



**HAL**  
open science

# Byzantine Auditable Atomic Register with Optimal Resilience

Antonella del Pozzo, Alessia Milani, Alexandre Rapetti

► **To cite this version:**

Antonella del Pozzo, Alessia Milani, Alexandre Rapetti. Byzantine Auditable Atomic Register with Optimal Resilience. 41st International Symposium on Reliable Distributed Systems (SRDS), Sep 2022, Vienna, Austria. pp.121-132, 10.1109/SRDS55811.2022.00020 . cea-04488713

**HAL Id: cea-04488713**

**<https://cea.hal.science/cea-04488713>**

Submitted on 4 Mar 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Preliminaries paper: Byzantine Tolerant Strong Auditable Atomic Register

Antonella Del Pozzo Université Paris-Saclay, CEA, List,  
F-91120, Palaiseau, France  
antonella.delpozzo@cea.fr

Antoine Lavandier Université Paris-Saclay, CEA, List,  
F-91120, Palaiseau, France  
antoine.lavandier@cea.fr

Alexandre Rapetti Aix-Marseille University,  
Université Paris-Saclay, CEA, List,  
F-91120, Palaiseau, France  
alexandre.rapetti@cea.fr

## Abstract

An auditable register extends the classical register with an audit operation that returns information on the read operations performed on the register. In this paper, we study Byzantine resilient auditable register implementations in an asynchronous message-passing system. Existing solutions implement the auditable register on top of at least  $4f+1$  servers, where at most  $f$  can be Byzantine. We show that  $4f+1$  servers are necessary to implement audibility without communication between servers, or implement does not implement strong audibility when relaxing the constraint on the servers' communication, letting them interact with each other. In this setting, it exists a solution using  $3f+1$  servers to implement a simple auditable atomic register. In this work, we implement strong auditable register using  $3f+1$  servers with server to server communication, this result reinforced that with communication between servers, audibility (event strong audibility) does not come with an additional cost in terms of the number of servers.

## I. INTRODUCTION

Outsourcing *storage* capabilities to third-party distributed storage are commons practices for both private and professional users. This helps to circumvent local space limitations, dependability, and accessibility problems. However, this opens to problems such as users that have to trust the distributed storage provider on data integrity, retrievability, and privacy. As emphasized by the relentless attacks on servers storing data [1] and by the recent worldwide advent of data protection regulations [2]–[4].

In this work, we address the problem of bringing *auditability* in a distributed storage system, i.e., the capability of detecting who has read the stored data. We consider a set of servers implementing the distributed storage and a set of clients (users) accessing it through read and write operations. Auditability implies the ability to report all the read operations performed by clients. Nevertheless, without reporting any client that did not read. Let us note that once a reader accesses a value, it can disclose it directly without being auditable. For that reason, audibility does not encompass this kind of behavior.

### A. Related work.

Most of the results of this work have already been published in [5]. In this article, we propose a new algorithm that implements a Strong Auditable Atomic Register (with completeness and strong accuracy) using  $3f+1$  servers, while in [5], the solution provides only an auditable atomic register (with completeness and accuracy).

### B. Our Contribution

Our contributions are the following:

- A new algorithm implementing a strong auditable atomic register with  $3f + 1$  servers.
- An experimentation in rust using Zenoh [6] of this algorithm.

**Paper organization.** The paper is organized as follows. Section II defines the system model. Section III formalizes the auditable register abstraction and properties. Section IV gives a lower bound on the number of servers to implement an auditable register. Section V presents an optimal resilient algorithm implementing the Auditable Atomic Register and gives the proof of its correctness. Section VI presents an approach to consider multiples writer, tolerating byzantine failure, in a special context where all correct writers aims to write the same value.

## II. SYSTEM MODEL

We consider an asynchronous message-passing distributed system composed of a finite set of sequential processes. Each process has a unique ID and is equipped with a cryptographic primitive  $\Sigma$  to sign the messages it sends. We assume that signatures are not forgeable. A process can be either a *client* or a *server*. We consider an arbitrary number of clients and  $n$  servers that implement a distributed register. The writer is a special client that owns the register and is the only one allowed to write on it. The other clients are the readers, who can read the register's content. In the following, we denote the readers as  $p_{r_1}, p_{r_2}, \dots$ , the writer (and auditor) of the register as  $p_w$  and the servers as  $s_1, \dots, s_n$ .

### A. Failure model

We consider that all correct processes follow the same protocol  $A$ . A process that executes any algorithm  $A' \neq A$  is considered Byzantine. The writer can only fail by crashing. At most  $f$  servers and any number of readers can be *Byzantine*. However, we consider that any Byzantine faulty reader does not cooperate with other faulty processes.

### B. Communication primitives

The writer broadcasts messages to servers using a reliable broadcast primitive [7]. The broadcast is done by invoking the `broadcast` primitive and the delivery of a broadcast message is notified by the event `deliver`. This primitive provides the following guarantees: **Validity**: *If a correct process deliver a message  $m$  from a correct process  $p_i$ , then  $p_i$  broadcast  $m$* ; **Integrity**: *No correct process deliver a message more than once*; **No-duplicity**: *No two correct processes deliver distinct messages from  $p_i$* ; **Termination-1**: *If the sender  $p_i$  is correct, all the correct processes eventually deliver its message*; **Termination-2**: *If a correct process deliver a message from  $p_i$  (possibly faulty) then all the correct processes eventually brb-deliver a message from  $p_i$* .

Processes can communicate with each other using a perfect point to point communication abstraction. Processes send messages by invoking the `send` primitive and the reception of a message is notified by the event `receive`. This abstraction provides the following guarantees: **Reliable delivery**: *If a correct process  $p_i$  sends a message  $m$  to a correct process  $p_j$ , then  $p_j$  eventually delivers  $m$* ; **Integrity**: *No correct process receive a message more than once*; **No creation**: *If some process  $p_i$  receives a message  $m$  with sender  $p_j$ , then  $m$  was previously sent to  $p_i$  by process  $p_j$* .

## III. SINGLE-WRITER/MULTI-READER ATOMIC AUDITABLE REGISTER

In this work we define a Single-Writer/Multi-Reader auditable atomic register, that can be transposed to the multi-writer multi-reader case [8]. In the following we first recall the atomic register specification and later we extend it with auditability.

A register  $R$  is a shared object that provides the processes with two operations,  $R.write(v)$  and  $R.read()$ . The first allows to assign a value  $v$  to the register, while the second allows the invoking process to obtain the value of the register  $R$ . Being the register a shared object, it can be concurrently accessed by processes and each operation is modeled by two events, an invocation and a response event.

