Building Regular Registers with Rational Malicious Servers and Anonymous Clients

Silvia Bonomi, Antonella Del Pozzo and Roberto Baldoni
Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy
{bonomi, delpozzo, baldoni}@dis.uniroma1.it

Abstract

The paper addresses the problem of emulating a regular register in a synchronous distributed system where clients invoking read() and write() operations are anonymous while server processes maintaining the state of the register may be compromised by rational adversaries (i.e., a server might behave as *rational malicious Byzantine* process). We first model our problem as a Bayesian game between a client and a rational malicious server where the equilibrium depends on the decisions of the malicious server (behave correctly and not be detected by clients vs returning a wrong register value to clients with the risk of being detected and then excluded by the computation). We prove such equilibrium exists and finally we design a protocol implementing the regular register that forces the rational malicious server to behave correctly.
Keywords: Regular Register, Rational Malicious Processes, Anonymity, Bayesian Game.

1 Introduction

To ensure high service availability, storage services are usually realized 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 example 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. A compromised replica is usually modeled trough an arbitrary failure (i.e. a Byzantine failure) that is made transparent to clients by employing Byzantine Fault Tolerance (BFT) techniques. 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, this approach has a strong limitation: a smart adversary may be able to compromise more than f replicas in long executions and may get access to the entire system when the attack is sufficiently long. To overcome this issue, Sousa et al. designed the *proactive-reactive recovery* mechanism [21]. The basic idea is to periodically reconfigure the set of replicas to rejuvenate servers that may be under attack (proactive mode) and/or when a failure is detected (reactive mode). This approach is effective in long executions but requires a fine tuning of the replication parameters (upper bound f on the number of possible compromised replicas in a given period, rejuvenation window, time required by the state transfer, etc...) and the presence of secure components in the system. In addition, it is extremely costly during good periods (i.e. periods of normal execution) as a high number of replicas must be deployed independently from their real need. In other words, the system pays the cost of an attack even

if the attack never takes place. In this paper, *we want to investigate the possibility to implement a distributed shared variable (i.e. a register) without making any assumption on the knowledge of the number of possible compromised replicas*, i.e. without relating the total number of replicas n to the number of possible compromised ones f. To overcome the impossibility result of [5, 18], we assume that (i) clients preserve their privacy and do not disclose their identifiers while interacting with server replicas (i.e. anonymous clients) and (ii) at least one server is always alive and never compromised by the attacker. We first model our protocol as a game between two parties, a client and a rational malicious server (i.e. a server controlled by rational adversaries) where each rational malicious server gets benefit by two conflicting goals: (i) it wants to have continuous access to the current value of the register and, (ii) it wants to compromise the validity of the register returning a fake value to a client. However, if the rational malicious server tries to accomplish goal (ii) it could be detected by a client and it could be excluded from the computation, precluding him to achieve its first goal. We prove that an equilibrium exists for such game. In addition, we design a distributed protocol implementing the register and reaching such equilibrium when rational malicious servers privilege goal (i) with with respect to goal (ii). As a consequence, rational malicious servers return correct values to clients to avoid to be detected by clients and excluded by the computation and the register implementation is proved to be correct. The rest of the paper is organized as follows: Section 2 discusses related works, Section 3 and Section 4 introduce respectively the system model and the problem statement. In Section 5 we model our problem as a Bayesian game and in Section 6 we provide a protocol matching the Bayesian Nash Equilibrium for our game. Finally, Section 7 presents a discussion and future work.

2 Related Work

Building a distributed storage able to resist arbitrary failures (i.e. Byzantine) is a widely investigated research topic. The Byzantine failure model captures the most general type of failure as no assumption is made on the behavior of faulty processes. Traditional solutions to build a Byzantine tolerant storage service can be divided into two categories: *replicated state machines* [19] and *Byzantine quorum systems* [5, 16, 17, 18]. Both the approaches are based on the idea that the state of the storage is replicated among processes and the main difference is in the number of replicas involved simultaneously in the state maintenance protocol. Replicated state machines approach requires that every non-faulty replica receives every request and processes requests in the same order before returning to the client [19] (i.e. it assumes that processes are able to totally order requests and execute them according to such order). Given the upper bound on the number of failures f, the replicated state machine approach requires only $2f + 1$ replicas in order to provide a correct register implementation. On the contrary, Byzantine quorum systems need just a sub-set of the replicas (i.e. *quorum*) to be involved simultaneously. The basic idea is that each operation is executed by a quorum and any two quorums must intersect (i.e. members of the quorum intersection act as witnesses for the correct execution of both the operations). Given the number of failures f, the quorum-based approach requires at least $3f + 1$ replicas in order to provide a correct register implementation in a fully asynchronous system [18]. Let us note that, in both the approaches, the knowledge of the upper bound on faulty servers f is required to provide deterministic correctness guarantees. In this paper, we follow an orthogonal approach. We are going to consider a particular case of byzantine failures and we study the cost, in terms of number of correct servers, of building a distributed storage (i.e. a register) when clients are anonymous and have no information about the number of faulty servers (i.e. they do not know the bound f). In particular, the type of byzantine processes considered here deviate from the protocol by following a strategy that brings them to optimize their own benefit (i.e., they are *rational*) and such strategy has the final aim to compromise the correctness of the storage (i.e., they are *malicious*).

In [15], the author presented Depot, a cloud storage system able to tolerate any number of Byzantine clients or servers, at the cost of a weak consistency semantics called *Fork-Join-Causal consistency* (i.e., a weak form of causal consistency).

In [3], the authors introduced the *BAR (Byzantine, Altruistic, Rational) model* to represent distributed systems with heterogeneous entities like peer-to-peer networks. This model allows to distinguish between Byzantine processes (i.e. processes deviating arbitrarily from the protocol and without any known strategy), altruistic processes (i.e. processes following the protocol - correct processes) and rational processes (i.e. processes that decide to follow or not the protocol, according to their individual utility). Under the BAR model, several problems have been investigated (e.g. reliable broadcast [7], data stream gossip [13], backup service through state machine replication [3]). Let us note that in the BAR model the utility of a process is measured through the cost it supports to run the protocol. In particular, each step of the algorithm (especially sending messages) has a cost and the objective of any rational process is to minimize its global cost. As a consequence, rational processes deviate from the protocol just by skipping to send messages if they are not properly encouraged by getting back a reward (i.e., they are *selfish*). In contrast with the BAR model, rational servers considered in this paper are malicious and then they get benefit from preventing the correct protocol execution rather than from saving messages, i.e., they deviate from the protocol with a different objective.

More recently, classical one-shot problems as leader election [1, 2], renaming and consensus [2] have been studied under the assumption of rational agents (or rational processes). The authors provide algorithms implementing such basic building blocks, both for synchronous and asynchronous network, under the so called *solution preference* assumption i.e., agents gain if the algorithm succeeds in its execution while they have zero profit if the algorithm fails. As a consequence, processes will not deviate from the algorithm if such deviation interferes with its correctness. Conversely, the model of rational malicious processes considered in this paper removes implicitly this assumption as they are governed by adversaries that gets benefit when the algorithm fails while in [1, 2] rational processes get benefit from the correct termination of the protocol (i.e. they are selfish according with the BAR model).

Finally, the model considered here can be seen as a particular case of BAR where rational servers take malicious actions and the considered application is similar to the one considered in [3]. However, in contrast to [3], we do not assume any trusted third party to identify users but we rather assume that clients are anonymous (e.g., they are connected through the Tor anonymous network [22]) and we investigate the impact of this assumption together with the rational model. To the best of our knowledge, this is the first paper that analyzes how the anonymity can help in managing rational malicious behaviors.

3 System Model

The distributed system is composed by a set of n servers implementing a distributed shared memory abstraction and by an arbitrary large but finite set of clients $\mathcal{C}$. Servers are fully identified (i.e. they have associated a unique identifier $s_1, s_2 \dots s_n$) while clients are anonymous, i.e. they share the same identifier.

