Stabilizing Byzantine-Fault Tolerant Storage

Silvia Bonomi*, Maria Potop-Butucaru‡ and Sébastien Tixeuil‡
*Universitá degli Studi di Roma La Sapienza
Roma, Italy
Email: bonomi@dis.uniroma1.it
‡ UPMC Sorbonne Université
Paris, France
Email: firstname.name@lip6.fr

***Abstract*—Distributed storage service is one of the main abstractions provided to developers of distributed applications due to its ability to hide the complexity generated by the various messages exchanged between processes. Many protocols have been proposed to build Byzantine-fault-tolerant (BFT) storage services on top of a message-passing system but none of them considers the possibility that well-behaving processes (*i.e.* correct processes) may experience transient failures due to, say, isolated errors during computation or bit alteration during message transfer. This paper proposes a stabilizing Byzantine-tolerant algorithm for emulating a *multi-writer multi-reader* regular register abstraction on top of a message passing system with $n > 5f$ servers, which we prove to be the minimal possible number of servers for stabilizing and tolerating f Byzantine servers. That is, each read operation returns the value written by the most recent write and write operations are totally ordered with respect to the happened before relation. Our algorithm is particularly appealing for cloud computing architectures where both processors and memory contents (including stale messages in transit) are prone to errors, faults and malicious behaviors. The proposed implementation extends previous BFT implementations in two ways. First, the algorithm works even when the local memory of processors and the content of the communication channels are initially corrupted in an arbitrary manner. Second, unlike previous solutions, our algorithm uses bounded logical timestamps, a feature difficult to achieve in the presence of transient errors.**

***Keywords*-Distributed System;Self-Stabilization Byzantine Fault Tolerance, Bounded Labels, Pseudo-Stabilization, Regular Register**

I. Introduction

Cloud Computing is one of the most popular emerging technologies. In a nutshell, clouds offer to organizations (clients) the possibility to store and access large amounts of data on remote servers managed by providers such as Amazon, Yahoo, or Google. The cloud model is very similar to the well known client-server model. However, clouds experience both faults and errors and are often the target of cyber attacks. Recently, mainstream media reported the unavailability of various cloud providers for several days due to multiple causes such as software bugs, errors in internal migrations or cyber attacks. It is now established that cloud storage systems must be able to mask to their clients the unexpected yet possible faults of processing entities, memory transient errors due to internal storage reorganization, software bugs, or transient or permanent cyber attacks. In these architectures, applying the classical technique consisting in restarting the system anytime an error or a fault is detected has an important impact on the clients quality of experience and overall satisfaction. In this context, fault and attack tolerant schemes occurring in theoretical distributed computing are extremely relevant for the daily practice of cloud computing, where important properties such as availability, reliability, serviceability, and fault-tolerance are mandatory. Typically, it is necessary to combine both "classical" fault-tolerance techniques as well as fault-recovery (that is, self-stabilization) to ensure the aforementioned properties. On the one hand, *self-stabilization* [1], [2] is a versatile technique that permits automatic recovery of a system following the occurrence of *transient* faults. Note that the automatic recovery addressed using self-stabilization techniques does not need explicit human intervention nor the stop and the reboot of the entire system. On the other hand, *fault-tolerance* [3] is traditionally used as a term indicating the potential of the system to tolerate a limited number of *permanent* faults, the most severe being Byzantine faults, where processing entities may deviate arbitrarily from their intended behavior. Providing core building blocks for application designers (such as consistent storage) that are highly resilient to various kinds of faults and errors is essential for cloud computing. However, making these systems tolerant to both transient errors and permanent faults is a challenging task since impossibility results are expected in many cases [4], [5], [6], [7].

In this paper we propose the first multi-writer multi-reader (MWMR) regular register emulation on top of a message passing system that is tolerant to both Byzantine failures and transient memory corruptions. Additionally, our solution uses bounded timestamps which is also novel in the context of the BFT implementations of regular registers. Our emulation works under the assumption that the number of servers in the system is $n > 5f$ (where f is an upper bound on the number of byzantine servers). Although, the proposed bound is higher than the lower bound required to build a non stabilizing BFT regular register (*i.e.* $3f + 1$), this difference is due mainly to the fact that temporarily the servers memory can be corrupted and for a short time they may have a behavior that is similar to Byzantine servers.

II. System Model and Problem Statement

We consider a distributed system composed of a set of n servers $S = \{s_1, s_2, \ldots, s_n\}$ emulating a shared memory, and

a disjoint set of an unknown but finite number of clients $C = \{c_1, c_2, \ldots, c_i, \ldots\}$ accessing the shared memory via read and write operations. We consider that the system is asynchronous and that processes may communicate only by exchanging messages. Each pair of processes (both clients and servers) is connected through reliable FIFO point to point channels, *i.e.* messages are not created, modified or lost by the channel and they are delivered following the FIFO order. Note that, this behavior can be ensured by using a stabilization preserving data-link protocol built on top of bounded, non-reliable but fair, non-FIFO communication channels [8].

Failure Model. Any process in the distributed system may fail: *clients (readers and writers)* can experience *crash failures* and there is no bound on the number of faulty clients, while *servers* can be *Byzantine*, *i.e.* they can deviate arbitrarily from the protocol. We assume that at most f servers can be Byzantine. In addition, both clients and servers can experience *transient failures* (*i.e.* failures of finite duration) that result in the corruption of their local state and channels contents (*i.e.* the values stored in the local variables may be altered as well as messages crossing channels). We assume that both clients and servers can start their execution in a corrupted state and these corruptions are transient and happen not too often to prevent the convergence of the protocols.

A. Regular Register

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* (or *return*) event. These events occur at two time instants (invocation time and reply/return time) according to the fictional global time.

An operation op is *complete* if both the invocation event and the return 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 return 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 (not accessible by processes) and the invocation time and return time of every operation are defined with respect to this fictional clock.

Given two operations op and op', and their invocation 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 (*i.e.* when the return event is generated for the current operation).

MWMR Regular Register Specification. 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 registers have been defined by Lamport [9]: *safe*, *regular* and *atomic*. In this paper, we consider a multi-writer/multiple reader regular register (defined in [10] and formalized in [11]):

- Termination: If a correct client invokes an operation op, it eventually returns from that operation (*i.e.* it generates a return event).
- Validity: A read operation returns the last value written before its invocation, or a value written by a write operation concurrent with it.
- Consistency: For any two read operations the set of writes that do not strictly follow either of them must be perceived by both reads as occurring in the same order.

