

## RESEARCH ARTICLE

# A Simple Byzantine Fault-Tolerant Algorithm for a Multi-Writer Regular Register

Khushboo Kanjani, Hyunyoung Lee, Whitney L. Maguffee and Jennifer L. Welch\*

*Department of Computer Science and Engineering*  
*Texas A & M University*  
*College Station, TX 77843-3112, U.S.A.*

(October 2009, revised February 2010)

Distributed storage systems have become popular for handling the enormous amounts of data in network-centric systems. A distributed storage system provides client processes with the abstraction of a shared variable that satisfies some consistency and reliability properties. Typically the properties are ensured through a replication-based implementation. This paper presents an algorithm for a replicated read-write register that can tolerate  $f$  Byzantine faulty servers when there are at least total  $3f + 1$  replica servers. The targeted consistency condition is weaker than the more frequently supported condition of atomicity, but it is still strong enough to be useful in some important applications. By weakening the consistency condition, the algorithm can support multiple writers more efficiently and more simply than the known multi-writer algorithms for atomic consistency.

**Keywords:** distributed storage system; regularity; multi-writer; Byzantine faulty server

## 1. Introduction

Distributed storage systems have become popular for handling the enormous amounts of data in network-centric systems using, for example, the Internet [19]. A distributed storage system provides client processes with the abstraction of a shared variable (say, a read-write register) that can be accessed concurrently by multiple client processes and that satisfies some consistency and reliability properties. The distributed storage system can provide many such shared variables to handle the vast amount of data and is implemented on top of an underlying network of server nodes, which store the actual information. To ensure fault-tolerance, availability, and improved throughput, the information is typically replicated among the servers, for example, Yahoo's ZooKeeper coordination service [10].

In large-scale distributed systems, the likelihood of some components experiencing failures is quite large. Thus it is important for a distributed storage system to be fault-tolerant. In this paper, we focus on two kinds of failures. First, we consider the possibility that some fraction of the servers can become arbitrarily corrupted (i.e., Byzantine faulty). Second, we also consider the possibility that some of the client processes can become non-responsive (i.e., crash faulty). Replication is a well-known technique for fault-tolerance, but imposes its own costs in

---

\*Corresponding author. Email: [welch@cse.tamu.edu](mailto:welch@cse.tamu.edu)

terms of additional storage required and the complexity of schemes for keeping replicas consistent (cf. Chapter 7 of [9]).

The behavior of a shared register is defined by a consistency condition which is a set of constraints on values returned by data accesses when those accesses may be interleaved or overlapping. A strong consistency condition like *atomicity* (or *linearizability*) [13] gives an impression of sequential behavior, which is convenient to work with [8] but it has a high implementation cost in terms of message and time complexity [4]. A well-known weaker condition for the case of a single writer, called *regularity*, was proposed by Lamport [13]: each read of a *regular* register returns the value written by either the latest write that precedes the read or by some write that overlaps the read, but no relative ordering is imposed on the values returned by different reads. Shao et al. [16, 17] have proposed versions of regularity for multiple writers.

It appears that implementing a multi-writer register is more challenging than a single-writer one. Indeed, the majority of the fault-tolerant implementations of multi-writer registers simply apply multiple copies of a single-writer protocol, one copy for each writer. There are two major limitations of this scheme: the implementation cost is proportional to the number of writers, and there is a fixed upper bound on the number of writers allowed. To overcome these limitations, in this paper, we focus on a direct implementation of a multi-writer register.

**Our contribution:** This paper presents an algorithm for a replicated read-write register that can tolerate Byzantine failures of up to a third of the replica servers and crash failures of any number of clients. The level of fault tolerance achieved is optimal as it matches the lower bound in [15]. The targeted consistency condition is a version of regularity that supports multiple writers. Although regularity is weaker than the more frequently supported condition of atomicity, it is still strong enough to be useful in some important applications such as mutual exclusion [16, 17]. By weakening the consistency condition, the algorithm can support multiple writers more efficiently and more simply than the known multi-writer algorithms for atomic consistency.

## 2. Related Work

Distributed storage systems have been an active area of research, and various lower bounds have been proved and protocols proposed for different system models.

The following lower bounds are known for the distributed storage problem. Martin et al. [15] have proved that any storage protocol tolerant of  $f$  Byzantine faulty servers requires at least  $3f + 1$  servers total to ensure safe<sup>1</sup> semantics and liveness; this result also holds for randomized protocols and for self-verifying data (data that cannot be undetectably altered, e.g., digitally signed data). Guerraoui and Vukolic [7] showed that more than one message round trip is required for the read protocol for any implementation of a safe register that tolerates any number of client crashes and up to  $f \geq n/4$  Byzantine servers when servers are inactive (and thus may not send unsolicited messages to readers). This result does not apply to our algorithm because the servers are not inactive. Abraham et al. [1] proved that implementing a shared register on top of other base shared objects requires at least two rounds of operations on the base objects for a write.

---

<sup>1</sup>*Safety* is an even weaker consistency condition than regularity [13], and thus the result holds also for all stronger conditions, including regularity and atomicity.