We consider a single-writer/multi-reader atomic register, that can be written by a predetermined process, the *writer*, and read by any processes, the *readers*. Intuitively, atomicity provides the illusion that all the read and write operations appear as if they have been executed sequentially.

The interaction between the processes and the register is modeled by a sequence of invocation and reply events, called a history  $H$ . Without loss of generality, we assume that no two events occur at the same time. An operation is said to be complete in a history  $H$ , if  $H$  contains both the invocation and the matching response for this operation. If the matching response is missing, the operation is said to be *pending*.

A history is sequential if each operation invocation is followed by the matching response. For a given history  $H$  we denote  $complete(H)$  the history obtained starting from  $H$  by appending zero or more responses to pending invocations and discarding the remaining pending invocations.

A history  $H$  is atomic if there is a sequential history  $\pi$  that contains all operations in  $complete(H)$  such that:

- 1) Each  $read \in \pi$  returns the value of the most recent preceding write, if there is one, and otherwise returns the initial value.
- 2) If the response of an operation  $Op_1$  occurs in  $complete(H)$ , before the invocation of operation  $Op_2$ , then  $Op_1$  appears before  $Op_2$  in  $\pi$

Moreover, a history  $H$  is wait-free if every operation invoked by a correct process has a matching response. All the histories generated on Atomic Register are **atomic** and **wait-free**.

We now define the *auditable atomic register* extending the atomic register with the `audit()` operation and defining its semantics. Let us recall that only the writer can perform that operation. The `audit()` operation invocation is  $auditReq(R)$ , and its response is  $auditRep(R, Eaudit)$ , with  $Eaudit$  the list of couples process-value  $(p, v)$  reported by the audit operation.

As shown in [9], it is not possible to implement an audit operation in the presence of Byzantine servers, if a single server stores a full copy of the value. Informally, a Byzantine reader could contact only the Byzantine servers, getting the value without leaving any trace to be detected.

A possible solution to this issue, as presented in [10], is to combine secret sharing for secrecy [11] and information dispersal for space efficiency [12]. When writing a value  $v$ , the writer does not send the whole value to each server, but generates a random key  $K$  and encrypts  $v$  with it. Then, for space efficiency, the writer uses information dispersal techniques to convert the encrypted value in  $n$  parts,  $v_1, v_2, \dots, v_n$ , of size  $\frac{|v|}{\tau}$  ( $\tau$  is the number of parts needed to reconstruct the value). Finally, the writer uses secret sharing techniques to convert the key  $K$  in  $n$  shares,  $sh_1, sh_2, \dots, sh_n$ , such that the share  $sh_i$  is encrypted with the public key of the server  $s_i$ . At this point, the writer can send to the servers  $(v_1, sh_1), \dots, (v_n, sh_n)$ . Each server stores only its block and decrypted share. The secret sharing scheme assures that (1) any  $\tau$  shares are enough for a reader to reconstruct the key  $K$ , and so the value, (2) that less than  $\tau$  shares give no information on the secret. Those techniques use fingerprints to tolerate alterations by faulty processes and allow reading processes to know when they collect  $\tau$  valid blocks to reconstruct the value.

For sake of simplicity in the presentation of our solution, we avoid the details of the secret sharing scheme implementation. We consider that for any value  $v$ , the writer constructs a set of blocks  $\{b_i = (v_i, sh_i)\}_{i \in [1, n]}$ , such that a block  $b_i$  can only be decrypted by a server  $s_i$ . Any  $\tau$  blocks are necessary and sufficient to reconstruct and read the value  $v$ .

We use the notion of effectively read, introduced in [13]. This notion captures the capability of a process to collect those  $\tau$  blocks to reconstruct a value regardless it returns it or not i.e., the corresponding response event may not appear in the history.

We consider the execution  $E$ , obtained by adding to the history  $H$  the communication events: send, receive, broadcast and deliver.

**Effectively read:** A value  $v \in \mathcal{V}$  is effectively read by a reader  $p_r$  in a given execution  $E$  if and only if  $\exists$  the invocation of a write( $v$ ) operation  $\in E$  and receive( $b_{v_j}$ ) events for  $\tau$  different blocks.

We can now define the *auditability* property as the conjunction of the completeness property and the accuracy property.

- **Completeness [13] :** For every value  $v$  written into the register, every process  $p$  that has *effectively read*  $v$ , before the invocation of an audit operation  $op$  in  $H$ ,  $p$  is reported by  $op$  to have read  $v$ .
- **Strong Accuracy:** A correct process  $p$ , that never *effectively read* the value  $v$ , will not be reported by any audit operation to have read  $v$ .

The completeness property assures that if a reader  $p$  succeeds in obtaining a value  $v$  before the invocation of the audit operation, then the  $E_{audit}$  list will contain the couple  $(p, v)$ . The strong accuracy property assures that if a correct reader  $p$  never effectively read  $v$ , then the  $E_{audit}$  list will never contain the couple  $(p, v)$ .

In this paper, we propose an optimal resilient solution of the Single-Writer Multi-Reader Strong Auditable Atomic Register.

In the following, we denote the read() (resp. the write() and audit()) operation to the register as  $Op_{r_i}$  (resp.  $Op_{w_i}$  and  $Op_{a_i}$ ).

#### IV. IMPOSSIBILITY RESULTS

In this section, we first recall an impossibility result from [13] that provides a necessary condition on the number of blocks  $\tau$  to have auditability, that we extend in our system model. Finally, we show that without communication between servers, it is impossible to implement an auditable register with less than  $4f + 1$  processes. Hereafter, the impossibility result presented in [13] with the complete proof in our system model.

*Theorem 1:* Let  $\tau$  be the number of blocks necessary to recover a value written into the register  $R$ . In presence of  $f$  Byzantine servers, it is impossible to provide completeness if  $\tau < 2f + 1$ .

**Proof** Let  $c$  be a Byzantine client. To read a value from the register,  $c$  needs to collect  $\tau$  blocks. In the following, we show that, if  $\tau < 2f + 1$ , a client  $c$  can read a value  $v$  and  $\langle c, v \rangle$  is not returned by any audit operation. Consider the execution where  $c$ , during the execution of read operation  $Op$ , obtains the  $\tau$  blocks from  $f$  Byzantine servers denoted  $S_1$ ,  $\tau - 2f$  correct servers denoted  $S_2$  and  $f$  other correct servers denoted  $S_3$ . The remaining  $n - \tau$  correct servers, denoted  $S_4$ , have no information about  $Op$ .