In this paper, we are going to consider a multi-writer/multiple-reader regular register and we will provide a distributed protocol satisfying the following definition:

Definition 1 (f-BTPS [12]): A distributed protocol $\mathcal{P}$ is f-Byzantine-tolerant and pseudo-stabilizing (f-btps for short) for specification $spec$ if and only if starting from any arbitrary configuration every execution of $\mathcal{P}$ involving at most f Byzantine processes has a suffix satisfying $spec$.

III. A Lower Bound for Pseudo Stabilizing Byzantine Tolerant Regular Register Implementation

In the following we prove that in the considered multi-fault model the regular register cannot be implemented with $n \leq 5f$ servers.

Note that it can be trivially proved that, when the system starts in an arbitrary configuration, in the absence of a write that completes successfully, read operations either return some garbage value or never return. Therefore, in the following we will make the assumption below.

Assumption 1: *The first write operation that succeeds a transient fault in the system does not stop until completed.*

The following theorem shows the upper bound on the number of faulty servers that any stabilizing protocol that implements a regular register timestamping operations, with one phase reads (no write back) and decision based on majority of correct processes. The class of protocols is denoted $\mathcal{TM}_{1R}$.

Theorem 1 (Upper bound on the number of faulty servers): There is no asynchronous stabilizing protocol in the class $\mathcal{TM}_{1R}$ that implements regular registers with $n \leq 5f$ servers in the presence of f Byzantine servers even when the Assumption 1 holds.

Proof Consider a system with 5 servers one of which is Byzantine. The generalization is trivial (each correct server is replaced by a group of t servers). Assume there is a stabilizing protocol $A \in \mathcal{TM}_{1R}$ implementing BT regular registers. In the following we denote by s_i the servers ($i \in \{1, .., 5\}$) and let s_5 be the Byzantine server. Consider the following

initial server configuration $(tsx, tsx, tsx, ts2, tb)$ where tsx and $ts2$ are timestamps of the correct servers corrupted by some transient fault and tb denotes the timestamp of the Byzantine server.

In the following we construct a possible execution of A, e, that violates the regularity. Assume e stars in a configuration that immediately follows the first successful write, $w0$. Assume that the timestamp introduced by this write is $ts0$. Assume also that when $w0$ computed the $ts0$ timestamp, timestamp $ts2 > ts0$ has not been included in the new timestamp computation (server $s4$ was slow in responding). That is, $ts0 > tsx$ and $ts0 > tb$. Once $w0$ completes, the configuration of the servers can be $(ts0, ts0, ts0, ts2, tb)$. In this configuration server $s4$ is slow and did not changed its timestamp. The writer completed upon the confirmation of servers $s1, s2, s3$ plus the Byzantine server. Consider now the operation $w1$ (that happens after $w0$) and that introduces the timestamp $ts1$ with $ts1 > ts0$ and $ts1 > tb$ and $ts1 < ts2$. Again, this is possible in A since $s4$ may be slow in sending its timestamp to the writer. After, $w1$ completes, the servers configuration can be $(ts1, ts1, ts1, ts2, tb)$. A read operation, $r1$, that happens after $w1$ can obtain the following multi-set of timestamps $\{ts1, ts1, ts2, ts2\}$ where the second $ts2$ is provided by the Byzantine server. Since A implements a regular register, based on the above input, the read operation should return the last written timestamp, namely $ts1$. Assume now $w2$ that follows $r1$ and that introduces $ts2$. The server configuration after $w2$ completes, can be $(ts2, ts2, ts1, ts2, tb)$ where server $s3$ is slow in modifying its timestamp. Finally, consider the read $r2$ that happens after $w2$. $r2$ can obtain the following multi-set of values $\{ts2, ts2, ts1, ts1\}$ where the second $ts1$ may come from the Byzantine server. Since $r2$ collected exactly the same multi-set of values as $r1$, then $r2$ has to return exactly the same value, namely $ts1$. This contradicts the assumption that A implements a regular register. That is, $r2$ was supposed to return the last written timestamp, namely $ts2$. $\square_{Theorem\ 1}$

IV. Tight Pseudo Stabilizing Byzantine Tolerant Regular Register Implementation

In this section, we propose and prove correct a BTPS algorithm that implements regular register provided that the number of servers $n > 5f$. The general idea of our algorithm is similar to classical quorum-based approaches to implement a BFT register [13], [14]. Both read() and write() operations are performed by interacting with a set of servers (*i.e.* a quorum) that is responsible to maintain the current value of the register and to answer to read requests. The main issues we have to face in the implementation are:

1) *the asynchrony of the communications* that makes possible to deliver messages related to the execution of an operation *op* after the end of the operation;
2) *the restriction to use only a finite set of labels to timestamp different operations* that forces clients to reuse the same label to timestamp different operations;
3) *the possibility of non-Byzantine servers to suffer transitory failures* when a client invokes an operation (*i.e.* they are in a transitory phase where they can return wrong information).

A. ToolBox

In asynchronous distributed systems, timestamps are used to associate temporal information to events generated by every process in absence of a fine grained synchronization among physical clocks. Israeli and Li [15] define timestamps as "*numerical labels that enable a system to keep track of temporal precedence relation among its data elements*". There exist several labeling (or time-stamping) systems used in distributed systems. An intuitive definition of a labeling system follows. Labels are elements of a set enhanced with a total antisymmetric binary relation (to compare labels) and a function to compute a new label given a set of existing labels. The simplest one is the set of natural numbers that allows a total order but induces an unbounded memory. In order to avoid this main drawback, Israeli and Li define bounded labeling systems. Similar existing bounded labeling schemes [15], [16], [17] do not tolerate the possibility of transient faults, since there exists initial configurations from which it is impossible to compute a new label.
In this paper, we consider the stabilizing bounded labeling scheme defined by Alon et al. [18] to timestamp write() operations. Such scheme ensures that given any subset of at most k labels, there exists a label that dominates each label of the input subset. Informally, this bounded labeling scheme builds a partial order over the finite set of labels L and given any pairs of labels ℓ_i, ℓ_j, it is always possible to decide if a precedence relation exists among ℓ_i and ℓ_j or if they are not comparable. More formally:

Definition 2 (k-SBLS [18]): A k-stabilizing bounded labeling system ($k \geq 2$) is a triplet $(L, \prec, \mathsf{next}())$ where L is a finite set, $\prec$ is an antisymmetric binary relation over L and next() is a function $next : L^k \rightarrow L$ such that:

$$\forall L' \subseteq L, |L'| \leq k \Rightarrow \forall \ell \in L', \ell \prec next(L').$$

Note that the precedence relation defined over the set of labels is not transitive. Alon et al. [18] proved that there exists a k-stabilizing bounded labeling system for any natural number k.