Communication model and timing assumptions. Processes can communicate only by exchanging messages through *reliable* communication primitives, i.e. messages are not created, duplicated or dropped. The system is synchronous in the following sense: all the communication primitives used to exchange messages guarantee a timely delivery property. In particular, we assume that clients communicate with servers trough a *timely* reliable broadcast primitive satisfying the following property:

- Reliable Broadcast Timely Delivery: there exists an integer δ, known by clients, such that if a client broadcast a message m at time t and a server s_i delivers m, then all the servers s_j deliver m by time $t + \delta$.

Servers-client and client-client communications are done through "point-to-point" *anonymous timely* channels (a particular case of the communication model presented in [9] for the most general case of homonyms). Considering that clients are identified by the same identifier ℓ, when a process sends a point-to-point message m to an identifier ℓ, all the clients will deliver m. More formally:

- Point-to-point Timely Delivery: there exists an integer $\delta' \leq \delta$, known by processes, such that if s_i sends a message m to a client identified by an identifier ℓ at time t, then all the clients identified by ℓ receive m by time $t + \delta$.

We assume that channels are *authenticated* (or "oral" model) , i.e. when a process identified by j receives a message m from a process identified by i, then p_j knows that m has been generated by a process having identifier i.

Failure model. Servers are partitioned into two disjoint sub-sets: *correct* servers and *malicious* servers (also called *attackers*). Correct servers behave according to the protocol executed in the distributed system (and discussed in Section 6) while malicious servers represent entities compromised by an adversary that may deviate from the protocol by dropping messages (omission failures), changing the content of a message, creating spurious messages, exchanging information outside the protocol etc... Malicious servers are *rational*, i.e. they deviate from the protocol by following a strategy that aims at increasing their own benefit (i.e., to perform actions that may prevent the correct execution of the protocol). We assume that rational malicious servers act independently i.e., they do not form a coalition and each of them acts for its individual gain. We will discuss in Section 7 issues arising in handling coalitions governed by a unique adversary. Servers may also fail by crashing and we identify as *alive* the set of non crashed servers[1]. However, we assume that there always exists at least one correct alive server in the distributed system. Clients can fail only by crashing and any number of client may fail during the computation. Let us note that clients do not know the subset of rational malicious processes.

4 Regular Registers

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely read() and write(). Informally, the write() operation updates the value stored in the shared variable while the read() obtains the value contained in the variable (i.e. the last written value). Every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event. These events occur at two time instants (invocation time and reply time) according to the fictional global time.
An operation *op* is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply). Contrary, an operation *op* is said to be *failed* if it is invoked by a process that crashes before the reply event occurs. According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock and the invocation time and

[1] Alive servers may be both correct or malicious.

response time of every operation are defined with respect to this fictional clock.
Given two operations op and op', and their invocation event and reply event times ($t_B(op)$ and $t_B(op')$) and return times ($t_E(op)$ and $t_E(op')$), we say that op *precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* ($op||op'$). Given a write(v) operation, the value v is said to be written when the operation is complete.
In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [14]: *safe*, *regular* and *atomic*. In this paper, we will consider a regular register which is specified as follows:

- Termination: If an alive client invokes an operation, it eventually returns from that operation.
- Validity: A read operation returns the last value written before its invocation, or a value written by a write operation concurrent with it.

Interestingly enough, safe, regular and atomic registers have the same computational power. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers. There is a huge number of papers in the literature discussing such transformations (e.g., [6, 11, 20, 23, 24] to cite a few). In this paper, we assume that the register is single writer in the sense that no two write() operations may be executed concurrently. However, any client in the system may issue a write() operation. This is not a limiting assumption as clients may use an access token to serialize their writes[2]. We will discuss in Section 7 how this assumption can be relaxed.

5 Modeling the Register protocol as a Game

In a distributed system where clients are completely disjoint from servers, it is possible to abstract any register protocol as a sequence of requests made by clients (e.g. a request to get the value or a request to update the value) and responses (or replies) provided by servers, plus some local computation. If all servers are correct, clients will always collect the expected replies and all replies will always provide the right information needed by the client to correctly terminate the protocol. On the contrary, a compromised server can, according to its strategy, omit to send a reply or can provide bad information to prevent the client from terminating correctly. In this case, in order to guarantee a correct execution, the client tries to detect such misbehavior, react and punish the server. Thus, a distributed protocol implementing a register in presence of rational malicious servers can be modeled as a two-party game between a client and each of the servers maintaining a copy of the register: the client wants to access correctly the register while the server wants to prevent the correct execution of a read() without being punished.

Players. The two players are respectively the client and the server. Each player can play with a different role: servers can be divided in to *correct* servers and *malicious* servers while clients can be divided in those asking a *risky request* (i.e., clients able to detect misbehaviors and punish server) and those asking for a *risk-less request* (i.e., clients unable to punish servers).

Strategies. Players' strategies are represented by all the possible actions that a process may take. Clients have just one strategy, identified by $\mathcal{R}$, that is *request information to servers*. Contrarily, servers have different strategies depending on their failure state:

[2]Let us recall that we are in a synchronous system and the mutual exclusion problem can be easily solved also in presence of failures.

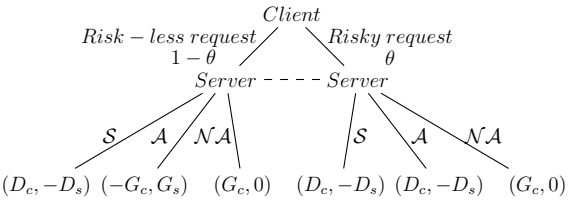

Figure 1: Extensive form of the game. Dotted line represents the unknown nature of requests from the risk point of view. The outcome pairs refer to the client and server gains respectively.

- malicious servers have three possible strategies: (i) $\mathcal{A}$, i.e. *attack the client* by sending back wrong information (i.e. it can reply with a wrong value, with a wrong timestamp or both), (ii) $\mathcal{NA}$, i.e. *not attack the client* behaving according to the protocol and (iii) $\mathcal{S}$, i.e. *be silent* omitting the answer to client's requests;
- correct servers have just the $\mathcal{NA}$ strategy.

Let us note that the game between a correct client and a correct server is trivial as they have just one strategy that is follow the protocol. Thus, in the following we are going to skip this case and we will consider only the game between a client and a rational malicious server.

Utility functions and extensive form of the game. Clients and servers have opposite utility functions. In particular:

- every client increases its utility when it is able to read a correct value from the register and it wants to maximize the number of successful read() operations;
- every server increases its utility when it succeeds to prevent the client from reading a correct value, while it loses when it is detected by the client and it is punished.

In the following, we will denote as G_c the gain obtained by the client when it succeeds in reading, G_s the gain obtained by the server when it succeeds in preventing the client from reading and as D_c the gain of the client when detecting the server and as D_s the loss of the server when it is detected. Such parameters are characteristic of every server and describe its behavior in terms of subjective gains/losses they are able to tolerate. Without loss of generality, we assume that G_c, G_s, D_c and D_s are all greater than 0, that all the servers have the same G_s and D_s[3] and that all the clients have the same G_c and D_c. Fig. 1 shows the extensive form of the game.

The game we are considering is a Bayesian game [10] as servers do not have knowledge about the client role but they can estimate the probability of receiving a risky request or a risk-less request i.e., they have a *belief* about the client role. We denote as θ (with $\theta \in [0, 1]$) the server belief of receiving a risky request (i.e. the client may detect that the server is misbehaving) and with $1 - \theta$ the server belief of receiving a risk-less request (i.e. the client is not be able to detect that the server is misbehaving).

Analysis of the Bayesian Game. In the following, we are going to analyze the existence (if any) of a *Bayesian Nash Equilibrium* i.e., a Nash Equilibrium[4] computed by considering the players' belief.