Several algorithms for Byzantine-fault-tolerant distributed storage have been proposed. The multi-writer algorithm in [14], which is based on Byzantine quorum systems, provides regular semantics for self-verifying data (data that cannot be undetectably altered, e.g., digitally signed data) and the weaker condition of safety when making no assumption of self-verifying data; the algorithms support multiple writers although the multi-writer versions of the consistency conditions are subject to multiple interpretations. The multi-writer algorithms in [3, 5, 6] all provide atomic semantics. Aiyer et al. [3] and Bazzi and Ding [5] give multi-writer register implementations by simulating  $m$  copies of the single-writer protocol where  $m$  is the number of writers. Cachin and Tessaro's algorithm [6] requires communication among the servers and digital signatures infrastructure. We focus on designing a multi-writer register directly without using multiple copies of a single writer register algorithm and without server-to-server communication. However, our algorithm provides a weaker consistency condition than atomicity and assumes reliable broadcasting. Another direction for weakening atomicity in the presence of Byzantine servers, proposed by Aguilera and Swaminathan [2], is to provide a register whose operations can be aborted by the clients.

Shao et al. [16, 17] initiated the study of how to extend the classic definition of regularity, which only considered a single writer process, to the case of multiple writers. Several different definitions were proposed, each with an accompanying protocol, and the comparative usefulness of the definitions for solving mutual exclusion was discussed. The protocols in these papers do not tolerate Byzantine failures of servers. We use a variant of one of these definitions, which offers a good tradeoff between efficiency of implementation and usefulness in applications.

Earlier versions of the algorithm in this paper appeared in [11, 12]; these earlier versions required  $n > 4f$ .

### 3. Definitions

A shared read/write register  $x$  supports two kinds of operations that can be invoked by users: a read, with invocations of the form  $read_i(x)$ , where  $i$  indicates the user, and responses of the form  $return_i(x, v)$ , where  $v$  is the value returned; and a write, with invocations of the form  $write_i(x, v)$ , where  $i$  indicates the user and  $v$  the value to be written, and responses of the form  $ack_i$ .

We consider a distributed system containing  $n$  server processes and any number of client processes. Each client works on behalf of a user by accepting invocations to read or write the shared register, executing some code, communicating with servers, and deciding how and when to provide a response to the user. Servers store the data and the clients communicate with servers to read and write the data. The interprocess communication is by passing messages over an asynchronous communication network. Such a model is suitable for wide-area networks, since there are no timing assumptions on the delay in passing a message. We assume reliable, FIFO communication channels between clients and servers, properties which can be approximated in practice. We also assume that at most  $f$  servers can be Byzantine faulty, where  $n > 3f$ , but when a Byzantine process sends a message it cannot hide or alter its id as the sender of the message. Furthermore, any number of clients can fail by crashing. In addition, it is assumed that broadcasts are reliable: if a process starts sending a message to a set of processes, all processes in the set are assured to receive the message.

Given an execution  $\alpha$  of the system, the projection of  $\alpha$  onto the invocations and responses of the reads and writes produces a **schedule** (denoted  $\sigma$ ). The goal is for the schedule of every execution to satisfy **mw-regularity**, i.e., to satisfy the

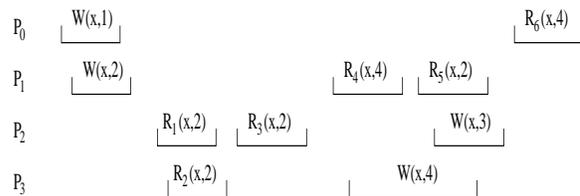


Figure 1. A schedule that satisfies mw-regularity

following three conditions.

**Well-Formedness:** For every client  $i$ , the projection of  $\sigma$  onto the events involving  $i$ , denoted  $\sigma|i$ , is a sequence of alternating invocations and responses, beginning with an invocation.

**Non-faulty Liveness:** For every client  $i$ ,  $\sigma|i$  ends with an invocation only if  $i$  is faulty (crashes).

We require the existence of a set  $ops(\sigma)$  consisting of all the completed operations in  $\sigma$  (matching invocations and responses) and *some* of the pending writes in  $\sigma$  (invocations without matching responses). The consistency condition of interest will be defined with respect to  $ops(\sigma)$ .

**MWR<sub>eg</sub>:** For each read operation  $r$  in  $ops(\sigma)$ , there exists a total order  $\tau_r$  on the set consisting of  $r$  and all write operations in  $ops(\sigma)$  such that

- $\tau_r$  is **legal**, meaning that  $r$  returns the value of the write that immediately precedes it in the total order (if  $r$  appears first, then it returns the initial value); and
- $\tau_r$  is  **$\sigma$ -consistent**, meaning that for all operations  $op_1$  and  $op_2$  in  $ops(\sigma)$ , if  $op_1$  ends before  $op_2$  begins in  $\sigma$ , then  $op_1$  precedes  $op_2$  in  $\tau_r$ .

Furthermore, for all reads  $r_1$  and  $r_2$  in  $ops(\sigma)$ ,  $\tau_{r_1}$  and  $\tau_{r_2}$  agree on the ordering of all writes that are relevant to both  $r_1$  and  $r_2$ , where a write  $w$  is **relevant** to a read  $r$  if  $w$  begins in  $\sigma$  before  $r$  ends.

