+1)f + 1$;
- $\Delta \leq \delta$, Table 6.4. $(k + \frac{k}{2} + 2)f + 1 - (\frac{k}{2} + 1)f = (k+1)f + 1$ [6].

Theorem 16 (Step 4.) *Any* read() *operation returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

Proof Let us consider a read() operation op_R. We are interested in the time interval $[t_B(op_R),\ t_B(op_R) + T_r - \delta]$. Since such operation lasts T_r, the reply messages sent by correct servers within $t_B(op_R) + T_r - \delta$ are delivered by the reading client. For $0 < \Delta < 3\delta$ during $[t, t+\delta]$ time interval there are $n - \frac{k}{2} - 1 \geq \#reply_{CAM}$ correct servers that have the time to deliver the read request and reply. For $\Delta \geq 3\delta$, there is a δ time period in which a mobile agent movement occurs. But in this case $T_r = 3\delta$, thus during such operation there are the remaining 2δ time during which there are $\#reply$ correct servers that reply. Now we have to prove that what those correct servers reply with is a valid value. There are two cases, op_R is concurrent with some write() operations or not.

- op_R is not concurrent with any write() operation. Let op_W be the last write() operation such that $t_E(op_W) \leq t_B(op_R)$ and let v be the last written value. For Lemma 25 after the write completion time t_{wC} there are $\#reply_{CAM}$ correct servers storing v. Since $t_B(op_R) + T_r - \delta \geq t_{wC}$, then there are $\#reply_{CAM}$ correct servers replying with v. So the last written value is returned.

- op_R is concurrent with some write() operation. Let us consider the time interval $[t_B(op_R),\ t_B(op_R) + T_r - \delta]$. In such time there can be at most $d-1$ write() operations[7]. For Lemma 26 the last written value before $t_B(op_R)$ is still present in $\#reply_{CAM}$ correct servers. Thus at least the last written value is returned. To conclude, for Lemma 11, during the read() operation there are at most $(k+1)f$ Byzantine servers, being $\#reply_{CAM} > (k+1)f$ then Byzantine servers may not force the reader to read another or older value, if an older values has $\#reply_{CAM}$ occurrences the one with the highest sequence number is chosen. $\square_{Theorem\ 16}$

Theorem 17 *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. If $n = n_{CAM}$ according to Tables 6.3 - 6.5, then $\mathcal{P}_{reg}$ instances (cf. 6.4.1, 6.4.2 and 6.4.3) implement a SWMR Regular Register in the $(\Delta S, CAM)$ round-free Mobile Byzantine Failure model.*

Proof The proof simply follows from Theorem 15 and Theorem 16. $\square_{Theorem\ 17}$

[6] $k = \lceil \frac{2\delta}{\Delta} \rceil$, thus $\lceil \frac{\delta}{\Delta} \rceil \leq \frac{k}{2}$.

[7] If the read() operation lasts $T_r = 2\delta$ time, then there can be 2 concurrent write() operations in $T_r - \delta$ time, if $T_r = 3\delta$ there can be 3 write() operations in $T_r - \delta$ time.

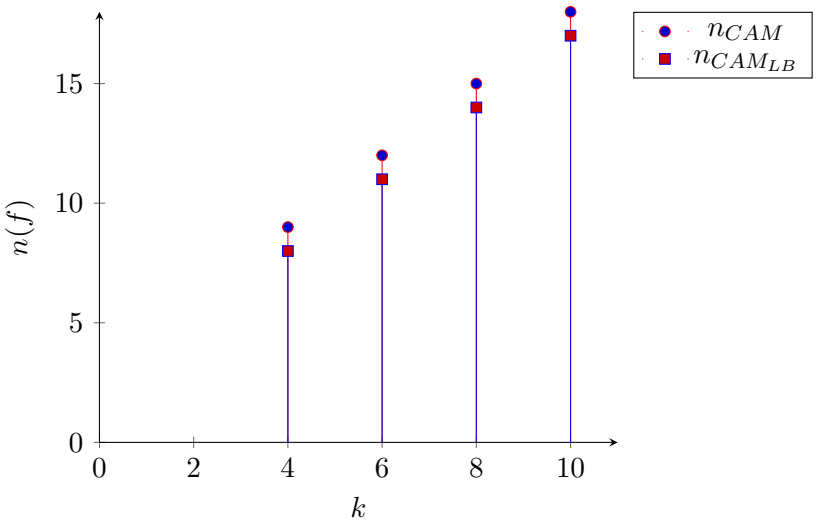


Figure 6.21. The red line is the n_{CAM} function for $\delta > \Delta$, $(k + \frac{k}{2} + 2)f + 1$. The blue line is the Function $n_{CAM_{LB}}$ described in Table 6.1 with values from Table 6.2. The distance between the two lines is just 1 server.

Lemma 27 *Protocol $\mathcal{P}_{reg}$ for $\Delta \geq 3\delta$ is tight.*

Proof The proof simply follows considering that for Lemma 8 there exists no protocol solving the safe register problem if $n \leq 3f$. $\square_{Lemma\ 27}$

Lemma 28 *Protocols $\mathcal{P}_{reg}$ for $0 < \Delta < \delta$ and Protocol $\mathcal{P}_{reg}$ for $\delta \leq \Delta < 3\delta$ are tight with respect to $\gamma \leq \delta$.*

Proof The proof follows from Theorem 14 using the values in Table 6.2 to compute $n_{CAM_{LB}}$ as defined in Table 6.1. From Lemma 23 we can set $\gamma \leq \delta$. In particular if $\delta \leq \Delta < 3\delta$ then lower bounds are respectively $4f$ if $k = 1$ and $5f$ if $k = 2$. Whose match values of $n_{CAM} = (k + 3)f + 1$. If $\delta > \Delta$, let us consider graphic depicted in Figure 6.21, where n_{CAM} and $n_{CAM_{LB}}$ are depicted for k increasing, proving that the bound for the protocol is just above, by one server, over the lower bound. $\square_{Lemma\ 27}$

6.4.5 Discussion

We proved that exists an optimal protocol $\mathcal{P}_{reg}$ that solves the Regular Register and matches bound proved in Section 6.3. Thus are verified the hypothesis for Lemma 20, which implies that lower bounds proved so far for the $(\Delta S, CAM)$ model holds for the (ITB, CAM) and (ITU, CAM) models. Clearly since parameters change from model to model then the exact values of those lower bounds are different.

Finally let us conclude this section with Figure 6.22. In such figure we case see that for $0 < \Delta < 3\delta$ (for simplicity we draw two specific cases) for any value of T_r the lower bounds do not change. For $\Delta \geq 3\delta$ lower bounds are lower when T_r is a

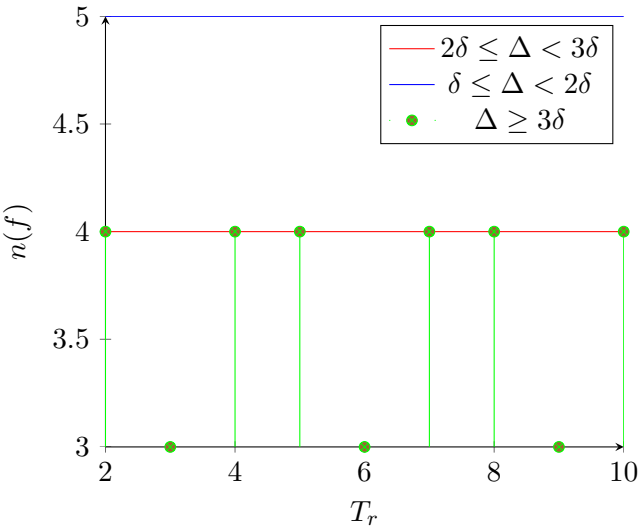


Figure 6.22. The red line is the $n_{CAM_{LB}}$ function for $k = 1$ and $T_r \in [2\delta, \ldots, 10\delta]$. The blue line is the $n_{CAM_{LB}}$ function for $k = 2$ and $T_r \in [2\delta, \ldots, 10\delta]$. The green dots is the $n_{CAM_{LB}}$ function for $\Delta \geq 3\delta$ and $T_r \in [2\delta, \ldots, 10\delta]$.

multiple of Δ. Intuitively the reason behind is that T_r has to be long enough to allow cured servers to terminate the maintenance() operation but on the other side it has to be not too long to allow mobile agents to move to much during a read() operation.

6.5 Upper Bounds for the (ITB, CAM) Synchronous model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ with respect to the number of replicas, that implements a SWMR Regular Register in a round-free synchronous system for (ITB, CAM) and (ITU, CAM) instances of the proposed MBF model. The difference with respect $(\Delta S, CAM)$ model is that the time at which mobile agents move is unknown. Notice that each mobile ma_i agent has it own Δ_i. Since we do not have any other information we consider $\Delta = min\{\Delta_1, \ldots, \Delta_f\}$. Following the approach used in the $(\Delta S, CAM)$ model, our solution is still based on the following two key points: (1) we implement a maintenance() operation, in this case executed on demand; (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 dependent by the time to terminate the maintenance() operation, δ and Δ (see Table 6.6). Contrarily to the solution presented in Section 6.4, we do not employ a a forwarding mechanism. Such mechanism is it not necessary since being the maintenance() operation on demand (i.e., $\gamma \leq 2\delta$) its duration increases and, informally speaking, there is no more need to rush to help cured servers to retrieve a lost value as soon as possible. In this case, the only propagating mechanism is on the maintenance() operation. In Table 6.6 n_{CAM} is the minimum number of required replicas, $\#reply_{CAM}$ is the minimum number of expected reply messages carrying the same value from $\#reply_{CAM}$ different servers and $\#echo_{CAM}$

$k = \lceil \frac{2\delta}{\Delta} \rceil$	$n_{CAM} \geq 2(k+1)f+1$	$\#reply_{CAM} \geq (k+1)f+1$	$\#echo_{CAM} \geq (k+1)f$	T_r	d
$k=1$	$4f+1$	$2f+1$	$2f$	2δ	3
$k=2$	$6f+1$	$3f+1$	$3f$	2δ	3
$k=4$	$10f+1$	$5f+1$	$5f$	2δ	3

Table 6.6. Parameters for $\mathcal{P}_{Rreg}$ Protocol in the (ITB, CAM) model.

is the minimum number of echo messages carrying the same value from $\#echo_{CAM}$ different servers. The last difference, with respect the $(\Delta S, CAM)$ model is the increased $\#echo_{CAM}$. In fact, since the maintenance() operation lasts δ time more then there are $\frac{k}{2}$ Byzantine servers more. What we can do, is to leverage of the failure awareness and make it possible for curing servers, to ignore information coming from servers that where Byzantine in the δ time before the beginning of the maintenance() operation, as we do in the $(\Delta S, CAM)$ model for $\Delta \geq 3\delta$.

6.5.1 $\mathcal{P}_{reg}$ Detailed Description.

The protocol $\mathcal{P}_{reg}$ for the (ITB, CAM) model is described in Figures 6.23 - 6.25, 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. In particular v is the value, sn is the associated sequence number and j is the identifier of server s_j that sent the reply back. 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 monotonically.

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 d 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 d values, it discards from V_i the value associated to the lowest sn.
- $pending\_read_i$: set variable used to collect identifiers of the clients that are currently reading.
- $cured_i$: boolean flag updated by the cured_state oracle. In particular, such variable is set to true when s_i becomes aware of its cured state and it is reset during the algorithm when s_i becomes correct.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through 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().

```
function awareAll():
(1)  broadcast ECHO(i, ⊥)
(2)  wait(δ);
(3)  broadcast ECHO(i, ⊥)

operation maintenance() executed while (TRUE) :
(4)  cured_i ← report_cured_state();
(5)  if (cured_i) then
(6)     cured_i ← false;
(7)     curing_state_i ← true;
(8)     V_i ← ∅; echo_vals_i ← ∅; pending_read_i ← ∅;curing_i ← ∅;
(9)     broadcast ECHO_REQ(i);
(10)    awareAll();
(11)    wait(2δ);
(12)    delete_cured_values(echo_vals);
(13)    insert(V_i, select_three_pairs_max_sn(echo_vals_i));
(14)    for each (j ∈ (curing_i)) do
(15)            send ECHO (i, V_i) to s_j;
(16)    endFor
(17)    curing_state_i ← false;
(18) endIf

when ECHO (j, V_j) is received:
(19)  for each (⟨v, sn⟩ ∈ V_j do
(20)          echo_vals_i ← echo_vals_i ∪ ⟨v, sn⟩_j;
(21)  endFor

when ECHO_REQ (j) is received:
(22)  curing_i ← curing_i ∪ j;
(23)  if (V_i ≠ ∅)
(24)     send ECHO(i, V_i);
(25)  endif
```

Figure 6.23. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the (ITB, CAM) model.

- $curing_i$: set used to collect servers running the maintenance() operation. Notice, to keep the code simple we do not explicitly manage how to empty such set since has not impact on safety properties.

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

- select_d_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 $\#echo_{CAM}$ 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.

- 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 6.6). If there are more pairs satisfying such condition, it returns the one with the highest sequence number.

- delete_cured_values(echo_vals): this function takes as input $echo\_vals_i$ and removes from fw_vals_i all values coming from servers that sent an ECHO() message containing $\perp$.

```
========== Client code ==========
operation write(v):
(1)  csn ← csn + 1;
(2)  broadcast WRITE(v, csn);
(3)  wait (δ);
(4)  return write_confirmation;

========== Server code ==========
when WRITE(v, csn) is received:
(5)    insert(V_i, ⟨v, csn⟩);
(6)    for each j ∈ (pending_read_i) do
(7)        send REPLY (i, {⟨v, csn⟩});
(8)    endFor
(9)    for each j ∈ (curing_i) do
(10)       send ECHO (i, V_i);
(11)   endFor
```

Figure 6.24. $\mathcal{A}_W$ algorithm implementing the write(v) operation in the (ITB, CAM) model.

The maintenance() **operation.** Such operation is executed by servers on demand when the oracle notifies them that are in a cured state. Notice that in the $(*, CAM)$ models servers knows when a mobile agent leaves them, thus depending on such knowledge they execute different actions. In particular, if a server s_i is not in a cured state then it does nothing, it just replies to ECHO_REQ() messages. Otherwise, if a server s_i is in a cured state it first cleans its local variables and *broadcast* to other servers an echo request then, after 2δ time units it removes value that may come from servers that were Byzantine before the maintenance() and updates 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 d tuples $\langle v, sn \rangle$. At the end it assigns $false$ to $cured_i$ variable, meaning that it is now correct and the $echo\_vals_i$ can now be emptied. Contrarily to the $(\Delta S, CAM)$ case, cured server notifies to all that it has been Byzantine in the previous δ time period. This is done invoking the awareAll function that broadcast a default value $\perp$ after δ time that a server discovered to be in a cured state.

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 sends a REPLY() message to all clients that are currently reading (clients in $pending\_read_i$) to notify them about the concurrent write() operation and to each server executing the maintenance() operation (servers in $curing_i$).

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.

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

```
========== Client code ==========
operation read():
(1)  reply_i ← ∅;
(2)  broadcast READ(i);
(3)  wait (2δ);
(4)  ⟨v, sn⟩ ← select_value(reply_i);
(5)  broadcast READ_ACK(i);
(6)  return v;
---------------------------------------
when REPLY (j, V_j) is received:
(7)    for each (⟨v, sn⟩ ∈ V_j) do
(8)              reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(9)    endFor


========== Server code ==========
when READ (j) is received:
(10)  pending_read_i ← pending_read_i ∪ {j};
(11)  if (V_i ≠ ∅)
(12)     then send REPLY (i, V_i);
(13)  endif
---------------------------------------
when READ_ACK (j) is received:
(14)  pending_read_i ← pending_read_i \ {j};
```

Figure 6.25. $\mathcal{A}_R$ algorithm implementing the read() operation in the (ITB, CAM) model.

6.5.2 $\mathcal{P}_{reg}$ for $\Delta \geq 4\delta$

In this work we do not analyze the case $\Delta \geq 4\delta$. When we plot $n_{CAM_{LB}}$ for $\Delta = 4\delta$ and $\gamma = 2\delta$ we obtain the graphic in Figure 6.26. As we can see when T_r is a multiple of 4δ the lower bound decrease to $3f$, with respect to $4f$ for all the other values of T_r. This means that it does not exist a protocol solving the Safe Register with less than $3f$ servers. On the other side our conjecture is that we can not solve the maintenance() operation in the ITB model with less than $3f$ servers. This is due to the uncoordinated agents movements, which implies that during the maintenance() operation there may be more Byzantine servers with respect to the previous model. For this reason we do not consider such specific case.

6.5.3 $\mathcal{P}_{reg}$ in the (ITU, CAM) model

By definition (ITU, CAM) is an instance of the (ITB, CAM) such that $\Delta = 1$. Thus, given δ and the relationship $\lceil \frac{2\delta}{\Delta} \rceil = k$ is it straightforward to have an algorithm to solve the SWMR Regular Register in the (ITU, CAM) model.

