Abstract

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 Byzantine failures of some of the replica servers. This new algorithm is both simpler and more fault-tolerant than the one it is based on. 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. By weakening the consistency condition, the algorithm can support multiple writers more efficiently than the known multi-writer algorithms for atomic consistency.

1 Introduction

Distributed storage systems have become popular for handling the enormous amounts of data in network-centric systems using, for example, the Internet [1]. 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 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.

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 terms of additional storage required and the complexity of schemes for keeping replicas consistent.

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) [2] gives an impression of sequential behavior, which is convenient to work with [3] 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 [2]: each read of a regular register returns the value of an overlapping write or of the latest preceding write, but no relative ordering is imposed on the values returned by different reads. Shao et al. [5,6] 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. 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. By weakening the consistency condition, the algorithm can support multiple writers more efficiently 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. [7] have proved that any storage protocol tolerant of \( f \) Byzantine faulty servers requires at least \( 3f + 1 \) servers total to ensure safe\(^1\) 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 [8] 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

\(^1\)Safety is an even weaker consistency condition than regularity, 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 recently. The algorithms most related to ours are compared in Table 1. The columns indicate whether the algorithms are wait-free\(^2\) with respect to client crashes (“WF”), whether they require server-to-server communication (“S2S”) or reliable broadcasting (“RB”), whether they implement atomic registers (“Atomic”), whether they are optimally resilient with respect to Byzantine failures of servers (“3\(f+1\)”), whether they support multiple concurrent writers (“MW”), and whether they have bounded message and time complexity (“BR”).

The multi-writer algorithm in [9], which is based on various Byzantine quorum systems, provides regular semantics for self-verifying data and the weaker condition of safety for non-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 [10,13,14] all provide atomic semantics. Aiyer et al. [14] and Bazzi and Ding [10] 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 [13] 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.

Shao et al. [5,6] 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, at least in the non-Byzantine environment, offers a good

\(^2\)Wait-free means that the algorithm tolerates any number of crash failures.

---

**Table 1: Comparison of algorithms for fault-tolerant distributed storage**

<table>
<thead>
<tr>
<th>Paper</th>
<th>WF</th>
<th>S2S</th>
<th>RB</th>
<th>Atomic</th>
<th>3(f+1)</th>
<th>MW</th>
<th>BR</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>yes</td>
<td>–</td>
<td>no</td>
<td>–</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>[7]</td>
<td>–</td>
<td>–</td>
<td>no</td>
<td>yes</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>[12]</td>
<td>yes</td>
<td>–</td>
<td>no</td>
<td>yes</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>[13]</td>
<td>yes</td>
<td>–</td>
<td>–</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>[14]</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>[15]</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

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.
tradeoff between efficiency of implementation and usefulness in applications.

3 Multi-Writer Regularity

Given an execution \(\alpha\) of the algorithm, the schedule of \(\alpha\) is defined to be the projection of \(\alpha\) onto the read, return, write and ack events.

The algorithm guarantees mw-regularity if for every execution, the schedule of the execution satisfies the following 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.

Nonfaulty 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 \(ops(\sigma)\), but first we need one more definition. A write \(w\) is relevant to a read \(r\) if \(w\) begins in \(\sigma\) before \(r\) ends.

MW-Regularity: 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 \(r_2\).

4 System Model

We consider a distributed system containing \(n\) server processes and any number of client processes. 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 \(n > 3f\), and that 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.
5 Algorithm

In this section we describe the three parts of the algorithm: the reader protocol, the writer protocol, and the server protocol. The interactions between a reader and the servers is depicted in Fig. 1 while the interactions between a writer and the servers is depicted in Fig. 2.