Fig. 1 gives an example of a schedule that satisfies mw-regularity. Possible total orders for all the read operations are as follows:

- $R_1$ :  $W(x, 1), W(x, 2), R_1(x, 2)$
- $R_2$ :  $W(x, 1), W(x, 2), R_2(x, 2)$
- $R_3$ :  $W(x, 1), W(x, 2), R_3(x, 2)$
- $R_4$ :  $W(x, 1), W(x, 2), W(x, 4), R_4(x, 4)$
- $R_5$ :  $W(x, 1), W(x, 2), R_5(x, 2), W(x, 3), W(x, 4)$
- $R_6$ :  $W(x, 1), W(x, 2), W(x, 3), W(x, 4), R_6(x, 4)$

This schedule is not atomic, since under atomicity if read  $R_4$  obtains the new value 4 of the register, the subsequent read  $R_5$  should not obtain the old value 2.

As shown in [16], if there is only one writer, then mw-regularity reduces to the standard definition of (single-writer) regularity. The relationships between mw-regularity and a number of other consistency conditions are demonstrated in [16, 18]. In particular, mw-regularity is incomparable with both sequential consistency and with causal consistency. The intuition for the incomparability of mw-regularity and sequential consistency is that mw-regularity, unlike sequential consistency, requires every total order to respect the real-time order of non-overlapping operations; on the other hand, sequential consistency, unlike mw-regularity, requires the existence of single total order of operations.

#### 4. Algorithm

In this section we describe the three parts of the algorithm: the reader protocol (Alg. 1), the writer protocol (Alg. 2), and the server protocol (Alg. 3). The interactions between a reader and the servers are illustrated in Fig. 2 while the interactions between a writer and the servers are illustrated in Fig. 3. The algorithm was inspired by that in [3].

---

##### Algorithm 1 Reader's protocol

---

```

1: Initially:
2:    $my\_ctr := 0$ 
3:
4: when read() occurs:
5:    $my\_ctr := my\_ctr + 1$ 
6:    $ts\_arr[s] := \perp, val\_arr[s] := \emptyset$  for each server  $s$ 
7:   send (GET_INFO,  $my\_ctr$ ) to all servers
8:
9: when (INFO,  $t, \langle v, ts \rangle$ ) message is received from server  $s$ :
10:  if ( $t = my\_ctr \wedge ts\_arr[s] = \perp$ ) then
11:     $ts\_arr[s] := ts$ 
12:     $val\_arr[s].add(\langle v, ts \rangle)$ 
13:    check()
14:  end if
15:
16: when (FWD,  $\langle v, ts \rangle$ ) is received from server  $s$ :
17:    $val\_arr[s].add(\langle v, ts \rangle)$ 
18:   check()
19:
20: procedure check():
21:  if ( $|\{s: ts\_arr[s] \neq \perp\}| \geq n - f$ ) then
22:    if there exists a pair  $\langle v, ts \rangle$  such that
23:      (not_old):  $ts$  is greater than or equal to at least  $2f + 1$  entries of  $ts\_arr$ 
24:      and
25:      (valid):  $\langle v, ts \rangle$  is contained in at least  $f + 1$  entries of  $val\_arr$  then
26:        send (DONE) to all servers
27:        return  $\langle v, ts \rangle$ 
28:    end if
29:  end if

```

---

*Reader's Protocol.* The timestamp  $ts$  is in the form of a pair  $\langle counter, id \rangle$  where  $counter$  is a non-negative integer and  $id$  is the id of the client process that performs the read. The  $ts\_arr$  array has an entry for each server. The entry for server  $s$  stores the first value received from  $s$  in an INFO message since the read began. The  $val\_arr$  array has an entry for each server. The entry for server  $s$  contains all values (together with their timestamps) that are received from  $s$  in INFO and FWD messages since the read began. For both arrays, in order for an INFO message not to be ignored, it must be tagged with a counter indicating that it is in response to the reader's GET\_INFO message for the current read.

When a read is invoked, the reader sends a GET\_INFO message to each server requesting the value and timestamp stored at that server. Upon receiving an INFO

message from a server, the reader updates the  $ts\_arr$  and  $val\_arr$  arrays appropriately. Upon receiving a `FWD` message from a server, the reader updates the  $val\_arr$  array with the new information, but does not change the  $ts\_arr$  array; this distinction is crucial to the correctness of the algorithm. Once the reader receives replies from  $n - f$  servers, the reader begins checking to see if any value it has stored in its  $val\_arr$  array meets the termination condition.

**Read Termination Condition.** The reader can stop and return a value once a value is found that meets both the **valid** and **not\_old** conditions. A value meets the **valid** condition when  $val\_arr$  indicates that more than  $f$  servers have sent that value to the reader. A value is **not\_old** when it has a timestamp that is greater than or equal to the timestamps recorded in  $ts\_arr$  for at least  $2f + 1$  servers. Once a write completes, at most  $f$  servers can hold old timestamps because a write only completes once the writer has received acknowledgments from at least  $n - f$  servers.