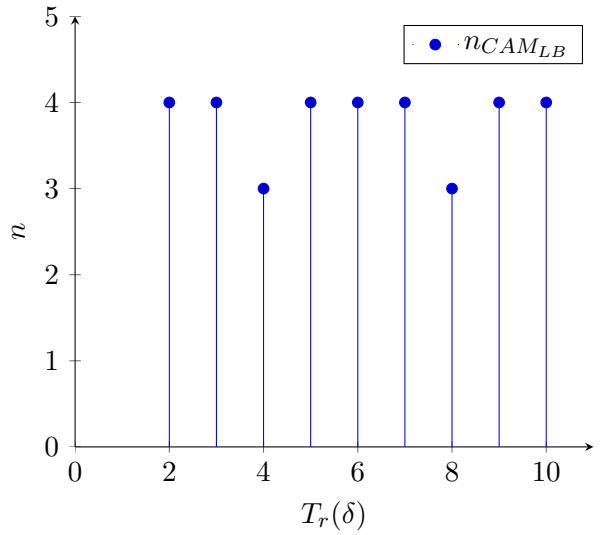


Figure 6.26. The blue line is the Function $n_{CAM_{LB}}$ described in Table 6.1 with values from Table 6.2. We consider $f = 1$ and $\gamma = 2\delta$.

6.5.4 Correctness (ITB, CAM)

To prove the correctness of $\mathcal{P}_{reg}$, we first show that the termination property is satisfied i.e, that read() and write() operations terminates.

Lemma 29 *If a correct client c_i invokes* write(v) *operation at time t then this operation terminates at time $t + \delta$.*

Proof The claim follows by considering that a write_confirmation event is returned to the writer client c_i after δ time, independently of the behavior of the servers (see lines 3-4, Figure 6.24). $\square_{Lemma\ 29}$

Lemma 30 *If a correct client c_i invokes* read() *operation at time t then this operation terminates at time $t + 2\delta$.*

Proof The claim follows by considering that a read() returns a value to the client after 2δ time, independently of the behavior of the servers (see lines 3-6, Figure 6.25). $\square_{Theorem\ 30}$

Theorem 18 (Termination) *If a correct client c_i invokes an operation, c_i returns from that operation in finite time.*

Proof The proof follows from Lemma 29 and Lemma 30. $\square_{Theorem\ 18}$

Notice, in the following, when not explicitly defined, we always consider as hypothesis that n_{CAM} follows values defined in Table 6.6 and proceed following the same four steps as in 6.4.4.

Before to prove the correctness of the maintenance() operation let us see how many Byzantine agent there may be during such operation. Since the cured server

run it as soon as the mobile agent ma_i leaves it, then ma_i movement are aligned to such operation, this agent contribution is $\frac{2\delta}{\Delta} = k$. All the others $f-1$ mobile agent are not aligned, thus their contribution is $Max\tilde{B}(t, t+2\delta) = k+1$. Thus there are $k + (k+1) \times (f-1)$ Byzantine servers during the 2δ time maintenance() operation.

Lemma 31 (Step 1) *Let $T_i = t$ be the time at which mobile agent ma_i leave s_c. Let v be the value stored at $\#echo_{CAM}$ servers $s_j \notin B(t, t+\delta) \wedge s_j \in Co(t+\delta)$, $v \in V_j \forall s_j \in Co(t+\delta)$. At time $t+2\delta$, at the end of the* maintenance()*, v is returned to s_c by the function* select_d_pairs_max_sn$(echo_vals_c)$.

Proof The proof follows considering that:

- the maintenance() employs a request-reply pattern and during such operation, by hypothesis, there are $\#echo_{CAM}$ servers that are never affected during the $[T_i, T_i+\delta]$ time period and are correct at time $T_i+\delta$. i.e., there are $\#echo_{CAM}$ servers that deliver the ECHO_REQ() message (the can be either correct or cured) but are correct at time $T_i + \delta$ such that the reply is delivered by s_c by time $T_i + 2\delta$.

- during the maintenance() operation there are $k + (k+1) \times (f-1)$ Byzantine servers, and $(\frac{k}{2})f$ servers that were Byzantine in $[t-\delta, t]$ time period, thus they could have sent incorrect messages as well.

- each cured servers, invokes AWAREALL() function, sends a $\perp$ message twice: when they are aware to be cured and δ time after. Thus by time $t+2\delta$ server running the maintenance removes from $echo\_vals$ the $(\frac{k}{2})f$ messages sent by those servers. In the end there are $k+(k+1)\times(f-1) = (k+1)f-1$ messages coming from Byzantine servers in the $echo\_vals_c$ set.

$\#echo_{CAM} = (k+1)f > (k+1)f - 1$ thus Byzantine servers can not force the function select_d_pairs_max_sn$(echo_vals_c)$ to return a not valid value and select_d_pairs_max_sn$(echo_vals_c)$ returns v that occurs $\#reply_{CAM}$ times, concluding the proof. $\square_{Lemma\ 31}$

Lemma 32 (Step 2.) *Let op_W be a* write(v) *operation invoked by a client c_k at time $t_B(op_W) = t$ then at time $t+\delta$ there are at least $\#reply_{CAM}$ servers $s_j \notin B(t+\delta)$ such that $v \in V_j$.*

Proof The proof follows considering that during the write() operation, $[t, t+\delta]$, there can be at most $(\frac{k}{2}+1)f$ mobile agents. Thus, during such time there are $n - (\frac{k}{2}+1)f = 2(k+1)f + 1 - (\frac{k}{2}+1)f = (k+\frac{k}{2}+1)f+1$ servers s_j that being either cured or correct, execute code in Figure 6.24, line 5, inserting v in V_j. Finally, $(k+\frac{k}{2}+1)f+1 > (k+1)f+1 = \#reply_{CAM}$ concluding the proof.

$\square_{Lemma\ 32}$

For simplicity, for now on, given a write() operation op_W we call $t_B(op_W)+\delta = t_{wC}$ the **completion time** of op_W, the time at which there are at least $\#reply_{CAM}$ servers storing the value written by op_W.

Lemma 33 (Step 3.) *Let op_W be a* write() *operation occurring at $t_B(op_W) = t$ and let v be the written value and let t_{wC} be its completion time. Then if there are no other* write() *operations after op_W, the value written by op_W is stored by all correct servers forever.*

Proof Following the same reasoning as Lemma 32, at time $t + \delta$, assuming that in $[t, t + \delta]$ there are $(\frac{k}{2} + 1)f$, then there are at least $(k + \frac{k}{2} + 1)f + 1$ servers s_j that being either cured or correct, execute code in Figure 6.24, line 5, inserting v in V_j. Now let us consider the following:

- Let $B_1 = \tilde{B}(t, t+\delta)$ be the set containing the $(\frac{k}{2}+1)f$ Byzantine servers during $[t, t + \delta]$, so that there are $(2k + 1)f + 1 - \frac{k}{2} = (k + \frac{k}{2} + 1)f + 1 \geq \#reply_{CUM}$ non faulty servers storing v;
  - there are $(\frac{k}{2})f$ Byzantine servers in B_1 that begin the maintenance() operation . At that time there are $\#reply_{CAM}$ non faulty servers storing v, being $\#reply_{CAM} > \#echo_{CAM}$, for Lemma 31 at the end of the maintenance() operation, by time $t + 3\delta$, those servers obtain v a result of select_d_pairs_max_sn($echo\_vals$) invocation, whose is stored in V since there are no other write() operation and since v has the highest associated sequence number.
- Let $B_2 = \tilde{B}(t + \delta, t + 2\delta)$ be the set containing Byzantine servers in the next δ period. Those servers are $\frac{k}{2}f$ (it is not $\frac{k}{2}f + 1$, otherwise we would count the Byzantine servers at $t+\delta$ twice). Thus, at $t+2\delta$ there are $(k+\frac{k}{2}+1)f+1-\frac{k}{2}f = (k + 1)f + 1 = \#reply_{CAM}$ non faulty servers storing v;
  - there are $(\frac{k}{2})f$ Byzantine servers in B_2 that begin the maintenance() operation during $[t + \delta, t + 2\delta]$ time interval. There are $\#reply_{CAM}$ non faulty servers storing v, being $\#reply_{CAM} > \#echo_{CAM}$, for Lemma 31 at the end of the maintenance() operation, by time $t + 4\delta$, those servers, get v invoking select_d_pairs_max_sn($echo\_vals$), whose is stored in V since there are no other write() operation and since v has the highest associated sequence number.
- Let $B_3 = \tilde{B}(t + 2\delta, t + 3\delta)$ be the set containing Byzantine servers in the next δ period. Those servers are $\frac{k}{2}f$. At $t + 3\delta$ there are $(k + 1)f + 1 - \frac{k}{2}f < \#reply_{CAM}$ non faulty servers storing v and the there are $(\frac{k}{2})f$ servers in B_1 that terminated the maintenance() operation storing v. Summing up there are $(k + 1)f + 1 - \frac{k}{2}f + \frac{k}{2}f = \#reply_{CAM}$ servers storing v.

Thus, after $t + 3\delta$ period there are servers becoming affected that lose v, but there are other f servers that become correct storing v, so that all correct servers store v. Since there are no more write() operation, this reasoning can be extended forever, concluding the proof. $\square_{Lemma\ 33}$

Lemma 34 (Step 3.) *Let $op_{W_0}, op_{W_1}, \ldots, op_{W_{k-1}}, op_{W_k}, op_{W_{k+1}}, \ldots$ be the sequence of* write() *operations issued on the regular register. Let us consider a particular op_{W_k},*

let v be the value written by op_{W_k} and let $t_E w_k$ be its completion time. Then the register stores v (there are at least $\#reply_{CAM}$ correct servers storing it) up to time at least $t_B W_{k+3}$.

Proof The proof simply follows considering that:

- for Lemma 33 if there are no more write() operation then v, after t_{wC}, is in the register forever.
- any new written value is store in an ordered set V (cf. Figure 6.24 line 5) whose dimension is 3.
- write() operations occur sequentially.

It follows that after the beginning of 3 write() operations, $op_{W_{k+1}}, op_{W_{k+2}}, op_{W_{k+3}}$, v it may be no more stored in the regular register. $\square_{Lemma\ 34}$

Theorem 19 (Step 4.) *Any* read() *operation returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

Proof Let us consider a read() operation op_R. We are interested in the time interval $[t_B(op_R),\ t_B(op_R) + \delta]$. Since such operation lasts 2δ, the reply messages sent by correct servers within $t_B(op_R)+\delta$ are delivered by the reading client. For $0 < \Delta < 4\delta$ during $[t, t+\delta]$ time interval there are $n - \frac{k}{2} - 1 \geq \#reply_{CAM}$ correct servers that have the time to deliver the read request and reply. Now we have to prove that what those correct servers reply with is a valid value. There are two cases, op_R is concurrent with some write() operations or not.

- op_R **is not concurrent with any** write() **operation**. Let op_W be the last write() operation such that $t_E(op_W) \leq t_B(op_R)$ and let v be the last written value. For Lemma 33 after the write completion time $t_C w$ there are $\#reply_{CAM}$ non faulty servers storing v. Since $t_B(op_R) + \delta \geq t_C w$, then there are $\#reply_{CAM}$ non faulty servers replying with v (Figure 6.25, lines 11-12). So the last written value is returned.
- op_R **is concurrent with some** write() **operation**. Let us consider the time interval $[t_B(op_R),\ t_B(op_R) + \delta]$. In such time there can be at most two write() operations. Thus for Lemma 34 the last written value before $t_B(op_R)$ is still present in $\#reply_{CAM}$ non faulty servers. Thus at least the last written value is returned.

To conclude, for Lemma 11, during the read() operation there are at most $(k+1)f$ Byzantine servers, being $\#reply_{CAM} > (k+1)f$ then Byzantine servers may not force the reader to read another or older value and even if an older values has $\#reply_{CAM}$ occurrences the one with the highest sequence number is chosen. $\square_{Theorem\ 19}$

Theorem 20 *Let n be the number of servers emulating the register and let f be the number of Byzantine agents in the (ITB, CAM) round-free Mobile Byzantine Failure model. Let δ be the upper bound on the communication latencies in the synchronous system. If $n = n_{CAM}$ according to Table 6.6 then $\mathcal{P}_{reg}$ implements a SWMR Regular Register in the (ITB, CAM) and (ITU, CAM) round-free Mobile Byzantine Failure model.*

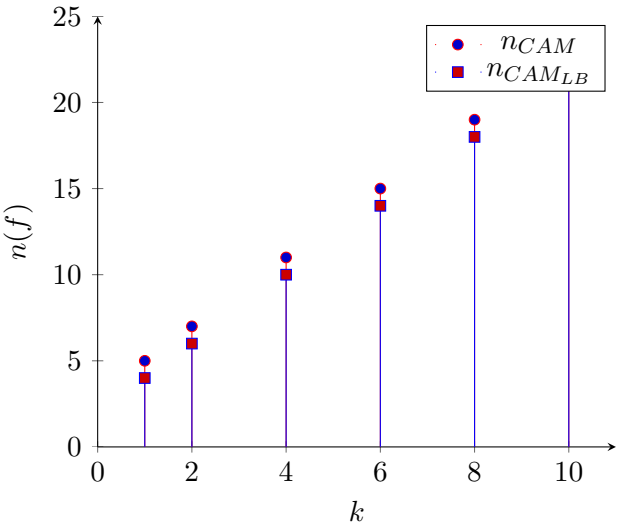


Figure 6.27. The red line is the n_{CAM} function for $\delta > \Delta$, $2(k+2)f+1$. The blue line is the Function $n_{CAM_{LB}}$ described in Table 6.1 with values from Table 6.2 (setting $\gamma = 2\delta$). The distance between the two lines is just 1 server.

Proof The proof simply follows from Theorem 18 and Theorem 19 and considering $\Delta = 1$ in the case of (ITU, CAM) model. $\square_{Theorem\ 17}$

Lemma 35 *Protocol $\mathcal{P}_{reg}$ for $0 < \Delta < 4\delta$ is tight with respect to $\gamma \leq 2\delta$.*

Proof The proof follows from Theorem 14 using the values in Table 6.2 to compute $n_{CAM_{LB}}$ as defined in Table 6.1. We can use such Theorem since does exists a tight protocol that solves Regular Register in the $(\Delta S, CAM)$ model so we can apply Lemma 20. From Lemma 31 we can set $\gamma \leq 2\delta$. Let us consider graphic depicted in Figure 6.27, where the two functions are depicted for k increasing, proving that the bound for the protocol is just above, by one server, over the lower bound. $\square_{Lemma\ 35}$

6.6 Upper Bounds for the $(\Delta S, CUM)$ Synchronous model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ with respect to the number of replicas, that implements a SWMR Regular Register in a round-free synchronous system for $(\Delta S, CUM)$ instance of the proposed MBF model. As for the $(\Delta S, CAM)$ model, the moment at which mobile agents move is known but servers are not aware of their failure state. As for the (Δ, CAM) model (cf. Section 6.4), our solution is based on the following two 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; Contrarily to the

$k = \lceil \frac{2\delta}{\Delta} \rceil, \delta \leq \Delta < 3\delta$	$n_{CUM} \geq (3k+2)f+1$	$\#reply_{CUM} \geq (2k+1)f+1$	$\#echo_{CUM} \geq (k+1)f+1$
$k=2$	$8f+1$	$5f+1$	$3f+1$
$k=1$	$5f+1$	$3f+1$	$2f+1$
$\Delta \geq 3\delta$	$4f+1$	$2f+1$	$2f+1$

Table 6.7. Parameters for $\mathcal{P}_{Rreg}$ Protocol in the $(\Delta S, CUM)$.

$(\Delta S, CAM)$ case, the values that populate auxiliary variables (i.e., not the register stored value) have a fixed life time. This is necessary since servers are never aware to be in a cured state and thus mobile agents, once they left, may force them to take wrong decisions. For this reason, there is no more a forwarding mechanism and no more a fw_vals set after a write() operation. The maintenance() operation is the only operation in charge to "push" values and is run any Δ time. It follows that is it necessary more time to spread the last written value to enough servers so that the value can be read. To this purpose, values that populate auxiliary variables can not be reset at each maintenance() operation (later it will be clearer), which implies that cured servers can not have a clean state after only one maintenance() operation. Thus in this model we have that cured servers are in such state for a longer time, in particular $\gamma \leq 2\delta$. This is the first case we saw where γ is greater than the maintenance() duration.

As in Section 6.4, 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 6.7).

6.6.1 $\mathcal{P}_{reg}$ Detailed Description for $\delta \geq \Delta$

The protocol $\mathcal{P}_{reg}$ for the $(\Delta S, CUM)$ model is described in Figures 6.28 - 6.30.