In addition, due to the asynchrony, clients need to distinguish if the received pairs $\langle value, ts \rangle$ are replies for the current read() or rather replies to an old read() operation that are arriving late. To manage this issue, still using a finite set of labels, we define a procedure (cf. Figure 3) executed by the client every time it needs a label for a read(). More in detail, we assume that every client c_i can use a finite set of labels $\mathcal{LR}_i$ to identify its read() operations. For each label, the reader stores the set of servers that have previously answered to a read() using that label (meaning that, this label is no more expected from the server) and the set of servers that still miss a reply carrying out the current label. Examining the obtained information, the client can pick up a free label (if it exists) or it can try to verify if the missing response is due to its bad initialization (*i.e.* client transitory phase). In

order to do so, the client exploits the FIFO property of the communication channels and send a FLUSH message to be reflected by servers. Once the message came back either the old reply has been received (*i.e.* the label can be now reused) or the client's knowledge was altered by a transitory failure and the label can now be used.

To handle the transitory phase, we introduce the notion of *Weighted Timestamp Graph* (WTsG). Informally, a WTsG is a node-weighted directed graph where vertexes are represented by write timestamps, the weight of the vertex is given by the number of occurrences of the write timestamp and given two vertex ts_i, ts_j there exists an edge in WTsG between them if ts_i precedes ts_j in the timestamps partial order. More formally:

Definition 3 (Weighted Timestamps Graph (WTsG)): Let $T = \{ts_1, ts_2, \ldots, ts_k\}$ be a set of timestamps. Let $\prec$ be the precedence relation defined over T. A *weighted timestamps graph* $\mathcal{G} = (V, E, w)$ is a graph where

- V is the set of different timestamps occurring in T;
- E is the set of directed edges (i, j) such that (i) $ts_i, ts_j \in T$ and (ii) $ts_i \prec ts_j$;
- $w : V \mapsto \mathbb{N}$ is a function that assign to each timestamp in V its occurrence in T (*i.e.* the number of times it appears in T).

In the following we use two different types of WTsG: (i) *local WTsG* and (ii) *union WTsG*. The local WTsG is the weighted graph obtained by every single server, considering all the timestamps it is aware about while the union WTsG is weighted graph obtained by every single server, considering all the timestamps it is aware about (including the past ones). The algorithm uses a WTsGs to infer if servers are in a transitory phase (and then to abort the operation) or if the current operation can be completed without violation of the register specification. More in detail, when a client wants to write a value v, it first inquiries servers to know the current timestamps they store and then it uses such information to compute a new timestamp that follows all the ones stored locally by servers. Once the timestamp is available, the writer performs the effective write on all the servers. In case of a read() operation, the reader inquiries servers to obtain their current values (together with their timestamps). Once it has collected "enough" replies, it constructs the local WTsG using the timestamps gathered through the replies and in case there exists a value that is witnessed by a sufficient large set of servers (*i.e.* for the case of our algorithm we need at least $2f + 1$ witnesses), then it can return the value. If the local WTsG does not include any value with enough witnesses, it may happen that a write() operation is currently running. In order to distinguish this case, the reader computes the union WTsG considering both the current timestamp declared by servers and a partial history of the last n written values that each server knows. Looking at the union WTsG, the reader is able to characterize whether a valid value exists in the system, or servers are experiencing a transitory phase and the operation can be aborted.

B. Single-Writer Multi-Reader Regular Register

In the sequel we provide a detailed description and the complete pseudo-code of the single-writer multi-reader algorithm. The extension to the multi-writer multi-reader will be discussed in the next section.

Local variables maintained by every client c_i. Each client c_i maintains the following local variables:

- $servers_i$: is a set containing the the current list of servers emulating the register;
- $write\_ts_i$: is the last timestamp used by c_i for a write() operation;
- r_label_i: is the label used to identify the current read() operation;
- $last_i$: is the last label used to identify a read() operation;
- $recent\_vals_i[]$: is an array of dimension n (where n is the number of servers) where the j-th entry contains the list of recent $\langle value, timestamp \rangle$ pairs received from s_j. It is used during the read() operation to build the timestamp union graph used to verify if there exists a value occurring "sufficiently enough" to be returned as the read value.
- ack_i: is a set variable used during the write() operation, where c_i collects acknowledgements from servers that accept the written values as new;
- $nack_i$: is a set variable used during the write() operation, where c_i collects non-acknowledgement messages from servers that do not accept the written values as new;
- WTS_i: is a set variable used during the first phase of the write() operation, where c_i collects the current timestamps received from servers;
- $reading_i$: is a boolean variable set to true at the beginning of a read() operation and it is reseted to false at the end.
- TSG_i: is a complex data structure representing the weighted timestamp Graph $WTsG$ built by c_i during the read() operation;
- $replies_i$: is a set variable used during the read() operation to collects $\langle values, timestamp \rangle$ pairs sent by servers;
- $recent\_labels_i[][]$: is an $n \times k$ boolean matrix (where n is the number of servers and k is the maximum number of labels that c_i can use to identify its read() operation). Such matrix is used by c_i to select a label to identify a read() operation and given the generic position (x, y), it is equal to 1 if server s_x is still processing operation labeled by ℓ_y, 0 otherwise.
- $safe_i$: is a set variable used during read() operations and it stores the identifiers of servers that are not using the label selected for the current operation;
- $slow_i$: is a set variable used during read() operations and it stores the identifiers of servers that are using the label selected for the current operation.

Local variables maintained by every server s_i. Each server s_i maintains the following local variables:

- v_i: it is the current value of the register stored locally by s_i;
- ts_i: is the current timestamp associated to the value.
- $old\_vals_i[]$: it is a sliding arrays of n entries where each server stores the last n written values. In particular,

```
operation write(v):
(01) ack_i ← ∅; nack_i ← ∅; WTS_i ← ∅
(02) for each (j ∈ servers_i) do FIFO_send GET_TS (i) to s_j.
(03) wait until (|WTS_i| ≥ n − f);
(04) write_ts_i ← Next(WTS_i);
(05) for each (j ∈ servers_i) do FIFO_send WRITE (v, write_ts_i, i) to s_j.
(06) wait until ((|ack_i| + |nack_i| ≥ n − f) ∧ (|ack_i| ≥ 2f + 1));
(07) return WRITE_CONFIRMATION.
---------------------------------------------------------------------
when TS_REPLY (ts, j) is received:
(08) if (∃⟨−, j⟩ ∈ WTS_i) then for each (⟨ts′, j⟩ ∈ WTS_i) do
(09)                                  WTS_i ← WTS_i/{⟨ts′, j⟩};
(10)                              endFor
(11) endIf
(12) WTS_i ← WTS_i ∪ {⟨ts, j⟩};
---------------------------------------------------------------------
when ACK(ts, j) is received:
(13) if (⟨ts, j⟩ ∉ nack_i) then ack_i ← ack_i ∪ {⟨ts, j⟩};
(14)                       else nack_i ← nack_i \ {⟨ts, j⟩};
(15) end if.
---------------------------------------------------------------------
when NACK(ts, j) is received:
(16) if (⟨ts, j⟩ ∉ ack_i) then nack_i ← nack_i ∪ {⟨ts, j⟩};
(17)                      else ack_i ← ack_i \ {⟨ts, j⟩};
(18) end if.
```