[3]Let us note that if two servers have different values for G_s and D_s, the analysis shown in the following is simply repeated for each server.

[4]Let us recall that a Nash Equilibrium exists when each player selects a strategy and none of the players increases its utility by changing strategy.

Let us note that in our game, clients have just one strategy. Thus, the existence of the equilibrium depends only on the decisions taken by servers according to their utility parameters G_s, D_s and their belief about the nature of a request (i.e., its evaluation of θ). Let us now compute the expected gain $E()$ of a server s_i while selecting strategies $\mathcal{S}$, $\mathcal{NA}$ and $\mathcal{A}$:

$$E(\mathcal{S}) = (-D_s \times (1-\theta)) + (-D_s \times \theta) = -D_s \tag{1}$$

$$E(\mathcal{NA}) = ((1-\theta) \times 0) + (\theta \times 0) = 0 \tag{2}$$

$$E(\mathcal{A}) = ((1-\theta) \times G_s) - (\theta \times D_s) \tag{3}$$

Lemma 1 *The strategy $\mathcal{S}$ is a dominated strategy.*

Proof A server s_i would choose to follows $\mathcal{S}$ over $\mathcal{NA}$ or $\mathcal{A}$ if (i) $E(\mathcal{S}) > E(\mathcal{A})$ or $E(\mathcal{S}) > E(\mathcal{NA})$. However, considering that both D_s and G_s are grater that 0, from equations $(1) - (3)$ we will have that s_i will never choose to play $\mathcal{S}$. $\square_{Lemma\ 1}$

Due to Lemma 1, servers have no convenience in playing $\mathcal{S}$, whatever the other player does. In fact, there would be no increment of their utility by playing $\mathcal{S}$ and then we will not consider such strategy anymore.

Let us note that a server s_i would prefer to play $\mathcal{NA}$ (i.e., to behave correctly) with respect to $\mathcal{A}$ (i.e., to deviate from the protocol) when $E(\mathcal{NA}) > E(\mathcal{A})$. Combining equations (3) and (2) we have that a s_i would prefer to play $\mathcal{NA}$ when

$$\frac{G_s}{(G_s + D_s)} < \theta. \tag{4}$$

from which we derive the following Lemmas:

Lemma 2 *Let s_i be a rational malicious server. If $D_s < G_s$ and $\theta < \frac{1}{2}$ then the best response of s_i is to play strategy $\mathcal{A}$ (i.e. $\mathcal{NA}$ is a dominated strategy).*

Proof Equation (4) can be rewritten as

$$\frac{1}{1+\alpha} > \theta$$

where $\alpha = \frac{D_s}{G_s}$. Considering that $G_s > D_s$ it follows that $\alpha \in (0, 1)$. Note that s_i would prefer to play $\mathcal{A}$ each time that inequality (4) is satisfied and that θ is upper bounded by $\frac{1}{2}$, it follows that s_i will prefer to play $\mathcal{A}$ for any $\theta \in [0, \frac{1}{2})$ and the claim follows. $\square_{Lemma\ 2}$

Lemma 3 *Let s_i be a rational malicious server. If $D_s > G_s$ and $\theta \geq \frac{1}{2}$ then the best response of s_i is to never play strategy $\mathcal{A}$ (i.e. $\mathcal{NA}$ is a dominant strategy).*

Proof Equation (4) can be rewritten as

$$\frac{1}{1+\alpha} > \theta$$

Note that $G_s < D_s$ and then $\alpha \in (1, \infty)$. Considering that s_i would prefer to play $\mathcal{A}$ each time that inequality (4) is satisfied and that θ is lower bounded by $\frac{1}{2}$, it follows that s_i will never prefer to play $\mathcal{A}$ for any $\theta \in [\frac{1}{2}, 1]$ and the claim follows.

$\square_{Lemma\ 3}$

Note that, in the system model considered here, there is no way for servers to get feedbacks on their actions and the belief θ cannot be updated dynamically through the Bayes rule stage by stage. Therefore, we cannot consider the dynamic form of the game.

6 A Protocol for a Regular Register with $\theta \geq \frac{1}{2}$ and $D_s > G_s$

In this section, we propose a protocol implementing a regular register in a synchronous distributed system with anonymous clients and up to $n-1$ malicious rational servers. The protocol works under the assumption that the server loss D_s in case of detection is higher than its gain G_s obtained when the client fails during a read (i.e. $D_s > G_s$). This assumption models a situation where the attacker is much more interested in having access to data stored in the register and occasionally interfere with the server rather than causing a reduction of the availability (e.g., no termination or validity violation).

Our protocol follows the classical quorum-based approach. When a client wants to write, it sends the new value together with its timestamp to servers and waits for acknowledgments. Similarly, when it wants to read, it asks for values and corresponding timestamps and then it tries to select a value among the received ones. Let us note that, due to the absence of knowledge on the upper bound of malicious processes, it could be impossible for a reader to select a value among those reported by servers and, in addition, the reader may be unable to distinguish well behaving servers from malicious ones. To overcome this issue we leverage on the following observation: the last client c_w writing a value v is able to recognize such value while reading after its write (as long as no other updates have been performed). This makes the writer c_w the only one able to understand which server s_i is reporting a wrong value $v_i \neq v$, detect it as malicious and punish it by excluding s_i from the computation. Thus, the basic idea behind the protocol is to exploit the synchrony of the system and the anonymity of clients to makes the writer indistinguishable from readers and "force" malicious servers to behave correctly.

Let us note that anonymity itself is not enough to make the writer indistinguishable from other clients. In fact, if we consider a naive solution where we add anonymity to a register implementation (for the example the one given by Attiya, Bar-Noy and Dolev [4]), we have that servers may exploit the synchrony of the channels to estimate when the end of the write operation occur and to infer whether a read request may arrives from the writer or from a different client (e.g., when it is received too close to a write request and before the expected end of the write). To this aim, we added in the write() operation implementation some *dummy* read requests. These messages are actually needed to generate message patterns that makes impossible to servers to distinguish if messages come from the writer itself or from a different client. As a consequence, delivering a read request, a server s_i is not able to distinguish if such request is risky (i.e. it comes from the writer) or is risk-less (i.e. it comes from a generic client).

In addition, we added a detection procedure that is executed both during read() and write() operations by any clients. In particular, such procedure checks that every server answered to a request and that the reported information are "coherent" with its knowledge (e.g., timestamps are not too old or too new). The detection is done first locally, by exploiting the information that clients collect during the protocol execution, and then, when a client detects a server s_j, it disseminates its detection so that the malicious server is permanently removed from the computation (collaborative detection).

Finally, the timestamp used to label a new written value is updated by leveraging acknowledgments sent by servers at the end of the preceding write() operation. In particular, during each write() operation, servers must acknowledge the write of the value by sending back the corresponding timestamp. This is done on the anonymous channels that deliver such message to all the clients that will update their local timestamp accordingly. As a consequence, any rational server is inhibited from deviating from the protocol, unless it accepts the high risk to be detected as faulty and removed from the system.