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

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

- V_i: an ordered set containing 3 tuples $\langle v, sn \rangle$, where v is a value and sn the corresponding sequence number. Such tuples are ordered incrementally according to their sn values.
- V_{safe_j}: this set has the same characteristic as V_j. The function insert$(V_{safe_i}, \langle v_k, sn_k \rangle)$ places the new value in V_{safe_i} according to the incremental order and if dimensions exceed 3 then it discards from V_{safe_i} the value associated to the lowest sn.
- W_i: is the set where servers store values coming directly from the writer, associating to it a timer, $\langle v, sn, timer \rangle$. Values from this set are deleted at the end of the maintenance() operation when the timer expires or has a value non compliant with the protocol.

- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through ECHO messages at the beginning of the maintenance() operation. The first one stores tuple $\langle v, sn \rangle_j$ propagated by servers just after the mobile Byzantine agents moved. Set $echo\_read_i$ stores identifiers of concurrently reading clients in order to notify cured servers and expedite termination of read().

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

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

- select_three_pairs_max_sn($echo\_vals_i$): this function takes as input the set $echo\_vals_i$ and returns, if they exist, 3 tuples $\langle v, sn \rangle$, such that there exist at least $\#echo_{CUM}$ occurrences in $echo\_vals_i$ of such tuple. If more than 3 of such tuples exist, the function returns the tuples with the highest sequence numbers.

- select_value($reply_i$): this function takes as input the $reply_i$ set of replies collected by client c_i and returns the pair $\langle v, sn \rangle$ occurring at least $\#reply_{CUM}$ times. If there are more pairs with the same occurrence, it returns the one with the highest sequence number.

- conCut(V_i, V_{safe_i}, W_i): this function takes as input three 3 dimension ordered sets and returns another 3 dimension ordered set. The returned set is composed by the concatenation of $V_{safe_i} \circ V_i \circ W_i$, without duplicates, truncated after the first 3 newest values (with respect to the timestamp). e.g., $V_i = \{\langle v_a, 1 \rangle, \langle v_b, 2 \rangle, \langle v_c, 3 \rangle, \langle v_d, 4 \rangle\}$ and $V_{safe_i} = \{\langle v_b, 2 \rangle, \langle v_d, 4 \rangle, \langle v_f, 5 \rangle\}$ and $W_i = \emptyset$, then the returned set is $\{\langle v_c, 3 \rangle, \langle v_d, 4 \rangle, \langle v_f, 5 \rangle\}$.

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

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

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

```
function timerCheck(W_i):
(1)  for each (⟨⟨v, csn⟩, timer⟩_j ∈ W_i) do
(2)            if (Expired(timer) ∧ (timer > 2δ)
(3)               W_i ← W_i \ ⟨⟨v, csn⟩, timer⟩_j;
(4)            endif
(5)  endFor
-----------------------------------------------------------------
operation maintenance() executed every T_i = t_0 + iΔ :
(6)  echo_vals_i ← ∅; V_i ← V_safe_i; V_safe ← ∅;
(7)  Set_i ← ∅;
(8)  for each⟨⟨v, csn⟩, timer⟩_j ∈ W_i do;
(9)  Set_i ← Set_i ∪ ⟨v, csn⟩_j;
(10) endFor
(11) broadcast ECHO(i, V_i ∪ Set_i, pending_read_i);
(12) wait(δ);
(13) timerCheck(W_i);
(14) V_i ← ∅;
-----------------------------------------------------------------
(15) when select_three_pairs_max_sn(echo_vals_i) ≠ ⊥
(16) insert(V_safe_i, select_three_pairs_max_sn(echo_vals_i));
(17) for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(18)           send REPLY (i, V_safe) to c_j;
(19) endFor
-----------------------------------------------------------------
when ECHO (j, S, pr) is received:
(20)  for each (⟨v, sn⟩_j ∈ S)
(21)           echo_vals_i ← echo_vals_i ∪ ⟨v, sn⟩_j;
(22)  endFor
(23)  echo_read_i ← echo_read_i ∪ pr;
```

Figure 6.28. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the $(\Delta S, CUM)$ model.

The read() operation. When a client wants to read, it broadcasts a READ() request to all servers and waits 2δ time to collect replies. When it is unblocked from the wait statement, it selects a value v occurring $\#reply_{CUM}$ number of times from the $reply_i$ set, sends an acknowledgement message to servers to inform that its operation is now terminated and returns v as result of the operation.

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

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

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

```
========= Client code ==========
operation write(v):
(1)  csn ← csn + 1;
(2)  broadcast WRITE(v, csn);
(3)  wait (δ);
(4)  return write_confirmation;

========= Server code ==========
when WRITE(v, csn) is received:
(5)    W_i ← W_i ∪ ⟨⟨v, csn⟩, setTimer(2δ)⟩;
(6)    broadcast ECHO(i, ⟨v, csn⟩, pending_read_i);
(7)    for each j ∈ (pending_read_i ∪ echo_read_i) do
(8)        send REPLY (i, {⟨v, csn⟩});
(9)    endFor
```

Figure 6.29. $\mathcal{A}_W$ algorithm implementing the write(v) operation in the $(\Delta S, CUM)$ model.

```
========= Client code ==========
operation read():
(1)  reply_i ← ∅;
(2)  broadcast READ(i);
(3)  wait (2δ);
(4)  ⟨v, sn⟩ ← select_value(reply_i);
(5)  broadcast READ_ACK(i);
(6)  return v;
---------------------------------
when REPLY (j, V_set) is received:
(7)    for each (⟨v, sn⟩ ∈ V_set) do
(8)            reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(9)    endFor


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

Figure 6.30. $\mathcal{A}_R$ algorithm implementing the read() operation in the $(\Delta S, CUM)$ model.

6.6.2 $\mathcal{P}_{reg}$ for $\delta > \Delta$

From Corollary 4, maintenance() operation can not last less than δ time. When $\delta > \Delta$, during such operation Byzantine agent movements may occur and contrarily to the $(\Delta S, CAM)$ model, servers are not aware of their failure state. Thus servers can not trigger maintenance() operation when they enter in a cured state and neither at each agent movements, they have to trigger it by themselves periodically. As we already declared, the maintenance() operation is not the main scope of such work, thus in such case we propose the same maintenance() implementation in both $(\Delta S, CUM)$ for $\delta > \Delta$ and (ITB, CUM) models (cf. 6.7). In such case the resulting curing time $\gamma \leq 4\delta$. Being γ the same in both cases, this implies that lower bounds

with respect to such parameter are the same as solution to solve the regular register problem. The are no clue about the optimality of γ in those models, but the solution to implement the regular register is optimal with respect to the γ deriving from the maintenance() operations we design.

6.6.3 Correctness $(\Delta S, CUM)$

To prove the correctness of $\mathcal{P}_{reg}$ we demonstrate that the termination property is satisfied i.e, that read() and write() operations terminates. For the validity property we follow the same four steps as defined in 6.4.4.

Lemma 36 *If a correct client c_i invokes* write(v) *operation at time t then this operation terminates at time $t + \delta$.*

Proof The claim simply follows by considering that a write_confirmation event is returned to the writer client c_i after δ time, independently of the behavior of the servers (see lines 3-4, Figure 6.29). $\square_{Lemma\ 36}$

Lemma 37 *If a correct client c_i invokes* read() *operation at time t then this operation terminates at time $t + 2\delta$.*

Proof The claim simply follows by considering that a read() returns a value to the client after 2δ time, independently of the behaviour of the servers (see lines 12-15, Figure 6.30). $\square_{Lemma\ 37}$

Theorem 21 (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 36 and Lemma 37. $\square_{Theorem\ 21}$

Lemma 38 (Step 1.) *Let v be the value stored at $\#echo_{CUM}$ correct servers $s_j \in Co(T_i)$, $v \in V_j \forall s_j \in Co(T_i)$. Then $\forall s_c \in Cu(T_i)$ at $T_i + \delta$ (i.e., at the end of the* maintenance()*) v is returned by the function* select_three_pairs_max_sn($echo\_vals_i$).

Proof By hypotheses at T_i there are $\#echo_{CUM}$ correct servers s_j storing the same v and running the code in Figure 6.28. In particular each server broadcasts a ECHO() message with attached the content of V_j which contains v (line 11). Messages sent by $\#echo_{CUM}$ correct servers are delivered by s_c and stored in $echo\_vals_c$. The system is synchronous, thus by time $Ti+\delta$ function select_three_pairs_max_sn($echo\_vals_c$) returns v. $\square_{Lemma\ 38}$

Lemma 39 *Let s_i be a correct server running the* maintenance() *operation at time T_i, then if v is returned by the function* select_three_pairs_max_sn($echo\_vals_i$) *there exist a* write() *operation that wrote such value.*

Proof Let us suppose that select_three_pairs_max_sn($echo\_vals_i$) returns v' and there no exist a write()(v'). This means that s_i collects in $echo\_vals_i$ more than $\#echo_{CUM}$ occurrences of v' coming from cured and Byzantine servers. Let us consider cured servers s_c at time T_c. At the beginning of the maintenance() operation s_c broadcasts values contained in V_i and W_i (Figure 6.28 line 11). V_i is reset at each operation with the content of V_{safe_i} which is reset at each operation (line 6). It follows that s_c broadcasts non valid values contained in V_i only during the maintenance() operation run a T_c. Contrarily, values in W_i, depending on k, are broadcast only at T_c or also at T_{c+1}. Let us consider two cases: $k = 1$ and $k = 2$.
case $k = 1$**:** In this case since $\Delta \geq 2\delta$ and the maximum value of the timer associated to a value is 2δ, then each cured server s_c broadcasts a non valid value contained in W_i only during the first maintenance() operation. Thus, during each maintenance() operation there are f Byzantine servers and f cured servers, those are not enough to send $\#echo_{CUM} = 2f + 1$ occurrences of v'. For Lemma 38 this is the necessary condition to return v' invoking select_three_pairs_max_sn($echo\_vals_i$), leading to a contradiction.
case $k = 2$**:** $\Delta \leq 2\delta$ and the maximum value of the timer associated to a value is 2δ, then each cured server s_c broadcasts a non valid value contained in W_i during the first and the second maintenance() operations. Thus, each cured server s_c broadcast a non valid value contained in W_i during two maintenance() operations. Summing up, during each maintenance() operation at time T_i there are f Byzantine servers, f cured servers and f servers that were cured during the previous operation. Those servers are not enough to send $\#echo_{CUM} = 3f + 1$ occurrences of v', for Lemma 38 this is the necessary condition to return v' invoking select_three_pairs_max_sn($echo\_vals_i$), leading to a contradiction and concluding the proof. $\square_{Lemma\ 39}$

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

Corollary 10 *Let s_i be a non faulty process and v a value in W_i. Such value is in W_i during at most k sequential* maintenance() *operations.*

Finally, considering that servers reply during a read() operation with values in W_i it follows that servers can be in a cured state for 2δ time.

Corollary 11 *Protocol $\mathcal{P}$ implements a* maintenance() *operation that implies $\gamma \leq 2\delta$.*

Lemma 40 *Let T_c be the time at which s_c become cured. Each cured server s_c can reply back with incorrect message to a* READ() *message during a period of 2δ time.*

Proof The proof directly follows considering that the content of a REPLY() message comes from the V_c, V_{safe_c} and W_i sets. The first one is filled with the content of V_{safe_c} at the beginning of each manteneance() operation and after δ time is reset (cf. Figure 6.28 lines 12-14). The second one is emptied at the beginning of each manteneance() operation and the third one keeps its value during k maintenance() operations (cf. Corollary 10). Thus by time $T_c + 2\delta$ s_c cleans all the values that could come from a mobile agent. $\square_{Lemma\ 40}$

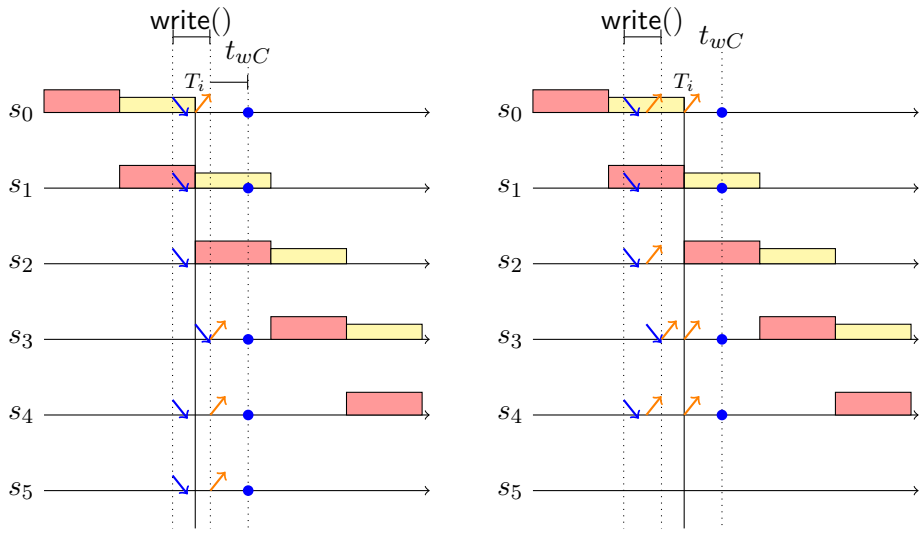


Figure 6.31. Blue arrows are the WRITE() message delivery, orange arrows are the WRITE_FW() messages sent. Blue dots are the time at which servers return v invoking select_three_pairs_max_sn($echo\_vals_i$).

Lemma 41 (Step 2.) *Let op_W be a* write(v) *operation invoked by a client c_k at time $t_B(op_W) = t$ then at time $t + \delta$ there are at least $n - 2f \geq \#reply_{CUM}$ non faulty servers $s_j \notin \tilde{B}(t, t + \delta)$ such that $v \in W_i$ and is returned by the function* concCut().

Proof Due to the communication channel synchrony, the WRITE messages from c_k are delivered by servers within the time interval $[t, t + \delta]$; any non faulty server s_j executes the correct algorithm code. When s_j delivers WRITE message it executes line 5 Figure 6.29, it stores the value in W_j and sets the associated timer to 2δ.
For Lemma 11 in the $[t, t+\delta]$ time interval there are maximum $2f$ Byzantine servers, thus at $t+\delta$ $v \in W_j$ at $n-2f = (3k+2)f+1-2f = 3kf+1 \geq \#reply_{CUM}$ correct servers if $\Delta < 3\delta$. Otherwise, $v \in W_j$ at $4f+1-2f = 2f+1 \geq \#reply_{CUM}$ correct servers if $\Delta \geq 3\delta$. Since write() operations are sequential, during $[t, t + \delta]$ there is only one new value inserted in W_i, which is returned by the function conCut() by construction. $\square_{Lemma\ 41}$

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

Lemma 42 (Step 3.) *Let op_W be a* write() *operation and let v be the written value. If there are no other* write() *operations, the value written by op_W is stored by all correct servers forever (i.e., v is returned invoking the* conCut() *function).*

Proof From Lemma 41 at time t_{wC} there are at least $n - 2f > \#reply_{CUM}$ correct servers s_j that returns v when invoke function conCut(). We consider two cases: $\delta \leq \Delta < 2\delta$ $(k = 2)$ and $\Delta \geq 2\delta$.
case 1 $\delta \leq \Delta < 2\delta$**:** in this case there is a set of $3kf + 1 = 6f + 1$ non faulty

servers s_i such that $v \in W_i$. At the beginning of the next maintenance() operation, for Corollary 10 those non faulty servers still have $v \in W_i$. Up to f mobile agents move and such set of servers decreases to $5f + 1 \geq \#echo_{CUM}$. For Lemma 38 at the end of the maintenance() all non faulty servers return v when invoking select_three_pairs_max_sn($echo\_vals_i$). Since there are no more write() operation and v is the last written value (i.e., has the highest sequence number), $n - f$ non faulty servers insert v in V_{safe_i}. It follows that cyclically before each mobile agents movements there are f servers more that store v thanks to the maintenance() and f servers that lose v because affected, but the remaining set of non faulty servers is enough to successfully run the maintenance() operation (cf. Lemma 38)) so all correct servers store v.
case 2 $\Delta \geq 2\delta$: let $[t, t + \delta]$ be the time interval during which op_W took place and let T_i be the time at which mobile agent move, two cases may occur, case (2.1) $T_i \in [t, t + \delta]$ and case (2.2) $T_i \notin [t, t + \delta]$. In **case (2.1)** T_i occurs during the write() operation. There are $n - 2f$ correct servers s_j having $v \in W_j$, Figure 6.29, line 5 .Those servers may deliver v before or after T_i. In the first case v is broadcast at the beginning of the maintenance() operation (cf. s_4 in the first part in Figure 6.31, Figure 6.28, line 11), $v \in W_j$ for Corollary 10. In the second case v is broadcast just after the delivery (cf. s_3 in the first part in Figure 6.31, Figure 6.29, line 6), at most v is delivered by time $t+\delta$ and hereafter broadcast. It follows that by time $t+2\delta$ all non Byzantine servers return v from function select_three_pairs_max_sn($echo\_vals_i$)(). Since there are no more write() operation and v is the last written value (i.e., has the highest sequence number), then v is inserted in V_{safe_i} at all correct servers. Being $\Delta \geq 2\delta$ then $t + 2\delta < T_{i+1}$, the next mobile agents movement. Finally in **case (2.2)**, since $T_i \notin [t, t + \delta]$ then at t_{wC} there are $n - f$ servers storing v. Which is the same situation that happens in case (2.1) at time $t + 2\delta < T_{i+1}$. It follows that cyclically before each agent movements there are f servers more that store v thanks to the maintenance() and f servers that lose v because faulty, but this set of non faulty servers is enough to successfully run the maintenance() operation (cf. Lemma 38)) so all correct servers store v.

$\square_{Lemma\ 42}$

Lemma 43 (Step 3.) *Let $op_{W_0}, op_{W_1}, \ldots, op_{W_{k-1}}, op_{W_k}, op_{W_{k+1}}, \ldots$ be the sequence of* write() *operation issued on the regular register. Let us consider a generic op_{W_k}, let v be the written value by such operation and let t_{wC} be its completion time. Then v is in the register (there are $\#reply_{CUM}$ correct servers storing it) up to time at least $t_B W_{k+3}$.*

Proof The proof simply follows considering that:

- for Lemma 42 if there are no more write() operation then v, after t_{wC}, is in the register forever;
- any new written value eventually is stored in ordered set V_{safe}, whose dimension is 3;
- write() operation occur sequentially.

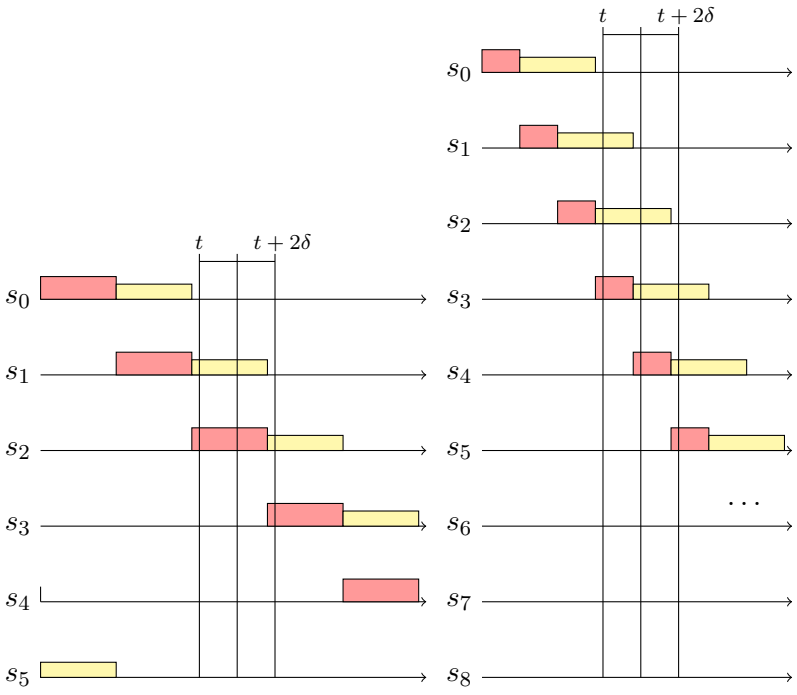


Figure 6.32. In the first scenario $\Delta \geq 2\delta$ and in second one is $\Delta \geq \delta$.

It follows that after 3 write() operations, $op_{W_{k+1}}, op_{W_{k+2}}, op_{W_{k+3}}$, v is no more stored in the regular register. $\square_{Lemma\ 43}$

Before to prove the validity property, let us consider how many Byzantine and cured servers can be present during a read() operation that last 2δ, cf. Figure 6.32. If $k = 2$ there can be up to $(k+1)f = 3f$ Byzantine servers and $2f$ cured servers. If $k = 1$ there can be up to $(k+1)f = 2f$ Byzantine servers and f cured servers.

Theorem 22 (Step 4.) *Any* read() *operation returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

Proof Let us consider a read() operation op_R. We are interested in the time interval $[t_B(op_R), t_B(op_R) + \delta]$. The operation lasts 2δ, thus reply messages sent by correct servers within $t_B(op_R) + \delta$ are delivered by the reading client. For $0 < \Delta < 4\delta$ during $[t, t + \delta]$ time interval there are $n - \frac{k}{2} - 1 \geq \#reply_{CUM}$ correct servers that have the time to deliver the read request and reply. Now we have to prove that what those correct servers reply with is a valid value. There are two cases, op_R is concurrent with some write() operations or not.

- op_R **is not concurrent with any** write() **operation**. Let op_W be the last write() operation such that $t_E(op_W) \leq t_B(op_R)$ and let v be the last written value. For Lemma 42 after the write completion time t_{wC} there are at least $\#reply_{CUM}$ correct servers storing v (i.e., $v \in \mathsf{conCut}(V_i, V_{safe_i}, W_i)$. Since $t_B(op_R) + \delta \geq t_C w$, then there are $\#reply_{CUM}$ correct servers replying with v. So the last written value is returned.
- op_R **is concurrent with some** write() **operation**. Let us consider the time

interval $[t_B(op_R),\ t_B(op_R) + \delta]$. In such time there can be at most three write() operations. Thus for Lemma 43 the last written value before $t_B(op_R)$ is still present in $\#reply_{CUM}$ correct servers. At least the last written value is returned. To conclude, for Lemma 40 Byzantine and cured servers can no force correct servers to store and thus to reply with a never written value. Only cured and Byzantine servers can reply with non valid values. As we stated, if $k = 1$ there are up to $3f$ non correct servers. If $k = 2$ there are $5f$ non correct servers. In both cases the threshold $\#reply_{CUM}$ is higher than the occurrences of non valid values that a reader can deliver. Mobile agents can not force the reader to read another or older value and even if an older values has $\#reply_{CUM}$ occurrences the one with the highest sequence number is chosen. $\square_{Theorem\ 22}$

Theorem 23 *Let n be the number of servers emulating the register and let f be the number of Byzantine agents in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model. Let δ be the upper bound on the communication latencies in the synchronous system. If (i) $\Delta \geq \delta$ and (ii) n follows values listed in Table 6.7, then $\mathcal{P}_{reg}$ implements a SWMR Regular Register in the $(\Delta S, CUM)$ round-free Mobile Byzantine Failure model.*

Proof The proof simply follows from Theorem 21 and Theorem 22. $\square_{Theorem\ 23}$

Lemma 44 *Protocol $\mathcal{P}_{reg}$ for $\Delta \geq \delta$ is tight with respect to $\gamma \leq 2\delta$.*

Proof The proof follows from Theorem 14 using the values in Table 6.2 to compute $n_{CUM_{LB}}$ as defined in Table 6.1. From Lemma 38 and Corollary 11 we can set $\gamma \leq 2\delta$. In particular if $\Delta \geq \delta$ then lower bounds are respectively $8f$ if $k = 1$ and $5f$ if $k = 2$, whose match $n_{CUM} = (3k + 2)f + 1$. Finally $4f$ if $\Delta \geq 3\delta$ matches the lower bound for the $(\Delta S, CUM)$ model (cf. Lemma 9), concluding the proof. $\square_{Lemma\ 44}$

6.7 Upper Bounds for the (ITB, CUM) Synchronous model

In this section, we present an optimal protocol $\mathcal{P}_{reg}$ with respect to the number of replicas, that implements a SWMR Regular Register in a round-free synchronous system for (ITB, CUM) and consequently (ITU, CUM) instances of the proposed MBF model. In this model we use all techniques we used so far, in particular the maintenance() operation needs to be carefully managed. In this model, as for the $(\Delta S, CUM)$ model, servers are not aware of their failure state, thus they have to run such operation either they are correct or cured. In addition, in the (ITB, CUM) model, the moment at which mobile agents move is not known, thus as for the (ITB, CAM) case, a request-reply pattern is used to implement the maintenance() operation. The read() and write() operations follows the same approach as in the previous models. Table 6.8 reports the parameters for the protocol. In particular n_{CUM} is the bound on the number of servers, $\#reply_{CUM}$ is minimum number of occurrences from different servers of a value to be accepted as a reply during a read() operation and $\#echo_{CUM}$ is the minimum number of occurrences from different servers of a value to be accepted during the maintenance() operation.

$k = \lceil \frac{2\delta}{\Delta} \rceil \geq 1$	$n_{CUM} \geq (5k+2)f+1$	$\#reply_{CUM} \geq (3k+1)f+1$	$\#echo_{CUM} \geq (3k)+1f$
$k=2$	$12f+1$	$7f+1$	$6f+1$
$k=1$	$7f+1$	$4f+1$	$4f+1$

	n_{CUM}	$\#reply_{CUM}$	$\#echo_{CUM}$
$3\delta \leq \Delta < 4\delta$	$6f+1$	$3f+1$	$3f+1$
$4\delta \leq \Delta < 5\delta$	$5f+1$	$3f+1$	$3f+1$
$\Delta \geq 5\delta$	$4f+1$	$3f+1$	$3f+1$

Table 6.8. Parameters for $\mathcal{P}_{Rreg}$ Protocol for the (ITB, CUM) model.

6.7.1 $\mathcal{P}_{reg}$ Detailed Description

The protocol $\mathcal{P}_{reg}$ for the (ITB, CUM) model is described in Figures 6.33 - 6.35, 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. In particular v is the value, sn is the associated sequence number and j is the identifier of server s_j that sent the reply back. 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 monotonically.

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

- V_i: an ordered set containing 3 tuples $\langle v, sn \rangle$, where v is a value and sn the corresponding sequence number. Such tuples are ordered incrementally according to their sn values.
- V_{safe_j}: this set has the same characteristic as V_j. The insert$(V_{safe_i}, \langle v_k, sn_k \rangle)$ function places the new value in V_{safe_i} according to the incremental order and if dimensions exceed 3 then it discards from V_{safe_i} the value associated to the lowest sn.
- W_i: is the set where servers store values coming directly from the writer, associating to it a timer, $\langle v, sn, timer \rangle$. Values from this set are deleted when the timer expires or has a value non compliant with the protocol.
- $pending\_read_i$: set variable used to collect identifiers of the clients that are currently reading.
- $echo\_vals_i$ and $echo\_read_i$: two sets used to collect information propagated through 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().
- $curing_i$: set used to collect servers running the maintenance() operation. Notice, to keep the code simple we do not explicitly manage how to empty such set since has not impact on safety properties.