An audit operation that starts after  $Op$  returns cannot wait for more than  $n - f$  responses from the servers. It is possible that those responses are the ones from  $S_1 \cup S_2 \cup S_4$ .  $S_1$  being the Byzantine, do not report  $Op$ . Processes in  $S_4$  have no information about the read operation of  $c$ . Then there are only the  $\tau - 2f$  servers of  $S_2$  that report process  $c$ . Since  $\tau < 2f + 1$ , there is no server that can report process  $c$  to have read  $v$ . □

Intuitively, the value of  $\tau$  has to be sufficiently big to (i) impede  $f$  Byzantine servers from collaborating and reconstructing the value and (ii) to force a reader when reading to contact sufficiently many correct servers to be auditable for that operation. Thus, the number of blocks  $\tau$  also corresponds to the number of servers that must be contacted to read. Without loss of generality, in the following we consider that each server stores at most one block for each value written.

We prove that in the absence of server to server communication and with up to  $f$  Byzantine servers, if the writer can crash then implementing an auditable register that ensures completeness requires at least  $4f + 1$  servers. Our result is proved for a safe register as defined in [14] (which is weaker than an atomic one). This result does not depend on the communication reliability.

We consider an auditable safe register, which is a safe register extended with the audit operation as defined in section III. A safe register ensures that if there is no write operation concurrent with a read operation  $Op$ ,  $Op$  returns the last value written in the register.

*Theorem 2:* No algorithm  $\mathcal{P}$  implements an auditable safe register in an asynchronous system with  $n < 4f + 1$  servers if the writer can crash and there is no server to server communication.

**Proof** Let us proceed by contradiction, assuming that  $\mathcal{P}$  exists. In particular, we consider the case of  $n = 4f$ .

Consider an execution where the writer  $p_w$  completes a write operation  $Op_w$ , and after  $Op_w$  returns, a correct reader  $p_r$  invokes a read operation  $Op_r$  which completes. Let  $v_1$  be the value written by  $Op_w$ , since  $\mathcal{P}$  exists, then  $Op_r$  returns  $v_1$ . Otherwise, we violate the safety property of the register.

As  $\mathcal{P}$  ensures the liveness property and that there are  $f$  Byzantine processes, we have that  $p_w$  cannot wait for more than  $n - f = 3f$  acknowledgments from servers before completing  $Op_w$ , i.e.,  $p_w$  cannot wait for more than  $2f$  acknowledgments from correct servers before terminating.

Let us separate servers in three groups,  $S_1$ ,  $S_2$  and  $S_3$  with  $|S_1| = 2f$ ,  $|S_2| = f$  and  $|S_3| = f$ . Servers in  $S_1$  and  $S_2$  are correct, while servers in  $S_3$  are Byzantine.

Let  $p_w$  crash after  $Op_w$  terminates but before any servers in  $S_2$  receive their block for  $v_1$ . Since servers do not communicate with each other and that  $p_w$  crashed, we can consider that no server in  $S_2$  ever receives the blocks for  $v_1$ . Then only  $2f$  correct servers, the ones in  $S_1$  have a block for  $v_1$ . Since we cannot rely on Byzantine servers and each server stores at most one block,  $p_r$  can collect at most  $2f$  blocks for  $v_1$ . According to our hypothesis,  $\mathcal{P}$  respect the safe semantic. Thus,  $p_r$  is able to read the value by collecting only  $2f$  different blocks. However, according to Theorem 1, doing so  $\mathcal{P}$  does not provide completeness, which is a contradiction.  $\square$

## V. SOLUTION SPECIFICATION

We provide an algorithm that implements a Single-Writer/Multi-Reader strong auditable wait-free atomic register.

According to the impossibility result given by Theorem 1, the writer uses information dispersal techniques, with  $\tau = 2f + 1$ . Our solution requires  $3f + 1$  servers, which is optimal resilient [15]. According to the impossibility result given by Theorem 2, we consider server to server communication, more in particular, we consider that the writer communicates with servers using a reliable broadcast abstraction. However, this nullifies the effect of using information dispersal techniques to prevent Byzantine servers from accessing the value. Indeed, all the servers would deliver the  $n$  blocks and then could reconstruct the value. To address this issue, the writer encrypts each block with the public key of the corresponding server, such that only the  $i$ -th server can decrypt the  $i$ -th block with its private key.

### A. Description of the algorithm

Messages have the following syntax:  $\langle TAG, payload \rangle$ .  $TAG$  represent the type of messages and  $payload$  is the content of the messages.

#### Variables at writer side:

- $ts$  is an integer which represents the timestamp associate to the value being written (or lastly written) into the register.
- $b_1, \dots, b_n$  are the blocks related to the value being written (or lastly written) into the register. It is such that the block in  $b_i$  is encrypted with the public key of the server  $s_i$ .

#### Variables at reader side:

All the following variables (except  $n_{seq}$ ) are reset at each new read operation invocation.

- $n_{seq}$  is an integer which represents the sequence number of the read operation of the reader  $p_r$ . This value is incremented at each read invocation.
- $Collected\_blocks$  is an array of  $n$  sets of tuple (block, timestamp). The  $i$ -th position stores all the blocks associated with their timestamps, received from server  $s_i$  in response to  $VAL\_REQ$  messages (if any).
- $Collected\_ts$  is an array of  $n$  lists of integers. In each position  $i$ , it stores the list of all the timestamp received from server  $s_i$  in response to  $TS\_REQ$  messages (if any).
- $min\_ts$  is an integer of the smallest timestamp stored in  $Collected\_ts$  that is greater than  $2f + 1$  timestamps in  $Collected\_ts$ .

#### Variables at audit side:

- $Collected\_log$  is an  $n$  dimension array that stores in each position  $i$  the log received from server  $s_i$  in response to  $AUDIT$  messages (if any). This variable is reinitialized at each audit invocation.
- $\mathcal{E}_{p_r, ts}$  is a list that stores the proof attesting that the reader  $p_r$  have read the value associated with timestamp  $ts$ .
- $E_A$  is a list that stores all the tuples process-timestamp, of all the read operation detected by the audit operation. This variable is reinitialized at each audit invocation.

### Variables at server side $s_i$ :

- $reg\_ts$  is an integer, which is the current timestamp at  $s_i$ . This value is used to prevent the reader to read an old value.
- $val$  is a list of tuple (block, timestamp) storing all the block receive by server  $s_i$ .
- $Log$  is a list of tuples reader ID, timestamp, signed either by the reader itself or by the writer. Those tuples are used as a proof that the reader effectively read.
- $Pending\_reads$  is a list of tuples, reader ID,  $n\_seq$ , that identifies all the pending read operations.