Local Variables at client c_ℓ. Each client keeps locally the following variables:

$-$ $replies$: is a set variable (initially empty) where c_ℓ temporarily stores values and timestamps received

```
Init:
(01) replies ← ∅; my_last_val ← ⊥; my_last_ts ← 0; last_ts ← 0;
(02) ack ← ∅; correct ← {s_1, s_2 ... s_n}; writing ← false;
---------------------------------------------------------------
operation read():
(03) if (last_ts = 0)
(04)   then return ⊥;
(05)   else replies ← ∅;
(06)        broadcast READ();
(07)        wait (2δ);
(08)        if (∀ s_i ∈ correct, ∃ < −, ts, val > ∈ replies)
(09)          then broadcast READACK();
(10)               return val;
(11)          else wait (δ);
(12)               if (∀ s_i ∈ correct, ∃ < −, ts, val > ∈ replies)
(13)                 then broadcast READACK();
(14)                      return val;
(15)                 else execute detection(replies_i, R)
(16)                      broadcast READACK();
(17)                      if (∀ s_i ∈ correct, ∃ < −, ts, val > ∈ replies)
(18)                        then return val;
(19)                        else abort ;
(20)                      endif
(21)               endif
(22)        endif
(23) endif
---------------------------------------------------------------
when REPLY(< j, ts, v, ots, ov >) is delivered:
(24) replies ← replies ∪ {< j, ts, v >};
(25) replies ← replies ∪ {< j, ots, ov >};
---------------------------------------------------------------
when DETECTED(s_j) is delivered:
(26) correct ← correct \ {s_j};
```

(a) Client Protocol

```
Init:
(01) val_i ← ∅; ts_i ← 0;
(02) old_val_i ← ⊥; old_ts_i ← 0; reading_i ← 0;
---------------------------------------------------------------
when READ() is delivered:
(03) reading_i ← reading_i + 1;
(04) send REPLY (< i, ts_i, val_i, old_ts_i, old_val_i >);
---------------------------------------------------------------
when READACK() is delivered:
(05) reading_i ← reading_i − 1;
```

(b) Server Protocol

Figure 2: The read() protocol for a synchronous system.

from servers. In particular, $replies$ stores tuples in the form $< j, ts, val >$ where j is the identifier of the server sending the message, val is the value sent by s_j and identified by timestamp ts.

$-$ my_last_val: is a variable (initially $\perp$) where c_ℓ stores the last value it wrote. It is updated any time that a WRITE() operation is invoked.

$-$ my_last_ts: is a variable (initially set to 0) where c_ℓ stores the timestamp associated to its last written value. It is updated any time that a WRITE() operation is invoked.

$-$ $last\_ts$: is a variable (initially set to 0) where c_ℓ stores the last accepted timestamp, i.e. the greatest timestamp acknowledged by correct servers.

$-$ ack: is a set variable (initially empty) where c_ℓ temporarily stores acknowledgment , received from

servers, for a write() operation. In particular, ack contains tuples in the form $< j, ts, - >$ where j is the identifier of the server sending the message and ts is the timestamp acknowledged by j. The empty value in the tuple is left to simplify the code of the detection() procedure.
$-$ $correct$: is a set variable storing identifiers of servers considered correct.
$-$ $writing$: is a boolean flag (initially set to false) that is set to true when the client starts the execution of a write() operation and is set to false at the end of the operation.

Local Variables at server s_i. Each server s_i has the following local variables:
$-$ val_i: is a set variable where s_i stores the current value(s)[5] of the register.
$-$ ts_i: is an integer variable storing the timestamp associated to the last written value.
$-$ old_val_i: is a set variable where s_i stores the last value written on the register before the current one (i.e. the last value before the one stored in val_i).
$-$ old_ts_i: is an integer variable storing the timestamp associated to the value in old_val_i.
$-$ $reading_i$: it is an integer variable which count the number of READ() messages (and therefore potential read() operations) delivered by s_i.

The read() operation (Fig. 2). When a client wants to read, it first checks if the $last\_ts$ variable is still equal to 0. If so, then there is no write() operation terminated before the invocation of the read() and the client returns the default value $\perp$ (line 04, Fig. 2(a)). Otherwise, s_i queries the servers to get the last value of the register by sending a READ() message (line 06, Fig. 2(a)) and remains waiting for 2δ times, i.e. the maximum round trip message delay (line 07, Fig. 2(a)).
When a server s_i delivers a READ() message, the $reading_i$ counter is increased by one and then s_i sends a REPLY$(< i, ts_i, val_i, old_ts_i, old_val_i >)$ message containing the current and old values and timestamp stored locally (lines 03 - 04, Fig. 2(b)).
When the reading client delivers a REPLY$(< j, ts, val, ots, ov >)$ message, it stores locally the reply in two tuples containing respectively the current and the old triples with server id, timestamp and corresponding value (lines 24 - 25, Fig. 2(a)). When the reader client is unblocked from the wait statement, it checks if there exists a pair $< ts, val >$ in the $replies$ set that has been reported by any servers it believes correct (line 08, Fig. 2(a)) and, in this case, it sends a READ_ACK() message (line 09, Fig. 2(a)) and it returns the corresponding value (line 10, Fig. 2(a)). Delivering the READ_ACK() message, a server s_i just decreases by one its $reading_i$ counter (line 05, Fig. 2(b)). On the contrary, a write() operation may be in progress. To check if it is the case, the client keeps waiting for other δ time units and then checks again if a good value exists (lines 11 - 12, Fig. 2(a)). If, after this period, the value is not yet found, it means that some of the servers behaved badly. Therefore, the client executes the detection() procedure to understand who is misbehaving (cfr. Fig. 4). Let us note that such procedure clean up the set of correct servers when they are detected to be malicious. Therefore, after the execution of the procedure, the reader checks for the last time if a good value exists in its $replies$ set and, if so, it returns such value (line 18, Fig. 2(a)); otherwise the special value abort is returned (line 19, Fig. 2(a)). In any case, a READ_ACK() is sent to block the forwarding of new values at the server side (line 16, Fig. 2(a)).

The write() operation (Fig. 3). When a client wants to write, it first sets its $writing$ flag to true, stores locally the value and the corresponding timestamp, obtained incrementing by one the one stored in $last\_ts$ variable (lines 01 - 02, Fig. 3(a)), sends a WRITE() message to servers, containing the value to be written and the corresponding timestamp (line 03, Fig. 3(a)), and remains waiting for δ time units.

[5] Such set may contain more than one value in case of crash of the writer during the execution of a write().

```
operation write(v):
(01) writing ← true; ack ← ∅;
(02) my_last_ts ← last_ts + 1; my_last_val ← v;
(03) broadcast WRITE(< my_last_val, my_last_ts >);
(04) wait(δ);
(05) replies ← ∅;
(06) broadcast READ();
(07) wait(δ);
(08) broadcast READ();
(09) execute detection(ack, A);
(10) wait(δ);
(11) execute detection(replies_i, R);
(12) broadcast READACK();
(13) broadcast READACK();
(14) writing ← false;
(15) return(ok).
```

```
when WRITE_ACK(ts, s_j) is delivered:
(16)  if (ts ≥ my_last_ts) then ack ← ack ∪ {< j, ts, − >} endif
```

```
when ∃ ts such that S = {j|∃ < j, ts', − >∈ ack} ∧ S ⊇ correct:
(17)  if (ts ≥ last_ts) then last_ts ← ts endif
(18)  for each < j, ts', − > ∈ ack such that ts' = ts do ack ← ack\ < j, ts', − > endFor.
```

(a) Client Protocol

```
when WRITE(< val, ts >) is delivered:
(01)  if (ts > ts_i)
(02)    then old_ts_i ← ts_i;
(03)         old_val_i ← val_i;
(04)         ts_i ← ts;
(05)         val_i ← {val};
(06)    else if (ts_i = ts) then val_i ← val_i ∪ {val}; endif
(07)  endIf
(08)  send WRITE_ACK(ts, i);
(09)  if (reading_i > 0) then send REPLY (< i, ts_i, val_i, old_ts_i, old_val_i >) endif.
```

(b) Server Protocol

Figure 3: write() protocol for a synchronous system.

When a server s_i delivers a WRITE(v, ts) message, it checks if the received timestamp is greater than the one stored in the ts_i variable. If so, s_i updates its local variables keeping the current value and timestamp as old and storing the received ones as current (lines 02 - 05, Fig. 3(b)). Contrarily, s_i checks if the timestamp is the same stored locally in ts_i. If this happens, it just adds the new value to the set val_i (line 06, Fig. 3(b)). In any case, s_i sends back an ACK() message with the received timestamp (lines 08, Fig. 3(b)) and forward the new value if some read() operation is in progress (lines 09, Fig. 3(b)). Delivering an ACK() message, the writer client checks if the timestamp is greater equal than its my_last_ts and, if so, it adds a tuple $< j, ts, - >$ to its ack set (line 16, Fig. 3(a)).