(a) Write Protocol: client side

```
when GET_TS (j) is received:
(01) FIFO_send TS_REPLY (ts_i, i) to c_j;
---------------------------------------------------------------------
when WRITE(v, ts, j) is delivered:
(02) if (ts_i ≺ ts) then FIFO_send ACK (ts, i) to c_j;
(03)                else FIFO_send NACK (ts, i) to c_j;
(04) endif
(05) old_vals_i[] ← update(⟨v_i, ts_i⟩);
(06) ts_i ← ts;
(07) v_i ← v;
(08) for each ⟨j, ℓ_j⟩ ∈ running_read_i do FIFO_send
REPLY (⟨i, v_i, ts_i, old_vals_i[]⟩, ℓ_j) to p_j;
```

(b) Write Protocol: server side

Figure 1: The Write Protocol for an asynchronous system

```
operation read(i):
(01) replies_i ← ∅;
(02) r_label_i ← find_read_label(recent_labels_i[][]);
(03) reading_i ← true;
(04) for each (j ∈ safe_i) do
(05)     FIFO_send READ (r_label_i, i) to s_j.
(06)     recent_labels_i[j][r_label_i] ← 1;
(07) endFor
(08) wait until (|replies_i| ≥ n − f) ∧ (replies_i ⊆ safe_i);
(09) TSG_i ← compute_ts_graph(replies_i);
(10) if (∃ node ∈ TSG_i : node.weight ≥ 2f + 1)
(11)   then reading_i ← false;
(12)        for each k ∈ safe_i do FIFO_send COMPLETE_READ(r_label_i, i);
(13)        return(node.value);
(14)   else TSG_i ←
(15)        compute_ts_union_graph(replies_i ∪ recent_vals_i[]);
(16)        if (∃ node ∈ TSG_i : node.weight ≥ 2f + 1)
(17)          then return(node.value);
(18)          else return abort;
(19)        endif
(20)        for each k ∈ safe_i do FIFO_send COMPLETE_READ(r_label_i, i);
(21)        reading_i ← false;
(22) endif
---------------------------------------------------------------------
when REPLY(⟨j, val, sn, old[]⟩, label) is delivered:
(23) if (r_label_i = label) then
(24)   replies_i ← replies_i ∪ {⟨j, val, sn, r_sn⟩};
(25)   recent_vals_i[j] ← old[];
(26) endif
(27) recent_labels_i[j][label] ← 0;
```

(a) Read Protocol: client side

```
when READ(ℓ_j, j) is delivered:
(01) running_read_i ← running_read_i ∪ {⟨j, ℓ_j⟩};
(02) FIFO_send REPLY (⟨i, v_i, ts_i, old_vals_i[]⟩, ℓ_j) to p_j;
---------------------------------------------------------------------
when COMPLETE_READ(ℓ_j, j) is delivered:
(03) running_read_i ← running_read_i \ {⟨j, ℓ_j⟩};
```

(b) Read Protocol: server side

Figure 2: The Read Protocol for an asynchronous system

given the i-th entry of the array, it stores the pair $\langle v, ts\rangle$ corresponding to the i-th write operation occurred (if exists) before than the current one, where v is the value and ts the associated timestamp.

- $running\_read_i$: is a set variable where each s_i stores information related to read operations that it knows are currently running. In particular, for any read, s_i stores a pair $\langle j, \ell_j\rangle$ where j is the identifier of the reader and ℓ_j is the label associated to the current read.

Let us note that the size of last two variables is bounded. In particular, the $old\_vals_i[]$: array stores at most n pairs $\langle v, ts\rangle$ and the $running\_read_i$ set stores a number of pairs $\langle id, lable\rangle$ that is bounded by the number of clients that is finite.

The write(v) **operation (Figure 1).** Before starting a write(v) operation, the writer client cleans its local variable and then inquiries all the servers, by sending a GET_TS() message, to know the set of current timestamps they actually store (line 05). Timestamps associated to write operations are computed according to the labeling scheme described earlier in the Section.

When a server receives the GET_TS message, it just answers by sending back its current timestamp.

Once the writer gets at least $n-f$ answers, it computes the timestamp for the current operation by invoking the Next() function that returns a label grater than all the ones received (line 04). Once the timestamp is available, the writer client sends a WRITE message to all the servers and waits until it receives at least $n-f$ answers, where at least $2f+1$ of them must be acknowledgment to the write, and then returns from the operation.

Delivering the WRITE() message, every server checks if the timestamp of the operation follows the local one according to the precedence relation $\prec$ and if it is so, it sends back an ACK message. On the contrary, it just sends back a NACK message. In any case, any server updates its local copy of the register, the local timestamp and the set of old values by shifting the previous stored one. Finally, the server forwards the new written value to all the concurrent readers stored in the $running\_read_i$ variable.

The read() **operation (Figure 2).** Before starting a read() operation, a client resets its $replies_i$ local variable and tries to get a suitable label among the bounded set it is able to use (cf. find_read_label() function description).
Once the label for the current read() operation is available, the reader c_i sends a READ() message to all the servers whose identifier is stored in the $safe_i$ variable[1] and at the same

[1]Let us note that such set is updated by the find_read_label procedure with the identifiers of servers that have no more pending messages labeled with the current label.

```
procedure find_read_label(used_label[][]):
(01) current_i ← select ℓ_j ∈ L \ last_i;
(02) safe_i ← ∅;
(03) for each (s_k ∈ servers_i);
(04)     FIFO_send FLUSH (ℓ_j) to s_k;
(05) endFor
(06) wait until Σ_{k=1}^{n} used_label[k][j] > f;
(07) for each (k : used_label[k][j] = 0)
(08)     do used_label[k][j] ← 1;
(09) endFor
(10) last_i ← ℓ_j;
(11) return last_i;
```

(a) Client Protocol

```
when FLUSH_ACK (label, k) is delivered:
(12) used_label[k][label] ← 0;
(13) if (label = current_i)
(14)    then safe_i ← safe_i ∪ {j};
(15)         if (reading_i)
(16)           then FIFO_send READ (label, i) to s_j.
(17)                used_label[k][label] ← 1;
(18)         endif
(19) endif
```

(b) Server Protocol

Figure 3: The find_label() procedure for an asynchronous system

time, it updates the $recent\_labels_i$ matrix accordingly (line 06, Figure 2(a)) and remains waiting until it gets a reply from at least $n - f$ servers belonging to the $safe_i$ set.