**Functions:** - **GenerateBlocks( $v$ )**. This function, invoked by the writer  $p_w$ , takes as input  $v$ , the value to write. Using information dispersal techniques, it returns an array of  $n$  encrypted blocks  $[b_1, \dots, b_n]$ , one per server. The block  $b_i$  is encrypted with the public key of the server  $s_i$ . Furthermore, to reconstruct the couple value-timestamp from the blocks, any combination of at least  $\tau = 2f + 1$  blocks are required, without what no information about  $v$  can be retrieved.

- **decrypt( $b_{v_i}$ )**. This function, invoked by server  $s_i$ , takes as input the  $i$ -th encrypted block, and using the private key of  $s_i$ , it decrypts the corresponding block and returns it.

- **GetValue( $Collected\_blocks$ )** This function, invoked by the readers, takes as input  $Collected\_blocks$ , an array of  $n$  lists of tuples blocks, timestamp. It returns a value if there are across the  $n$  lists,  $\tau$  different blocks corresponding to that value and bottom otherwise. If multiple values can be returned, then the function returns the one with the highest timestamp associated.

**The write operation (Fig 1 and Fig 3).** At the beginning of the write operation, the writer increase its timestamp and generates  $n$  blocks, one for each server, with the  $i$ -th block encrypted with the public key of server  $s_i$ . Then the writer broadcast a *WRITE* message to all servers that contains the timestamp and all the blocks. Once a correct server  $s_i$  receives a *WRITE* messages,  $s_i$  add the timestamp and the decrypted  $i$ -th block to  $val$ . Then  $s_i$  updates its timestamp in  $reg\_ts$ . Finally, the server acknowledge the writer in an *WRITE\_ACK*. Once the writer receives  $n - f$  ACKs from different servers, it terminates.

**The read operation. (Fig 2 and Fig 3)** The read operation takes place in several phases. Foremost, the reader starts by increasing its sequence number. This sequence number is included in each message the reader sends, such that only response from servers with the same sequence number are considered.

In the first phase, the reader sends *TS\_REQ* to message to all servers. When a correct server  $s_i$  receives such message, it sends its timestamp  $reg\_ts$  to the reader. After collecting  $n - f$  timestamp from different servers, the reader begins the next phase.

In the second phase, the reader sends the *VAL\_REQ* message that contains  $min\_ts$ , to all servers. Timestamp  $min\_ts$  is a timestamp received in a *TS\_RESP* message, such that it is greater than or equal to  $2f + 1$  other timestamps received. If there are more than one that respect this condition, then the reader selects the smallest timestamp that satisfies such condition (cf. line 15 figure 2). When a correct server  $s_i$  receives a *VAL\_REQ* message, then if the timestamp in  $reg\_ts$  is greater than or equal to  $min\_ts$  receives in the *VAL\_REQ* message, then  $s_i$  sends back  $min\_ts$  in a *VAL\_RESP* message. If  $reg\_ts$  is smaller than  $min\_ts$ ,  $s_i$  waits to receive such timestamp. When the reader have collected  $f + 1$   $min\_ts$  it starts the next phase.

In the third phase the reader sends a *BLOCK\_REQ* message to all servers with the timestamp collected  $f + 1$  at the preceding phases. When a correct server  $s_i$  receive such messages, it sends back the corresponding block. If it has not yet such block, it waits to receive it. When the reader receive  $n - f$  responses, it returns.

**The audit operation ((Fig 1 and Fig 3)).** Such operation is similar as the one describe in [13]. When a process  $p_a$  performed an Audit operation, it sends an *AUDIT\_REQ* messages to all server. When a server receives an *AUDIT\_REQ* messages, it sends back to  $p_a$  an *AUDIT\_RESP* with its log. Then  $p_a$  stores in  $\mathcal{E}_{p_r}$  all occurrence of  $p_r$  in the different logs it receives. If a process  $p_r$  occurs more than  $t$  times in  $\mathcal{E}_{p_r}$ , then it is added in the response  $E_A$  of the audit operation.

### B. The algorithm

#### C. Proof

In the following, we prove that Algorithms in section V-B solves the Auditable Atomic Register problem. We first show that it satisfies the atomicity property, then that it satisfies wait-freedom and finally the completeness and accuracy property.

In this section we use the following notation. Given an operation  $Op_x, x \in \{r, w, a\}$ , we denote  $t_{start}(x)$  the operation invocation time instant and as  $t_{end}(x)$  as the response time. For each couple of operations  $Op_x$  and  $Op_y$  ( $x \neq y$ ), we say that  $Op_x$  succeeds  $Op_y$  if and only if  $t_{start}(Op_x) > t_{end}(Op_y)$ . For conciseness, when  $x$  is a write or read operation, we refer to the timestamp associated to a value written by a write operation or returned by a read operation as  $Op_x.ts$ . Let us recall that,  $Op_r.ts$  it is not related to  $n\_seq$ , which is the sequence number associated to that read operation.

1) *Atomicity Proof:* According to line 3 figure 3 the following observation holds :

*Observation 1:* A correct server  $s_i$  updates  $reg\_ts$  with a new timestamp, only if this timestamp is greater than the previous one stored in  $reg\_ts$ . Hence, timestamps stored at correct servers increase monotonically

*Lemma 1:* Let  $Op_w$  be a write operation with timestamp  $ts$ . After  $Op_w$  returns the value of  $reg\_ts$  is greater than or equal to  $ts$  in at least  $f + 1$  correct servers.

**Algorithm 1** Pseudo-code of the write and audit operations at writer  $p_w$ .

---

```

Initialization
 $ts \leftarrow 0$ 
 $b_i \leftarrow \perp \forall i \in [1, n]$ 
 $Acks[i] \leftarrow \perp, \forall i \in [1, n]$ 

1: Write( $v$ ){
2:    $ts \leftarrow ts + 1$ 
3:    $[b_1, \dots, b_n] \leftarrow GenerateBlocks(v)$ 
4:   broadcast( $WRITE, ts, [b_1, \dots, b_n]$ ) to all servers
5:   wait until  $|\{x : Acks[x] = ts\}| \geq n - f$ 
6:   return
  }

7: upon receive ( $WRITE\_ACK, t$ ) from server  $s_i$ 
8:   if ( $ts = t$ ) {  $Acks[i] \leftarrow t$  }

9: Audit() {
   $Collected\_log[i] \leftarrow \emptyset, \forall i \in [1, n]$ 
10:  for  $i \in [1, n]$  send( $AUDIT\_REQ$ ) to server  $s_i$ 
11:  wait until  $|\{i : Collected\_log[i] \neq \emptyset\}| \geq n - f$ 
12:  for all  $(p_r, ts, n\_seq) \in \bigcup_{i \in [1, n]} Collected\_log[i]$ 
13:    for  $1 \leq k \leq n$  if  $(p_r, ts, n\_seq)_{\sigma_{p_r}} \in Collected\_log[k] \wedge (p_r, ts, n\_seq)_{\sigma_{p_r}} \notin E_A$ 
14:      then  $E_A \leftarrow E_A \cup (p_r, ts, n\_seq)_{\sigma_{p_r}}$ 
15:  return  $E_A$ 
  }

16: upon receive ( $AUDIT\_RESP, Log$ ) from server  $s_i$ 
17:  if  $Log = \emptyset$  then {  $Collected\_log[i] \leftarrow \perp$  }
18:  else  $Collected\_log[i] \leftarrow Log$ 