When the writer is unblocked from the wait statement, it sends a READ() message, waits for δ time units and sends another READ() message (lines 06 - 08, Fig. 3(a)). This messages has two main objectives: (i) create a message pattern that makes impossible to malicious servers to distinguish a real reader from the writer and (ii) collects values to detect misbehaving servers. In this way, a rational malicious server, that aims at remaining in the system, is inhibited from misbehaving as it could be detected from the writer and removed from the computation. The writer, in fact, executes the detection() procedure both on the set of ack set and on the $replies$ set collected during the write() (lines 09 - 11, Fig. 3(a)). Finally, the writer sends two READ_ACK() messages to block the forwarding of replies, resets its $writing$ flag to false and returns

```
procedure detection(replies_set, set_type):
(01) S = {j|∃ < j, −, − >∈ replies_set};
(02) if (correct ⊈ S)
(03)   then for each s_j ∈ (correct_i \ S) do
(04)                trigger detect(s_j);
(05)                correct_i ← correct_i \ {s_j};
(06)                broadcast DETECTED(s_j);
(07)        endFor
(08) endif
(09) if (set_type = R)
(10)   then if (writing)
(11)          then R = {j|∃ < j, my_last_val, my_last_ts >∈ replies_set};
(12)               if (correct ⊈ R)
(13)                 then for each s_j ∈ (correct_i \ R) do
(14)                           trigger detect(s_j);
(15)                           correct_i ← correct_i \ {s_j};
(16)                           broadcast DETECTED(s_j);
(17)                      endFor
(18)               endIf
(19)          else for each < j, ts, − >∈ replies_set such that ts < last_ts − 1 do
(20)                           trigger detect(s_j);
(21)                           correct ← correct \ {s_j};
(22)                           broadcast DETECTED(s_j);
(23)               endFor
(24)               for each < j, ts, val >∈ replies_set such that ts = my_last_ts do
(25)                           D_i = {v | (∃ < j, ts, val >∈ replies_set) ∧ (ts = my_last_ts)};
(26)                           if ((my_last_val ≠ ⊥) ∧ (my_last_ts = last_ts) ∧ (last_val ∉ D_i))
(27)                             then trigger detect(s_j);
(28)                                  correct ← correct \ {s_j};
(29)                                  broadcast DETECTED(s_j);
(30)                           endif
(31)               endFor
(32)               for each < j, ts, val >∈ replies_set such that ts > last_ts + 1 do
(33)                           trigger detect(s_j);
(34)                           correct_i ← correct_i \ {s_j};
(35)                           broadcast DETECTED(s_j);
(36)               endFor
(37)        endif
(38)   else for each < j, ts, − >∈ replies_set such that ts ≠ my_last_ts do
(39)                 trigger detect(s_j);
(40)                 correct ← correct \ {s_j};
(41)                 broadcast DETECTED(s_j);
(42)        endFor
(43) endif.
```

Figure 4: detection() function invoked by an anonymous client for a synchronous system.

from the operation (lines 12 - 15, Fig. 3(a)).
Let us note that, the execution of a write() operation triggers the update of the $last\_ts$ variable at any client. This happens when in the ack set there exists a timestamp reported by any correct server (lines 17 - 18, Fig. 3(a)).

The detection() **procedure (Fig 4).** This procedure is used by clients to detect servers misbehaviors during the execution of read() and write() operations. It takes as parameter a set (that can be the $replies$ set or the ack set) and a flag that identifies the type of the set (i.e. A for ack, R for replies). In both cases, the client checks if it has received at least one message from any server it saw correct and detects as faulty all the servers omitting a message (lines 01 - 08). If the set to be checked is a set of ACK() messages, the client (writer) just checks if some server s_j acknowledged a timestamp that is different from the one it is using in

the current write() and, if so, s_j is detected as malicious (lines 38 - 42).
On the contrary, if the set is the $replies$ set (flagged as R), the client checks if it is running the procedure while it is writing or reading (line 10). If the client is writing, it just updated the state of the register. Thus, the writer checks that all servers sent back the pair $< v, ts >$ corresponding to the one stored locally in the variables my_last_val and my_last_ts. If someone reported a bad value or timestamp, it is detected as misbehaving (lines 11 - 18). If the client is reading, it is able to detect servers sending back timestamps that are too old (lines 19 - 23) or too new to be correct (lines 32 - 36) or servers sending back the right timestamp but with a wrong value (lines 24 - 31).

6.1 Correctness Proofs

In this section, we prove that the protocol presented in Fig. 2 - 4 terminates when clients do not crash during the operations execution (Lemma 4, Lemma 5 and Theorem 1), then we prove some properties of the timestamp mechanism used to label write() operations. In particular, we will prove that the protocol ensures the increasing monotonic order of timestamps (Lemma 8) and the consistency of the variable storing the last used timestamp (Lemma 7). In Lemma 9 we prove that the last written value persists locally at each correct server, while Theorem 2 proves that if servers behave correctly then the protocol emulates a regular register. Then we prove the accuracy of the detection function (Lemma 11, Lemma 12 and Theorem 3). Finally, we prove that the proposed protocol allows malicious servers to compute $\theta \geq \frac{1}{2}$ (Theorem 4) and we show that this makes possible to emulate a regular register in the case $D_s > G_s$ (Theorem 5).

To ease of presentation, in the following, we sometimes use the notation c_i, c_j to denote different clients, however such identifiers are not known to processes.

Lemma 4 *Let c_ℓ be an anonymous client invoking a* write() *operation. If c_ℓ does not crash and executes the protocol in Fig. 3 then it eventually returns from the* write() *operation.*

Proof The proof simply follows by observing that in the write() operation code (e.g. Fig. 3(a)) the return event happens after three wait() statement. Thus, considering that c_ℓ is correct, it will be unblocked, from the last wait() statement, 3δ time units after the write() invocation and the claim follows. $\square_{Lemma\ 4}$

Lemma 5 *Let c_ℓ be an anonymous client invoking a* read() *operation. If c_ℓ does not crash and executes the protocol in Fig. 2 then it eventually returns from the* read() *operation.*

Proof The proof simply follows by observing that in the read() operation code a return event is defined in every branch of the code and it only depends on wait() statements. Thus, considering that c_ℓ is correct, it will be unblocked in a finite time after the read() invocation and the claim follows. $\square_{Lemma\ 5}$

Theorem 1 (Termination) *Let c_ℓ be an anonymous client invoking an operation op. If c_ℓ does not crash and executes the protocol in Fig. 2 - 4 then it eventually returns from op.*

Proof The proof directly follows from Lemma 4 and Lemma 5. $\square_{Theorem\ 1}$

Lemma 6 *Let op be a* write() *operation and let ts be the timestamp associated by the writer to op. If a malicious server s_i deviates from the protocol by omitting the* WRITE_ACK(ts) *message or by sending a* WRITE_ACK(ts') *message (with $ts' \neq ts$) and the writer does not crash, then s_i will be detected as malicious by any client.*