Byzantine servers may send `INFO` or `FWD` messages containing incorrect timestamps and values. Nevertheless, the termination condition ensures that the value returned by a read was written by some actual write (i.e., is **valid**), and furthermore, this write did not occur too far in the past (i.e., is **not\_old**). Specifically, if  $w$  is the write that wrote the return value, then there is no other write  $w'$  that started after  $w$  finished and ended before the read began.

A value meets the **valid** condition when  $val\_arr$  indicates that more than  $f$  servers have sent that value to the reader. Thus at least one nonfaulty server sent the value and so the value was written by some write (and not manufactured by a faulty server).

A value meets the **not\_old** condition when its timestamp is at least as large as the timestamps recorded in  $ts\_arr$  for at least  $2f + 1$  servers. Once  $w$  has finished, at least  $n - f$  servers have the value written by  $w$ , since  $w$  does not finish until the writer has received acknowledgments from at least  $n - f$  servers. Thus, at most  $f$  servers have older values. The maximum number of old timestamps that the reader can store in its  $ts\_arr$  is  $2f$ ,  $f$  from the nonfaulty, but out-of-date, servers, and  $f$  from the Byzantine servers sending values with artificially low timestamps<sup>1</sup>. Thus the **not\_old** condition will not be met for an out-of-date value.

**Writer's Protocol.** When a writer attempts a write, it first does a read to obtain a recent timestamp. The counter component of the timestamp for the current write is chosen to be one greater than the counter component of the timestamp obtained from the read. After constructing the timestamp, the writer sends a `WRITE_INFO` message to each server. It is assumed that all messages sent by the writer are delivered. With this assumption, the crash of a writer will not cause discord in the system. The writer then waits for  $n - f$  servers to acknowledge receipt of the new value and then the write terminates.

**Server's Protocol.** Each server keeps in its local state a value  $rval$ , a timestamp  $rts$ , and a set of readers  $readers$ . The variable  $rval$  contains the value written and the variable  $rts$  keeps the timestamp (counter and id of the writer) associated with the value. Once a `GET_INFO` message is received at a server, the server sends  $rval$  and

---

<sup>1</sup>If, on the other hand, the Byzantine servers send timestamps that are artificially high, the **valid** check will not allow the corresponding (artificial) value to be returned.

**Algorithm 2** Writer's protocol

---

```

1: Initially:
2:    $my\_ctr := 0$ 
3:
4: when write( $v$ ) occurs:
5:    $my\_val := v$ 
6:    $recv\_ack[s] := \text{false}$  for all servers  $s$ 
7:    $\langle v', \langle r\_ctr, id \rangle \rangle := \text{read}()$  // invoke a read to find a recent timestamp
8:    $my\_ctr := r\_ctr + 1$ 
9:   send(WRITE_INFO,  $\langle my\_val, \langle my\_ctr, my\_id \rangle \rangle$ ) to all servers
10:
11: when (WRITE_ACK,  $t$ ) is received from server  $s$ :
12:   if ( $t = my\_ctr$ ) then
13:      $recv\_ack[s] := \text{true}$ 
14:     if ( $|\{s:recv\_ack[s] = \text{true}\}| \geq n - f$ ) then
15:       return ACK
16:     end if
17:   end if

```

---

$rts$  to the reader that made the request and adds the reader to *readers*. Once the server receives a DONE message from a reader, the server removes that reader from *readers*. When a writer sends information to the server, the server updates *rval* and *rts* if the new value has a higher timestamp and sends an acknowledgment to the writer. If *readers* is not empty, the server forwards the value and timestamp to each member of *readers*.

**Algorithm 3** Server's protocol

---

```

1: Initially:
2:    $readers := \emptyset$ 
3:    $rval := 0$ 
4:    $rts := \langle 0, 0 \rangle$ 
5:
6: when (GET_INFO,  $t$ ) message is received from reader  $r$ :
7:    $readers.add(r)$ 
8:   send (INFO,  $t, \langle rval, rts \rangle$ ) to  $r$ 
9:
10: when (DONE) message is received from reader  $r$ :
11:    $readers.remove(r)$ 
12:
13: when (WRITE_INFO,  $\langle v, \langle ctr, w \rangle \rangle$ ) is received from writer  $w$ :
14:   if ( $rts < \langle ctr, w \rangle$ ) then //new value  $v$  has later timestamp
15:      $rval := v$ 
16:      $rts := \langle ctr, w \rangle$ 
17:   end if
18:   for each  $r$  in  $readers$  do
19:     send (FWD,  $\langle v, \langle ctr, w \rangle \rangle$ ) to  $r$ 
20:   end for
21:   send (WRITE_ACK,  $ctr$ ) to  $w$ 