```

---

**Proof**  $Op_w$  terminates when the condition at line 5 of figure 1 is evaluated to true. Hence,  $Op_w$  terminates only if the writer received ( $WRITE\_ACK, ts$ ), from at least  $2f + 1$  different servers. A correct server, sends ( $WRITE\_ACK, ts$ ) (line 4 of figure 3) to the writer, if it receives ( $WRITE, ts, -, -, -$ ) from the writer and after the execution of line 3 in Figure 3. Since there are at most  $f$  of the  $3f + 1$  servers are Byzantine, once the write operation completes, the writer has received at least  $f + 1$  ( $WRITE\_ACK, ts$ ) from correct servers. Observation 1 concludes the proof.  $\square$

*Lemma 2:* Let  $Op_r$  be a complete read operation and let  $ts$  the timestamp corresponding to the value it returns. At any time after  $Op_r$  returns the value of  $reg\_ts$  is greater than or equal to  $ts$  in at least  $f + 1$  correct servers.

**Proof** Since  $Op_r$  terminates, it has satisfied the condition  $validBlocks(ts)$  (line 9 figure 2). For this condition to be true, the reader must have received  $2f + 1$  different valid blocks for timestamp  $ts$ , piggybacked by  $VAL\_REP$  messages (lines 18 figure 2) sent by different servers. A correct server sends such messages, only after it has  $reg\_ts \geq ts$  (lines 11 and 3). Since there are at most  $f$  Byzantine servers, and by observation 1, the claims follow.  $\square$

*Lemma 3:* Let  $Op_w$  be a complete write operation with timestamp  $ts$  and let  $Op_r$  be a complete read operation by a correct process  $p_r$  associated to a timestamp  $ts'$ . If  $Op_r$  succeeds  $Op_w$  in real-time order then  $ts' \geq ts$

**Proof** Since  $Op_r$  returns a value associated to timestamp  $ts'$ , the condition  $notOld(ts')$  (line 9 figure 2) is satisfied. In the following, we show that  $notOld(ts')$  true implies that  $ts' \geq ts$ .

Let us consider that  $notOld(ts')$  is true. As  $Collected\_ts$  is reinitialized at the beginning of each new read operation,  $notOld(ts')$  is true, if all timestamps receives from at least  $2f + 1$  different servers piggybacked by  $VAL\_RESP$  (line 18 figure 2) or  $TS\_RESP$  (line 18 figure 2) messages are smaller than or equal to  $ts'$ . According to Lemma 1, as  $Op_w$  terminates, the content of  $reg\_ts$  is greater than or to equal  $ts$  in at least  $f + 1$  correct servers. So, during  $Op_r$ , in response to ( $VAL\_REQ, n\_seq$ ) (line 13 figure 3), the reader can collect at most  $2f = n - (n - 2f)$  messages for a timestamp smaller than  $ts$ . Thus,  $notOld()$  always remain false for any timestamp smaller than  $ts$ , and since  $notOld(ts')$  is true,  $ts' \geq ts$ .  $\square$

**Algorithm 2** Pseudo-code of the read operation at reader  $p_r$ .**Definitions:**

$$\text{validBlocks}(ts) \triangleq \text{GetValue}(\text{Collected\_blocks}) = (v, ts)$$

$$\text{notOld}(ts) \triangleq |\{i : \min(\text{Collected\_ts}[i]) \leq ts\}| \geq 2f + 1$$
**Initialization :**

$$n\_seq \leftarrow 0$$

$$\text{Collected\_blocks}[i] \leftarrow (\perp, \perp), \forall i \in [1, n]$$

$$\text{Collected\_ts}[i] \leftarrow \emptyset, \forall i \in [1, n]$$

$$\text{min\_ts} \leftarrow \perp$$

```

1: Read(){
2:    $n\_seq \leftarrow n\_seq + 1$ 
3:    $\text{Collected\_blocks}[i] \leftarrow (\perp, \perp), \forall i \in [1, n]$ 
4:    $\text{Collected\_ts}[i] \leftarrow \emptyset, \forall i \in [1, n]$ 
5:    $\text{min\_ts} \leftarrow \perp$ 
6:   for  $i \in [1, n]$  send( $TS\_REQ, n\_seq$ ) to server  $s_i$ 
7:   wait until ( $|\{ts \in \text{Collected\_ts} : ts = \text{min\_ts}\}| \geq f + 1$ )
8:   for  $i \in [1, n]$  send( $BLOCK\_REQ, (p_r, \text{min\_ts}, n\_seq)_{\sigma_{p_r}}$ ) to server  $s_i$ 
9:   wait until ( $\text{validBlocks}(ts) \wedge \text{notOld}(ts)$ )
10:  return  $\text{GetValue}(\text{Collected\_blocks})$ 

11: upon receive ( $TS\_RESP, ts, num$ ) from server  $s_i$ 
12:   if ( $num = n\_seq$ )
13:      $\text{Collected\_ts}[i] \leftarrow \text{Collected\_ts}[i] \cup ts$ 
14:     if ( $|\{x : \text{Collected\_ts}[x] \neq \perp\}| \geq n - f$ )
15:        $\text{min\_ts} \leftarrow \min(\{ts \in \text{Collected\_ts} \mid \text{notOld}(ts)\})$ 
16:       for  $i \in [1, n]$  send ( $VAL\_REQ, \text{min\_ts}, n\_seq$ ) to server  $s_i$ 

17: upon receive ( $VAL\_RESP, ts, num$ ) from server  $s_i$ 
18:   if ( $num = n\_seq$ )  $\{\text{Collected\_ts}[i] \leftarrow \text{Collected\_ts}[i] \cup ts\}$ 

19: upon receive ( $BLOCK\_RESP, b_i, ts, num$ ) from server  $s_i$ 
20:   if ( $num = n\_seq$ )
21:      $\text{Collected\_blocks}[i] \leftarrow (b_i, ts)$ 

```

*Lemma 4:* If a complete read operation  $Op_r$ , invoked by a correct process  $p_r$ , returns a value corresponding to a timestamp  $ts > 0$ , then it exists a write operation  $Op_w$ , with timestamp  $ts$ , that starts before  $Op_r$  terminates.