Delivering a READ message, every server s_k adds c_i to the list of running readers and replies by sending back its local copy of the register together with its timestamp and its list of recent write operations through a REPLY message (line 02, Figure 2(b)).

Delivering a REPLY$(\langle j, val, sn, old\rangle, \ell_j)$ message from a server s_k, c_i first checks if the received message is a reply for the current read and if it is so, it stores the received information locally (lines 23 - 26, Figure 2(a)). Then, it updates the $recent\_labels_i$ data structure to remember that server s_j has answered to a read() operation identified by ℓ_j (line 27, Figure 2(a)).

When the client is unblocked from the wait statement in line 08, it computes the local timestamps graph obtained by considering the timestamps collected in the $replies_i$ set and it looks if in the graph exists at least one node with weight at least $2f + 1$ (line 10, Figure 2(a)). If it is so, the corresponding value is returned, otherwise the union WTsG is computed by using both the timestamps collected with replies and the old ones received so far form servers (line 15, Figure 2(a)). Also in this case, the client looks if in the graph exists at least one node with weight at least $2f + 1$ and if so, it returns the corresponding value, otherwise it aborts the read operation. Finally, it sends a COMPLETE_READ message to any server to let them know that it is no more reading.

The find_read_label() **Procedure (Figure 3).** This procedure is used during the read() operation to find a label to distinguish the current read() from previous ones. The basic idea is to exploit the FIFO property of the communication channels inducing a message pattern that allows the client to understand when it can re-use a certain label. In particular, the procedure first selects a label ℓ_j, different from the last one used in the previous read() (line 01, Figure 3(a)), then it sends a FLUSH(ℓ_j) message to every server and finally it waits until the column of the $recent\_labels_i[][]$ corresponding to the label ℓ_j contains less than f entries set to 1 (line 06, Figure 3(a)).

Delivering the FLUSH message, each server s_k just answers with a FLUSH_ACK message by sending back the label together with its id (line 01, Figure 3(b)).

Delivering a FLUSH_ACK$(label, k)$ message, c_i updates its $recent\_labels_i[k][label]$ to 0 (line 12, Figure 3(a)) and then it checks if the received message is associated to the current flush procedure (line 13, Figure 3(a)). In positive case, s_k is added to the set of safe servers that can participate to the read() operation labeled by ℓ_j (line 14, Figure 3(a)) and in case the read() operation is still running, it proceeds sending a read request to s_k (and flagging again the label ℓ_j as used).

C. Correctness Proof

In the following we prove the correctness of our protocol under the following assumptions:

> **Assumption 2:** *Write operations are quiescent,* i.e. *after a burst of* write() *operations executed by the writer, there exist a sufficiently long period where the writer does not take any operation.*

Let us note that the first assumption guarantees that the algorithm works with a finite memory space bounded by the length of the write() operations burst. Such memory is in fact required by servers to store the history of write() operations and manage continuous writes performed by a fast writer (cf. $old\_vals_i[]$ variable). In the absence of this assumption, the labels may wrap around several times without the reader being able to distinguish between labels introduced by the writer at different moments.

Informally the correctness proof of the algorithm presented in Figures 1 - 2 goes as follows :

- *Termination*. For both read() and write() operations, the termination of the protocol depends on the reception of $n - f$ replies/acknowledgments carrying out the timestamp associated to the operation.
  Concerning the timestamp attached to a write() operation, it is possible to observe that it is computed starting from a subset of $n - f$ timestamps stored by servers. As a consequence, the write timestamp will be greater than them and can be recognized and counted by the client that stores locally old timestamps.
  Concerning the timestamp attached to a read() operation, it is possible to observe that a label ℓ_i can be used only after the reader is sure that it is no more used by at least $n - f$ servers. Considering that the number of Byzantine servers is bounded by f, we have that at least $n - f$ servers will eventually answer to the client and it will receives such number of replies letting the operations

terminate.

- *Validity.* Concerning the validity of a read() operation, it is obtained (as in classical BFT register implementation) by leveraging on the presence of at lest $2f + 1$ correct processes with the last value in every intersection of quorums that have acknowledged the last write() and the current read().

- *Pseudo-stabilization.* The pseudo-stabilization property follows from the fact that when the first write() operation terminates correctly, at least $3f + 1$ correct servers updated their local state. Thus, from the end of the first write(), a majority of servers cleaned up its corrupted state and any following read() is able to return a valid value.

In the following we present the detailed proof.

Lemma 1: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. If $n \geq 5f + 1$, then every write operation eventually terminates.

Proof Assume that there is a $write()$ operation that does not terminate. A client may be blocked in line 03 and line 06. Let us note that since faulty servers are at most f, any client will always receive at least $n - f$ answer and thus, the only line where the writer is blocked is in line 06 where the writer waits for a number of acknowledgements. This implies that the writer didn't receives the $2f+1$ acknowledgements (line 03 in the client code). Note that the write operation has two phases. In the first phase the writer receives at least $4f+1$ labels that are used to compute the next label. In this set there are at least $3f + 1$ labels sent by correct processes. The writer computes the next on this set then diffuses the next value and waits for at least $4f+1$ replies. Since the number of processes is $5f+1$ and the number of Byzantine processes is f, the writer receives at least $4f+1$ replies as follows : at least $2f+1$ that send an ACK (these processes are correct and participated in the first phase), f Byzantines that may send a NACK (even if their labels have been included in the computation of the next label in the previous phase) and f correct that may send a NACK since their labels have not been included in the first phase of the write(). It follows that at least $2f + 1$ processes send an ACK hence the write operation does not block. $\square_{Lemma\ 1}$

Lemma 2: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. Let op be a write(v) operation terminated at some time t labeled with the timestamp $ts_v \in L$. If $n \geq 5f + 1$ then at time t there exists at least $3f + 1$ servers storing the value v and the label ts_v.