Algorithm 1 Reader’s Protocol

1: when read() occurs:
2: \( ts[s] := \perp, vals[s] := \emptyset \) for each server \( s \)
3: send \((\text{GET},\text{INFO})\) to all servers
4: 
5: when \((\text{INFO}, \text{val})\) message is received from server \( s \):
6: \( vals[s].add(\text{val}) \)
7: if \( (ts[s] = \perp) \) then
8: \( ts[s] := \text{val.ts} \)
9: end if
10: check()
11: 
12: when \((\text{FWD}, \text{val})\) received from server \( s \):
13: \( vals[s].add(\text{val}) \)
14: check()
15: 
16: procedure check():
17: if \( (|s: ts[s] \neq \perp| \geq n - f) \) then
18: if \( (\exists \text{val}: |s:ts[s] \leq \text{val.ts}| \geq 2f + 1 \&\& |s: val \in vals[s]| \geq f + 1) \) then
19: send \((\text{DONE})\) to all servers
20: end if
21: end if
22: end if

Reader’s Protocol When a read is invoked, the reader sends \((\text{GET},\text{INFO})\) messages to each server requesting the value contained at that server. Upon receiving an INFO or FWD message from a server, the reader enters the information sent into the \( ts \) and \( vals \) arrays. The \( ts \) array is an array containing the timestamps associated with the first value each server sends to the reader. The \( vals \) array contains all of the values sent by the servers to the reader during the read. Once the reader receives replies from \( n - f \) servers, the reader begins checking to see if any value meets the termination condition.

Reader Termination Conditions 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 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 sent by 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. Because \( f \) Byzantine servers may send timestamps lower than the timestamp of the completed write.
with the highest timestamp, in order for a value to meet the not old condition, it must have a timestamp greater than or equal to the timestamps sent by the possibly old or Byzantine servers and one updated server.

---

**Algorithm 2** Writer’s protocol

1. **Initially:**
2. \[ \text{ts} := 0 \]
3. 
4. **when** write(v) occurs:
5. \[ \text{recv}_\text{ack}[s] := \text{false for all servers s} \]
6. \[ \langle w, t, \text{wid} \rangle := \text{read() [invoke a read to find a recent timestamp]} \]
7. \[ \text{ts} := \text{max}(\text{ts}, t) + 1 \]
8. send(WRITE_INFO,⟨v, ts⟩) to all servers
9. 
10. **when** (WRITE_ACK, t) is received from server s:
11. \[ \text{if} \ (t = \text{ts}) \ \text{then} \]
12. \[ \text{recv}_\text{ack}[s] := \text{true} \]
13. \[ \text{if} \ (|\{s: \text{recv}_\text{ack}[s] = \text{true}\}| \geq n - f) \ \text{then} \]
14. \[ \text{return} \ \text{ACK} \]
15. \[ \text{end if} \]
16. \[ \text{end if} \]

**Writer’s Protocol**  When a writer attempts a write, it first does a read to find the timestamp that should be associated with the value it wants to write. Once the writer completes the read, it 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.
Algorithm 3 Server’s protocol

1: **Initially:**
2:   readers := ∅
3:   rval := ⟨0,0,0⟩
4: 
5: when (GET_INFO) message received from reader r:
6:   readers.insert(r)
7:   send (INFO,rval) to r
8: 
9: when (DONE) message received from reader r:
10:   readers.remove(r)
11: 
12: when (WRITE_INFO, ⟨v,ts⟩) is received from writer w:
13:   if (rval.(count,id) < ⟨ts,w⟩) then [new val has later timestamp]
14:     rval := ⟨v,ts,w⟩
15:   end if
16:   for each r in readers do
17:     send (FWD,⟨v,ts,w⟩) to r
18:   end for
19:   send (WRITE_ACK,ts) to w

Server’s Protocol  Each server contains a value rval and a set of readers readers. The value rval contains the actual value written, the numerical timestamp associated with the value, and the id of the writer that wrote the value. Once a GET_INFO message is received at a server, the server sends rval to the reader that made the request and adds the reader to readers. Once the server receives a DONE message from that reader, the server will remove that reader from readers. When writers send information to the server, the server updates rval if the new value has a higher timestamp and sends an acknowledgement to the writer. If readers is not empty, the server will forward values sent to it to each member of readers.