```

---

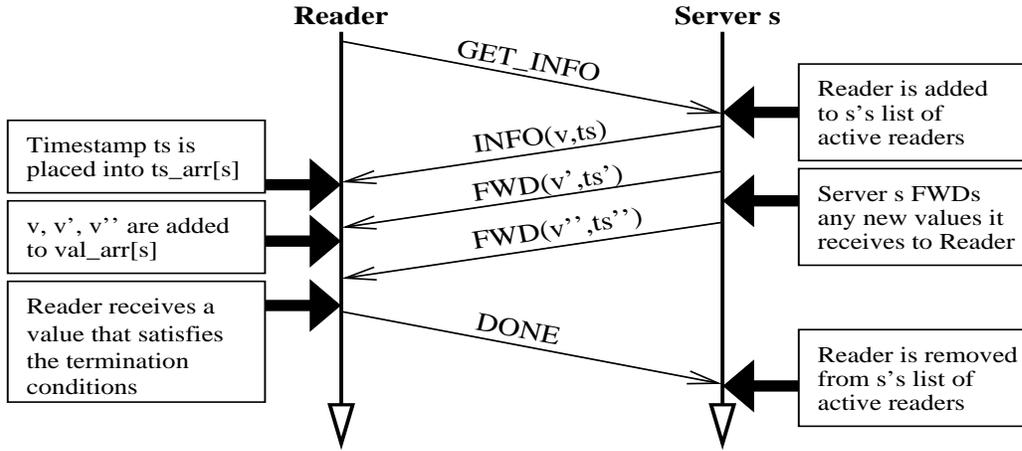


Figure 2. Reader – Server Interaction

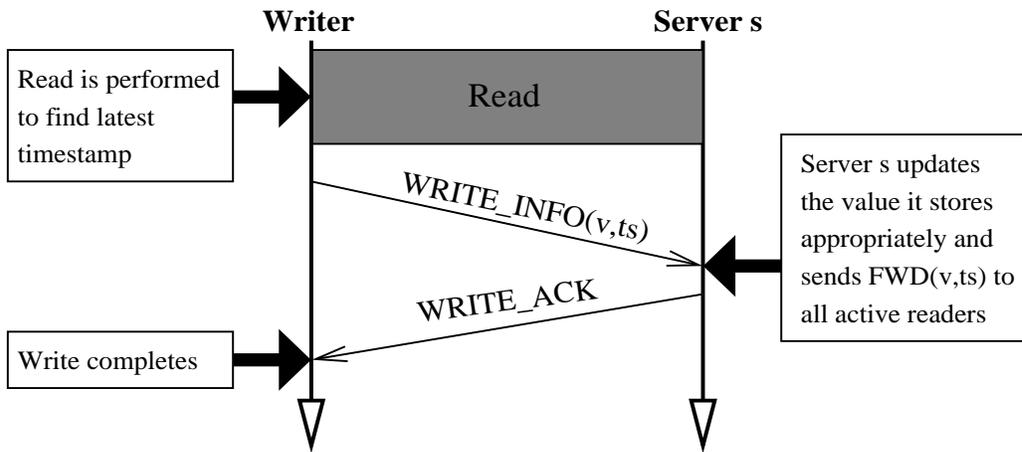


Figure 3. Writer – Server Interaction

### 5. Analysis

Now we prove that our algorithm is correct, in that it satisfies mw-regularity. We also analyze the complexity of our algorithm.

#### 5.1 Correctness

Consider any execution  $e$  of the algorithm in which at most  $f$  of the  $n$  servers are Byzantine faulty, for  $n > 3f$ , and any number of clients crash. Well-Formedness is maintained by the algorithm, since a return is executed at a client only if a read is pending at that client, and an ack is executed at a client only if a write is pending at that client.

To show Non-faulty Liveness, we prove next that every operation invoked by a non-faulty client eventually terminates.

**THEOREM 5.1** *Every read operation invoked by a non-faulty client terminates.*

*Proof* Suppose for contradiction that some read  $r$  in  $e$  by some non-faulty client process  $p$  never terminates. Because  $r$  never terminates,  $p$  eventually hears from all non-faulty servers and all of the Byzantine servers which ever respond to its GET\_INFO request. After this point,  $p$ 's  $ts\_arr$  no longer changes. Call this point in time  $t_{stop}$ . Let  $vts$  be the  $(2f + 1)^{st}$  smallest timestamp in  $p$ 's  $ts\_arr$  at time  $t_{stop}$ . Let

$T$  be the largest timestamp from a non-faulty server in  $p$ 's  $ts\_arr$  after  $t_{stop}$ .

*Claim.*  $T$  is greater than or equal to  $vts$ .

*Proof.* Suppose in contradiction  $T$  is smaller than  $vts$ . Then all timestamps from non-faulty servers are smaller than the  $(2f+1)^{st}$  smallest entry in  $ts\_arr$ . It follows that there are at most  $2f$  timestamps from non-faulty servers in  $ts\_arr$  at time  $t_{stop}$ . Since  $p$  has heard from all non-faulty servers by time  $t_{stop}$ , this contradicts the fact that there are at least  $2f + 1$  non-faulty servers because there are more than  $3f$  servers overall.

Let  $w$  be the write whose timestamp is  $T$  and let  $v$  be the value written by  $w$ . Let  $\langle v, T \rangle$  be the value-timestamp pair sent in the `WRITE_INFO` message of  $w$  to all servers. Note that  $w$  succeeds in sending its `WRITE_INFO` message, and by our assumption about reliable broadcasts, all non-faulty servers eventually receive the `WRITE_INFO` message of  $w$ .