Proof Any write operation has two phases. Following the behavior of Byzantine servers in each phase we can distinguish several scenarios:

1) Byzantine nodes reply in both phases. In this case at least $3f+1$ correct processes sent their timestamps that are included in the computation of the new timestamp. That is, the new timestamp is greater than the timestamps of at least $3f+1$ correct processes. In the second phase the Byzantine nodes can reply either by NACK or by ACK. It follows that at least $3f+1$ replies come from correct processes. Over the at least $3f + 1$ replies either the timestamps of these processes have been included in the computation of the next timestamp hence all these processes replied with an ACK and changed their local timestamp to the new one or at least $2f + 1$ replied with an ACK and changed their local timestamp and f replied with NACK and changed to the new timestamp. Overall, $3f+1$ correct servers store the new timestamp.
2) Byzantine nodes do not reply in the first phase but reply in the second phase. In this case in the first phase at least $4f + 1$ replies come from correct processes (all correct processes in the system). The new timestamp is greater than any timestamp of these processes. In the second phase, at least $3f + 1$ responses come from correct processes that reply with ACK and adopt the new timestamp.
3) Byzantine nodes reply in the first phase but not in the second phase. In this case, the new timestamp computation includes the value of the timestamps from at least $3f + 1$ correct processes. In the second phase, $4f + 1$ processes are waited and all of them adopt the new timestamp.
4) Byzantine nodes simulate crash in both phases. This case is similar to the second item. In both phases at least $4f + 1$ correct processes reply and all of them reply with an ACK message.

Overall, after a write operation for a value v with the timestamp ts_v the value v with the timestamp ts_v is stored in at least $3f + 1$ servers. $\square_{Lemma\ 2}$

Lemma 3: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. If $n \geq 5f + 1$, then the find_read_label() procedure in Figure 3 eventually returns a label $\ell_i \in \mathcal{L}$.

Proof Let us suppose by contradiction that there exists a client c_i invoking the find_read_label() procedure and that such procedure never returns a label $\ell_j \in \mathcal{L}$. It implies that c_i never executes line 09 and it remains blocked in the waiting state in line 06. it follows that in the $used\_label[][]$ matrix, the column corresponding to the current selected label always contains more that f entries equal to 1. The client modifies the $used\_label[][]$ matrix by setting the entry (k, j) to 0 when (i) it receives a REPLY message from server s_k containing the label ℓ_j (cf. line 27, Figure 2) or (ii) it receives a FLUSH_ACK message from server s_k containing the label ℓ_j (cf. line 12, Figure 3).
Thus, if the procedure does not return a label, it follows that c_i never receives "enough" REPLY and FLUSH_ACK messages. A FLUSH_ACK message is sent by any correct servers s_i when it delivers a FLUSH(ℓ_j) message that is sent by the client, at the beginning of the procedure (in line 04), to every server.
Considering that channels are FIFO, messages are not lost and

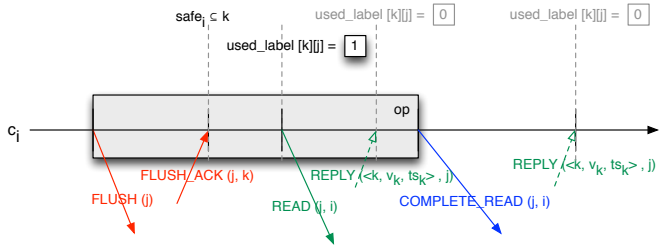

Figure 4: Projection of read() operation events at client c_i.

correct servers are at least $n - f$, it follows that eventually all correct servers triggered by the client answer with a FLUSH_ACK message that, in turns, triggers the $used\_label[][]$ matrix update and the claim follows.

$\square_{Lemma\ 3}$

Lemma 4: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. If $n \geq 5f + 1$, then when the find_read_label() procedure in Figure 3 terminates, $|safe_i| \geq n - f$.

Proof Note that by lemma the procedure find_read_label() always returns. The claim simply follows by observing that, for any execution of the find_read_label() procedure when a label ℓ_j is selected, (i) the client c_i sends a FLUSH(ℓ_j) message to any server in the system, (ii) any correct servers s_k eventually replies with a FLUSH_ACK(ℓ_j, k) message, (iii) the variable $safe_i$ is filled in by the client c_i with the identifier of any server s_k sending a FLUSH_ACK(ℓ_j, s_k) message (line 13, Figure 3) and (iv) correct servers are $n - f$. $\square_{Lemma\ 4}$

Lemma 5: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2 Let c_i be a client issuing a read() operation op_r. Let ℓ_j be the label returned by the find_read_label() procedure in Figure 3 at some time t and used by c_i during op_r. Let $\mathcal{S} \subseteq safe_i$ be the subset of correct servers that participate in op_r. For any $s_k \in \mathcal{S}$, there not exist a REPLY $(\langle k, v_k, ts_k \rangle, \ell_j)$ message, generated by a read() operation op'_r preceding op_r, that must be delivered by c_i after time t.

Proof Let us suppose by contradiction that there exists a correct server $s_k \in \mathcal{S}$ that delivers a REPLY $(\langle k, v_k, ts_k \rangle, \ell_j)$ message, generated by a read() operation op'_r preceding op_r, after time t and that op'_r and op_r are identified by the same label ℓ_j. Note that if $s_k \in \mathcal{S}$ then c_i delivered a FLUSH_ACK(ℓ_j, k) message before time t. Thus, if the REPLY $(\langle k, v_k, ts_k \rangle, \ell_j)$ message exists, its delivery to c_i follows (according to the happened-before relation) the delivery of the FLUSH_ACK(ℓ_j, k) message triggered by the current read operation op_r.

Let us consider the events generated by a read() operation at a client c_i (shown in Figure 4) and the following facts:

1) for any read() operation op executed by c_i, the send of a FLUSH(ℓ_j) message to a server s_k precedes, according to the happened-before relation, the delivery of a FLUSH_ACK(ℓ_j, k) message from s_k (cf. Figure 3).
2) if $s_k \in \mathcal{S}$, then c_i delivered a FLUSH_ACK(ℓ_j, k) message before time t (cf. line 13, Figure 3);
3) for any read() operation op executed by c_i, the delivery of a FLUSH_ACK(ℓ_j, k) message (and the insertion of s_k in the $safe_i$ variable) precedes, according to the happened-before relation, the send of a READ(ℓ_j, i) message (cf. line 05, Figure 2);
4) for any read() operation op executed by c_i, the send of a READ(ℓ_j, i) message to a server s_k precedes, according to the happened-before relation, the delivery of a REPLY $(\langle k, v_k, ts_k \rangle, \ell_j)$ message from s_k (cf. Figure 2).

Due to facts 1-4, given two read() operations op'_r and op_r such that op'_r precedes op_r, the READ(ℓ_j, i) message sent by op'_r precedes, according to the happened-before relation, the FLUSH(ℓ_j) message sent by op_r. Therefore, considering that channels connecting c_i and s_k are FIFO, it follows that necessarily s_k first sends a REPLY $(\langle k, v_k, ts_k \rangle, \ell_j)$ for op'_r and later sends a FLUSH_ACK(ℓ_j, k) message as reaction to the FLUSH(ℓ_j) sent during op_r and the claim follows.

$\square_{Lemma\ 5}$

Lemma 6: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. If $n \geq 5f + 1$, then any read() operation invoked on the regular register eventually terminates.