Effective Write  Because clients may crash, it is not the case that all writes will have an effect on the system. We will call a write effective if it succeeds in sending a WRITE_INFO message to all servers. Though an effective write may crash after sending the message, the message will still impact the system.
6 Analysis

6.1 Wait-Free

Theorem 1 Every read terminates.
Proof Suppose for contradiction that some read \( r \) never terminates. Because \( r \) never terminates, it eventually hears from all non-faulty servers and all of the Byzantine servers which will respond to its GETINFO request. At this point, the reader’s timestamp array will no longer change. We will call this point in time \( t_{\text{stop}} \). Let \( vts \) be the \((2f + 1)^{\text{st}}\) smallest timestamp in the reader’s timestamp array at time \( t_{\text{stop}} \). Let \( T \) be the largest timestamp from a non-faulty server in \( ts \) after \( t_{\text{stop}} \).

Claim. \( T \) is greater than or equal to \( vts \).

Proof. Suppose \( T \) is smaller than \( vts \). Then all timestamps from non-faulty servers are smaller than the \((2f + 1)^{\text{st}}\) smallest entry in \( ts \). It follows that there are at most \( 2f \) timestamps from non-faulty servers in \( ts \) at time \( t_{\text{stop}} \). This would mean that there are only \( 2f \) non-faulty servers because the process conducting \( r \) has heard from all non-faulty servers. 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 \((v, T)\) be the value-timestamp pair sent in \( w \)'s WRITEINFO message to all servers. Note that \( w \) succeeds in sending its WRITEINFO message, and by our assumption about reliable broadcasts, all non-faulty servers will eventually receive \( w \)'s WRITEINFO message.

Now we show that, in contradiction to our assumption that \( r \) never terminates, the value-timestamp pair \((v, T)\) eventually satisfies \( \text{not old} \) and \( \text{valid} \) and thus \( r \) does terminate.

First we show that, for every non-faulty server \( s \), \((v, T)\) appears in the \( \text{vals}[s] \) sets of the process executing \( r \) during the execution of \( r \). Consider any non-faulty server \( s \) whose entry in the timestamp array at time \( t_{\text{stop}} \) is \( T \). It follows that the process executing \( r \) received an INFO message from \( s \) after \( r \) began containing \((v, T)\). Thus \( p \) stored \( T \) in the timestamp array and \( v \) in \( \text{vals}[s] \) during \( r \).

Now consider any non-faulty server \( s' \) whose entry in the timestamp array at time \( t_{\text{stop}} \) is \( T' \). Because \( T \) is the highest timestamp in the array at time \( t_{\text{stop}} \) sent by a non-faulty server, \( T' \) must be smaller than \( T \). This means that when \( s' \) sent its INFO message to the process executing \( r \), \( T' \) was the highest timestamp associated with a write for which \( s' \) had received a WRITEINFO message. This is the case because servers store values in order of increasing timestamp. Thus, \( s' \) has not received the WRITEINFO message for \( w \). Because \( s' \) is non-faulty and the WRITEINFO message for \( w \) is ensured to reach \( s' \), \( s' \) will eventually receive the WRITEINFO message for \( w \) and send a FWD message to the process running \( r \) containing \((v, T)\). Then \( v \) will be placed into \( \text{vals}[s]\). Thus \( v \) will eventually be in \( \text{vals}[s] \) for every non-faulty server \( s \). So \( v \) will be in \( \text{vals}[s] \) for at least \( 2f + 1 \) servers and \( v \) will satisfy \( \text{valid} \). Based on the earlier claim, \((v, T)\) satisfies \( \text{not old} \). Thus \((v, T)\) meets both conditions and \( r \) can terminate.

Theorem 2 Every write terminates.

Proof A write completes once the process executing it 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 the writer sent a WRITE_INFO message to each server, each of the \( n - f \) non-faulty servers will eventually receive the WRITE_INFO message and send an acknowledgement to the writer. The writer will receive these and the write will terminate.

### 6.2 MW-Regularity

Consider any execution of the algorithm and let \( \sigma \) be the schedule of the execution. From inspection of the code and assumption that the client program invoking the operations is well-behaved, we see that \( \sigma \) satisfies Well-Formedness. Theorems 1 and 2 prove that \( \sigma \) satisfies Nonfaulty Liveness (i.e., every operation invoked by a nonfaulty client terminates). We now show that \( \sigma \) satisfies MW-Regularity.