Now we show that, in contradiction to our assumption that  $r$  never terminates, the value-timestamp pair  $\langle v, T \rangle$  eventually satisfies **not\_old** and **valid** and thus  $r$  does terminate.

First we show that, for every non-faulty server  $s$ ,  $\langle v, T \rangle$  appears in the  $val\_arr[s]$  set of  $p$  during the execution of  $r$ . Consider any non-faulty server  $s$  whose entry in  $ts\_arr$  at time  $t_{stop}$  is  $T$ . It follows that  $p$  received an `INFO` message containing  $\langle v, T \rangle$  from  $s$  after  $r$  began. Thus  $p$  stored  $T$  in  $ts\_arr[s]$  and  $v$  in  $val\_arr[s]$  during  $r$ .

Now consider any non-faulty server  $s'$  whose entry in  $ts\_arr$  at time  $t_{stop}$  is  $T' \neq T$ . Because  $T$  is the highest timestamp in the array at time  $t_{stop}$  sent by a non-faulty server,  $T'$  must be smaller than  $T$ . This means that when  $s'$  sent its `INFO` message to  $p$ ,  $T'$  was the highest timestamp associated with a write for which  $s'$  had received a `WRITE_INFO` message. This is the case because servers store values in order of increasing timestamp. Thus,  $s'$  had not received the `WRITE_INFO` message for  $w$  when  $s'$  sent its `INFO` message with  $T'$  to  $p$ . Because  $s'$  is non-faulty and the `WRITE_INFO` message for  $w$  is ensured to reach  $s'$ ,  $s'$  eventually receives the `WRITE_INFO` message for  $w$  and sends a `FWD` message to  $p$  containing  $\langle v, T \rangle$ . When  $p$  receives this message,  $\langle v, T \rangle$  is placed into  $val\_arr[s']$ . Thus  $v$  eventually is in  $val\_arr[s]$  for every non-faulty server  $s$ . So  $v$  will be in  $val\_arr[s]$  for at least  $2f + 1$  servers and  $v$  will satisfy **valid**. Based on the Claim,  $\langle v, T \rangle$  satisfies **not\_old**. Thus  $\langle v, T \rangle$  meets both conditions and  $r$  terminates, which is a contradiction. ■

**THEOREM 5.2** *Every write operation invoked by a non-faulty client terminates.*

*Proof* Consider any write operation invoked by a non-faulty client  $p$ . The embedded read operation completes by Theorem 5.1. The write completes once  $p$  has received an acknowledgment from  $n - f$  distinct servers. Because only  $f$  servers can be Byzantine, there are at least  $n - f$  non-faulty servers. Because communication is reliable and  $p$  sends a `WRITE_INFO` message to each server, each of the  $n - f$  non-faulty servers eventually receives the `WRITE_INFO` message and sends an acknowledgment to  $p$ . These messages are received by  $p$  and the write terminates. ■

We now show that the execution satisfies MWReg. First, we give an intuitive overview of why MWReg is satisfied. For each read we construct a total order on a particular subset of the operations. By definition, the construction is legal. The bulk of the proof is to show that the construction is  $\sigma$ -consistent. To do this, we must show that every pair of operations that do not overlap in the execution are placed in the same order in the construction. The case of a read followed by a write works correctly because of the way the read's **valid** check uses  $val\_arr$  (cf. Lemma 5.5). The case of a write followed by a read works correctly because of the

way the **not\_old** check uses  $ts\_arr$  (cf. Lemma 5.3). The case of a write followed by a write works correctly because of the way the timestamps are used (cf. Lemma 5.4).

Let  $\sigma$  be the schedule of the execution. Let  $ops(\sigma)$  be the set of all completed operations and all incomplete but “effective” writes in  $\sigma$ . A write is **effective** if it executes Line 9 of the writer’s protocol, i.e., if it succeeds in broadcasting its `WRITE_INFO` message to all the servers. Let  $ts(op)$  be the timestamp of operation  $op$  (if  $op$  is a write, it is the timestamp assigned to the value; if  $op$  is a read, it is the timestamp associated with the value returned).

**LEMMA 5.3** *For each read operation  $r$  in  $ops(\sigma)$  and each write operation  $w$  in  $ops(\sigma)$ , if  $w$  ends before  $r$  begins, then  $ts(w) \leq ts(r)$ .*

*Proof* Fix a read operation  $r$  and a write operation  $w$  in  $ops(\sigma)$  such that  $w$  ends before  $r$  begins. Let  $p$  be the client process that executes  $r$ . After  $w$  has completed, at least  $n - f$  servers have  $rts \geq ts(w)$  because of the counter value included in the `WRITE_ACK` message. Thus at most  $f$  non-faulty servers have  $rts < ts(w)$  after  $w$  has completed. Since the  $rts$  variable at each non-faulty server changes over time only by increasing, and since the counter value is included in the `GET_INFO` and `INFO` messages, When  $r$  is executed,  $p$  receives timestamps less than  $ts(w)$  from at most  $2f$  servers. Thus **not\_old**( $T$ ) cannot be true for any  $T < ts(w)$  during the execution of  $r$ , and  $ts(r)$  must be at least  $ts(w)$ . ■

**LEMMA 5.4** *The write operations in  $ops(\sigma)$  are totally ordered by timestamp and this total order is  $\sigma$ -consistent.*

*Proof* The use of the process id together with the incrementing of the counter in the timestamp ensures that every write has a unique timestamp.