Proof Assume that the $read()$ operation does not terminate. This implies that the reader didn't get at least $n - f$ safe labels, which is impossible according to Lemma 4. $\square_{Lemma\ 6}$

Lemma 7: Let $(L, \prec, \mathsf{next}())$ be a k-stabilizing bounded labeling scheme used in the algorithm in Figures 1 - 2. Let op be a write(v) operation terminated at some time t and let op' be the first write() operation starting after op. If $n \geq 5f + 1$ then any read() operation starting at some time $t' > t$ returns either the last value written or a value concurrently written.

Proof From Lemma 6 every read eventually returns (terminates). Assume the read operation always aborts. We examine in the following two scenarios:

Scenario 1: no write is concurrent with the read operation. Let op be the last write that happen before the current read for the value v with the timestamp ts_v. From Lemma 2 $\langle v, ts \rangle$ is stored in at least $3f + 1$ servers. The client waits until it receives at least $4f + 1$ replies from safe servers. Over the $4f + 1$, f may come from Byzantine servers and at least $3f + 1$ from correct servers. Over these correct servers, following Lemma 2, at most f servers do not have the same value. It follows that there exists a node in the TS_graph_i that has the weight at least $2f + 1$. Hence, the read operation returns a non abort value and this value is the last written value.

Scenario 2: one or several write operations are concurrent with the read operation. The client waits for at least $4f + 1$ servers replies. f of them may come from Byzantine processes. It follows that at least $3f + 1$ come from correct servers. Assume that w_0 is the last write before the current read and $w_1, w_2, \ldots w_k$ are k concurrent writes. Since w_k started

then the values written by $w_0 \dots w_{k-1}$ are stored in at least $3f+1$ correct servers either in the current timestamp or in the old values vector. Hence, there is at least a node in the TS_graph_i computed over the replies that has a node with the weight at least $2f+1$. The value returned is either w_0 or one of the values in the set $w_1, w_2, \dots w_k$.

From above we can conclude that the read operation eventually does not abort and the returned value is always either the last written value or a concurrent write.

$\square_{Lemma\ 7}$

Theorem 2: Let $\mathcal{A}$ be the algorithm presented in Figures 1 - 2. If the labeling scheme $(L, \prec, \mathsf{next}())$ used in $\mathcal{A}$ is a k-stabilizing bounded labeling scheme and $n \geq 5f+1$ then $\mathcal{A}$ is f-Byzantine tolerant and is a pseudo-stabilizing SWMR regular register implementation.

Proof The proof follows directly from Lemma 7 $\square_{Theorem\ 2}$

D. Multi-Writers Multi-Reader Regular Register

The protocol proposed in the previous section implements the multi-writer multi-reader regular register with the following modification: each value written by a writer is associated a tuple (id, timestamp) where id is the identity of the writer and timestamp is a k-bounded label. In the following we prove that two consecutive (in the sense of the happened before relation) or concurrent write operations can be totally ordered. More specifically, we prove that write operations can be ordered as if they were issued by a single writer.

Lemma 8: Let $\mathcal{A}'$ be the algorithm presented in Figures 1 - 2 with the above modification. In any execution of $\mathcal{A}'$ concurrent or consecutive write operations are totally ordered.

Proof The use of identifiers and the bounded labeling scheme ensures that concurrent write operations can be totally ordered. Consider now $w1$ and $w2$, two consecutive operations, such that $w1$ ends before $w2$ begins and no other write operation is executed between $w1$ and $w2$. Let $ts1$ be the timestamp of $w1$. The timestamp of $w2$ is computed after $w2$ invoked the read of existing timestamps. Since $w1$ already completed, the timestamp written by $w1$ is acknowledged (either with an ack or with a nack) by at least $3f+1$ correct servers. When $w2$ starts gathering timestamps, it will wait until it gets at least $n-f$ replies. Over these $4f+1$ timestamps: f can come from the byzantine servers, f from servers that haven't yet ack/nack the $w1$ value (so they may have a different value) and $2f+1$ timestamps from servers that already store $ts1$. Hence, the timestamps computed by $w2$ is greater than $ts1$.

$\square_{Lemma\ 8}$

Theorem 3: Let $\mathcal{A}'$ be the algorithm presented in Figures 1 - 2 with the above modification. If the labeling scheme $(L, \prec, \mathsf{next}())$ used in $\mathcal{A}'$ is a k-stabilizing bounded labeling scheme and $n \geq 5f+1$ then $\mathcal{A}'$ is f-Byzantine tolerant and is a pseudo-stabilizing MWMR regular register implementation.

Proof The proof follows directly from Theorem 2 and Lemma 8. $\square_{Theorem\ 3}$

V. Related Works

Traditional solutions to construct Byzantine-tolerant storage can be divided into two categories: replicated state machines [19] and Byzantine quorum systems [20], [10], [21]. Replicated state machines typically use $2f+1$ server replicas and require that every non-faulty replica agrees to process requests in the same order [19]. Quorum systems, introduced by Malkhi and Reiter [10], do not rely on any form of agreement. They only need that relevant subsets of the replicas (*i.e. quorums*) are involved simultaneously. The authors provide a simple wait-freedom implementation of a safe register using $5f$ servers. A protocol for implementing a single-writer and multiple-reader atomic register that ensures wait-freedom using only $3f+1$ servers was later proposed [13]. This is achieved at the cost of longer (two phases) read and write operations. Finally, a multi-writer multi-reader regular register using $3f+1$ servers and unbounded timestamps was recently proposed [14].

In the context of self-stabilization, to the best of our knowledge, no previous work addresses the problem of simulating registers despite both Byzantine behaviors and memory corruption. However, the simulation of crash tolerant and self-stabilizing registers has been extensively studied. The self-stabilizing simulation of an atomic single-writer single-reader shared register in a message-passing system was proposed for the fist time by Dolev et al. [22]. This simulation does not consider that crash faults of processors may occur in the system during execution. In a different context (simulation of shared registers using shared registers with weaker properties [23], [24]), researches focused on self-stabilizing regular and safe registers implementation, and still did not consider permanent or intermittent failures. Alon et al. [25] introduced a crash-fault tolerant and "practically" stabilizing scheme for simulating atomic memory in a message passing system is presented. There, practically means that the system reaches a long enough period, that can be regarded as infinite for all purposes, in which interesting properties such are linearizability are guaranteed. Strictly, in every infinite execution suffix, linearizability is violated infinitely often, leaving open the question of suffix-closed linearizability guaranteeing algorithms that are both stabilizing and fault tolerant. Later, Dolev et al. [26] introduced a pseudo-stabilizing simulation of atomic registers for slightly weaker properties that is also tolerant to crash faults. None of the aforementioned works addresses the possibility of Byzantine behaviors.