```
operation timerCheck(W_i) executed while (TRUE) :
(1)  for each (⟨⟨v, csn⟩, timer⟩_j ∈ W_i) do
(2)          if (Expires(timer) ∧ (timer > 4δ))
(3)             W_i ← W_i \ ⟨⟨v, csn⟩, timer⟩_j;
(4)          endif
(5)  endFor
-----------------------------------------------------------------
operation maintenance() executed while (TRUE) :
(6)  echo_vals_i ← ∅; V_i ← V_safe_i; V_safe ← ∅;
(7)  rand ← new_rand();
(8)  broadcast ECHO_REQ(i, rand);
(9)  wait(2δ);
-----------------------------------------------------------------
when select_three_pairs_max_sn(echo_vals_i) ≠ ⊥
(10) insert(V_safe_i, select_three_pairs_max_sn(echo_vals_i));
(11) for each (j ∈ (pending_read_i ∪ echo_read_i)) do
(12)         send REPLY (i, V_safe) to c_j;
(13) endFor
-----------------------------------------------------------------
when ECHO (j, S, pr, r) is received:
(14)  if (rand = r)then:
(15)     echo_vals_i ← echo_vals_i ∪ ⟨v, sn⟩_j;
(16)     echo_read_i ← echo_read_i ∪ pr;
(17)  endIf
-----------------------------------------------------------------
when ECHO_REQ (j, r) is received:
(18)  Set_i ← ∅;
(19)  for each⟨⟨v, csn⟩, epoch⟩_j ∈ W_i do;
(20)  Set_i ← Set_i ∪ ⟨v, csn⟩_j;
(21)  endFor
(22)  send ECHO(i, V_i ∪ Set_i, r) to s_j;
```

Figure 6.33. $\mathcal{A}_M$ algorithm implementing the maintenance() operation (code for server s_i) in the (ITB, CUM) model.

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 $\#echo_{CUM}$ occurrences in $echo\_vals_i$ of such tuple. If more than three of such tuples exist, the function returns the tuples with the highest sequence numbers.
- select_value($reply_i$): this function takes as input the $reply_i$ set of replies collected by client c_i and returns the pair $\langle v, sn\rangle$ occurring occurring at least $\#reply_{CUM}$ times. If there are more pairs with the same occurrence, it returns the one with the highest sequence number.
- conCut(V_i, V_{safe_i}, W_i): this function takes as input three 3 dimension ordered sets and returns another 3 dimension ordered set. The returned set is composed by the concatenation of $V_{safe_i} \circ V_i \circ W_i$, without duplicates, truncated after the first 3 newest values (with respect to the timestamp). e.g., $V_i = \{\langle v_a, 1\rangle, \langle v_b, 2\rangle, \langle v_c, 3\rangle, \langle v_d, 4\rangle\}$ and $V_{safe_i} = \{\langle v_b, 2\rangle, \langle v_d, 4\rangle, \langle v_f, 5\rangle\}$ and $W_i = \emptyset$, then the returned set is $\{\langle v_c, 3\rangle, \langle v_d, 4\rangle, \langle v_f, 5\rangle\}$.

The maintenance() operation. Such operation is executed by servers every 2δ times. Each time s_i resets its variables, except for W_i (that is continuously checked

```
========== Client code ==========
operation write(v):
(1) csn ← csn + 1;
(2) broadcast WRITE(v, csn);
(3) wait (δ);
(4) return write_confirmation;

========== Server code ==========
when WRITE(v, csn) is received:
(5)   W_i ← W_i ∪ ⟨⟨v, csn⟩, setTimer(4δ)}⟩;
(6)   for each j ∈ (pending_read_i ∪ echo_read_i) do
(7)       send REPLY (i, {⟨v, csn⟩});
(8)   endFor
(9)   broadcast ECHO(i, ⟨v, csn⟩);
```

Figure 6.34. $\mathcal{A}_W$ algorithm implementing the write(v) operation in the (ITB, CUM) model.

by the function timerCheck()) and the content of V_{safe_i}, which overrides the content of V_i, before to be reset. Then s_i choses a random number to associate to such particular maintenance() operation instance [8], broadcast the ECHO_REQ() message and waits 2δ before to restart the operation. In the meantime ECHO() messages are delivered and stored in the $echo\_vals_i$ set. When there is value v whose occurrence overcomes the $\#echo_{CUM}$ threshold, such value is stored in V_{safe_i} and a REPLY() message with v is sent to current reader clients (if any).
Notice that, contrarily to all the previous models, servers are not aware about their failure state and do not synchronize the maintenance() operation with each other. The first consequence is a that a mobile agent may leave a cured server running such operation with garbage in server variables, making the operation unfruitful. Such server has to wait 2δ to run again the maintenance() operation with clean variables, so that next time it will be effective, which implies $\gamma \leq 4\delta$.
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 message, it updates W_i, associating to such value a timer 4δ. 4δ it is a consequence of the double maintenance() operation that a cured server has to run in order to be sure to be correct. Thus if a server is correct it keeps v in W_i during 4δ, which is enough for our purposes. On the other side a cured servers keeps a value (not necessarily coming from a write() operation) no more than the time it is in a cured state, 4δ, which is safe. After storing v in W_i, such value is inserted in REPLY() message to all clients that are currently reading (clients in $pending\_read_i$) to notify them about the concurrent write() operation and to any server executing the maintenance() operation (servers in $curing_i$).
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

[8] Is it out of the scope of this work to describe such function, we assume that Byzantine server can not predict the random number chosen next. The aim of such number is to prevent Byzantine servers to send reply to maintenance() operations before their invocation, or, in other words, it prevents correct servers to accept those replies.

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 sends a reply back to c_j. Note that, in the REPLY() message is carried the result of conCut(V_i, V_{safe_i}, W_i). In this case, if the server is correct then V_i contains valid values, and V_{safe_i} contains valid values by construction, since it comes from values sent during the current maintenance(). If the server is cured, then V_i and W_i may contain any value. Finally, s_i forwards a READ_FW message to inform other servers about c_j read request. This is useful in case some server missed the READ(j) message as it was affected by mobile Byzantine agent when such message has been delivered.

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

```
========= Client code ==========
operation read():
(1)  reply_i ← ∅;
(2)  broadcast READ(i);
(3)  wait (2δ);
(4)  ⟨v, sn⟩ ← select_value(reply_i);
(5)  broadcast READ_ACK(i);
(6)  return v;
---------------------------------------
when REPLY (j, V_j) is received:
(7)    for each (⟨v, sn⟩ ∈ V_j) do
(8)              reply_i ← reply_i ∪ {⟨j, ⟨v, sn⟩⟩};
(9)    endFor


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

Figure 6.35. $\mathcal{A}_R$ algorithm implementing the read() operation in the (ITB, CUM) model.