Proof The detection() function is executed by the writer client on the ack set at line 09, Fig. 3(a). The ack set is emptied at the beginning of every write() operation and it is filled-in by the writer when it deliver WRITE_ACK(ts') messages. Such messages are sent by servers when delivering a WRITE($< ts, val >$) message sent by the writer at the beginning of the operation. The proof simply follows by considering that the writer client knows the real value of the timestamp associated to the write and stored in its my_last_ts local variable (line 02, Fig. 3(a)). Thus, when the writer executes the detection() function the writer checks (i) if it has received a WRITE_ACK() message from any servers it sees (line 01-07, Fig. 4) and (ii) if all the alive servers acknowledge the right timestamp (line 38-42, Fig. 4). Considering that the communications are timely and the detection happens 2δ time units after the broadcast of the WRITE($< ts, val >$) message (i.e. after the maximum round trip delay), if some servers do not answer they are detected as malicious as they omitted to answer. In the second case, since the writer know its timestamp and channel are authenticated, it is able to detect as malicious the server answering with a different timestamp. Finally, considering that (i) the writer notifies to all the other clients its detections, (ii) such detections are done δ time units before the end of the write and (iii) clients notifications delay is also bounded by δ, it follows that at the end of the write any client detected the malicious servers and the claim follows. $\square_{Lemma\ 6}$

Lemma 7 *At the end of every* write() *operation any client stores in its $last\_ts$ variable the same timestamp.*

Proof Every client initializes its $last\_ts$ variable to 0 during the init phase (line 01, Fig. 2(a)). Such variable is updated at line 17, Fig. 3(a) when the client stores in its ack set a timestamp ts' that has been acknowledged by any alive server and that is greater than the previous one (to preserve the monotonically increasing order of timestamps). Then, we just need to prove that if a client updates its $last\_ts$ variable then all the clients will update it as well.

Considering that (i) WRITE_ACK(ts') messages are sent by servers when a WRITE($< val, ts' >$) message is delivered, (ii) WRITE($< val, ts' >$) messages are sent to all the servers and (iii) WRITE_ACK(ts') messages are sent through point-to-point anonymous channels, we have that all clients deliver WRITE_ACK(ts') messages from the same set of servers. In addition, due to Lemma 6, we have that the set of alive processes is shared by every client and the claim follows.

$\square_{Lemma\ 7}$

Lemma 8 *Let op and op' be two* write() *operations such that $op \prec op'$. Let ts and ts' respectively the timestamp associated to op and to op', the $ts < ts'$.*

Proof The proof simply follows by Lemma 7 and considering that the timestamp associated to a write() operation is computed by incrementing the $last\_ts$ variable by one. $\square_{Lemma\ 8}$

Corollary 1 *If there not exists two concurrent* write() *operations and a clients do not crash during the execution of a* write() *then for any pair of* write() *w_1, w_2 such that $w_1 \prec w_2$ and there not exists any w_3 such that $w_1 \prec w_3 \prec w_2$ then the timestamp ts_2 associated to w_1 is equal to $ts_1 + 1$.*

Proof The proof simply follows by considering that timestamps are computed by incrementing the $last\_ts$ variable and it is updated at most once during each write() operation. $\square_{Corollary\ 1}$

Lemma 9 *At the end of a* write(v) *operation, every server s_i behaving correctly stores the value v in its val_i local variable.*

Proof Every server s_i updates its $value_i$ variable in line 05 or line 06, Fig. 3(b). In particular, this happens when the timestamp attached to the WRITE() message and associated to the write() operation is greater equal than the one stored in ts_i. Due to Lemma 8, timestamps follows a monotonically increasing order and thus every write() operation will have a timestamp that is greater equal than the one previously stored. As a consequence, when delivering a WRITE($< val, ts' >$) message, any server s_i will always execute line 05 or 06, Fig. 3(b) storing locally the new value and the claim follows. $\square_{Lemma\ 9}$

To the ease of presentation, let us assume that the default value $\perp$ is written by a fictional instantaneous write($\perp$) operation preceding every operation op executed by clients and let us define a valid value as follows:

Definition 1 *Let v be the value returned by a* read() *operation op invoked on the regular register. v is said to be* valid *if*

(i) it is the value written by the last write() *operation terminated before op or*

(ii) it is the value written by a write() *operation concurrent with op.*

Lemma 10 *If all the servers behave correctly (i.e. they follow the protocol presented in Fig. 2(b) - 3(b)) then any* read() *operation returns a valid value.*

Proof Let us suppose by contradiction that all servers follow the protocol and that there exists a read() operation op that returns a value v that is not valid.
Let v_old be the value written by the last write() terminated before the invocation of op.

If v is not valid, it means that v is different from v_old and from the value v' written by a concurrent write(v'), if it exists.

Case 1 - No write() **operation is concurrent with** op**.** Due to Lemma 9, at time t when op is invoked, any server will store locally in their $value_i$ variable the value v_old, together with its timestamp. Since the value stored in the register is updated only when a WRITE() message is delivered (line 05 or 06, Fig. 3(b)), and this happens only when a write() operation is triggered (line 03, Fig. 3(a)), we have that, if no write() operation is concurrent with op, v_old, together with its timestamp, will be stored and not updated by any server during the whole execution of op. Considering that any server s_i behaves correctly, while delivering a READ() message, s_i will answer by sending a REPLY($< i, ts_i, value_i >$) containing the same value and the same timestamp to the reader (line 04, Fig. 2(b)). Thus, at time $t + 2\delta$ the $replies$ set will contains n tuple $< v\_old, ts >$ and the condition at line 08, Fig. 2(a) holds terminating the read() operation with v_old and we have a contradiction as it is a valid value.

Case 2 - There exists a write(v') **operation** op' **concurrent with** op**.** Without loss of generality, let x be the timestamp associated to the value v_old written by the last terminated write() operation preceding op. Let us denote by op' the write(v') operation concurrent with the read() operation op.
Let us note that, according to the protocol in Fig. 2(a), while executing the read() operation op, the reader client will inquiry servers to get the value of the register together with its timestamp. Considering that, by assumption, any alive server s_i behaves correctly, it follows that delivering a READ() message s_i will answer by sending back a REPLY() message containing both the current and the old value and timestamp. Such values are modified only al line 05 or 06, Fig. 3(b) when a server s_i deliver a WRITE() message.

Let $t_B(op')$ and $t_B(op)$ the time at which respectively op' and op are invoked and let $t_E(op')$ and $t_E(op)$ be respectively the return time of op' and op.

Let us consider the following cases:

- **Case 2.1 - $\mathbf{t_B(op') + \delta < t_B(op) < t_B(op') + 2\delta}$.**
  At the beginning of the write() (i.e. at time $t_B(op')$), the writer client sends a WRITE() message (line 03, Fig. 3(a)) that will be delivered by any alive server by time $t_B(op') + \delta$. When a server s_i delivers a WRITE() message, it will update its val_i variable to v' and its ts_i to the current timestamp[6]. It follow that from time $t_B(op') + \delta$ any alive server will store locally the value v' written by op'. Let us note that, since op' lasts until time $t_E(op') = t_B(op') + 3\delta$ and, by assumption, there not exist concurrent write() operations, such variables will not be updated anymore before time $t_B(op') + 3\delta$.

  When an alive server s_i delivers a READ() message (between time $t_B(op') + \delta$ and $t_B(op') + 2\delta$), it executes line 04, Fig. 2(b) sending to the reader both the current and the old values and timestamp through a REPLY() message. Delivering such REPLY() message, the reader will store the values in its $replies$ local variable and waits until time $t_B(op) + 2\delta$ before checking the content of such variable. Considering that, by assumption, all the servers behave correctly, they will send the content of their val_i and ts_i variable without changing them. As a consequence, the reader will store locally in its $replies$ set the same pair $< v', ts >$ from any alive server. Thus, at time $t_B(op) + 2\delta \leq t_B(op') + 3\delta$, evaluating the condition at line 08, Fig. 2(a) the reader will select and return v' and we have a contradiction.

- **Case 2.2 - $\mathbf{t_B(op) < t_B(op') + \delta}$.**
  In this case, the WRITE() message is concurrent (wrt. the happened before relation) with the READ() message. As a consequence, we may have that a server s_i delivering the READ() message before than the WRITE() message and a server s_j delivering the WRITE() message before than the READ() message. As a consequence, s_i will answer by sending a reply() message containing the old value v_old and the previous value (no more valid) while s_j will answer by sending v_old and v'. However, when the client evaluates the $replies$ set, it will find an occurrence of v_old for any alive server and evaluating the condition at line 12, Fig. 2(a) the reader will select and return v_old and we have a contradiction.