Suppose write  $w_1$  ends before write  $w_2$  begins in  $\sigma$ . Since  $w_2$  encompasses a complete read, call it  $r$ , to decide its timestamp, Lemma 5.3 ensures that  $ts(r) \geq ts(w_1)$ . Since  $ts(w_2)$  is created by adding one to the counter in  $ts(r)$ , it follows that  $ts(w_1) < ts(w_2)$ . ■

Let setting the initial values for  $rval$  and  $rts$  in the servers to zero and  $\langle 0,0 \rangle$  respectively, be considered the first write.

**LEMMA 5.5** *For every read operation  $r$  in  $ops(\sigma)$ , there exists a write operation  $w$  in  $ops(\sigma)$ , denoted  $\rho(r)$ , that is relevant to  $r$  such that the timestamps of  $r$  and  $w$  are the same and the value returned by  $r$  is the value written by  $w$ .*

*Proof* The value  $v$  returned by a client process  $p$  at the end of its execution of a read  $r$  must satisfy the **valid** condition. The **valid** condition requires that at least  $f + 1$  servers have sent the same value before  $p$  may return it. This ensures that at least one non-faulty server  $s$  has sent  $v$ . Because  $s$  is non-faulty,  $v$  is written by a write. Because  $r$  is still in progress when  $p$  receives  $v$ , the write of  $v$  must have started before  $r$  ends. ■

Consider any read  $r$  in  $ops(\sigma)$ . We construct a total order  $\tau_r$  on the set consisting of  $r$  and all writes in  $ops(\sigma)$  as follows.

- Order all the writes in  $ops(\sigma)$  that are relevant to  $r$  before any write that is not relevant to  $r$ .
- Order all the writes that are relevant to  $r$  in timestamp order among themselves.
- Order all the writes that are not relevant to  $r$  in timestamp order among themselves.
- Order  $r$  immediately after  $\rho(r)$ , from Lemma 5.5, and before the following write.

The total order  $\tau_r$  is legal by construction (the rule explicitly states to place the unique read  $r$  in the proper place).

Table 1. Complexity of the read/write operations

	read()	write()
Rounds	1	2
Messages	$3n$	$5n + R_c$
Message size	constant	constant

We now show that  $\tau_r$  is  $\sigma$ -consistent. Consider any two writes  $w_1$  and  $w_2$  where  $w_1$  ends in  $\sigma$  before  $w_2$  begins. If both writes are relevant to  $r$  or both writes are not relevant to  $r$ ,  $\sigma$ -consistency of  $\tau_r$  follows from Lemma 5.4. If  $w_1$  is relevant to  $r$  and  $w_2$  is not relevant to  $r$ ,  $\sigma$ -consistency of  $\tau_r$  follows from the fact that  $w_1$  starts before  $r$  ends and  $w_2$  starts after  $r$  ends, and thus  $w_2$  cannot end before  $w_1$  starts.

Consider any write  $w$  that starts after  $r$  ends in  $\sigma$ . Let  $w' = \rho(r)$ , from Lemma 5.5. Since  $w'$  is relevant to  $r$  but  $w$  is not,  $w$  appears after  $w'$  in  $\tau_r$ . Since  $r$  is ordered immediately after  $w'$  in  $\tau_r$ ,  $w$  appears after  $r$  in  $\tau_r$ .

Consider any write  $w$  that ends before  $r$  starts in  $\sigma$ . By Lemma 5.3,  $ts(w) \leq ts(r)$ . Since  $r$  appears immediately after the write with timestamp equal to  $ts(r)$ ,  $r$  must appear after  $w$  in  $\tau_r$ .

Finally, the construction of the total orders ensures that for all reads  $r_1$  and  $r_2$  in  $ops(\sigma)$ , all writes in  $ops(\sigma)$  that are relevant to both reads are ordered consistently in  $\tau_{r_1}$  and  $\tau_{r_2}$ . Thus we have:

**THEOREM 5.6** *The algorithm ensures MWReg.*

## 5.2 Complexity

**THEOREM 5.7** *The read protocol has bounded message and time complexity.*

*Proof* It has already been shown that a read terminates. In the read protocol, the reader and servers are involved in one round trip of communication and the reader sends a DONE message to each server at the end of the read. It follows that  $3n$  messages are sent to complete a read. Only one  $\langle \text{value, timestamp} \rangle$  pair is sent in each message. ■

**THEOREM 5.8** *The write protocol has bounded message and time complexity.*

*Proof* It has already been shown that a write terminates. A write requires a read and one round trip of communication with servers. The number of messages generated by a write is thus  $5n + R_c$  where  $R_c$  is the number of read operations concurrent with the write. Only one  $\langle \text{value, timestamp} \rangle$  pair is sent in each message. ■

Each server only needs to store one copy of the most recent value and timestamp, no matter how many writers there are.