6.7.2 $\mathcal{P}_{reg}$ for the (ITU, CUM) model

By definition (ITU, CUM) is an instance of the (ITB, CUM) such that $\Delta = 1$. Thus, given δ and the relationship $\lceil \frac{2\delta}{\Delta} \rceil = k$ is it straightforward to have an algorithm to solve the SWMR Regular Register in the (ITU, CUM) model.

6.7.3 Correctness (ITB, CUM)

To prove the correctness of $\mathcal{P}_{reg}$ we demonstrate that the termination property is satisfied i.e, that read() and write() operations terminates. For the validity property we follow te same four steps as defined in 6.4.4.

Lemma 45 *If a correct client c_i invokes* write(v) *operation at time t then this operation terminates at time $t + \delta$.*

Proof The claim simply follows by considering that a write_confirmation event is returned to the writer client c_i after δ time, independently of the behavior of the servers (see lines 3-4, Figure 6.34). $\square_{Lemma\ 45}$

Lemma 46 *If a correct client c_i invokes* read() *operation at time t then this operation terminates at time $t + 2\delta$.*

Proof The claim simply follows by considering that a read() returns a value to the client after 2δ time, independently of the behaviour of the servers (see lines 12-15, Figure 6.35). $\square_{Lemma\ 46}$

Theorem 24 (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 45 and Lemma 46. $\square_{Theorem\ 24}$

To easy the next Lemmas let us use state the following result.

Lemma 47 *Let $[t, t + 2\delta]$ be a generic interval, then there are always at least $\#reply_{CUM}$ correct servers that reply during the $[t, t + \delta]$ time interval.*

Proof This follows considering the definition of minimum number of correct replies during a time interval (cf. Corollary 6). Since does exist a tight protocol $\mathcal{P}$ solving a regular register in the $(\Delta S, CAM)$ model, then for Lemma 20, is it possible to apply values from Table 6.2 to compute the minimum number of correct replies during the considered time interval, substituting values in each case the result is always at least $\#reply_{CUM}$. $\square_{Lemma\ 47}$

Lemma 48 (Step 1.) *Let T_i be the time at which mobile agent ma_i leave s_c and let $t \leq T_i + 2\delta$ the time at which s_c run the second* maintenance() *operation. Let v be the value stored at $\#echo_{CUM}$ servers $s_j \notin B(t, t + \delta)$, $v \in V_j \forall s_j \notin B(t, t + \delta)$. At time $t + 2\delta$, at the end of the* maintenance(), *v is returned to s_c by the function* select_three_pairs_max_sn($echo\_vals_c$).

Proof The proof follows considering that:

- the maintenance() employs a request-reply pattern and during such operation, by hypothesis, there are $\#echo_{CUM}$ servers that are never affected during the $[t, t + \delta]$ time period and are storing v at time $t + \delta$. i.e., there are $\#echo_{CUM}$ servers that deliver the ECHO_REQ() message (the can be either correct or cured) but are storing v in V at time $t + \delta$ such that the reply is delivered by s_c by time $t + 2\delta$.

- during the maintenance() operation can incorrectly contribute $(k+1)f$ Byzantine servers, and $(2k)f$ servers that were Byzantine in $[t-4\delta, t]$ time period, thus they could be still in a cured state [9].
- when the ECHO_REQ() message is sent, s_c uses a random number in order to be able to accept only ECHO() message sent after t.

$\#echo_{CUM} = (3k)f + 1 > 3kf$ thus Byzantine servers can not force the function select_three_pairs_max_sn($echo\_vals_c$) to return a not valid value so it returns v that occurs $\#reply_{CUM}$ times, which is true since there exist $\#echo\_CUM$ non faulty servers that reply to the ECHO_REQ() message sending back v, concluding the proof. $\square_{Lemma\ 38}$

In the sequel we consider $\gamma \leq 4\delta$. In the previous Lemma we proved that cured servers s_c can get valid values in 2δ time. Contrarily to all the previous model, the maintenance() operation is triggered each 2δ. Thus a mobile agent, just before to leave could leave s_c with the timer just reset and garbage in the $echo\_set_c$ and V_c sets, which does not allow s_c to correctly terminate the operation. Thus s_c has to wait 2δ before to effectively starts a correct maintenance() operation. In the sequel we refer to the **first maintenance** as the operation that may be ineffective and we refer to the **second maintenance** as the operation that allows a cured server to retrieve and store valid values. It is straightforward that $\gamma \leq 4\delta$ and the next Corollary just follows.

Corollary 12 *Protocol $\mathcal{P}$ implements a* maintenance() *operation that implies $\gamma \leq 4\delta$.*

Lemma 49 (Step 2.) *Let op_W be a* write(v) *operation invoked by a client c_k at time $t_B(op_W) = t$ then at time $t + \delta$ there are at least $n - 2f > \#reply_{CUM}$ non faulty servers s_i such that $v \in W_i$ (so that when s_i invokes* conCut(V_i, V_{safe_i}, W_i) *v is returned).*

Proof When the WRITE() message is delivered by non faulty servers s_i, such message is stored in W_i and a timer associated to it is set to 4δ, after that the value expires. For Lemma 11 in the $[t, t+\delta]$ time interval there are maximum $2f$ Byzantine servers. All the remaining $n - 2f$ non faulty servers execute the correct protocol code, Figure 6.34 line 5 inserting v in W_i. Since write() operations are sequential, during $[t, t + \delta]$ there is only one new value inserted in W_i, which is returned by the function conCut() by construction. $\square_{Lemma\ 49}$

For simplicity, for now on, given a write() operation op_W we call $t_B(op_W)+\delta = t_{wC}$ the **completion time** of op_W, the time at which there are at least $\#reply_{CUM}$ servers storing the value written by op_W.

Lemma 50 (Step 3.) *Let op_W be a* write() *operation and let v be the written value and let t_{wC} be its time completion. Then if there are no other* write() *operation, the value written by op_W is stored by all correct servers forever (i.e., $v \in$ conCut(V_i, V_{safe_i}, W_i)).*

[9] We prove hereafter that $\gamma \leq 4\delta$, but to prove it we have first to prove that the maintenance() lasts 2δ time.

Proof From Lemma 41 at time t_{wC} there are at least $n - 2f > \#reply_{CUM}$ non faulty servers s_j such that $v \in W_i$. For sake of simplicity let us consider Figure 6.36. Let us consider that:

- for Lemma 49, all non faulty servers s_i have v in W_i at most at t_{wC};
- when s_i runs the next maintenance(), at the end of such operation, v is returned by select_three_pairs_max_sn($echo\_vals_i$) function and since it is the value with the highest sequence number (there are no other write() operation) then v is inserted in V_{safe_i} (cf. Figure 6.33 line 10), thus such value is present in the ECHO() message replies for the next 2δ time;
- this is trivially true up to time $t' = t + 4\delta$, for the timer associated to each v in W_i. In $[t, t']$ there are $2k+1$ Byzantine servers, thus $v \in W_j$ at $n-(2k+1)$ non faulty servers, and $n - (2k + 1) = (3k + 1)f + 1 = \#reply_{CUM} \geq \#echo_{CUM}$;
- for each non faulty server the next maintenance() operation op_M can happen either in $[t', t' + \delta]$ or in $[t' + \delta, t' + 2\delta]$ (cf. Figure 6.36)s_{10} and s_{11} respectively:
  - $t_B(op_M) \in [t', t' + \delta]$ (cf. s_{10} Figure 6.36): s_{10} starts op_{M_1} before $t' + \delta$, let us name it server type A. This means that $t_B(op_{M_{-1}}) + \delta < t' - \delta$, thus for Lemma 48, at the end of the operation $v \in V_{safe_{10}}$ and during op_{M_1} $v \in V_{10}$;
  - $t_B(op_M) \in [t' + \delta, t' + 2\delta]$ (cf. s_{11} Figure 6.36): s_{11} starts op_{M_1} after $t' + 2\delta$ let us name it server type B. This means that $t_B(op_{M_{-1}}) + \delta > t'$, thus at the end of the operation we can not say that $v \in V_{safe_10}$ but at least during $op_{M_{-1}}$ $v \in V_{11}$.

If all non faulty servers are type A, during op_{M_1} all non faulty servers have $v \in V$ and insert v in the ECHO() message. The same happens if all non faulty servers are type B, during $op_{M_{-1}}$, all of them inter v in the ECHO() message and the maintenance() operation terminates with such value. If the situation is mixed, then servers type B, when run $op_{M_{-1}}$, deliver ECHO() messages from both type A and type B servers. Thus if there are enough occurrence of v they can store $v \in V_{safe_b}$ and during op_{M_1} $v \in V_b$. During such operation both servers type A and type B have $vinV$. Again, if there are enough occurrences of v, the operation ends with $v \in V_{safe_b}$. It follows that servers type A, when run op_{M_1} delivers ECHO() messages containing v from both type A and type B servers. During the time interval $[t', t' + 2\delta]$ there are k correct servers that are affected by mobile agent, cf. Figure 6.36, s_5 and s_6. At the same time there is server s_0, type A, that terminate its maintenanace() with $v \in V_{safe_0}$, and thus compensates s_5, allowing s_1, type B, to terminate the maintenanace() operation with $v \in V_{safe_1}$, which compensates s_6. This cycle, between type A and type B servers can be extended forever. By hypothesis there are no more write() operation, thus all correct servers have $v \in V_{safe}$ or V, and v is returned when servers invoke function conCut(). $\square_{Lemma\ 50}$

Lemma 51 (Step 3.) *Let $op_{W_0}, op_{W_1}, \ldots, op_{W_{k-1}}, op_{W_k}, op_{W_{k+1}}, \ldots$ be the sequence of* write() *operation issued on the regular register. Let us consider a generic op_{W_k},*

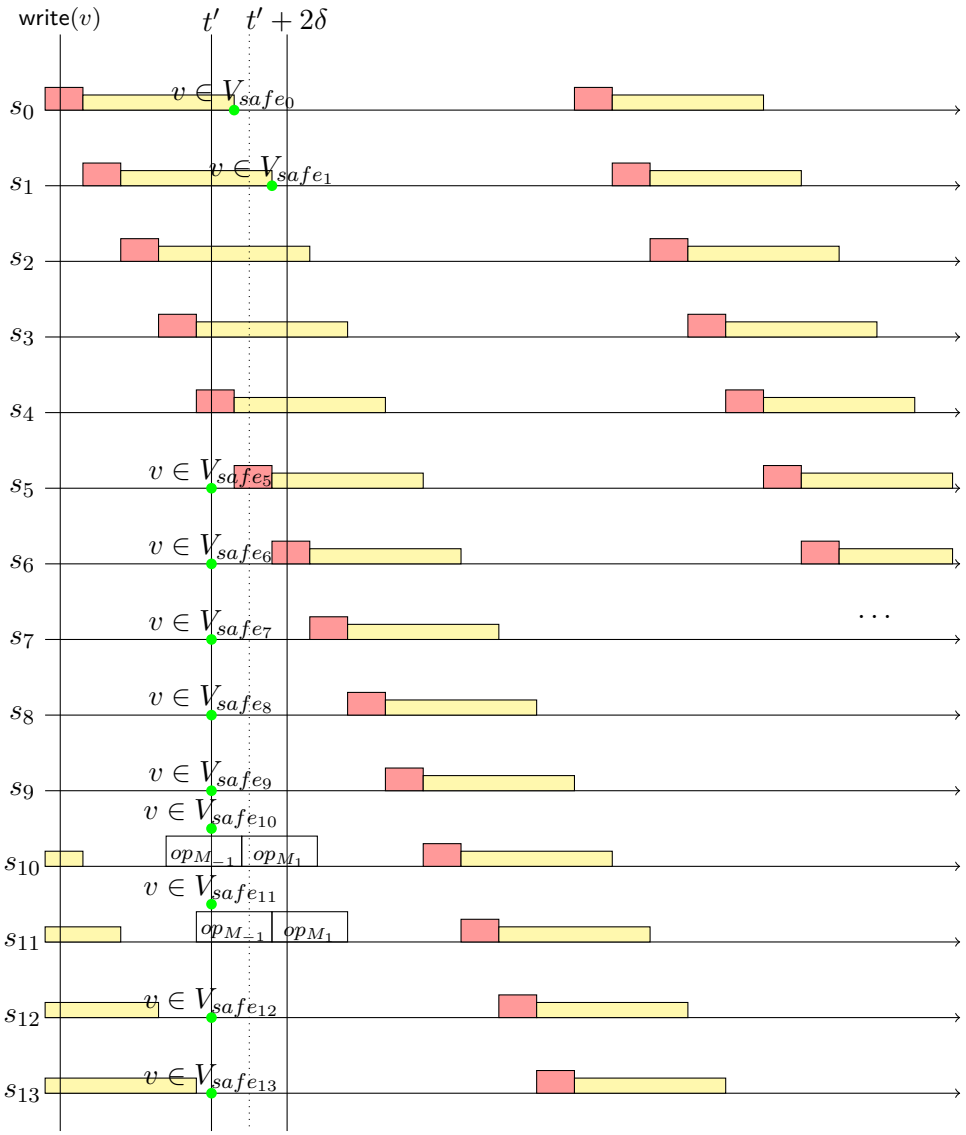


Figure 6.36. maintenance() operation op_{M_1} analysis after a write() operation, $t' = t + 4\delta$. White rectangles are maintenance() operation run by correct servers. In particular s_{10} runs such operation during the first δ period after t', while $s_1 1$ runs it during the second δ period.

let v be the written value by such operation and let t_{wC} be its completion time. Then v is in the register (there are $\#reply_{CUM}$ correct servers that return it when invoke the function conCut()*) up to time at least $t_B W_{k+3}$.*

Proof The proof simply follows considering that:

- for Lemma 50 if there are no more write() operation then v, after t_{wC}, is in the register forever.
- any new written value eventually is stored in an ordered set V_{safe} and then V(cf. Figure 6.33 line 6 or line 10) whose dimension is three.
- write() operation occur sequentially.

It follows that after three write() operations, $op_{W_{k+1}}, op_{W_{k+2}}, op_{W_{k+3}}$ in V V_{safe} and W there are three values whose sequence number is higher than the one associated to v, thus by construction conCut() does not return v anymore, v is no more stored in the regular register. $\square_{Lemma\ 51}$

Theorem 25 (Step 4.) *Any* read() *operation returns the last value written before its invocation, or a value written by a* write() *operation concurrent with it.*

Proof Let us consider a read() operation op_R. We are interested in the time interval $[t_B(op_R), t_B(op_R) + \delta]$. Since such operation lasts 2δ, the reply messages sent by correct servers within $t_B(op_R)+\delta$ are delivered by the reading client. During $[t, t+\delta]$, for Lemma 47 there are at least $\#reply_{CUM}$ correct servers that reply. Now we have to prove that what those correct servers reply with is a valid value. There are two cases, op_R is concurrent with some write() operations or not.

- op_R **is not concurrent with any** write() **operation**. Let op_W be the last write() operation such that $t_E(op_W) \leq t_B(op_R)$ and let v be the last written value. For Lemma 50 after the write completion time t_{wC} there are at least $\#reply_{CUM}$ correct servers storing v (i.e., $v \in$ conCut(V_j, V_{safe_j})). Since $t_B(op_R) + 2\delta \geq t_C w$, then there are $\#reply_{CUM}$ correct servers replying with v (cf. Lemma 47), by hypothesis there are no further write() operation and v has the highest sequence number. It follows that the last written value v is returned.
- op_R **is concurrent with some** write() **operation**. Let us consider the time interval $[t_B(op_R), t_B(op_R) + \delta]$. In such time there can be at most two write() operations. Thus for Lemma 51 the last written value before $t_B(op_R)$ is still present in $\#reply_{CUM}$ correct servers and all of them reply (cf. Lemma 47) thus at least the last written value is returned. To conclude, for Lemma 11, during the read() operation there are at most $(k+1)f$ Byzantine servers and $2k$ cured servers [10], being $\#reply_{CUM} = (3k+1)f + 1 > (3k+1)f$ then Byzantine servers may not force the reader to read another or older value and even if an older values has $\#reply_{CUM}$ occurrences the one with the highest sequence number is returned, concluding the proof. $\square_{Theorem\ 25}$

[10] Servers where affected in the previous 4δ time period, thus they are still running the two maintenance() operations, that last at most 4δ.