- **Case 2.3 - $\mathbf{t_B(op') + 2\delta < t_B(op) < t_E(op')}$.**
  In this case, the READ() message may be delivered also after the end of op'. If no more write() operations occur before the end of op, we fall down to case 2.1 and the claim follows.
  Contrarily, we fall down the the situation of case 2.2 where the concurrent write is a new write and the claim follow again.

□$_{Lemma\ 10}$

Theorem 2 *Let c_ℓ be a client invoking a* read() *operation op. If all the servers behave correctly, the protocol shown in Fig. 2 - 4 implements a regular register.*

Proof The proof directly follows by Theorem 1 and Lemma 10. □$_{Theorem\ 2}$

Lemma 11 *Let c_j be a client sending a* READ() *message at time t and let* REPLY$(< i, ts, v, ots, ov >)$ *be the message delivered by c_j at time $t' > t$ as reply to the* READ(). *Let lts be the value stored locally in the $lats\_ts$ variable by c_j at time t'. If $|lts - ts| > 1$ then s_i is malicious.*

[6]Let us note that such variable will be always updated, as in line 05 or 06, Fig. 3(b), due to Lemma 8.

Proof Let us suppose by contradiction that s_i sends to c_j a REPLY$(< i, ts, v, ots, ov >)$ such that $|lts - ts| > 1$ and s_i is correct.

The $last\_ts$ local variable is updated to a certain value x by any client c_j when (i) c_j delivered an ACK$(< j, x, - >)$ message from any server it believes correct and (ii) x is strictly greater than the previous value stored in $last\_ts$ (line 17, Fig. 3(a)).

Let t' be the time at which c_j delivers the REPLY$(< i, ts, v, ots, ov >)$ message and let us denote with t_{up} the time at which c_j updated its $last\_ts$ local variable to lts. If t_{ack} is the time at which s_i delivered such WRITE(v, lts) message and it sent back the ACK$(< i, lts, - >)$ message to clients, it follows that $t_{ack} < t_{up}$.

If at time t' $lts \leq ts$, it follows that $t_{up} \leq t'$, otherwise $t_{up} > t'$. Let us consider the two cases separately.

- **Case 1 - $\mathbf{t_{up} \leq t'}$**. In this case $t' > t_{ack}$. Let us note that the value ts sent by s_i in the REPLY message is the value stored locally by s_i in the variable ts_i. Let us call t_{rp} the time at which s_i sends the REPLY$(< i, ts, v, ots, ov >)$ message to c_j and let us consider the following cases:
  - **Case 1.1 - $\mathbf{t_{rp} < t_{ack}}$**. In this case, $ts < lts$ and in particular, considering that write() operations are sequential, due to Corollary 1, $ts = lts - 1$. Thus, $|lts - ts| = 1$ and we have a contradiction.
  - **Case 1.2 - $\mathbf{t_{rp} > t_{ack}}$**. In this case, $ts = lts$ and again we have a contradiction.

- **Case 2 - $\mathbf{t_{up} > t'}$**. Let us note that the value ts sent by s_i in the REPLY message is the value stored locally by s_i in the variable ts_i. Let us call t_{rp} the time at which s_i sends the REPLY$(< i, ts, v, ots, ov >)$ message to c_j and let us consider the following cases:
  - **Case 2.1 - $\mathbf{t_{rp} < t_{ack}}$**. In this case, $ts = lts$ and we have a contradiction.
  - **Case 2.2 - $\mathbf{t_{rp} > t_{ack}}$**. In this case, $ts < lts$ and in particular, considering that write() operations are sequential, due to Corollary 1, $ts = lts - 1$. Thus, $|lts - ts| = 1$ and we have a contradiction.

$\square_{Lemma\ 11}$

Lemma 12 *Let c_j be a client sending a* READ() *message at time t and let* REPLY$(< i, ts, v, ots, ov >)$ *be the message delivered by c_j at time $t' > t$ as reply to the* READ(). *Let $mlts$ be the value stored locally in the my_last_ts variable and let mv be the value stored locally in the my_last_value variable by c_j at time t'. If $mlts = ts$ and $mv \neq v$ then s_i is malicious.*

Proof The proof simply follows by considering that a client c_j updates its my_last_ts and my_last_val local variables only at the beginning of a write() operation.

Due to Corollary 1, every write() operation has a unique timestamp thus if c_j delivers a value $v \neq mv$ it means that the server altered it and it cannot be correct.

$\square_{Lemma\ 12}$

Lemma 13 *Let c_j a client sending a* WRITE$(< j, ts, val >)$ *message at time t. If there exists a server s_j such that s_j does not sent the triple $< j, ts, val >$ by time $t + 3\delta$ then s_j is faulty.*

Proof Considering that (i) the first READ() request is sent at time $t+\delta$ (line 06, Fig. 3(a)), (ii) the detection() procedure on $replies$ set is performed at time $t + 2\delta$ (line 11, Fig. 3(a)) and (iii) there not exists concurrent $write()$ operations, then any correct server will always provide the expected triple and the claim follows.

$\square_{Lemma\ 13}$

Lemma 14 *Let c_j a client sending a* WRITE(j, ts, val) *message at time t. If there exists a server s_j such that the triple $< j, ts, - >$ does not appear in the ack set at time $t + 2\delta$, then s_j is detected as faulty.*

Proof Considering that (i) the detection() procedure on ack set is performed at time $t + 2\delta$ (line 09, Fig. 3(a)) and (ii) there not exists concurrent write() operations, then a correct server will provide the expected triple and the claim follows.

$\square_{Lemma\ 14}$

Theorem 3 *A correct alive server s_i is never detected as malicious.*

Proof A server s_i deviates from the protocol if:

1. s_i omits to send ACK() messages during a write() or to send REPLY() messages during a read() (lines 02-07, Fig. 4).
2. s_i sends bad timestamps (i.e. too old lines 19-23, Fig. 4 or too new lines 32-36, Fig. 4).
3. s_i sends a pair $< value, ts >$ with the correct timestamp and the wrong value (lines 24-31, Fig. 4).

Thus, a correct server may be erroneously detected as faulty only if one of the previous cases occur. However, due to Lemmas 11 - 14, we have that if one of the previous situation happens, s_i is necessarily malicious and the claim follows.

$\square_{Theorem\ 3}$

Theorem 4 *Let s_i be a malicious server executing the protocol in Fig. 2 - 4. The belief of s_i is always $\theta \geq \frac{1}{2}$.*

Proof Let us recall that clients executing read() and write() operations are completely anonymous, i.e. they cannot be identified by the servers. In addition, they are completely autonomous in the execution of the operations, i.e. servers have no knowledge about operations invocation frequency and they cannot know or estimate if the same client will execute an operation in the near future.

Let us now consider the game presented in Section 5 and let us consider when this happen in the protocol. Let us recall that a request is said to be risky if the server can be detected by the client and removed from the computation.

Note that, according to the detection() procedure shown in Fig. 4, the writer of the last write() operation is the only one able to detect a malicious server playing strategy $\mathcal{A}$.

Thus, to assess the value of θ, a malicious server computes the probability that the request is sent by the last writer client.

Every client starts a game each time that it interacts with servers and in particular this happen:

- when a writer client sends a WRITE() message (line 03, Fig. 3(a))
- when a writer client sends a READ() message (line 06, Fig. 2(a) and lines 06 and 08, Fig. 3(a))

Let un now compute the value of θ, i.e. the belief of a malicious server that the current request is risky, in both cases.