VI. Concluding Remarks

This paper addressed the problem of emulating a shared memory, *i.e.* a register, offering the *multi-writer multi-reader* regular semantics in the presence of both Byzantine servers and possible initial corruption of correct ones. Additionally, reader clients can crash at any time while writer clients can crash at any moment if the correct servers are not in a transient phase and only after the first write completes correctly if servers are hit by transient errors. Moreover, clients (readers and writers) can be hit by transient errors at any moment.

We provided an optimal stabilizing algorithm that is able to correctly implement a MWMR regular register abstraction as soon as the number of servers responsible for the variable management is $n > 5f$ (where f is an upper bound on the number of Byzantine servers). In particular, the algorithm ensures that when transient faults hit the system, after the first write() operation completes, any subsequent read() eventually returns a value satisfying the regular semantics.

To the best of our knowledge, this is the first work on shared memory implementation dealing with both Byzantine failures and transient failures, enabling the possibility that *every* server may fail in some way. Let us note that transitory failures represent a particular case of Byzantine failures. However, we showed that distinguish among the two types and consider them separately, makes possible to provide an implementation able to tolerate f Byzantine servers and n servers suffering transitory failures by deploying $n \geq 5f+1$ servers. Let us note that, even though the number of server replicas (*i.e.* $5f+1$) is greater than the lower bound required to build a BFT regular register (*i.e.* $3f+1$), we prove that this is a tight bound needed to cope with asynchrony, the combination of two different failure models and possible failures of the writer process that can leave the system in an inconsistent state.

Moreover, we conjecture that our assumption on the quiescence of write() operations is necessary in order to achieve pseudo-stabilization with bounded timestamps in the considered multi-fault model. This conjecture is based on the asynchrony and the impossibility for the reader client to know whether stabilization is achieved. In fact, in order to define a termination condition for read() operations, we need to allow the reader to reason over a set of values (*i.e.* the whole history of the writes in case of infinite memory or the last k write operations in case of quiescence) otherwise it will not be able to distinguish if the different states it perceives from servers are due to the transitory phase (all different values due to corrupted initial states) or to the writer that continuously updates the last written value.

Finally, note that when reader clients are Byzantine our protocol still verifies the MWMR regular register specification. That is, the read protocol is performed in one phase so Byzantine readers cannot modify the value and the timestamp maintained by the correct servers.

ACKNOWLEDGEMENT

The authors are grateful to Shlomi Dolev for the many insights related to our algorithm, its correctness, lower and upper bounds on the number of faulty processes tolerated. The authors kindly thank Michel Raynal for pointing us the work [11]. The authors also thank Swan Dubois for discussions on a preliminary version of this paper.

REFERENCES

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *CACM*, vol. 17, no. 11, pp. 643–644, 1974.

[2] S. Dolev, *Self-stabilization*. MIT Press, 2000.

[3] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[4] E. Anagnostou and V. Hadzilacos, "Tolerating transient and permanent failures (extended abstract)," in *7th International Workshop on Distributed Algorithms (WDAG 1993)*, 1993, pp. 174–188.

[5] A. S. Gopal and K. J. Perry, "Unifying self-stabilization and fault-tolerance (preliminary version)," in *PODC 1993*, 1993, pp. 195–206.

[6] S. Dolev, "Possible and impossible self-stabilizing digital clock synchronization in general graphs," *Real-Time Systems*, vol. 12, no. 1, pp. 95–107, 1997.

[7] S. Dolev and J. L. Welch, "Self-stabilizing clock synchronization in the presence of byzantine faults," *Journal of the ACM*, vol. 51, no. 5, pp. 780–799, 2004.

[8] S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Stabilizing data-link over non-fifo channels with optimal fault-resilience," *Information Processing Letters*, vol. 111, no. 18, pp. 912–920, 2011.

[9] L. Lamport, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[10] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, Oct. 1998. [Online]. Available: http://dx.doi.org/10.1007/s004460050050

[11] C. Shao, E. Pierce, and J. L. Welch, "Multi-writer consistency conditions for shared memory objects," in *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*, 2003, pp. 106–120.

[12] C. Delporte-Gallet, S. Devismes, and H. Fauconnier, "Stabilizing leader election in partial synchronous systems with crash failures," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 45–58, 2010.

[13] A. Aiyer, L. Alvisi, and R. A. Bazzi, "Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions," in *Proceedings of the 26th annual ACM symposium on Principles of Distributed Computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 310–311. [Online]. Available: http://doi.acm.org/10.1145/1281100.1281147

[14] K. Kanjani, H. Lee, W. L. Maguffee, and J. L. Welch, "A simple byzantine-fault-tolerant algorithm for a multi-writer regular register," *Int. J. Parallel Emerg. Distrib. Syst.*, vol. 25, no. 5, pp. 423–435, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1080/17445761003691890

[15] A. Israeli and M. Li, "Bounded time-stamps," *Distributed Computing*, vol. 6, no. 4, pp. 205–209, 1993.

[16] D. Dolev and N. Shavit, "Bounded concurrent time-stamping," *SIAM J. on Comp.*, vol. 26, no. 2, pp. 418–455, 1997.

[17] R. Gawlick, N. A. Lynch, and N. Shavit, "Concurrent timestamping made simple," in *ISTCS 1992*, 1992, pp. 171–183.

[18] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Brief announcement: Sharing memory in a self-stabilizing manner," in *DISC 2010*, 2010.

[19] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[20] R. A. Bazzi, "Synchronous byzantine quorum systems," *Distributed Computing*, vol. 13, no. 1, pp. 45–52, Jan. 2000. [Online]. Available: http://dx.doi.org/10.1007/s004460050004

[21] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal byzantine storage," in *Proceedings of the 16th International Conference on Distributed Computing*, ser. DISC '02. London, UK, UK: Springer-Verlag, 2002, pp. 311–325. [Online]. Available: http://dl.acm.org/citation.cfm?id=645959.676126

[22] S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self-stabilizing message-driven protocols," *SIAM Journal on Computing*, vol. 26, no. 1, pp. 273–290, 1997.

[23] S. Dolev and T. Herman, "Dijkstra's self-stabilizing algorithm in unsupportive environments," in *WSS 2001*, 2001, pp. 67–81.

[24] C. Johnen and L. Higham, "Fault-tolerant implementations of regular registers by safe registers with applications to networks," in *ICDCN 2009*, 2009, pp. 337–348.

[25] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Pragmatic self-stabilization of atomic memory in message-passing systems," in *SSS'11*, 2011.

[26] S. Dolev, S. Dubois, M. G. Potop-Butucaru, and S. Tixeuil, "Crash resilient and pseudo-stabilizing atomic registers," in *OPODIS*, 2012, pp. 135–150.