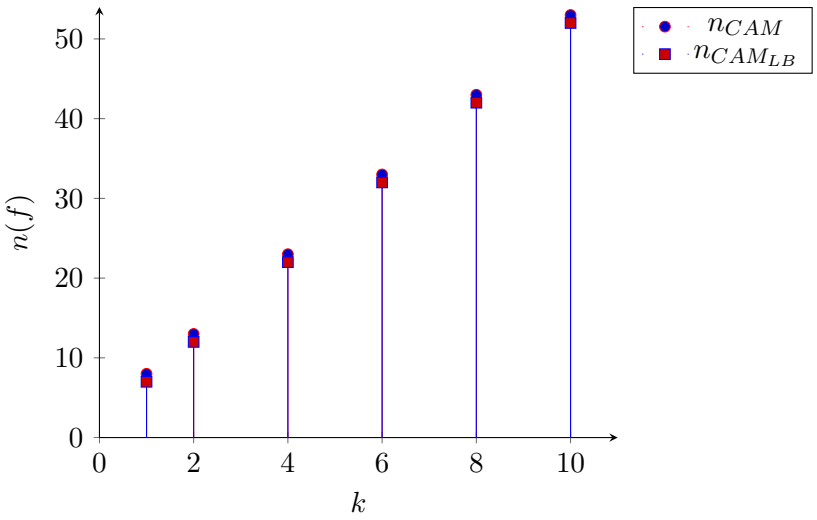


Figure 6.37. The red line is the n_{CUM} function for $k \geq 1$, $(5k+2)f+1$. The blue line is the Function $n_{CAM_{LB}}$ described in Table 6.1 with values from Table 6.2 (setting $\gamma = 4\delta$). The distance between the two lines is just 1 server.

Theorem 26 *Let n be the number of servers emulating the register and let f be the number of Byzantine agents in the (ITB, CUM) round-free Mobile Byzantine Failure model. Let δ be the upper bound on the communication latencies in the synchronous system. If $n \geq (5k+2)f+1$, then $\mathcal{P}_{reg}$ implements a SWMR Regular Register in the (ITB, CUM) round-free Mobile Byzantine Failure model.*

Proof The proof simply follows from Theorem 24 and Theorem 25. $\square_{Theorem\ 26}$

Lemma 52 *Protocol $\mathcal{P}_{reg}$ is tight in the (ITB, CUM) model with respect to $\gamma \leq 4\delta$.*

Proof The proof follows from Theorem 14 using the values in Table 6.2 to compute $n_{CUM_{LB}}$ as defined in Table 6.8. We can use such Theorem since does exists a tight protocol that solves Regular Register in the $(\Delta S, CUM)$ model so we can apply Lemma 20. From Corollary 12, $\gamma \leq 4\delta$. For $k \geq 1$, let us consider graphic depicted in Figure 6.37, the two functions are depicted for k increasing, proving that the bound for the protocol is just above, by one server, over the lower bound. For $\Delta \geq 3\delta$ is it enough to substitute values in Table 6.2 to compute $n_{CUM_{LB}}$ concluding the proof. $\square_{Lemma\ 52}$

6.8 Concluding remarks

In the following tables are reported part of the results found so far. For simplicity we are reporting particular cases for $\delta \geq \Delta$ considering ΔS and ITB movement models.

CAM	ΔS $T_r = 2\delta$ $\gamma \leq \delta$	ITB $T_r = 2\delta$ $\gamma \leq 2\delta$
$\delta \leq \Delta < 2\delta$	$n \geq 4f + 1$	$n \geq 4f + 1$
$2\delta \leq \Delta < 3\delta$	$n \geq 5f + 1$	$n \geq 6f + 1$

CAM	ΔS $T_r = 3\delta$ $\gamma \leq \delta$
$\Delta \geq 3\delta$	$n \geq 3f + 1$

CUM	ΔS $T_r = 2\delta$ $\gamma \leq 2\delta$	ITB $T_r = 2\delta$ $\gamma \leq 4\delta$
$\delta \leq \Delta < 2\delta$	$n \geq 5f + 1$	$n \geq 7f + 1$
$2\delta \leq \Delta < 3\delta$	$n \geq 8f + 1$	$n \geq 12f + 1$

CUM	ΔS $T_r = 3\delta$ $\gamma \leq 2\delta$	ITB $T_r = 2\delta$ $\gamma \leq 4\delta$
$3\delta \leq \Delta < 4\delta$	$n \geq 4f + 1$	$n \geq 6f + 1$
$4\delta \leq \Delta < 5\delta$	$n \geq 4f + 1$	$n \geq 5f + 1$
$\Delta \geq 5\delta$	$n \geq 4f + 1$	$n \geq 4f + 1$

It is interesting to notice that for $\Delta \geq 3\delta$ the Register protocol lower bounds match the lower bounds imposed by the maintenance() operation, respectively $n \geq 3f + 1$ for the $(\Delta S, CAM)$ model (cf. Lemma 8) and $n \geq 4f + 1$ for the $(\Delta S, CUM)$ model (cf. Lemma 9). Thus, those bounds are optimal in terms of correct replicas with respect to the optimal maintenance() operation. Interestingly those bounds match also the lower bounds presented in the Round-Based MBF models (Chapter 5). In particular the $n \geq 3f + 1$ lower bound for the $(\Delta S, CAM)$ model matches the Garay's model lower bound, where cured servers are aware about their failure state as for the $(*, CAM)$ model. On the other side, the $n \geq 4f + 1$ lower bound for the $(\Delta S, CUM)$ model matches the Bonnet's model (and Sasaki's model) lower bound, where cured servers are not aware about their failure state as for the $(*, CUM)$ model.
Concerning the (ITB, CAM) models, as we stated, we conjecture that, even for $\Delta \geq 3\delta$ there exists no protocol solving the Regular Register with less than $n \geq 4f+1$ replicas. On the other side, for the (ITB, CUM) models we proposed a protocol solving the regular register for $n \geq 4f + 1$ when $\Delta > 5\delta$. Thus, also in this case the protocol for the maintenance() operation is optimal in terms of correct replicas.

Chapter 7

Approximate Agreement in the Round Based Model

In this chapter we address Approximate Agreement problem in the Mobile Byzantine Failure model. Our contribution is three-fold. First, refined the problem specification to adapt it to the Mobile Byzantine Failure environment. Then, we propose the *the first mapping* from the existing variants of Mobile Byzantine models to the Mixed-Mode faults model.This mapping further help us to prove the correctness of class MSR (Mean-Subsequence-Reduce) algorithms in our context and is of independent interest. We also prove *lower bounds* for solving Approximate Agreement under all existing Mobile Byzantine faults models

7.1 Mobile Byzantine Approximate Agreement specification.

The Byzantine Approximate Agreement problem has been accurately specified in [28]. Here, we adapt such specification to the case of Mobile Byzantine Failures. Informally, in the Byzantine Approximate Agreement, each process starts proposing a real-value input and eventually every correct process decides a real-valued output; given any two correct processes, their decided values can differ for at most ϵ, where ϵ is the tolerance in the approximation. When considering Mobile Byzantine Failures we have that each process p_i can switch between faulty and correct states several times during the computation. Thus, we need to extend the specification of Byzantine Approximate Agreement to Mobile Byzantine Approximate Agreement in order to take into account such aspect. The key point in the extension is to specify that multiple decisions can be taken by the same process due to the fact that its failure state is not permanent. However, every decision taken by a process p_i while it is correct must be "consistent" with the others (i.e., the decided value should be at most ϵ far from values decided by others correct processes). More formally:

- Eventual – Convergence: There exists a round r such that, for every round $r' > r$, every correct process in r' decides a value.

- $\epsilon -$ Agreement: Let v_i and v_j be two values decided respectively by p_i and p_j when they are correct, then v_i and v_j are within ϵ of each other i.e., $|v_i - v_j| \leq \epsilon$;

- Validity: Let V be the set of initial values proposed by correct processes at round r_0 and let v_{min} and v_{max} be respectively the minimum and the maximum value in V. Let v_i be a value decided by process p_i when it is correct then v_i must be in the range $[v_{min}, v_{max}]$.

7.2 Lower Bounds

In order to formulate the strongest impossibility results related to Approximate Agreement in the Mobile Byzantine faults model we examine a weaker version of this problem referred in [19] as *Simple Approximate Agreement*. Each correct node has a real value from $[0, 1]$ as input and chooses a real value. Correct behaviors must satisfy the following properties: *Agreement:* The maximum difference between values chosen by correct nodes must be strictly smaller than the maximum difference between the inputs, or be equal to the latter difference if it is zero. *Validity*: Each correct node chooses a value in the range of the inputs of the nodes.

We prove lower bounds for each Mobile Byzantine faults models: Garay's (M1), Bonnet's(M2), Sasaki's (M3) and Burhman's (M4). The bounds for the models (M3) and (M4) result from the classical bounds proved in [19] and the mapping defined in 7.3.3. In the case of models (M1) and (M2), since the behavior of cured processes cannot be totally controlled by the Byzantine adversary, specific proofs are needed. Note that the lower bounds below do not concern the class of algorithms whose computations end before the end of the first round and that start in a configuration where there are f Byzantine processes and no *cured* ones. It is trivial that for this class of algorithms the lower bounds are the same as those proven in [19] (i.e., $n \geq 3f + 1$).

Theorem 27 (Lower bound for Garay's model) *There is no algorithm that solves Simple Approximate Agreement in the Garay's model (M1) under the Mobile Byzantine faults model if* $n \leq 4f$.

Proof The proof goes by contradiction. Suppose that there exists an algorithm $\mathcal{A}$ verifying the Simple Approximate Agreement properties in the (M1) Mobile Byzantine faults model with $n \leq 4f$. Consider w.l.g. a system with four processes and one Byzantine mobile agent. The generalization of the proof can be done by replacing any process with a group of f processes.

Consider the system with four processes denoted p_0, p_1, p_2, p_3 and consider that $p0$ is occupied by the Byzantine agent while p_1 is cured and p_2 and p_3 are correct processes. Note that the cured process in (M1) model is silent. Consider three executions of $\mathcal{A}$ denoted $E1$, $E2$ and $E3$ constructed as follows. In $E1$ the correct processes propose both the value 0. It follows, from the Agreement and Validy properties of $\mathcal{A}$, that the value chosen by p_1, p_2 and p_3 should be 0 (independently of the value sent by the Byzantine process, assume it 1). In $E2$ the correct processes propose both 1. It follows, from the Agreement and Validity properties of $\mathcal{A}$, that the

value chosen by p_1, p_2 and p_3 is 1 (independently of the value sent by the Byzantine process, assume it 0).

The $E3$ brings the contradiction: some correct processes choose 1 while others choose 0, which contradicts the Agreement property of $\mathcal{A}$. The execution $E3$ is as follows: the process occupied by the Byzantine agent sends 0 to process $p2$ and 1 to process $p3$. Let us consider only the processes p_2 and p_3. The multiset held by p_2 is {0,0,1}. This multiset is identical with the one p_2 gathered in $E1$, hence its choice in $E3$ should be 0 (identical to the one in E1). The multiset gathered by p_3 in $E3$ is {1,0,1} and identical with the one p_3 gathered in $E2$. Thus, p_3 should choose 1 in $E3$. Execution $E3$ violates the Agreement property of Simple Approximate Agreement. This contradicts the assumption that $\mathcal{A}$ verifies the Simple Approximate Agreement properties. $\square_{Theorem\ 27}$

Theorem 28 (Lower bound for Bonnet's model) *There is no algorithm that solves Simple Approximate Agreement in the Bonnet's model (M2) under the Mobile Byzantine faults model if* $n \leq 5f$.

Proof The proof follows the same general idea as the proof of Theorem 27. Suppose that exists an algorithm $\mathcal{A}$ verifying Simple Approximate Agreement properties in Mobile Byzantine model (M2) with $n \leq 5f$. In all of them we consider five processes p_0, p_1, p_2, p_3 and p_4, where p_0 is occupied by a Byzantine agent while p_1 is cured (its state may be corrupted) and p_2, p_3 and p_4 are correct processes.

Consider three executions: $E1$, $E2$ and $E3$. Execution $E1$ starts in a configuration where p_2, p_3 and p_4 propose 0 while p_1 proposes 1. Assume p_0 sends 1 to all processes. Each non faulty process gathers in $E1$ the multi-set {1,1,0,0,0} and following the Agreement and Validity properties of $\mathcal{A}$, they have all to choose 0 in $E1$.

Execution $E2$ starts in a configuration where p_2, p_3 and p_4 propose 1 while p_1 proposes 0. Assume p_0 sends 0 to all processes. Each non faulty process gathers in $E2$ the multi-set {0,0,1,1,1} and following the Agreement and Validity properties of $\mathcal{A}$, they have all to choose 1 in $E2$.

Execution $E3$ brings the contradiction. Assume that in $E3$ p_0 sends 0 to p_2 and 1 to p_3. $p2$ gathers the multiset {1,1,0,0,0} hence it has the same multi-set as in $E1$. $p2$ then chooses 0. $p3$ gathers the multi-set {0,0,1,1,1} and since this multi-set is identical with the one gathered in $E2$, $p3$ has to make the same choice, namely 1. Execution $E3$ violates the Agreement property, hence $\mathcal{A}$ do not implement the Simple Approximate Agreement. $\square_{Theorem\ 28}$

Theorem 29 (Lower bound for Sasaki's model) *There is no algorithm that solves Simple Approximate Agreement in the Sasaki's model (M3) under the Mobile Byzantine faults model if* $n \leq 6f$.

Proof The proof follows directly from the lower bound for the Simple Approximate Agreement [19] and the mapping defined in 7.3.3. Note that in the Sasaski's model the number of processes with asymmetric behavior is $2f$ where f is the number of Byzantine agents. $\square_{Theorem\ 29}$

Theorem 30 (Lower bound for Burhman's model) *There is no algorithm that solves Simple Approximate Agreement in the Burhman's model (M4) under the Mobile Byzantine faults model if* $n \leq 3f$.

Proof The proof follows directly from the lower bound for Simple Approximate Agreement [19] and the mapping defined 7.3.3. Note that in the Burhman's model in each round there are exactly f asymmetric faulty processes. $\square_{Theorem\ 30}$

7.3 Upper Bounds

In this chapter, we prove that the family of *Mean-Subsequence-Reduce* (MRS) algorithms is able to solve the Mobile Byzantine Approximate Agreement Problem. In order to to that, we will show a mapping between each MBF model presented in Section 7.3.3 and the mixed-fault model considered in [22] where MSR have been proved to work for a certain mix of Byzantine failure types.

In the following, we first introduce the mixed-fault model presented in [22], then we provide some background notions and formalization about MRS and then we will show the mapping.

7.3.1 Mixed-fault Model

In [22], three particular categories of failures have been considered: (i) *benign*, (ii) *symmetric* and (iii) *asymmetric*.

- A process p_i is said to be *benign faulty* if it exposes a self-incriminating, or immediately self-evident fault to all non-faulty processes. An example of benign fault is a crash failure or an omitted reply in a synchronous system. Indeed, given the knowledge about upper bounds on latencies in synchronous systems, such behaviors can be immediately detected by every non-faulty process. .
- A process p_i is said to be *symmetrically faulty* if its behavior is perceived identically by all non-faulty processes. A symmetric fault is generally a malicious fault such as unexpected message broadcast to all processes.
- A process p_i is said to be *asymmetrically faulty* if its behavior may be perceived differently by different non-faulty processes. An asymmetric fault is a classical arbitrary fault such as a broadcast where the sender can send different values to different correct processes.

7.3.2 Background on Mean-Subsequence-Reduce Algorithms

Convergent voting algorithms represent a family of algorithms that can be used to solve the Byzantine Approximate Agreement problem. *Convergent voting algorithms* start from an initial set of proposed values $\{v_1, v_2, \ldots v_n\}$ and guarantee that any process p_i converges to a value v_i satisfying the Byzantine Approximate Agreement specification. In [22] *convergent voting algorithms* are called *Mean-Subsequence-Reduce* (MSR).

More in details, any algorithm in this family proceeds in rounds and during any round r_j, every process p_i executes the following actions:

1. *send-phase*: p_i sends its "voted" value to the others;
2. *received-phase*: p_i aggregates values in a multiset N_{r_k};
3. *computation-phase*: p_i applies a deterministic function $\mathcal{F}(N_{r_k})$ to decide the value to vote in the next round r_{k+1}.

Their computation function can be expressed in the general form:

$$\mathcal{F}_{MSR}(N_{r_k}) = \mathsf{mean}[Sel(Red(N_{r_k}))]$$

where Sel is a selection function and Red is a reduction function used to filter values.

The correctness of MSR algorithms in the Mixed-mode faults model is guaranteed by the *single-step convergence* property. Informally, at the end of each round r_k, the range of values voted by correct processes shrinks with respect to the beginning of the round.

In [22], the authors proved that, given the number of benign faults b, the number of symmetric faults s and the number of asymmetric faults a, the minimum number of processes n needed to solve the Byzantine Approximate Agreement by an algorithm in the class MSR is

$$n > 3a + 2s + b \tag{7.1}$$

7.3.3 Mapping MBF on to Mixed-Fault Model

In order to prove that MSR algorithms are able to solve the Mobile Byzantine Approximate Agreement problem, we will map each Mobile Byzantine Failure model to the Mixed-mode faults and we will exploit the constraint on n established in the Mixed-fault model to compute the number of processes required to solve Mobile Byzantine Approximate Agreement problem in the Mobile Byzantine Failure model.