## 6. Conclusions

We have presented an efficient replica-based algorithm to implement a shared read-write register that can tolerate Byzantine failures of less than a third of the replica servers. The consistency condition provided by the register is a variant of a form of multi-writer regularity called MWReg in [16, 17]. This condition is weaker than the more commonly provided condition of atomicity, but nevertheless is useful for some important applications, such as mutual exclusion [16, 17].

The algorithm has several desirable properties. First, unlike other known algorithms for Byzantine-tolerant distributed storage, which support  $m$  writers by simulating  $m$  copies of a single-writer protocol, our algorithm directly supports

multiple writers. As a result, the time and message complexity of the algorithm is independent of the number of writers. The algorithm has optimal fault-tolerance in that it is resilient to less than a third of the servers being Byzantine faulty, and to any number of crash failures of clients. Furthermore, no communication is required between clients or between servers. Reducing the number of values that are stored at a reader during a read is an open question.

Numerous additional open questions remain. First, can an inherent separation with respect to scalability be shown between Byzantine-server-tolerant implementations of multi-writer registers that are atomic and those that satisfy a weaker condition such as regularity? Currently there is no known (Byzantine-server-tolerant) multi-writer implementation which satisfies atomicity and does not put an explicit bound on the number of writers. Second, there are several other versions of multi-writer regularity proposed by Shao in [16]. How can these other conditions be made tolerant to Byzantine servers? The algorithms proposed in [16] to implement the consistency conditions have a modular structure based on quorums; is there a modular way to tolerate Byzantine servers, say by using Byzantine quorums [14]?

### Acknowledgments

We thank Z. Asad, M.A.R. Chaudhry, and the referees for helpful comments. This research was supported in part by NSF grant DMI-0500265, NSF grant CNS-0614929, Texas Higher Education Coordinating Board grants ARP 000512-0130-2007, and ARP 000512-0007-2006. The work of Whitney L. Maguffee was supported by the CRA-W DREU program under NSF grant CNS-0540631. The work of Khushboo Kanjani and Whitney L. Maguffee was performed while they were at Texas A&M University.

### References

- [1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, *Byzantine disk paxos: Optimal resilience with Byzantine shared memory*, *Distributed Computing* 18 (2006), pp. 387–408.
- [2] M. Aguilera and R. Swaminathan, *Remote Storage with Byzantine Servers*, in *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 280–289.
- [3] A.S. Aiyer, L. Alvisi, and R.A. Bazzi, *Bounded Wait-Free Implementation of Optimally Resilient Byzantine Storage without (Unproven) Cryptographic Assumptions*, in *Proceedings of the International Symposium on Distributed Computing*, 2007, pp. 7–19.
- [4] H. Attiya and J.L. Welch, *Sequential consistency versus linearizability*, *ACM Trans. Comput. Syst.* 12 (1994), pp. 91–122.
- [5] R. Bazzi and Y. Ding, *Non-skipping timestamps for Byzantine Data Storage Systems*, in *Proceedings of the International Symposium on Distributed Computing*, 2004, pp. 405–419.
- [6] C. Cachin and S. Tessaro, *Optimal Resilience for Erasure-Coded Byzantine Distributed Storage*, in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005, pp. 115–124.
- [7] R. Guerraoui and M. Vukolic, *How Fast Can a Very Robust Read Be?*, in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2006, pp. 248–257.
- [8] M.P. Herlihy and J.M. Wing, *Linearizability: a correctness condition for concurrent objects*, *ACM Trans. Program. Lang. Syst.* 12 (1990), pp. 463–492.
- [9] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.
- [10] F.P. Junqueira and B.C. Reed, *The life and times of a ZooKeeper*, in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2009, p. 46.
- [11] K. Kanjani, *Supporting fault-tolerant communication in networks*, Master’s thesis, Texas A&M University (2008).
- [12] K. Kanjani, H. Lee, and J.L. Welch, *Byzantine fault-tolerant implementation of a multi-writer regular register*, in *Proceedings of the 14th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2009.
- [13] L. Lamport, *On interprocess communication, part I: Basic formalism*, *Distributed Computing* 1 (1986), pp. 77–85.
- [14] D. Malkhi and M. Reiter, *Byzantine quorum systems*, *Distributed Computing* 11 (1998), pp. 203–213.
- [15] J. Martin, L. Alvisi, and M. Dahlin, *Minimal Byzantine Storage*, in *Proceedings of the International Symposium on Distributed Computing*, 2002, pp. 311–325.

- [16] C. Shao, *Multi-writer consistency conditions for shared memory objects*, Master's thesis, Texas A&M University (2007).
- [17] C. Shao, E. Pierce, and J.L. Welch, *Multi-writer Consistency Conditions for Shared Memory Objects*, in *Proceedings of the International Symposium on Distributed Computing*, 2003, pp. 106–120.
- [18] C. Shao, J.L. Welch, E. Pierce, and H. Lee, *Multi-writer consistency conditions for shared memory registers*, Tech. Rep. 2010-1-1, Texas A&M University, 2010.
- [19] P. Yianilos and S. Sobti, *The evolving field of distributed storage*, *IEEE Internet Computing* (2001), pp. 35–39.