- **Case 1 - s_i delivers a** WRITE() **message**. WRITE() messages are sent only during a write() operation. Thus, when delivering such a message, s_i is sure that its answer will be processed by the writer and thus it evaluates $\theta = 1$.
- **Case 2 - s_i delivers a** READ() **message**. READ() messages are sent both during write() operations and read() operations. Let us note that a server knows that (i) operations are executed sequentially by each process, i.e. clients cannot invoke a new operation unless the previous one is terminated and (ii) write() operations last for 3δ time units.

  Therefore, to evaluate the risk of playing strategy $\mathcal{A}$, a server computes the probability that a READ() message comes from the writer knowing the protocol. Let Δ be the amount of time between the last WRITE() message delivered by s_i and the considered READ() message.

  - **Case 2.1 - $\boldsymbol{\Delta > 3\delta}$**. In this case, s_i can only know that the previous write operation is terminated. However, considering that it has no further information about other operations invoked by the same client, when delivering the READ() message it can only evaluate the probability that it comes from the same client by flipping a coin and thus it evaluates $\theta = \frac{1}{2}$.
  - **Case 2.2 - $\boldsymbol{\Delta \leq 3\delta}$**. In this case, s_i may assume that the previous write operation is still running. However, by looking to the write() protocol, the writer will send 2 READ() messages before terminating the operation. Considering that (i) channels may deliver a message sent at some time t in any time interval between time t and $t + \delta$ and (ii) readers invokes operations independently from the writer, it follows that when delivering a READ() message, s_i is not able to infer additional information allowing to distinguish the writer from other clients and use it to calculate θ. In other words, the probability that a READ() message is sent from the writer does not change knowing the number of READ() messages delivered in the past and it is again computed by flipping a coin. Therefore, s_i evaluates $\theta = \frac{1}{2}$.

$\square_{Theorem\ 4}$

Theorem 5 *If $D_s > G_s$ the protocol shown in Fig. 2 - 4 implements a regular register with only* 1 *alive correct server.*

Proof Let us first notice that the existence of the detection function creates the dualism between risky requests and risk-less request. In addition, the existence of one correct alive server prevents the attacker to create a collusion able to make the condition at lines 08, 12 and 17 satisfied with not valid values. Thus, the claim simply follows by Lemma 3, Theorem 2 and Theorem 4 $\square_{Theorem\ 5}$

7 Discussion and Concluding Remarks

This paper addressed the problem of building a regular register in a distributed system where clients are anonymous and servers maintaining the register state may be rational malicious processes. We modeled our problem as a two-parties Bayesian game and we designed a distributed protocol able to reach the Bayesian

Nash Equilibrium and to emulate a regular register when the loss in case of detection is greater than the gain obtained from the deviation (i.e. $D_s > G_s$). To the best of our knowledge, our protocol is the first register protocol working in the absence of knowledge on the number of compromised replicas.

The protocol relies on the following assumptions: (i) rational malicious servers act independently and do not form a coalition, (ii) the system is synchronous, (iii) clients are anonymous and (iv) write operations are serialized. Let us now discuss how this assumptions impact on our results, how they can be relaxed and how we plan to address the remaining issues as future works.

When clients are not anonymous, servers are able to understand the nature of the request i.e., they are able to understand if the client that is sending the request is able to detect its potential misbehavior or not and to compute whether $\theta = 0$ or $\theta = 1$. A simple solution to overcome this issue, is to increase the degree of collaboration among clients. In particular, it is possible to introduce a verification phase at the end of the $\mathsf{read}()$ where a client, that is not able to complete the operation, asks to others to verify the information reported by servers. This extension has the effect to discourage servers from reporting bad values but has the main drawback of requiring an high collaboration among clients while in the current solution, the only form of client collaboration is represented by the advertisement of the detection. We are currently studying how it is possible to weaken the anonymity assumption to move towards a model with homonyms like the one presented in [9] without increasing the level of client collaboration.

Concerning the serialization of $\mathsf{write}()$ operations, this can be easily managed by using a set instead of an integer to store values. Concurrent writes are stored in such set and the whole set is returned to the client that will select the value reported by every server.

As future works, we are investigating how to solve the same problem under weaker synchrony assumption or considering a different type of rational attacker like the attacker that aims at breaking the validity of the register regardless the risk of being detected (i.e. the game with $G_s > D_s$) or an attacker that controls a coalition of processes. Addressing these points is actually far from be trivial. Considering a fully asynchronous system, in fact, makes impossible to use our punishment mechanism as clients are not able to distinguish alive but silent servers from those crashed. Additionally, when the attacker is able to compromise and control a coalition of processes, the model provided in this paper is no more adeguate and we are studying if and how it is possible to define a *Bayesian Coalitional Game* [12] for our problem and if an equilibrium can be reached in this case.

References

[1] Abraham, I., Dolev, D., Halpern, J. Y. *Distributed Protocols for Leader Election: A Game-Theoretic Perspective.* DISC 2013: 61-75

[2] Afek, Y., Ginzberg, Y., Landau Feibish, S. and Sulamy, M. *Distributed computing building blocks for rational agents.* PODC 2014: 406-415.

[3] Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J. P., and Porth, C. *BAR fault tolerance for cooperative services*. ACM SIGOPS Operating Systems Review. ACM, 2005. p. 45-58.

[4] Attiya, H., Bar-Noy, A., and Dolev, D. *Sharing memory robustly in message-passing systems*. Journal of the ACM 42, 1, 1995, 124-142.

[5] Bazzi R. A., *Synchronous Byzantine Quorum Systems*, Distributed Computing 13(1), 45-52, 2000.

[6] Chaudhuri S., Kosa M.J. and Welch J., *One-write Algorithms for Multivalued Regular and Atomic Registers.* Acta Informatica, 37:161-192, 2000.

[7] Clement, A., Li, H. C., Napper, J., Martin, J., Alvisi, L., Dahlin, M. *BAR primer*, DSN 2008: 287-296

[8] Clement, A., Napper, J., Li, H., Martin, J. P., Alvisi, L., and Dahlin, M. *Theory of BAR games*, PODC 2007: 358-359

[9] Delporte-Gallet, C., Fauconnier, H., Tran-The, H. *Uniform Consensus with Homonyms and Omission Failures* ICDCN 2013: 161-175

[10] Fudenberg, D., Tirole, J. *Game theory*, 1991. Cambridge, Massachusetts.

[11] Haldar S. and Vidyasankar K., *Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables.* JACM, 42(1):186-203, 1995.

[12] Ieong, S., Shoham, Y. *Bayesian Coalitional Games.* AAAI. 2008: 95-100

[13] Li, H. C., Clement, A., Wong, E. L., Napper, J., Roy, I., Alvisi, L., Dahlin, M. *BAR Gossip* OSDI 2006: 191-204

[14] Lamport. L., On Interprocess Communication, Part 1: Models, Part 2: Algorirhms, *Distributed Computing*, 1(2):77-101, 1986.

[15] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. *Depot: Cloud storage with minimal trust*, ACM TOCS 29(4), 2011

[16] Malkhi D., Reiter M. K. *Byzantine Quorum Systems*, Distributed Computing 11(4), 203-213, 1998.

[17] Martin J., Alvisi L., Dahlin M., *Small Byzantine Quorum Systems*, DSN 2002: 374-388.

[18] Martin J., Alvisi L., Dahlin M.. *Minimal Byzantine Storage*, DISC 2002.

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

[20] Singh A.K., Anderson J.H. and Gouda M., *The Elusive Atomic Register.* JACM, 41(2):331-334, 1994.

[21] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., Verissimo, P. *Highly available intrusion-tolerant services with proactive-reactive recovery*, IEEE TPDS 21(4): 452-465, 2010 .

[22] The Tor Project `https://www.torproject.org`.

[23] Vidyasankar K., *Converting Lamport's Regular Register to Atomic Register.* IPL, 28(6):287-290, 1988

[24] Vityani P. and Awerbuch B., *Atomic Shared Register Access by Asynchronous Hardware.* FOCS 1987, 223-243.