Note that the behavior of Mobile Byzantine processes concern only the send/receive phases of MSR algorithms. Therefore, we focus on the behavior of the faulty processes during the execution of these phases. In order to match our models the send-phase of MSR algorithms should be sightly modified in order to prevent correct processes to participate to the communication as per the requirement of the **M1** model.

Lemma 53 *Let $\mathcal{T}b_{r_k}$ be the set of cured processes at the beginning of round r_k in model* **M1**. *If the send phase*

if (*cured*) nop; **else** *send*(*vote*) *to all processes*;

is executed by any $p_j \in \mathcal{T}b_{r_k}$ then the computation executed in round r_k is equivalent to the computation under Mixed-mode fault model with $a = f$ and $b = |\mathcal{T}b_{r_k}|$.

Proof A cured process, in **M1** is aware of its failure state thus if it is forced to skip the send phase then it is detected by any correct process in round r_k. $\square_{Lemma\ 53}$

Lemma 54 *Let $\mathcal{T}s_{r_k}$ be the set of cured processes at the beginning of round r_k in model* **M2**. *If the send phase*

$$send(vote)\ to\ all\ processes;$$

is executed by any $p_j \in \mathcal{T}s_{r_k}$ then the computation executed in round r_k is equivalent to the computation under Mixed-mode fault model executed with $a = f$ and $s = |\mathcal{T}s_{r_k}|$.

Proof A cured process in **M2** is not aware of its state, hence it sends its vote to every process in the system. This value may be the result of a corrupted state. This is identical to the behavior of a process exhibiting a symmetric fault. $\square_{Lemma\ 55}$

Lemma 55 *Let $\mathcal{T}a_{r_k}$ be the set of cured processes at the beginning of round r_k in model* **M3**. *If send phase*

$$send(vote)\ to\ all\ processes;$$

is executed by any $p_j \in \mathcal{T}a_{r_k}$ then the computation executed in round r_k is equivalent to the computation under Mixed-mode fault model executed with $a = f + |\mathcal{T}a_{r_k}|$.

Proof A cured process in **M3** is not aware of its state hence it sends its vote to every process in the system. Moreover, Byzantine agent prepares the outgoing message queue (cf. [40]). Thus, a cured process executes the sending phase as any correct process. However, differently from the correct processes it sends possibly different values (left behind by the Byzantine agent) to every process in the system. This is identical to the behavior of a process exhibiting an asymmetric fault. $\square_{Lemma\ 55}$

Lemma 56 *Let $\mathcal{T}c_{r_k}$ be the set of cured processes at the beginning of round r_k in model* **M4**. *If the send phase*

$$send(vote)\ to\ all\ processes;$$

is executed by any $p_j \in \mathcal{T}c_{r_k}$ then the computation executed in round r_k is equivalent to the computation under Mixed-mode fault model executed with $a = f$.

Proof In this failure model, Byzantine agents move along with the messages. Thus during the sending phase there are no processes in $\mathcal{T}c_{r_k}$. $\square_{Lemma\ 56}$

Table 7.1 summarizes the mapping results proven in Lemmas 53-56.

Given Lemmas 53-56 and considering equation (1), it is possible to define the number n of processes required to run a MSR algorithm. Results are shown in Table 7.2 for each Mobile Byzantine Failure model.

In the previous Section we shown that each of the four round based MBF models described in Chapter 4 can be mapped in a particular configuration of the Mixed-fault model. Let us note that such mapping holds if we take a snapshot of the

	M1	M2	M3	M4
Asymmetric	faulty	faulty	faulty, cured	faulty
Symmetric		cured		
Benign	cured			

Table 7.1. Mapping between the behavior of faulty processes in the Mixed-Mode faulty model and faulty and cured processes in the four Mobile Byzantine faulty models.

	n_{Mi}
M1	$n > 3f + b = 4f$
M2	$n > 3f + 2s = 5f$
M3	$n > 3(f + a) = 6f$
M4	$n > 3f = 3f$

Table 7.2. Number of required replicas in each failure model.

computation at the beginning of each round. In the following, we will prove that the mobility does not affect the mapping as moving from one round to the following does not alter the proportion of processes in the mixed-fault model. This will allow us to prove that in presence of mobile Byzantine agents the MSR family of algorithms verifies the Byzantine Approximate Agreement specification.
In order to do that, we first characterize configurations produced by a MSR algorithm in presence of static Byzantine faulty nodes. Then, we prove that each configuration produced in presence of mobile Byzantine agents has the same characterization. Hence, the mobility of Byzantine agents does not affect the correctness of MSR family. Moreover, we prove that the necessary condition over the number of replicas in [22] still holds in the Mobile Byzantine failures model with the mapping defined in the previous section.

7.3.4 Preliminaries and Basic Notation

In order to proceed with our proof, we first need to recall some basic notations from [15, 22]. Let V be a set of values:

- $min(V)$: is the minimum value of the elements in V;
- $max(V)$ is the maximum value of the elements in V;
- $\rho(V)$: (also called *range* of V) is the interval of real spanned by V (i.e., $\rho(V) = [min(V), max(V)]$);
- $\delta(V)$: (also called *diameter* of V) is the difference between the maximum and the minimum values of V (i.e., $\delta(V) = min(V) - max(V)$);
- $N^i_{r_k}$: is the multi-set of values received by a non-faulty process p_i in a given round r_k. Let $U \subseteq N^i_{r_k}$ be the subset of values generated by non-faulty processes [1].

[1] Since the communication graph is fully connected then this set is equal for any correct process

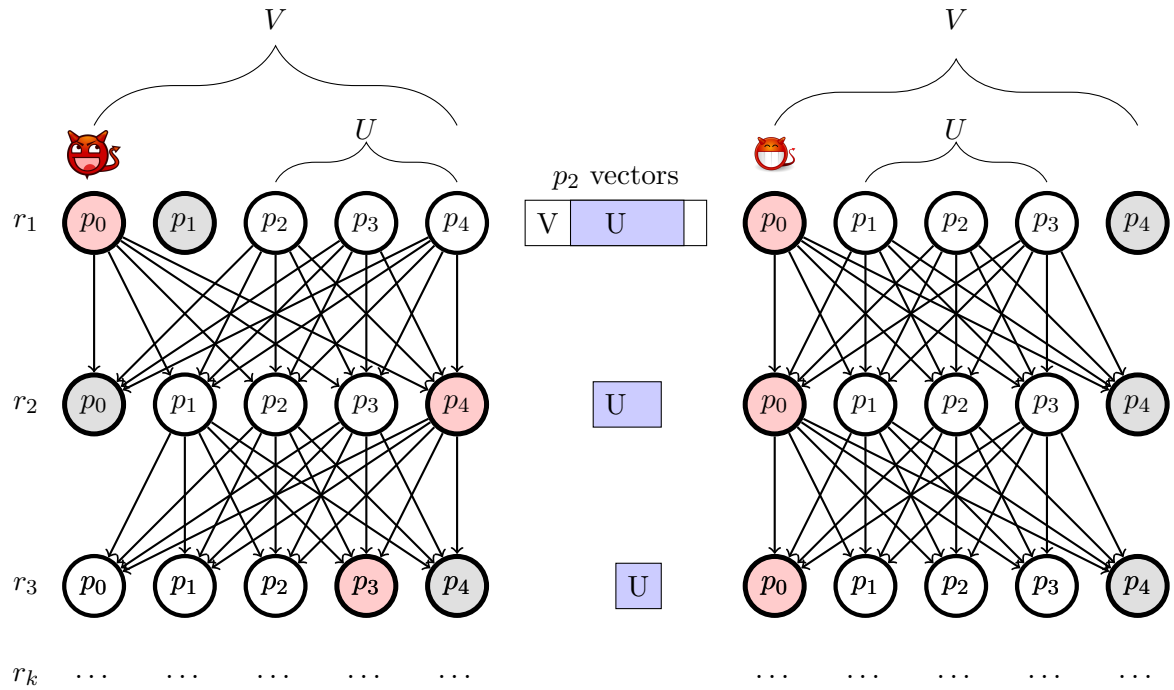


Figure 7.1. On the left **M1** model and on the right the "Mixed-Mode" Failure model. Processes are colored according to the mapping defined in 7.3.3. V and U are the proposed values sets by all and by correct processes respectively. In both cases, round after round U is shrinking and the computation is carried out by the same fraction of correct processes. What change is that in the first case they change identifiers over the rounds.

In addition, we need to recall the two fundamental properties that allows to prove the correctness of the MSR algorithms family. If $n > 3a + 2s + b$ then the following two properties hold:

Property 1 *For each non-faulty process p_i, the value computed at the end of round r_k is in the range of non-faulty values, i.e.,*

$$\mathcal{F}_{MSR}(N^i_{r_k}) \in \rho(U).$$

Property 2 *For each pair of non-faulty processes p_i and p_j, the difference between their computed values is strictly less than the diameter of the sub-multiset of non-faulty values received, i.e.,*

$$|\mathcal{F}_{MSR}(N^i_{r_k}) - \mathcal{F}_{MSR}(N^j_{r_k})| < \delta(U).$$

In the following $v^i_{r_k}$ denotes the value obtained at the end of round r_k (computation phase) by process p_i, applying the MSR function vector $N^i_{r_k}$ (i.e., $v^i_{r_k} \leftarrow \mathcal{F}_{MSR}(N^i_{r_k})$) and we will refer to such value as *correct value*.

Lemma 57 *Let $\mathcal{T}*_{r_k}$ be the set of cured processes at the beginning of round r_k in the models* **M1-M4**. *If $n > n_{Mi}$ and every $p_j \in \mathcal{T}*_{r_k}$ executes computation-phase of a MSR-algorithm then at the end of r_k we have $|\mathcal{T} *_{r_k}| = 0$.*

Proof The proof is done by induction. During the first round r_0 no Byzantine agent moved yet. Thus, at the end of r_0 trivially $|\mathcal{T} *_{r_0}| = 0$. In the next round r_1 Byzantine agents move thus affecting up to f processes. Therefore, at the beginning of r_1 there are up to f cured processes, $|\mathcal{T} *_{r_1}| \leq f$. If we substitute, for each model **M1-M4** (cf. Table 7.1), values in $n > 3a + 2s + b$ if follows that despite agents movement, $n > n_{M_i}$ still holds. Thus, for the definition of $\mathcal{F}_{MSR}()$ the value that each process computes at *computation-phase* is correct. Hence, at the end of round r_1 we have $|\mathcal{T} *_{r_1}| = 0$. For each further r_k the reasoning is similar. $\square_{Lemma\ 57}$

From Lemma 57 it follows that during each round there are not cured processes related to the previous round but only the ones due to the last Byzantine agents movement, hence the corollary below.

Corollary 13 *Let $\mathcal{T}_{r_k}$ be the set of cured processes at the beginning of round r_k. $\forall r_k$, $|\mathcal{T}_{r_k}| \leq f$.*

Definition 20 (configuration C_{r_k}) *Let configuration C_{r_k} be a set of n tuples $\langle$failure state, proposing value$\rangle_i$ representing the state of each process p_i at round r_k. Note that processes, depending on the failure model, may or may not be aware of their failure state.*

Definition 21 (AA_{r_k}) *Let $\mathcal{AA}$ be a generic instance of the MSR family and let AA_{r_k} be the $r_k - th$ execution of the protocol $\mathcal{AA}$ at round r_k, such that $C_{r_k} \leftarrow AA_{r_k}(C_{r_k-1})$. It takes as input C_{r_k-1} and returns C_{r_k}.*

Definition 22 (static computation) *A sequence of k $\mathcal{AA}$ executions, such that $C_{r_k} \leftarrow AA_{r_k-1}(AA_{r_k-2}(\dots\ AA_{r_1}(C_{r_0}))\dots)$ is said a static computation if in every configuration $C_{r_1}, ..., C_{r_k}$, there exists a subset of at least $n - (3a + 2s + b)$ correct processes that are correct during the whole computation.*

Note that with fixed a,s and b, the relation $n > 3a + 2s + b$ always holds in a static computation of a MSR algorithm ([22]).

Definition 23 (mobile computation) *A sequence of k $\mathcal{AA}$ executions, such that $C_{r_k} \leftarrow AA_{r_k-1}(AA_{r_k-2}(\dots\ AA_{r_1}(C_{r_0}))\dots)$ is said to be a mobile computation if for any two subsequent configurations C_{r_k}, C_{r_k+1}, any process may change the failure state but the relation $n > 3a + 2s + b$ holds at each round.*

Definition 24 (configurations equivalence) *A configuration C_{r_k} is said to be equivalent to a configuration $\bar{C}_{r_k}$ if:*

- *C_{r_k} and $\bar{C}_{r_k}$ produce the same U;*
- *$\forall k$, C_{r_k} has at least the same number of tuples $\langle$correct, correct value $\rangle$ as $\bar{C}_{r_k}$.*

Note that in a static computation a correct process is correct for the whole computation, while in a mobile one is correct with respect to the observed round.

Definition 25 (correct computation) *A computation $C_{r_0}, \dots, C_{r_k}$ is a correct computation if it is possible to build a static computation $\bar{C}_{r_0}, \dots, \bar{C}_{r_k}$ such that, $\forall j \in [0, k]$, C_{r_j} is equivalent to $\bar{C}_{r_j}$.*

Given a static computation $\bar{C}_{r_0}, \ldots, \bar{C}_{r_k}$ of an algorithm in the MSR class, if $n > 3a + 2s + b$, then each configuration $\bar{C}_{r_j}, j \in [0, k]$, is characterized as follows:

- up to a asymmetric Byzantine processes;
- up to s symmetric Byzantine processes;
- up to b benign faults;
- at least $n - (a + s + b)$ correct processes such that each p_j of them computes a correct value $v_j^{r_j}$.

The first three points are due to the failures static nature. The last one is given by the failures static nature plus the correctness of the algorithm in the static case (as proven in [22]).

7.3.5 MSR correctness under Mobile Byzantine fault model

In the following we prove that despite the mobility of Byzantine agents, the MSR family of algorithms satisfies the Mobile Byzantine Approximate Agreement specification. In the presence of mobile Byzantine agents, each round is characterized by correct, cured and faulty processes. As we showed previously, depending on the failure model considered, cured processes behave accordingly to a different kind of fault (asymmetric, symmetric or benign). Figure 7.1 presents an example of execution of a MSR algorithm in presence of Mobile Byzantine Failures (left side) and Mixed-Mode Failures (right side) and informally shows that at the beginning of each round we obtain the same configuration that satisfy the properties required for the convergence.

The following theorem proves the mapping between the Mobile Byzantine faults model and the Mixed-mode fault model. Let us start proving that if $n > n_{Mi}$ then a mobile computation is also a correct computation, as defined in subsection 7.3.4.

Theorem 31 *Let us consider a mobile computation $C_0, C_1, \ldots, C_k, \forall k \in \mathbb{N}$ of an algorithm $\mathcal{AA}$ in the class MSR. If in each round $n > n_{Mi}$ (cf. Table 7.2) then the sequence $C_0, \ldots, C_k$ is a correct computation.*

Proof We have to show that for each iteration of $\mathcal{AA}$ we can build a static computation equivalent to the dynamic one. The proof is done by induction. Let us denote by $\mathcal{C}$, $\mathcal{T}*$ and $\mathcal{B}$ the set of correct, cured and Byzantine processes respectively and let t_* denote the cardinality of $\mathcal{T}*$. Let us denote, in the static case, by $\mathcal{C}'$, $\mathcal{T}'$, and $\mathcal{B}'$ the set of correct, non correct (which may be asymmetric, symmetric, or benign), and asymmetric faulty processes, respectively, and let t'_* denote the cardinality of $\mathcal{T}'$.

- Rounds $0 \rightarrow 1$: At the begining of round 0, Byzantine agents never move. Thus, the configuration is as follows:
  - $\mathcal{C}$: $\forall i \in \mathcal{C}, \langle correct, v_i^{init} \rangle_i$, $|\mathcal{C}| \geq n - (f)$;
  - $\mathcal{B}$: $\forall j \in \mathcal{B}, \langle faulty, \perp\ ^2 \rangle_j$, $|\mathcal{B}| \leq f$.

[2] We use $\perp$ to indicate that it can be any value

The protocol executes its first iteration. Processes exchange their value and each non Byzantine process p_i updates its state: $\langle$failure state, proposing value $\leftarrow v_i^0 = \mathcal{F}_{MSR}(V^0)\rangle$. At this point the situation is as follow:

- $\mathcal{C}$: $\forall i \in \mathcal{C}, \langle correct, v_i^0\rangle_i$, $|\mathcal{C}| \geq n - (f)$;
- $\mathcal{B}$: $\forall j \in \mathcal{B}, \langle faulty, \perp\rangle_j$, $|\mathcal{B}| \leq f$.

Up to now, the same happens in a static computation. At the begining of round 1, at most f Byzantine agents move affecting other processes. Thus there are up to $t_* = f$ cured processes storing a non correct value (e.g., $v^0 \notin \rho(N^0)$).