**Proof** Since  $Op_r$  terminates, it has satisfied the condition  $\text{validBlocks}(ts)$  (line 9 of Figure 2). Thus, the reader received  $2f + 1$  distinct valid blocks for timestamp  $ts$ , piggybacked by  $BLOCK\_RESP$  messages (lines 10 and of Figure 2) sent by distinct servers. As all the variables are reinitialized at the beginning of a read operation, and as there are at most  $f$  Byzantine servers, at least  $f + 1$  correct servers sent a block to  $p_r$  during the execution of  $Op_r$ . A correct server  $s$  sends a block corresponding to a timestamp  $ts > 0$  only after it has received the corresponding  $WRITE$  message from the writer; thus, after the invocation of a write operation  $Op_w$  for timestamp  $ts$ . It follows that  $Op_w$  began before  $Op_r$  completes.  $\square$

*Lemma 5:* Let  $Op_r$  be a complete read operation invoked by a correct process  $p_r$ , and let  $Op_{r'}$  be a complete read operation invoked by a correct process  $p_{r'}$  ( $p_r$  and  $p_{r'}$  may be the same process). Let  $ts$  and  $ts'$  be the timestamps associated with  $Op_r$  and  $Op_{r'}$  respectively. If  $Op_{r'}$  succeeds  $Op_r$  in real-time order then  $ts' \geq ts$ .

**Proof** The proof follows the same approach as the proof of Lemma 3.

Let  $ts'$  be  $Op_{r'}.ts$  and let  $ts$  be  $Op_r.ts$ . As  $Op_{r'}$  returns a value associated to a timestamp  $ts'$ , the conditions  $\text{validBlocks}(ts')$  and  $\text{notOld}(ts')$  are true for timestamp  $ts$  (line 9 figure 2). In the following, we show that if condition  $\text{notOld}(ts')$  implies that  $ts' \geq ts$ .

Let us consider that  $\text{notOld}(ts')$  is true. As  $\text{Collected\_ts}$  is reinitialized at the beginning of each new read operation,  $\text{notOld}(ts')$  is true, if all timestamps receives from at least  $2f + 1$  different servers piggybacked by  $VAL\_RESP$  (line



**Algorithm 3** Pseudo-code at server  $s_i$ .

---

```

Initialization
 $reg\_ts_i \leftarrow 0$ 
 $val_i \leftarrow \emptyset$ 
 $Log \leftarrow \emptyset$ 
 $Reads\_temp_i \leftarrow \emptyset$ 

// Write Protocol Messages
1: upon receive ( $WRITE, ts, [b_1, \dots, b_n]$ ) from writer  $p_w$ 
2:    $val_i \leftarrow val_i \cup (b_i, ts)$ 
3:   if  $reg\_ts_i < ts$  { $reg\_ts_i \leftarrow ts$ }
4:   send  $WRITE\_ACK$  to  $p_w$ 

// Read Protocol Messages
5: upon receive ( $TS\_REQ, n\_seq$ ) from reader  $p_r$ 
6:   send( $TS\_RESP, reg\_ts, n\_seq$ )

7: upon receive ( $VAL\_REQ, ts, n\_seq$ ) from reader  $p_r$ 
8:   wait for ( $reg\_ts \geq ts$ )
9:   send( $VAL\_RESP, ts, n\_seq$ ) to  $p_r$ 

10: upon receive ( $BLOCK\_REQ, (p_r, ts, n\_seq)_{\sigma(p_r)}$ ) from reader  $p_r$ 
11:   wait for  $reg\_ts \geq ts$ 
12:    $Log \leftarrow Log \cup (p_r, ts, n\_seq)_{\sigma(p_r)}$ 
13:   send( $BLOCK\_RESP, val[ts], n\_seq$ ) to  $p_r$ 

// Audit protocol Messages
14: upon receive ( $AUDIT\_REQ$ ) from owner  $p_R$  of R
15:   send( $AUDIT\_RESP, Log$ ) to  $p_R$ 

```

---

▷ Pseudo code for server  $i$ 

18 figure 2) or  $TS\_RESP$  (line 18 figure 2) messages are smaller than or equal to  $ts'$ . According to Lemma 2, as  $Op_r$  returns for timestamp  $ts$ , then at server side, the content of  $reg\_ts$  is greater than or to equal  $ts$  in at least  $f + 1$  correct servers. So the reader can collect at most  $2f = n - (n - 2f)$  timestamps smaller than  $ts$ . Thus,  $notOld$  always remain false for any timestamp smaller than  $ts$ , hence  $ts' \geq ts$ . □

*Lemma 6:* Let  $Op_r$  be a complete read operation with timestamp  $ts$  and let  $Op_w$  be a complete write operation by  $p_w$  associated to a timestamp  $ts'$ . If  $Op_w$  succeeds  $Op_r$  in real-time order then  $ts' \geq ts$

**Proof** We proceed considering first the case in which  $ts > 0$  and then the case in which  $ts = 0$ . From Lemma 4, if  $ts > 0$  then it exists  $Op_{w'}$  with timestamp  $ts$  that starts before the end of  $Op_r$ . Considering that, there is a unique writer and that its execution is sequential, then  $Op_{w'}$  terminates before  $Op_w$  starts. As timestamps growth monotonically (Observation 1),  $ts' > ts$ .

Consider now the case  $ts = 0$ . As timestamps grow monotonically (Observation 1), and the initial value of the timestamp is 0 (line 0 of Figure 1) then all write operations have their timestamp greater than 0. In particular this is true for  $Op_w$ , such that  $ts' > ts$ , which concludes the proof. □

Let  $E$  be any execution of our algorithm and let  $H$  be the corresponding history. We construct  $complete(H)$  by removing all the invocations of the read operations that have no matching response and by completing a pending  $write(v)$  operation if there is a complete read operation that returns  $v$ . Observe that only the last write operation of the writer can be pending.

Then, we explicitly construct a sequential history  $\pi$  containing all the operations in  $complete(H)$ . First we put in  $\pi$  all the write operations according to the order in which they occur in  $H$ , because write operations are executed sequentially by the unique writer, this sequence is well-defined. Also this order is consistent with that of the timestamps associated with the values written.

Next, we add the read operations one by one, in the order of their response in  $H$ . A read operation that returns a value with timestamp  $ts$  is placed immediately before the write that follows in  $\pi$  the write operation associated to  $ts$  (or at the end if this write does not exist). By construction of  $\pi$  every read operation returns the value of the last preceding write in  $\pi$ . It remains to prove that  $\pi$  preserves the real-time order of non-overlapping operations.