Let \( ops(\sigma) \) be the set of all completed operations and all effective writes in \( \sigma \). (Recall that a write is effective if it executes Line 8, 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 1** 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** After \( w \) has completed, at least \( n - f \) servers have \( rts \geq ts(w) \). Thus at most \( f \) nonfaulty servers have \( rts < ts(w) \). Since \( rts \) at each nonfaulty server changes over time only by increasing, when \( r \) executes, it 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 2** 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 \( w_2 \) ends before write \( w_1 \) begins in \( \sigma \). Since \( w_2 \) encompasses a complete read, call it \( r \), to decide its timestamp, Lemma 1 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 in the servers to \( \langle 0,0,0 \rangle \) be considered the first write.

**Lemma 3** For every read operation \( r \) in \( ops(\sigma) \), there exists a write operation \( w \) in \( ops(\sigma) \) 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 read must satisfy the valid condition. The valid condition requires that at least \( f + 1 \) servers have sent the same value before a reader may return it. This ensures that at least one non-faulty server \( s \) has sent \( v \). Because \( s \) is non-faulty, \( v \) is from a write. Because \( r \) is still in progress when it receives \( v \), the write of \( s \) must have started before \( r \) ends.
Consider any read $r$ in $\text{ops}(\sigma)$. We construct a total order $\tau_r$ on the set containing $r$ and all writes in $\text{ops}(\sigma)$ as follows.

- Order all the writes in $\text{ops}(\sigma)$ 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 the write from which it reads (i.e., the write whose timestamp is the same as that of the value returned by $r$) 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).

We now show that $\tau_r$ is $\sigma$-consistent. For any two writes that are relevant to $r$, and for any two writes that are not relevant to $r$, $\sigma$-consistency of $\tau_r$ follows from Lemma 2. For write $w_1$ that is relevant to $r$ and write $w_2$ that 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. Then $w$ is not relevant to $r$, but $r$ reads from some write $w'$ that is relevant to $r$, by Lemma 3. Thus $w$ 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. Thus $w$ is relevant to $r$. By Lemma 1, $ts(w) \leq ts(r)$. Accordingly, $r$ cannot appear before $w$ in $\tau_r$ because $r$ appears directly after the write with a timestamp equal to $ts(r)$. So if $r$ appears before $w$ in $\tau_r$, then $ts(r)$ must be smaller than $ts(w)$. This contradicts Lemma 1. Therefore $r$ 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 $\text{ops}(\sigma)$, all writes in $\text{ops}(\sigma)$ that are relevant to both reads are ordered consistently in $\tau_{r_1}$ and $\tau_{r_2}$.

Thus we have:

**Theorem 3** The algorithm ensures MW-Regularity.

### 6.3 Complexity

**Theorem 4** 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 during a read. Only one $\langle\text{value},\text{timestamp}\rangle$ pair is sent in each message.

**Theorem 5** 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 $5n + |R|$ where $|R|$ is the number of read operations concurrent with the write. Only one ⟨value, timestamp⟩ pair is sent in each message.

<table>
<thead>
<tr>
<th>Table 2: Complexity of the read/write operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Rounds</td>
</tr>
<tr>
<td>Messages</td>
</tr>
<tr>
<td>Message size</td>
</tr>
</tbody>
</table>

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

7 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 [5,6]. This condition is weaker than the more commonly provided condition of atomicity, but nevertheless is useful for some important applications, such as mutual exclusion [5,6].

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 bounded and 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 (direct) 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, it would be interesting to investigate a hybrid scheme for distributed storage combining replication and erasure coding [13]. Third, there are several other versions of multi-writer regularity proposed by Shao in [6]. How can these other conditions be made tolerant to Byzantine servers? The algorithms proposed in [6] 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 [9]?

Acknowledgments

The work of Whitney L. Maguffee was supported by the CRA-W DREU program under NSF grant CNS-0540631. 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 by Khushboo Kanjani was performed while she was at Texas A&M University.

References