- $\mathcal{C}$: $\forall i \in \mathcal{C}, \langle correct, v_i^{init}\rangle_i$, $|\mathcal{C}| \geq n - (f + t_*)$;
- $\mathcal{T}$: $\forall k \in \mathcal{T}, \langle cured, \perp\rangle_k$, $|\mathcal{T}| \leq t_*$;
- $\mathcal{B}$: $\forall j \in \mathcal{B}, \langle faulty, \perp\rangle_j$, $|\mathcal{B}| \leq f$.

At the begining of round 1, there are at least $n - (f + t_*)$ correct processes. If we map it to the Mixed-mode failures model (cf. Table 7.1), this is equivalent to a static configuration where there are f asymmetric processes and t_* non correct that may be asymmetric, symmetric or benign:

- $\mathcal{C}'$: $\forall i \in \mathcal{C}', \langle correct, v_i^{init}\rangle_i$, $|\mathcal{C}'| \geq n - (f + t'_*)$;
- $\mathcal{T}'$: $\forall k \in \mathcal{T}', \langle *, \perp\rangle_k$, $|\mathcal{T}'| \leq t'_*$;
- $\mathcal{B}'$: $\forall j \in \mathcal{B}', \langle asymmetric, \perp\rangle_j$, $|\mathcal{B}'| \leq f$.

The mobile and static configurations are equivalent. Thus the current mobile configuration (and the mobile computation up to now) is correct.

- Rounds $1 \rightarrow 2$: From the previous point, the configuration at the beginning of round 1 is correct. The second iteration of the protocol takes place. Processes exchange their value and each non Byzantine process p_i updates its state: $\langle$failure state, proposing value $\leftarrow v_i^1 = \mathcal{F}_{MSR}(N_i^1)\rangle$. At this point, for Lemma 57, each process in $\mathcal{T}*$ becomes correct. In other words, there are up to f Byzantine processes and at least $n - f$ correct processes. We are in the same situation as at the end of previous round 0.
  At the beginning of next round, at most f Byzantine agents can move to other processes, leaving up to $t_* = f$ cured processes with non correct value. Thus there are at least $n - (f + t_*)$ correct processes at the begining of round 2. The mobile and static configurations are equivalent. Thus the current mobile configuration (and the mobile computation up to now) is correct.

- Rounds $i \rightarrow i + 1$: generalizing, for each round starting with a correct configuration we can apply the previous reasoning ending in a subsequent round characterized by a correct configuration.

$\square_{Theorem\ 31}$

In the following we prove the correctness of any algorithm in the class MSR under Mobile Byzantine failure model.

Lemma 58 (Termination) *Let AA be an algorithm in the class MSR. If $n > n_{Mi}$, AA under Mobile Byzantine fault model verifies the* Eventual-Convergence *property of the Byzantine Approximation Agreement.*

Proof From Theorem 31, if $n > n_{Mi}$ then algorithm AA generates a sequence of correct configurations, i.e., a sequence of converging values exactly as in [15, 22], thus the Eventual-Convergence property is satisfied in the same way the Termination is satisfied by the [15, 22] solutions. $\square_{Lemma\ 58}$

Lemma 59 (ϵ-Agreement) *Let AA be an algorithm in the class MSR. If $n > n_{Mi}$, AA under Mobile Byzantine fault model verifies the* ϵ-Agreement *property of the Byzantine Approximation Agreement.*

Proof From Theorem 31, if $n > n_{Mi}$ then algorithm AA generates a sequence of correct configurations, i.e., a sequence of converging values exactly as in [15, 22]. Thus, the ϵ-Agreement property is satisfied in the same way this is satisfied by the [15, 22] solutions.

In the following we prove that once ϵ-Agreement is achieved among the currently non faulty processors, it is preserved among the (possible different) uninfected processors. Let us consider an arbitrarily long mobile computation $C_0, \ldots, C_k$. If ϵ-Agreement is achieved then there exists a round $r_a, a \in [0, k]$ where all non faulty processes agree on values that are ϵ close to each other. Considering that $n > n_{Mi}$ then from Theorem 31 the whole mobile computation $C_0, \ldots, C_k$ is correct. Thus from round to round the two properties $P1$ and $P2$ hold and correct processes values can not diverge from each other.

$\square_{Lemma\ 59}$

Lemma 60 (Validity) *Let AA be an algorithm in the class MSR. If $n > n_{Mi}$, AA under Mobile Byzantine fault model verifies the* Validity *property of the Byzantine Approximation Agreement.*

Proof From Theorem 31, if $n > n_{Mi}$ then algorithm AA generates a sequence of correct configurations, i.e., a sequence of converging values exactly as in the validity proof in [15, 22]. $\square_{Lemma\ 60}$

The three above lemmas provide the proof of the theorem below.

Theorem 32 *If $n > n_{M_i}$ then the class MSR verifies the Byzantine Approximate Agreement specification.*

Chapter 8

Conclusions

In this thesis we have deeply studied the Mobile Byzantine Failures model. The importance of such model arises when we have to design a Byzantine Tolerant protocol ables to tolerate f Byzantine replicas in a long lasting execution. Thus, a situation where it is more likely that the number of Byzantine replicas exceeds f, yielding to the necessity to restore the correct state of compromised replicas. In this thesis, we coped with the new challenges that such model unveils, from new definitions of failure states to the impact of the operations duration. Indeed, in such models we can not assume anymore that during an operation there are f Byzantine processes but depending on the characteristic of the system model considered we have to be able to compute how many Byzantine processes can be involved during each operation, so that also the way to prove lower bounds has been rethought.
In particular in this work we first proposed optimal solutions for the Atomic Register problem in the already defined mobile Byzantine round-based models. As we saw, solving such problem in presence of mobile Byzantine failures implies higher lower bounds with respect to solutions coping with Byzantine failure. On the other side, given the round-based nature of the model the solution is algorithmically simple. This observation advocate to our main contribution, a general round-free mobile Byzantine failure model which can be deployed in four different models. Those models describe the different processes awareness about their failure state and mobile agent movements (coordinated or uncoordinated).
We prove that in each model (round-based and consequently round-free) an additional operation, the maintenance, is necessary to implement registers. Moreover we prove that maintenance is not solvable in a system model where communications are asynchronous. Later we provided a framework to compute lower bounds in each instance of the MBF models.
Once lower bounds have been proved we solve the Regular Register problem in all the four different model variants. In those cases, register protocols, in addition to read and write operations, implement the maintenance operation to cope with mobile Byzantine failures. Interestingly, we saw that read and write operations do not really change through the four different failures models. This is due to the maintenance operation that changes from a model to another and makes transparent the mobile agents movements to the read a write operations. In short, such operation guarantees that during each operation there is a set of correct processes, large enough, to provide

a valid value to the reading client.
Finally we proposed an optimal solution for the Approximate Agreement specification in the mobile Byzantine round-based models. Interestingly, concerning the Approximate Agreement, we show how the already existing solutions can be used also in presence of mobile Byzantine failures and under which conditions. Moreover, we found out that, as in the Byzantine failures model, consensus problem and approximate agreement problem share the same lower bounds on the fraction of required correct servers.

Future works. The round-free MBF models presented in this thesis open to several future works. Fundamentally, any known problem solved in presence of Byzantine failures can be studied in presence of Mobile Byzantine failures and nevertheless is it worthy to study the maintenance operation optimality in the different MBF models. From our side, the most immediate future work concerns the Atomic Register problem, which is the last register specification left to be solved. [6] considers for the first time both transient failures [1] and Mobile Byzantine failures in the round based specification. In particular in such failures model the Atomic Register is optimally solved. This open to the study of Register problem in a model prone to both transient failures and round-free Mobile Byzantine failures.
Besides Registers, the natural consequence of the round-free MBF models definition is the study of Consensus and Approximate Agreement in such models and in particular the study of the necessary conditions to solve those problems. Contrarily to the Approximate Agreement problem, the existence of a process that is always correct is a necessary condition to solve Consensus in the round-based MBF model. Thus the first question to address is if one process always correct is enough in the round-free models or if we need more.
Regarding the MBF model itself another interesting study concerns the possible mapping between mobile Byzantine and churn. Informally, in models prone to churn there is a fraction of processes that leave the system and another fraction that join the system. Join the system means that those processes has to retrieve the state of the other correct processes, exactly what happens for cured processes running the maintenance operation.

[1] Local variables of any process can be arbitrarily modified [16]. It is nevertheless assumed that transient failures are quiescent i.e., there exists a time (unknown to the processes) after which no more transient failures are going to happen.

Bibliography

[1] The high price of it downtime. http://www.networkcomputing.com/networking/high-price-it-downtime/856595126. Accessed: 2017-01-27.

[2] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, **42** (1995), 124.

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

[4] Berman, P., Garay, J. A., and Perry, K. J. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pp. 410–415 (1989).

[5] Bonnet, F., Défago, X., Nguyen, T. D., and Potop-Butucaru, M. Tight bound on mobile byzantine agreement. *Theor. Comput. Sci.*, **609** (2016), 361. Available from: http://dx.doi.org/10.1016/j.tcs.2015.10.019, doi:10.1016/j.tcs.2015.10.019.

[6] Bonomi, S., del Pozzo, A., and Potop-Butucaru, M. Tight self-stabilizing mobile byzantine-tolerant atomic register. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pp. 6:1–6:10. ACM, New York, NY, USA (2016). ISBN 978-1-4503-4032-8. Available from: http://doi.acm.org/10.1145/2833312.2833320, doi:10.1145/2833312.2833320.

[7] Bonomi, S., Pozzo, A. D., Potop-Butucaru, M., and Tixeuil, S. Approximate agreement under mobile byzantine faults. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pp. 727–728 (2016). Available from: http://dx.doi.org/10.1109/ICDCS.2016.68, doi:10.1109/ICDCS.2016.68.

[8] Bonomi, S., Pozzo, A. D., Potop-Butucaru, M., and Tixeuil, S. Optimal mobile byzantine fault tolerant distributed storage. In *Proceedings of the ACM International Conference on Principles of Distributed Computing (ACM PODC 2016)*. ACM Press, Chicago, USA (2016).

[9] Bouzid, Z., Potop-Butucaru, M. G., and Tixeuil, S. Byzantine convergence in robot networks: The price of asynchrony. In *Principles of Distributed*

Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings (edited by T. F. Abdelzaher, M. Raynal, and N. Santoro), vol. 5923 of *Lecture Notes in Computer Science*, pp. 54–70. Springer (2009). Available from: `http://dx.doi.org/10.1007/978-3-642-10877-8_7`, `doi:10.1007/978-3-642-10877-8_7`.

[10] Bouzid, Z., Potop-Butucaru, M. G., and Tixeuil, S. Optimal byzantine-resilient convergence in uni-dimensional robot networks. *Theor. Comput. Sci.*, **411** (2010), 3154. Available from: `http://dx.doi.org/10.1016/j.tcs.2010.05.006`, `doi:10.1016/j.tcs.2010.05.006`.

[11] Buhrman, H., Garay, J. A., and Hoepman, J.-H. Optimal resiliency against mobile faults. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pp. 83–88. IEEE (1995).

[12] Cachin, C. and Tessaro, S. Optimal resilience for erasure-coded byzantine distributed storage. In *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 115–124. IEEE (2006).

[13] Charron-Bost, B., Függer, M., and Nowak, T. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pp. 528–539 (2015).

[14] Denning, D. E. An intrusion-detection model. *IEEE Transactions on software engineering*, (1987), 222.

[15] Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. Reaching approximate agreement in the presence of faults. *Journal of the ACM (JACM)*, **33** (1986), 499.

[16] Dolev, S. *Self-Stabilization*. MIT Press (2000). ISBN 0-262-04178-2.

[17] Fekete, A. D. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, **4** (1990), 9.

[18] Fekete, A. D. Asynchronous approximate agreement. *Inf. Comput.*, **115** (1994), 95.

[19] Fischer, M. J., Lynch, N. A., and Merritt, M. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, **1** (1986), 26.

[20] FISCHER, M. J. and LYNCII, N. A. A lower bound fob the time to assube intebactive consistencv. *INFORMATION PROCESSING LETTERS*, (1982).

[21] Garay, J. A. Reaching (and maintaining) agreement in the presence of mobile faults. In *International Workshop on Distributed Algorithms*, pp. 253–264. Springer (1994).

[22] Kieckhafer, R. M. and Azadmanesh, M. H. Reaching approximate agreement with mixed-mode faults. *Parallel and Distributed Systems, IEEE Transactions on*, **5** (1994), 53.

[23] Lamport, L. On interprocess communication. *Distributed computing*, **1** (1986), 86.

[24] Lamport, L. On interprocess communication. part i: Basic formalism. *DC*, **1** (1986), 77.

[25] Lamport, L. On interprocess communication. part ii: Algorithms. *DC*, **1** (1986), 86.

[26] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **4** (1982), 382.

[27] Li, C., Hurfin, M., and Wang, Y. Approximate byzantine consensus in sparse, mobile ad-hoc networks. *J. Parallel Distrib. Comput.*, **74** (2014), 2860.

[28] Lynch, N. A. *Distributed Algorithms*. Morgan Kaufmann (1996).

[29] Lynch, N. A. and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pp. 272–281. IEEE (1997).

[30] Lynch, N. A. and Tuttle, M. R. An introduction to input/output automata. *CWI Quarterly*, **2** (1989), 219. Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7751;http://www.bibsonomy.org/bibtex/2919d591f0c2a2754fc7e4aeb008d14ca/giuliano.losa`.

[31] Malkhi, D. and Reiter, M. K. Secure and scalable replication in phalanx. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pp. 51–58. IEEE (1998).

[32] Martin, J.-P., Alvisi, L., and Dahlin, M. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pp. 311–325. Springer (2002).

[33] Martin, J.-P., Alvisi, L., and Dahlin, M. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pp. 311–325. Springer (2002).

[34] Mendes, H. and Herlihy, M. Multidimensional approximate agreement in byzantine asynchronous systems. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pp. 391–400 (2013).

[35] Mendes, H., Herlihy, M., Vaidya, N. H., and Garg, V. K. Multidimensional agreement in byzantine systems. *Distributed Computing*, **28** (2015), 423.

[36] Ostrovsky, R. and Yung, M. How to withstand mobile virus attacks. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pp. 51–59. ACM (1991).

[37] Pierce, E. and Alvisi, L. A recipe for atomic semantics for byzantine quorum systems. Tech. rep., Technical report, University of Texas at Austin, Department of Computer Sciences (2000).

[38] Platania, M., Obenshain, D., Tantillo, T., Sharma, R., and Amir, Y. Towards a practical survivable intrusion tolerant replication system. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, pp. 242–252. IEEE (2014).

[39] Reischuk, R. A new solution for the Byzantine generals problem. *Information and Control*, **64** (1985), 23.

[40] Sasaki, T., Yamauchi, Y., Kijima, S., and Yamashita, M. Mobile byzantine agreement on arbitrary network. In *International Conference On Principles Of Distributed Systems*, pp. 236–250. Springer (2013).

[41] Services, I. G. Improving systems availability. Tech. rep., IBM Global Services (1998).

[42] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel & Distributed Systems*, (2009), 452.

[43] Sousa, P., Neves, N. F., and Verissimo, P. How resilient are distributed f fault/intrusion-tolerant systems? In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pp. 98–107. IEEE (2005).

[44] Sousa, P., Neves, N. F., and Verissimo, P. Hidden problems of asynchronous proactive recovery. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep'07)* (2007).

[45] Stolz, D. and Wattenhofer, R. Byzantine approximate agreement with median validity. In *to appear OPODIS'15* (2015).

[46] Su, L. and Vaidya, N. H. Reaching approximate byzantine consensus with multi-hop communication. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, pp. 21–35 (2015).

[47] Tseng, L. and Vaidya, N. H. Iterative approximate byzantine consensus under a generalized fault model. In *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pp. 72–86 (2013).

[48] Tseng, L. and Vaidya, N. H. Asynchronous convex hull consensus in the presence of crash faults. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pp. 396–405 (2014).

[49] Tseng, L. and Vaidya, N. H. Iterative approximate consensus in the presence of byzantine link failures. In *Networked Systems - Second International Conference, NETYS 2014, Marrakech, Morocco, May 15-17, 2014. Revised Selected Papers*, pp. 84–98 (2014).

[50] Vaidya, N. H., Tseng, L., and Liang, G. Iterative approximate byzantine consensus in arbitrary directed graphs. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pp. 365–374 (2012).

[51] Vitanyi, P. M. and Awerbuch, B. Atomic shared register access by asynchronous hardware. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pp. 233–243. IEEE (1986).

[52] Yung, M. The "Mobile Adversary" Paradigm in Distributed Computation and Systems. In *PODC* (edited by C. Georgiou and P. G. Spirakis), pp. 171–172. ACM (2015). ISBN 978-1-4503-3617-8. Available from: `http://dblp.uni-trier.de/db/conf/podc/podc2015.html#Yung15`.