*Theorem 3:* Let  $Op_1$  and  $Op_2$  be two operations in  $H$ . If  $Op_1$  ends before the invocation of  $Op_2$  then  $Op_1$  precedes  $Op_2$  in  $\pi$ .

**Proof**

Since  $\pi$  is consistent with the order of timestamps, we have to show that the claim is true for all operations with different timestamps.

There are four possible scenarios:  $Op_1$  and  $Op_2$  are respectively a write and a read operation, then the claim holds by Lemma 3.  $Op_1$  and  $Op_2$  are two reads operations, then the claim holds by Lemma 5.  $Op_1$  and  $Op_2$  are respectively a read and a write operation, then the claim holds by Lemma 6. If  $Op_1$  and  $Op_2$  are two write operations the claim holds by the Observation 1. □

2) *Liveness Proof:*

*Lemma 7:* If  $p_w$  is correct (if the writer don't crash), then each write operation invoked by  $p_w$  eventually terminates.

**Proof** The write operation has the following structure. The writer broadcasts a *WRITE* message to all servers (line 4 of Figure 1) and waits  $n - f$  ACKs (line 5 of Figure 1) from different servers before terminate.

Since  $p_w$  is correct, the channel communications properties assure that all correct servers deliver the message *WRITE* broadcast by  $p_w$ . Considering that: (i) servers do not apply any condition to send back *WRITE\_ACK* messages (line 4 of Figure 3), and that (ii) at most  $f$  servers can be faulty,  $p_w$  always receives the  $n - f$  *WRITE\_ACK* replies from correct servers necessary to stop waiting, which concludes the proof. □

*Lemma 8:* Let  $Op_w$  be a complete write operation that writes  $v$  with timestamp  $ts$ . If a correct server  $s$  updates  $reg\_ts$  with  $ts$ , then all correct servers eventually adds the block corresponding to  $v$  in  $val$ .

**Proof** Let us recall that a correct server updates  $reg\_ts$  with  $ts$  only upon receiving a *WRITE* message (line 1 figure 3). When a server updates  $reg\_ts$  with  $ts$ , it also add to  $val$  (line 2 figure 3) the associate block it receives in the *WRITE* message from the writer (line 1 figure 3). As there is a reliable broadcast between the writer and servers, eventually all correct servers receive the *WRITE* message and updates  $val$  with their block associate to  $ts$ . □

*Lemma 9:* When a correct reader  $p_r$  receives the response to *TS\_REQ* from all correct servers, in at least one correct server  $reg\_ts \geq min\_ts$ .

**Proof** By contradiction, assume that in all correct servers  $reg\_ts < min\_ts$ .

Then, in response to *TS\_REQ* messages, all correct servers send their  $reg\_ts$ , all inferior to  $min\_ts$ . We note  $tsMax$  the greatest timestamp receives by  $p_r$  from correct servers. Then, it exists in  $Collected\_ts$   $2f + 1$  timestamp  $\leq tsMax$ . By assumption,  $tsMax < min\_ts$ , which is in contradiction with the condition line 15. □

*Lemma 10:* A read operation  $Op_r$  invoked by a correct process  $p_r$  always terminates.

**Proof** First, observe that if  $p_r$  satisfies the conditions at line 9 figure 2,  $p_r$  terminates. Then, let us show by construction that those conditions are necessarily satisfied.

A correct reader  $p_r$  starts the read operation, after it reinitialized all the variables. Then,  $p_r$  sends a *TS\_REQ* messages to all the servers. Consider the moment  $p_r$  receives the response from the  $2f + 1$  correct servers, and sends a *VAL\_REQ* message to all the servers for timestamp  $min\_ts$ . Notice that according to Lemma 9 at least one correct server has set  $reg\_ts$  to  $min\_ts$ . As at least one correct server set  $reg\_ts$  to  $min\_ts$ , then from the reliable broadcast, eventually all servers will set  $reg\_ts$  to  $min\_ts$ . Then, the reader will eventually receive the  $2f + 1$  response from correct servers such that the condition line 7 is satisfied.

Then the reader sends *BLOCK\_REQ* messages for timestamp  $min\_ts$  to all servers. As at least one correct server set  $reg\_ts$  to  $min\_ts$ , from Lemma 8, eventually all correct servers have in  $val$  the block associate with timestamp  $min\_ts$ . Then, in response to *BLOCK\_REQ*, all correct servers can send their block corresponding to timestamp  $ts$  and the condition *validBlocks* is satisfied, such that the reader can return for the value-timestamp pair corresponding to timestamp  $ts$ . □

3) *Auditability:*

*Lemma 11:* Algorithm presented figure 1 to 3 solves the completeness property

**Proof** For a reader  $p_r$  to returns for a valid value  $v$  with timestamp  $ts$ , then  $p_r$  receives at least  $\tau$  messages from different servers (line 9 figure 2) with the block corresponding to  $v$ . A correct server sends a block with associate timestamp  $ts$  to a reader  $p_r$  (line 13, figure 3), only after it adds to its log the reader  $p_r$  associate with timestamp  $ts$ , line 12 figure 3. □

Thus, if a correct server  $s$  sends a block with associate timestamp  $ts$  to a reader  $p_r$ ,  $s$  stores  $p_r$  ID associate with  $ts$  in its log. If  $\tau \geq 2f + 1$ , since there is at most  $f$  Byzantine, in the worst case, at least  $\tau - f \geq f + 1$  correct servers, denoted  $P_C$ , records  $Op_r$  in their logs.

Let  $Op_a$  be an audit operation, invoked by a process  $p_a$ , that starts after  $p_r$  returns. So when  $Op_a$  starts,  $p_r$  is in PC's log. Then,  $p_a$  waits  $2f + 1$  responses (line 11 figure 1) after sending *AUDIT* request (line 10 figure 1) to servers. As there is at most  $f$  Byzantine servers,  $p_a$  gets the responses from at least  $f + 1$  ( $n - 2f$ ) correct servers. In particular,  $p_a$  get at least one response from a server in  $P_C$ . Finally, with  $t \leq \tau - 2f = 1$ ,  $p_r$  and is reported by  $Op_a$  to have read the value associate to timestamp  $ts$ . □

*Lemma 12:* Algorithm presented figure 1 to 3 solves the completeness (with collusion) property

**Proof** For a reader  $p_r$  to returns for a valid value  $v$  with timestamp  $ts$ , then  $p_r$  receives at least  $\tau$  messages from different servers (line 9 figure 2) with the block corresponding to  $v$ . Notice that if  $p_r$  is faulty, it can also receive those blocks not directly from the servers but for some other faulty process in  $\mathcal{B}$ . However, those other faulty process at some point must have receives those blocks from the servers.

A correct server sends a block with associate timestamp  $ts$  to a reader  $p'_r$  (line 13, figure 3), only after it adds to its log the reader  $p'_r$  associate with timestamp  $ts$  (line 12 figure 3). Thus, if a correct server  $s$  sends a block with associate timestamp  $ts$  to a reader  $p'_r$ ,  $s$  stores  $p'_r$  ID associate with  $ts$  in its log. If  $\tau \geq 2f + 1$ , since there is at most  $f$  Byzantine, in the worst case, at least  $\tau - f \geq f + 1$  correct servers, denoted  $P_C$ , records in their logs directly the reader  $p_r$ , or if  $p_r$  is faulty, some faulty process in  $\mathcal{B}$ .

Let  $Op_a$  be an audit operation, invoked by a process  $p_a$ , that starts after  $p_r$  returns. So when  $Op_a$  starts,  $p_r$  is in PC's log. Then,  $p_a$  waits  $2f + 1$  responses (line 11 figure 1) after sending *AUDIT* request (line 10 figure 1) to servers. As there is at most  $f$  Byzantine servers,  $p_a$  gets the responses from at least  $f + 1$  ( $n - 2f$ ) correct servers. In particular,  $p_a$  get at least one response from a server in  $P_C$ . Finally, with  $t \leq \tau - 2f = 1$ , if  $p_r$  is correct,  $p_r$  is reported by  $Op_a$  to have read the value associate to timestamp  $ts$ , otherwise some faulty process in  $\mathcal{B}$  are. □

*Lemma 13:* Algorithm presented figure 1 to 3 with  $t \geq 1$  solves the strong accuracy property

**Proof** We have to prove that a correct reader  $p_r$  that never invoked a read operation cannot be reported by an audit operation. With  $t = 1$ , a reader  $p_r$  is reported by an audit operation if one server respond to the *AUDIT* message with a correct record of  $p_r$  in its log. Thanks to the use of signature, a false record cannot be created by a Byzantine server. The signature used to attest the validity of a record are of two kind. If a correct server add  $p_r$  in its log before responding to *VAL\_REQ* messages (line 12, figure 3), then it uses the reader signature. So for a process to have a valid record to  $p_r$  in its log, the process  $p_r$  must have sent *VAL\_REQ* messages to some servers, i.e.  $p_r$  must have invoked a read operation. □

*Theorem 4:* Algorithm presented figure 1 to 3 with  $n = 3f + 1$ ,  $\tau = 2f + 1$  and  $t = 1$  solves the strong auditability property

**Proof** Directly from Lemma 13 and Lemma 11 with  $t = 1$  and  $\tau = 2f + 1$ . □

## VI. VALID READS USING MULTIPLE WRITERS

In this section we explore the guarantees obtained by switching from a single writer than can crash to multiples writers, with some byzantines.

We consider  $N_w \geq 2 * f_w + 1$  writers are trying to write au unique value  $v$  in an auditable distributed register. We present a protocol for the writers that is compatible with any write operation that completes in a single round like the one presented in V.

- 1 Writers perform the secret sharing in a deterministic way<sup>1</sup>.
- 2 Writers send to each server its share.
- 3 Servers wait until they receive the same share from  $f_w + 1$  distinct writers before accepting it.

Using this setup, the following properties arise :

*Theorem 5:* Correct servers only accept valid shares.

**Proof** A correct server waits until collecting  $f_w + 1$  copy of the same share before committing it to its storage. Because at most  $f_w$  writers can be byzantines, at least one correct writer communicated the share to our server and as such, the share is necessarily correct. □

<sup>1</sup>This can be achieved by using  $v$  to seed a PRNG that the writers then use to perform the secret sharing

*Theorem 6:* If one correct server accepts a share then, eventually, every correct server will accept their share.

**Proof** From the previous theorem we know that the accepted share is valid. From this we know that the writers are in the process of sending shares to every server. Because there are more than  $f_w + 1$  correct writers and we are operating in an eventually consistent network, eventually, every server will receive  $f_w + 1$  times their share and thus every correct server will *eventually* accept their share.  $\square$

With this, we basically reduced the writing process to one with a single correct writer. We therefore removed the need for protocols to be crash-tolerant, while keeping every other property they might have. In particular, this means that readers of distributed registers implementing the above write sequence know that values that they read from the register must originate from correct writers by construction, hence the name **Valid Reads**.

#### REFERENCES

- [1] I. T. R. Center, “At mid-year, u.s. data breaches increase at record pace,” in *In ITRC*, 2018.
- [2] *General Data Protection Regulation*. Regulation (EU) 2016/679 <https://gdpr-info.eu/>.
- [3] *California Consumer Privacy Act*. State of California Department of Justice <https://oag.ca.gov/privacy/ccpa>.
- [4] *Personal Information Protection Law of the People’s Republic of China*. 30th meeting of the Standing Committee of the 13th National People’s Congress of the People’s Republic of China on August 20.
- [5] A. Del Pozzo, A. Milani, and A. Rapetti, “Byzantine auditable atomic register with optimal resilience,” in *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2022, pp. 121–132.
- [6] F. Desbiens, “zenoh,” in *Building Enterprise IoT Solutions with Eclipse IoT Technologies: An Open Source Approach to Edge Computing*. Springer, 2022, pp. 155–185.
- [7] G. Bracha and S. Toueg, “Asynchronous consensus and broadcast protocols,” *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.
- [8] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
- [9] V. V. Cogo and A. Bessani, “Brief Announcement: Auditable Register Emulations,” in *35th International Symposium on Distributed Computing (DISC 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Gilbert, Ed., vol. 209. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 53:1–53:4. [Online]. Available: <https://drops.dagstuhl.de/opus/volltexte/2021/14855>
- [10] H. Krawczyk, “Secret sharing made short,” in *Annual international cryptology conference*. Springer, 1993, pp. 136–146.
- [11] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [12] M. O. Rabin, “Efficient dispersal of information for security, load balancing, and fault tolerance,” *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989.
- [13] V. V. Cogo and A. Bessani, “Auditable register emulations,” *arXiv preprint arXiv:1905.08637*, 2019.
- [14] L. Lamport, “On interprocess communication,” *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.
- [15] J.-P. Martin, L. Alvisi, and M. Dahlin, “Minimal byzantine storage,” in *International Symposium on Distributed Computing*. Springer, 2002, pp. 311–325.